The Lean Language Reference

This is the *Lean Language Reference*. It is intended to be a comprehensive, precise description of Lean: a reference work in which Lean users can look up detailed information, rather than a tutorial intended for new users. For other documentation, please refer to the Lean documentation overview. This manual covers Lean version `4.22.0-rc3`.

Lean is an **interactive theorem prover** based on dependent type theory, designed for use both in cutting-edge mathematics and in software verification. Lean's core type theory is expressive enough to capture very complicated mathematical objects, but simple enough to admit independent implementations, reducing the risk of bugs that affect soundness. The core type theory is implemented in a minimal kernel that does nothing other than check proof terms. This core theory and kernel are supported by advanced automation, realized in an expressive tactic language. Each tactic produces a term in the core type theory that is checked by the kernel, so bugs in tactics do not threaten the soundness of Lean as a whole. Along with many other parts of Lean, the tactic language is user-extensible, so it can be built up to meet the needs of a given formalization project. Tactics are written in Lean itself, and can be used immediately upon definition; rebuilding the prover or loading external modules is not required.

Lean is also a pure **functional programming language**, with features such as a run-time system based on reference counting that can efficiently work with packed array structures, multi-threading, and monadic `IO`. As befits a programming language, Lean is primarily implemented in itself, including the language server, build tool, elaborator, and tactic system. This very book is written in Verso, a documentation authoring tool written in Lean.

Familiarity with Lean's programming features is valuable even for users whose primary interest is in writing proofs, because Lean programs are used to implement new tactics and proof automation. Thus, this reference manual does not draw a barrier between the two aspects, but rather describes them together so they can shed light on one another.

Table of Contents

1. Introduction

2. Elaboration and Compilation

3. Interacting with Lean

4. The Type System

5. Source Files and Modules

6. Namespaces and Sections

7. Definitions

8. Axioms

9. Attributes

10. Terms

11. Type Classes

12. Coercions

13. Tactic Proofs

14. Functors, Monads and do-Notation

15. IO

16. The Simplifier

17. The `grind` tactic

18. Basic Propositions

19. Basic Types

20. Notations and Macros

21. Run-Time Code

22. Build Tools and Distribution

Error Explanations

The Module System

Release Notes

Index

1. Introduction

The *Lean Language Reference* is intended as a comprehensive, precise description of Lean. It is a reference work in which Lean users can look up detailed information, rather than a tutorial for new users. At the moment, this reference manual is a public preview. For tutorials and learning materials, please visit the Lean documentation page.

This document describes version `4.22.0-rc3` of Lean.

1.1. History

Leonardo de Moura launched the Lean project when he was at Microsoft Research in 2013, and Lean 0.1 was officially released on June 16, 2014. The goal of the Lean project is to combine the high level of trust provided by a small, independently-implementable logical kernel with the convenience and automation of tools like SMT solvers, while scaling to large problems. This vision still guides the development of Lean, as we invest in improved automation, improved performance, and user-friendliness; the trusted core proof checker is still minimal and independent implementations exist.

The initial versions of Lean were primarily configured as C++ libraries in which client code could carry out trustworthy proofs that were independently checkable. In these early years, the design of Lean rapidly evolved towards traditional interactive provers, first with tactics written in Lua, and later with a dedicated front-end syntax. January 20, 2017 saw the first release of the Lean 3.0 series. Lean 3 achieved widespread adoption by mathematicians, and pioneered self-extensibility: tactics, notations, and top-level commands could all be defined in Lean itself. The mathematics community built Mathlib, which at the end of Lean 3 had over one million lines of formalized mathematics, with all proofs mechanically checked. The system itself, however, was still implemented in C++, which imposed limits on Lean's flexibility and made it more difficult to develop due to the diverse skills required.

Development of Lean 4 began in 2018, culminating in the 4.0 release on September 8, 2023. Lean 4 represents an important milestone: as of version 4, Lean is self-hosted - approximately 90% of the code that implements Lean is itself written in Lean. Lean 4's rich extension API provides users with the ability to adapt it to their needs, rather than relying on the core developers to add necessary features. Additionally, self-hosting makes the development process much faster, so features and performance can be delivered more quickly; Lean 4 is faster and scales to larger problems than Lean 3. Mathlib was successfully ported to Lean 4 in 2023 through a community effort supported by the Lean developers, and it has now grown to over 1.5 million lines. Even though Mathlib has grown by 50%, Lean 4 checks it faster than Lean 3 could check its smaller library. The development process for Lean 4 was approximately as long as that of all prior versions combined, and we are now delighted with its design—no further rewrites are planned.

Leonardo de Moura and his co-founder, Sebastian Ullrich, launched the Lean Focused Research Organization (FRO) nonprofit in July of 2023 within Convergent Research, with philanthropic support from the Simons Foundation International, the Alfred P. Sloan Foundation, and Richard Merkin. The FRO currently has more than ten employees working to support the growth and scalability of Lean and the broader Lean community.

1.2. Typographical Conventions

This document makes use of a number of typographical and layout conventions to indicate various aspects of the information being presented.

1.2.1. Lean Code

This document contains many Lean code examples. They are formatted as follows:

```
def hello : IO Unit := IO.println "Hello, world!"
```

Compiler output (which may be errors, warnings, or just information) is shown both in the code and separately:

```
#eval s!"The answer is {2 + 2}"
```

```
theorem bogus : False := by sorry
```

```
example := Nat.succ "two"
```

Informative output, such as the result of `#eval`, is shown like this:

```
"The answer is 4"
```

Warnings are shown like this:

```
declaration uses 'sorry'
```

Error messages are shown like this:

```
Application type mismatch: In the application
  Nat.succ "two"
the argument
  "two"
has type
  String : Type
but is expected to have type
  Nat : Type
```

The presence of tactic proof states is indicated by the presence of small lozenges that can be clicked to show the proof state, such as after `rfl` below:

```
example : 2 + 2 = 4 := by rfl
```

Proof states may also be shown on their own. When attempting to prove that `2 + 2 = 4`, the initial proof state is:

⊢ 2 + 2 = 4

After using `rfl`, the resulting state is:

All goals completed! 🐙

Identifiers in code examples are hyperlinked to their documentation.

Examples of code with syntax errors are shown with an indicator of where the parser error occurred, along with the error message:

```
def f : Option Nat → Type
  | some 0 => Unit
  | => Option (f t)
  | none => Empty
```

```
<example>:3:3-3:6: unexpected token '=>'; expected term
```

1.2.2. Examples

Illustrative examples are in callout boxes, as below:

▼ *Example: Even Numbers*

This is an example of an example.

One way to define even numbers is via an inductive predicate:

```
inductive Even : Nat → Prop where
  | zero : Even 0
  | plusTwo : Even n → Even (n + 2)
```

1.2.3. Technical Terminology

Technical terminology refers to terms used in a very specific sense when writing technical material, such as this reference. Uses of technical terminology are frequently hyperlinked to their definition sites, using links like this one.

1.2.4. Constant, Syntax, and Tactic References

Definitions, inductive types, syntax formers, and tactics have specific descriptions. These descriptions are marked as follows:

```
/--
Evenness: a number is even if it can be evenly divided by two.
-/
inductive Even : Nat → Prop where
  | /-- 0 is considered even here -/
    zero : Even 0
  | /-- If `n` is even, then so is `n + 2`. -/
    plusTwo : Even n → Even (n + 2)
```

inductive predicate

```
Even : Nat → Prop
```

Evenness: a number is even if it can be evenly divided by two.

Constructors

| `zero : Even 0`

0 is considered even here

| `plusTwo {n : Nat} : Even n → Even (n + 2)`

If *n* is even, then so is *n* + 2.

Open-Source Licenses

Editable Combobox With Both List and Inline Autocomplete Example, from the W3C's ARIA Authoring Practices Guide (APG)

https://www.w3.org/WAI/ARIA/apg/patterns/combobox/examples/combobox-autocomplete-both/

The search box component includes code derived from the example code in the linked article from the W3C's ARIA Authoring Practices Guide (APG).

`W3C-20150513`

Software and Document License - 2023 Version

Permission to copy, modify, and distribute this work, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the work or portions thereof, including modifications:

* The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.

* Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C software and document short notice should be included.

* Notice of any changes or modifications, through a copyright statement on the new code or document such as "This software or document includes material copied from or derived from "Editable Combobox With Both List and Inline Autocomplete Example" at https://www.w3.org/WAI/ARIA/apg/patterns/combobox/examples/combobox-autocomplete-both/. Copyright © 2024 World Wide Web Consortium. https://www.w3.org/copyright/software-license-2023/"

elasticlunr.js

http://elasticlunr.com/

Elasticlunr.js is used for full-text search

`MIT`

The MIT License

Copyright (C) 2017 by Wei Song

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

fuzzysort v3.1.0

https://github.com/farzher/fuzzysort

The fuzzysort library is used in the search box to quickly filter results.

MIT

The MIT License

Copyright (c) 2018 Stephen Kamenar

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

KaTeX

https://katex.org/

KaTeX is used to render mathematical notation.

MIT

The MIT License

Copyright (c) 2013-2020 Khan Academy and other contributors

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Popper.js

https://popper.js.org/docs/v2/

Popper.js is used (as a dependency of Tippy.js) to show information (primarily in Lean code) when hovering the mouse over an item of interest.

`MIT`

The MIT License

Copyright (c) 2019 Federico Zivolo

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Tippy.js

https://atomiks.github.io/tippyjs/

Tippy.js is used together with Popper.js to show information (primarily in Lean code) when hovering the mouse over an item of interest.

`MIT`

The MIT License

Copyright (c) 2017-present atomiks

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2. Elaboration and Compilation

Roughly speaking, Lean's processing of a source file can be divided into the following stages:

Parsing
: The parser transforms sequences of characters into syntax trees of type `Syntax`. Lean's parser is extensible, so the `Syntax` type is very general.

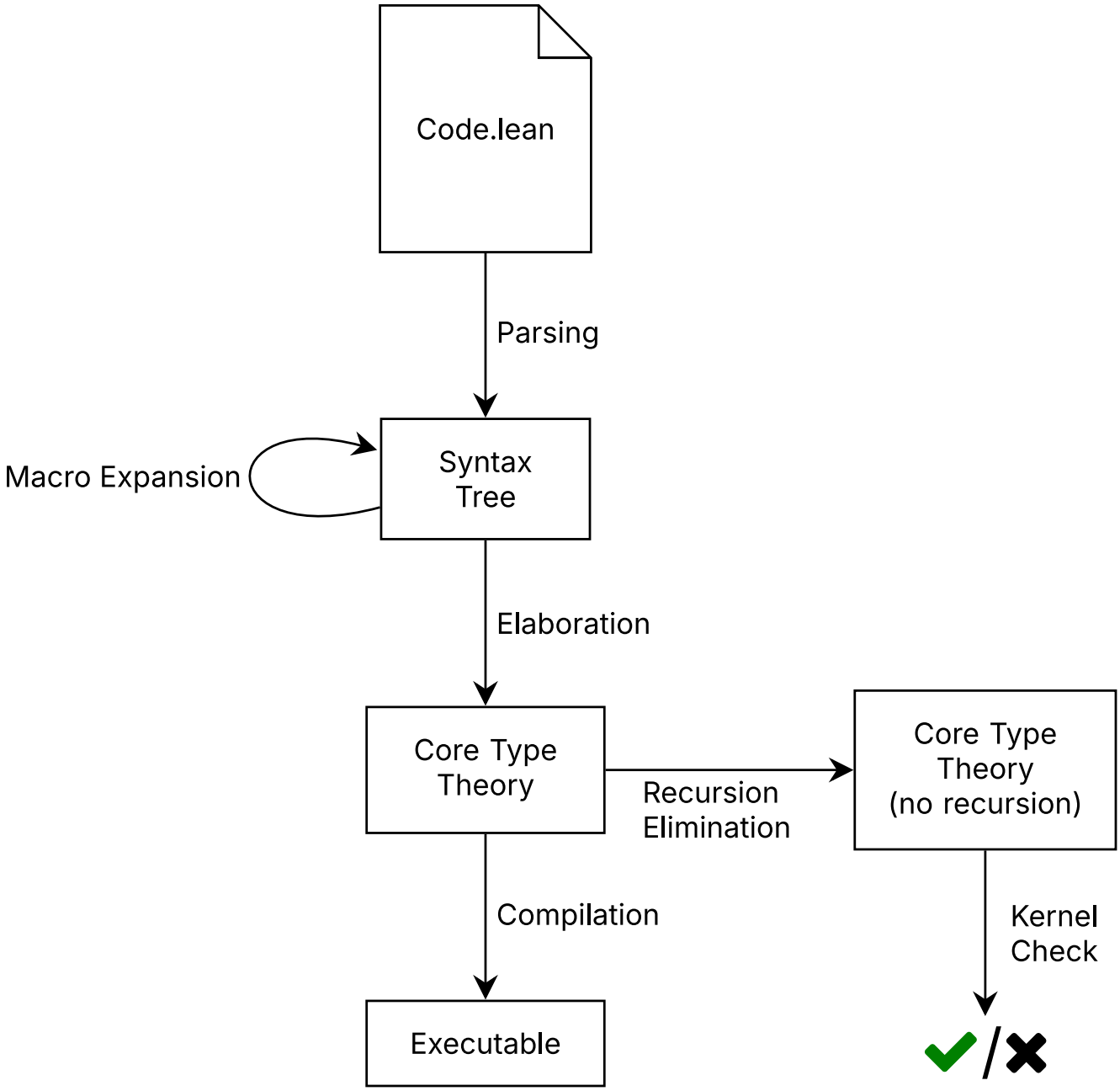
Macro Expansion
: Macros are transformations that replace syntactic sugar with more basic syntax. Both the input and output of macro expansion have type `Syntax`.

Elaboration
: Elaboration is the process of transforming Lean's user-facing syntax into its core type theory. This core theory is much simpler, enabling the trusted kernel to be very small. Elaboration additionally produces metadata, such as proof states or the types of expressions, used for Lean's interactive features, storing them in a side table.

Kernel Checking
: Lean's trusted kernel checks the output of the elaborator to ensure that it follows the rules of the type theory.

Compilation

The compiler transforms elaborated Lean code into executables that can be run.



The Lean Pipeline

In reality, the stages described above do not strictly occur one after the other. Lean parses a single command (top-level declaration), elaborates it, and performs any necessary kernel checks. Macro expansion is part of elaboration; before translating a piece of syntax, the elaborator first expands any macros present at the outermost layer. Macro syntax may remain at deeper layers, but it will be expanded when the elaborator reaches those layers. There are multiple kinds of elaboration: command elaboration implements the effects of each top-level command (e.g. declaring inductive types, saving definitions, evaluating expressions), while term elaboration is responsible for constructing the terms that occur in many commands (e.g. types in signatures, the right-hand sides of definitions, or expressions to be evaluated). Tactic execution is a specialization of term elaboration.

When a command is elaborated, the state of Lean changes. New definitions or types may have been saved for future use, the syntax may be extended, or the set of names that can be referred to without explicit qualification may have changed. The next command is parsed and elaborated in this updated state, and itself updates the state for subsequent commands.

2.1. Parsing

Lean's parser is a recursive-descent parser that uses dynamic tables based on Pratt parsing (Pratt, 1973) (i) Vaughan Pratt, 1973. “Top down operator precedence”. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* to resolve operator precedence and associativity. When grammars are unambiguous, the parser does not need to backtrack; in the case of ambiguous grammars, a memoization table similar to that used in Packrat parsing avoids exponential blowup. Parsers are highly extensible: users may define new syntax in any command, and that syntax becomes available in the next command. The open namespaces in the current section scope also influence which parsing rules are used, because parser extensions may be set to be active only when a given namespace is open.

When ambiguity is encountered, the longest matching parse is selected. If there is no unique longest match, then both matching parses are saved in the syntax tree in a choice node to be resolved later by the elaborator. When the parser fails, it returns a `Syntax.missing` node, allowing for error recovery.

When successful, the parser saves sufficient information to reconstruct the original source file. Unsuccessful parses may miss some information for the regions of the file that cannot be parsed. The `SourceInfo` record type records information about the origin of a piece of syntax, including its source location and the surrounding whitespace. Based on the `SourceInfo` field, there are three relationships that `Syntax` can have to a source file:

- `SourceInfo.original` indicates that the syntax value was produced directly by the parser.
- `SourceInfo.synthetic` indicates that the syntax value was produced programmatically, e.g. by the macro expander. Synthetic syntax may nonetheless be marked *canonical*, in which case the Lean user interface treats it as if the user had written it. Synthetic syntax is annotated with positions in the original file, but does not include leading or trailing whitespace.
- `SourceInfo.none` indicates no relationship to a file.

The parser maintains a token table that tracks the reserved words that are currently part of the language. Defining new syntax or opening namespaces can cause a formerly-valid identifier to become a keyword.

Each production in Lean's grammar is named. The name of a production is called its *kind*. These syntax kinds are important, because they are the key used to look up the interpretation of the syntax in the elaborator's tables.

Syntax extensions are described in more detail in a dedicated chapter.

2.2. Macro Expansion and Elaboration

Having parsed a command, the next step is to elaborate it. The precise meaning of *elaboration* depends on what is being elaborated: elaborating a command effects a change in the state of Lean, while elaborating a term results in a term in Lean's core type theory. Elaboration of both commands and terms may be recursive, both because of command combinators such as `in` and because terms may contain other terms.

Command and term elaboration have different capabilities. Command elaboration may have side effects on an environment, and it has access to run arbitrary computations in `IO`. Lean environments contain the usual mapping from names to definitions along with additional data defined in environment extensions, which are additional tables associated with an environment; environment extensions are used to track most other information about Lean code, including `simp` lemmas, custom pretty printers, and internals such as the compiler's intermediate representations. Command elaboration also maintains a message log with the contents of the compiler's informational output, warnings, and errors, a set of info trees that associate metadata with the original syntax (used for interactive features such as displaying proof states, identifier completion, and showing documentation), accumulated debugging traces, the open section scopes, and some internal state related to macro expansion. Term elaboration may modify all of these fields except the open scopes. Additionally, it has access to all the machinery needed to create fully-explicit terms in the core language from Lean's terse, friendly syntax, including unification, type class instance synthesis, and type checking.

The first step in both term and command elaboration is macro expansion. There is a table that maps syntax kinds to macro implementations; macro implementations are monadic functions that transform the macro syntax into new syntax. Macros are saved in the same table and execute in the same monad for terms, commands, tactics, and any other macro-extensible part of Lean. If the syntax returned by the macro is itself a macro, then that syntax is again expanded—this process is repeated until either a syntax whose kind is not a macro is produced, or until a maximum number of iterations is reached, at which point Lean produces an error. Typical macros process some outer layer of their syntax, leaving some subterms untouched. This means that even when macro expansion has been completed, there still may be macro invocations remaining in the syntax below the top level. New macros may be added to the macro table. Defining new macros is described in detail in the section on macros.

After macro expansion, both the term and command elaborators consult tables that map syntax kinds to elaboration procedures. Term elaborators map syntax and an optional expected type to a core language expression using the very powerful monad mentioned above. Command elaborators accept syntax and return no value, but may have monadic side effects on the global command state. While both term and command elaborators have access to `IO`, it's unusual that they perform side effects; exceptions include interactions with external tools or solvers.

The elaborator tables may be extended to enable the use of new syntax for both terms and commands by extending the tables. See the section on elaborators for a description of how to add additional elaborators to Lean. When commands or terms contain further commands or terms, they recursively invoke the appropriate elaborator on the nested syntax. This elaborator will then expand

macros before invoking elaborators from the table. While macro expansion occurs prior to elaboration for a given “layer” of the syntax, macro expansion and elaboration are interleaved in general.

2.2.1. Info Trees

When interacting with Lean code, much more information is needed than when simply importing it as a dependency. For example, Lean's interactive environment can be used to view the types of selected expressions, to step through all the intermediate states of a proof, to view documentation, and highlight all occurrences of a bound variable. The information necessary to use Lean interactively is stored in a side table called the *info trees* during elaboration.

Info trees relate metadata to the user's original syntax. Their tree structure corresponds closely to the tree structure of the syntax, although a given node in the syntax tree may have many corresponding info tree nodes that document different aspects of it. This metadata includes the elaborator's output in Lean's core language, the proof state active at a given point, suggestions for interactive identifier completion, and much more. The metadata can also be arbitrarily extended; the constructor `Info.ofCustomInfo` accepts a `Dynamic` type. This can be used to add information to be used by custom code actions or other user interface extensions.

2.3. The Kernel

Lean's trusted *kernel* is a small, robust implementation of a type checker for the core type theory. It does not include a syntactic termination checker, nor does it perform unification; termination is guaranteed by elaborating all recursive functions into uses of primitive recursors, and unification is expected to have already been carried out by the elaborator. Before new inductive types or definitions are added to the environment by the command or term elaborators, they must be checked by the kernel to guard against potential bugs in elaboration.

Lean's kernel is written in C++. There are independent re-implementations in Rust and Lean, and the Lean project is interested in having as many implementations as possible so that they can be cross-checked against each other.

The language implemented by the kernel is a version of the Calculus of Constructions, a dependent type theory with the following features:

- Full dependent types
- Inductively-defined types that may be mutually inductive or include recursion nested under other inductive types
- An impredicative, definitionally proof-irrelevant, extensional universe of propositions
- A predicative, non-cumulative hierarchy of universes of data
- Quotient types with a definitional computation rule

- Propositional function extensionality (i) Function extensionality is a theorem that can be proved using quotient types, but it is such an important consequence that it's worth listing separately.
- Definitional η-equality for functions and products
- Universe-polymorphic definitions
- Consistency: there is no axiom-free closed term of type `False`

This theory is rich enough to express leading-edge research mathematics, and yet simple enough to admit a small, efficient implementation. The presence of explicit proof terms makes it feasible to implement independent proof checkers, increasing our confidence. It is described in detail by Carneiro (2019) (i) Mario Carneiro, 2019. *The Type Theory of Lean*. Masters thesis, Carnegie Mellon University and Ullrich (2023) (i) Sebastian Ullrich, 2023. *An Extensible Theorem Proving Frontend*. Dr. Ing. dissertation, Karlsruhe Institute of Technology.

Lean's type theory does not feature subject reduction, the definitional equality is not necessarily transitive, and it is possible to make the type checker fail to terminate. None of these metatheoretic properties cause problems in practice—failures of transitivity are exceedingly rare, and as far as we know, non-termination has not occurred except when crafting code specifically to exercise it. Most importantly, logical soundness is not affected. In practice, apparent non-termination is indistinguishable from sufficiently slow programs; the latter are the causes observed in the wild. These metatheoretic properties are a result of having impredicativity, quotient types that compute, definitional proof irrelevance, and propositional extensionality; these features are immensely valuable both to support ordinary mathematical practice and to enable automation.

2.4. Elaboration Results

Lean's core type theory does not include pattern matching or recursive definitions. Instead, it provides low-level recursors that can be used to implement both case distinction and primitive recursion. Thus, the elaborator must translate definitions that use pattern matching and recursion into definitions that use recursors. (i) More details on the elaboration of recursive definitions is available in the dedicated section on the topic. This translation is additionally a proof that the function terminates for all potential arguments, because all functions that can be translated to recursors also terminate.

The translation to recursors happens in two phases: during term elaboration, uses of pattern matching are replaced by appeals to *auxiliary matching functions* that implement the particular case distinction that occurs in the code. These auxiliary functions are themselves defined using recursors, though they do not make use of the recursors' ability to actually implement recursive behavior. (i) They use the `casesOn` construction that is described in the section on recursors and elaboration. The term elaborator thus returns core-language terms in which pattern matching has been replaced with the use of special functions that implement case distinction, but these terms may still contain recursive occurrences of the function being defined. A definition that still includes recursion, but has otherwise been elaborated to the core language, is called a pre-definition. To see auxiliary pattern matching functions in Lean's output, set the option `pp.match` to `false`.

option

`pp.match`

Default value: `true`

(pretty printer) disable/enable 'match' notation

The pre-definition is then sent to the compiler and to the kernel. The compiler receives the pre-definition as-is, with recursion intact. The version sent to the kernel, on the other hand, undergoes a second transformation that replaces explicit recursion with uses of recursors, well-founded recursion, or . This split is for three reasons:

- The compiler can compile `partial` functions that the kernel treats as opaque constants for the purposes of reasoning.
- The compiler can also compile `unsafe` functions that bypass the kernel entirely.
- Translation to recursors does not necessarily preserve the cost model expected by programmers, in particular laziness vs strictness, but compiled code must have predictable performance. The other strategies used to justify recursive definitions result in internal terms that are even further from the program as it was written.

The compiler stores an intermediate representation in an environment extension.

For straightforwardly structurally recursive functions, the translation will use the type's recursor. These functions tend to be relatively efficient when run in the kernel, their defining equations hold definitionally, and they are easy to understand. Functions that use other patterns of recursion that cannot be captured by the type's recursor are translated using well-founded recursion, which is structural recursion on a proof that some *measure* decreases at each recursive call, or using partial fixpoints, which logically capture at least part of a function's specification by appealing to domain-theoretic constructions. Lean can automatically derive many of these termination proofs, but some require manual proofs. Well-founded recursion is more flexible, but the resulting functions are often slower to execute in the kernel due to the proof terms that show that a measure decreases, and their defining equations may hold only propositionally. To provide a uniform interface to functions defined via structural and well-founded recursion and to check its own correctness, the elaborator proves equational lemmas that relate the function to its original definition. In the function's namespace, `eq_unfold` relates the function directly to its definition, `eq_def` relates it to the definition after instantiating implicit parameters, and N lemmas `eq_N` relate each case of its pattern-matching to the corresponding right-hand side, including sufficient assumptions to indicate that earlier branches were not taken.

▼ *Example: Equational Lemmas*

Given the definition of `thirdOfFive`:

```
def thirdOfFive : List α → Option α
  | [_, _, x, _, _] => some x
  | _ => none
```

equational lemmas are generated that relate `thirdOfFive` to its definition.

`thirdOfFive.eq_unfold` states that it can be unfolded to its original definition when no arguments are provided:

```
thirdOfFive.eq_unfold.{u_1} :
  @thirdOfFive.{u_1} = fun {α : Type u_1} x =>
    match x with
    | [head, head_1, x, head_2, head_3] => some x
    | x => none
```

`thirdOfFive.eq_def` states that it matches its definition when applied to arguments:

```
thirdOfFive.eq_def.{u_1} {α : Type u_1} :
  ∀ (x : List α),
    thirdOfFive x =
      match x with
      | [head, head_1, x, head_2, head_3] => some x
      | x => none
```

`thirdOfFive.eq_1` shows that its first defining equation holds:

```
thirdOfFive.eq_1.{u} {α : Type u}
    (head head_1 x head_2 head_3 : α) :
  thirdOfFive [head, head_1, x, head_2, head_3] = some x
```

`thirdOfFive.eq_2` shows that its second defining equation holds:

```
thirdOfFive.eq_2.{u_1} {α : Type u_1} :
  ∀ (x : List α),
    (∀ (head head_1 x_1 head_2 head_3 : α),
      x = [head, head_1, x_1, head_2, head_3] → False) →
    thirdOfFive x = none
```

The final lemma `thirdOfFive.eq_2` includes a premise that the first branch could not have matched (that is, that the list does not have exactly five elements).

▼ *Example: Recursive Equational Lemmas*

Given the definition of `everyOther`:

```
def everyOther : List α → List α
  | [] => []
  | [x] => [x]
  | x :: _ :: xs => x :: everyOther xs
```

equational lemmas are generated that relate `everyOther`'s recursor-based implementation to its original recursive definition.

`everyOther.eq_unfold` states that `everyOther` with no arguments is equal to its unfolding:

```
everyOther.eq_unfold.{u} :
  @everyOther.{u} = fun {α} x =>
    match x with
    | [] => []
    | [x] => [x]
    | x :: _ :: xs => x :: everyOther xs
```

`everyOther.eq_def` states that a `everyOther` is equal to its definition when applied to arguments:

```
everyOther.eq_def.{u} {α : Type u} :
  ∀ (x : List α),
    everyOther x =
      match x with
      | [] => []
      | [x] => [x]
      | x :: _ :: xs => x :: everyOther xs
```

`everyOther.eq_1` demonstrates its first pattern:

```
everyOther.eq_1.{u} {α : Type u} : everyOther [] = ([] : List α)
```

`everyOther.eq_2` demonstrates its second pattern:

```
everyOther.eq_2.{u} {α : Type u} (x : α) : everyOther [x] = [x]
```

`everyOther.eq_3` demonstrates its final pattern:

```
everyOther.eq_3.{u} {α : Type u} (x y : α) (xs : List α) :
  everyOther (x :: y :: xs) = x :: everyOther xs
```

Because the patterns do not overlap, no assumptions about prior patterns not having matched are necessary for the equational lemmas.

After elaborating a module, having checked each addition to the environment with the kernel, the changes that the module made to the global environment (including extensions) are serialized to a `.olean` file. In these files, Lean terms and values are represented just as they are in memory; thus the file can be directly memory-mapped. All code paths that lead to Lean adding to the environment involve the new type or definition first being checked by the kernel. However, Lean is a very open, flexible system. To guard against the possibility of poorly-written metaprograms jumping through hoops to add unchecked values to the environment, a separate tool `lean4checker` can be used to validate that the entire environment in a `.olean` file satisfies the kernel.

In addition to the `.olean` file, the elaborator produces a `.ilean` file, which is an index used by the language server. This file contains information needed to work interactively with the module without fully loading it, such as the source positions of definitions. The contents of `.ilean` files are an

implementation detail and may change at any release.

Finally, the compiler is invoked to translate the intermediate representation of functions stored in its environment extension into C code. A C file is produced for each Lean module; these are then compiled to native code using a bundled C compiler. If the `precompileModules` option is set in the build configuration, then this native code can be dynamically loaded and invoked by Lean; otherwise, an interpreter is used. For most workloads, the overhead of compilation is larger than the time saved by avoiding the interpreter, but some workloads can be sped up dramatically by pre-compiling tactics, language extensions, or other extensions to Lean.

2.5. Initialization

Before starting up, the elaborator must be correctly initialized. Lean itself contains initialization code that must be run in order to correctly construct the compiler's initial state; this code is run before loading any modules and before the elaborator is invoked. Furthermore, each dependency may itself contribute initialization code, *e.g.* to set up environment extensions. Internally, each environment extension is assigned a unique index into an array, and this array's size is equal to the number of registered environment extensions, so the number of extensions must be known in order to correctly allocate an environment.

After running Lean's own builtin initializers, the module's header is parsed and the dependencies' `.olean` files are loaded into memory. A “pre-environment” is constructed that contains the union of the dependencies' environments. Next, all initialization code specified by the dependencies is executed in the interpreter. At this point, the number of environment extensions is known, so the pre-environment can be reallocated into an environment structure with a correctly-sized extensions array.

syntax

Initialization Blocks

An `initialize` block adds code to the module's initializers. The contents of an `initialize` block are treated as the contents of a `do` block in the `IO` monad.

Sometimes, initialization only needs to extend internal data structures by side effects. In that case the contents are expected to have type `IO Unit`:

```
command ::= ...
    | initialize
        doSeqItem*
```

Initialization may also be used to construct values that contain references to internal state, such as attributes that are backed by an environment extension. In this form of `initialize`, initialization should return the specified type in the `IO` monad.

```
command ::= ...
    | initialize ident : term ←
```

```
        doSeqItem*
```

syntax

Compiler-Internal Initializers

Lean's internals also define code that must run during initialization. However, because Lean is a bootstrapping compiler, special care must be taken with initializers defined as part of Lean itself, and Lean's own initializers must run prior to importing or loading *any* modules. These initializers are specified using `builtin_initialize`, which should not be used outside the compiler's implementation.

```
command ::= ...
    | builtin_initialize
        doSeqItem*

command ::= ...
    | builtin_initialize ident : term ←
        doSeqItem*
```

3. Interacting with Lean

Lean is designed for interactive use, rather than as a batch-mode system in which whole files are fed in and then translated to either object code or error messages. Many programming languages designed for interactive use provide a REPL, (i) Short for "**R**ead-**E**val-**P**rint **L**oop", because code is parsed ("read"), evaluated, and the result displayed, with this process repeated as many times as desired. at which code can be input and tested, along with commands for loading source files, type checking terms, or querying the environment. Lean's interactive features are based on a different paradigm. Rather than a separate command prompt outside of the program, Lean provides commands for accomplishing the same tasks in the context of a source file. By convention, commands that are intended for interactive use rather than as part of a durable code artifact are prefixed with `#`.

Information from Lean commands is available in the *message log*, which accumulates output from the elaborator. Each entry in the message log is associated with a specific source range and has a *severity*. There are three severities: `information` is used for messages that do not indicate a problem, `warning` indicates a potential problem, and `error` indicates a definite problem. For interactive commands, results are typically returned as informational messages that are associated with the command's leading keyword.

3.1. Evaluating Terms

The `#eval` command is used to run code as a program. In particular, it is capable of executing `IO` actions, it uses a call-by-value evaluation strategy, `partial` functions are executed, and both types and proofs are erased. Use `#reduce` to instead reduce terms using the reduction rules that are part

of definitional equality.

syntax

Evaluating Terms

```
command ::= ...
    | #eval term

command ::= ...
    | #eval! term
```

#eval e evaluates the expression e by compiling and evaluating it.

- The command attempts to use `ToExpr`, `Repr`, or `ToString` instances to print the result.
- If e is a monadic value of type `m ty`, then the command tries to adapt the monad *m* to one of the monads that **#eval** supports, which include `IO`, `CoreM`, `MetaM`, `TermElabM`, and `CommandElabM`. Users can define `MonadEval` instances to extend the list of supported monads.

The **#eval** command gracefully degrades in capability depending on what is imported. Importing the `Lean.Elab.Command` module provides full capabilities.

Due to unsoundness, **#eval** refuses to evaluate expressions that depend on **sorry**, even indirectly, since the presence of **sorry** can lead to runtime instability and crashes. This check can be overridden with the **#eval!** e command.

Options:

- If `eval.pp` is true (default: true) then tries to use `ToExpr` instances to make use of the usual pretty printer. Otherwise, only tries using `Repr` and `ToString` instances.
- If `eval.type` is true (default: false) then pretty prints the type of the evaluated value.
- If `eval.derive.repr` is true (default: true) then attempts to auto-derive a `Repr` instance when there is no other way to print the result.

See also: **#reduce** e for evaluation by term reduction.

#eval always elaborates and compiles the provided term. It then checks whether the term transitively depends on any uses of **sorry**, in which case evaluation is terminated unless the command was invoked as `#eval!`. This is because compiled code may rely on compile-time invariants (such as array lookups being in-bounds) that are ensured by proofs of suitable statements, and running code that contains incomplete proofs (or uses of **sorry** that "prove" incorrect statements) can cause Lean itself to crash.

The way the code is run depends on its type:

- If the type is in the `IO` monad, then it is executed in a context where standard output and standard error are captured and redirected to the Lean message log. If the returned value's type is not `Unit`, then it is displayed as if it were the result of a non-monadic expression.
- If the type is in one of the internal Lean metaprogramming monads (`CommandElabM`, `TermElabM`, `MetaM`, or `CoreM`), then it is run in the current context. For example, the environment will contain the definitions that are in scope where `#eval` is invoked. As with `IO`, the resulting value is displayed as if it were the result of a non-monadic expression. When Lean is running under Lake, its working directory (and thus the working directory for `IO` actions) is the current `workspace`.
- If the type is in some other monad *m*, and there is a `MonadLiftT` *m* `CommandElabM` or `MonadEvalT` *m* `CommandElabM` instance, then `MonadLiftT.monadLift` or `MonadEvalT.monadEval` is used to transform the monad into one that may be run with `#eval`, after which it is run as usual.
- If the term's type is not in any of the supported monads, then it is treated as a pure value. The compiled code is run, and the result is displayed.

Auxiliary definitions or other environment modifications that result from elaborating the term in `#eval` are discarded. If the term is an action in a metaprogramming monad, then changes made to the environment by the running the monadic action are preserved.

Results are displayed using a `ToExpr`, `ToString`, or `Repr` instance, if they exist. If not, and `eval.derive.repr` is `true`, Lean attempts to derive a suitable `Repr` instance. It is an error if no suitable instance can be found or derived. Setting `eval.pp` to `false` disables the use of `ToExpr` instances by `#eval`.

▼ *Example: Displaying Output*

`#eval` cannot display functions:

```
#eval fun x => x + 1
```

```
could not synthesize a 'ToExpr', 'Repr', or 'ToString'
instance for type
  Nat → Nat
```

It is capable of deriving instances to display output that has no `ToString` or `Repr` instance:

```
inductive Quadrant where
  | nw | sw | se | ne

#eval Quadrant.nw
```

```
Quadrant.nw
```

The derived instance is not saved. Disabling `eval.derive.repr` causes `#eval` to fail:

```
set_option eval.derive.repr false
#eval Quadrant.nw
```

```
could not synthesize a 'ToExpr', 'Repr', or 'ToString'
instance for type
  Quadrant
```

option

`eval.pp`

Default value: `true`

('#eval' command) enables using 'ToExpr' instances to pretty print the result, otherwise uses 'Repr' or 'ToString' instances

option

`eval.type`

Default value: `false`

('#eval' command) enables pretty printing the type of the result

option

`eval.derive.repr`

Default value: `true`

('#eval' command) enables auto-deriving 'Repr' instances as a fallback

Monads can be given the ability to execute in `#eval` by defining a suitable `MonadLift` `MonadLift` is described in the section on lifting monads. or `MonadEval` instance. Just as `MonadLiftT` is the transitive closure of `MonadLift` instances, `MonadEvalT` is the transitive closure of `MonadEval` instances. As with `MonadLiftT` users should not define additional instances of `MonadEvalT` directly.

type class

```
MonadEval.{u, v, w} (m : semiOutParam (Type u → Type v))
  (n : Type u → Type w) : Type (max (max (u + 1) v) w)
```

Typeclass used for adapting monads. This is similar to `MonadLift`, but instances are allowed to make use of default state for the purpose of synthesizing such an instance, if necessary. Every `MonadLift` instance gives a `MonadEval` instance.

The purpose of this class is for the **`#eval`** command, which looks for a `MonadEval` *`m CommandElabM`* or `MonadEval` *`m`* `IO` instance.

Instance Constructor

`MonadEval.mk.{u, v, w}`

Methods

`monadEval : {α : Type u} → m α → n α`

Evaluates a value from monad *m* into monad *n*.

type class

```
MonadEvalT.{u, v, w} (m : Type u → Type v) (n : Type u → Type w) :
  Type (max (max (u + 1) v) w)
```

The transitive closure of `MonadEval`.

Instance Constructor

`MonadEvalT.mk.{u, v, w}`

Methods

`monadEval : {α : Type u} → m α → n α`

Evaluates a value from monad *m* into monad *n*.

3.2. Reducing Terms

The `#reduce` command repeatedly applies reductions to a term until no further reductions are possible. Reductions are performed under binders, but to avoid unexpected slowdowns, proofs and types are skipped unless the corresponding options to `#reduce` are enabled. Unlike `#eval` command, reduction cannot have side effects and the result is displayed as a term rather than via a `ToString` or `Repr` instance.

Generally speaking, `#reduce` is primarily useful for diagnosing issues with definitional equality and proof terms, while `#eval` is more suitable for computing the value of a term. In particular, functions defined using well-founded recursion or as partial fixpoints are either very slow to compute with the reduction engine, or will not reduce at all.

syntax

Reducing Terms

```
command ::= ...
    | #reduce ((proofs := true))? ((types := true))? term
```

`#reduce <expression>` reduces the expression `<expression>` to its normal form. This involves applying reduction rules until no further reduction is possible.

By default, proofs and types within the expression are not reduced. Use modifiers `(proofs := true)` and `(types := true)` to reduce them. Recall that propositions are types in Lean.

Warning: This can be a computationally expensive operation, especially for complex expressions.

Consider using `#eval <expression>` for simple evaluation/execution of expressions.

▼ *Example: Reducing Functions*

Reducing a term results in its normal form in Lean's logic. Because the underlying term is reduced and then displayed, there is no need for a `ToString` or Repr instance. Functions can be displayed just as well as any other term.

In some cases, this normal form is short and resembles a term that a person might write:

```
#reduce (fun x => x + 1)
```

```
fun x => x.succ
```

In other cases, the details of the elaboration of functions such as addition to Lean's core logic are exposed:

```
#reduce (fun x => 1 + x)
```

```
fun x => (Nat.rec ⟨fun x => x, PUnit.unit⟩ (fun n n_ih =>
⟨fun x => (n_ih.1 x).succ, n_ih⟩) x).1 1
```

3.3. Checking Types

syntax

Checking Types

`#check` can be used to elaborate a term and check its type.

```
command ::= ...
    | #check term
```

If the provided term is an identifier that is the name of a global constant, then `#check` prints its signature. Otherwise, the term is elaborated as a Lean term and its type is printed.

Elaboration of the term in `#check` does not require that the term is fully elaborated; it may contain metavariables. If the term as written *could* have a type, elaboration succeeds. If a required instance could never be synthesized, then elaboration fails; synthesis problems that are due to metavariables do not block elaboration.

▼ *Example: `#check` and Underdetermined Types*

In this example, the type of the list's elements is not determined, so the type contains a metavariable:

```
#check fun x => [x]
```

```
fun x => [x] : ?m.9 → List ?m.9
```

In this example, both the type of the terms being added and the result type of the addition are unknown, because HAdd allows terms of different types to be added. Behind the scenes, a metavariable represents the unknown HAdd instance.

```
#check fun x => x + x
```

```
fun x => x + x : (x : ?m.12) → ?m.19 x
```

syntax

Testing Type Errors

```
command ::= ...
    | #check_failure term
```

This variant of `#check` elaborates the term using the same process as `#check`. If elaboration succeeds, it is an error; if it fails, there is no error. The partially-elaborated term and any type information that was discovered are added to the message log.

▼ *Example: Checking for Type Errors*

Attempting to add a string to a natural number fails, as expected:

```
#check_failure "one" + 1
```

```
failed to synthesize
  HAdd String Nat ?m.96

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Nonetheless, a partially-elaborated term is available:

```
"one" + 1 : ?m.96
```

3.4. Synthesizing Instances

syntax

Synthesizing Instances

```
command ::= ...
    | #synth term
```

The #synth command attempts to synthesize an instance for the provided class. If it succeeds, then the resulting instance term is output.

3.5. Querying the Context

The #print family of commands are used to query Lean for information about definitions.

syntax

Printing Definitions

```
command ::= ...
    | #print ident
```

Prints the definition of a constant.

Printing a definition with #print prints the definition as a term. Theorems that were proved using tactics may be very large when printed as terms.

syntax

Printing Strings

```
command ::= ...
    | #print str
```

Adds the string literal to Lean's message log.

syntax

Printing Axioms

```
command ::= ...
    | #print axioms ident
```

Lists all axioms that the constant transitively relies on.

▼ *Example: Printing Axioms*

These two functions each swap the elements in a pair of bitvectors:

```
def swap (x y : BitVec 32) : BitVec 32 × BitVec 32 :=
  (y, x)

def swap' (x y : BitVec 32) : BitVec 32 × BitVec 32 :=
  let x := x ^^^ y
  let y := x ^^^ y
  let x := x ^^^ y
  (x, y)
```

They can be proven equal using function extensionality, the simplifier, and **`by_decide`**:

```
theorem swap_eq_swap' : swap = swap' := by
  funext x y
  simp only [swap, swap', Prod.mk.injEq]
  by_decide
```

The resulting proof makes use of a number of axioms:

```
#print axioms swap_eq_swap'
```

```
'swap_eq_swap'' depends on axioms: [propext,
Classical.choice, Lean.ofReduceBool, Quot.sound]
```

syntax

Printing Equations

The command `#print equations`, which can be abbreviated `#print eqns`, displays the equational lemmas for a function.

```
command ::= ...
    | #print equations ident

command ::= ...
    | #print eqns ident
```

▼ *Example: Printing Equations*

```
def intersperse (x : α) : List α → List α
  | y :: z :: zs => y :: x :: intersperse x (z :: zs)
```

```
  | xs => xs

#print equations intersperse
```

```
equations:
@[defeq] theorem intersperse.eq_1.{u_1} : ∀ {α : Type u_1} (x
y z : α) (zs : List α),
  intersperse x (y :: z :: zs) = y :: x :: intersperse x (z
:: zs)
theorem intersperse.eq_2.{u_1} : ∀ {α : Type u_1} (x : α)
(x_1 : List α),
  (∀ (y z : α) (zs : List α), x_1 = y :: z :: zs → False) →
intersperse x x_1 = x_1
```

It does not print the defining equation, nor the unfolding equation:

```
#check intersperse.eq_def
```

```
intersperse.eq_def.{u_1} {α : Type u_1} (x : α) (x✝ : List α)
:
  intersperse x x✝ =
    match x✝ with
    | y :: z :: zs => y :: x :: intersperse x (z :: zs)
    | xs => xs
```

```
#check intersperse.eq_unfold
```

```
intersperse.eq_unfold.{u_1} :
  @intersperse = fun {α} x x_1 =>
    match x_1 with
    | y :: z :: zs => y :: x :: intersperse x (z :: zs)
    | xs => xs
```

syntax

Scope Information

#where gives a description of the state of the current scope scope. This includes the current namespace, **open** namespaces, **universe** and **variable** commands, and options set with **set_option**.

```
command ::= ...
    | #where
```

▼ *Example: Scope Information*

The #where command displays all the modifications made to the current section scope, both in the current scope and in the scopes in which it is nested.

```
section
open Nat

namespace A
variable (n : Nat)
namespace B

open List
set_option pp.funBinderTypes true

#where

end A.B
end
```

```
namespace A.B

open Nat List

variable (n : Nat)

set_option pp.funBinderTypes true
```

syntax

Checking the Lean Version

Shows the current Lean version. Prints `Lean.versionString`.

```
command ::= ...
    | #version
```

3.6. Testing Output with `#guard_msgs`

The `#guard_msgs` command can be used to ensure that the messages output by a command are as expected. Together with the interaction commands in this section, it can be used to construct a file that will only elaborate if the output is as expected; such a file can be used as a test driver in Lake.

syntax

Documenting Expected Output

```
command ::= ...
    | docComment?
      #guard_msgs ((guardMsgsSpecElt,*))? in
      command
```

`/-- ... -/ #guard_msgs in` cmd captures the messages generated by the command cmd and checks that they match the contents of the docstring.

Basic example:

```
/--
error: unknown identifier 'x'
-/
#guard_msgs in
example : α := x
```

This checks that there is such an error and then consumes the message.

By default, the command captures all messages, but the filter condition can be adjusted. For example, we can select only warnings:

```
/--
warning: declaration uses 'sorry'
-/
#guard_msgs(warning) in
example : α := sorry
```

or only errors

```
#guard_msgs(error) in
example : α := sorry
```

In the previous example, since warnings are not captured there is a warning on `sorry`. We can drop the warning completely with

```
#guard_msgs(error, drop warning) in
example : α := sorry
```

In general, **`#guard_msgs`** accepts a comma-separated list of configuration clauses in parentheses:

```
#guard_msgs (configElt,*) in cmd
```

By default, the configuration list is `(check all, whitespace := normalized, ordering := exact)`.

Message filters select messages by severity:

- `info`, `warning`, `error`: (non-trace) messages with the given severity level.
- `trace`: trace messages
- `all`: all messages.

The filters can be prefixed with the action to take:

- `check` (the default): capture and check the message
- `drop`: drop the message
- `pass`: let the message pass through

If no filter is specified, `check all` is assumed. Otherwise, these filters are processed in left-to-right order, with an implicit `pass all` at the end.

Whitespace handling (after trimming leading and trailing whitespace):

- `whitespace := exact` requires an exact whitespace match.
- `whitespace := normalized` converts all newline characters to a space before matching (the default). This allows breaking long lines.
- `whitespace := lax` collapses whitespace to a single space before matching.

Message ordering:

- `ordering := exact` uses the exact ordering of the messages (the default).
- `ordering := sorted` sorts the messages in lexicographic order. This helps with testing commands that are non-deterministic in their ordering.

For example, `#guard_msgs (error, drop all) in cmd` means to check warnings and drop everything else.

The command elaborator has special support for `#guard_msgs` for linting. The `#guard_msgs` itself wants to capture linter warnings, so it elaborates the command it is attached to as if it were a top-level command. However, the command elaborator runs linters for *all* top-level commands, which would include `#guard_msgs` itself, and would cause duplicate and/or uncaptured linter warnings. The top-level command elaborator only runs the linters if `#guard_msgs` is not present.

▼ *Example: Testing Return Values*

The `#guard_msgs` command can ensure that a set of test cases pass:

```
def reverse : List α → List α := helper []
where
  helper acc
    | [] => acc
    | x :: xs => helper (x :: acc) xs

/-- info: [] -/
#guard_msgs in
#eval reverse ([] : List Nat)

/-- info: ['c', 'b', 'a'] -/
#guard_msgs in
#eval reverse "abc".toList
```

The behavior of the `#guard_msgs` command can be specified in three ways:

1. Providing a filter that selects a subset of messages to be checked
2. Specifying a whitespace comparison strategy
3. Deciding to sort messages by their content or by the order in which they were produced

These configuration options are provided in parentheses, separated by commas.

syntax

Specifying `#guard_msgs` Behavior

```
guardMsgsSpecElt ::=
    guardMsgsFilter

guardMsgsSpecElt ::= ...
    | whitespace := guardMsgsWhitespaceArg

guardMsgsSpecElt ::= ...
    | ordering := guardMsgsOrderingArg
```

There are three kinds of options for `#guard_msgs`: filters, whitespace comparison strategies, and orderings.

syntax

Output Filters for `#guard_msgs`

```
guardMsgsFilter ::=
    drop? all

guardMsgsFilter ::= ...
    | drop? info

guardMsgsFilter ::= ...
    | drop? warning

guardMsgsFilter ::= ...
    | drop? error
```

A message filter specification for **`#guard_msgs`**.

- `info`, `warning`, `error`: capture (non-trace) messages with the given severity level.
- `trace`: captures trace messages
- `all`: capture all messages.

The filters can be prefixed with

- `check` (the default): capture and check the message

- `drop`: drop the message
- `pass`: let the message pass through

If no filter is specified, `check all` is assumed. Otherwise, these filters are processed in left-to-right order, with an implicit `pass all` at the end.

syntax

Whitespace Comparison for `#guard_msgs`

```
guardMsgsWhitespaceArg ::=
    exact

guardMsgsWhitespaceArg ::= ...
    | lax

guardMsgsWhitespaceArg ::= ...
    | normalized
```

Leading and trailing whitespace is always ignored when comparing messages. On top of that, the following settings are available:

- `whitespace := exact` requires an exact whitespace match.
- `whitespace := normalized` converts all newline characters to a space before matching (the default). This allows breaking long lines.
- `whitespace := lax` collapses whitespace to a single space before matching.

The option `guard_msgs.diff` controls the content of the error message that `#guard_msgs` produces when the expected message doesn't match the produced message. By default, `#guard_msgs` shows a line-by-line difference, with a leading + used to indicate lines from the produced message and a leading − used to indicate lines from the expected message. When messages are large and only differ by a small amount, this can make it easier to notice where they differ. Setting `guard_msgs.diff` to `false` causes `#guard_msgs` to instead show just the produced message, which can be compared with the expected message in the source file. This can be convenient if the difference between the message is confusing or overwhelming.

option

`guard_msgs.diff`

Default value: `true`

When true, show a diff between expected and actual messages if they don't match.

▼ *Example: Displaying Differences*

The `#guard_msgs` command can be used to test definition of a rose tree `Tree` and a function `Tree.big` that creates them:

```
inductive Tree (α : Type u) : Type u where
  | val : α → Tree α
  | branches : List (Tree α) → Tree α

def Tree.big (n : Nat) : Tree Nat :=
  if n = 0 then .val 0
  else if n = 1 then .branches [.big 0]
  else .branches [.big (n / 2), .big (n / 3)]
```

However, it can be difficult to spot where test failures come from when the output is large:

```
set_option guard_msgs.diff false
/--
info: Tree.branches
  [Tree.branches
     [Tree.branches [Tree.branches [Tree.branches [Tree.val 0],
Tree.val 0], Tree.branches [Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 2], Tree.branches
[Tree.val 0]]],
   Tree.branches
     [Tree.branches [Tree.branches [Tree.val 0], Tree.branches
[Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 0], Tree.val 0]]]
-/
#guard_msgs in
#eval Tree.big 20
```

The evaluation produces:

```
Tree.branches
  [Tree.branches
     [Tree.branches [Tree.branches [Tree.branches [Tree.val
0], Tree.val 0], Tree.branches [Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 0],
Tree.branches [Tree.val 0]]],
   Tree.branches
     [Tree.branches [Tree.branches [Tree.val 0],
Tree.branches [Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 0], Tree.val
0]]]
```

Without `guard_msgs.diff`, the `#guard_msgs` command reports this error:

```
❌ Docstring on `#guard_msgs` does not match generated
message:

info: Tree.branches
```

```
  [Tree.branches
     [Tree.branches [Tree.branches [Tree.branches [Tree.val
0], Tree.val 0], Tree.branches [Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 0],
Tree.branches [Tree.val 0]]],
   Tree.branches
     [Tree.branches [Tree.branches [Tree.val 0],
Tree.branches [Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 0], Tree.val
0]]]
```

Enabling guard_msgs.diff highlights the differences instead, making the error more apparent:

```
set_option guard_msgs.diff true in
/--
info: Tree.branches
  [Tree.branches
     [Tree.branches [Tree.branches [Tree.branches [Tree.val 0],
Tree.val 0], Tree.branches [Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 2], Tree.branches
[Tree.val 0]]],
   Tree.branches
     [Tree.branches [Tree.branches [Tree.val 0], Tree.branches
[Tree.val 0]],
      Tree.branches [Tree.branches [Tree.val 0], Tree.val 0]]]
-/
#guard_msgs in
#eval Tree.big 20
```

```
❌ Docstring on `#guard_msgs` does not match generated
message:

  info: Tree.branches
    [Tree.branches
       [Tree.branches [Tree.branches [Tree.branches [Tree.val
0], Tree.val 0], Tree.branches [Tree.val 0]],
-       Tree.branches [Tree.branches [Tree.val 2],
Tree.branches [Tree.val 0]]],
+       Tree.branches [Tree.branches [Tree.val 0],
Tree.branches [Tree.val 0]]],
     Tree.branches
       [Tree.branches [Tree.branches [Tree.val 0],
Tree.branches [Tree.val 0]],
        Tree.branches [Tree.branches [Tree.val 0], Tree.val
0]]]
```

3.7. Formatted Output

The Repr type class is used to provide a standard representation for data that can be parsed and evaluated to obtain an equivalent value. This is not a strict correctness criterion: for some types, especially those with embedded propositions, it is impossible to achieve. However, the output produced by a Repr instance should be as close as possible to something that can be parsed and evaluated.

In addition to being machine-readable, this representation should be convenient for humans to understand—in particular, lines should not be too long, and nested values should be indented. This is achieved through a two-step process:

1. The Repr instance produces an intermediate document of type Std.Format, which compactly represents a *set* of strings that differ with respect to the placement of newlines and indentation.
2. A rendering process selects the “best” representative from the set, according to criteria such as a desired maximum line length.

In particular, Std.Format can be built compositionally, so Repr instances don't need to take the surrounding indentation context into account.

3.7.1. Format

A Format (i The API described here is an adaptation of Wadler's (Philip Wadler, 2003. “A Prettier Printer”. In *The Fun of Programming, A symposium in honour of Professor Richard Bird's 60th birthday.*) It has been modified to be efficient in a strict language and with support for additional features such as metadata tags.) is a compact representation of a set of strings. The most important Format operations are:

Strings

A String can be made into a Format using the text constructor. This constructor is registered as a coercion from String to Format, so it is often unnecessary to invoke it explicitly. `text str` represents the singleton set that contains only `str`. If the string contains newline characters (`'\n'`), then they are unconditionally inserted as newlines into the resulting output, regardless of groups. They are, however, indented according to the current indentation level.

Appending

Two Formats can be appended using the ++ operator from the Append Format instance.

Groups and Newlines

The constructor line represents the set that contains both `"\n" ++ indent` and `" "`, where `indent` is a string with enough spaces to indent the line correctly. Imperatively, it can be thought of as a newline that will be “flattened” to a space if there is sufficient room on the current line. Newlines occur in *groups*: the nearest enclosing application of the group operator determines which group the newline belongs to. By default, either all lines in a

group represent "\n" or all represent " "; groups may also be configured to fill lines, in which case the minimal number of `line`s in the group represent "\n". Uses of `line` that do not belong to a group always represent "\n".

Indentation

When a newline is inserted, the output is also indented. `nest n` increases the indentation of a document by *n* spaces. This is not sufficient to represent all Lean syntax, which sometimes requires that columns align exactly. `align` is a document that ensures that the output string is at the current indentation level, inserting just spaces if possible, or a newline followed by spaces if needed.

Tagging

Lean's interactive features require the ability to associate output with the underlying values that they represent. This allows Lean development environments to present elaborated terms when hovering over terms proof states or error messages, for example. Documents can be *tagged* with a `Nat` value *n* using `tag n`; these `Nat`s should be mapped to the underlying value in a side table.

▼ *Example: Widths and Newlines*

```
open Std Format
```

The helper `parenSeq` creates a parenthesized sequence, with grouping and indentation to make it responsive to different output widths.

```
def parenSeq (xs : List Format) : Format :=
  group <|
    nest 2 (text "(" ++ line ++ joinSep xs line) ++
    line ++
    ")"
```

This document represents a parenthesized sequence of numbers:

```
def lst : Format := parenSeq nums
where nums := [1, 2, 3, 4, 5].map (text s!"{·}")
```

Rendering it with the default line width of 120 characters places the entire sequence on one line:

```
#eval IO.println lst.pretty
```

```
( 1 2 3 4 5 )
```

Because all the `line`s belong to the same `group`, they will either all be rendered as spaces or all be rendered as newlines. If only 9 characters are available, all of the `line`s in `lst` become newlines:

```
#eval IO.println (lst.pretty (width := 9))
```

```
(
  1
  2
  3
  4
  5
)
```

This document contains three copies of `lst` in a further parenthesized sequence:

```
def lsts := parenSeq [lst, lst, lst]
```

At the default width, it remains on one line:

```
#eval IO.println lsts.pretty
```

```
( ( 1 2 3 4 5 ) ( 1 2 3 4 5 ) ( 1 2 3 4 5 ) )
```

If only 20 characters are available, each occurrence of `lst` ends up on its own line. This is because converting the outer `group` to newlines is sufficient to keep the string within 20 columns:

```
#eval IO.println (lsts.pretty (width := 20))
```

```
(
  ( 1 2 3 4 5 )
  ( 1 2 3 4 5 )
  ( 1 2 3 4 5 )
)
```

If only 10 characters are available, each number must be on its own line:

```
#eval IO.println (lsts.pretty (width := 10))
```

```
(
  (
    1
    2
    3
    4
    5
  )
  (
    1
    2
    3
    4
    5
  )
  (
    1
    2
    3
```

```
    4
    5
  )
)
```

▼ *Example: Grouping and Filling*

```
open Std Format
```

The helper `parenSeq` creates a parenthesized sequence, with each element placed on a new line and indented:

```
def parenSeq (xs : List Format) : Format :=
  nest 2 (text "(" ++ line ++ joinSep xs line) ++
  line ++
  ")"
```

`nums` contains the numbers one through twenty, as a list of formats:

```
def nums : List Format := Nat.fold 20 (init := []) fun i _ ys =>
text s!"{20 - i}" :: ys

#eval nums
```

Because `parenSeq` does not introduce any groups, the resulting document is rendered on a single line:

```
#eval IO.println (pretty (parenSeq nums))
```

This can be fixed by grouping them. `grouped` does so with `group`, while `filled` does so with `fill`.

```
def grouped := group (parenSeq nums)
def filled := fill (parenSeq nums)
```

Both grouping operators cause uses of `line` to render as spaces. Given sufficient space, both render on a single line:

```
#eval IO.println (pretty grouped)
```

```
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 )
```

```
#eval IO.println (pretty filled)
```

```
( 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 )
```

However, difference become apparent when there is not sufficient space on a single line. Unless *all* newlines in a `group` can be spaces, none can:

```
#eval IO.println (pretty (width := 30) grouped)
```

```
(
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
  11
  12
  13
  14
  15
  16
  17
  18
  19
  20
)
```

Using `fill`, on the other hand, only inserts newlines as required to avoid being two wide:

```
#eval IO.println (pretty (width := 30) filled)
```

```
( 1 2 3 4 5 6 7 8 9 10 11 12
  13 14 15 16 17 18 19 20 )
```

The behavior of `fill` can be seen clearly with longer sequences:

```
#eval IO.println (pretty (width := 30) (fill (parenSeq (nums ++ nums ++ nums ++ nums))))
```

```
( 1 2 3 4 5 6 7 8 9 10 11 12
  13 14 15 16 17 18 19 20 1 2
  3 4 5 6 7 8 9 10 11 12 13 14
  15 16 17 18 19 20 1 2 3 4 5
  6 7 8 9 10 11 12 13 14 15 16
  17 18 19 20 1 2 3 4 5 6 7 8
  9 10 11 12 13 14 15 16 17 18
  19 20 )
```

▼ *Example: Newline Characters in Strings*

Including a newline character in a string causes the rendering process to unconditionally insert a newline. These newlines do, however, respect the current indentation level.

The document `str` consists of an embedded string with two newlines:

```
open Std Format

def str : Format := text "abc\nxyz\n123"
```

Printing the string both with and without grouping results in the newlines being used:

```
#eval IO.println str.pretty
```

```
abc
xyz
123
```

```
#eval IO.println (group str).pretty
```

```
abc
xyz
123
```

Because the string does not terminate with a newline, the last line of the first string is on the same line as the first line of the second string:

```
#eval IO.println (str ++ str).pretty
```

```
abc
xyz
123abc
xyz
123
```

Increasing the indentation level, however, causes all three lines of the string to begin at the same column:

```
#eval IO.println (text "It is:" ++ indentD str).pretty
```

```
It is:
  abc
  xyz
  123
```

```
#eval IO.println (nest 8 <| text "It is:" ++ align true ++ str).pretty
```

```
It is:  abc
        xyz
        123
```

3.7.1.1. Documents

inductive type

```
Std.Format : Type
```

A representation of a set of strings, in which the placement of newlines and indentation differ.

Given a specific line width, specified in columns, the string that uses the fewest lines can be selected.

The pretty-printing algorithm is based on Wadler's paper *A Prettier Printer*.

Constructors

| `nil : Std.Format`

The empty format.

| `line : Std.Format`

A position where a newline may be inserted if the current group does not fit within the allotted column width.

| `align (force : Bool) : Std.Format`

`align` tells the formatter to pad with spaces to the current indentation level, or else add a newline if we are already at or past the indent.

If *`force`* is true, then it will pad to the indent even if it is in a flattened group.

Example:

```
open Std Format in
#eval IO.println (nest 2 <| "." ++ align ++ "a" ++ line ++ "b")
```

```
. a
  b
```

| `text : String → Std.Format`

A node containing a plain string.

If the string contains newlines, the formatter emits them and then indents to the current level.

| `nest (indent : Int) (f : Std.Format) : Std.Format`

`nest` *`indent f`* increases the current indentation level by *`indent`* while rendering *`f`*.

Example:

```
open Std Format in
def fmtList (l : List Format) : Format :=
  let f := joinSep l  (", " ++ Format.line)
  group (nest 1 <| "[" ++ f ++ "]")
```

This will be written all on one line, but if the text is too large, the formatter will put in linebreaks after the commas and indent later lines by 1.

| `append : Std.Format → Std.Format → Std.Format`

Concatenation of two Formats.

```
group :
  Std.Format →
    (behavior :
        optParam Std.Format.FlattenBehavior
          Std.Format.FlattenBehavior.allOrNone) →
      Std.Format
```

Creates a new flattening group for the given inner Format.

```
tag : Nat → Std.Format → Std.Format
```

Used for associating auxiliary information (e.g. Exprs) with Format objects.

inductive type

Std.Format.FlattenBehavior : Type

Determines how groups should have linebreaks inserted when the text would overfill its remaining space.

- `allOrNone` will make a linebreak on every `Format.line` in the group or none of them.

```
[1,
 2,
 3]
```

- `fill` will only make linebreaks on as few `Format.lines` as possible:

```
[1, 2,
 3]
```

Constructors

```
allOrNone : Std.Format.FlattenBehavior
```

Either all Format.lines in the group will be newlines, or all of them will be spaces.

```
fill : Std.Format.FlattenBehavior
```

As few Format.lines in the group as possible will be newlines.

def

Std.Format.fill (f : Std.Format) : Std.Format

Creates a group in which as few `Format.lines` as possible are rendered as newlines.

This is an alias for `Format.group`, with `FlattenBehavior` set to `fill`.

3.7.1.2. Empty Documents

The empty string does not have a single unique representative in `Std.Format`. All of the following represent the empty string:

- `.nil`
- `.text ""`
- `.text "" ++ .nil`
- `.nil ++ .text ""`

Use `Std.Format.isEmpty` to check whether a document contains zero characters, and `Std.Format.isNil` to specifically check whether it is the constructor `Std.Format.nil`.

def

```
Std.Format.isEmpty : Std.Format → Bool
```

Checks whether the given format contains no characters.

def

```
Std.Format.isNil : Std.Format → Bool
```

Checks whether a `Format` is the constructor `Format.nil`.

This does not check whether the resulting rendered strings are always empty. To do that, use `Format.isEmpty`.

3.7.1.3. Sequences

The operators in this section are useful when there is some kind of repeated content, such as the elements of a list. This is typically done by including `line` in their separator parameters, using a bracketing operator

def

```
Std.Format.join (xs : List Std.Format) : Std.Format
```

Concatenates a list of `Format`s with `++`.

def

```
Std.Format.joinSep.{u} {α : Type u} [Std.ToFormat α] :
  List α → Std.Format → Std.Format
```

Intercalates the given list with the given `sep` format.

The list items are formatting using `ToFormat.format`.

def

```
Std.Format.prefixJoin.{u} {α : Type u} [Std.ToFormat α]
  (pre : Std.Format) : List α → Std.Format
```

Concatenates the given list after prepending `pre` to each element.

The list items are formatting using `ToFormat.format`.

def

```
Std.Format.joinSuffix.{u} {α : Type u} [Std.ToFormat α] :
  List α → Std.Format → Std.Format
```

Concatenates the given list after appending the given suffix to each element.

The list items are formatting using `ToFormat.format`.

3.7.1.4. Indentation

These operators make it easier to achieve a consistent indentation style on top of `Std.Format.nest`.

def

```
Std.Format.nestD (f : Std.Format) : Std.Format
```

Increases the indentation level by the default amount.

def

```
Std.Format.defIndent : Nat
```

The default indentation level, which is two spaces.

def

```
Std.Format.indentD (f : Std.Format) : Std.Format
```

Insert a newline and then f, all nested by the default indent amount.

3.7.1.5. Brackets and Parentheses

These operators make it easier to achieve a consistent parenthesization style.

def

```
Std.Format.bracket (l : String) (f : Std.Format) (r : String) :
  Std.Format
```

Creates a format `l ++ f ++ r` with a flattening group, nesting the contents by the length of `l`.

The group's `FlattenBehavior` is `allOrNone`; for `fill` use `Std.Format.bracketFill`.

def

```
Std.Format.sbracket (f : Std.Format) : Std.Format
```

Creates the format `"[" ++ f ++ "]"` with a flattening group, nesting by one space.

`sbracket` is short for “square bracket”.

def

```
Std.Format.paren (f : Std.Format) : Std.Format
```

Creates the format `"(" ++ f ++ ")"` with a flattening group, nesting by one space.

def

```
Std.Format.bracketFill (l : String) (f : Std.Format) (r : String) :
  Std.Format
```

Creates a format `l ++ f ++ r` with a flattening group, nesting the contents by the length of `l`.

The group's `FlattenBehavior` is `fill`; for `allOrNone` use `Std.Format.bracketFill`.

3.7.1.6. Rendering

The `ToString Std.Format` instance invokes `Std.Format.pretty` with its default arguments.

There are two ways to render a document:

- Use `pretty` to construct a `String`. The entire string must be constructed up front before any can be sent to a user.
- Use `prettyM` to incrementally emit the `String`, using effects in some `Monad`. As soon as each line is rendered, it is emitted. This is suitable for streaming output.

def

```
Std.Format.pretty (f : Std.Format) (width : Nat :=
Std.Format.defWidth)
  (indent column : Nat := 0) : String
```

Renders a `Format` to a string.

- *width*: the total width
- *indent*: the initial indentation to use for wrapped lines (subsequent wrapping may increase the indentation)
- *column*: begin the first line wrap *column* characters earlier than usual (this is useful when the output String will be printed starting at *column*)

def

```
Std.Format.defWidth : Nat
```

The default width of the targeted output, which is 120 columns.

def

```
Std.Format.prettyM {m : Type → Type} (f : Std.Format) (w : Nat)
  (indent : Nat := 0) [Monad m] [Std.Format.MonadPrettyFormat m] :
  m Unit
```

Renders a `Format` using effects in the monad *m*, using the methods of `MonadPrettyFormat`.

Each line is emitted as soon as it is rendered, rather than waiting for the entire document to be rendered.

- *w*: the total width
- *indent*: the initial indentation to use for wrapped lines (subsequent wrapping may increase the indentation)

type class

```
Std.Format.MonadPrettyFormat (m : Type → Type) : Type
```

A monad that can be used to incrementally render `Format` objects.

Instance Constructor

Std.Format.MonadPrettyFormat.mk

Methods

```
pushOutput : String → m Unit
```

Emits the string `s`.

`pushNewline : Nat → m Unit`

Emits a newline followed by `indent` columns of indentation.

`currColumn : m Nat`

Gets the current column at which the next string will be emitted.

`startTag : Nat → m Unit`

Starts a region tagged with `tag`.

`endTags : Nat → m Unit`

Exits the scope of `count` opened tags.

3.7.1.7. The `ToFormat` Class

The `Std.ToFormat` class is used to provide a standard means to format a value, with no expectation that this formatting be valid Lean syntax. These instances are used in error messages and by some of the sequence concatenation operators.

type class

`Std.ToFormat.{u} (α : Type u) : Type u`

Specifies a "user-facing" way to convert from the type α to a `Format` object. There is no expectation that the resulting string is valid code.

The `Repr` class is similar, but the expectation is that instances produce valid Lean code.

Instance Constructor

`Std.ToFormat.mk.{u}`

Methods

`format : α → Std.Format`

3.7.2. `Repr`

A `Repr` instance describes how to represent a value as a `Std.Format`. Because they should emit valid Lean syntax, these instances need to take precedence into account. Inserting the maximal number of parentheses would work, but it makes it more difficult for humans to read the resulting output.

type class

`Repr.{u} (α : Type u) : Type u`

The standard way of turning values of some type into `Format`.

When rendered this `Format` should be as close as possible to something that can be parsed as the input value.

Instance Constructor

`Repr.mk.{u}`

Methods

`reprPrec : α → Nat → Std.Format`

Turn a value of type *α* into a `Format` at a given precedence. The precedence value can be used to avoid parentheses if they are not necessary.

def

`repr.{u_1} {α : Type u_1} [Repr α] (a : α) : Std.Format`

Turns *a* into a `Format` using its `Repr` instance. The precedence level is initially set to 0.

def

`reprStr.{u_1} {α : Type u_1} [Repr α] (a : α) : String`

Turns *a* into a `String` using its `Repr` instance, rendering the `Format` at the default width of 120 columns.

The precedence level is initially set to 0.

▼ *Example: Maximal Parentheses*

The type `NatOrInt` can contain a `Nat` or an `Int`:

```
inductive NatOrInt where
  | nat : Nat → NatOrInt
  | int : Int → NatOrInt
```

This `Repr NatOrInt` instance ensures that the output is valid Lean syntax by inserting many parentheses:

```
instance : Repr NatOrInt where
  reprPrec x _ :=
    .nestD <| .group <|
      match x with
      | .nat n =>
```

```
          .text "(" ++ "NatOrInt.nat" ++ .line ++ "(" ++ repr n
++ "))"
      | .int i =>
          .text "(" ++ "NatOrInt.int" ++ .line ++ "(" ++ repr i
++ "))"
```

Whether it contains a `Nat`, a non-negative `Int`, or a negative `Int`, the result can be parsed:

```
open NatOrInt in
#eval do
  IO.println <| repr <| nat 3
  IO.println <| repr <| int 5
  IO.println <| repr <| int (-5)
```

```
(NatOrInt.nat (3))
(NatOrInt.int (5))
(NatOrInt.int (-5))
```

However, `(NatOrInt.nat (3))` is not particularly idiomatic Lean, and redundant parentheses can make it difficult to read large expressions.

The method `Repr.reprPrec` has the following signature:

```
Repr.reprPrec.{u} {α : Type u} [Repr α] : α → Nat → Std.Format
```

The first explicit parameter is the value to be represented, while the second is the precedence of the context in which it occurs. This precedence can be used to decide whether to insert parentheses: if the precedence of the syntax being produced by the instance is greater than that of its context, parentheses are necessary.

3.7.2.1. How To Write a `Repr` Instance

Lean can produce an appropriate `Repr` instance for most types automatically using instance deriving. In some cases, however, it's necessary to write an instance by hand:

When writing a custom `Repr` instance, please follow these conventions:

Precedence

Check precedence, adding parentheses as needed, and pass the correct precedence to the `reprPrec` instances of embedded data. Each instance is responsible for surrounding itself in parentheses if needed; instances should generally not parenthesize recursive calls to `reprPrec`.

Function application has the maximum precedence, **`max_prec`**. The helpers `Repr.addAppParen` and `reprArg` respectively insert parentheses around applications when needed and pass the appropriate precedence to function arguments.

Fully-Qualified Names

A Repr instance does have access to the set of open namespaces in a given position. All names of constants in the environment should be fully qualified to remove ambiguity.

Default Nesting

Nested data should be indented using nestD to ensure consistent indentation across instances.

Grouping and Line Breaks

The output of every Repr instance that includes line breaks should be surrounded in a group. Furthermore, if the resulting code contains notional expressions that are nested, a group should be inserted around each nested level. Line breaks should usually be inserted in the following positions:

- Between a constructor and each of its arguments
- After `:=`
- After `,`
- Between the opening and closing braces of structure instance notation and its contents
- After, but not before, an infix operator

Parentheses and Brackets

Parentheses and brackets should be inserted using Std.Format.bracket or its specializations Std.Format.paren for parentheses and Std.Format.sbracket for square brackets. These operators align the contents of the parenthesized or bracketed expression in the same way that Lean's do. Trailing parentheses and brackets should not be placed on their own line, but rather stay with their contents.

def

```
Repr.addAppParen (f : Std.Format) (prec : Nat) : Std.Format
```

Adds parentheses to *f* if the precedence *prec* from the context is at least that of function application.

Together with reprArg, this can be used to correctly parenthesize function application syntax.

def

```
reprArg.{u_1} {α : Type u_1} [Repr α] (a : α) : Std.Format
```

Turns *a* into a Format using its Repr instance, with the precedence level set to that of function application.

Together with Repr.addAppParen, this can be used to correctly parenthesize function application syntax.

▼ *Example: Inductive Types with Constructors*

The inductive type `N.NatOrInt` can contain a `Nat` or an `Int`:

```
namespace N

inductive NatOrInt where
  | nat : Nat → NatOrInt
  | int : Int → NatOrInt
```

The `Repr NatOrInt` instance adheres to the conventions:

- The right-hand side is a function application, so it uses `Repr.addAppParen` to add parentheses if necessary.
- Parentheses are wrapped around the entire body with no additional `line`s.
- The entire function application is grouped, and it is nested the default amount.
- The function is separated from its parameters by a use of `line`; this newline will usually be a space because the `Repr Nat` and `Repr Int` instances are unlikely to produce long output.
- Recursive calls to `reprPrec` pass **`max_prec`** because they are in function parameter positions, and function application has the highest precedence.

```
instance : Repr NatOrInt where
  reprPrec
    | .nat n =>
      Repr.addAppParen <|
        .group <| .nestD <|
          "N.NatOrInt.nat" ++ .line ++ reprPrec n max_prec
    | .int i =>
      Repr.addAppParen <|
        .group <| .nestD <|
          "N.NatOrInt.int" ++ .line ++ reprPrec i max_prec

#eval IO.println (repr (NatOrInt.nat 5))
```

```
N.NatOrInt.nat 5
```

```
#eval IO.println (repr (NatOrInt.int 5))
```

```
N.NatOrInt.int 5
```

```
#eval IO.println (repr (NatOrInt.int (-5)))
```

```
N.NatOrInt.int (-5)
```

```
#eval IO.println (repr (some (NatOrInt.int (-5))))
```

```
some (N.NatOrInt.int (-5))
```

```
#eval IO.println (repr <| (List.range 10).map (NatOrInt.nat))
```

```
[N.NatOrInt.nat 0,
 N.NatOrInt.nat 1,
 N.NatOrInt.nat 2,
 N.NatOrInt.nat 3,
 N.NatOrInt.nat 4,
 N.NatOrInt.nat 5,
 N.NatOrInt.nat 6,
 N.NatOrInt.nat 7,
 N.NatOrInt.nat 8,
 N.NatOrInt.nat 9]
```

```
#eval IO.println (Std.Format.pretty (width := 3) (repr <|
(List.range 10).map (NatOrInt.nat)))
```

```
[N.NatOrInt.nat
   0,
 N.NatOrInt.nat
   1,
 N.NatOrInt.nat
   2,
 N.NatOrInt.nat
   3,
 N.NatOrInt.nat
   4,
 N.NatOrInt.nat
   5,
 N.NatOrInt.nat
   6,
 N.NatOrInt.nat
   7,
 N.NatOrInt.nat
   8,
 N.NatOrInt.nat
   9]
```

▼ *Example: Infix Syntax*

```
inductive AddExpr where
  | nat : Nat → AddExpr
  | add : AddExpr → AddExpr → AddExpr

instance : OfNat AddExpr n where
  ofNat := .nat n

instance : Add AddExpr where
  add := .add

protected def AddExpr.reprPrec : AddExpr → Nat → Std.Format
```

```
  | .nat n, p  =>
    Repr.reprPrec n p
  | .add e1 e2, p =>
    let out : Std.Format :=
      .nestD <| .group <|
        AddExpr.reprPrec e1 64 ++ " " ++ "+" ++ .line ++
        AddExpr.reprPrec e2 65
    if p ≥ 65 then out.paren else out

instance : Repr AddExpr := ⟨AddExpr.reprPrec⟩

#eval IO.println (repr (((2 + 3) + 4) : AddExpr))
#eval IO.println (repr ((2 + 3 + 4) : AddExpr))
#eval IO.println (repr ((2 + (3 + 4)) : AddExpr))
#eval IO.println (repr ([2 + (3 + 4), (2 + 3) + 4] : List
AddExpr))
#eval IO.println <| (repr ([2 + (3 + 4), (2 + 3) + 4] : List
AddExpr)).pretty (width := 0)
```

3.7.2.2. Atomic Types

When the elements of a list are sufficiently small, it can be both difficult to read and wasteful of space to render the list with one element per line. To improve readability, `List` has two `Repr` instances: one that uses `Std.Format.bracket` for its contents, and one that uses `Std.Format.bracketFill`. The latter is defined after the former and is thus selected when possible; however, it requires an instance of the empty type class `ReprAtom`.

If the `Repr` instance for a type never generates spaces or newlines, then it should have a `ReprAtom` instance. Lean has `ReprAtom` instances for types such as `String`, `UInt8`, `Nat`, `Char`, and `Bool`.

type class

```
ReprAtom.{u} (α : Type u) : Type
```

Auxiliary class for marking types that should be considered atomic by `Repr` methods. We use it at `Repr (List α)` to decide whether `bracketFill` should be used or not.

Instance Constructor

`ReprAtom.mk.{u}`

▼ *Example: Atomic Types and `Repr`*

All constructors of the inductive type `ABC` are without parameters:

```
inductive ABC where
  | a
  | b
```

```
  | c
deriving Repr
```

The derived `Repr ABC` instance is used to display lists:

```
def abc : List ABC := [.a, .b, .c]

def abcs : List ABC := abc ++ abc ++ abc

#eval IO.println ((repr abcs).pretty (width := 14))
```

Because of the narrow width, line breaks are inserted:

```
[ABC.a,
 ABC.b,
 ABC.c,
 ABC.a,
 ABC.b,
 ABC.c,
 ABC.a,
 ABC.b,
 ABC.c]
```

However, converting the list to a `List Nat` leads to a differently-formatted result.

```
def ABC.toNat : ABC → Nat
  | .a => 0
  | .b => 1
  | .c => 2

#eval IO.print ((repr (abcs.map ABC.toNat)).pretty (width := 14))
```

There are far fewer line breaks:

```
[0, 1, 2, 0,
 1, 2, 0, 1,
 2]
```

This is because of the existence of a `ReprAtom Nat` instance. Adding one for `ABC` leads to similar behavior:

```
instance : ReprAtom ABC := ⟨⟩

#eval IO.println ((repr abcs).pretty (width := 14))
```

```
[ABC.a, ABC.b,
 ABC.c, ABC.a,
 ABC.b, ABC.c,
 ABC.a, ABC.b,
 ABC.c]
```

4. The Type System

Terms, also known as *expressions*, are the fundamental units of meaning in Lean's core language. They are produced from user-written syntax by the elaborator. Lean's type system relates terms to their *types*, which are also themselves terms. Types can be thought of as denoting sets, while terms denote individual elements of these sets. A term is *well-typed* if it has a type under the rules of Lean's type theory. Only well-typed terms have a meaning.

Terms are a dependently typed λ-calculus: they include function abstraction, application, variables, and `let`-bindings. In addition to bound variables, variables in the term language may refer to constructors, type constructors, recursors, defined constants, or opaque constants. Constructors, type constructors, recursors, and opaque constants are not subject to substitution, while defined constants may be replaced with their definitions.

A *derivation* demonstrates the well-typedness of a term by explicitly indicating the precise inference rules that are used. Implicitly, well-typed terms can stand in for the derivations that demonstrate their well-typedness. Lean's type theory is explicit enough that derivations can be reconstructed from well-typed terms, which greatly reduces the overhead that would be incurred from storing a complete derivation, while still being expressive enough to represent modern research mathematics. This means that proof terms are sufficient evidence of the truth of a theorem and are amenable to independent verification.

In addition to having types, terms are also related by *definitional equality*. This is the mechanically-checkable relation that syntactically equates terms modulo their computational behavior. Definitional equality includes the following forms of reduction:

β (beta)
: Applying a function abstraction to an argument by substitution for the bound variable

δ (delta)
: Replacing occurrences of defined constants by the definition's value

ι (iota)
: Reduction of recursors whose targets are constructors (primitive recursion)

ζ (zeta)
: Replacement of let-bound variables by their defined values

Quotient reduction
: Reduction of the quotient type's function lifting operator when applied to an element of a quotient

Terms in which all possible reductions have been carried out are in *normal form*.

Definitional equality includes η-equivalence of functions and single-constructor inductive types. That is, **`fun`** *`x`* `=>` `f` *`x`* is definitionally equal to `f`, and `S.mk x.f1 x.f2` is definitionally equal to `x`, if `S` is a structure with fields `f1` and `f2`. It also features *proof irrelevance*: any two proofs of the

same proposition are definitionally equal. It is reflexive and symmetric, but not transitive.

Definitional equality is used by conversion: if two terms are definitionally equal, and a given term has one of them as its type, then it also has the other as its type. Because definitional equality includes reduction, types can result from computations over data.

▼ *Example: Computing types*

When passed a natural number, the function `LengthList` computes a type that corresponds to a list with precisely that many entries in it:

```
def LengthList (α : Type u) : Nat → Type u
  | 0 => PUnit
  | n + 1 => α × LengthList α n
```

Because Lean's tuples nest to the right, multiple nested parentheses are not needed:

```
example : LengthList Int 0 := ()

example : LengthList String 2 :=
  ("Hello", "there", ())
```

If the length does not match the number of entries, then the computed type will not match the term:

```
example : LengthList String 5 :=
  ("Wrong", "number", ())
```

```
Application type mismatch: In the application
  ("number", ())
the argument
  ()
has type
  Unit : Type
but is expected to have type
  LengthList String 3 : Type
```

The basic types in Lean are universes, function types, the quotient former `Quot`, and type constructors of inductive types. Defined constants, applications of recursors, function application, axioms or opaque constants may additionally give types, just as they can give rise to terms in any other type.

4.1. Functions

Function types are a built-in feature of Lean. Functions map the values of one type (the *domain*) into those of another type (the *codomain*), and *function types* specify the domain and codomain of functions.

There are two kinds of function type:

Dependent

Dependent function types explicitly name the parameter, and the function's codomain may refer explicitly to this name. Because types can be computed from values, a dependent function may return values from any number of different types, depending on its argument. Dependent functions are sometimes referred to as *dependent products*, because they correspond to an indexed product of sets.

Non-Dependent

Non-dependent function types do not include a name for the parameter, and the codomain does not vary based on the specific argument provided.

▼ *Example: Dependent Function Types*

The function `two` returns values in different types, depending on which argument it is called with:

```
def two : (b : Bool) → if b then Unit × Unit else String :=
  fun b =>
    match b with
    | true => ((), ())
    | false => "two"
```

The body of the function cannot be written with `if...then...else...` because it does not refine types the same way that `match` does.

In Lean's core language, all function types are dependent: non-dependent function types are dependent function types in which the parameter name does not occur in the codomain. Additionally, two dependent function types that have different parameter names may be definitionally equal if renaming the parameter makes them equal. However, the Lean elaborator does not introduce a local binding for non-dependent functions' parameters.

▼ *Example: Definitional Equality of Dependent and Non-Dependent Functions*

The types `(x : Nat) → String` and `Nat → String` are definitionally equal:

```
example : ((x : Nat) → String) = (Nat → String) := rfl
```

Similarly, the types `(n : Nat) → n + 1 = 1 + n` and `(k : Nat) → k + 1 = 1 + k` are definitionally equal:

```
example : ((n : Nat) → n + 1 = 1 + n) = ((k : Nat) → k + 1 = 1 + k) := rfl
```

▼ *Example: Non-Dependent Functions Don't Bind Variables*

A dependent function is required in the following statement that all elements of an array are non-zero:

```
def AllNonZero (xs : Array Nat) : Prop :=
  (i : Nat) → (lt : i < xs.size) → xs[i] ≠ 0
```

This is because the elaborator for array access requires a proof that the index is in bounds. The non-dependent version of the statement does not introduce this assumption:

```
def AllNonZero (xs : Array Nat) : Prop :=
  (i : Nat) → (i < xs.size) → xs[i] ≠ 0
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed,
and 'Panic' error message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that
index is valid
xs : Array Nat
i : Nat
⊢ i < xs.size
```

While the core type theory does not feature implicit parameters, function types do include an indication of whether the parameter is implicit. This information is used by the Lean elaborator, but it does not affect type checking or definitional equality in the core theory and can be ignored when thinking only about the core type theory.

▼ *Example: Definitional Equality of Implicit and Explicit Function Types*

The types `{α : Type} → (x : α) → α` and `(α : Type) → (x : α) → α` are definitionally equal, even though the first parameter is implicit in one and explicit in the other.

```
example :
    ({α : Type} → (x : α) → α)
    =
    ((α : Type) → (x : α) → α)
  := rfl
```

4.1.1. Function Abstractions

In Lean's type theory, functions are created using *function abstractions* that bind a variable. (i) In various communities, function abstractions are also known as *lambdas*, due to Alonzo Church's notation for them, or *anonymous functions* because they don't need to be defined with a name in the

global environment. When the function is applied, the result is found by β-reduction: substituting the argument for the bound variable. In compiled code, this happens strictly: the argument must already be a value. When type checking, there are no such restrictions; the equational theory of definitional equality allows β-reduction with any term.

In Lean's term language, function abstractions may take multiple parameters or use pattern matching. These features are translated to simpler operations in the core language, where all functions abstractions take exactly one parameter. Not all functions originate from abstractions: type constructors, constructors, and recursors may have function types, but they cannot be defined using function abstractions alone.

4.1.2. Currying

In Lean's core type theory, every function maps each element of the domain to a single element of the codomain. In other words, every function expects exactly one parameter. Multiple-parameter functions are implemented by defining higher-order functions that, when supplied with the first parameter, return a new function that expects the remaining parameters. This encoding is called *currying*, popularized by and named after Haskell B. Curry. Lean's syntax for defining functions, specifying their types, and applying them creates the illusion of multiple-parameter functions, but the result of elaboration contains only single-parameter functions.

4.1.3. Extensionality

Definitional equality of functions in Lean is *intensional*. This means that definitional equality is defined *syntactically*, modulo renaming of bound variables and reduction. To a first approximation, this means that two functions are definitionally equal if they implement the same algorithm, rather than the usual mathematical notion of equality that states that two functions are equal if they map equal elements of the domain to equal elements of the codomain.

Definitional equality is used by the type checker, so it's important that it be predictable. The syntactic character of intensional equality means that the algorithm to check it can be feasibly specified. Checking extensional equality involves proving essentially arbitrary theorems about equality of functions, and there is no clear specification for an algorithm to check it. This makes extensional equality a poor choice for a type checker. Function extensionality is instead made available as a reasoning principle that can be invoked when proving the proposition that two functions are equal.

In addition to reduction and renaming of bound variables, definitional equality does support one limited form of extensionality, called *η-equivalence*, in which functions are equal to abstractions whose bodies apply them to the argument. Given `f` with type `(x : α) → β x`, `f` is definitionally equal to `fun x => f x`.

When reasoning about functions, the theorem `funext` (i Unlike some intensional type theories, `funext` is a theorem in Lean. It can be proved using quotient types.) or the corresponding tactics `funext` or `ext` can be used to prove that two functions are equal if they map equal inputs to equal outputs.

theorem

```
funext.{u, v} {α : Sort u} {β : α → Sort v} {f g : (x : α) → β x}
  (h : ∀ (x : α), f x = g x) : f = g
```

Function extensionality. If two functions return equal results for all possible arguments, then they are equal.

It is called “extensionality” because it provides a way to prove two objects equal based on the properties of the underlying mathematical functions, rather than based on the syntax used to denote them. Function extensionality is a theorem that can be proved using quotient types.

4.1.4. Totality and Termination

Functions can be defined recursively using `def`. From the perspective of Lean's logic, all functions are *total*, meaning that they map each element of the domain to an element of the codomain in finite time. Some programming language communities use the term *total* in a different sense, where functions are considered total if they do not crash due to unhandled cases but non-termination is ignored. The values of total functions are defined for all type-correct arguments, and they cannot fail to terminate or crash due to a missing case in a pattern match.

While the logical model of Lean considers all functions to be total, Lean is also a practical programming language that provides certain "escape hatches". Functions that have not been proven to terminate can still be used in Lean's logic as long as their codomain is proven nonempty. These functions are treated as uninterpreted functions by Lean's logic, and their computational behavior is ignored. In compiled code, these functions are treated just like any others. Other functions may be marked unsafe; these functions are not available to Lean's logic at all. The section on partial and unsafe function definitions contains more detail on programming with recursive functions.

Similarly, operations that should fail at runtime in compiled code, such as out-of-bounds access to an array, can only be used when the resulting type is known to be inhabited. These operations result in an arbitrarily chosen inhabitant of the type in Lean's logic (specifically, the one specified in the type's `Inhabited` instance).

▼ *Example: Panic*

The function `thirdChar` extracts the third element of an array, or panics if the array has two or fewer elements:

```
def thirdChar (xs : Array Char) : Char := xs[2]!
```

The (nonexistent) third elements of `#['!']` and `#['-', 'x']` are equal, because they result in the same arbitrarily-chosen character:

```
example : thirdChar #['!'] = thirdChar #['-', 'x'] := rfl
```

Indeed, both are equal to `'A'`, which happens to be the default fallback for `Char`:

```
example : thirdChar #['!'] = 'A' := rfl
example : thirdChar #['-', 'x'] = 'A' := rfl
```

4.1.5. API Reference

The `Function` namespace contains general-purpose helpers for working with functions.

def

```
Function.comp.{u, v, w} {α : Sort u} {β : Sort v} {δ : Sort w}
  (f : β → δ) (g : α → β) : α → δ
```

Function composition, usually written with the infix operator `∘`. A new function is created from two existing functions, where one function's output is used as input to the other.

Examples:

- `Function.comp List.reverse (List.drop 2) [3, 2, 4, 1] = [1, 4]`
- `(List.reverse ∘ List.drop 2) [3, 2, 4, 1] = [1, 4]`

Conventions for notations in identifiers:

- The recommended spelling of `∘` in identifiers is `comp`.

def

```
Function.const.{u, v} {α : Sort u} (β : Sort v) (a : α) : β → α
```

The constant function that ignores its argument.

If `a : α`, then `Function.const β a : β → α` is the "constant function with value `a`". For all arguments `b : β`, `Function.const β a b = a`.

Examples:

- `Function.const Bool 10 true = 10`
- `Function.const Bool 10 false = 10`
- `Function.const String 10 "any string" = 10`

def

```
Function.curry.{u_1, u_2, u_3} {α : Type u_1} {β : Type u_2}
  {φ : Sort u_3} : (α × β → φ) → α → β → φ
```

Transforms a function from pairs into an equivalent two-parameter function.

Examples:

- `Function.curry (fun (x, y) => x + y) 3 5 = 8`
- `Function.curry Prod.swap 3 "five" = ("five", 3)`

def

```
Function.uncurry.{u_1, u_2, u_3} {α : Type u_1} {β : Type u_2}
  {φ : Sort u_3} : (α → β → φ) → α × β → φ
```

Transforms a two-parameter function into an equivalent function from pairs.

Examples:

- `Function.uncurry List.drop (1, ["a", "b", "c"]) = ["b", "c"]`
- `[("orange", 2), ("android", 3) ].map (Function.uncurry String.take) = ["or", "and"]`

4.2. Propositions

Propositions are meaningful statements that admit proof. Nonsensical statements are not propositions, but false statements are. All propositions are classified by `Prop`.

Propositions have the following properties:

Definitional proof irrelevance
: Any two proofs of the same proposition are completely interchangeable.

Run-time irrelevance
: Propositions are erased from compiled code.

Impredicativity
: Propositions may quantify over types from any universe whatsoever.

Restricted Elimination
: With the exception of subsingletons, propositions cannot be eliminated into non-proposition types.

Extensionality
: Any two logically equivalent propositions can be proven to be equal with the `propext` axiom.

axiom

```
propext {a b : Prop} : (a ↔ b) → a = b
```

The axiom of **propositional extensionality**. It asserts that if propositions *a* and *b* are logically equivalent (i.e. we can prove *a* from *b* and vice versa), then *a* and *b* are *equal*, meaning that we can replace *a* with *b* in all contexts.

For simple expressions like `a ∧ c ∨ d → e` we can prove that because all the logical connectives respect logical equivalence, we can replace *a* with *b* in this expression without using `propext`. However, for higher order expressions like `P a` where `P : Prop → Prop` is unknown, or indeed for `a = b` itself, we cannot replace *a* with *b* without an axiom which says exactly this.

This is a relatively uncontroversial axiom, which is intuitionistically valid. It does however block computation when using **`#reduce`** to reduce proofs directly (which is not recommended), meaning that canonicity, the property that all closed terms of type `Nat` normalize to numerals, fails to hold when this (or any) axiom is used:

```
set_option pp.proofs true

def foo : Nat := by
  have : (True → True) ↔ True := ⟨λ _ => trivial, λ _ _ =>
trivial⟩
  have := propext this ▸ (2 : Nat)
  exact this

#reduce foo
-- propext { mp := fun x x => True.intro, mpr := fun x =>
True.intro } ▸ 2

#eval foo -- 2
```

`#eval` can evaluate it to a numeral because the compiler erases casts and does not evaluate proofs, so `propext`, whose return type is a proposition, can never block it.

4.3. Universes

Types are classified by *universes*. (i) Universes are also referred to as *sorts*. Each universe has a *level*, which is a natural number. The `Sort` operator constructs a universe from a given level. If the level of a universe is smaller than that of another, the universe itself is said to be smaller. With the exception of propositions (described later in this chapter), types in a given universe may only quantify over types in smaller universes. `Sort 0` is the type of propositions, while each `Sort (u + 1)` is a type that describes data.

Every universe is an element of the next larger universe, so `Sort 5` includes `Sort 4`. This means that the following examples are accepted:

```
example : Sort 5 := Sort 4
example : Sort 2 := Sort 1
```

On the other hand, `Sort 3` is not an element of `Sort 5`:

```
example : Sort 5 := Sort 3
```

```
type mismatch
  Type 2
```

```
has type
  Type 3 : Type 4
but is expected to have type
  Type 4 : Type 5
```

Similarly, because `Unit` is in `Sort 1`, it is not in `Sort 2`:

```
example : Sort 1 := Unit

example : Sort 2 := Unit
```

```
type mismatch
  Unit
has type
  Type : Type 1
but is expected to have type
  Type 1 : Type 2
```

Because propositions and data are used differently and are governed by different rules, the abbreviations `Type` and `Prop` are provided to make the distinction more convenient. `Type u` is an abbreviation for `Sort (u + 1)`, so `Type 0` is `Sort 1` and `Type 3` is `Sort 4`. `Type 0` can also be abbreviated `Type`, so `Unit : Type` and `Type : Type 1`. `Prop` is an abbreviation for `Sort 0`.

4.3.1. Predicativity

Each universe contains dependent function types, which additionally represent universal quantification and implication. A function type's universe is determined by the universes of its argument and return types. The specific rules depend on whether the return type of the function is a proposition.

Predicates, which are functions that return propositions (that is, where the result of the function is some type in `Prop`) may have argument types in any universe whatsoever, but the function type itself remains in `Prop`. In other words, propositions feature *impredicative* quantification, because propositions can themselves be statements about all propositions (and all other types).

▼ *Example: Impredicativity*

Proof irrelevance can be written as a proposition that quantifies over all propositions:

```
example : Prop := ∀ (P : Prop) (p1 p2 : P), p1 = p2
```

A proposition may also quantify over all types, at any given level:

```
example : Prop := ∀ (α : Type), ∀ (x : α), x = x
example : Prop := ∀ (α : Type 5), ∀ (x : α), x = x
```

For universes at level 1 and higher (that is, the `Type u` hierarchy), quantification is *predicative*. For these universes, the universe of a function type is the least upper bound of the argument and return types' universes.

▼ *Example: Universe levels of function types*

Both of these types are in `Type 2`:

```
example (α : Type 1) (β : Type 2) : Type 2 := α → β
example (α : Type 2) (β : Type 1) : Type 2 := α → β
```

▼ *Example: Predicativity of* `Type`

This example is not accepted, because α's level is greater than 1. In other words, the annotated universe is smaller than the function type's universe:

```
example (α : Type 2) (β : Type 1) : Type 1 := α → β
```

```
type mismatch
  α → β
has type
  Type 2 : Type 3
but is expected to have type
  Type 1 : Type 2
```

Lean's universes are not cumulative; a type in `Type u` is not automatically also in `Type (u + 1)`. Each type inhabits precisely one universe.

▼ *Example: No cumulativity*

This example is not accepted because the annotated universe is larger than the function type's universe:

```
example (α : Type 2) (β : Type 1) : Type 3 := α → β
```

```
type mismatch
  α → β
has type
  Type 2 : Type 3
but is expected to have type
  Type 3 : Type 4
```

4.3.2. Polymorphism

Lean supports *universe polymorphism*, which means that constants defined in the Lean environment can take universe parameters. These parameters can then be instantiated with universe levels when the constant is used. Universe parameters are written in curly braces following a dot after a constant name.

▼ *Example: Universe-polymorphic identity function*

When fully explicit, the identity function takes a universe parameter u. Its signature is:

```
id.{u} {α : Sort u} (x : α) : α
```

Universe variables may additionally occur in universe level expressions, which provide specific universe levels in definitions. When the polymorphic definition is instantiated with concrete levels, these universe level expressions are also evaluated to yield concrete levels.

▼ *Example: Universe level expressions*

In this example, `Codec` is in a universe that is one greater than the universe of the type it contains:

```
structure Codec.{u} : Type (u + 1) where
  type : Type u
  encode : Array UInt32 → type → Array UInt32
  decode : Array UInt32 → Nat → Option (type × Nat)
```

Lean automatically infers most level parameters. In the following example, it is not necessary to annotate the type as `Codec.{0}`, because `Char`'s type is `Type 0`, so u must be `0`:

```
def Codec.char : Codec where
  type := Char
  encode buf ch := buf.push ch.val
  decode buf i := do
    let v ← buf[i]?
    if h : v.isValidChar then
      let ch : Char := ⟨v, h⟩
      return (ch, i + 1)
    else
      failure
```

Universe-polymorphic definitions in fact create a *schematic definition* that can be instantiated at a variety of levels, and different instantiations of universes create incompatible values.

▼ *Example: Universe polymorphism and definitional equality*

This can be seen in the following example, in which T is a gratuitously-universe-polymorphic function that always returns `true`. Because it is marked **opaque**, Lean can't check equality by unfolding the definitions. Both instantiations of T have the parameters and the same type, but their differing universe instantiations make them incompatible.

```
opaque T.{u} (_ : Nat) : Bool :=
  (fun (α : Sort u) => true) PUnit.{u}

set_option pp.universes true

def test.{u, v} : T.{u} 0 = T.{v} 0 := rfl
```

```
type mismatch
  rfl.{?u.46}
has type
  Eq.{?u.46} ?m.48 ?m.48 : Prop
but is expected to have type
  Eq.{1} (T.{u} 0) (T.{v} 0) : Prop
```

Auto-bound implicit arguments are as universe-polymorphic as possible. Defining the identity function as follows:

```
def id' (x : α) := x
```

results in the signature:

```
id'.{u} {α : Sort u} (x : α) : α
```

▼ *Example: Universe monomorphism in auto-bound implicit parameters*

On the other hand, because `Nat` is in universe `Type 0`, this function automatically ends up with a concrete universe level for α, because `m` is applied to both `Nat` and α, so both must have the same type and thus be in the same universe:

```
partial def count [Monad m] (p : α → Bool) (act : m α) : m Nat :=
do
  if p (← act) then
    return 1 + (← count p act)
  else
    return 0
```

4.3.2.1. Level Expressions

Levels that occur in a definition are not restricted to just variables and addition of constants. More complex relationships between universes can be defined using level expressions.

```
Level ::= 0 | 1 | 2 | ...  -- Concrete levels
        | u, v             -- Variables
        | Level + n        -- Addition of constants
        | max Level Level  -- Least upper bound
        | imax Level Level -- Impredicative LUB
```

Given an assignment of level variables to concrete numbers, evaluating these expressions follows the usual rules of arithmetic. The `imax` operation is defined as follows:

$$\mathbf{imax}\ u\ v = \begin{cases} 0 & \text{when } v = 0 \\ \mathbf{max}\ u\ v & \text{otherwise} \end{cases}$$

`imax` is used to implement impredicative quantification for `Prop`. In particular, if `A : Sort u` and `B : Sort v`, then `(x : A) → B : Sort (imax u v)`. If `B : Prop`, then the function type is itself a `Prop`; otherwise, the function type's level is the maximum of `u` and `v`.

4.3.2.2. Universe Variable Bindings

Universe-polymorphic definitions bind universe variables. These bindings may be either explicit or implicit. Explicit universe variable binding and instantiation occurs as a suffix to the definition's name. Universe parameters are defined or provided by suffixing the name of a constant with a period (`.`) followed by a comma-separated sequence of universe variables between curly braces.

▼ *Example: Universe-polymorphic map*

The following declaration of `map` declares two universe parameters (`u` and `v`) and instantiates the polymorphic `List` with each in turn:

```
def map.{u, v} {α : Type u} {β : Type v}
    (f : α → β) :
    List.{u} α → List.{v} β
  | [] => []
  | x :: xs => f x :: map f xs
```

Just as Lean automatically instantiates implicit parameters, it also automatically instantiates universe parameters. When automatic implicit parameter insertion is enabled (i.e. the `autoImplicit` option is set to `true`, which is the default), it is not necessary to explicitly bind universe variables; they are inserted automatically. When it is set to `false`, then they must be added explicitly or declared using the `universe` command.

▼ *Example: Automatic Implicit Parameters and Universe Polymorphism*

When `autoImplicit` is `true` (which is the default setting), this definition is accepted even though it does not bind its universe parameters:

```
set_option autoImplicit true
def map {α : Type u} {β : Type v} (f : α → β) : List α → List β
  | [] => []
  | x :: xs => f x :: map f xs
```

When `autoImplicit` is `false`, the definition is rejected because u and v are not in scope:

```
set_option autoImplicit false
def map {α : Type u} {β : Type v} (f : α → β) : List α → List β
  | [] => []
  | x :: xs => f x :: map f xs
```

```
unknown universe level 'u'
unknown universe level 'v'
```

In addition to using `autoImplicit`, particular identifiers can be declared as universe variables in a particular section scope using the `universe` command.

syntax

Universe Parameter Declarations

```
command ::= ...
    | universe ident ident*
```

Declares one or more universe variables for the extent of the current scope.

Just as the `variable` command causes a particular identifier to be treated as a parameter with a particular type, the `universe` command causes the subsequent identifiers to be implicitly quantified as as universe parameters in declarations that mention them, even if the option `autoImplicit` is `false`.

▼ *Example: The `universe` command when `autoImplicit` is `false`*

```
set_option autoImplicit false
universe u
def id₃ (α : Type u) (a : α) := a
```

Because the automatic implicit parameter feature only insert parameters that are used in the declaration's header, universe variables that occur only on the right-hand side of a definition are not inserted as arguments unless they have been declared with `universe` even when `autoImplicit` is `true`.

▼ *Example: Automatic universe parameters and the `universe` command*

This definition with an explicit universe parameter is accepted:

```
def L.{u} := List (Type u)
```

Even with automatic implicit parameters, this definition is rejected, because u is not mentioned in the header, which precedes the `:=`:

```
set_option autoImplicit true
def L := List (Type u)
```

```
unknown universe level 'u'
```

With a universe declaration, u is accepted as a parameter even on the right-hand side:

```
universe u
def L := List (Type u)
```

The resulting definition of L is universe-polymorphic, with u inserted as a universe parameter.

Declarations in the scope of a `universe` command are not made polymorphic if the universe variables do not occur in them or in other automatically-inserted arguments.

```
universe u
def L := List (Type 0)
#check L
```

4.3.2.3. Universe Lifting

When a type's universe is smaller than the one expected in some context, *universe lifting* operators can bridge the gap. These are wrappers around terms of a given type that are in larger universes than the wrapped type. There are two lifting operators:

- `PLift` can lift any type, including propositions, by one level. It can be used to include proofs in data structures such as lists.
- `ULift` can lift any non-proposition type by any number of levels.

structure

```
PLift.{u} (α : Sort u) : Type u
```

Lifts a proposition or type to a higher universe level.

`PLift` α wraps a proof or value of type α. The resulting type is in the next largest universe after that of α. In particular, propositions become data.

The related type `ULift` can be used to lift a non-proposition type by any number of levels.

Examples:

- `(False : Prop)`
- `(PLift False : Type)`
- `([.up (by trivial), .up (by simp), .up (by decide)] : List (PLift True))`
- `(Nat : Type 0)`
- `(PLift Nat : Type 1)`

Constructor

`PLift.up.{u}`

Wraps a proof or value to increase its type's universe level by 1.

Fields

down : α

Extracts a wrapped proof or value from a universe-lifted proposition or type.

structure

`ULift.{r, s} (α : Type s) : Type (max s r)`

Lifts a type to a higher universe level.

`ULift` α wraps a value of type α. Instead of occupying the same universe as α, which would be the minimal level, it takes a further level parameter and occupies their maximum. The resulting type may occupy any universe that's at least as large as that of α.

The resulting universe of the lifting operator is the first parameter, and may be written explicitly while allowing α's level to be inferred.

The related type `PLift` can be used to lift a proposition or type by one level.

Examples:

- `(Nat : Type 0)`
- `(ULift Nat : Type 0)`
- `(ULift Nat : Type 1)`

- `(ULift Nat : Type 5)`
- `(ULift.{7} (PUnit : Type 3) : Type 7)`

Constructor

`ULift.up.{r, s}`

Wraps a value to increase its type's universe level.

Fields

down : *α*

Extracts a wrapped value from a universe-lifted type.

4.4. Inductive Types

Inductive types are the primary means of introducing new types to Lean. While universes, functions, and quotient types are built-in primitives that could not be added by users, every other type in Lean is either an inductive type or defined in terms of universes, functions, and inductive types. Inductive types are specified by their *type constructors* and their *constructors*; their other properties are derived from these. Each inductive type has a single type constructor, which may take both universe parameters and ordinary parameters. Inductive types may have any number of constructors; these constructors introduce new values whose types are headed by the inductive type's type constructor.

Based on the type constructor and the constructors for an inductive type, Lean derives a *recursor*. Logically, recursors represent induction principles or elimination rules; computationally, they represent primitive recursive computations. The termination of recursive functions is justified by translating them into uses of the recursors, so Lean's kernel only needs to perform type checking of recursor applications, rather than including a separate termination analysis. Lean additionally produces a number of helper constructions based on the recursor, (i) The term *recursor* is always used, even for non-recursive types. which are used elsewhere in the system.

Structures are a special case of inductive types that have exactly one constructor. When a structure is declared, Lean generates helpers that enable additional language features to be used with the new structure.

This section describes the specific details of the syntax used to specify both inductive types and structures, the new constants and definitions in the environment that result from inductive type declarations, and the run-time representation of inductive types' values in compiled code.

4.4.1. Inductive Type Declarations

syntax

Inductive Type Declarations

```
command ::= ...
    | declModifiers
      inductive declId optDeclSig where
        (| declModifiers ident optDeclSig)*
      (deriving ident,*)?
```

Declares a new inductive type. The meaning of the *declModifiers* is as described in the section on declaration modifiers.

After declaring an inductive type, its type constructor, constructors, and recursor are present in the environment. New inductive types extend Lean's core logic—they are not encoded or represented by some other already-present data. Inductive type declarations must satisfy a number of well-formedness requirements to ensure that the logic remains consistent.

The first line of the declaration, from `inductive` to `where`, specifies the new type constructor's name and type. If a type signature for the type constructor is provided, then its result type must be a universe, but the parameters do not need to be types. If no signature is provided, then Lean will attempt to infer a universe that's just big enough to contain the resulting type. In some situations, this process may fail to find a minimal universe or fail to find one at all, necessitating an annotation.

The constructor specifications follow `where`. Constructors are not mandatory, as constructorless inductive types such as `False` and `Empty` are perfectly sensible. Each constructor specification begins with a vertical bar (`'|'`, Unicode `'VERTICAL BAR' (U+007c)`), declaration modifiers, and a name. The name is a raw identifier. A declaration signature follows the name. The signature may specify any parameters, modulo the well-formedness requirements for inductive type declarations, but the return type in the signature must be a saturated application of the type constructor of the inductive type being specified. If no signature is provided, then the constructor's type is inferred by inserting sufficient implicit parameters to construct a well-formed return type.

The new inductive type's name is defined in the current namespace. Each constructor's name is in the inductive type's namespace.

4.4.1.1. Parameters and Indices

Type constructors may take two kinds of arguments: *parameters* and *indices*. Parameters must be used consistently in the entire definition; all occurrences of the type constructor in each constructor in the declaration must take precisely the same argument. Indices may vary among the occurrences of the type constructor. All parameters must precede all indices in the type constructor's signature.

Parameters that occur prior to the colon (`':'`) in the type constructor's signature are considered parameters to the entire inductive type declaration. They are always parameters that must be uniform throughout the type's definition. Generally speaking, parameters that occur after the colon are indices that may vary throughout the definition of the type. However, if the option `inductive.autoPromoteIndices` is `true`, then syntactic indices that could have been

parameters are made into parameters. An index could have been a parameter if all of its type dependencies are themselves parameters and it is used uniformly as an uninstantiated variable in all occurrences of the inductive type's type constructor in all constructors.

option

`inductive.autoPromoteIndices`

Default value: `true`

Promote indices to parameters in inductive types whenever possible.

Indices can be seen as defining a *family* of types. Each choice of indices selects a type from the family, which has its own set of available constructors. Type constructors with indices are said to specify *indexed families* of types.

4.4.1.2. Example Inductive Types

▼ *Example: A constructorless type*

`Vacant` is an empty inductive type, equivalent to Lean's `Empty` type:

```
inductive Vacant : Type where
```

Empty inductive types are not useless; they can be used to indicate unreachable code.

▼ *Example: A constructorless proposition*

`No` is a false proposition, equivalent to Lean's `False`:

```
inductive No : Prop where
```

▼ *Example: A unit type*

`Solo` is equivalent to Lean's `Unit` type:

```
inductive Solo where
  | solo
```

It is an example of an inductive type in which the signatures have been omitted for both the type constructor and the constructor. Lean assigns `Solo` to `Type`:

```
#check Solo
```

```
Solo : Type
```

The constructor is named `Solo.solo`, because constructor names are the type constructor's namespace. Because `Solo` expects no arguments, the signature inferred for `Solo.solo` is:

```
#check Solo.solo
```

```
Solo.solo : Solo
```

▼ *Example: A true proposition*

`Yes` is equivalent to Lean's `True` proposition:

```
inductive Yes : Prop where
  | intro
```

Unlike One, the new inductive type `Yes` is specified to be in the `Prop` universe.

```
#check Yes
```

```
Yes : Prop
```

The signature inferred for `Yes.intro` is:

```
#check Yes.intro
```

```
Yes.intro : Yes
```

▼ *Example: A type with parameter and index*

An `EvenOddList α b` is a list where α is the type of the data stored in the list and b is `true` when there are an even number of entries:

```
inductive EvenOddList (α : Type u) : Bool → Type u where
  | nil : EvenOddList α true
  | cons : α → EvenOddList α isEven → EvenOddList α (not isEven)
```

This example is well typed because there are two entries in the list:

```
example : EvenOddList String true :=
  .cons "a" (.cons "b" .nil)
```

This example is not well typed because there are three entries in the list:

```
example : EvenOddList String true :=
  .cons "a" (.cons "b" (.cons "c" .nil))
```

```
type mismatch
  EvenOddList.cons "a" (EvenOddList.cons "b"
(EvenOddList.cons "c" EvenOddList.nil))
has type
```

```
  EvenOddList String !!!true : Type
but is expected to have type
  EvenOddList String true : Type
```

In this declaration, α is a parameter, because it is used consistently in all occurrences of `EvenOddList`. b is an index, because different `Bool` values are used for it at different occurrences.

▼ *Example: Parameters before and after the colon*

In this example, both parameters are specified before the colon in `Either`'s signature.

```
inductive Either (α : Type u) (β : Type v) : Type (max u v) where
  | left : α → Either α β
  | right : β → Either α β
```

In this version, there are two types named α that might not be identical:

```
inductive Either' (α : Type u) (β : Type v) : Type (max u v)
where
  | left : {α : Type u} → {β : Type v} → α → Either' α β
  | right : β → Either' α β
```

```
Mismatched inductive type parameter in
  Either' α β
The provided argument
  α
is not definitionally equal to the expected parameter
  α✝

Note: The value of parameter 'α✝' must be fixed throughout
the inductive declaration. Consider making this parameter an
index if it must vary.
```

Placing the parameters after the colon results in parameters that can be instantiated by the constructors:

```
inductive Either'' : Type u → Type v → Type (max u v + 1) where
  | left : {α : Type u} → {β : Type v} → α → Either'' α β
  | right : β → Either'' α β
```

A larger universe is required for this type because constructor parameters must be in universes that are smaller than the inductive type's universe. `Either''.right`'s type parameter is discovered via Lean's ordinary rules for automatic implicit parameters.

4.4.1.3. Anonymous Constructor Syntax

If an inductive type has just one constructor, then this constructor is eligible for *anonymous constructor syntax*. Instead of writing the constructor's name applied to its arguments, the explicit arguments can be enclosed in angle brackets (`'⟨'` and `'⟩'`, Unicode `MATHEMATICAL LEFT ANGLE BRACKET (U+0x27e8)` and `MATHEMATICAL RIGHT ANGLE BRACKET (U+0x27e9)`) and separated with commas. This works in both pattern and expression contexts. Providing arguments by name or converting all implicit parameters to explicit parameters with `@` requires using the ordinary constructor syntax.

syntax

Anonymous Constructors

Constructors can be invoked anonymously by enclosing their explicit arguments in angle brackets, separated by commas.

```
term ::= ...
    | ⟨ term,* ⟩
```

▼ *Example: Anonymous constructors*

The type `AtLeastOne α` is similar to `List α`, except there's always at least one element present:

```
inductive AtLeastOne (α : Type u) : Type u where
  | mk : α → Option (AtLeastOne α) → AtLeastOne α
```

Anonymous constructor syntax can be used to construct them:

```
def oneTwoThree : AtLeastOne Nat :=
  ⟨1, some ⟨2, some ⟨3, none⟩⟩⟩
```

and to match against them:

```
def AtLeastOne.head : AtLeastOne α → α
  | ⟨x, _⟩ => x
```

Equivalently, traditional constructor syntax could have been used:

```
def oneTwoThree' : AtLeastOne Nat :=
  .mk 1 (some (.mk 2 (some (.mk 3 none))))

def AtLeastOne.head' : AtLeastOne α → α
  | .mk x _ => x
```

4.4.1.4. Deriving Instances

The optional `deriving` clause of an inductive type declaration can be used to derive instances of type classes. Please refer to the section on instance deriving for more information.

4.4.2. Structure Declarations

syntax

Structure Declarations

```
command ::= ...
    | declModifiers
      structure declId bracketedBinder* (: term)?
        (extends (ident : )?term,*)?
        where
        (declModifiers ident ::)?
        structFields
      (deriving ident,*)?
```

Declares a new structure type.

Structures are inductive types that have only a single constructor and no indices. In exchange for these restrictions, Lean generates code for structures that offers a number of conveniences: projection functions are generated for each field, an additional constructor syntax based on field names rather than positional arguments is available, a similar syntax may be used to replace the values of certain named fields, and structures may extend other structures. Just like other inductive types, structures may be recursive; they are subject to the same restrictions regarding strict positivity. Structures do not add any expressive power to Lean; all of their features are implemented in terms of code generation.

4.4.2.1. Structure Parameters

Just like ordinary inductive type declarations, the header of the structure declaration contains a signature that may specify both parameters and a resulting universe. Structures may not define indexed families.

4.4.2.2. Fields

Each field of a structure declaration corresponds to a parameter of the constructor. Auto-implicit arguments are inserted in each field separately, even if their names coincide, and the fields become constructor parameters that quantify over types.

▼ *Example: Auto-implicit parameters in structure fields*

The structure `MyStructure` contains a field whose type is an auto-implicit parameter:

```
structure MyStructure where
  field1 : α
  field2 : α
```

The type constructor `MyStructure` takes two universe parameters:

```
MyStructure.{u, v} : Type (max u v)
```

The resulting type is in `Type` rather than `Sort` because the constructor fields quantify over types in `Sort`. In particular, both fields in its constructor `MyStructure.mk` take an implicit type parameter:

```
MyStructure.mk.{u, v}
  (field1 : {α : Sort u} → α)
  (field2 : {α : Sort v} → α)
  : MyStructure.{u,v}
```

For each field, a projection function is generated that extracts the field's value from the underlying type's constructor. This function is in the structure's name's namespace. Structure field projections are handled specially by the elaborator (as described in the section on structure inheritance), which performs extra steps beyond looking up a namespace. When field types depend on prior fields, the types of the dependent projection functions are written in terms of earlier projections, rather than explicit pattern matching.

▼ *Example: Dependent projection types*

The structure `ArraySized` contains a field whose type depends on both a structure parameter and an earlier field:

```
structure ArraySized (α : Type u) (length : Nat)  where
  array : Array α
  size_eq_length : array.size = length
```

The signature of the projection function `size_eq_length` takes the structure type's parameter as an implicit parameter and refers to the earlier field using the corresponding projection:

```
ArraySized.size_eq_length.{u}
  {α : Type u} {length : Nat}
  (self : ArraySized α length)
  : self.array.size = length
```

Structure fields may have default values, specified with `:=`. These values are used if no explicit value is provided.

▼ *Example: Default values*

An adjacency list representation of a graph can be represented as an array of lists of `Nat`. The size of the array indicates the number of vertices, and the outgoing edges from each vertex are stored in the array at the vertex's index. Because the default value `#[]` is provided for the field `adjacency`, the empty graph `Graph.empty` can be constructed without providing any field values.

```
structure Graph where
  adjacency : Array (List Nat) := #[]

def Graph.empty : Graph := {}
```

Structure fields may additionally be accessed via their index, using dot notation. Fields are numbered beginning with `1`.

4.4.2.3. Structure Constructors

Structure constructors may be explicitly named by providing the constructor name and `::` prior to the fields. If no name is explicitly provided, then the constructor is named `mk` in the structure type's namespace. Declaration modifiers may additionally be provided along with an explicit constructor name.

▼ *Example: Non-default constructor name*

The structure `Palindrome` contains a string and a proof that the string is the same when reversed:

```
structure Palindrome where
  ofString ::
  text : String
  is_palindrome : text.data.reverse = text.data
```

Its constructor is named `Palindrome.ofString`, rather than `Palindrome.mk`.

▼ *Example: Modifiers on structure constructor*

The structure `NatStringBimap` maintains a finite bijection between natural numbers and strings. It consists of a pair of maps, such that the keys each occur as values exactly once in the other map. Because the constructor is private, code outside the defining module can't construct new instances and must use the provided API, which maintains the invariants of the type. Additionally, providing the default constructor name explicitly is an opportunity to attach a documentation comment to the constructor.

```
structure NatStringBimap where
  /--
  Build a finite bijection between some
  natural numbers and strings
  -/
  private mk ::
  natToString : Std.HashMap Nat String
  stringToNat : Std.HashMap String Nat

def NatStringBimap.empty : NatStringBimap := ⟨{}, {}⟩

def NatStringBimap.insert
    (nat : Nat) (string : String)
    (map : NatStringBimap) :
    Option NatStringBimap :=
  if map.natToString.contains nat ||
      map.stringToNat.contains string then
    none
  else
    some (NatStringBimap.mk (map.natToString.insert nat string)
(map.stringToNat.insert string nat))
```

Because structures are represented by single-constructor inductive types, their constructors can be invoked or matched against using anonymous constructor syntax. Additionally, structures may be constructed or matched against using *structure instance* notation, which includes the names of the fields together with values for them.

syntax

Structure Instances

```
term ::= ...
    | { structInstField,*
        (: term)? }
```

Constructs a value of a constructor type given values for named fields. Field specifiers may take two forms:

```
structInstField ::= ...
    | structInstLVal := term

structInstField ::= ...
    | ident
```

A `structInstLVal` is a field name (an identifier), a field index (a natural number), or a term in square brackets, followed by a sequence of zero or more subfields. Subfields are either a field name or index preceded by a dot, or a term in square brackets.

This syntax is elaborated to applications of structure constructors. The values provided for fields are by name, and they may be provided in any order. The values provided for subfields are used to initialize fields of constructors of structures that are themselves found in fields. Terms in square brackets are not allowed when constructing a structure; they are used in structure updates.

Field specifiers that do not contain `:=` are field abbreviations. In this context, the identifier `f` is an abbreviation for `f := f`; that is, the value of `f` in the current scope is used to initialize the field `f`.

Every field that does not have a default value must be provided. If a tactic is specified as the default argument, then it is run at elaboration time to construct the argument's value.

In a pattern context, field names are mapped to patterns that match the corresponding projection, and field abbreviations bind a pattern variable that is the field's name. Default arguments are still present in patterns; if a pattern does not specify a value for a field with a default value, then the pattern only matches the default.

The optional type annotation allows the structure type to be specified in contexts where it is not otherwise determined.

▼ *Example: Patterns and default values*

The structure `AugmentedIntList` contains a list together with some extra information, which is empty if omitted:

```
structure AugmentedIntList where
  list : List Int
  augmentation : String := ""
```

When testing whether the list is empty, the function `isEmpty` is also testing whether the `augmentation` field is empty, because the omitted field's default value is also used in pattern contexts:

```
def AugmentedIntList.isEmpty : AugmentedIntList → Bool
  | {list := []} => true
  | _ => false

#eval {list := [], augmentation := "extra" :
AugmentedIntList}.isEmpty
```

```
false
```

syntax

Structure Updates

```
term ::= ...
    | {term with
        structInstField,*
        (: term)?}
```

Updates a value of a constructor type. The term that precedes the `with` clause is expected to have a structure type; it is the value that is being updated. A new instance of the structure is created in which every field not specified is copied from the value that is being updated, and the specified fields are replaced with their new values. When updating a structure, array values may also be replaced by including the index to be updated in square brackets. This updating does not require that the index expression be in bounds for the array, and out-of-bounds updates are discarded.

▼ *Example: Updating arrays*

Updating structures may use array indices as well as projection names. Updates at indices that are out of bounds are ignored:

```
structure AugmentedIntArray where
  array : Array Int
  augmentation : String := ""
deriving Repr

def one : AugmentedIntArray := {array := #[1]}
def two : AugmentedIntArray := {one with array := #[1, 2]}
def two' : AugmentedIntArray := {two with array[0] := 2}
def two'' : AugmentedIntArray := {two with array[99] := 3}
#eval (one, two, two', two'')
```

```
({ array := #[1], augmentation := "" },
 { array := #[1, 2], augmentation := "" },
 { array := #[2, 2], augmentation := "" },
 { array := #[1, 2], augmentation := "" })
```

Values of structure types may also be declared using `where`, followed by definitions for each field. This may only be used as part of a definition, not in an expression context.

▼ *Example: `where` for structures*

The product type in Lean is a structure named `Prod`. Products can be defined using their projections:

```
def location : Float × Float where
  fst := 22.807
```

```
  snd := -13.923
```

4.4.2.4. Structure Inheritance

Structures may be declared as extending other structures using the optional `extends` clause. The resulting structure type has all of the fields of all of the parent structure types. If the parent structure types have overlapping field names, then all overlapping field names must have the same type.

The resulting structure has a *field resolution order* that affects the values of fields. When possible, this resolution order is the C3 linearization of the structure's parents. Essentially, the field resolution order should be a total ordering of the entire set of parents such that every `extends` list is in order. When there is no C3 linearization, a heuristic is used to find an order nonetheless. Every structure type is first in its own field resolution order.

The field resolution order is used to compute the default values of optional fields. When the value of a field is not specified, the first default value defined in the resolution order is used. References to fields in the default value use the field resolution order as well; this means that child structures that override default fields of parent constructors may also change the computed default values of parent fields. Because the child structure is the first element of its own resolution order, default values in the child structure take precedence over default values from the parent structures.

When the new structure extends existing structures, the new structure's constructor takes the existing structure's information as additional arguments. Typically, this is in the form of a constructor parameter for each parent structure type. This parent value contains all of the parent's fields. If the parents' fields overlap, however, then the subset of non-overlapping fields from one or more of the parents is included instead of an entire value of the parent structure to prevent duplicating field information.

There is no subtyping relation between a parent structure type and its children. Even if structure B extends structure A, a function expecting an A will not accept a B. However, conversion functions are generated that convert a structure into each of its parents. These conversion functions are called *parent projections*. Parent projections are in the child structure's namespace, and their name is the parent structure's name preceded by `to`.

▼ *Example: Structure type inheritance with overlapping fields*

In this example, a `Textbook` is a `Book` that is also an `AcademicWork`:

```
structure Book where
  title : String
  author : String

structure AcademicWork where
  author : String
  discipline : String
```

```
structure Textbook extends Book, AcademicWork

#check Textbook.toBook
```

Because the field `author` occurs in both `Book` and `AcademicWork`, the constructor `Textbook.mk` does not take both parents as arguments. Its signature is:

```
Textbook.mk (toBook : Book) (discipline : String) : Textbook
```

The conversion functions are:

```
Textbook.toBook (self : Textbook) : Book

Textbook.toAcademicWork (self : Textbook) : AcademicWork
```

The latter combines the `author` field of the included `Book` with the unbundled `Discipline` field, and is equivalent to:

```
def toAcademicWork (self : Textbook) : AcademicWork :=
  let .mk book discipline := self
  let .mk _title author := book
  .mk author discipline
```

The resulting structure's projections can be used as if its fields are simply the union of the parents' fields. The Lean elaborator automatically generates an appropriate projection when fields are used. Likewise, the field-based initialization and structure update notations hide the details of the encoding of inheritance. The encoding is, however, visible when using the constructor's name, when using anonymous constructor syntax, or when referring to fields by their index rather than their name.

▼ *Example: Field Indices and Structure Inheritance*

```
structure Pair (α : Type u) where
  fst : α
  snd : α
deriving Repr

structure Triple (α : Type u) extends Pair α where
  thd : α
deriving Repr

def coords : Triple Nat := {fst := 17, snd := 2, thd := 95}
```

Evaluating the first field index of `coords` yields the underlying `Pair`, rather than the contents of the field `fst`:

```
#eval coords.1
```

```
{ fst := 17, snd := 2 }
```

The elaborator translates `coords.fst` into `coords.toPair.fst`.

▼ *Example: No structure subtyping*

Given these definitions of even numbers, even prime numbers, and a concrete even prime:

```
structure EvenNumber where
  val : Nat
  isEven : 2 ∣ val := by decide

structure EvenPrime extends EvenNumber where
  notOne : val ≠ 1 := by decide
  isPrime : ∀ n, n ≤ val → n ∣ val  → n = 1 ∨ n = val

def two : EvenPrime where
  val := 2
  isPrime := by
    intros
    repeat' (cases ‹Nat.le _ _›)
    all_goals omega

def printEven (num : EvenNumber) : IO Unit :=
  IO.print num.val
```

it is a type error to apply `printEven` directly to `two`:

```
#check printEven two
```

```
Application type mismatch: In the application
  printEven two
the argument
  two
has type
  EvenPrime : Type
but is expected to have type
  EvenNumber : Type
```

because values of type `EvenPrime` are not also values of type `EvenNumber`.

The `#print` command displays the most important information about structure types, including the parent projections, all the fields with their default values, the constructor, and the field resolution order. When working with deep hierarchies that contain inheritance diamonds, this information can be very useful.

▼ *Example: `#print` and Structure Types*

This collection of structure types models a variety of bicycles, both electric and non-electric and both ordinary-sized and large family bicycles. The final structure type, `ElectricFamilyBike`, contains a diamond in its inheritance graph, because both `FamilyBike` and `ElectricBike` extend `Bicycle`.

```
structure Vehicle where
  wheels : Nat

structure Bicycle extends Vehicle where
  wheels := 2

structure ElectricVehicle extends Vehicle where
  batteries : Nat := 1

structure FamilyBike extends Bicycle where
  wheels := 3

structure ElectricBike extends Bicycle, ElectricVehicle

structure ElectricFamilyBike
    extends FamilyBike, ElectricBike where
  batteries := 2
```

The `#print` command displays the important information about each structure type:

```
#print ElectricBike
```

```
structure ElectricBike : Type
number of parameters: 0
parents:
  ElectricBike.toBicycle : Bicycle
  ElectricBike.toElectricVehicle : ElectricVehicle
fields:
  Vehicle.wheels : Nat :=
    2
  ElectricVehicle.batteries : Nat :=
    1
constructor:
  ElectricBike.mk (toBicycle : Bicycle) (batteries : Nat) :
ElectricBike
field notation resolution order:
  ElectricBike, Bicycle, ElectricVehicle, Vehicle
```

An `ElectricFamilyBike` has three wheels by default because `FamilyBike` precedes `Bicycle` in its resolution order:

```
#print ElectricFamilyBike
```

```
structure ElectricFamilyBike : Type
number of parameters: 0
parents:
  ElectricFamilyBike.toFamilyBike : FamilyBike
  ElectricFamilyBike.toElectricBike : ElectricBike
fields:
  Vehicle.wheels : Nat :=
    3
  ElectricVehicle.batteries : Nat :=
    2
constructor:
  ElectricFamilyBike.mk (toFamilyBike : FamilyBike)
(batteries : Nat) : ElectricFamilyBike
field notation resolution order:
  ElectricFamilyBike, FamilyBike, ElectricBike, Bicycle,
ElectricVehicle, Vehicle
```

4.4.3. Logical Model

4.4.3.1. Recursors

Every inductive type is equipped with a recursor. The recursor is completely determined by the signatures of the type constructor and the constructors. Recursors have function types, but they are primitive and are not definable using `fun`.

4.4.3.1.1. Recursor Types

The recursor takes the following parameters:

The inductive type's parameters
: Because parameters are consistent, they can be abstracted over the entire recursor.

The *motive*
: The motive determines the type of an application of the recursor. The motive is a function whose arguments are the type's indices and an instance of the type with these indices instantiated. The specific universe for the type that the motive determines depends on the inductive type's universe and the specific constructors—see the section on subsingleton elimination for details.

A *minor premise* for each constructor
: For each constructor, the recursor expects a function that satisfies the motive for an arbitrary application of the constructor. Each minor premise abstracts over all of the constructor's parameters. If the constructor's parameter's type is the inductive type itself, then the minor premise additionally takes a parameter whose type is the motive applied to that parameter's value—this will receive the result of recursively processing the recursive parameter.

The *major premise*, or target
: Finally, the recursor takes an instance of the type as an argument, along with any index values.

The result type of the recursor is the motive applied to these indices and the major premise.

▼ *Example: The recursor for* `Bool`

`Bool`'s recursor `Bool.rec` has the following parameters:

- The motive computes a type in any universe, given a `Bool`.
- There are minor premises for both constructors, in which the motive is satisfied for both `false` and `true`.
- The major premise is some `Bool`.

The return type is the motive applied to the major premise.

```
Bool.rec.{u} {motive : Bool → Sort u}
  (false : motive false)
  (true : motive true)
  (t : Bool) : motive t
```

▼ *Example: The recursor for* `List`

`List`'s recursor `List.rec` has the following parameters:

- The parameter α comes first, because the motive, minor premises, and major premise need to refer to it.
- The motive computes a type in any universe, given a `List α`. There is no connection between the universe levels u and v.
- There are minor premises for both constructors:
  - The motive is satisfied for `List.nil`
  - The motive should be satisfiable for any application of `List.cons`, given that it is satisfiable for the tail. The extra parameter `motive tail` is because `tail`'s type is a recursive occurrence of `List`.
- The major premise is some `List α`.

Once again, the return type is the motive applied to the major premise.

```
List.rec.{u, v} {α : Type v} {motive : List α → Sort u}
  (nil : motive [])
  (cons : (head : α) → (tail : List α) → motive tail →
    motive (head :: tail))
  (t : List α) : motive t
```

▼ *Example: Recursor with parameters and indices*

Given the definition of `EvenOddList`:

```
inductive EvenOddList (α : Type u) : Bool → Type u where
  | nil : EvenOddList α true
  | cons : α → EvenOddList α isEven → EvenOddList α (not isEven)
```

The recursor `EvenOddList.rec` is very similar to that for `List`. The difference comes from the presence of the index:

- The motive now abstracts over any arbitrary choice of index.
- The minor premise for `nil` applies the motive to `nil`'s index value `true`.
- The minor premise `cons` abstracts over the index value used in its recursive occurrence, and instantiates the motive with its negation.
- The major premise additionally abstracts over an arbitrary choice of index.

```
EvenOddList.rec.{u, v} {α : Type v}
  {motive : (isEven : Bool) → EvenOddList α isEven → Sort u}
  (nil : motive true EvenOddList.nil)
  (cons : {isEven : Bool} →
    (head : α) →
    (tail : EvenOddList α isEven) → motive isEven tail →
    motive (!isEven) (EvenOddList.cons head tail)) :
  {isEven : Bool} → (t : EvenOddList α isEven) → motive isEven t
```

When using a predicate (that is, a function that returns a `Prop`) for the motive, recursors express induction. The minor premises for non-recursive constructors are the base cases, and the additional arguments supplied to minor premises for constructors with recursive arguments are the induction hypotheses.

4.4.3.1.1.1. Subsingleton Elimination

Proofs in Lean are computationally irrelevant. In other words, having been provided with **some** proof of a proposition, it should be impossible for a program to check **which** proof it has received. This is reflected in the types of recursors for inductively defined propositions or predicates. For these types, if there's more than one potential proof of the theorem then the motive may only return another `Prop`. If the type is structured such that there's only at most one proof anyway, then the motive may return a type in any universe. A proposition that has at most one inhabitant is called a *subsingleton*. Rather than obligating users to *prove* that there's only one possible proof, a conservative syntactic approximation is used to check whether a proposition is a subsingleton. Propositions that fulfill both of the following requirements are considered to be subsingletons:

- There is at most one constructor.
- Each of the constructor's parameter types is either a `Prop`, a parameter, or an index.

▼ *Example:* `True` *is a subsingleton*

`True` is a subsingleton because it has one constructor, and this constructor has no parameters. Its recursor has the following signature:

```
True.rec.{u} {motive : True → Sort u}
  (intro : motive True.intro)
  (t : True) : motive t
```

▼ *Example:* `False` *is a subsingleton*

`False` is a subsingleton because it has no constructors. Its recursor has the following signature:

```
False.rec.{u} (motive : False → Sort u) (t : False) : motive t
```

Note that the motive is an explicit parameter. This is because it is not mentioned in any further parameters' types, so it could not be solved by unification.

▼ *Example:* `And` *is a subsingleton*

`And` is a subsingleton because it has one constructor, and both of the constructor's parameters' types are propositions. Its recursor has the following signature:

```
And.rec.{u} {a b : Prop} {motive : a ∧ b → Sort u}
  (intro : (left : a) → (right : b) → motive (And.intro left
right))
  (t : a ∧ b) : motive t
```

▼ *Example:* `Or` *is not a subsingleton*

`Or` is not a subsingleton because it has more than one constructor. Its recursor has the following signature:

```
Or.rec {a b : Prop} {motive : a ∨ b → Prop}
  (inl : ∀ (h : a), motive (.inl h))
  (inr : ∀ (h : b), motive (.inr h))
  (t : a ∨ b) : motive t
```

The motive's type indicates that `Or.rec` can only be used to produce proofs. A proof of a disjunction can be used to prove something else, but there's no way for a program to inspect *which* of the two disjuncts was true and used for the proof.

▼ *Example:* Eq *is a subsingleton*

Eq is a subsingleton because it has just one constructor, Eq.refl. This constructor instantiates Eq's index with a parameter value, so all arguments are parameters:

```
Eq.refl.{u} {α : Sort u} (x : α) : Eq x x
```

Its recursor has the following signature:

```
Eq.rec.{u, v} {α : Sort v} {x : α}
  {motive : (y : α) → x = y → Sort u}
  (refl : motive x (.refl x))
  {y : α} (t : x = y) : motive y t
```

This means that proofs of equality can be used to rewrite the types of non-propositions.

4.4.3.1.2. Reduction

In addition to adding new constants to the logic, inductive type declarations also add new reduction rules. These rules govern the interaction between recursors and constructors; specifically recursors that have constructors as their major premise. This form of reduction is called *ι-reduction* (iota reduction).

When the recursor's major premise is a constructor with no recursive parameters, the recursor application reduces to an application of the constructor's minor premise to the constructor's arguments. If there are recursive parameters, then these arguments to the minor premise are found by applying the recursor to the recursive occurrence.

4.4.3.2. Well-Formedness Requirements

Inductive type declarations are subject to a number of well-formedness requirements. These requirements ensure that Lean remains consistent as a logic when it is extended with the inductive type's new rules. They are conservative: there exist potential inductive types that do not undermine consistency, but that these requirements nonetheless reject.

4.4.3.2.1. Universe Levels

Type constructors of inductive types must either inhabit a universe or a function type whose return type is a universe. Each constructor must inhabit a function type that returns a saturated application of the inductive type. If the inductive type's universe is `Prop`, then there are no further restrictions on universes, because `Prop` is impredicative. If the universe is not `Prop`, then the following must hold for each parameter to the constructor:

- If the constructor's parameter is a parameter (in the sense of parameters vs indices) of the inductive type, then this parameter's type may be no larger than the type constructor's universe.
- All other constructor parameters must be smaller than the type constructor's universe.

▼ *Example: Universes, constructors, and parameters*

`Either` is in the greater of its arguments' universes, because both are parameters to the inductive type:

```
inductive Either (α : Type u) (β : Type v) : Type (max u v) where
  | inl : α → Either α β
  | inr : β → Either α β
```

CanRepr is in a larger universe than the constructor parameter α, because α is not one of the inductive type's parameters:

```
inductive CanRepr : Type (u + 1) where
  | mk : (α : Type u) → [Repr α] → CanRepr
```

Constructorless inductive types may be in universes smaller than their parameters:

```
inductive Spurious (α : Type 5) : Type 0 where
```

It would, however, be impossible to add a constructor to Spurious without changing its levels.

4.4.3.2.2. Strict Positivity

All occurrences of the type being defined in the types of the parameters of the constructors must be in *strictly positive* positions. A position is strictly positive if it is not in a function's argument type (no matter how many function types are nested around it) and it is not an argument of any expression other than type constructors of inductive types. This restriction rules out unsound inductive type definitions, at the cost of also ruling out some unproblematic ones.

▼ *Example: Non-strictly-positive inductive types*

The type Bad would make Lean inconsistent if it were not rejected:

```
inductive Bad where
  | bad : (Bad → Bad) → Bad
```

```
(kernel) arg #1 of 'Bad.bad' has a non positive occurrence of
the datatypes being declared
```

This is because it would be possible to write a circular argument that proves False under the assumption Bad. `Bad.bad` is rejected because the constructor's parameter has type Bad → Bad, which is a function type in which Bad occurs as an argument type.

This declaration of a fixed point operator is rejected, because `Fix` occurs as an argument to `f`:

```
inductive Fix (f : Type u → Type u) where
  | fix : f (Fix f) → Fix f
```

```
(kernel) arg #2 of 'Fix.fix' contains a non valid occurrence
of the datatypes being declared
```

`Fix.fix` is rejected because `f` is not a type constructor of an inductive type, but `Fix` itself occurs as an argument to it. In this case, `Fix` is also sufficient to construct a type equivalent to `Bad`:

```
def Bad : Type := Fix fun t => t → t
```

4.4.3.2.3. Prop vs Type

Lean rejects universe-polymorphic types that could not, in practice, be used polymorphically. This could arise if certain instantiations of the universe parameters would cause the type itself to be a `Prop`. If this type is not a subsingleton, then is recursor can only target propositions (that is, the motive must return a `Prop`). These types only really make sense as `Prop`s themselves, so the universe polymorphism is probably a mistake. Because they are largely useless, Lean's inductive type elaborator has not been designed to support these types.

When such universe-polymorphic inductive types are indeed subsingletons, it can make sense to define them. Lean's standard library defines `PUnit` and `PEmpty`. To define a subsingleton that can inhabit `Prop` or a `Type`, set the option `bootstrap.inductiveCheckResultingUniverse` to `false`.

option

`bootstrap.inductiveCheckResultingUniverse`

Default value: `true`

by default the `inductive`/`structure` commands report an error if the resulting universe is not zero, but may be zero for some universe parameters. Reason: unless this type is a subsingleton, it is hardly what the user wants since it can only eliminate into `Prop`. In the `Init` package, we define subsingletons, and we use this option to disable the check. This option may be deleted in the future after we improve the validator

▼ *Example: Overly-universe-polymorphic `Bool`*

Defining a version of `Bool` that can be in any universe is not allowed:

```
inductive PBool : Sort u where
  | true
  | false
```

```
invalid universe polymorphic resulting type, the resulting
universe is not 'Prop', but it may be 'Prop' for some
parameter values:
  Sort u
```

```
Possible solution: use levels of the form 'max 1 _' or '_ +
1' to ensure the universe is of the form 'Type _'.
```

4.4.3.3. Constructions for Termination Checking

In addition to the type constructor, constructors, and recursors that Lean's core type theory prescribes for inductive types, Lean constructs a number of useful helpers. First, the equation compiler (which translates recursive functions with pattern matching in to applications of recursors) makes use of these additional constructs:

- `recOn` is a version of the recursor in which the major premise is prior to the minor premise for each constructor.
- `casesOn` is a version of the recursor in which the major premise is prior to the minor premise for each constructor, and recursive arguments do not yield induction hypotheses. It expresses case analysis rather than primitive recursion.
- `below` computes a type that, for some motive, expresses that *all* inhabitants of the inductive type that are subtrees of the major premise satisfy the motive. It transforms a motive for induction or primitive recursion into a motive for strong recursion or strong induction.
- `brecOn` is a version of the recursor in which `below` is used to provide access to all subtrees, rather than just immediate recursive parameters. It represents strong induction.
- `noConfusion` is a general statement from which injectivity and disjointness of constructors can be derived.
- `noConfusionType` is the motive used for `noConfusion` that determines what the consequences of two constructors being equal would be. For separate constructors, this is `False`; if both constructors are the same, then the consequence is the equality of their respective parameters.

For well-founded recursion, it is frequently useful to have a generic notion of size available. This is captured in the `SizeOf` class.

type class

```
SizeOf.{u} (α : Sort u) : Sort (max 1 u)
```

`SizeOf` is a typeclass automatically derived for every inductive type, which equips the type with a "size" function to `Nat`. The default instance defines each constructor to be `1` plus the sum of the sizes of all the constructor fields.

This is used for proofs by well-founded induction, since every field of the constructor has a smaller size than the constructor itself, and in many cases this will suffice to do the proof that a recursive function is only called on smaller values. If the default proof strategy fails, it is recommended to supply a custom size measure using the **`termination_by`** argument on the function definition.

Instance Constructor

`SizeOf.mk.{u}`

Methods

`sizeOf : α → Nat`

The "size" of an element, a natural number which decreases on fields of each inductive type.

4.4.4. Run-Time Representation

An inductive type's run-time representation depends both on how many constructors it has, how many arguments each constructor takes, and whether these arguments are relevant.

4.4.4.1. Exceptions

Not every inductive type is represented as indicated here—some inductive types have special support from the Lean compiler:

- The representation of the fixed-width integer types `UInt8`, …, `UInt64`, `Int8`, …, `Int64`, and `USize` depends on the whether the code is compiled for a 32- or 64-bit architecture. Their representation is described in a dedicated section.
- `Char` is represented by `uint32_t`. Because `Char` values never require more than 21 bits, they are always unboxed.
- `Float` is represented by a pointer to a Lean object that contains a `double`.
- An *enum inductive* type of at least 2 and at most 2^{32} constructors, each of which has no parameters, is represented by the first type of **`uint8_t`**, **`uint16_t`**, **`uint32_t`** that is sufficient to assign a unique value to each constructor. For example, the type `Bool` is represented by **`uint8_t`**, with values `0` for `false` and `1` for `true`.
- `Decidable` α is represented the same way as `Bool`
- `Nat` and `Int` are represented by **`lean_object *`**. Their representations are described in more detail in the section on natural numbers and the section on integers.

4.4.4.2. Relevance

Types and proofs have no run-time representation. That is, if an inductive type is a `Prop`, then its values are erased prior to compilation. Similarly, all theorem statements and types are erased. Types with run-time representations are called *relevant*, while types without run-time representations are called *irrelevant*.

▼ *Example: Types are irrelevant*

Even though `List.cons` has the following signature, which indicates three parameters:

```
List.cons.{u} {α : Type u} : α → List α → List α
```

its run-time representation has only two, because the type argument is run-time irrelevant.

▼ *Example: Proofs are irrelevant*

Even though `Fin.mk` has the following signature, which indicates three parameters:

```
Fin.mk {n : Nat} (val : Nat) : val < n → Fin n
```

its run-time representation has only two, because the proof is erased.

In most cases, irrelevant values simply disappear from compiled code. However, in cases where some representation is required (such as when they are arguments to polymorphic constructors), they are represented by a trivial value.

4.4.4.3. Trivial Wrappers

If an inductive type has exactly one constructor, and that constructor has exactly one run-time relevant parameter, then the inductive type is represented identically to its parameter.

▼ *Example: Zero-Overhead Subtypes*

The structure `Subtype` bundles an element of some type with a proof that it satisfies a predicate. Its constructor takes four arguments, but three of them are irrelevant:

```
Subtype.mk.{u} {α : Sort u} {p : α → Prop}
  (val : α) (property : p val) : Subtype p
```

Thus, subtypes impose no runtime overhead in compiled code, and are represented identically to the type of the `val` field.

▼ *Example: Signed Integers*

The signed integer types `Int8`, ..., `Int64`, `ISize` are structures with a single field that wraps the corresponding unsigned integer type. They are represented by the unsigned C types **`uint8_t`**, ..., **`uint64_t`**, **`size_t`**, respectively, because they have a trivial structure.

4.4.4.4. Other Inductive Types

If an inductive type doesn't fall into one of the categories above, then its representation is determined by its constructors. Constructors without relevant parameters are represented by their index into the list of constructors, as unboxed unsigned machine integers (scalars). Constructors with

relevant parameters are represented as an object with a header, the constructor's index, an array of pointers to other objects, and then arrays of scalar fields sorted by their types. The header tracks the object's reference count and other necessary bookkeeping.

Recursive functions are compiled as they are in most programming languages, rather than by using the inductive type's recursor. Elaborating recursive functions to recursors serves to provide reliable termination evidence, not executable code.

4.4.4.4.1. FFI

From the perspective of C, these other inductive types are represented by `lean_object *`. Each constructor is stored as a `lean_ctor_object`, and `lean_is_ctor` will return true. A `lean_ctor_object` stores the constructor index in its header, and the fields are stored in the `m_objs` portion of the object. Lean assumes that `sizeof(size_t) == sizeof(void*)`—while this is not guaranteed by C, the Lean run-time system contains an assertion that fails if this is not the case.

The memory order of the fields is derived from the types and order of the fields in the declaration. They are ordered as follows:

- Non-scalar fields stored as `lean_object *`
- Fields of type USize
- Other scalar fields, in decreasing order by size

Within each group the fields are ordered in declaration order. **Warning**: Trivial wrapper types still count toward a field being treated as non-scalar for this purpose.

- To access fields of the first kind, use `lean_ctor_get(val, i)` to get the `i`th non-scalar field.
- To access USize fields, use `lean_ctor_get_usize(val, n+i)` to get the `i`th USize field and n is the total number of fields of the first kind.
- To access other scalar fields, use `lean_ctor_get_uintN(val, off)` or `lean_ctor_get_usize(val, off)` as appropriate. Here `off` is the byte offset of the field in the structure, starting at `n*sizeof(void*)` where n is the number of fields of the first two kinds.

For example, a structure such as

```
structure S where
  ptr_1 : Array Nat
  usize_1 : USize
  sc64_1 : UInt64
  -- wrappers don't count as scalars:
  ptr_2 : { x : UInt64 // x > 0 }
  sc64_2 : Float -- `Float` is 64 bit
  sc8_1 : Bool
  sc16_1 : UInt16
```

```
sc8_2 : UInt8
sc64_3 : UInt64
usize_2 : USize
-- trivial wrapper around `UInt32`
ptr_3 : Char
sc32_1 : UInt32
sc16_2 : UInt16
```

would get re-sorted into the following memory order:

- `S.ptr_1` - `lean_ctor_get(val, 0)`
- `S.ptr_2` - `lean_ctor_get(val, 1)`
- `S.ptr_3` - `lean_ctor_get(val, 2)`
- `S.usize_1` - `lean_ctor_get_usize(val, 3)`
- `S.usize_2` - `lean_ctor_get_usize(val, 4)`
- `S.sc64_1` - `lean_ctor_get_uint64(val, sizeof(void*)*5)`
- `S.sc64_2` - `lean_ctor_get_float(val, sizeof(void*)*5 + 8)`
- `S.sc64_3` - `lean_ctor_get_uint64(val, sizeof(void*)*5 + 16)`
- `S.sc32_1` - `lean_ctor_get_uint32(val, sizeof(void*)*5 + 24)`
- `S.sc16_1` - `lean_ctor_get_uint16(val, sizeof(void*)*5 + 28)`
- `S.sc16_2` - `lean_ctor_get_uint16(val, sizeof(void*)*5 + 30)`
- `S.sc8_1` - `lean_ctor_get_uint8(val, sizeof(void*)*5 + 32)`
- `S.sc8_2` - `lean_ctor_get_uint8(val, sizeof(void*)*5 + 33)`

4.4.5. Mutual Inductive Types

Inductive types may be mutually recursive. Mutually recursive definitions of inductive types are specified by defining the types in a `mutual ... end` block.

▼ *Example: Mutually Defined Inductive Types*

The type `EvenOddList` in a prior example used a Boolean index to select whether the list in question should have an even or odd number of elements. This distinction can also be expressed by the choice of one of two mutually inductive types `EvenList` and `OddList`:

```
mutual
  inductive EvenList (α : Type u) : Type u where
    | nil : EvenList α
    | cons : α → OddList α → EvenList α
  inductive OddList (α : Type u) : Type u where
    | cons : α → EvenList α → OddList α
end
```

```
example : EvenList String := .cons "x" (.cons "y" .nil)
example : OddList String := .cons "x" (.cons "y" (.cons "z"
.nil))

example : OddList String := .cons "x" (.cons "y" .nil)
```

```
Unknown identifier `OddList.nil`

Note: Inferred this identifier from the expected type of
`.nil`:
  OddList String
```

4.4.5.1. Requirements

The inductive types declared in a `mutual` block are considered as a group; they must collectively satisfy generalized versions of the well-formedness criteria for non-mutually-recursive inductive types. This is true even if they could be defined without the `mutual` block, because they are not in fact mutually recursive.

4.4.5.1.1. Mutual Dependencies

Each type constructor's signature must be able to be elaborated without reference to the other inductive types in the `mutual` group. In other words, the inductive types in the `mutual` group may not take each other as arguments. The constructors of each inductive type may mention the other type constructors in the group in their parameter types, with restrictions that are a generalization of those for recursive occurrences in non-mutual inductive types.

▼ *Example: Mutual inductive type constructors may not mention each other*

These inductive types are not accepted by Lean:

```
mutual
  inductive FreshList (α : Type) (r : α → α → Prop) : Type where
    | nil : FreshList α r
    | cons (x : α) (xs : FreshList α r) (fresh : Fresh r x xs)
  inductive Fresh
      (r : α → FreshList α → Prop) :
      α → FreshList α r → Prop where
    | nil : Fresh r x .nil
    | cons : r x y → (f : Fresh r x ys) → Fresh r x (.cons y ys
f)
end
```

The type constructors may not refer to the other type constructors in the `mutual` group, so `FreshList` is not in scope in the type constructor of `Fresh`:

```
unknown identifier 'FreshList'
```

4.4.5.1.2. Parameters Must Match

All inductive types in the `mutual` group must have the same parameters. Their indices may differ.

▼ *Example: Differing numbers of parameters*

Even though `Both` and `OneOf` are not mutually recursive, they are declared in the same `mutual` block and must therefore have identical parameters:

```
mutual
  inductive Both (α : Type u) (β : Type v) where
    | mk : α → β → Both α β
  inductive Optional (α : Type u) where
    | none
    | some : α → Optional α
end
```

```
invalid inductive type, number of parameters mismatch in
mutually inductive datatypes
```

▼ *Example: Differing parameter types*

Even though `Many` and `OneOf` are not mutually recursive, they are declared in the same `mutual` block and must therefore have identical parameters. They both have exactly one parameter, but `Many`'s parameter is not necessarily in the same universe as `Optional`'s:

```
mutual
  inductive Many (α : Type) : Type u where
    | nil : Many α
    | cons : α → Many α → Many α
  inductive Optional (α : Type u) where
    | none
    | some : α → Optional α
end
```

```
invalid mutually inductive types, parameter 'α' has type
  Type u : Type (u + 1)
but is expected to have type
  Type : Type 1
```

4.4.5.1.3. Universe Levels

The universe levels of each inductive type in a mutual group must obey the same requirements as non-mutually-recursive inductive types. Additionally, all the inductive types in a mutual group must be in the same universe, which implies that their constructors are similarly limited with respect to their parameters' universes.

▼ *Example: Universe mismatch*

These mutually-inductive types are a somewhat complicated way to represent run-length encoding of a list:

```
mutual
  inductive RLE : List α → Type where
  | nil : RLE []
  | run (x : α) (n : Nat) :
    n ≠ 0 → PrefixRunOf n x xs ys → RLE ys → RLE xs

  inductive PrefixRunOf : Nat → α → List α → List α → Type where
  | zero
    (noMore : ¬∃zs, xs = x :: zs := by simp) :
    PrefixRunOf 0 x xs xs
  | succ :
    PrefixRunOf n x xs ys →
    PrefixRunOf (n + 1) x (x :: xs) ys
end

example : RLE [1, 1, 2, 2, 3, 1, 1, 1] :=
  .run 1 2 (by decide) (.succ (.succ .zero)) <|
  .run 2 2 (by decide) (.succ (.succ .zero)) <|
  .run 3 1 (by decide) (.succ .zero) <|
  .run 1 3 (by decide) (.succ (.succ (.succ (.zero)))) <|
  .nil
```

Specifying `PrefixRunOf` as a `Prop` would be sensible, but it cannot be done because the types would be in different universes:

```
mutual
  inductive RLE : List α → Type where
  | nil : RLE []
  | run
    (x : α) (n : Nat) :
    n ≠ 0 → PrefixRunOf n x xs ys → RLE ys →
    RLE xs

  inductive PrefixRunOf : Nat → α → List α → List α → Prop where
  | zero
    (noMore : ¬∃zs, xs = x :: zs := by simp) :
    PrefixRunOf 0 x xs xs
  | succ :
    PrefixRunOf n x xs ys →
    PrefixRunOf (n + 1) x (x :: xs) ys
end
```

```
invalid mutually inductive types, resulting universe
mismatch, given
  Prop
```

```
expected type
  Type
```

This particular property can be expressed by separately defining the well-formedness condition and using a subtype:

```
def RunLengths α := List (α × Nat)
def NoRepeats : RunLengths α → Prop
  | [] => True
  | [_] => True
  | (x, _) :: ((y, n) :: xs) =>
    x ≠ y ∧ NoRepeats ((y, n) :: xs)
def RunsMatch : RunLengths α → List α → Prop
  | [], [] => True
  | (x, n) :: xs, ys =>
    ys.take n = List.replicate n x ∧
    RunsMatch xs (ys.drop n)
  | _, _ => False
def NonZero : RunLengths α → Prop
  | [] => True
  | (_, n) :: xs => n ≠ 0 ∧ NonZero xs
structure RLE (xs : List α) where
  rle : RunLengths α
  noRepeats : NoRepeats rle
  runsMatch : RunsMatch rle xs
  nonZero : NonZero rle

example : RLE [1, 1, 2, 2, 3, 1, 1, 1] where
  rle := [(1, 2), (2, 2), (3, 1), (1, 3)]
  noRepeats := by simp [NoRepeats]
  runsMatch := by simp [RunsMatch]
  nonZero := by simp [NonZero]
```

4.4.5.1.4. Positivity

Each inductive type that is defined in the `mutual` group may occur only strictly positively in the types of the parameters of the constructors of all the types in the group. In other words, in the type of each parameter to each constructor in all the types of the group, none of the type constructors in the group occur to the left of any arrows, and none of them occur in argument positions unless they are an argument to an inductive type's type constructor.

▼ *Example: Mutual strict positivity*

In the following mutual group, `Tm` occurs in a negative position in the argument to `Binding.scope`:

```
mutual
  inductive Tm where
```

```
    | app : Tm → Tm → Tm
    | lam : Binding → Tm
  inductive Binding where
    | scope : (Tm → Tm) → Binding
end
```

Because `Tm` is part of the same mutual group, it must occur only strictly positively in the arguments to the constructors of `Binding`. It occurs, however, negatively:

```
(kernel) arg #1 of 'Binding.scope' has a non positive
occurrence of the datatypes being declared
```

▼ *Example: Nested positions*

The definitions of `LocatedStx` and `Stx` satisfy the positivity condition because the recursive occurrences are not to the left of any arrows and, when they are arguments, they are arguments to inductive type constructors.

```
mutual
  inductive LocatedStx where
    | mk (line col : Nat) (val : Stx)
  inductive Stx where
    | atom (str : String)
    | node (kind : String) (args : List LocatedStx)
end
```

4.4.5.2. Recursors

Mutual inductive types are provided with primitive recursors, just like non-mutually-defined inductive types. These recursors take into account that they must process the other types in the group, and thus will have a motive for each inductive type. Because all inductive types in the `mutual` group are required to have identical parameters, the recursors still take the parameters first, abstracting them over the motives and the rest of the recursor. Additionally, because the recursor must process the group's other types, it will require cases for each constructor of each of the types in the group. The actual dependency structure between the types is not taken into account; even if an additional motive or constructor case is not really required due to there being fewer mutual dependencies than there could be, the generated recursor still requires them.

▼ *Example: Even and odd*

```
mutual
  inductive Even : Nat → Prop where
    | zero : Even 0
    | succ : Odd n → Even (n + 1)
  inductive Odd : Nat → Prop where
```

```
    | succ : Even n → Odd (n + 1)
end

Even.rec
  {motive_1 : (a : Nat) → Even a → Prop}
  {motive_2 : (a : Nat) → Odd a → Prop}
  (zero : motive_1 0 Even.zero)
  (succ : {n : Nat} → (a : Odd n) → motive_2 n a → motive_1 (n +
1) (Even.succ a)) :
  (∀ {n : Nat} (a : Even n), motive_1 n a → motive_2 (n + 1)
(Odd.succ a)) →
  ∀ {a : Nat} (t : Even a), motive_1 a t

Odd.rec
  {motive_1 : (a : Nat) → Even a → Prop}
  {motive_2 : (a : Nat) → Odd a → Prop}
  (zero : motive_1 0 Even.zero)
  (succ : ∀ {n : Nat} (a : Odd n), motive_2 n a → motive_1 (n +
1) (Even.succ a)) :
  (∀ {n : Nat} (a : Even n), motive_1 n a → motive_2 (n + 1)
(Odd.succ a)) → ∀ {a : Nat} (t : Odd a), motive_2 a t
```

▼ *Example: Spuriously mutual types*

The types Two and Three are defined in a mutual block, even though they do not refer to each other:

```
mutual
  inductive Two (α : Type) where
    | mk : α → α → Two α
  inductive Three (α : Type) where
    | mk : α → α → α → Three α
end
```

Two's recursor, Two.rec, nonetheless requires a motive and a case for Three:

```
Two.rec.{u} {α : Type}
  {motive_1 : Two α → Sort u}
  {motive_2 : Three α → Sort u}
  (mk : (a a_1 : α) → motive_1 (Two.mk a a_1)) :
  ((a a_1 a_2 : α) → motive_2 (Three.mk a a_1 a_2)) → (t : Two α)
→ motive_1 t
```

4.4.5.3. Run-Time Representation

Mutual inductive types are represented identically to non-mutual inductive types in compiled code and in the runtime. The restrictions on mutual inductive types exist to ensure Lean's consistency as a logic, and do not impact compiled code.

4.4.5.4. Nested Inductive Types

Nested inductive types are inductive types in which recursive occurrences of the type being defined are parameters to other inductive type constructors. These recursive occurrences are “nested” underneath the other type constructors. Nested inductive types that satisfy certain requirements can be translated into mutual inductive types; this translation demonstrates that they are sound. Internally, the kernel performs this translation; if it succeeds, then the *original* nested inductive type is accepted. This avoids performance and usability issues that would arise from details of the translation surfacing.

Nested recursive occurrences must satisfy the following requirements:

- They must be nested *directly* under an inductive type's type constructor. Terms that reduce to such nested occurrences are not accepted.
- Local variables such as the constructor's parameters may not occur in the arguments to the nested occurrence.
- The nested occurrences must occur strictly positively. They must occur strictly positively in the position in which they are nested, and the type constructor in which they are nested must itself occur in a strictly positive position.
- Constructor parameters whose types include nested occurrences may not be used in ways that rely on the specific choice of outer type constructor. The translated version will not be usable in those contexts.
- Nested occurrences may not be used as parameters to the outer type constructor that occur in the types of the outer type's indices.

▼ *Example: Nested Inductive Types*

Instead of using two constructors, the natural numbers can be defined using `Option`:

```
inductive ONat : Type where
  | mk (pred : Option ONat)
```

Arbitrarily-branching trees, also known as *rose trees*, are nested inductive types:

```
inductive RTree (α : Type u) : Type u where
  | empty
  | node (val : α) (children : List (RTree α))
```

▼ *Example: Invalid Nested Inductive Types*

This declaration of arbitrarily-branching rose trees declares an alias for `List`, rather than using `List` directly:

```
abbrev Children := List

inductive RTree (α : Type u) : Type u where
  | empty
  | node (val : α) (children : Children (RTree α))
```

```
(kernel) arg #3 of 'RTree.node' contains a non valid
occurrence of the datatypes being declared
```

This declaration of arbitrarily-branching rose trees tracks the depth of the tree using an index. The constructor `DRTree.node` has an automatic implicit parameter *n* that represents the depths of all sub-trees. However, local variables such as constructor parameters are not permitted as arguments to nested occurrences:

```
inductive DRTree (α : Type u) : Nat → Type u where
  | empty : DRTree α 0
  | node (val : α) (children : List (DRTree α n)) : DRTree α (n +
1)
```

```
(kernel) invalid nested inductive datatype 'List', nested
inductive datatypes parameters cannot contain local
variables.
```

This declaration includes a non-strictly-positive occurrence of the inductive type, nested under an `Option`:

```
inductive WithCheck where
  | done
  | check (f : Option WithCheck → Bool)
```

```
(kernel) arg #1 of 'WithCheck.check' has a non positive
occurrence of the datatypes being declared
```

This rose tree has a branching factor that's limited by its parameter:

```
inductive BRTree (branches : Nat) (α : Type u) : Type u where
  | mk :
    (children : List (BRTree branches α)) →
    children.length < branches →
    BRTree branches α
```

Only nested inductive types that can be translated to mutual inductive types are allowed. However, translating this type would require a translation of `List.length` to the translated types, but function definitions may not occur in mutual blocks with inductive types. The resulting error message shows that the function was not translated, but was applied to a term of the translated type:

```
(kernel) application type mismatch
  List.length children
argument has type
  @_nested.List_1 branches α
but function has type
  List (@BRTree branches α) → Nat
```

It is acceptable to use the parameter with the nested occurrence with fully polymorphic functions, such as `id`:

```
inductive RTree'' (α : Type u) : Type u where
  | mk :
    (children : List (BRTree branches α)) →
    id children = children →
    BRTree branches α
```

In this case, the function applies equally well to the translated version as it does to the original.

A *palindrome* is a list that is the same when reversed:

```
inductive Palindrome (α : Type) : List α → Prop where
  | nil : Palindrome α []
  | single : Palindrome α [x]
  | cons (x : α) (p : Palindrome α xs) : Palindrome α (x :: xs ++
[x])
```

In this predicate, the list is an index whose type depends on the parameter, which is explicit for clarity. This means it cannot be used

The translation from nested inductive types to mutual inductive types proceeds as follows:

Nested occurrences become inductive types

Nested occurrences of the inductive type are translated into new inductive types in the same mutual group, which replace the original nested occurrences. These new inductive types have the same constructors as the outer inductive type, except the original parameters are instantiated by the translated version of the type. The original inductive type becomes an alias for the version in which the nested occurrences have been rewritten. This process is repeated if the resulting type is also a nested inductive type (e.g. a type nested under `Array` becomes a type nested under `List`, because `Array`'s constructor takes a `List`).

Conversions to and from the nested types

Conversions between the outer inductive type applied to the new alias and the generated auxiliary types are generated. These conversions are then proved to be mutual inverses.

Constructor reconstruction

Each constructor of the original type is defined as a function that returns the constructor of the translated type, after applying the appropriate conversions.

Recursor reconstruction

The recursor for the nested inductive type is constructed from the recursor for the translated type. In the translation, the motives for the nested occurrences are composed with the conversion functions and the minor premises use them as needed. The proofs that the conversion functions are mutually inverse are needed because the encoded constructors convert in one direction, but end up applied to the result of the conversion in the other direction.

▼ *Example: Translating Nested Inductive Types*

This nested inductive type represents the natural numbers:

```
inductive ONat where
  | mk (pred : Option ONat) : ONat

#check ONat.rec
```

The first step in the internal translation is to replace the nested occurrences with auxiliary inductive types that “inline” the resulting type. In this case, the nested occurrence is under `Option`; thus, the auxiliary type has the constructors of `Option`, with `ONat'` substituted for the type parameter:

```
mutual
inductive ONat' where
  | mk (pred : OptONat) : ONat'

inductive OptONat where
  | none
  | some : ONat' → OptONat
end
```

`ONat'` is the encoding of `ONat`:

```
def ONat := ONat'
```

The next step is to define conversion functions that translate the original nested type to and from the auxiliary type:

```
def OptONat.ofOption : Option ONat → OptONat
  | Option.none => OptONat.none
  | Option.some o => OptONat.some o
def OptONat.toOption : OptONat → Option ONat
  | OptONat.none => Option.none
  | OptONat.some o => Option.some o
```

These conversion functions are mutually inverse:

```
def OptONat.to_of_eq_id o :
    OptONat.toOption (ofOption o) = o := by
  cases o <;> rfl
def OptONat.of_to_eq_id o :
```

```
    OptONat.ofOption (OptONat.toOption o) = o := by
  cases o <;> rfl
```

The original constructor is translated to an application of the translation's corresponding constructor, with the appropriate conversion applied for the nested occurrence:

```
def ONat.mk (pred : Option ONat) : ONat :=
  ONat'.mk (.ofOption pred)
```

Finally, the original type's recursor can be translated. The translated recursor uses the translated type's recursor. The original nested occurrences are translated using the conversions, and the proofs that the conversions are mutually inverse are used to rewrite types as needed.

```
noncomputable def ONat.rec
    {motive1 : ONat → Sort u}
    {motive2 : Option ONat → Sort u}
    (h1 :
      (pred : Option ONat) → motive2 pred →
      motive1 (ONat.mk pred))
    (h2 : motive2 none)
    (h3 : (o : ONat) → motive1 o → motive2 (some o)) :
    (t : ONat) → motive1 t :=
  @ONat'.rec motive1 (motive2 ∘ OptONat.toOption)
    (fun pred ih =>
      OptONat.of_to_eq_id pred ▸ h1 pred.toOption ih)
    h2
    h3
```

4.5. Quotients

Quotient types allow a new type to be formed by decreasing the granularity of an existing type's propositional equality. In particular, given an type A and an equivalence relation $\sim$, the quotient $A/\sim$ contains the same elements as A, but every pair of elements that are related by $\sim$ are considered equal. Equality is respected universally; nothing in Lean's logic can observe any difference between two equal terms. Thus, quotient types provide a way to build an impenetrable abstraction barrier. In particular, all functions from a quotient type must prove that they respect the equivalence relation.

def

```
Quotient.{u} {α : Sort u} (s : Setoid α) : Sort u
```

Quotient types coarsen the propositional equality for a type so that terms related by some equivalence relation are considered equal. The equivalence relation is given by an instance of Setoid.

Set-theoretically, `Quotient s` can seen as the set of equivalence classes of α modulo the `Setoid` instance's relation `s.r`. Functions from `Quotient s` must prove that they respect `s.r`: to define a function `f : Quotient s → β`, it is necessary to provide `f' : α → β` and prove that for all `x : α` and `y : α`, `s.r x y → f' x = f' y`. `Quotient.lift` implements this operation.

The key quotient operators are:

- `Quotient.mk` places elements of the underlying type α into the quotient.
- `Quotient.lift` allows the definition of functions from the quotient to some other type.
- `Quotient.sound` asserts the equality of elements related by `r`
- `Quotient.ind` is used to write proofs about quotients by assuming that all elements are constructed with `Quotient.mk`.

`Quotient` is built on top of the primitive quotient type `Quot`, which does not require a proof that the relation is an equivalence relation. `Quotient` should be used instead of `Quot` for relations that actually are equivalence relations.

A proof that two elements of the underlying type are related by the equivalence relation is sufficient to prove that they are equal in the `Quotient`. However, definitional equality is unaffected by the use of `Quotient`: two elements in the quotient are definitionally equal if and only if they are definitionally equal in the underlying type.

Quotient types are not widely used in programming. However, they occur regularly in mathematics:

Integers

The integers are traditionally defined as a pair of natural numbers (n, k) that encodes the integer $n - k$. In this encoding, two integers (n_1, k_1) and (n_2, k_2) are equal if $n_1 + k_2 = n_2 + k_1$.

Rational Numbers

The number $\frac{n}{d}$ can be encoded as the pair (n, d), where $d \neq 0$. Two rational numbers $\frac{n_1}{d_1}$ and $\frac{n_2}{d_2}$ are equal if $n_1 d_2 = n_2 d_1$.

Real Numbers

The real numbers can be represented as a Cauchy sequence, but this encoding is not unique. Using a quotient type, two Cauchy sequences can be made equal when their difference converges to zero.

Finite Sets

Finite sets can be represented as lists of elements. With a quotient types, two finite sets can be made equal if they contain the same elements; this definition does not impose any requirements (such as decidable equality or an ordering relation) on the type of elements.

One alternative to quotient types would be to reason directly about the equivalence classes introduced by the relation. The downside of this approach is that it does not allow *computation*: in addition to knowing *that* there is an integer that is the sum of 5 and 8, it is useful for $5 + 8 = 13$ to not be a theorem that requires proof. Defining functions out of sets of equivalence classes relies on non-computational classical reasoning principles, while functions from quotient types are ordinary computational functions that additionally respect an equivalence relation.

4.5.1. Alternatives to Quotient Types

While `Quotient` is a convenient way to form quotients with reasonable computational properties, it is often possible to define quotients in other ways.

In general, a type Q is said to be the quotient of A by an equivalence relation $\sim$ if it respects the universal property of quotients: there is a function $q : A \to Q$ with the property that $q(a) = q(b)$ if and only if $a \sim b$ for all a and b in A.

Quotients formed with `Quotient` have this property up to propositional equality: elements of A that are related by $\sim$ are equal, so they cannot be distinguished. However, members of the same equivalence class are not necessarily definitionally equal in the quotient.

Quotients may also be implemented by designating a single representative of each equivalence class in A itself, and then defining Q as pair of elements in A with proofs that they are such a canonical representative. Together with a function that maps each a in A to its canonical representative, Q is a quotient of A. Due to proof irrelevance, representatives in Q of the same equivalence class are definitionally equal.

Such a manually implemented quotient Q can be easier to work with than `Quotient`. In particular, because each equivalence class is represented by its single canonical representative, there's no need to prove that functions from the quotient respect the equivalence relation. It can also have better computational properties due to the fact that the computations give normalized values (in contrast, elements of `Quotient` can be represented in multiple ways). Finally, because the manually implemented quotient is an inductive type, it can be used in contexts where other kinds of types cannot, such as when defining a nested inductive type. However, not all quotients can be manually implemented.

▼ *Example: Manually Quotiented Integers*

When implemented as pairs of `Nat`s, each equivalence class according to the desired equality for integers has a canonical representative in which at least one of the `Nat`s is zero. This can be represented as a Lean structure:

```
structure Z where
  a : Nat
  b : Nat
  canonical : a = 0 ∨ b = 0
```

Due to proof irrelevance, every value of this structure type that represents the same integer is *already* equal. Constructing a Z can be made more convenient with a wrapper that uses the fact that subtraction of natural numbers truncates at zero to automate the construction of the proof:

```
def Z.mk' (n k : Nat) : Z where
  a := n - k
  b := k - n
  canonical := by omega
```

This construction respects the equality demanded of integers:

```
theorem Z_mk'_respects_eq :
    (Z.mk' n k = Z.mk' n' k') ↔ (n + k' = n' + k) := by
  simp [Z.mk']
  omega
```

To use this type in examples, it's convenient to have Neg, OfNat, and ToString instances. These instances make it easier to read or write examples.

```
instance : Neg Z where
  neg n := Z.mk' n.b n.a

instance : OfNat Z n where
  ofNat := Z.mk' n 0

instance : ToString Z where
  toString n :=
    if n.a = 0 then
      if n.b = 0 then "0"
      else s!"-{n.b}"
    else toString n.a

#eval (5 : Z)
```

```
5
```

```
#eval (-5 : Z)
```

```
-5
```

Addition is addition of the underlying Nats:

```
instance : Add Z where
  add n k := Z.mk' (n.a + k.a) (n.b + k.b)

#eval (-5 + 22: Z)
```

```
17
```

Because each equivalence class is uniquely represented, there's no need to write a proof that these functions from Z respect the equivalence relation. However, in practice, the API for quotients should be implemented for manually-constructed quotients and proved to respect the universal property.

▼ *Example: Built-In Integers as Quotients*

Lean's built-in integer type `Int` satisfies the universal property of quotients, and can thus be thought of as a quotient of pairs of `Nat`s. The canonical representative of each equivalence class can be computed via comparison and subtraction: (i) This `toInt` function is called `Int.subNatNat` in the standard library.

```
def toInt (n k : Nat) : Int :=
  if n < k then - (k - n : Nat)
  else if n = k then 0
  else (n - k : Nat)
```

It satisfies the universal property. Two pairs of `Nat`s are represent the same integer if and only if `toInt` computes the same `Int` for both pairs:

```
theorem toInt_sound :
    n + k' = k + n' ↔
    toInt n k = toInt n' k' := by
  simp only [toInt]
  split <;> split <;> omega
```

4.5.2. Setoids

Quotient types are built on setoids. A *setoid* is a type paired with a distinguished equivalence relation. Unlike a quotient type, the abstraction barrier is not enforced, and proof automation designed around equality cannot be used with the setoid's equivalence relation. Setoids are useful on their own, in addition to being a building block for quotient types.

type class

```
Setoid.{u} (α : Sort u) : Sort (max 1 u)
```

A setoid is a type with a distinguished equivalence relation, denoted ≈.

The `Quotient` type constructor requires a `Setoid` instance.

Instance Constructor

`Setoid.mk.{u}`

Methods

`r : α → α → Prop`

x ≈ y is the distinguished equivalence relation of a setoid.

`iseqv : Equivalence Setoid.r`

The relation x ≈ y is an equivalence relation.

theorem

```
Setoid.refl.{u} {α : Sort u} [Setoid α] (a : α) : a ≈ a
```

A setoid's equivalence relation is reflexive.

theorem

```
Setoid.symm.{u} {α : Sort u} [Setoid α] {a b : α} (hab : a ≈ b) : b ≈ a
```

A setoid's equivalence relation is symmetric.

theorem

```
Setoid.trans.{u} {α : Sort u} [Setoid α] {a b c : α} (hab : a ≈ b)
  (hbc : b ≈ c) : a ≈ c
```

A setoid's equivalence relation is transitive.

4.5.3. Equivalence Relations

An *equivalence relation* is a relation that is reflexive, symmetric, and transitive.

syntax

Equivalence Relations

Equivalence according to some canonical equivalence relation for a type is written using ≈, which is overloaded using the type class `HasEquiv`.

```
term ::= ...
    | term ≈ term
```

type class

```
HasEquiv.{u, v} (α : Sort u) : Sort (max u (v + 1))
```

`HasEquiv α` is the typeclass which supports the notation `x ≈ y` where `x y : α`.

Instance Constructor

`HasEquiv.mk.{u, v}`

Methods

`Equiv : α → α → Sort v`

`x ≈ y` says that *x* and *y* are equivalent. Because this is a typeclass, the notion of equivalence is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of ≈ in identifiers is `equiv`.

The fact that a relation *r* is actually an equivalence relation is stated `Equivalence r`.

structure

`Equivalence.{u} {α : Sort u} (r : α → α → Prop) : Prop`

An equivalence relation `r : α → α → Prop` is a relation that is

- reflexive: `r x x`,
- symmetric: `r x y` implies `r y x`, and
- transitive: `r x y` and `r y z` implies `r x z`.

Equality is an equivalence relation, and equivalence relations share many of the properties of equality.

Constructor

`Equivalence.mk.{u}`

Fields

`refl : ∀ (x : α), r x x`

An equivalence relation is reflexive: `r x x`

`symm : ∀ {x y : α}, r x y → r y x`

An equivalence relation is symmetric: `r x y` implies `r y x`

`trans : ∀ {x y z : α}, r x y → r y z → r x z`

An equivalence relation is transitive: `r x y` and `r y z` implies `r x z`

Every `Setoid` instance leads to a corresponding `HasEquiv` instance.

4.5.4. Quotient API

The quotient API relies on a pre-existing `Setoid` instance.

4.5.4.1. Introducing Quotients

The type `Quotient` expects an instance of `Setoid` as an ordinary parameter, rather than as an instance implicit parameter. This helps ensure that the quotient uses the intended equivalence relation. The instance can be provided either by naming the instance or by using `inferInstance`.

A value in the quotient is a value from the setoid's underlying type, wrapped in `Quotient.mk`.

def

```
Quotient.mk.{u} {α : Sort u} (s : Setoid α) (a : α) : Quotient s
```

Places an element of a type into the quotient that equates terms according to an equivalence relation.

The setoid instance is provided explicitly. `Quotient.mk'` uses instance synthesis instead.

Given `v : α`, `Quotient.mk s v : Quotient s` is like `v`, except all observations of `v`'s value must respect `s.r`. `Quotient.lift` allows values in a quotient to be mapped to other types, so long as the mapping respects `s.r`.

def

```
Quotient.mk'.{u} {α : Sort u} [s : Setoid α] (a : α) : Quotient s
```

Places an element of a type into the quotient that equates terms according to an equivalence relation.

The equivalence relation is found by synthesizing a `Setoid` instance. `Quotient.mk` instead expects the instance to be provided explicitly.

Given `v : α`, `Quotient.mk' v : Quotient s` is like `v`, except all observations of `v`'s value must respect `s.r`. `Quotient.lift` allows values in a quotient to be mapped to other types, so long as the mapping respects `s.r`.

▼ *Example: The Integers as a Quotient Type*

The integers, defined as pairs of natural numbers where the represented integer is the difference of the two numbers, can be represented via a quotient type. This representation is not unique: both `(4, 7)` and `(1, 4)` represent −3.

Two encoded integers should be considered equal when they are related by `Z.eq`:

```
def Z' : Type := Nat × Nat

def Z.eq (n k : Z') : Prop :=
  n.1 + k.2 = n.2 + k.1
```

This relation is an equivalence relation:

```
def Z.eq.eqv : Equivalence Z.eq where
  refl := by
    intro (x, y)
    simp +arith [eq]
  symm := by
    intro (x, y) (x', y') heq
    simp_all only [eq]
    omega
  trans := by
    intro (x, y) (x', y') (x'', y'')
    intro heq1 heq2
    simp_all only [eq]
    omega
```

Thus, it can be used as a `Setoid`:

```
instance Z.instSetoid : Setoid Z' where
  r := Z.eq
  iseqv := Z.eq.eqv
```

The type Z of integers is then the quotient of `Z'` by the `Setoid` instance:

```
def Z : Type := Quotient Z.instSetoid
```

The helper `Z.mk` makes it simpler to create integers without worrying about the choice of `Setoid` instance:

```
def Z.mk (n : Z') : Z := Quotient.mk _ n
```

However, numeric literals are even more convenient. An `OfNat` instance allows numeric literals to be used for integers:

```
instance : OfNat Z n where
  ofNat := Z.mk (n, 0)
```

4.5.4.2. Eliminating Quotients

Functions from quotients can be defined by proving that a function from the underlying type respects the quotient's equivalence relation. This is accomplished using `Quotient.lift` or its binary counterpart `Quotient.lift₂`. The variants `Quotient.liftOn` and `Quotient.liftOn₂` place the quotient parameter first rather than last in the parameter list.

def

```
Quotient.lift.{u, v} {α : Sort u} {β : Sort v} {s : Setoid α}
  (f : α → β) : (∀ (a b : α), a ≈ b → f a = f b) → Quotient s → β
```

Lifts a function from an underlying type to a function on a quotient, requiring that it respects the quotient's equivalence relation.

Given `s : Setoid α` and a quotient `Quotient s`, applying a function `f : α → β` requires a proof `h` that `f` respects the equivalence relation `s.r`. In this case, the function `Quotient.lift f h : Quotient s → β` computes the same values as `f`.

`Quotient.liftOn` is a version of this operation that takes the quotient value as its first explicit parameter.

def

```
Quotient.liftOn.{u, v} {α : Sort u} {β : Sort v} {s : Setoid α}
  (q : Quotient s) (f : α → β) (c : ∀ (a b : α), a ≈ b → f a = f b)
: β
```

Lifts a function from an underlying type to a function on a quotient, requiring that it respects the quotient's equivalence relation.

Given `s : Setoid α` and a quotient value `q : Quotient s`, applying a function `f : α → β` requires a proof c that `f` respects the equivalence relation `s.r`. In this case, the term `Quotient.liftOn q f h : β` reduces to the result of applying `f` to the underlying `α` value.

`Quotient.lift` is a version of this operation that takes the quotient value last, rather than first.

def

```
Quotient.lift₂.{uA, uB, uC} {α : Sort uA} {β : Sort uB} {φ : Sort
uC}
  {s₁ : Setoid α} {s₂ : Setoid β} (f : α → β → φ)
  (c :
    ∀ (a₁ : α) (b₁ : β) (a₂ : α) (b₂ : β),
      a₁ ≈ a₂ → b₁ ≈ b₂ → f a₁ b₁ = f a₂ b₂)
  (q₁ : Quotient s₁) (q₂ : Quotient s₂) : φ
```

Lifts a binary function from the underlying types to a binary function on quotients. The function must respect both quotients' equivalence relations.

`Quotient.lift` is a version of this operation for unary functions. `Quotient.liftOn₂` is a version that take the quotient parameters first.

def

```
Quotient.liftOn₂.{uA, uB, uC} {α : Sort uA} {β : Sort uB} {φ : Sort
uC}
  {s₁ : Setoid α} {s₂ : Setoid β} (q₁ : Quotient s₁) (q₂ : Quotient
s₂)
  (f : α → β → φ)
  (c :
    ∀ (a₁ : α) (b₁ : β) (a₂ : α) (b₂ : β),
      a₁ ≈ a₂ → b₁ ≈ b₂ → f a₁ b₁ = f a₂ b₂) :
  φ
```

Lifts a binary function from the underlying types to a binary function on quotients. The function must respect both quotients' equivalence relations.

`Quotient.liftOn` is a version of this operation for unary functions. `Quotient.lift₂` is a version that take the quotient parameters last.

▼ *Example: Integer Negation and Addition*

Given the encoding Z of integers as a quotient of pairs of natural numbers, negation can be implemented by swapping the first and second projections:

```
def neg' : Z' → Z
  | (x, y) => ⟦(y, x)⟧
```

This can be transformed into a function from Z to Z by proving that negation respects the equivalence relation:

```
instance : Neg Z where
  neg :=
    Quotient.lift neg' <| by
      intro n k equiv
      apply Quotient.sound
      simp only [· ≈ ·, Setoid.r, Z.eq] at *
      omega
```

Similarly, `Quotient.lift₂` is useful for defining binary functions from a quotient type. Addition is defined point-wise:

```
def add' (n k : Nat × Nat) : Z :=
  ⟦(n.1 + k.1, n.2 + k.2)⟧
```

Lifting it to the quotient requires a proof that addition respects the equivalence relation:

```
instance : Add Z where
  add (n : Z) :=
    n.lift₂ add' <| by
      intro n k n' k'
      intro heq heq'
      apply Quotient.sound
      cases n; cases k; cases n'; cases k'
      simp_all only [· ≈ ·, Setoid.r, Z.eq]
      omega
```

When the function's result type is a subsingleton, `Quotient.recOnSubsingleton` or `Quotient.recOnSubsingleton₂` can be used to define the function. Because all elements of a subsingleton are equal, such a function automatically respects the equivalence relation, so there is no proof obligation.

def

```
Quotient.recOnSubsingleton.{u, v} {α : Sort u} {s : Setoid α}
  {motive : Quotient s → Sort v}
  [h : ∀ (a : α), Subsingleton (motive (Quotient.mk s a))]
  (q : Quotient s) (f : (a : α) → motive (Quotient.mk s a)) :
motive q
```

An alternative recursion or induction principle for quotients that can be used when the target type is a subsingleton, in which all elements are equal.

In these cases, the proof that the function respects the quotient's equivalence relation is trivial, so any function can be lifted.

`Quotient.rec` does not assume that the target type is a subsingleton.

def

```
Quotient.recOnSubsingleton₂.{uA, uB, uC} {α : Sort uA} {β : Sort uB}
  {s₁ : Setoid α} {s₂ : Setoid β}
  {motive : Quotient s₁ → Quotient s₂ → Sort uC}
  [s :
    ∀ (a : α) (b : β),
      Subsingleton (motive (Quotient.mk s₁ a) (Quotient.mk s₂ b))]
  (q₁ : Quotient s₁) (q₂ : Quotient s₂)
  (g :
    (a : α) → (b : β) → motive (Quotient.mk s₁ a) (Quotient.mk s₂
b)) :
  motive q₁ q₂
```

An alternative induction or recursion operator for defining binary operations on quotients that can be used when the target type is a subsingleton.

In these cases, the proof that the function respects the quotient's equivalence relation is trivial, so any function can be lifted.

4.5.4.3. Proofs About Quotients

The fundamental tools for proving properties of elements of quotient types are the soundness axiom and the induction principle. The soundness axiom states that if two elements of the underlying type are related by the quotient's equivalence relation, then they are equal in the quotient type. The induction principle follows the structure of recursors for inductive types: in order to prove that a predicate holds all elements of a quotient type, it suffices to prove that it holds for an application of `Quotient.mk` to each element of the underlying type. Because `Quotient` is not an inductive type, tactics such as **`cases`** and **`induction`** require that `Quotient.ind` be specified explicitly with the `using` modifier.

theorem

```
Quotient.sound.{u} {α : Sort u} {s : Setoid α} {a b : α} :
  a ≈ b → Quotient.mk s a = Quotient.mk s b
```

The **quotient axiom**, which asserts the equality of elements related in the setoid.

Because `Quotient` is built on a lower-level type `Quot`, `Quotient.sound` is implemented as a theorem. It is derived from `Quot.sound`, the soundness axiom for the lower-level quotient type `Quot`.

theorem

```
Quotient.ind.{u} {α : Sort u} {s : Setoid α}
  {motive : Quotient s → Prop} :
  (∀ (a : α), motive (Quotient.mk s a)) → ∀ (q : Quotient s),
motive q
```

A reasoning principle for quotients that allows proofs about quotients to assume that all values are constructed with `Quotient.mk`.

▼ *Example: Proofs About Quotients*

Given the definition of integers as a quotient type from the prior examples, `Quotient.ind` and `Quotient.sound` can be used to prove that negation is an additive inverse. First, `Quotient.ind` is used to replace instances of n with applications of `Quotient.mk`. Having done so, the left side of the equality becomes definitionally equal to a single application of

Quotient.mk, via unfolding definitions and the computation rule for Quotient.lift. This makes Quotient.sound applicable, which yields a new goal: to show that both sides are related by the equivalence relation. This is provable using simp_arith.

```
theorem Z.add_neg_inverse (n : Z) : n  + (-n) = 0 := by
  cases n using Quotient.ind
  apply Quotient.sound
  simp +arith [· ≈ ·, Setoid.r, eq]
```

For more specialized use cases, Quotient.rec, Quotient.recOn, and Quotient.hrecOn can be used to define dependent functions from a quotient type to a type in any other universe. Stating that a dependent function respects the quotient's equivalence relation requires a means of dealing with the fact that the dependent result type is instantiated with different values from the quotient on each side of the equality. Quotient.rec and Quotient.recOn use the Quotient.sound to equate the related elements, inserting the appropriate cast into the statement of equality, while Quotient.hrecOn uses heterogeneous equality.

def

```
Quotient.rec.{u, v} {α : Sort u} {s : Setoid α}
  {motive : Quotient s → Sort v}
  (f : (a : α) → motive (Quotient.mk s a))
  (h : ∀ (a b : α) (p : a ≈ b), ⋯ ▸ f a = f b) (q : Quotient s) :
  motive q
```

A dependent recursion principle for Quotient. It is analogous to the recursor for a structure, and can be used when the resulting type is not necessarily a proposition.

While it is very general, this recursor can be tricky to use. The following simpler alternatives may be easier to use:

- Quotient.lift is useful for defining non-dependent functions.
- Quotient.ind is useful for proving theorems about quotients.
- Quotient.recOnSubsingleton can be used whenever the target type is a Subsingleton.
- Quotient.hrecOn uses heterogeneous equality instead of rewriting with Quotient.sound.

Quotient.recOn is a version of this recursor that takes the quotient parameter first.

def

```
Quotient.recOn.{u, v} {α : Sort u} {s : Setoid α}
  {motive : Quotient s → Sort v} (q : Quotient s)
  (f : (a : α) → motive (Quotient.mk s a))
  (h : ∀ (a b : α) (p : a ≈ b), ⋯ ▸ f a = f b) : motive q
```

A dependent recursion principle for Quotient. It is analogous to the recursor for a structure, and can be used when the resulting type is not necessarily a proposition.

While it is very general, this recursor can be tricky to use. The following simpler alternatives may be easier to use:

- Quotient.lift is useful for defining non-dependent functions.
- Quotient.ind is useful for proving theorems about quotients.
- Quotient.recOnSubsingleton can be used whenever the target type is a Subsingleton.
- Quotient.hrecOn uses heterogeneous equality instead of rewriting with Quotient.sound.

Quotient.rec is a version of this recursor that takes the quotient parameter last.

def

```
Quotient.hrecOn.{u, v} {α : Sort u} {s : Setoid α}
  {motive : Quotient s → Sort v} (q : Quotient s)
  (f : (a : α) → motive (Quotient.mk s a))
  (c : ∀ (a b : α), a ≈ b → f a ≍ f b) : motive q
```

A dependent recursion principle for Quotient that uses heterogeneous equality, analogous to a recursor for a structure.

Quotient.recOn is a version of this recursor that uses Eq instead of HEq.

If two elements of a type are equal in a quotient, then they are related by the setoid's equivalence relation. This property is called Quotient.exact.

theorem

```
Quotient.exact.{u} {α : Sort u} {s : Setoid α} {a b : α} :
  Quotient.mk s a = Quotient.mk s b → a ≈ b
```

If two values are equal in a quotient, then they are related by its equivalence relation.

4.5.5. Logical Model

Like functions and universes, quotient types are a built-in feature of Lean's type system. However, the underlying primitives are based on the somewhat simpler Quot type rather than on Quotient, and Quotient is defined in terms of Quot. The primary difference is that Quot is based on an arbitrary relation, rather than a Setoid instance. The provided relation need not be an equivalence relation; the rules that govern Quot and Eq automatically extend the provided relation into its

reflexive, transitive, symmetric closure. When the relation is already an equivalence relation, `Quotient` should be used instead of `Quot` so Lean can make use of the fact that the relation is an equivalence relation.

The fundamental quotient type API consists of `Quot`, `Quot.mk`, `Quot.lift`, `Quot.ind`, and `Quot.sound`. These are used in the same way as their `Quotient`-based counterparts.

primitive

```
Quot.{u} {α : Sort u} (r : α → α → Prop) : Sort u
```

Low-level quotient types. Quotient types coarsen the propositional equality for a type α, so that terms related by some relation r are considered equal in `Quot r`.

Set-theoretically, `Quot r` can seen as the set of equivalence classes of α modulo r. Functions from `Quot r` must prove that they respect r: to define a function f : `Quot r` → β, it is necessary to provide f' : α → β and prove that for all x : α and y : α, $r\ x\ y \to f'\ x = f'\ y$.

`Quot` is a built-in primitive:

- `Quot.mk` places elements of the underlying type α into the quotient.
- `Quot.lift` allows the definition of functions from the quotient to some other type.
- `Quot.sound` asserts the equality of elements related by r.
- `Quot.ind` is used to write proofs about quotients by assuming that all elements are constructed with `Quot.mk`.

The relation r is not required to be an equivalence relation; the resulting quotient type's equality extends r to an equivalence as a consequence of the rules for equality and quotients. When r is an equivalence relation, it can be more convenient to use the higher-level type `Quotient`.

primitive

```
Quot.mk.{u} {α : Sort u} (r : α → α → Prop) (a : α) : Quot r
```

Places an element of a type into the quotient that equates terms according to the provided relation.

Given v : α and relation r : $\alpha \to \alpha \to$ `Prop`, `Quot.mk r v` : `Quot r` is like v, except all observations of v's value must respect r.

`Quot.mk` is a built-in primitive:

- `Quot` is the built-in quotient type.
- `Quot.lift` allows the definition of functions from the quotient to some other type.
- `Quot.sound` asserts the equality of elements related by r.

- Quot.ind is used to write proofs about quotients by assuming that all elements are constructed with Quot.mk.

primitive

```
Quot.lift.{u, v} {α : Sort u} {r : α → α → Prop} {β : Sort v}
  (f : α → β) (a : ∀ (a b : α), r a b → f a = f b) : Quot r → β
```

Lifts a function from an underlying type to a function on a quotient, requiring that it respects the quotient's relation.

Given a relation `r : α → α → Prop` and a quotient `Quot r`, applying a function `f : α → β` requires a proof `a` that `f` respects `r`. In this case, `Quot.lift f a : Quot r → β` computes the same values as `f`.

Lean's type theory includes a definitional reduction from `Quot.lift f h (Quot.mk r v)` to `f v`.

Quot.lift is a built-in primitive:

- Quot is the built-in quotient type.
- Quot.mk places elements of the underlying type `α` into the quotient.
- Quot.sound asserts the equality of elements related by `r`
- Quot.ind is used to write proofs about quotients by assuming that all elements are constructed with Quot.mk; it is analogous to the recursor for a structure.

primitive

```
Quot.ind.{u} {α : Sort u} {r : α → α → Prop} {β : Quot r → Prop}
  (mk : ∀ (a : α), β (Quot.mk r a)) (q : Quot r) : β q
```

A reasoning principle for quotients that allows proofs about quotients to assume that all values are constructed with Quot.mk.

Quot.rec is analogous to the recursor for a structure, and can be used when the resulting type is not necessarily a proposition.

Quot.ind is a built-in primitive:

- Quot is the built-in quotient type.
- Quot.mk places elements of the underlying type `α` into the quotient.
- Quot.lift allows the definition of functions from the quotient to some other type.
- Quot.sound asserts the equality of elements related by `r`.

axiom

```
Quot.sound.{u} {α : Sort u} {r : α → α → Prop} {a b : α} :
  r a b → Quot.mk r a = Quot.mk r b
```

The **quotient axiom**, which asserts the equality of elements related by the quotient's relation.

The relation r does not need to be an equivalence relation to use this axiom. When r is not an equivalence relation, the quotient is with respect to the equivalence relation generated by r.

`Quot.sound` is part of the built-in primitive quotient type:

- `Quot` is the built-in quotient type.
- `Quot.mk` places elements of the underlying type α into the quotient.
- `Quot.lift` allows the definition of functions from the quotient to some other type.
- `Quot.ind` is used to write proofs about quotients by assuming that all elements are constructed with `Quot.mk`; it is analogous to the recursor for a structure.

Quotient types are described in more detail in the Lean Language Reference.

4.5.5.1. Quotient Reduction

In addition to the above constants, Lean's kernel contains a reduction rule for `Quot.lift` that causes it to reduce when used with `Quot.mk`, analogous to ι-reduction for inductive types. Given a relation r over α, a function f from α to β, and a proof $resp$ that f respects r, the term `Quot.lift f resp (Quot.mk r x)` is definitionally equal to $f\ x$.

```
variable
  (r : α → α → Prop)
  (f : α → β)
  (ok : ∀ x y, r x y → f x = f y)
  (x : α)

example : Quot.lift f ok (Quot.mk r x) = f x := rfl
```

4.5.5.2. Quotients and Inductive Types

Because `Quot` is not an inductive type, types implemented as quotients may not occur around nested occurrences in inductive type declarations. These types declarations must be rewritten to remove the nested quotient, which can often be done by defining a quotient-free version and then separately defining an equivalence relation that implements the desired equality relation.

▼ *Example: Nested Inductive Types and Quotients*

The nested inductive type of rose trees nests the recursive occurrence of `RoseTree` under `List`:

```
inductive RoseTree (α : Type u) where
  | leaf : α → RoseTree α
  | branch : List (RoseTree α) → RoseTree α
```

However, taking a quotient of the `List` that identifies all elements in the style of squash types causes Lean to reject the declaration:

```
inductive SetTree (α : Type u) where
  | leaf : α → SetTree α
  | branch :
    Quot (fun (xs ys : List (SetTree α)) => True) →
    SetTree α
```

```
(kernel) arg #2 of 'SetTree.branch' contains a non valid
occurrence of the datatypes being declared
```

4.5.5.3. Low-Level Quotient API

`Quot.liftOn` is an version of `Quot.lift` that takes the quotient type's value first, by analogy to `Quotient.liftOn`.

def

```
Quot.liftOn.{u, v} {α : Sort u} {β : Sort v} {r : α → α → Prop}
  (q : Quot r) (f : α → β) (c : ∀ (a b : α), r a b → f a = f b) : β
```

Lifts a function from an underlying type to a function on a quotient, requiring that it respects the quotient's relation.

Given a relation $r : \alpha \to \alpha \to \text{Prop}$ and a quotient's value $q : \text{Quot } r$, applying a $f : \alpha \to \beta$ requires a proof c that f respects r. In this case, `Quot.liftOn q f h : β` evaluates to the result of applying f to the underlying value in α from q.

`Quot.liftOn` is a version of the built-in primitive `Quot.lift` with its parameters re-ordered.

Quotient types are described in more detail in the Lean Language Reference.

Lean also provides convenient elimination from `Quot` into any subsingleton without further proof obligations, along with dependent elimination principles that correspond to those used for `Quotient`.

def

```
Quot.recOnSubsingleton.{u, v} {α : Sort u} {r : α → α → Prop}
  {motive : Quot r → Sort v}
  [h : ∀ (a : α), Subsingleton (motive (Quot.mk r a))] (q : Quot r)
  (f : (a : α) → motive (Quot.mk r a)) : motive q
```

An alternative induction principle for quotients that can be used when the target type is a subsingleton, in which all elements are equal.

In these cases, the proof that the function respects the quotient's relation is trivial, so any function can be lifted.

`Quot.rec` does not assume that the type is a subsingleton.

def

```
Quot.rec.{u, v} {α : Sort u} {r : α → α → Prop}
  {motive : Quot r → Sort v} (f : (a : α) → motive (Quot.mk r a))
  (h : ∀ (a b : α) (p : r a b), ⋯ ▸ f a = f b) (q : Quot r) :
motive q
```

A dependent recursion principle for `Quot`. It is analogous to the recursor for a structure, and can be used when the resulting type is not necessarily a proposition.

While it is very general, this recursor can be tricky to use. The following simpler alternatives may be easier to use:

- `Quot.lift` is useful for defining non-dependent functions.
- `Quot.ind` is useful for proving theorems about quotients.
- `Quot.recOnSubsingleton` can be used whenever the target type is a `Subsingleton`.
- `Quot.hrecOn` uses heterogeneous equality instead of rewriting with `Quot.sound`.

`Quot.recOn` is a version of this recursor that takes the quotient parameter first.

def

```
Quot.recOn.{u, v} {α : Sort u} {r : α → α → Prop}
  {motive : Quot r → Sort v} (q : Quot r)
  (f : (a : α) → motive (Quot.mk r a))
  (h : ∀ (a b : α) (p : r a b), ⋯ ▸ f a = f b) : motive q
```

A dependent recursion principle for `Quot` that takes the quotient first. It is analogous to the recursor for a structure, and can be used when the resulting type is not necessarily a proposition.

While it is very general, this recursor can be tricky to use. The following simpler alternatives may be easier to use:

- `Quot.lift` is useful for defining non-dependent functions.
- `Quot.ind` is useful for proving theorems about quotients.

- `Quot.recOnSubsingleton` can be used whenever the target type is a `Subsingleton`.
- `Quot.hrecOn` uses heterogeneous equality instead of rewriting with `Quot.sound`.

`Quot.rec` is a version of this recursor that takes the quotient parameter last.

def

```
Quot.hrecOn.{u, v} {α : Sort u} {r : α → α → Prop}
  {motive : Quot r → Sort v} (q : Quot r)
  (f : (a : α) → motive (Quot.mk r a))
  (c : ∀ (a b : α), r a b → f a ≍ f b) : motive q
```

A dependent recursion principle for `Quot` that uses heterogeneous equality, analogous to a recursor for a structure.

`Quot.recOn` is a version of this recursor that uses `Eq` instead of `HEq`.

4.5.6. Quotients and Function Extensionality

Because Lean's definitional equality includes a computational reduction rule for `Quot.lift`, quotient types are used in the standard library to prove function extensionality, which would need to be an axiom otherwise. This is done by first defining a type of functions quotiented by extensional equality, for which extensional equality holds by definition.

```
variable {α : Sort u} {β : α → Sort v}

def extEq (f g : (x : α) → β x) : Prop :=
  ∀ x, f x = g x

def ExtFun (α : Sort u) (β : α → Sort v) :=
  Quot (@extEq α β)
```

Extensional functions can be applied just like ordinary functions. Application respects extensional equality by definition: if applying to functions gives equal results, then applying them gives equal results.

```
def extApp
    (f : ExtFun α β)
    (x : α) :
    β x :=
  f.lift (· x) fun g g' h => by
    exact h x
```

To show that two functions that are extensionally equal are in fact equal, it suffices to show that the functions that result from extensionally applying the corresponding extensional functions are equal. This is because

```
extApp (Quot.mk _ f)
```

is definitionally equal to

```
fun x => (Quot.mk extEq f).lift (· x) (fun _ _ h => h x)
```

which is definitionally equal to `fun x => f x`, which is definitionally equal (by η-equivalence) to `f`. A propositional version of the computation rule for `Quot.lift` would not suffice, because the reducible expression occurs in the body of a function and rewriting by an equality in a function would already require function extensionality.

From here, it is enough to show that the extensional versions of the two functions are equal. This is true due to `Quot.sound`: the fact that they are in the quotient's equivalence relation is an assumption. This proof is a much more explicit version of the one in the standard library:

```
theorem funext'
    {f g : (x : α) → β x}
    (h : ∀ x, f x = g x) :
    f = g := by
  suffices extApp (Quot.mk _ f) = extApp (Quot.mk _ g) by
    unfold extApp at this
    dsimp at this
    exact this
  suffices Quot.mk extEq f = Quot.mk extEq g by
    apply congrArg
    exact this
  apply Quot.sound
  exact h
```

4.5.7. Squash Types

Squash types are a quotient by the relation that relates all elements, transforming it into a subsingleton. In other words, if α is inhabited, then `Squash α` has a single element, and if α is uninhabited, then `Squash α` is also uninhabited. Unlike `Nonempty α`, which is a proposition stating that α is inhabited and is thus represented by a dummy value at runtime, `Squash α` is a type that is represented identically to α. Because `Squash α` is in the same universe as α, it is not subject to the restrictions on computing data from propositions.

def

```
Squash.{u} (α : Sort u) : Sort u
```

The quotient of α by the universal relation. The elements of `Squash α` are those of α, but all of them are equal and cannot be distinguished.

`Squash α` is a `Subsingleton`: it is empty if α is empty, otherwise it has just one element. It is the “universal `Subsingleton`” mapped from α.

`Nonempty α` also has these properties. It is a proposition, which means that its elements (i.e. proofs) are erased from compiled code and represented by a dummy value. `Squash α` is a `Type u`, and its representation in compiled code is identical to that of `α`.

Consequently, `Squash.lift` may extract an `α` value into any subsingleton type β, while `Nonempty.rec` can only do the same when β is a proposition.

def

```
Squash.mk.{u} {α : Sort u} (x : α) : Squash α
```

Places a value into its squash type, in which it cannot be distinguished from any other.

def

```
Squash.lift.{u_1, u_2} {α : Sort u_1} {β : Sort u_2} [Subsingleton β]
  (s : Squash α) (f : α → β) : β
```

Extracts a squashed value into any subsingleton type.

If `β` is a subsingleton, a function `α → β` cannot distinguish between elements of `α` and thus automatically respects the universal relation that `Squash` quotients with.

theorem

```
Squash.ind.{u} {α : Sort u} {motive : Squash α → Prop}
  (h : ∀ (a : α), motive (Squash.mk a)) (q : Squash α) : motive q
```

A reasoning principle that allows proofs about squashed types to assume that all values are constructed with `Squash.mk`.

5. Source Files and Modules

The smallest unit of compilation in Lean is a single module. Modules correspond to source files, and are imported into other modules based on their filenames. In other words, the names and folder structures of files are significant in Lean code.

Every Lean file defines a module. A module's name is derived from a combination of its filename and the way in which Lean was invoked: Lean has a *root directory* in which it expects to find code, and the module's name is the names of the directories from the root to the filename, with dots (`.`) interspersed and `.lean` removed. For example, if Lean is invoked with `Projects/MyLib/src` as its root, the file `Projects/MyLib/src/Literature/Novel/SciFi.lean` would contain a module named `Literature.Novel.SciFi`.

5.1. Encoding and Representation

Lean modules are Unicode text files encoded in UTF-8. Lines may end either with newline characters (`"\n"`, Unicode `'LINE FEED (LF)' (U+000A)`) or with a form feed and newline sequence (`"\r\n"`, Unicode `'CARRIAGE RETURN (CR)' (U+000D)` followed by `'LINE FEED (LF)' (U+000A)`). However, Lean normalizes line endings when parsing or comparing files, so all files are compared as if all their line endings are `"\n"`.

5.2. Concrete Syntax

Lean's concrete syntax is extensible. In a language like Lean, it's not possible to completely describe the syntax once and for all, because libraries may define syntax in addition to new constants or inductive types. Rather than completely describing the language here, the overall framework is described, while the syntax of each language construct is documented in the section to which it belongs.

5.2.1. Whitespace

Tokens in Lean may be separated by any number of *whitespace* character sequences. Whitespace may be a space (`" "`, Unicode `'SPACE (SP)' (U+0020)`), a valid newline sequence, or a comment. Neither tab characters nor carriage returns not followed by newlines are valid whitespace sequences.

5.2.2. Comments

Comments are stretches of the file that, despite not being whitespace, are treated as such. Lean has two syntaxes for comments:

Line comments

A `--` that does not occur as part of a token begins a *line comment*. All characters from the initial `-` to the newline are treated as whitespace.

Block comments

A `/-` that does not occur as part of a token and is not immediately followed by a `-` character begins a *block comment*. The block comment continues until a terminating `-/` is found. Block comments may be nested; a `-/` only terminates the comment if prior nested block comment openers `/-` have been terminated by a matching `-/`.

`/--` and `/-!` begin *documentation* rather than comments, which are also terminated with `-/` and may contain nested block comments. Even though documentation resembles comments, they are their own syntactic category; their valid placement is determined by Lean's grammar.

5.2.3. Keywords and Identifiers

An identifier consists of one or more identifier components, separated by `'.'`.

Identifier components consist of a letter or letter-like character or an underscore (`'_'`), followed by zero or more identifier continuation characters. Letters are English letters, upper- or lowercase, and the letter-like characters include a range of non-English alphabetic scripts, including the Greek script which is widely used in Lean, the Coptic script, the members of the Unicode letter-like symbol block, which contains a number of double-struck characters (including ℕ and ℤ) and abbreviations, the Latin-1 supplemental letters (with the exception of × and ÷), and the Latin Extended-A block. Identifier continuation characters consist of letters, letter-like characters, underscores (`'_'`), exclamation marks (`!`), question marks (`?`), subscripts, and single quotes (`'`). As an exception, underscore alone is not a valid identifier.

Identifiers components may also be surrounded by double guillemets (`'«'` and `'»'`). Such identifier components may contain any character at all aside from `'»'`, even `'«'`, `'.'`, and newlines. The guillemets are not part of the resulting identifier component, so «x» and x denote the same identifier. «`Nat.add`», on the other hand, is an identifier with a single component, while `Nat.add` has two.

Some potential identifier components may be reserved keywords. The specific set of reserved keywords depends on the set of active syntax extensions, which may depend on the set of imported modules and the currently-opened namespaces; it is impossible to enumerate for Lean as a whole. These keywords must also be quoted with guillemets to be used as identifier components in most syntactic contexts. Contexts in which keywords may be used as identifiers without guillemets, such as constructor names in inductive types, are *raw identifier* contexts.

Identifiers that contain one or more `'.'` characters, and thus consist of more than one identifier component, are called hierarchical identifiers. Hierarchical identifiers are used to represent both module names and names in a namespace.

5.3. Structure

syntax

Modules

```
module ::=
    header command*
```

A module consists of a *module header* followed by a sequence of *commands*.

5.3.1. Module Headers

Module headers list the modules that should be elaborated prior to the current module. Their declarations are visible in the current module.

syntax

Module Headers

The module header consists of a sequence of `import` statements:

```
header ::=
    import*
```

The optional `prelude` keyword should only be used in Lean's source code:

```
header ::= ...
    | prelude
      import*
```

If present, the `prelude` keyword indicates that the module is part of the implementation of the Lean *prelude*, which is the code that is available without explicitly importing any modules—it should not be used outside of Lean's implementation.

syntax

Prelude Modules

```
prelude ::=
    prelude
```

syntax

Imports

```
import ::= ...
    | import ident
```

Imports the module. Importing a module makes its contents available in the current module, as well as those from modules transitively imported by its imports.

Modules do not necessarily correspond to namespaces. Modules may add names to any namespace, and importing a module has no effect on the set of currently open namespaces.

The imported module name is translated to a filename by replacing dots (`'.'`) in its name with directory separators and appending `.lean` or `.olean`. Lean searches its include path for the corresponding intermediate build product or importable module file.

5.3.2. Commands

Commands are top-level statements in Lean. Some examples are inductive type declarations, theorems, function definitions, namespace modifiers like `open` or `variable`, and interactive queries such as `#check`. The syntax of commands is user-extensible, and commands may even add new syntax that is used to parse subsequent commands. Specific Lean commands are documented in the corresponding chapters of this manual, rather than being listed here.

5.4. Elaborated Modules

When Lean elaborates a module, the result is an environment. The environment includes the constants, inductive types, theorems, type classes, instances, and everything else declared in the module, along with side tables that track data as diverse as simp sets, namespace aliases, and documentation comments.

As the module is processed by Lean, commands add content to the environment. A module's environment can be serialized to a `.olean` file, which contains both the environment and a compacted heap region with the run-time objects needed by the environment. This means that an imported module can be loaded without re-executing all of its commands.

5.5. Packages, Libraries, and Targets

Lean modules are organized into *packages*, which are units of code distribution. A package may contain multiple libraries or executables.

Code in a package that is intended for use by other Lean packages is organized into libraries. Code that is intended to be compiled and run as independent programs is organized into executables. Packages, libraries, and executables are described in detail in the section on Lake, the standard Lean build tool.

6. Namespaces and Sections

Names are organized into hierarchical *namespaces*, which are collections of names. Namespaces are the primary means of organizing APIs in Lean: they provide an ontology of operations, grouping related items. Additionally, while this is not done by giving them names in the namespace, the effects of features such as syntax extensions, instances, and attributes can be attached to a namespace.

Sorting operations into namespaces organizes libraries conceptually, from a global perspective. Any given Lean file will, however, typically not use all names equally. Sections provide a means of ordering a local view of the globally-available collection of names, as well as a way to precisely control the scope of compiler options along with language extensions, instances, and attributes. They also allow parameters shared by many declarations to be declared centrally and propagated as needed using the `variable` command.

6.1. Namespaces

Names that contain periods (that aren't inside guillemets) are hierarchical names; the periods separate the *components* of a name. All but the final component of a name are the namespace, while the final component is the name itself.

Namespaces serve to group related definitions, theorems, types, and other declarations. When a namespace corresponds to a type's name, generalized field notation can be used to access its contents. In addition to organizing names, namespaces also group syntax extensions, attributes, and

instances.

Namespaces are orthogonal to modules: a module is a unit of code that is elaborated, compiled, and loaded together, but there is no necessary connection between a module's name and the names that it provides. A module may contain names in any namespace, and the nesting structure of hierarchical modules is unrelated to that of hierarchical namespaces.

There is a root namespace, ordinarily denoted by simply omitting a namespace. It can be explicitly indicated by beginning a name with `_root_`. This can be necessary in contexts where a name would otherwise be interpreted relative to an ambient namespace (e.g. from a section scope) or local scope.

▼ *Example: Explicit Root Namespace*

Names in the current namespace take precedence over names in the root namespace. In this example, `color` in the definition of `Forest.statement` refers to `Forest.color`:

```
def color := "yellow"
namespace Forest
def color := "green"
def statement := s!"Lemons are {color}"
end Forest

#eval Forest.statement
```

```
"Lemons are green"
```

Within the `Forest` namespace, references to `color` in the root namespace must be qualified with `_root_`:

```
namespace Forest
def nextStatement :=
  s!"Ripe lemons are {_root_.color}, not {color}"
end Forest

#eval Forest.nextStatement
```

```
"Ripe lemons are yellow, not green"
```

6.1.1. Namespaces and Section Scopes

Every section scope has a *current namespace*, which is determined by the `namespace` command. The `namespace` command is described in the section on commands that introduce section scopes. Names that are declared within the section scope are added to the current namespace. If the declared name has more than one component, then its namespace is nested within the current namespace; the body of the declaration's current namespace is the nested namespace. Section scopes also include a set of *opened namespaces*, which are namespaces whose contents are in scope without additional qualification. Resolving an identifier to a particular name takes the current namespace and opened namespaces into account. However, protected declarations (that is, those

with the `protected` modifier) are not brought into scope when their namespace is opened. The rules for resolving identifiers into names that take the current namespace and opened namespaces into account are described in the section on identifiers as terms.

▼ *Example: Current Namespace*

Defining an inductive type results in the type's constructors being placed in its namespace, in this case as `HotDrink.coffee`, `HotDrink.tea`, and `HotDrink.cocoa`.

```
inductive HotDrink where
  | coffee
  | tea
  | cocoa
```

Outside the namespace, these names must be qualified unless the namespace is opened:

```
#check HotDrink.tea
```

```
HotDrink.tea : HotDrink
```

```
#check tea
```

```
unknown identifier 'tea'
```

```
section
open HotDrink
#check tea
end
```

```
HotDrink.tea : HotDrink
```

If a function is defined directly inside the `HotDrink` namespace, then the body of the function is elaborated with the current namespace set to `HotDrink`. The constructors are in scope:

```
def HotDrink.ofString? : String → Option HotDrink
  | "coffee" => some coffee
  | "tea" => some tea
  | "cocoa" => some cocoa
  | _ => none
```

Defining another inductive type creates a new namespace:

```
inductive ColdDrink where
  | water
  | juice
```

From within the `HotDrink` namespace, `HotDrink.toString` can be defined without an explicit prefix. Defining a function in the `ColdDrink` namespace requires an explicit `_root_` qualifier to avoid defining `HotDrink.ColdDrink.toString`:

```
namespace HotDrink
```

```
def toString : HotDrink → String
  | coffee => "coffee"
  | tea => "tea"
  | cocoa => "cocoa"

def _root_.ColdDrink.toString : ColdDrink → String
  | .water => "water"
  | .juice => "juice"

end HotDrink
```

The open command opens a namespace, making its contents available in the current section scope. There are many variations on opening namespaces, providing flexibility in managing the local scope.

syntax

Opening Namespaces

The open command is used to open a namespace:

```
command ::= ...
    | open openDecl
```

open declaration

Opening Entire Namespaces

A sequence of one or more identifiers results in each namespace in the sequence being opened:

```
openDecl ::= ...
    | ident ident*
```

Each namespace in the sequence is considered relative to all currently-open namespaces, yielding a set of namespaces. Every namespace in this set is opened before the next namespace in the sequence is processed.

▼ *Example: Opening Nested Namespaces*

Namespaces to be opened are considered relative to the currently-open namespaces. If the same component occurs in different namespace paths, a single open command can be used to open all of them by iteratively bringing each into scope. This example defines names in a variety of namespaces:

```
namespace A -- _root_.A
def a1 := 0
namespace B -- _root_.A.B
```

```
def a2 := 0
namespace C -- _root_.A.B.C
def a3 := 0
end C
end B
end A
namespace B -- _root_.B
def a4 := 0
namespace C -- _root_.B.C
def a5 := 0
end C
end B
namespace C -- _root_.C
def a6 := 0
end C
```

The names are:

- A.a1
- A.B.a2
- A.B.C.a3
- B.a4
- B.C.a5
- C.a6

All six names can be brought into scope with a single iterated open command:

```
section
open A B C
example := [a1, a2, a3, a4, a5, a6]
end
```

If the initial namespace in the command is A.B instead, then neither _root_.A, _root_.B, nor _root_.B.C is opened:

```
section
open A.B C
example := [a1, a2, a3, a4, a5, a6]
end
```

```
unknown identifier 'a1'
unknown identifier 'a4'
unknown identifier 'a5'
```

Opening A.B makes A.B.C visible as C along with _root_.C, so the subsequent C opens both.

open declaration

Hiding Names

A `hiding` declaration specifies a set of names that should *not* be brought into scope. In contrast to opening an entire namespace, the provided identifier must uniquely designate a namespace to be opened.

```
openDecl ::= ...
    | ident hiding ident ident*
```

open declaration

Renaming

A `renaming` declaration allows some names from the opened namespace to be renamed; they are accessible under the new name in the current section scope. The provided identifier must uniquely designate a namespace to be opened.

```
openDecl ::= ...
    | ident renaming (ident → ident),*
```

An ASCII arrow (`->`) may be used instead of the Unicode arrow (→).

open declaration

Restricted Opening

Parentheses indicate that *only* the names listed in the parentheses should be brought into scope.

```
openDecl ::= ...
    | ident (ident ident*)
```

The indicated namespace is added to each currently-opened namespace, and each name is considered in each resulting namespace. All of the listed names must be unambiguous; that is, they must exist in exactly one of the considered namespaces.

open declaration

Scoped Declarations Only

The `scoped` keyword indicates that all scoped attributes, instances, and syntax from the provided namespaces should be opened, while not making any of the names available.

```
openDecl ::= ...
    | scoped ident ident*
```

▼ *Example: Opening Scoped Declarations*

In this example, a scoped notation and a definition are created in the namespace NS:

```
namespace NS
scoped notation "{!{" e "}!}" => (e, e)
def three := 3
end NS
```

Outside of the namespace, the notation is not available:

```
1 def x := {!{ "pear" }!}
```

```
<example>:1:21-1:22: unexpected token '!'; expected '}'
```

An `open scoped` command makes the notation available:

```
open scoped NS
def x := {!{ "pear" }!}
```

However, the name `NS.three` is not in scope:

```
def y := three
```

```
unknown identifier 'three'
```

6.1.2. Exporting Names

Exporting a name makes it available in the current namespace. Unlike a definition, this alias is completely transparent: uses are resolved directly to the original name. Exporting a name to the root namespace makes it available without qualification; the Lean standard library does this for names such as the constructors of `Option` and key type class methods such as `get`.

syntax

Exporting Names

The `export` command adds names from other namespaces to the current namespace, as if they had been declared in it. When the current namespace is opened, these exported names are also brought into scope.

```
command ::= ...
    | export ident (ident*)
```

Internally, exported names are registered as aliases of their targets. From the perspective of the kernel, only the original name exists; the elaborator resolves aliases as part of resolving identifiers to names.

▼ *Example: Exported Names*

The declaration of the inductive type `Veg.Leafy` establishes the constructors `Veg.Leafy.spinach` and `Veg.Leafy.cabbage`:

```
namespace Veg
inductive Leafy where
  | spinach
  | cabbage
export Leafy (spinach)
end Veg
export Veg.Leafy (cabbage)
```

The first `export` command makes `Veg.Leafy.spinach` accessible as `Veg.spinach` because the current namespace is `Veg`. The second makes `Veg.Leafy.cabbage` accessible as `cabbage`, because the current namespace is the root namespace.

6.2. Section Scopes

Many commands have an effect for the current *section scope* (sometimes just called "scope" when clear). Every Lean module has a section scope. Nested scopes are created via the `namespace` and `section` commands, as well as the `in` command combinator.

The following data are tracked in section scopes:

The Current Namespace

The *current namespace* is the namespace into which new declarations will be defined. Additionally, name resolution includes all prefixes of the current namespace in the scope for global names.

Opened Namespaces

When a namespace is *opened*, its names become available without an explicit prefix in the current scope. Additionally, scoped attributes and scoped syntax extensions in namespaces that have been opened are active in the current section scope.

Options

Compiler options are reverted to their original values at the end of the scope in which they were modified.

Section Variables

Section variables are names (or instance implicit parameters) that are automatically added as parameters to definitions. They are also added as universally-quantified assumptions to theorems when they occur in the theorem's statement.

6.2.1. Controlling Section Scopes

The `section` command creates a new section scope, but does not modify the current namespace, opened namespaces, or section variables. Changes made to the section scope are reverted when the section ends. Sections may optionally be named; the `end` command that closes a named section must use the same name. If section names have multiple components (that is, if they contain `.`-separated names), then multiple nested sections are introduced. Section names have no other effect, and are a readability aid.

syntax

Sections

The `section` command creates a section scope that lasts either until an end command or the end of the file.

```
command ::= ...
    | section ident?
```

▼ *Example: Named Section*

The name `english` is defined in the `Greetings` namespace.

```
def Greetings.english := "Hello"
```

Outside its namespace, it cannot be evaluated.

```
#eval english
```

```
unknown identifier 'english'
```

Opening a section allows modifications to the global scope to be contained. This section is named `Greetings`.

```
section Greetings
```

Even though the section name matches the definition's namespace, the name is not in scope because section names are purely for readability and ease of refactoring.

```
#eval english
```

```
unknown identifier 'english'
```

Opening the namespace `Greetings` brings `Greetings.english` as `english`:

```
open Greetings

#eval english
```

```
"Hello"
```

The section's name must be used to close it.

```
end
```

```
Missing name after `end`: Expected the current scope name
`Greetings`

Hint: To end the current scope `Greetings`, specify its name:
  end Greetings
```

```
end Greetings
```

When the section is closed, the effects of the open command are reverted.

```
#eval english
```

```
unknown identifier 'english'
```

The namespace command creates a new section scope. Within this section scope, the current namespace is the name provided in the command, interpreted relative to the current namespace in the surrounding section scope. Like sections, changes made to the section scope are reverted when the namespace's scope ends.

To close a namespace, the **end** command requires a suffix of the current namespace, which is removed. All section scopes introduced by the namespace command that introduced part of that suffix are closed.

syntax

Namespace Declarations

The namespace command modifies the current namespace by appending the provided identifier.

creates a section scope that lasts either until an **end** command or the end of the file.

```
command ::= ...
    | namespace ident
```

syntax

Section and Namespace Terminators

Without an identifier, **end** closes the most recently opened section, which must be anonymous.

```
command ::= ...
    | end
```

With an identifier, it closes the most recently opened section section or namespace. If it is a section, the identifier be a suffix of the concatenated names of the sections opened since the most recent `namespace` command. If it is a namespace, then the identifier must be a suffix of the current namespace's extensions since the most recent `section` that is still open; afterwards, the current namespace will have had this suffix removed.

```
command ::= ...
    | end ident
```

The `end` that closes a `mutual` block is part of the syntax of `mutual`, rather than the `end` command.

▼ *Example: Nesting Namespaces and Sections*

Namespaces and sections may be nested. A single `end` command may close one or more namespaces or one or more sections, but not a mix of the two.

After setting the current namespace to `A.B.C` with two separate commands, `B.C` may be removed with a single `end`:

```
namespace A.B
namespace C
end B.C
```

At this point, the current namespace is `A`.

Next, an anonymous section and the namespace `D.E` are opened:

```
section
namespace D.E
```

At this point, the current namespace is `A.D.E`. An `end` command cannot close all three due to the intervening section:

```
end A.D.E
```

```
Invalid name after `end`: Expected `D.E`, but found `A.D.E`
```

Instead, namespaces and sections must be ended separately.

```
end D.E
end
end A
```

Rather than opening a section for a single command, the `in` combinator can be used to create single-command section scope. The `in` combinator is right-associative, allowing multiple scope modifications to be stacked.

syntax

Local Section Scopes

The `in` command combinator introduces a section scope for a single command.

```
command ::= ...
    | command in
      command
```

▼ *Example: Using `in` for Local Scopes*

The contents of a namespace can be made available for a single command using `in`.

```
def Dessert.cupcake := "delicious"

open Dessert in
#eval cupcake
```

After the single command, the effects of `open` are reverted.

```
#eval cupcake
```

```
unknown identifier 'cupcake'
```

6.2.2. Section Variables

Section variables are parameters that are automatically added to declarations that mention them. This occurs whether or not the option `autoImplicit` is `true`. Section variables may be implicit, strict implicit, or explicit; instance implicit section variables are treated specially.

When the name of a section variable is encountered in a non-theorem declaration, it is added as a parameter. Any instance implicit section variables that mention the variable are also added. If any of the variables that were added depend on other variables, then those variables are added as well; this process is iterated until no more dependencies remain. All section variables are added in the order in which they are declared, before all other parameters. Section variables are added only when they occur in the *statement* of a theorem. Otherwise, modifying the proof of a theorem could change its statement if the proof term made use of a section variable.

Variables are declared using the `variable` command.

syntax

Variable Declarations

```
command ::= ...
    | variable bracketedBinder bracketedBinder*
```

The bracketed binders allowed after `variable` match the syntax used in definition headers.

▼ *Example: Section Variables*

In this section, automatic implicit parameters are disabled, but a number of section variables are defined.

```
section
set_option autoImplicit false
universe u
variable {α : Type u} (xs : List α) [Zero α] [Add α]
```

Because automatic implicit parameters are disabled, the following definition fails:

```
def addAll (lst : List β) : β :=
  lst.foldr (init := 0) (· + ·)
```

```
unknown identifier 'β'
```

On the other hand, not even `xs` needs to be written directly in the definition:

```
def addAll :=
  xs.foldr (init := 0) (· + ·)
```

To add a section variable to a theorem even if it is not explicitly mentioned in the statement, mark the variable with the `include` command. All variables marked for inclusion are added to all theorems. The `omit` command removes the inclusion mark from a variable; it's typically a good idea to use it with `in`.

▼ *Example: Included and Omitted Section Variables*

This section's variables include a predicate as well as everything needed to prove that it holds universally, along with a useless extra assumption.

```
section
variable {p : Nat → Prop}
variable (pZero : p 0) (pStep : ∀ n, p n → p (n + 1))
variable (pFifteen : p 15)
```

However, only `p` is added to this theorem's assumptions, so it cannot be proved.

```
theorem p_all : ∀ n, p n := by
  intro n
  induction n
```

The `include` command causes the additional assumptions to be added unconditionally:

```
include pZero pStep pFifteen

theorem p_all : ∀ n, p n := by
  intro n
  induction n <;> simp [*]
```

Because the spurious assumption *pFifteen* was inserted, Lean issues a warning:

```
automatically included section variable(s) unused in theorem
'p_all':
  pFifteen
consider restructuring your `variable` declarations so that
the variables are not in scope or explicitly omit them:
  omit pFifteen in theorem ...

Note: This linter can be disabled with `set_option
linter.unusedSectionVars false`
```

This can be avoided by using **omit**to remove *pFifteen*:

```
include pZero pStep pFifteen

omit pFifteen in
theorem p_all : ∀ n, p n := by
  intro n
  induction n <;> simp [*]

end
```

7. Definitions

The following commands in Lean are definition-like:

- `def`
- `abbrev`
- `example`
- `theorem`
- `opaque`

All of these commands cause Lean to elaborate a term based on a signature. With the exception of `example`, which discards the result, the resulting expression in Lean's core language is saved for future use in the environment. The `instance` command is described in the section on instance declarations.

7.1. Modifiers

Declarations accept a consistent set of *modifiers*, all of which are optional. Modifiers change some aspect of the declaration's interpretation; for example, they can add documentation or change its scope. The order of modifiers is fixed, but not every kind of declaration accepts every kind of modifier.

syntax

Declaration Modifiers

Modifiers consist of the following, in order, all of which are optional:

1. a documentation comment,
2. a list of attributes,
3. namespace control, specifying whether the resulting name is private or protected,
4. the `noncomputable` keyword, which exempts a definition from compilation,
5. the `unsafe` keyword, and
6. a recursion modifier `partial` or `nonrec`, which disable termination proofs or disallow recursion entirely.

```
declModifiers ::=
    docComment?
    attributes?
    (private | protected | public)?
    noncomputable?
    unsafe?
    (partial | nonrec)?
```

Documentation comments are used to provide in-source API documentation for the declaration that they modify. Documentation comments are not, in fact comments: it is a syntax error to put a documentation comment in a position where it is not processed as documentation. They also occur in positions where some kind of text is required, but string escaping would be onerous, such as the desired messages on the `#guard_msgs` command.

syntax

Documentation Comments

Documentation comments are like ordinary block comments, but they begin with the sequence `/--` rather than `/-`; just like ordinary comments, they are terminated with `-/`.

```
docComment ::=
    /--
```

```
...
-/
```

Attributes are an extensible collection of modifiers that associate additional information with declarations. They are described in a dedicated section.

If a declaration is marked `private`, then it is not accessible outside the module in which it is defined. If it is `protected`, then opening its namespace does not bring it into scope.

Functions marked `noncomputable` are not compiled and cannot be executed. Functions must be noncomputable if they use noncomputable reasoning principles such as the axiom of choice or excluded middle to produce data that is relevant to the answer that they return, or if they use features of Lean that are exempted from code generation for efficiency reasons, such as recursors. Noncomputable functions are very useful for specification and reasoning, even if they cannot be compiled and executed.

The `unsafe` marker exempts a definition from kernel checking and enables it to access features that may undermine Lean's guarantees. It should be used with great care, and only with a thorough understanding of Lean's internals.

7.2. Headers and Signatures

The *header* of a definition or declaration consists of the constant being declared or defined, if relevant, together with its signature. The *signature* of a constant specifies how it can be used. The information present in the signature is more than just the type, including information such as universe level parameters and the default values of its optional parameters. In Lean, signatures are written in a consistent format in different kinds of declarations.

7.2.1. Declaration Names

Most headers begin with a *declaration name*, which is followed by the signature proper: its parameters and the resulting type. A declaration name is a name that may optionally include universe parameters.

syntax

Declaration Names

Declaration names without universe parameters consist of an identifier:

```
declId ::=
    ident
```

Declaration names with universe parameters consist of an identifier followed by a period and one or more universe parameter names in braces:

```
declId ::= ...
    | ident.{ident, ident,*}
```

These universe parameter names are binding occurrences.

Examples do not include declaration names, and names are optional for instance declarations.

7.2.2. Parameters and Types

After the name, if present, is the header's signature. The signature specifies the declaration's parameters and type.

syntax

Declaration Signatures

A signature consists of zero or more parameters, followed by a colon and a type.

```
declSig ::=
    (ident | hole | bracketedBinder)* : term
```

syntax

Optional Signatures

Signatures are often optional. In these cases, parameters may be supplied even if the type is omitted.

```
optDeclSig ::=
    (ident | hole | bracketedBinder)* (: term)?
```

Parameters may have three forms:

- An identifier, which names a parameter but does not provide a type. These parameters' types must be inferred during elaboration.
- An underscore (_), which indicates a parameter that is not accessible by name in the local scope. These parameters' types must also inferred during elaboration.
- A bracketed binder, which may specify every aspect of one or more parameters, including their names, their types, default values, and whether they are explicit, implicit, strictly implicit, or instance-implicit.

7.2.3. Bracketed Parameter Bindings

Parameters other than identifiers or underscores are collectively referred to as *bracketed binders* because every syntactic form for specifying them has some kind of brackets, braces, or parentheses. All bracketed binders specify the type of a parameter, and most include parameter names. The name is optional for instance implicit parameters. Using an underscore (_) instead of a parameter name indicates an anonymous parameter.

syntax

Explicit Parameters

Parenthesized parameters indicate explicit parameters. If more than one identifier or underscore is provided, then all of them become parameters with the same type.

```
bracketedBinder ::=
    ((ident | hole) (ident | hole)* : term)
```

syntax

Optional and Automatic Parameters

Parenthesized parameters with a `:=` assign default values to parameters. Parameters with default values are called *optional parameters*. At a call site, if the parameter is not provided, then the provided term is used to fill it in. Prior parameters in the signature are in scope for the default value, and their values at the call site are substituted into the default value term.

If a tactic script is provided, then the tactics are executed at the call site to synthesize a parameter value; parameters that are filled in via tactics are called *automatic parameters*.

```
bracketedBinder ::= ...
    | ((ident | hole) (ident | hole)* : term := term)
```

syntax

Implicit Parameters

Parameters in curly braces indicate implicit parameters. Unless provided by name at a call site, these parameters are expected to be synthesized via unification at call sites. Implicit parameters are synthesized at all call sites.

```
bracketedBinder ::= ...
    | {(ident | hole) (ident | hole)* : term}
```

syntax

Strict Implicit Parameters

Parameters in double curly braces indicate strict implicit parameters. ⦃ … ⦄ and {{ … }} are equivalent. Like implicit parameters, these parameters are expected to be synthesized via unification at call sites when they are not provided by name. Strict implicit parameters are only synthesized at call sites when subsequent parameters in the signature are also provided.

```
bracketedBinder ::= ...
    | ⦃(ident | hole) (ident | hole)* : term⦄

bracketedBinder ::= ...
    | {{(ident | hole) (ident | hole)* : term}}
```

syntax

Instance Implicit Parameters

Parameters in square brackets indicate instance implicit parameters, which are synthesized at call sites using instance synthesis.

```
bracketedBinder ::= ...
    | [(ident :)? term]
```

The parameters are always in scope in the signature's type, which occurs after the colon. They are also in scope in the declaration's body, while names bound in the type itself are only in scope in the type. Thus, parameter names are used twice:

- As names in the declaration's function type, bound as part of a dependent function type.
- As names in the declaration's body. In function definitions, they are bound by a `fun`.

▼ *Example: Parameter Scope*

The signature of add contains one parameter, n. Additionally, the signature's type is `(k : Nat) → Nat`, which is a function type that includes k. The parameter n is in scope in the function's body, but k is not.

```
def add (n : Nat) : (k : Nat) → Nat
  | 0 => n
  | k' + 1 => 1 + add n k'
```

Like add, the signature of `mustBeEqual` contains one parameter, n. It is in scope both in the type, where it occurs in a proposition, and in the body, where it occurs as part of the message.

```
def mustBeEqual (n : Nat) : (k : Nat) → n = k → String :=
  fun _ =>
    fun
    | rfl => s!"Equal - both are {n}!"
```

The section on function application describes the interpretation of optional, automatic, implicit, and instance implicit parameters in detail.

7.2.4. Automatic Implicit Parameters

By default, otherwise-unbound names that occur in signatures are converted into implicit parameters when possible These parameters are called *automatic implicit parameters*. This is possible when they are not in the function position of an application and when there is sufficient information available in the signature to infer their type and any ordering constraints on them. This process is iterated: if the inferred type for the freshly-inserted implicit parameter has dependencies that are not uniquely determined, then these dependencies are replaced with further implicit parameters.

Implicit parameters that don't correspond to names written in signatures are assigned names akin to those of inaccessible hypotheses in proofs, which cannot be referred to. They show up in signatures with a trailing dagger (`'†'`). This prevents an arbitrary choice of name by Lean from becoming part of the API by being usable as a named argument.

▼ *Example: Automatic Implicit Parameters*

In this definition of `map`, α and β are not explicitly bound. Rather than this being an error, they are converted into implicit parameters. Because they must be types, but nothing constrains their universes, the universe parameters `u` and `v` are also inserted.

```
def map (f : α → β) : (xs : List α) → List β
  | [] => []
  | x :: xs => f x :: map f xs
```

The full signature of `map` is:

```
map.{u, v} {α : Type u} {β : Type v}
  (f : α → β) (xs : List α) :
  List β
```

▼ *Example: No Automatic Implicit Parameters*

In this definition, α and β are not explicitly bound. Because `autoImplicit` is disabled, this is an error:

```
set_option autoImplicit false

def map (f : α → β) : (xs : List α) → List β
```

```
  | [] => []
  | x :: xs => f x :: map f xs
```

```
unknown identifier 'α'
```

```
unknown identifier 'β'
```

The full signature allows the definition to be accepted:

```
set_option autoImplicit false

def map.{u, v} {α : Type u} {β : Type v}
    (f : α → β) :
    (xs : List α) → List β
  | [] => []
  | x :: xs => f x :: map f xs
```

Universe parameters are inserted automatically for parameters without explicit type annotations. The type parameters' universes can be inferred, and the appropriate universe parameters inserted, even when `autoImplicit` is disabled:

```
set_option autoImplicit false

def map {α β} (f : α → β) :
    (xs : List α) → List β
  | [] => []
  | x :: xs => f x :: map f xs
```

▼ *Example: Iterated Automatic Implicit Parameters*

Given a number bounded by *n*, represented by the type `Fin n`, an `AtLeast i` is a natural number paired with a proof that it is at least as large as as `i`.

```
structure AtLeast (i : Fin n) where
  val : Nat
  val_gt_i : val ≥ i.val
```

These numbers can be added:

```
def AtLeast.add (x y : AtLeast i) : AtLeast i :=
  AtLeast.mk (x.val + y.val) <| by
    cases x
    cases y
    dsimp only
    omega
```

The signature of `AtLeast.add` requires multiple rounds of automatic implicit parameter insertion. First, `i` is inserted; but its type depends on the upper bound *n* of `Fin n`. In the second round, *n* is inserted, using a machine-chosen name. Because *n*'s type is `Nat`, which has

no dependencies, the process terminates. The final signature can be seen with `#check`:

```
#check AtLeast.add
```

```
AtLeast.add {n✝ : Nat} {i : Fin n✝} (x y : AtLeast i) :
AtLeast i
```

Automatic implicit parameter insertion takes place after the insertion of parameters due to section variables. Parameters that correspond to section variables have the same name as the corresponding variable, even when they do not correspond to a name written directly in the signature, and disabling automatic implicit parameters has no effect the parameters that correspond to section variables. However, when automatic implicit parameters are enabled, section variable declarations that contain otherwise-unbound variables receive additional section variables that follow the same rules as those for implicit parameters.

Automatic implicit parameters insertion is controlled by two options. By default, automatic implicit parameter insertion is *relaxed*, which means that any unbound identifier may be a candidate for automatic insertion. Setting the option `relaxedAutoImplicit` to `false` disables relaxed mode and causes only identifiers that consist of a single character followed by zero or more digits to be considered for automatic insertion.

option

`relaxedAutoImplicit`

Default value: `true`

When "relaxed" mode is enabled, any atomic nonempty identifier is eligible for auto bound implicit locals (see option `autoImplicit`).

option

`autoImplicit`

Default value: `true`

Unbound local variables in declaration headers become implicit arguments. In "relaxed" mode (default), any atomic identifier is eligible, otherwise only single character followed by numeric digits are eligible. For example, `def f (x : Vector α n) : Vector α n :=` automatically introduces the implicit variables {α n}.

▼ *Example: Relaxed vs Non-Relaxed Automatic Implicit Parameters*

Misspelled identifiers or missing imports can end up as unwanted implicit parameters, as in this example:

```
inductive Answer where
  | yes
```

```
  | maybe
  | no

def select (choices : α × α × α) : Asnwer →  α
  | .yes => choices.1
  | .maybe => choices.2.1
  | .no => choices.2.2
```

The resulting error message states that the argument's type is not a constant, so dot notation cannot be used in the pattern:

```
Invalid dotted identifier notation: The expected type of
`.yes`
  Asnwer
is not of the form `C ...` or `... → C ...` where C is a
constant
```

This is because the signature is:

```
select.{u_1, u_2}
  {α : Type u_1}
  {Asnwer : Sort u_2}
  (choices : α × α × α) :
  Asnwer → α
```

Disabling relaxed automatic implicit parameters makes the error more clear, while still allowing the type to be inserted automatically:

```
set_option relaxedAutoImplicit false

def select (choices : α × α × α) : Asnwer →  α
  | .yes => choices.1
  | .maybe => choices.2.1
  | .no => choices.2.2
```

```
unknown identifier 'Asnwer'
```

Correcting the error allows the definition to be accepted.

```
set_option relaxedAutoImplicit false

def select (choices : α × α × α) : Answer →  α
  | .yes => choices.1
  | .maybe => choices.2.1
  | .no => choices.2.2
```

Turning off automatic implicit parameters entirely leads to the definition being rejected:

```
set_option autoImplicit false

def select (choices : α × α × α) : Answer →  α
  | .yes => choices.1
  | .maybe => choices.2.1
```

```
  | .no => choices.2.2
```

```
unknown identifier 'α'
```

7.3. Definitions

Definitions add a new constant to the global environment as a name that stands for a term. As part of the kernel's definitional equality, this new constant may be replaced via δ-reduction with the term that it stands for. In the elaborator, this replacement is governed by the constant's reducibility. The new constant may be universe polymorphic, in which case occurrences may instantiate it with different universe level parameters.

Function definitions may be recursive. To preserve the consistency of Lean's type theory as a logic, recursive functions must either be opaque to the kernel (e.g. by declaring them `partial`) or proven to terminate with one of the strategies described in the section on recursive definitions.

The headers and bodies of definitions are elaborated together. If the header is incompletely specified (e.g. a parameter's type or the codomain is missing), then the body may provide sufficient information for the elaborator to reconstruct the missing parts. However, instance implicit parameters must be specified in the header or as section variables.

syntax

Definitions

Definitions that use `:=` associate the term on the right-hand side with the constant's name. The term is wrapped in a `fun` for each parameter, and the type is found by binding the parameters in a function type. Definitions with `def` are semireducible.

```
command ::= ...
    | declModifiers
      def declId optDeclSig := term
```

Definitions may use pattern matching. These definitions are desugared to uses of `match`.

```
command ::= ...
    | declModifiers
      def declId optDeclSig
        (| term => term)*
```

Values of structure types, or functions that return them, may be defined by providing values for their fields, following `where`:

```
command ::= ...
    | declModifiers
      def declId optDeclSig where
        structInstField*
```

syntax

Abbreviations

Abbreviations are identical to definitions with `def`, except they are reducible.

```
command ::= ...
    | declModifiers
      abbrev declId optDeclSig := term

command ::= ...
    | declModifiers
      abbrev declId optDeclSig
        (| term => term)*

command ::= ...
    | declModifiers
      abbrev declId optDeclSig where
        structInstField*
```

Opaque constants are defined constants that are not subject to δ-reduction in the kernel. They are useful for specifying the existence of some function. Unlike axioms, opaque declarations can only be used for types that are inhabited, so they do not risk introducing inconsistency. Also unlike axioms, the inhabitant of the type is used in compiled code. The **`implemented_by`** attribute can be used to instruct the compiler to emit a call to some other function as the compilation of an opaque constant.

syntax

Opaque Constants

Opaque definitions with right-hand sides are elaborated like other definitions. This demonstrates that the type is inhabited; the inhabitant plays no further role.

```
command ::= ...
    | declModifiers
      opaque declId declSig := term
```

Opaque constants may also be specified without right-hand sides. The elaborator fills in the right-hand side by synthesizing an instance of `Inhabited`, or `Nonempty` if that fails.

```
command ::= ...
    | declModifiers
      opaque declId declSig
```

7.4. Theorems

Because propositions are types whose inhabitants count as proofs, theorems and definitions are technically very similar. However, because their use cases are quite different, they differ in many details:

- The theorem statement must be a proposition. The types of definitions may inhabit any universe.
- A theorem's header (that is, the theorem statement) is completely elaborated before the body is elaborated. Section variables only become parameters to the theorem if they (or their dependents) are mentioned in the header. This prevents changes to a proof from unintentionally changing the the theorem statement.
- Theorems are irreducible by default. Because all proofs of the same proposition are definitionally equal, there few reasons to unfold a theorem.

Theorems may be recursive, subject to the same conditions as recursive function definitions. However, it is more common to use tactics such as `induction` or `fun_induction` instead.

syntax

Theorems

The syntax of theorems is like that of definitions, except the codomain (that is, the theorem statement) in the signature is mandatory.

```
command ::= ...
    | declModifiers
      theorem declId declSig := term

command ::= ...
    | declModifiers
      theorem declId declSig
        (| term => term)*

command ::= ...
    | declModifiers
      theorem declId declSig where
        structInstField*
```

7.5. Example Declarations

An example is an anonymous definition that is elaborated and then discarded. Examples are useful for incremental testing during development and to make it easier to understand a file.

syntax

Examples

```
command ::= ...
    | declModifiers
      example optDeclSig := term

command ::= ...
    | declModifiers
      example optDeclSig
        (| term => term)*

command ::= ...
    | declModifiers
      example optDeclSig where
        structInstField*
```

7.6. Recursive Definitions

Allowing arbitrary recursive function definitions would make Lean's logic inconsistent. General recursion makes it possible to write circular proofs: "proposition P is true because proposition P is true". Outside of proofs, an infinite loop could be assigned the type `Empty`, which can be used with `nomatch` or `Empty.rec` to prove any theorem.

Banning recursive function definitions outright would render Lean far less useful: inductive types are key to defining both predicates and data, and they have a recursive structure. Furthermore, most useful recursive functions do not threaten soundness, and infinite loops usually indicate mistakes in definitions rather than intentional behavior. Instead of banning recursive functions, Lean requires that each recursive function is defined safely. While elaborating recursive definitions, the Lean elaborator also produces a justification that the function being defined is safe. (i) The section on the elaborator's output in the overview of elaboration contextualizes the elaboration of recursive definitions in the overall context of the elaborator.

There are five main kinds of recursive functions that can be defined:

Structurally recursive functions

Structurally recursive functions take an argument such that the function makes recursive calls only on strict sub-components of said argument. (i) Strictly speaking, arguments whose types are indexed families are grouped together with their indices, with the whole collection considered as a unit. The elaborator translates the recursion into uses of the argument's recursor. Because every type-correct use of a recursor is guaranteed to avoid infinite regress, this translation is evidence that the function terminates. Applications of functions defined via recursors are definitionally equal to the result of the recursion, and are typically relatively efficient inside the kernel.

Recursion over well-founded relations

Many functions are also difficult to convert to structural recursion; for instance, a function may terminate because the difference between an array index and the size of the array decreases as the index increases, but `Nat.rec` isn't applicable because the index that increases is the function's argument. Here, there is a measure of termination that decreases at each recursive call, but the measure is not itself an argument to the function. In these cases, well-founded recursion can be used to define the function. Well-founded recursion is a technique for systematically transforming recursive functions with a decreasing measure into recursive functions over proofs that every sequence of reductions to the measure eventually terminates at a minimum. Applications of functions defined via well-founded recursion are not necessarily definitionally equal to their return values, but this equality can be proved as a proposition. Even when definitional equalities exist, these functions are frequently slow to compute with because they require reducing proof terms that are often very large.

Recursive functions as partial fixpoints

The definition of a function can be understood as an equation that specifies its behavior. In certain cases, the existence of a function that satisfies this specification can be proven even when the recursive function does not necessarily terminate for all inputs. This strategy is even applicable in some cases where the function definition does not necessarily terminate for all inputs. These partial functions emerge as fixed points of these equations are called *partial fixpoints*.

In particular, any function whose return type is in certain monads (e.g. `Option`) can be defined using this strategy. Lean generates additional partial correctness theorems for these monadic functions. As with well-founded recursion, applications of functions defined as partial fixpoints are not definitionally equal to their return values, but Lean generates theorems that propositionally equate the function to its unfolding and to the reduction behavior specified in its definition.

Partial functions with nonempty codomains

For many applications, it's not important to reason about the implementation of certain functions. A recursive function might be used only as part of the implementation of proof automation steps, or it might be an ordinary program that will never be formally proved correct. In these cases, the Lean kernel does not need either definitional or propositional equalities to hold for the definition; it suffices that soundness is maintained. Functions marked `partial` are treated as opaque constants by the kernel and are neither unfolded nor reduced. All that is required for soundness is that their return type is inhabited. Partial functions may still be used in compiled code as usual, and they may appear in propositions and proofs; their equational theory in Lean's logic is simply very weak.

Unsafe recursive definitions

Unsafe definitions have none of the restrictions of partial definitions. They may freely make use of general recursion, and they may use features of Lean that break assumptions about its equational theory, such as primitives for casting (`unsafeCast`), checking pointer equality (`ptrAddrUnsafe`), and observing reference counts (`isExclusiveUnsafe`). However, any declaration that refers to an unsafe definition must itself be marked `unsafe`, making it clear when logical soundness is not guaranteed. Unsafe operations can be used to replace the

implementations of other functions with more efficient variants in compiled code, while the kernel still uses the original definition. The replaced function may be opaque, which results in the function name having a trivial equational theory in the logic, or it may be an ordinary function, in which case the function is used in the logic. Use this feature with care: logical soundness is not at risk, but the behavior of programs written in Lean may diverge from their verified logical models if the unsafe implementation is incorrect.

As described in the overview of the elaborator's output, elaboration of recursive functions proceeds in two phases:

1. The definition is elaborated as if Lean's core type theory had recursive definitions. Aside from using recursion, this provisional definition is fully elaborated. The compiler generates code from these provisional definitions.
2. A termination analysis attempts to use the four techniques to justify the function to Lean's kernel. If the definition is marked `unsafe` or `partial`, then that technique is used. If an explicit `termination_by` clause is present, then the indicated technique is the only one attempted. If there is no such clause, then the elaborator performs a search, testing each parameter to the function as a candidate for structural recursion, and attempting to find a measure with a well-founded relation that decreases at each recursive call.

This section describes the rules that govern recursive functions. After a description of mutual recursion, each of the five kinds of recursive definitions is specified, along with the tradeoffs between reasoning power and flexibility that go along with each.

7.6.1. Mutual Recursion

Just as a recursive definition is one that mentions the name being defined in the body of the definition, *mutually recursive* definitions are definitions that may be recursive or mention one another. To use mutual recursion between multiple declarations, they must be placed in a mutual block.

syntax

Mutual Declaration Blocks

The general syntax for mutual recursion is:

```
command ::= ...
    | mutual
        declaration*
      end
```

where the declarations must be definitions or theorems.

The declarations in a mutual block are not in scope in each others' signatures, but they are in scope in each others' bodies. Even though the names are not in scope in signatures, they will not be inserted as auto-bound implicit parameters.

▼ *Example: Mutual Block Scope*

Names defined in a mutual block are not in scope in each others' signatures.

```
mutual
  abbrev NaturalNum : Type := Nat
  def n : NaturalNum := 5
end
```

```
unknown identifier 'NaturalNum'
```

Without the mutual block, the definition succeeds:

```
abbrev NaturalNum : Type := Nat
def n : NaturalNum := 5
```

▼ *Example: Mutual Block Scope and Automatic Implicit Parameters*

Names defined in a mutual block are not in scope in each others' signatures. Nonetheless, they cannot be used as automatic implicit parameters:

```
mutual
  abbrev α : Type := Nat
  def identity (x : α) : α := x
end
```

```
unknown identifier 'α'
```

With a different name, the implicit parameter is automatically added:

```
mutual
  abbrev α : Type := Nat
  def identity (x : β) : β := x
end
```

Elaborating recursive definitions always occurs at the granularity of mutual blocks, as if there were a singleton mutual block around every declaration that is not itself part of such a block. Local definitions introduced via `let rec` and `where` are lifted out of their context, introducing parameters for captured free variables as necessary, and treated as if they were separate definitions within the `mutual` block as well. Thus, helpers defined in a `where` block may use mutual recursion both with one another and with the definition in which they occur, but they may not mention each other in their type signatures.

After the first step of elaboration, in which definitions are still recursive, and before translating recursion using the techniques above, Lean identifies the actually (mutually) recursive cliques among the definitions in the mutual block and processes them separately and in dependency order.

7.6.2. Structural Recursion

Structurally recursive functions are those in which each recursive call is on a structurally smaller term than the argument. The same parameter must decrease in all recursive calls; this parameter is called the *decreasing parameter*. Structural recursion is stronger than the primitive recursion that recursors provide, because the recursive call can use more deeply nested sub-terms of the argument, rather than only an immediate sub-term. The constructions used to implement structural recursion are, however, implemented using the recursor; these helper constructions are described in the section on inductive types.

The rules that govern structural recursion are fundamentally *syntactic* in nature. There are many recursive definitions that exhibit structurally recursive computational behavior, but which are not accepted by these rules; this is a fundamental consequence of the analysis being fully automatic. Well-founded recursion provides a semantic approach to demonstrating termination that can be used in situations where a recursive function is not structurally recursive, but it can also be used when a function that computes according to structural recursion doesn't satisfy the syntactic requirements.

▼ *Example: Structural Recursion vs Subtraction*

The function `countdown` is structurally recursive. The parameter *n* was matched against the pattern *n'* `+ 1`, which means that *n'* is a direct subterm of *n* in the second branch of the pattern match:

```
def countdown (n : Nat) : List Nat :=
  match n with
  | 0 => []
  | n' + 1 => n' :: countdown n'
```

Replacing pattern matching with an equivalent Boolean test and subtraction results in an error:

```
def countdown' (n : Nat) : List Nat :=
  if n == 0 then []
  else
    let n' := n - 1
    n' :: countdown' n'
```

```
fail to show termination for
  countdown'
with errors
failed to infer structural recursion:
Cannot use parameter n:
  failed to eliminate recursive application
    countdown' n'


failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded
```

```
relation
  - Use `decreasing_by` to specify your own tactic for
discharging this kind of goal
n : Nat
h✝ : ¬(n == 0) = true
n' : Nat := n - 1
⊢ n - 1 < n
```

This is because there was no pattern matching on the parameter *n*. While this function indeed terminates, the argument that it does so is based on properties of if, the equality test, and subtraction, rather than being a generic feature of `Nat` being an inductive type. These arguments are expressed using well-founded recursion, and a slight change to the function definition allows Lean's automatic support for well-founded recursion to construct an alternative termination proof. This version branches on the decidability of propositional equality for `Nat` rather than the result of a Boolean equality test:

```
def countdown' (n : Nat) : List Nat :=
  if n = 0 then []
  else
    let n' := n - 1
    n' :: countdown' n'
```

Here, Lean's automation automatically constructs a termination proof from facts about propositional equality and subtraction. It uses well-founded recursion rather than structural recursion behind the scenes.

Structural recursion may be used explicitly or automatically. With explicit structural recursion, the function definition declares which parameter is the decreasing parameter. If no termination strategy is explicitly declared, Lean performs a search for a decreasing parameter as well as a decreasing measure for use with well-founded recursion. Explicitly annotating structural recursion has the following benefits:

- It can speed up elaboration, because no search occurs.
- It documents the termination argument for readers.
- In situations where structural recursion is explicitly desired, it prevents the accidental use of well-founded recursion.

7.6.2.1. Explicit Structural Recursion

To explicitly use structural recursion, a function or theorem definition can be annotated with a `termination_by structural` clause that specifies the decreasing parameter. The decreasing parameter may be a reference to a parameter named in the signature. When the signature specifies a function type, the decreasing parameter may additionally be a parameter not named in the signature; in this case, names for the remaining parameters may be introduced by writing them before an arrow (`=>`).

▼ *Example: Specifying Decreasing Parameters*

When the decreasing parameter is a named parameter to the function, it can be specified by referring to its name.

```
def half (n : Nat) : Nat :=
  match n with
  | 0 | 1 => 0
  | n + 2 => half n + 1
termination_by structural n
```

When the decreasing parameter is not named in the signature, a name can be introduced locally in the `termination_by` clause.

```
def half : Nat → Nat
  | 0 | 1 => 0
  | n + 2 => half n + 1
termination_by structural n => n
```

syntax

Explicit Structural Recursion

The `termination_by structural` clause introduces a decreasing parameter.

```
terminationBy ::= ...
    | termination_by structural (ident* =>)? term
```

The identifiers before the optional => can bring function parameters into scope that are not already bound in the declaration header, and the mandatory term must indicate one of the function's parameters, whether introduced in the header or locally in the clause.

The decreasing parameter must satisfy the following conditions:

- Its type must be an inductive type.
- If its type is an indexed family, then all indices must be parameters of the function.
- If the inductive or indexed family of the decreasing parameter has data type parameters, then these data type parameters may themselves only depend on function parameters that are part of the fixed prefix.

A *fixed parameter* is a function parameter that is passed unmodified in all recursive calls and is not an index of the recursive parameter's type. The *fixed prefix* is the longest prefix of the function's parameters in which all are fixed.

▼ *Example: Ineligible decreasing parameters*

The decreasing parameter's type must be an inductive type. In `notInductive`, a function is specified as the decreasing parameter:

```
def notInductive (x : Nat → Nat) : Nat :=
  notInductive (fun n => x (n+1))
termination_by structural x
```

```
cannot use specified measure for structural recursion:
  its type is not an inductive
```

If the decreasing parameter is an indexed family, all the indices must be variables. In `constantIndex`, the indexed family `Fin'` is instead applied to a constant value:

```
inductive Fin' : Nat → Type where
  | zero : Fin' (n+1)
  | succ : Fin' n → Fin' (n+1)

def constantIndex (x : Fin' 100) : Nat := constantIndex ⋯.zero
termination_by structural x
```

```
cannot use specified measure for structural recursion:
  its type Fin' is an inductive family and indices are not
variables
    Fin' 100
```

The parameters of the decreasing parameter's type must not depend on function parameters that come after varying parameters or indices. In `afterVarying`, the fixed prefix is empty, because the first parameter n varies, so p is not part of the fixed prefix:

```
inductive WithParam' (p : Nat) : Nat → Type where
  | zero : WithParam' p (n+1)
  | succ : WithParam' p n → WithParam' p (n+1)

def afterVarying (n : Nat) (p : Nat) (x : WithParam' p n) : Nat
:=
  afterVarying (n+1) p ⋯.zero
termination_by structural x
```

```
failed to infer structural recursion:
Cannot use parameter x:
  failed to eliminate recursive application
    afterVarying (n + 1) p WithParam'.zero
```

Furthermore, every recursive call of the functions must be on a *strict sub-term* of the decreasing parameter.

- The decreasing parameter itself is a sub-term, but not a strict sub-term.

- If a sub-term is the discriminant of a `match` expression or other pattern-matching syntax, the pattern that matches the discriminant is a sub-term in the right-hand side of each match alternative. In particular, the rules of match generalization are used to connect the discriminant to the occurrences of the pattern term in the right-hand side; thus, it respects definitional equality. The pattern is a *strict* sub-term if and only if the discriminant is a strict sub-term.
- If a sub-term is a constructor applied to arguments, then its recursive arguments are strict sub-terms.

▼ *Example: Nested Patterns and Sub-Terms*

In the following example, the decreasing parameter *n* is matched against the nested pattern `.succ (.succ n)`. Therefore `.succ (.succ n)` is a (non-strict) sub-term of *n*, and consequently both *n* and `.succ n` are strict sub-terms, and the definition is accepted.

```
def fib : Nat → Nat
  | 0 | 1 => 1
  | .succ (.succ n) =>  fib n + fib (.succ n)
termination_by structural n => n
```

For clarity, this example uses `.succ n` and `.succ (.succ n)` instead of the equivalent `Nat`-specific *n*+1 and *n*+2.

▼ *Example: Matching on Complex Expressions Can Prevent Elaboration*

In the following example, the decreasing parameter *n* is not directly the discriminant of the `match` expression. Therefore, *n'* is not considered a sub-term of *n*.

```
def half (n : Nat) : Nat :=
  match Option.some n with
  | .some (n' + 2) => half n' + 1
  | _ => 0
termination_by structural n
```

```
failed to infer structural recursion:
Cannot use parameter n:
  failed to eliminate recursive application
    half n'
```

Using well-founded recursion, and explicitly connecting the discriminant to the pattern of the match, this definition can be accepted.

```
def half (n : Nat) : Nat :=
  match h : Option.some n with
  | .some (n' + 2) => half n' + 1
  | _ => 0
termination_by n
```

```
  decreasing_by simp_all; omega
```

Similarly, the following example fails: although `xs.tail` would reduce to a strict sub-term of `xs`, this is not visible to Lean according to the rules above. In particular, `xs.tail` is not definitionally equal to a strict sub-term of `xs`.

```
def listLen : List α → Nat
  | [] => 0
  | xs => listLen xs.tail + 1
termination_by structural xs => xs
```

▼ *Example: Simultaneous Matching vs Matching Pairs for Structural Recursion*

An important consequence of the strategies that are used to prove termination is that **simultaneous matching of two discriminants is not equivalent to matching a pair**. Simultaneous matching maintains the connection between the discriminants and the patterns, allowing the pattern matching to refine the types of the assumptions in the local context as well as the expected type of the `match`. Essentially, the elaboration rules for `match` treat the discriminants specially, and changing discriminants in a way that preserves the run-time meaning of a program does not necessarily preserve the compile-time meaning.

This function that finds the minimum of two natural numbers is defined by structural recursion on its first parameter:

```
def min' (n k : Nat) : Nat :=
  match n, k with
  | 0, _ => 0
  | _, 0 => 0
  | n' + 1, k' + 1 => min' n' k' + 1
termination_by structural n
```

Replacing the simultaneous pattern match on both parameters with a match on a pair causes termination analysis to fail:

```
def min' (n k : Nat) : Nat :=
  match (n, k) with
  | (0, _) => 0
  | (_, 0) => 0
  | (n' + 1, k' + 1) => min' n' k' + 1
termination_by structural n
```

```
failed to infer structural recursion:
Cannot use parameter n:
  failed to eliminate recursive application
    min' n' k'
```

This is because the analysis only considers direct pattern matching on parameters when matching recursive calls to strictly-smaller argument values. Wrapping the discriminants in a pair breaks the connection.

▼ *Example: Structural Recursion Under Pairs*

This function that finds the minimum of the two components of a pair can't be elaborated via structural recursion.

```
def min' (nk : Nat × Nat) : Nat :=
  match nk with
  | (0, _) => 0
  | (_, 0) => 0
  | (n' + 1, k' + 1) => min' (n', k') + 1
termination_by structural nk
```

```
failed to infer structural recursion:
Cannot use parameter nk:
  the type Nat × Nat does not have a `.brecOn` recursor
```

This is because the parameter's type, Prod, is not recursive. Thus, its constructor has no recursive parameters that can be exposed by pattern matching.

This definition is accepted using well-founded recursion, however:

```
def min' (nk : Nat × Nat) : Nat :=
  match nk with
  | (0, _) => 0
  | (_, 0) => 0
  | (n' + 1, k' + 1) => min' (n', k') + 1
termination_by nk
```

▼ *Example: Structural Recursion and Definitional Equality*

Even though the recursive occurrence of `countdown` is applied to a term that is not a strict sub-term of the decreasing parameter, the following definition is accepted:

```
def countdown (n : Nat) : List Nat :=
  match n with
  | 0 => []
  | n' + 1 => n' :: countdown (n' + 0)
termination_by structural n
```

This is because `n' + 0` is definitionally equal to `n'`, which is a strict sub-term of `n`. Sub-terms that result from pattern matching are connected to the discriminant using the rules for match generalization, which respect definitional equality.

In `countdown'`, the recursive occurrence is applied to `0 + n'`, which is not definitionally equal to `n'` because addition on natural numbers is structurally recursive in its second parameter:

```
def countdown' (n : Nat) : List Nat :=
  match n with
  | 0 => []
  | n' + 1 => n' :: countdown' (0 + n')
termination_by structural n
```

```
failed to infer structural recursion:
Cannot use parameter n:
  failed to eliminate recursive application
    countdown' (0 + n')
```

7.6.2.2. Mutual Structural Recursion

Lean supports the definition of mutually recursive functions using structural recursion. Mutual recursion may be introduced using a mutual block, but it also results from `let rec` expressions and `where` blocks. The rules for mutual structural recursion are applied to a group of actually mutually recursive, lifted definitions, that results from the elaboration steps for mutual groups. If every function in the mutual group has a `termination_by structural` annotation indicating that function's decreasing argument, then structural recursion is used to translate the definitions.

The requirements on the decreasing argument above are extended:

- All the types of all the decreasing arguments must be from the same inductive type, or more generally from the same mutual group of inductive types.
- The parameters of the decreasing parameter's types must be the same for all functions, and may depend only on the *common* fixed prefix of function arguments.

The functions do not have to be in a one-to-one correspondence to the mutual inductive types. Multiple functions can have a decreasing argument of the same type, and not all types that are mutually recursive with the decreasing argument need have a corresponding function.

▼ *Example: Mutual Structural Recursion Over Non-Mutual Types*

The following example demonstrates mutual recursion over a non-mutual inductive data type:

```
mutual
  def even : Nat → Prop
    | 0 => True
    | n+1 => odd n
  termination_by structural n => n

  def odd : Nat → Prop
    | 0 => False
    | n+1 => even n
```

```
  termination_by structural n => n
end
```

▼ *Example: Mutual Structural Recursion Over Mutual Types*

The following example demonstrates recursion over mutually inductive types. The functions `Exp.size` and `App.size` are mutually recursive.

```
mutual
  inductive Exp where
    | var : String → Exp
    | app : App → Exp

  inductive App where
    | fn : String → App
    | app : App → Exp → App
end

mutual
  def Exp.size : Exp → Nat
    | .var _ => 1
    | .app a => a.size
  termination_by structural e => e

  def App.size : App → Nat
    | .fn _ => 1
    | .app a e => a.size + e.size + 1
  termination_by structural a => a
end
```

The definition of `App.numArgs` is structurally recursive over type `App`. It demonstrates that not all inductive types in the mutual group need to be handled.

```
def App.numArgs : App → Nat
  | .fn _ => 0
  | .app a _ => a.numArgs + 1
termination_by structural a => a
```

7.6.2.3. Inferring Structural Recursion

If no `termination_by` clauses are present in a recursive or mutually recursive function definition, then Lean attempts to infer a suitable structurally decreasing argument, effectively by trying all suitable parameters in sequence. If this search fails, Lean then attempts to infer well-founded recursion.

For mutually recursive functions, all combinations of parameters are tried, up to a limit to avoid combinatorial explosion. If only some of the mutually recursive functions have `termination_by structural` clauses, then only those parameters are considered, while for the other functions all parameters are considered for structural recursion.

A `termination_by?` clause causes the inferred termination annotation to be shown. It can be automatically added to the source file using the offered suggestion or code action.

▼ *Example: Inferred Termination Annotations*

Lean automatically infers that the function `half` is structurally recursive. The `termination_by?` clause causes the inferred termination annotation to be displayed, and it can be automatically added to the source file with a single click.

```
def half : Nat → Nat
  | 0 | 1 => 0
  | n + 2 => half n + 1
termination_by?
```

```
Try this: termination_by structural x => x
```

7.6.2.4. Elaboration Using Course-of-Values Recursion

In this section, the construction used to elaborate structurally recursive functions is explained in more detail. This elaboration uses the below and brecOn constructions that are automatically generated from inductive types' recursors.

▼ *Example: Recursion vs Recursors*

Addition of natural numbers can be defined via recursion on the second argument. This function is straightforwardly structurally recursive.

```
def add (n : Nat) : Nat → Nat
  | .zero => n
  | .succ k => .succ (add n k)
```

Defined using `Nat.rec`, it is much further from the notations that most people are used to.

```
def add' (n : Nat) :=
  Nat.rec (motive := fun _ => Nat)
    n
    (fun k soFar => .succ soFar)
```

Structural recursive calls made on data that isn't the immediate child of the function parameter requires either creativity or a complex yet systematic encoding.

```
def half : Nat → Nat
  | 0 | 1 => 0
```

```
  | n + 2 => half n + 1
```

One way to think about this function is as a structural recursion that flips a bit at each call, only incrementing the result when the bit is set.

```
def helper : Nat → Bool → Nat :=
  Nat.rec (motive := fun _ => Bool → Nat)
    (fun _ => 0)
    (fun _ soFar =>
      fun b =>
        (if b then Nat.succ else id) (soFar !b))

def half' (n : Nat) : Nat := helper n false

#eval [0, 1, 2, 3, 4, 5, 6, 7, 8].map half'
```

```
[0, 0, 1, 1, 2, 2, 3, 3, 4]
```

Instead of creativity, a general technique called course-of-values recursion can be used. Course-of-values recursion uses helpers that can be systematically derived for every inductive type, defined in terms of the recursor; Lean derives them automatically. For every `Nat` *n*, the type `n.below (motive := mot)` provides a value of type `mot k` for all `k < n`, represented as an iterated dependent pair type. The course-of-values recursor `Nat.brecOn` allows a function to use the result for any smaller `Nat`. Using it to define the function is inconvenient:

```
noncomputable def half'' (n : Nat) : Nat :=
  Nat.brecOn n (motive := fun _ => Nat)
    fun k soFar =>
      match k, soFar with
      | 0, _ | 1, _ => 0
      | _ + 2, ⟨_, ⟨h, _⟩⟩ => h + 1
```

The function is marked `noncomputable` because the compiler doesn't support generating code for course-of-values recursion, which is intended for reasoning rather that efficient code. The kernel can still be used to test the function, however:

```
#reduce [0,1,2,3,4,5,6,7,8].map half''
```

```
[0, 0, 1, 1, 2, 2, 3, 3, 4]
```

The dependent pattern matching in the body of `half''` can also be encoded using recursors (specifically, `Nat.casesOn`), if necessary:

```
noncomputable def half''' (n : Nat) : Nat :=
  n.brecOn (motive := fun _ => Nat)
    fun k =>
      k.casesOn
        (motive :=
          fun k' =>
            (k'.below (motive := fun _ => Nat)) →
            Nat)
        (fun _ => 0)
```

```
          (fun k' =>
            k'.casesOn
              (motive :=
                fun k'' =>
                  (k''.succ.below (motive := fun _ => Nat)) →
                  Nat)
              (fun _ => 0)
              (fun _ soFar => soFar.2.1.succ))
```

This definition still works.

```
#reduce [0,1,2,3,4,5,6,7,8].map half''
```

```
[0, 0, 1, 1, 2, 2, 3, 3, 4]
```

However, it is now far from the original definition and it has become difficult for most people to understand. Recursors are an excellent logical foundation, but not an easy way to write programs or proofs.

The structural recursion analysis attempts to translate the recursive pre-definition into a use of the appropriate structural recursion constructions. At this step, pattern matching has already been translated into the use of matcher functions; these are treated specially by the termination checker. Next, for each group of parameters, a translation using `brecOn` is attempted.

▼ *Example: Course-of-Values Tables*

This definition is equivalent to `List.below`:

```
def List.below' {α : Type u} {motive : List α → Sort u} :
    List α → Sort (max (u + 1) u)
  | [] => PUnit
  | _ :: xs => motive xs ×' xs.below' (motive := motive)
```

In other words, for a given motive, `List.below'` is a type that contains a realization of the motive for all suffixes of the list.

More recursive arguments require further nested iterations of the product type. For instance, binary trees have two recursive occurrences.

```
inductive Tree (α : Type u) : Type u where
  | leaf
  | branch (left : Tree α) (val : α) (right : Tree α)
```

Its corresponding course-of-values table contains the realizations of the motive for all subtrees:

```
def Tree.below' {α : Type u} {motive : Tree α → Sort u} :
    Tree α → Sort (max (u + 1) u)
  | .leaf => PUnit
  | .branch left _val right =>
```

```
    (motive left ×' left.below' (motive := motive)) ×'
    (motive right ×' right.below' (motive := motive))
```

For both lists and trees, the `brecOn` operator expects just a single case, rather than one per constructor. This case accepts a list or tree along with a table of results for all smaller values; from this, it should satisfy the motive for the provided value. Dependent case analysis of the provided value automatically refines the type of the memo table, providing everything needed.

The following definitions are equivalent to `List.brecOn` and `Tree.brecOn`, respectively. The primitive recursive helpers `List.brecOnTable` and `Tree.brecOnTable` compute the course-of-values tables along with the final results, and the actual definitions of the `brecOn` operators simply project out the result.

```
def List.brecOnTable {α : Type u}
    {motive : List α → Sort u}
    (xs : List α)
    (step :
      (ys : List α) →
      ys.below' (motive := motive) →
      motive ys) :
    motive xs ×' xs.below' (motive := motive) :=
  match xs with
  | [] => ⟨step [] PUnit.unit, PUnit.unit⟩
  | x :: xs =>
    let res := xs.brecOnTable (motive := motive) step
    let val := step (x :: xs) res
    ⟨val, res⟩

def Tree.brecOnTable {α : Type u}
    {motive : Tree α → Sort u}
    (t : Tree α)
    (step :
      (ys : Tree α) →
      ys.below' (motive := motive) →
      motive ys) :
    motive t ×' t.below' (motive := motive) :=
  match t with
  | .leaf => ⟨step .leaf PUnit.unit, PUnit.unit⟩
  | .branch left val right =>
    let resLeft := left.brecOnTable (motive := motive) step
    let resRight := right.brecOnTable (motive := motive) step
    let branchRes := ⟨resLeft, resRight⟩
    let val := step (.branch left val right) branchRes
    ⟨val, branchRes⟩

def List.brecOn' {α : Type u}
    {motive : List α → Sort u}
    (xs : List α)
    (step :
      (ys : List α) →
```

```
      ys.below' (motive := motive) →
      motive ys) :
    motive xs :=
  (xs.brecOnTable (motive := motive) step).1

def Tree.brecOn' {α : Type u}
    {motive : Tree α → Sort u}
    (t : Tree α)
    (step :
      (ys : Tree α) →
      ys.below' (motive := motive) →
      motive ys) :
    motive t :=
  (t.brecOnTable (motive := motive) step).1
```

The `below` construction is a mapping from each value of a type to the results of some function call on *all* smaller values; it can be understood as a memoization table that already contains the results for all smaller values. The notion of "smaller value" that is expressed in the `below` construction corresponds directly to the definition of strict sub-terms.

Recursors expect an argument for each of the inductive type's constructors; these arguments are called with the constructor's arguments (and the result of recursion on recursive parameters) during ι-reduction. The course-of-values recursion operator `brecOn`, on the other hand, expects just a single case that covers all constructors at once. This case is provided with a value and a `below` table that contains the results of recursion on all values smaller than the given value; it should use the contents of the table to satisfy the motive for the provided value. If the function is structurally recursive over a given parameter (or parameter group), then the results of all recursive calls will be present in this table already.

When the body of the recursive function is transformed into an invocation of `brecOn` on one of the function's parameters, the parameter and its course-of-values table are in scope. The analysis traverses the body of the function, looking for recursive calls. If the parameter is matched against, then its occurrences in the local context are generalized and then instantiated with the pattern; this is also true for the type of the course-of-values table. Typically, this pattern matching results in the type of the course-of-values table becoming more specific, which gives access to the recursive results for smaller values. This generalization process implements the rule that patterns are sub-terms of match discriminants. When an recursive occurrence of the function is detected, the course-of-values table is consulted to see whether it contains a result for the argument being checked. If so, the recursive call can be replaced with a projection from the table. If not, then the parameter in question doesn't support structural recursion.

▼ *Example: Elaboration Walkthrough*

The first step in walking through the elaboration of `half` is to manually desugar it to a simpler form. This doesn't match the way Lean works, but its output is much easier to read when there are fewer `OfNat` instances present. This readable definition:

```
def half : Nat → Nat
  | 0 | 1 => 0
  | n + 2 => half n + 1
```

can be rewritten to this somewhat lower-level version:

```
def half : Nat → Nat
  | .zero | .succ .zero => .zero
  | .succ (.succ n) => half n |>.succ
```

The elaborator begins by elaborating a pre-definition in which recursion is still present but the definition is otherwise in Lean's core type theory. Turning on the compiler's tracing of pre-definitions, as well as making the pretty printer more explicit, makes the resulting pre-definition visible:

```
set_option trace.Elab.definition.body true in
set_option pp.all true in

def half : Nat → Nat
  | .zero | .succ .zero => .zero
  | .succ (.succ n) => half n |>.succ
```

The returned trace message is:

```
[Elab.definition.body] half : Nat → Nat :=
    fun (x : Nat) =>
      half.match_1.{1} (fun (x : Nat) => Nat) x
        (fun (_ : Unit) => Nat.zero)
        (fun (_ : Unit) => Nat.zero)
        fun (n : Nat) => Nat.succ (half n)
```

The auxiliary match function's definition is:

```
#print half.match_1
```

```
def half.match_1.{u_1} : (motive : Nat → Sort u_1) →
  (x : Nat) → (Unit → motive Nat.zero) → (Unit → motive 1) →
((n : Nat) → motive n.succ.succ) → motive x :=
fun motive x h_1 h_2 h_3 => Nat.casesOn x (h_1 ()) fun n =>
Nat.casesOn n (h_2 ()) fun n => h_3 n
```

Formatted more readably, this definition is:

```
def half.match_1'.{u} :
    (motive : Nat → Sort u) → (x : Nat) →
    (Unit → motive Nat.zero) → (Unit → motive 1) →
```

```
    ((n : Nat) → motive n.succ.succ) →
    motive x :=
  fun motive x h_1 h_2 h_3 =>
    Nat.casesOn x (h_1 ()) fun n =>
      Nat.casesOn n (h_2 ()) fun n =>
        h_3 n
```

In other words, the specific configuration of patterns used in `half` are captured in `half.match_1`.

This definition is a more readable version of `half`'s pre-definition:

```
def half' : Nat → Nat :=
  fun (x : Nat) =>
    half.match_1 (motive := fun _ => Nat) x
      (fun _ => 0) -- Case for 0
      (fun _ => 0) -- Case for 1
      (fun n => Nat.succ (half' n)) -- Case for n + 2
```

To elaborate it as a structurally recursive function, the first step is to establish the `bRecOn` invocation. The definition must be marked **`noncomputable`** because Lean does not support code generation for recursors such as `Nat.brecOn`.

```
noncomputable
def half'' : Nat → Nat :=
  fun (x : Nat) =>
    x.brecOn fun n table =>

      ⋯
/- To translate:
    half.match_1 (motive := fun _ => Nat) x
      (fun _ => 0) -- Case for 0
      (fun _ => 0) -- Case for 1
      (fun n => Nat.succ (half' n)) -- Case for n + 2
-/
```

The next step is to replace occurrences of `x` in the original function body with the `n` provided by `brecOn`. Because `table`'s type depends on `x`, it must also be generalized when splitting cases with `half.match_1`, leading to a motive with an extra parameter.

```
noncomputable
def half'' : Nat → Nat :=
  fun (x : Nat) =>
    x.brecOn fun n table =>
      (half.match_1
        (motive :=
          fun k =>
            k.below (motive := fun _ => Nat) →
            Nat)
        n
        ⋯
        ⋯
        _)
```

```
      table
/- To translate:
      (fun _ => 0) -- Case for 0
      (fun _ => 0) -- Case for 1
      (fun n => Nat.succ (half' n)) -- Case for n + 2
-/
```

The three cases' placeholders expect the following types:

```
don't know how to synthesize placeholder for argument 'h_1'
context:
x n : Nat
table : Nat.below n
⊢ Unit → (fun k => Nat.below k → Nat) Nat.zero
```

```
don't know how to synthesize placeholder for argument 'h_2'
context:
x n : Nat
table : Nat.below n
⊢ Unit → (fun k => Nat.below k → Nat) 1
```

```
don't know how to synthesize placeholder for argument 'h_3'
context:
x n : Nat
table : Nat.below n
⊢ (n : Nat) → (fun k => Nat.below k → Nat) n.succ.succ
```

The first two cases in the pre-definition are constant functions, with no recursion to check:

```
noncomputable
def half'' : Nat → Nat :=
  fun (x : Nat) =>
    x.brecOn fun n table =>
      (half.match_1
        (motive :=
          fun k =>
            k.below (motive := fun _ => Nat) →
            Nat)
        n
        (fun () _ => .zero)
        (fun () _ => .zero)
        _)
      table
/- To translate:
      (fun n => Nat.succ (half' n)) -- Case for n + 2
-/
```

The final case contains a recursive call. It should be translated into a lookup into the course-of-values table. A more readable representation of the last hole's type is:

```
(n : Nat) →
Nat.below (motive := fun _ => Nat) n.succ.succ →
Nat
```

which is equivalent to

```
(n : Nat) →
Nat ×' (Nat ×' Nat.below (motive := fun _ => Nat) n) →
Nat
```

The first `Nat` in the course-of-values table is the result of recursion on `n + 1`, and the second is the result of recursion on `n`. The recursive call can thus be replaced by a lookup, and the elaboration is successful:

```
noncomputable
def half'' : Nat → Nat :=
  fun (x : Nat) =>
    x.brecOn fun n table =>
      (half.match_1
        (motive :=
          fun k =>
            k.below (motive := fun _ => Nat) →
            Nat)
        n
        (fun () _ => .zero)
        (fun () _ => .zero)
        (fun _ table => Nat.succ table.2.1)
      table
```

The actual elaborator keeps track of the relationship between the parameter being checked for structural recursion and the positions in the course-of-values tables by inserting sentinel types with fresh names into the motive.

7.6.3. Well-Founded Recursion

Functions defined by *well-founded recursion* are those in which each recursive call has arguments that are *smaller* (in a suitable sense) than the functions' parameters. In contrast to structural recursion, in which recursive definitions must satisfy particular *syntactic* requirements, definitions that use well-founded recursion employ *semantic* arguments. This allows a larger class of recursive definitions to be accepted. Furthermore, when Lean's automation fails to construct a termination proof, it is possible to specify one manually.

All definitions are treated identically by the Lean compiler. In Lean's logic, definitions that use well-founded recursion typically do not reduce definitionally. The reductions do hold as propositional equalities, however, and Lean automatically proves them. This does not typically make it more difficult to prove properties of definitions that use well-founded recursion, because the propositional reductions can be used to reason about the behavior of the function. It does mean, however, that using these functions in types typically does not work well. Even when the reduction behavior happens to hold definitionally, it is often much slower than structurally recursive definitions in the kernel, which must unfold the termination proof along with the definition. When possible, recursive function that are intended for use in types or in other situations where definitional equality is important should be defined with structural recursion.

To explicitly use well-founded recursion, a function or theorem definition can be annotated with a `termination_by` clause that specifies the *measure* by which the function terminates. The measure should be a term that decreases at each recursive call; it may be one of the function's parameters or a tuple of the parameters, but it may also be any other term. The measure's type must be equipped with a well-founded relation, which determines what it means for the measure to decrease.

syntax

Explicit Well-Founded Recursion

The `termination_by` clause introduces the termination argument.

```
terminationBy ::= ...
    | termination_by (ident* =>)? term
```

The identifiers before the optional => can bring function parameters into scope that are not already bound in the declaration header, and the mandatory term must indicate one of the function's parameters, whether introduced in the header or locally in the clause.

▼ *Example: Division by Iterated Subtraction*

Division can be specified as the number of times the divisor can be subtracted from the dividend. This operation cannot be elaborated using structural recursion because subtraction is not pattern matching. The value of n does decrease with each recursive call, so well-founded recursion can be used to justify the definition of division by iterated subtraction.

```
def div (n k : Nat) : Nat :=
  if k = 0 then 0
  else if k > n then 0
  else 1 + div (n - k) k
termination_by n
```

7.6.3.1. Well-Founded Relations

A relation < is a *well-founded relation* if there exists no infinitely descending chain

$$x_0 \succ x_1 \succ \cdots$$

In Lean, types that are equipped with a canonical well-founded relation are instances of the `WellFoundedRelation` type class.

type class

```
WellFoundedRelation.{u} (α : Sort u) : Sort (max 1 u)
```

A type that has a standard well-founded relation.

Instances are used to prove that functions terminate using well-founded recursion by showing that recursive calls reduce some measure according to a well-founded relation. This relation can combine well-founded relations on the recursive function's parameters.

Instance Constructor

`WellFoundedRelation.mk.{u}`

Methods

`rel : α → α → Prop`

A well-founded relation on α.

`wf : WellFounded WellFoundedRelation.rel`

A proof that `rel` is, in fact, well-founded.

The most important instances are:

- `Nat`, ordered by `(· < ·)`.
- `Prod`, ordered lexicographically: $(a_1, b_1) < (a_2, b_2)$ if and only if $a_1 < a_2$ or $a_1 = a_2$ and $b_1 < b_2$.
- Every type that is an instance of the `SizeOf` type class, which provides a method `SizeOf.sizeOf`, has a well-founded relation. For these types, $x_1 < x_2$ if and only if `sizeOf` x_1 < `sizeOf` x_2. For inductive types, a `SizeOf` instance is automatically derived by Lean.

Note that there exists a low-priority instance `instSizeOfDefault` that provides a `SizeOf` instance for any type, and always returns `0`. This instance cannot be used to prove that a function terminates using well-founded recursion because `0 < 0` is false.

▼ *Example: Default Size Instance*

Function types in general do not have a well-founded relation that's useful for termination proofs. Instance synthesis thus selects `instSizeOfDefault` and the corresponding well-founded relation. If the measure is a function, the default `SizeOf` instance is selected and the proof cannot succeed.

```
def fooInst (b : Bool → Bool) : Unit := fooInst (b ∘ b)
termination_by b
decreasing_by

guard_target =
    @sizeOf (Bool → Bool) (instSizeOfDefault _) (b ∘ b) < sizeOf
b
```

```
  simp only [sizeOf, default.sizeOf]
  guard_target = 0 < 0
  simp
  guard_target = False
  sorry
```

7.6.3.2. Termination proofs

Once a measure is specified and its well-founded relation is determined, Lean determines the termination proof obligation for every recursive call.

The proof obligation for each recursive call is of the form $g\ a_1\ a_2\ \ldots < g\ p_1\ p_2\ \ldots$, where:

- g is the measure as a function of the parameters,
- $<$ is the inferred well-founded relation,
- $a_1\ a_2\ \ldots$ are the arguments of the recursive call and
- $p_1\ p_2\ \ldots$ are the parameters of the function definition.

The context of the proof obligation is the local context of the recursive call. In particular, local assumptions (such as those introduced by `if h : _`, `match h : _ with` or `have`) are available. If a function parameter is the discriminant of a pattern match (e.g. by a `match` expression), then this parameter is refined to the matched pattern in the proof obligation.

The overall termination proof obligation consists of one goal for each recursive call. By default, the tactic `decreasing_trivial` is used to prove each proof obligation. A custom tactic script can be provided using the optional `decreasing_by` clause, which comes after the `termination_by` clause. This tactic script is run once, with one goal for each proof obligation, rather than separately on each proof obligation.

▼ *Example: Termination Proof Obligations*

The following recursive definition of the Fibonacci numbers has two recursive calls, which results in two goals in the termination proof.

```
def fib (n : Nat) :=
  if h : n ≤ 1 then
    1
  else
    fib (n - 1) + fib (n - 2)
termination_by n
decreasing_by
  skip
```

```
n : Nat
h : ¬n ≤ 1
⊢ n - 1 < n
```

```
n : Nat
h : ¬n ≤ 1
⊢ n - 2 ≤ n
```

Here, the measure is simply the parameter itself, and the well-founded order is the less-than relation on natural numbers. The first proof goal requires the user to prove that the argument of the first recursive call, namely `n - 1`, is strictly smaller than the function's parameter, `n`.

Both termination proofs can be easily discharged using the `omega` tactic.

```
def fib (n : Nat) :=
  if h : n ≤ 1 then
    1
  else
    fib (n - 1) + fib (n - 2)
termination_by n
decreasing_by
  · omega
  · omega
```

▼ *Example: Refined Parameters*

If a parameter of the function is the discriminant of a pattern match, then the proof obligations mention the refined parameter.

```
def fib : Nat → Nat
  | 0 | 1 => 1
  | .succ (.succ n) => fib (n + 1) + fib n
termination_by n => n
decreasing_by
  skip
```

```
n : Nat
⊢ n + 1 ≤ n.succ.succ
```

```
n : Nat
⊢ n ≤ n.succ.succ
```

Additionally, the context is enriched with additional assumptions that can make it easier to prove termination. Some examples include:

- In the branches of an if-then-else expression, a hypothesis that asserts the current branch's condition is added, much as if the dependent if-then-else syntax had been used.
- In the function argument to certain higher-order functions, the context of the function's body is enriched with assumptions about the argument.

This list is not exhaustive, and the mechanism is extensible. It is described in detail in the section on preprocessing.

▼ *Example: Enriched Proof Obligation Contexts*

Here, the `if` does not add a local assumption about the condition (that is, whether `n ≤ 1`) to the local contexts in the branches.

```
def fib (n : Nat) :=
  if n ≤ 1 then
    1
  else
    fib (n - 1) + fib (n - 2)
termination_by n
decreasing_by
  skip
```

Nevertheless, the assumptions are available in the context of the termination proof:

```
n : Nat
h✝ : ¬n ≤ 1
⊢ n - 1 ≤ n
```

```
n : Nat
h✝ : ¬n ≤ 1
⊢ n - 2 ≤ n
```

Termination proof obligations in body of a `for…in` loop are also enriched, in this case with a `Std.Range` membership hypothesis:

```
def f (xs : Array Nat) : Nat := Id.run do
  let mut s := xs.sum
  for i in [:xs.size] do
    s := s + f (xs.take i)
  pure s
termination_by xs
decreasing_by
  skip
```

```
xs : Array Nat
 i : Nat
h✝ : i ∈ { stop := xs.size, step_pos := Nat.zero_lt_one }
⊢ sizeOf (xs.take i) < sizeOf xs
```

Similarly, in the following (contrived) example, the termination proof contains an additional assumption showing that `x ∈ xs`.

```
def f (n : Nat) (xs : List Nat) : Nat :=
  List.sum (xs.map (fun x => f x []))
termination_by xs
```

```
  decreasing_by
    skip

  n : Nat
 xs : List Nat
  x : Nat
ht : x ∈ xs
⊢ sizeOf [] ≤ sizeOf xs
```

This feature requires special setup for the higher-order function under which the recursive call is nested, as described in the section on preprocessing. In the following definition, identical to the one above except using a custom, equivalent function instead of `List.map`, the proof obligation context is not enriched:

```
  def List.myMap := @List.map
  def f (n : Nat) (xs : List Nat) : Nat :=
    List.sum (xs.myMap (fun x => f x []))
  termination_by xs
  decreasing_by
    skip

  n : Nat
 xs : List Nat
  x : Nat
⊢ sizeOf [] ≤ sizeOf xs
```

7.6.3.3. Default Termination Proof Tactic

If no `decreasing_by` clause is given, then the **`decreasing_tactic`** is used implicitly, and applied to each proof obligation separately.

tactic

`decreasing_tactic`

The tactic **`decreasing_tactic`** mainly deals with lexicographic ordering of tuples, applying `Prod.Lex.right` if the left components of the product are definitionally equal, and `Prod.Lex.left` otherwise. After preprocessing tuples this way, it calls the **`decreasing_trivial`** tactic.

tactic

`decreasing_trivial`

Extensible helper tactic for **`decreasing_tactic`**. This handles the "base case" reasoning after applying lexicographic order lemmas. It can be extended by adding more macro definitions, e.g.

```
macro_rules | `(tactic| decreasing_trivial) => `(tactic| linarith)
```

The tactic `decreasing_trivial` is an extensible tactic that applies a few common heuristics to solve a termination goal. In particular, it tries the following tactics and theorems:

- `simp_arith`
- `assumption`
- theorems `Nat.sub_succ_lt_self`, `Nat.pred_lt_of_lt`, and `Nat.pred_lt`, which handle common arithmetic goals
- `omega`
- **`array_get_dec`** and **`array_mem_dec`**, which prove that the size of array elements is less than the size of the array
- **`sizeOf_list_dec`** that the size of list elements is less than the size of the list
- `String.Iterator.sizeOf_next_lt_of_hasNext` and `String.Iterator.sizeOf_next_lt_of_atEnd`, to handle iteration through a string using `for`

This tactic is intended to be extended with further heuristics using `macro_rules`.

▼ *Example: No Backtracking of Lexicographic Order*

A classic example of a recursive function that needs a more complex measure is the Ackermann function:

```
def ack : Nat → Nat → Nat
  | 0,     n     => n + 1
  | m + 1, 0     => ack m 1
  | m + 1, n + 1 => ack m (ack (m + 1) n)
termination_by m n => (m, n)
```

The measure is a tuple, so every recursive call has to be on arguments that are lexicographically smaller than the parameters. The default `decreasing_tactic` can handle this.

In particular, note that the third recursive call has a second argument that is smaller than the second parameter and a first argument that is definitionally equal to the first parameter. This allowed `decreasing_tactic` to apply `Prod.Lex.right`.

```
Prod.Lex.right {α β} {ra : α → α → Prop} {rb : β → β → Prop}
  (a : α) {b₁ b₂ : β}
  (h : rb b₁ b₂) :
  Prod.Lex ra rb (a, b₁) (a, b₂)
```

It fails, however, with the following modified function definition, where the third recursive call's first argument is provably smaller or equal to the first parameter, but not syntactically equal:

```
def synack : Nat → Nat → Nat
  | 0,     n     => n + 1
```

```
  | m + 1, 0     => synack m 1
  | m + 1, n + 1 => synack m (synack (m / 2 + 1) n)
termination_by m n => (m, n)
```

```
failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded
relation
  - Use `decreasing_by` to specify your own tactic for
discharging this kind of goal
case h
m n : Nat
⊢ m / 2 + 1 < m + 1
```

Because `Prod.Lex.right` is not applicable, the tactic used `Prod.Lex.left`, which resulted in the unprovable goal above.

This function definition may require a manual proof that uses the more general theorem `Prod.Lex.right'`, which allows the first component of the tuple (which must be of type `Nat`) to be less or equal instead of strictly equal:

```
Prod.Lex.right' {β} (rb : β → β → Prop)
  {a₂ : Nat} {b₂ : β} {a₁ : Nat} {b₁ : β}
  (h₁ : a₁ ≤ a₂) (h₂ : rb b₁ b₂) :
  Prod.Lex Nat.lt rb (a₁, b₁) (a₂, b₂)

def synack : Nat → Nat → Nat
  | 0, n => n + 1
  | m + 1, 0 => synack m 1
  | m + 1, n + 1 => synack m (synack (m / 2 + 1) n)
termination_by m n => (m, n)
decreasing_by
  · apply Prod.Lex.left
    omega
  -- the next goal corresponds to the third recursive call
  · apply Prod.Lex.right'
    · omega
    · omega
  · apply Prod.Lex.left
    omega
```

The `decreasing_tactic` tactic does not use the stronger `Prod.Lex.right'` because it would require backtracking on failure.

7.6.3.4. Inferring Well-Founded Recursion

If a recursive function definition does not indicate a termination measure, Lean will attempt to discover one automatically. If neither `termination_by` nor `decreasing_by` is provided, Lean will try to infer structural recursion before attempting well-founded recursion. If a `decreasing_by` clause is present, only well-founded recursion is attempted.

To infer a suitable termination measure, Lean considers multiple *basic termination measures*, which are termination measures of type `Nat`, and then tries all tuples of these measures.

The basic termination measures considered are:

- all parameters whose type have a non-trivial `SizeOf` instance
- the expression $e_2 - e_1$ whenever the local context of a recursive call has an assumption of type $e_1 < e_2$ or $e_1 \le e_2$, where e_1 and e_2 are of type `Nat` and depend only on the function's parameters. (i) This approach is based on work by Panagiotis Manolios and Daron Vroon, 2006. "Termination Analysis with Calling Context Graphs". In *Proceedings of the International Conference on Computer Aided Verification (CAV 2006).* (LNCS 4144).
- in a mutual group, an additional basic measure is used to distinguish between recursive calls to other functions in the group and recursive calls to the function being defined (for details, see the section on mutual well-founded recursion)

Candidate measures are basic measures or tuples of basic measures. If any of the candidate measures allow all proof obligations to be discharged by the termination proof tactic (that is, the tactic specified by `decreasing_by`, or `decreasing_trivial` if there is no `decreasing_by` clause), then an arbitrary such candidate measure is selected as the automatic termination measure.

A `termination_by?` clause causes the inferred termination annotation to be shown. It can be automatically added to the source file using the offered suggestion or code action.

To avoid the combinatorial explosion of trying all tuples of measures, Lean first tabulates all basic termination measures, determining whether the basic measure is decreasing, strictly decreasing, or non-decreasing. A decreasing measure is smaller for at least one recursive call and never increases at any recursive call, while a strictly decreasing measure is smaller at all recursive calls. A non-decreasing measure is one that the termination tactic could not show to be decreasing or strictly decreasing. A suitable tuple is chosen based on the table. (i) This approach is based on Lukas Bulwahn, Alexander Krauss, and Tobias Nipkow, 2007. "Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL". In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLS 2007).* (LNTCS 4732). This table shows up in the error message when no automatic measure could be found.

▼ *Example: Termination failure*

If there is no `termination_by` clause, Lean attempts to infer a measure for well-founded recursion. If it fails, then it prints the table mentioned above. In this example, the `decreasing_by` clause simply prevents Lean from also attempting structural recursion; this keeps the error message specific.

```
def f : (n m l : Nat) → Nat
  | n+1, m+1, l+1 => [
      f (n+1) (m+1) (l+1),
      f (n+1) (m-1) (l),
      f (n)   (m+1) (l) ].sum
```

```
  | _, _, _ => 0
decreasing_by all_goals decreasing_tactic
```

```
Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof
attempted)
            n m l
1) 32:6-25 = = =
2) 33:6-23 = < _
3) 34:6-23 < _ _
Please use `termination_by` to specify a decreasing measure.
```

The three recursive calls are identified by their source positions. This message conveys the following facts:

- In the first recursive call, all arguments are (provably) equal to the parameters
- In the second recursive call, the first argument is equal to the first parameter and the second argument is provably smaller than the second parameter. The third parameter was not checked for this recursive call, because it was not necessary to determine that no suitable termination argument exists.
- In the third recursive call, the first argument decreases strictly, and the other arguments were not checked.

When termination proofs fail in this manner, a good technique to discover the problem is to explicitly indicate the expected termination argument using `termination_by`. This will surface the messages from the failing tactic.

▼ *Example: Array Indexing*

The purpose of considering expressions of the form $e_2 - e_1$ as measures is to support the common idiom of counting up to some upper bound, in particular when traversing arrays in possibly interesting ways. In the following function, which performs binary search on a sorted array, this heuristic helps Lean to find the $j - i$ measure.

```
def binarySearch (x : Int) (xs : Array Int) : Option Nat :=
  go 0 xs.size
where
  go (i j : Nat) (hj : j ≤ xs.size := by omega) :=
    if h : i < j then
      let mid := (i + j) / 2
      let y := xs[mid]
      if x = y then
        some mid
      else if x < y then
        go i mid
      else
        go (mid + 1) j
```

```
  else
    none
termination_by?
```

The fact that the inferred termination argument uses some arbitrary measure, rather than an optimal or minimal one, is visible in the inferred measure, which contains a redundant `j`:

```
Try this: termination_by (j, j - i)
```

▼ *Example: Termination Proof Tactics During Inference*

The tactic indicated by `decreasing_by` is used slightly differently when inferring the termination measure than it is in the actual termination proof.

- During inference, it is applied to a *single* goal, attempting to prove $<$ or $\leq$ on `Nat`.
- During the termination proof, it is applied to many simultaneous goals (one per recursive call), and the goals may involve the lexicographic ordering of pairs.

A consequence is that a `decreasing_by` block that addresses goals individually and which works successfully with an explicit termination argument can cause inference of the termination measure to fail:

```
def ack : Nat → Nat → Nat
  | 0, n => n + 1
  | m + 1, 0 => ack m 1
  | m + 1, n + 1 => ack m (ack (m + 1) n)
decreasing_by
  · apply Prod.Lex.left
    omega
  · apply Prod.Lex.right
    omega
  · apply Prod.Lex.left
    omega
```

It is advisable to always include a `termination_by` clause whenever an explicit `decreasing_by` proof is given.

▼ *Example: Inference too powerful*

Because `decreasing_tactic` avoids the need to backtrack by being incomplete with regard to lexicographic ordering, Lean may infer a termination measure that leads to goals that the tactic cannot prove. In this case, the error message is the one that results from the failing tactic rather than the one that results from being unable to find a measure. This is what happens in `notAck`:

```
def notAck : Nat → Nat → Nat
  | 0, n => n + 1
  | m + 1, 0 => notAck m 1
  | m + 1, n + 1 => notAck m (notAck (m / 2 + 1) n)
decreasing_by all_goals decreasing_tactic
```

```
failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded
relation
  - Use `decreasing_by` to specify your own tactic for
discharging this kind of goal
case h
m n : Nat
⊢ m / 2 + 1 < m + 1
```

In this case, explicitly stating the termination measure helps.

7.6.3.5. Mutual Well-Founded Recursion

Lean supports the definition of mutually recursive functions using well-founded recursion. Mutual recursion may be introduced using a mutual block, but it also results from `let rec` expressions and `where` blocks. The rules for mutual well-founded recursion are applied to a group of actually mutually recursive, lifted definitions, that results from the elaboration steps for mutual groups.

If any function in the mutual group has a `termination_by` or `decreasing_by` clause, well-founded recursion is attempted. If a termination measure is specified using `termination_by` for *any* function in the mutual group, then *all* functions in the group must specify a termination measure, and they have to have the same type.

If no termination argument is specified, the termination argument is inferred, as described above. In the case of mutual recursion, a third class of basic measures is considered during inference, namely for each function in the mutual group the measure that is `1` for that function and `0` for the others. This allows Lean to order the functions so that some calls from one function to another are allowed even if the parameters do not decrease.

▼ *Example: Mutual recursion without parameter decrease*

In the following mutual function definitions, the parameter does not decrease in the call from `g` to `f`. Nonetheless, the definition is accepted due to the ordering imposed on the functions themselves by the additional basic measure.

```
mutual
  def f : (n : Nat) → Nat
    | 0 => 0
    | n+1 => g n
  termination_by?
```

```
  def g (n : Nat) : Nat := (f n) + 1
  termination_by?
end
```

The inferred termination argument for `f` is:

```
Try this: termination_by n => (n, 0)
```

The inferred termination argument for `g` is:

```
Try this: termination_by (n, 1)
```

7.6.3.6. Preprocessing Function Definitions

Lean *preprocesses* the function's body before determining the proof obligations at each call site, transforming it into an equivalent definition that may include additional information. This preprocessing step is primarily used to enrich the local context with additional assumptions that may be necessary in order to solve the termination proof obligations, freeing users from the need to perform equivalent transformations by hand. Preprocessing uses the simplifier and is extensible by the user.

The preprocessing happens in three steps:

1. Lean annotates occurrences of a function's parameter, or a subterm of a parameter, with the `wfParam` gadget.

   ```
   wfParam {α} (a : α) : α
   ```

   More precisely, every occurrence of the function's parameters is wrapped in `wfParam`. Whenever a `match` expression has *any* discriminant wrapped in `wfParam`, the gadget is removed and every occurrence of a pattern match variable (regardless of whether it comes from the discriminant with the `wfParam` gadget) is wrapped in `wfParam`. The `wfParam` gadget is additionally floated out of projection function applications.

2. The annotated function body is simplified using the simplifier, using only simplification rules from the **`wf_preprocess`** custom simp set.

3. Finally, any left-over `wfParam` markers are removed.

Annotating function parameters that are used for well-founded recursion allows the preprocessing simplification rules to distinguish between parameters and other terms.

attribute

Preprocessing Simp Set for Well-Founded Recursion

```
attr ::= ...
    | wf_preprocess
```

Theorems tagged with the `wf_preprocess` attribute are used during the processing of functions defined by well-founded recursion. They are applied to the function's body to add additional hypotheses, such as replacing `if c then _ else _` with `if h : c then _ else _` or `xs.map` with `xs.attach.map`. Also see `wfParam`.

def

```
wfParam.{u} {α : Sort u} (a : α) : α
```

The `wfParam` gadget is used internally during the construction of recursive functions by wellfounded recursion, to keep track of the parameter for which the automatic introduction of `List.attach` (or similar) is plausible.

Some rewrite rules in the **`wf_preprocess`** simp set apply generally, without heeding the `wfParam` marker. In particular, the theorem `ite_eq_dite` is used to extend the context of an if-then-else expression branch with an assumption about the condition: This assumption's name should be an inaccessible name based on h, as is indicated by using `binderNameHint` with the term `()`. Binder name hints are described in the tactic language reference.

```
ite_eq_dite {P : Prop} {α : Sort u} {a b : α} [Decidable P]  :
  (if P then a else b) =
  if h : P then
    binderNameHint h () a
  else
    binderNameHint h () b
```

Other rewrite rules use the `wfParam` marker to restrict their applicability; they are used only when a function (like `List.map`) is applied to a parameter or subterm of a parameter, but not otherwise. This is typically done in two steps:

1. A theorem such as `List.map_wfParam` recognizes a call of `List.map` on a function parameter (or subterm), and uses `List.attach` to enrich the type of the list elements with the assertion that they are indeed elements of that list:

   ```
   List.map_wfParam (xs : List α) (f : α → β) :
     (wfParam xs).map f = xs.attach.unattach.map f
   ```

2. A theorem such as `List.map_unattach` makes that assertion available to the function parameter of `List.map`.

   ```
   List.map_unattach (P : α → Prop)
     (xs : List { x : α // P x }) (f : α → β) :
     xs.unattach.map f = xs.map fun ⟨x, h⟩ =>
       binderNameHint x f <|
       binderNameHint h () <|
       f (wfParam x)
   ```

This theorem uses the `binderNameHint` gadget to preserve a user-chosen binder name, should *f* be a lambda expression.

By separating the introduction of `List.attach` from the propagation of the introduced assumption, the desired the $x \in xs$ assumption is made available to *f* even in chains such as `(xs.reverse.filter p).map f`.

This preprocessing can be disabled by setting the option `wf.preprocess` to `false`. To see the preprocessed function definition, before and after the removal of `wfParam` markers, set the option `trace.Elab.definition.wf` to `true`.

option

`trace.Elab.definition.wf`

Default value: `false`

enable/disable tracing for the given module and submodules

▼ *Example: Preprocessing for a custom data type*

This example demonstrates what is necessary to enable automatic well-founded recursion for a custom container type. The structure type `Pair` is a homogeneous pair: it contains precisely two elements, both of which have the same type. It can be thought of as being similar to a list or array that always contains precisely two elements.

As a container, `Pair` can support a `map` operation. To support well-founded recursion in which recursive calls occur in the body of a function being mapped over a `Pair`, some additional definitions are required, including a membership predicate, a theorem that relates the size of a member to the size of the containing pair, helpers to introduce and eliminate assumptions about membership, **`wf_preprocess`** rules to insert these helpers, and an extension to the **`decreasing_trivial`** tactic. Each of these steps makes it easier to work with `Pair`, but none are strictly necessary; there's no need to immediately implement all steps for every type.

```
/-- A homogeneous pair -/
structure Pair (α : Type u) where
  fst : α
  snd : α

/-- Mapping a function over the elements of a pair -/
def Pair.map (f : α → β) (p : Pair α) : Pair β where
  fst := f p.fst
  snd := f p.snd
```

Defining a nested inductive data type of binary trees that uses `Pair` and attempting to define its `map` function demonstrates the need for preprocessing rules.

```
/-- A binary tree defined using `Pair` -/
inductive Tree (α : Type u) where
  | leaf : α → Tree α
  | node : Pair (Tree α) → Tree α
```

A straightforward definition of the `map` function fails:

```
def Tree.map (f : α → β) : Tree α → Tree β
  | leaf x => leaf (f x)
  | node p => node (p.map (fun t' => t'.map f))
termination_by t => t
```

```
failed to prove termination, possible solutions:
  - Use `have`-expressions to prove the remaining goals
  - Use `termination_by` to specify a different well-founded
relation
  - Use `decreasing_by` to specify your own tactic for
discharging this kind of goal
α : Type u_1
p : Pair (Tree α)
t' : Tree α
⊢ sizeOf t' < 1 + sizeOf p
```

Clearly the proof obligation is not solvable, because nothing connects t' to p.

The standard idiom to enable this kind of function definition is to have a function that enriches each element of a collection with a proof that they are, in fact, elements of the collection. Stating this property requires a membership predicate.

```
inductive Pair.Mem (p : Pair α) : α → Prop where
  | fst : Mem p p.fst
  | snd : Mem p p.snd

instance : Membership α (Pair α) where
  mem := Pair.Mem
```

Every inductive type automatically has a `SizeOf` instance. An element of a collection should be smaller than the collection, but this fact must be proved before it can be used to construct a termination proof:

```
theorem Pair.sizeOf_lt_of_mem {α} [SizeOf α]
    {p : Pair α} {x : α} (h : x ∈ p) :
    sizeOf x < sizeOf p := by
  cases h <;> cases p <;> (simp; omega)
```

The next step is to define `attach` and `unattach` functions that enrich the elements of the pair with a proof that they are elements of the pair, or remove said proof. Here, the type of `Pair.unattach` is more general and can be used with any subtype; this is a typical pattern.

```
def Pair.attach (p : Pair α) : Pair {x : α // x ∈ p} where
  fst := ⟨p.fst, .fst⟩
```

```
    snd := ⟨p.snd, ‹_›.snd⟩

  def Pair.unattach {P : α → Prop} :
      Pair {x : α // P x} → Pair α :=
    Pair.map Subtype.val
```

`Tree.map` can now be defined by using `Pair.attach` and `Pair.sizeOf_lt_of_mem` explicitly:

```
  def Tree.map (f : α → β) : Tree α → Tree β
    | leaf x => leaf (f x)
    | node p => node (p.attach.map (fun ⟨t', _⟩ => t'.map f))
  termination_by t => t
  decreasing_by
    have := Pair.sizeOf_lt_of_mem ‹_›
    simp_all +arith
    omega
```

This transformation can be made fully automatic. The preprocessing feature of well-founded recursion can be used to automate the introduction of the `Pair.attach` function. This is done in two stages. First, when `Pair.map` is applied to one of the function's parameters, it is rewritten to an `attach`/`unattach` composition. Then, when a function is mapped over the result of `Pair.unattach`, the function is rewritten to accept the proof of membership and bring it into scope.

```
  @[wf_preprocess]
  theorem Pair.map_wfParam (f : α → β) (p : Pair α) :
      (wfParam p).map f = p.attach.unattach.map f := by
    cases p
    simp [wfParam, Pair.attach, Pair.unattach, Pair.map]

  @[wf_preprocess]
  theorem Pair.map_unattach {P : α → Prop}
      (p : Pair (Subtype P)) (f : α → β) :
      p.unattach.map f =
      p.map fun ⟨x, h⟩ =>
        binderNameHint x f <|
        f (wfParam x) := by
    cases p; simp [wfParam, Pair.unattach, Pair.map]
```

Now the function body can be written without extra considerations, and the membership assumption is still available to the termination proof.

```
  def Tree.map (f : α → β) : Tree α → Tree β
    | leaf x => leaf (f x)
    | node p => node (p.map (fun t' => t'.map f))
  termination_by t => t
  decreasing_by
    have := Pair.sizeOf_lt_of_mem ‹_›
    simp_all
    omega
```

The proof can be made fully automatic by adding `sizeOf_lt_of_mem` to the `decreasing_trivial` tactic, as is done for similar built-in theorems.

```
macro "sizeOf_pair_dec" : tactic =>
  `(tactic| with_reducible
    have := Pair.sizeOf_lt_of_mem ‹_›
    omega
    done)

macro_rules
  | `(tactic| decreasing_trivial) =>
    `(tactic| sizeOf_pair_dec)

def Tree.map (f : α → β) : Tree α → Tree β
  | leaf x => leaf (f x)
  | node p => node (p.map (fun t' => t'.map f))
termination_by t => t
```

To keep the example short, the **`sizeOf_pair_dec`** tactic is tailored to this particular recursion pattern and isn't really general enough for a general-purpose container library. It does, however, demonstrate that libraries can be just as convenient in practice as the container types in the standard library.

7.6.3.7. Theory and Construction

This section gives a very brief glimpse into the mathematical constructions that underlie termination proofs via well-founded recursion, which may surface occasionally. The elaboration of functions defined by well-founded recursion is based on the `WellFounded.fix` operator.

def

```
WellFounded.fix.{u, v} {α : Sort u} {C : α → Sort v} {r : α → α →
Prop}
  (hwf : WellFounded r) (F : (x : α) → ((y : α) → r y x → C y) → C
x)
  (x : α) : C x
```

A well-founded fixpoint. If satisfying the motive `C` for all values that are smaller according to a well-founded relation allows it to be satisfied for the current value, then it is satisfied for all values.

This function is used as part of the elaboration of well-founded recursion.

The type `α` is instantiated with the function's (varying) parameters, packed into one type using `PSigma`. The `WellFounded` relation is constructed from the termination measure via `invImage`.

def

```
invImage.{u_1, u_2} {α : Sort u_1} {β : Sort u_2} (f : α → β)
  (h : WellFoundedRelation β) : WellFoundedRelation α
```

The inverse image of a well-founded relation is well-founded.

The function's body is passed to `WellFounded.fix`, with parameters suitably packed and unpacked, and recursive calls are replaced with a call to the value provided by `WellFounded.fix`. The termination proofs generated by the `decreasing_by` tactics are inserted in the right place.

Finally, the equational and unfolding theorems for the recursive function are proved from `WellFounded.fix_eq`. These theorems hide the details of packing and unpacking arguments and describe the function's behavior in terms of the original definition.

In the case of mutual recursion, an equivalent non-mutual function is constructed by combining the function's arguments using `PSum`, and pattern-matching on that sum type in the result type and the body.

The definition of `WellFounded` builds on the notion of *accessible elements* of the relation:

inductive predicate

```
WellFounded.{u} {α : Sort u} (r : α → α → Prop) : Prop
```

A relation r is `WellFounded` if all elements of α are accessible within r. If a relation is `WellFounded`, it does not allow for an infinite descent along the relation.

If the arguments of the recursive calls in a function definition decrease according to a well founded relation, then the function terminates. Well-founded relations are sometimes called *Artinian* or said to satisfy the “descending chain condition”.

Constructors

```
| intro.{u} {α : Sort u} {r : α → α → Prop}
    (h : ∀ (a : α), Acc r a) : WellFounded r
```

If all elements are accessible via r, then r is well-founded.

inductive predicate

```
Acc.{u} {α : Sort u} (r : α → α → Prop) : α → Prop
```

`Acc` is the accessibility predicate. Given some relation r (e.g. <) and a value x, `Acc r x` means that x is accessible through r:

x is accessible if there exists no infinite sequence $\ldots < y_2 < y_1 < y_0 < x$.

Constructors

```
| intro.{u} {α : Sort u} {r : α → α → Prop} (x : α)
    (h : ∀ (y : α), r y x → Acc r y) : Acc r x
```

A value is accessible if for all y such that $r\ y\ x$, y is also accessible. Note that if there exists no y such that $r\ y\ x$, then x is accessible. Such an x is called a *base case*.

▼ *Example: Division by Iterated Subtraction: Termination Proof*

The definition of division by iterated subtraction can be written explicitly using well-founded recursion.

```
noncomputable def div (n k : Nat) : Nat :=
  (inferInstanceAs (WellFoundedRelation Nat)).wf.fix
    (fun n r =>
      if h : k = 0 then 0
      else if h : k > n then 0
      else 1 + (r (n - k) <| by
        show (n - k) < n
        omega))
    n
```

The definition must be marked `noncomputable` because well-founded recursion is not supported by the compiler. Like recursors, it is part of Lean's logic.

The definition of division should satisfy the following equations:

- `∀{n k : Nat}, (k = 0) → div n k = 0`
- `∀{n k : Nat}, (k > n) → div n k = 0`
- `∀{n k : Nat}, (k ≠ 0) → (¬ k > n) → div n k = div (n - k) k`

This reduction behavior does not hold definitionally:

```
theorem div.eq0 : div n 0 = 0 := by rfl
```

```
tactic 'rfl' failed, the left-hand side
  div n 0
is not definitionally equal to the right-hand side
  0
n : Nat
⊢ div n 0 = 0
```

However, using `WellFounded.fix_eq` to unfold the well-founded recursion, the three equations can be proved to hold:

```
theorem div.eq0 : div n 0 = 0 := by
  unfold div
  apply WellFounded.fix_eq
```

```
theorem div.eq1 : k > n → div n k = 0 := by
  intro h
  unfold div
  rw [WellFounded.fix_eq]

simp only [gt_iff_lt, dite_eq_ite, ite_eq_left_iff, Nat.not_lt]
  intros; omega

theorem div.eq2 :
    ¬ k = 0 → ¬ (k > n) →
    div n k = 1 + div (n - k) k := by
  intros
  unfold div
  rw [WellFounded.fix_eq]
  simp_all only [
      gt_iff_lt, Nat.not_lt,
      dite_false, dite_eq_ite,
      ite_false, ite_eq_right_iff
    ]
  omega
```

7.6.4. Partial Fixpoint Recursion

All definitions are fundamentally equations: the new constant being defined is equal to the right-hand side of the definition. For functions defined by structural recursion, this equation holds definitionally, and there is a unique value returned by application of the function. For functions defined by well-founded recursion, the equation may hold only propositionally, but all type-correct applications of the function to arguments are equal to the respective values prescribed by the definition. In both cases, the fact that the function terminates for all inputs means that the value computed by applying the function is always uniquely determined.

In some cases where a function does not terminate for all arguments, the equation may not *uniquely* determine the function's return value for each input, but there are nonetheless functions for which the defining equation holds. In these cases, a definition as a *partial fixpoint* may still be possible. Any function that satisfies the defining equation can be used to demonstrate that the equation does not create a logical contradiction, and the equation can then be proven as a theorem about this function. As with the other strategies for defining recursive functions, compiled code uses the function as it was originally written; like definitions in terms of eliminators or recursion over accessibility proofs, the function used to define the partial fixpoint is used only to justify the function's equations in Lean's logic for purposes of mathematical reasoning.

The term *partial fixpoint* is specific to Lean. Functions declared `partial` do not require termination proofs, so long as the type of their return values is inhabited, but they are completely opaque from the perspective of Lean's logic. Partial fixpoints, on the other hand, can be rewritten using their defining equations while writing proofs. Logically speaking, partial fixpoints are total functions that

don't reduce definitionally when applied, but for which equational rewrite rule are provided. They are *partial* in the sense that the defining equation does not necessarily specify a value for all possible arguments.

While partial fixpoints do allow functions to be defined that cannot be expressed using structural or well-founded recursion, the technique is also useful in other cases. Even in cases where the defining equation fully describes the function's behavior and a termination proof using well-founded recursion would be possible, it may simply be more convenient to define the function as a partial fixpoint to avoid a having to write a termination proof.

Defining recursive functions as partial fixpoints only occurs when explicitly requested by annotating the definition with `partial_fixpoint`.

There are two classes of functions that can be defined as partial fixpoints:

- Tail-recursive functions whose return type is inhabited type
- Functions that return values in a suitable monad, such as the `Option` monad

Both classes are backed by the same theory and construction: least fixpoints of monotone equations in chain-complete partial orders.

Just as with structural and well-founded recursion, Lean allows mutually recursive functions to be defined as partial fixpoints. To use this feature, every function definition in a mutual block must be annotated with the `partial_fixpoint` modifier.

▼ *Example: Definition by Partial Fixpoint*

The following function finds the least natural number for which the predicate *p* holds. If p never holds, then this equation does not specify the behavior: the function `find` could return 42 or any other `Nat` in that case and still satisfy the equation.

```
def find (p : Nat → Bool) (i : Nat := 0) : Nat :=
  if p i then
    i
  else
    find p (i + 1)
partial_fixpoint
```

The elaborator can prove that functions satisfying the equation exist. Within Lean's logic, `find` is defined to be an arbitrary such function.

7.6.4.1. Tail-Recursive Functions

A recursive function can be defined as a partial fixpoint if the following two conditions hold:

1. The function's return type is inhabited (as with functions marked `partial`) - either a `Nonempty` or `Inhabited` instance works.
2. All recursive calls are in tail position of the function.

An expression is in *tail position* in the function body if it is:

- the function body itself,
- the branches of a `match` expression that is in tail position,
- the branches of an `if` expression that is in tail position, and
- the body of a `let` expression that is in tail position.

In particular, the discriminant of a `match` expression, the condition of an `if` expression and the arguments of functions are not tail positions.

▼ *Example: Loops are Tail Recursive Functions*

Because the function body itself is a tail position, the infinitely looping function `loop` is tail recursive. It can be defined as a partial fixpoint.

```
def loop (x : Nat) : Nat := loop (x + 1)
partial_fixpoint
```

▼ *Example: Tail Recursion with Branching*

`Array.find` could also be constructed using well-founded recursion with a termination proof, but may be more convenient to define using `partial_fixpoint`, where no termination proof is needed.

```
def Array.find (xs : Array α) (p : α → Bool)
    (i : Nat := 0) : Option α :=
  if h : i < xs.size then
    if p xs[i] then
      some xs[i]
    else
      Array.find xs p (i + 1)
  else
    none
partial_fixpoint
```

If the result of the recursive call is not just returned, but passed to another function, it is not in tail position and this definition fails.

```
def List.findIndex (xs : List α) (p : α → Bool) : Int :=
  match xs with
  | [] => -1
  | x::ys =>
    if p x then
      0
    else
      have r := List.findIndex ys p
      if r = -1 then -1 else r + 1
```

```
partial_fixpoint
```

The error message on the recursive call is:

```
Could not prove 'List.findIndex' to be monotone in its
recursive calls:
  Cannot eliminate recursive call `List.findIndex ys p`
enclosed in
    if ys✝.findIndex p = -1 then -1 else ys✝.findIndex p + 1
  Tried to apply 'monotone_ite', but failed.
  Possible cause: A missing `MonoBind` instance.
  Use `set_option trace.Elab.Tactic.monotonicity true` to
debug.
```

7.6.4.2. Monadic functions

Defining a function as a partial fixpoint is more powerful if the function's return type is a monad that is an instance of `Lean.Order.MonoBind`, such as `Option`. In this case, recursive call are not restricted to tail-positions, but may also occur inside higher-order monadic functions such as `bind` and `List.mapM`.

The set of higher-order functions for which this works is extensible, so no exhaustive list is given here. The aspiration is that a monadic recursive function definition that is built using abstract monadic operations like `bind`, but that does not open the abstraction of the monad (e.g. by matching on the `Option` value), is accepted. In particular, using do-notation should work.

▼ *Example: Monadic functions*

The following function implements the Ackermann function in the `Option` monad, and is accepted without an (explicit or implicit) termination proof:

```
def ack : (n m : Nat) → Option Nat
  | 0,   y   => some (y+1)
  | x+1, 0   => ack x 1
  | x+1, y+1 => do ack x (← ack (x+1) y)
partial_fixpoint
```

Recursive calls may also occur within higher-order functions such as `List.mapM`, if they are set up appropriately, and do-notation:

```
structure Tree where cs : List Tree

def Tree.rev (t : Tree) : Option Tree := do
  Tree.mk (← t.cs.reverse.mapM (Tree.rev ·))
partial_fixpoint

def Tree.rev' (t : Tree) : Option Tree := do
  let mut cs := []
  for c in t.cs do
```

```
    cs := (← c.rev') :: cs
  return Tree.mk cs
partial_fixpoint
```

Pattern matching on the result of the recursive call will prevent the definition by partial fixpoint from going through:

```
def List.findIndex (xs : List α) (p : α → Bool) : Option Nat :=
  match xs with
  | [] => none
  | x::ys =>
    if p x then
      some 0
    else
      match List.findIndex ys p with
      | none => none
      | some r => some (r + 1)
partial_fixpoint
```

```
Could not prove 'List.findIndex' to be monotone in its
recursive calls:
  Cannot eliminate recursive call `List.findIndex ys p`
enclosed in
    match ys✝.findIndex p with
    | none => none
    | some r => some (r + 1)
```

In this particular case, using `Functor.map` instead of explicit pattern matching helps:

```
def List.findIndex (xs : List α) (p : α → Bool) : Option Nat :=
  match xs with
  | [] => none
  | x::ys =>
    if p x then
      some 0
    else
      (· + 1) <$> List.findIndex ys p
partial_fixpoint
```

7.6.4.3. Partial Correctness Theorems

For every function defined as a partial fixpoint, Lean proves that the defining equation is satisfied. This enables proofs by rewriting. However, these equational theorems are not sufficient for reasoning about the behavior of the function on arguments for which the function specification does not terminate. Code paths that lead to infinite recursion at runtime would end up as infinite chains of rewrites in a potential proof.

Partial fixpoints in suitable monads, on the other hand, provide additional theorems that map the undefined values from non-termination to suitable values in the monad. In the `Option` monad, then partial fixpoint equals `Option.none` on all function inputs for which the defining equation specifies non-termination. From this fact, Lean proves a *partial correctness theorem* for the function which allows facts to be concluded when the function's result is `Option.some`.

▼ *Example: Partial Correctness Theorem*

Recall `List.findIndex` from an earlier example:

```
def List.findIndex (xs : List α) (p : α → Bool) : Option Nat :=
  match xs with
  | [] => none
  | x::ys =>
    if p x then
      some 0
    else
      (· + 1) <$> List.findIndex ys p
partial_fixpoint
```

With this function definition, Lean automatically proves the following partial correctness theorem:

```
List.findIndex.partial_correctness.{u_1} {α : Type u_1}
  (p : α → Bool)
  (motive : List α → Nat → Prop)
  (h :
    ∀ (findIndex : List α → Option Nat),
      (∀ (xs : List α) (r : Nat), findIndex xs = some r → motive
xs r) →
        ∀ (xs : List α) (r : Nat),
          (match xs with
              | [] => none
              | x :: ys =>
                if p x = true then some 0
                else (fun x => x + 1) <$> findIndex ys) = some r
→
            motive xs r)
  (xs : List α) (r : Nat) :
  xs.findIndex p = some r →
    motive xs r
```

Here, the motive is a relation between the parameter and return types of `List.findIndex`, with the `Option` removed from the return type. If, when given an arbitrary partial function with a signature that's compatible with `List.findIndex`, the following hold:

- the motive is satisfied for all inputs for which the arbitrary function returns a value (rather than `none`),

- taking one rewriting step with the defining equation, in which the recursive calls are replaced by the arbitrary function, also implies the satisfaction of the motive

then the motive is satsified for all inputs for which the `List.findIndex` returns `some`.

The partial correctness theorem is a reasoning principle. It can be used to prove that the resulting number is a valid index in the list and that the predicate holds for that index:

```
theorem List.findIndex_implies_pred
    (xs : List α) (p : α → Bool) :
    xs.findIndex p = some i →
    ∃x, xs[i]? = some x ∧ p x := by
  apply List.findIndex.partial_correctness
            (motive := fun xs i => ∃ x, xs[i]? = some x ∧ p x)
  intro findIndex ih xs r hsome
  split at hsome
  next => contradiction
  next x ys =>
    split at hsome
    next =>
      have : r = 0 := by simp_all
      simp_all
    next =>

simp only [Option.map_eq_map, Option.map_eq_some'] at hsome
      obtain ⟨r', hr, rfl⟩ := hsome
      specialize ih _ _ hr
      simpa
```

7.6.4.4. Mutual Recursion with Partial Fixpoints

Lean supports the definition of mutually recursive functions using partial fixpoint. Mutual recursion may be introduced using a mutual block, but it also results from `let rec` expressions and `where` blocks. The rules for mutual well-founded recursion are applied to a group of actually mutually recursive, lifted definitions, that results from the elaboration steps for mutual groups.

If all functions in the mutual group have the `partial_fixpoint` clause, then this strategy is used.

7.6.4.5. Theory and Construction

The construction builds on a variant of the Knaster–Tarski theorem: In a chain-complete partial order, every monotone function has a least fixed point.

The necessary theory is found in the `Lean.Order` namespace. This is not meant to be a general purpose library of order theoretic results. Instead, the definitions and theorems in `Lean.Order` are only intended as implementation details of the `partial_fixpoint` feature, and they should be considered a private API that may change without notice.

The notion of a partial order, and that of a chain-complete partial order, are represented by the type classes `Lean.Order.PartialOrder` and `Lean.Order.CCPO`, respectively.

type class

```
Lean.Order.PartialOrder.{u} (α : Sort u) : Sort (max 1 u)
```

A partial order is a reflexive, transitive and antisymmetric relation.

This is intended to be used in the construction of **`partial_fixpoint`**, and not meant to be used otherwise.

Instance Constructor

`Lean.Order.PartialOrder.mk.{u}`

Methods

`rel : α → α → Prop`

A “less-or-equal-to” or “approximates” relation.

This is intended to be used in the construction of **`partial_fixpoint`**, and not meant to be used otherwise.

`rel_refl : ∀ {x : α}, x ⊑ x`

The “less-or-equal-to” or “approximates” relation is reflexive.

`rel_trans : ∀ {x y z : α}, x ⊑ y → y ⊑ z → x ⊑ z`

The “less-or-equal-to” or “approximates” relation is transitive.

`rel_antisymm : ∀ {x y : α}, x ⊑ y → y ⊑ x → x ≃ y`

The “less-or-equal-to” or “approximates” relation is antisymmetric.

type class

```
Lean.Order.CCPO.{u} (α : Sort u) : Sort (max 1 u)
```

A chain-complete partial order (CCPO) is a partial order where every chain has a least upper bound.

This is intended to be used in the construction of **`partial_fixpoint`**, and not meant to be used otherwise.

Instance Constructor

`Lean.Order.CCPO.mk.{u}`

Extends

☑ PartialOrder α

Methods

rel : α → α → Prop

Inherited from PartialOrder α

rel_refl : ∀ {x : α}, x ⊑ x

Inherited from PartialOrder α

rel_trans : ∀ {x y z : α}, x ⊑ y → y ⊑ z → x ⊑ z

Inherited from PartialOrder α

rel_antisymm : ∀ {x y : α}, x ⊑ y → y ⊑ x → x = y

Inherited from PartialOrder α

csup : (α → Prop) → α

The least upper bound of a chain.

This is intended to be used in the construction of **partial_fixpoint**, and not meant to be used otherwise.

csup_spec : ∀ {x : α} {c : α → Prop}, chain c → (CCPO.csup c ⊑ x ↔ ∀ (y : α), c y → y ⊑ x)

csup c is the least upper bound of the chain *c* when all elements x that are at least as large as *csup c* are at least as large as all elements of *c*, and vice versa.

A function is monotone if it preserves partial orders. That is, if *x ⊑ y* then *f x ⊑ f y*. The operator ⊑ represent Lean.Order.PartialOrder.rel.

def

Lean.Order.monotone.{u, v} {α : Sort u} [PartialOrder α] {β : Sort v}
[PartialOrder β] (f : α → β) : Prop

A function is monotone if it maps related elements to related elements.

This is intended to be used in the construction of **partial_fixpoint**, and not meant to be used otherwise.

The fixpoint of a monotone function can be taken using fix, which indeed constructs a fixpoint, as shown by fix_eq,

def

```
Lean.Order.fix.{u} {α : Sort u} [CCPO α] (f : α → α)
  (hmono : monotone f) : α
```

The least fixpoint of a monotone function is the least upper bound of its transfinite iteration.

The `monotone f` assumption is not strictly necessarily for the definition, but without this the definition is not very meaningful and it simplifies applying theorems like `fix_eq` if every use of `fix` already has the monotonicty requirement.

This is intended to be used in the construction of **`partial_fixpoint`**, and not meant to be used otherwise.

theorem

```
Lean.Order.fix_eq.{u} {α : Sort u} [CCPO α] {f : α → α}
  (hf : monotone f) : fix f hf = f (fix f hf)
```

The main fixpoint theorem for fixedpoints of monotone functions in chain-complete partial orders.

This is intended to be used in the construction of **`partial_fixpoint`**, and not meant to be used otherwise.

To construct the partial fixpoint, Lean first synthesizes a suitable `CCPO` instance.

- If the function's result type has a dedicated instance, like `Option` has with `instCCPOOption`, this is used together with the instance for the function type, `instCCPOPi`, to construct an instance for the whole function's type.
- Otherwise, if the function's type can be shown to be inhabited by a witness *w*, then the instance `FlatOrder.instCCPO` for the wrapper type `FlatOrder w` is used. In this order, *w* is a least element and all other elements are incomparable.

Next, the recursive calls in the right-hand side of the function definitions are abstracted; this turns into the argument `f` of `fix`. The monotonicity requirement is solved by the **`monotonicity`** tactic, which applies compositional monotonicity lemmas in a syntax-driven way.

The tactic solves goals of the form `monotone (fun x => … x …)` using the following steps:

- Applying `monotone_const` when there is no dependency on *x* left.
- Splitting on `match` expressions.
- Splitting on `if` expressions.
- Moving **`let`** expression to the context, if the value and type do not depend on *x*.
- Zeta-reducing a **`let`** expression when value and type do depend on *x*.
- Applying lemmas annotated with **`partial_fixpoint_monotone`**

The following monotonicity lemmas are registered, and should allow recursive calls under the given higher-order functions in the arguments indicated by · (but not the other arguments, shown as _).

Theorem	Pattern
monotone_allM	Array.allM · _ _ _
monotone_anyM	Array.anyM · _ _ _
monotone_anyM_loop	Array.anyM.loop · _ _ ⋯ _
monotone_array_filterMapM	Array.filterMapM · _
monotone_array_forM	Array.forM · _ _ _
monotone_array_forRevM	Array.forRevM · _ _ _
monotone_findIdxM?	Array.findIdxM? · _
monotone_findM?	Array.findM? · _
monotone_findRevM?	Array.findRevM? · _
monotone_findSomeM?	Array.findSomeM? · _
monotone_findSomeRevM?	Array.findSomeRevM? · _
monotone_flatMapM	Array.flatMapM · _
monotone_foldlM	Array.foldlM · _ _ _ _
monotone_foldlM_loop	Array.foldlM.loop · _ _ ⋯ _ _ _
monotone_foldrM	Array.foldrM · _ _ _ _
monotone_foldrM_fold	Array.foldrM.fold · _ _ _ ⋯ _
monotone_forIn	forIn _ _ ·
monotone_forIn'	forIn' _ _ ·
monotone_forIn'_loop	Array.forIn'.loop _ · _ ⋯ _
monotone_mapFinIdxM	_.mapFinIdxM ·
monotone_mapM	Array.mapM · _
monotone_modifyM	_.modifyM _ ·
monotone_map	_ <$> ·
monotone_allM	List.allM · _
monotone_anyM	List.anyM · _
monotone_filterAuxM	List.filterAuxM · _ _
monotone_filterM	List.filterM · _
monotone_filterRevM	List.filterRevM · _
monotone_findM?	List.findM? · _
monotone_findSomeM?	List.findSomeM? · _

Theorem	Pattern
monotone_foldlM	List.foldlM · _ _
monotone_foldrM	List.foldrM · _ _
monotone_forIn	forIn _ _ ·
monotone_forIn'	forIn' _ _ ·
monotone_forIn'_loop	List.forIn'.loop _ · _ _ …
monotone_forM	_.forM ·
monotone_mapM	List.mapM · _
monotone_bindM	Option.bindM · _
monotone_elimM	Option.elimM · · ·
monotone_getDM	_.getDM ·
monotone_mapM	Option.mapM · _
monotone_fst	·.fst
monotone_mk	⟨·, ·⟩
monotone_snd	·.snd
monotone_seq	· <*> ·
monotone_seqLeft	· <* ·
monotone_seqRight	· *> ·
coind_impl	· → ·
coind_monotone_and	· ∧ ·
coind_monotone_exists	Exists ·
coind_monotone_forall	∀ (y : _), _ _ _
coind_monotone_or	· ∨ ·
coind_not	¬·
implication_order_monotone_and	· ∧ ·
implication_order_monotone_exists	Exists ·
implication_order_monotone_forall	∀ (y : _), _ _ _
implication_order_monotone_or	· ∨ ·
ind_impl	· → ·
ind_not	¬·
monotone_bind	· >>= ·
monotone_dite	dite _ · ·
monotone_ite	if _ then · else ·

7.6.5. Partial and Unsafe Definitions

While most Lean functions can be reasoned about in Lean's type theory as well as compiled and run, definitions marked `partial` or `unsafe` cannot be meaningfully reasoned about. From the perspective of the logic, `partial` functions are opaque constants, and theorems that refer to `unsafe` definitions are summarily rejected. In exchange for the inability to use these functions for reasoning, there are far fewer requirements placed on them; this can make it possible to write programs that would be impractical or cost-prohibitive to prove anything about, while not giving up formal reasoning for the rest. In essence, the `partial` subset of Lean is a traditional functional programming language that is nonetheless deeply integrated with the theorem proving features, and the `unsafe` subset features the ability to break Lean's runtime invariants in certain rare situations, at the cost of less integration with Lean's theorem-proving features. Analogously, `noncomputable` definitions may use features that don't make sense in programs, but are meaningful in the logic.

7.6.5.1. Partial Functions

The `partial` modifier may only be applied to function definitions. Partial functions are not required to demonstrate termination, and Lean does not attempt to do so. These functions are “partial” in the sense that they do not necessarily specify a mapping from each element of the domain to an element of the codomain, because they might fail to terminate for some or all elements of the domain. They are elaborated into pre-definitions that contain explicit recursion, and type checked using the kernel; however, they are subsequently treated as opaque constants by the logic.

The function's return type must be inhabited; this ensures soundness. Otherwise, a partial function could have a type such as `Unit → Empty`. Together with `Empty.elim`, the existence of such a function could be used to prove `False` even if it does not reduce.

With partial definitions, the kernel is responsible for the following:

- It ensures that the pre-definition's type is indeed a well-formed type.
- It checks that the pre-definition's type is a function type.
- It ensures that the function's codomain is inhabited by demanding a `Nonempty` or `Inhabited` instance.
- It checks that the resulting term would be type-correct if Lean had recursive definitions.

Even though recursive definitions are not part of the kernel's type theory, the kernel can still be used to check that the body of the definition has the right type. This works the same way as in other functional languages: uses of recursion are type checked by checking the body in an environment in which the definition is already associated with its type. Having ensured that it type checks, the body is discarded and only the opaque constant is retained by the kernel. As with all Lean functions, the compiler generates code from the elaborated pre-definition.

Even though partial functions are not unfolded by the kernel, it is still possible to reason about other functions that call them so long as this reasoning doesn't depend on the implementation of the partial function itself.

▼ *Example: Partial Functions in Proofs*

The recursive function `nextPrime` inefficiently computes the next prime number after a given number by repeatedly testing candidates with trial division. Because there are infinitely many prime numbers, it always terminates; however, formulating this proof would be nontrivial. It is thus marked `partial`.

```
def isPrime (n : Nat) : Bool := Id.run do
  for i in [2:n] do
    if i * i > n then return true
    if n % i = 0 then return false
  return true

partial def nextPrime (n : Nat) : Nat :=
  let n := n + 1
  if isPrime n then n else nextPrime n
```

It is nonetheless possible to prove that the following two functions are equal:

```
def answerUser (n : Nat) : String :=
  s!"The next prime is {nextPrime n}"

def answerOtherUser (n : Nat) : String :=
  " ".intercalate [
    "The",
    "next",
    "prime",
    "is",
    toString (nextPrime n)
  ]
```

The proof contains two `simp` steps to demonstrate that the two functions are not syntactically identical. In particular, the desugaring of string interpolation resulted in an extra `toString ""` at the end of `answerUser`'s result.

```
theorem answer_eq_other : answerUser = answerOtherUser := by
  funext n
  simp only [answerUser, answerOtherUser]
  simp only [toString, String.append_empty]
  rfl
```

7.6.5.2. Unsafe Definitions

Unsafe definitions have even fewer safeguards than partial functions. Their codomains do not need to be inhabited, they are not restricted to function definitions, and they have access to features of Lean that might violate internal invariants or break abstractions. As a result, they cannot be used at all as part of mathematical reasoning.

While partial functions are treated as opaque constants by the type theory, unsafe definitions may only be referenced from other unsafe definitions. As a consequence, any function that calls an unsafe function must be unsafe itself. Theorems are not allowed to be declared unsafe.

In addition to unrestricted use of recursion, unsafe functions can cast from one type to another, check whether two values are the very same object in memory, retrieve pointer values, and run `IO` actions from otherwise-pure code. Using these operators requires a thorough understanding of the Lean implementation.

unsafe def

```
unsafeCast.{u, v} {α : Sort u} {β : Sort v} (a : α) : β
```

This function will cast a value of type α to type β, and is a no-op in the compiler. This function is **extremely dangerous** because there is no guarantee that types α and β have the same data representation, and this can lead to memory unsafety. It is also logically unsound, since you could just cast `True` to `False`. For all those reasons this function is marked as **`unsafe`**.

It is implemented by lifting both α and β into a common universe, and then using `cast (lcProof : ULift (PLift α) = ULift (PLift β))` to actually perform the cast. All these operations are no-ops in the compiler.

Using this function correctly requires some knowledge of the data representation of the source and target types. Some general classes of casts which are safe in the current runtime:

- `Array` α to `Array` β where α and β have compatible representations, or more generally for other inductive types.
- `Quot` α `r` and α.
- `@Subtype` α p and α, or generally any structure containing only one non-`Prop` field of type α.
- Casting α to/from `NonScalar` when α is a boxed generic type (i.e. a function that accepts an arbitrary type α and is not specialized to a scalar type like `UInt8`).

unsafe def

```
ptrEq.{u_1} {α : Type u_1} (a b : α) : Bool
```

Compares two objects for pointer equality.

Two objects are pointer-equal if, at runtime, they are allocated at exactly the same address. This function is unsafe because it can distinguish between definitionally equal values.

unsafe def

```
ptrEqList.{u_1} {α : Type u_1} (as bs : List α) : Bool
```

Compares two lists of objects for element-wise pointer equality. Returns `true` if both lists are the same length and the objects at the corresponding indices of each list are pointer-equal.

Two objects are pointer-equal if, at runtime, they are allocated at exactly the same address. This function is unsafe because it can distinguish between definitionally equal values.

unsafe opaque

```
ptrAddrUnsafe.{u} {α : Type u} (a : α) : USize
```

Returns the address at which an object is allocated.

This function is unsafe because it can distinguish between definitionally equal values.

unsafe opaque

```
isExclusiveUnsafe.{u} {α : Type u} (a : α) : Bool
```

Returns `true` if a is an exclusive object.

An object is exclusive if it is single-threaded and its reference counter is 1. This function is unsafe because it can distinguish between definitionally equal values.

unsafe def

```
unsafeIO {α : Type} (fn : IO α) : Except IO.Error α
```

A monad that can have side effects on the external world or throw exceptions of type ε.

`BaseIO` is a version of this monad that cannot throw exceptions. `IO` sets the exception type to `IO.Error`.

unsafe def

```
unsafeEIO {ε α : Type} (fn : EIO ε α) : Except ε α
```

Executes arbitrary side effects in a pure context, with exceptions indicated via `Except`. This a **dangerous** operation that can easily undermine important assumptions about the meaning of Lean programs, and it should only be used with great care and a thorough understanding of compiler internals, and even then only to implement observationally pure operations.

This function is not a good way to convert an `EIO α` or `IO α` into an α. Instead, use do-notation.

Because the resulting value is treated as a side-effect-free term, the compiler may re-order, duplicate, or delete calls to this function. The side effect may even be hoisted into a constant, causing the side effect to occur at initialization time, even if it would otherwise never be called.

unsafe def

```
unsafeBaseIO {α : Type} (fn : BaseIO α) : α
```

Executes arbitrary side effects in a pure context. This a **dangerous** operation that can easily undermine important assumptions about the meaning of Lean programs, and it should only be used with great care and a thorough understanding of compiler internals, and even then only to implement observationally pure operations.

This function is not a good way to convert a `BaseIO α` into an α. Instead, use **do**-notation.

Because the resulting value is treated as a side-effect-free term, the compiler may re-order, duplicate, or delete calls to this function. The side effect may even be hoisted into a constant, causing the side effect to occur at initialization time, even if it would otherwise never be called.

Frequently, unsafe operators are used to write fast code that takes advantage of low-level details. Just as Lean code may be replaced at runtime with C code via the FFI, safe Lean code may be replaced with unsafe Lean code for runtime programs. This is accomplished by adding the **`implemented_by`** attribute to the function that is to be replaced, which is often an **`opaque`** definition. While this does not threaten Lean's soundness as a logic because the constant to be replaced has already been checked by the kernel and the unsafe replacement is only used in run-time code, it is still risky. Both C code and unsafe code may execute arbitrary side effects.

attribute

Replacing Run-Time Implementations

The **`implemented_by`** attribute instructs the compiler to replace one constant with another in compiled code. The replacement constant may be unsafe.

```
attr ::= ...
    | implemented_by ident
```

▼ *Example: Checking Equality with Pointers*

Ordinarily, a `BEq` instance's equality predicate must fully traverse both of its arguments to determine whether they are equal. If they are, in fact, the very same object in memory, this is wasteful indeed. A pointer equality test can be used prior to the traversal to catch this case.

The type being compared is `Tree`, a type of binary trees.

```
inductive Tree α where
  | empty
  | branch (left : Tree α) (val : α) (right : Tree α)
```

An unsafe function may use pointer equality to terminate the structural equality test more quickly, falling back to structural checks when pointer equality fails.

```
unsafe def Tree.fastBEq [BEq α] (t1 t2 : Tree α) : Bool :=
  if ptrEq t1 t2 then
    true
  else
    match t1, t2 with
    | .empty, .empty => true
    | .branch l1 x r1, .branch l2 y r2 =>
      if ptrEq x y || x == y then
        l1.fastBEq l2 && r1.fastBEq r2
      else false
    | _, _ => false
```

An **implemented_by** attribute on an opaque definition bridges the worlds of safe and unsafe code.

```
@[implemented_by Tree.fastBEq]
opaque Tree.beq [BEq α] (t1 t2 : Tree α) : Bool

instance [BEq α] : BEq (Tree α) where
  beq := Tree.beq
```

▼ *Example: Taking Advantage of Run-Time Representations*

Because a Fin is represented identically to its underlying Nat, List.map Fin.val can be replaced by unsafeCast to avoid a linear-time traversal that, in practice, does nothing:

```
unsafe def unFinImpl (xs : List (Fin n)) : List Nat :=
  unsafeCast xs

@[implemented_by unFinImpl]
def unFin (xs : List (Fin n)) : List Nat :=
  xs.map Fin.val
```

From the perspective of the Lean kernel, unFin is defined using List.map:

```
theorem unFin_length_eq_length {xs : List (Fin n)} :
    (unFin xs).length = xs.length := by
  simp [unFin]
```

In compiled code, there is no traversal of the list.

This kind of replacement is risky: the correspondence between the proof and the compiled code depends fully on the equivalence of the two implementations, which cannot be proved in Lean. The correspondence relies on details of Lean's implementation. These “escape hatches” should be used very carefully.

7.6.6. Controlling Reduction

While checking proofs and programs, Lean takes *reducibility*, also known as *transparency*, into account. A definition's reducibility controls the contexts in which it is unfolded during elaboration and proof execution.

There are three levels of reducibility:

Reducible
: Reducible definitions are unfolded essentially everywhere, on demand. Type class instance synthesis, definitional equality checks, and the rest of the language treat the definition as being essentially an abbreviation. This is the setting applied by the `abbrev` command.

Semireducible
: Semireducible definitions are not unfolded by potentially expensive automation such as type class instance synthesis or `simp`, but they are unfolded while checking definitional equality and while resolving generalized field notation. The `def` command generally creates semireducible definitions unless a different reducibility level is specified with an attribute; however, definitions that use well-founded recursion are irreducible by default.

Irreducible
: Irreducible definitions are not unfolded at all during elaboration. Definitions can be made irreducible by applying the `irreducible` attribute.

▼ *Example: Reducibility and Instance Synthesis*

These three aliasees for `String` are respectively reducible, semireducible, and irreducible.

```
abbrev Phrase := String

def Clause := String

@[irreducible]
def Utterance := String
```

The reducible and semireducible aliases are unfolded during the elaborator's definitional equality check, causing them to be considered equivalent to `String`:

```
def hello : Phrase := "Hello"

def goodMorning : Clause := "Good morning"
```

The irreducible alias, on the other hand, is rejected as the type for a string, because the elaborator's definitional equality test does not unfold it:

```
def goodEvening : Utterance := "Good evening"
```

```
type mismatch
  "Good evening"
has type
  String : Type
but is expected to have type
  Utterance : Type
```

Because `Phrase` is reducible, the `ToString String` instance can be used as a `ToString Phrase` instance:

```
#synth ToString Phrase
```

However, `Clause` is semireducible, so the `ToString String` instance cannot be used:

```
#synth ToString Clause
```

```
failed to synthesize
  ToString Clause

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

The instance can be explicitly enabled by creating a `ToString Clause` instance that reduces to the `ToString String` instance. This example works because semireducible definitions are unfolded while checking definitional equality:

```
instance : ToString Clause := inferInstanceAs (ToString String)
```

▼ *Example: Reducibility and Generalized Field Notation*

Generalized field notation unfolds reducible and semireducible declarations while searching for matching names. Given the semireducible alias `Sequence` for `List`:

```
def Sequence := List

def Sequence.ofList (xs : List α) : Sequence α := xs
```

generalized field notation allows `List.reverse` to be accessed from a term of type `Sequence Nat`.

```
#check let xs : Sequence Nat := .ofList [1,2,3]; xs.reverse
```

However, declaring `Sequence` to be irreducible prevents the unfolding:

```
attribute [irreducible] Sequence

#check let xs : Sequence Nat := .ofList [1,2,3]; xs.reverse
```

```
Invalid field `reverse`: The environment does not contain
`Sequence.reverse`
  xs
has type
  Sequence Nat
```

attribute

Reducibility Annotations

A definition's reducibility can be set using one of the three reducibility attributes:

```
attr ::= ...
    | reducible

attr ::= ...
    | semireducible

attr ::= ...
    | irreducible
```

These attributes can only be applied globally in the same file as the definition being modified, but they may be `locally` applied anywhere.

7.6.6.1. Reducibility and Tactics

The tactics `with_reducible`, `with_reducible_and_instances`, and `with_unfolding_all` control which definitions are unfolded by most tactics.

▼ *Example: Reducibility and Tactics*

The functions `plus`, `sum`, and `tally` are all synonyms for `Nat.add` that are respectively reducible, semireducible, and irreducible:

```
abbrev plus := Nat.add

def sum := Nat.add

@[irreducible]
def tally := Nat.add
```

The reducible synonym is unfolded by `simp`:

```
theorem plus_eq_add : plus x y = x + y := by simp
```

The semireducible synonym is not, however, unfolded by `simp`:

```
theorem sum_eq_add : sum x y = x + y := by
  simp
```

Nonetheless, the definitional equality check induced by `rfl` unfolds the `sum`:

```
theorem sum_eq_add : sum x y = x + y := by
  rfl
```

The irreducible `tally`, however, is not reduced by definitional equality.

```
theorem tally_eq_add : tally x y = x + y := by
  rfl
```

The `simp` tactic can unfold any definition, even irreducible ones, when they are explicitly provided:

```
theorem tally_eq_add : tally x y = x + y := by
  simp [tally]
```

Similarly, part of a proof can be instructed to ignore irreducibility by placing it in a `with_unfolding_all` block:

```
theorem tally_eq_add : tally x y = x + y := by
with_unfolding_all rfl
```

7.6.6.2. Modifying Reducibility

The reducibility of a definition can be globally modified in the module in which it is defined by applying the appropriate attribute with the `attribute` command. In other modules, the reducibility of imported definitions can be modified by applying the attribute with the `local` modifier. The `seal` and `unseal` commands are a shorthand for this process.

syntax

Local Irreducibility

The `seal foo` command ensures that the definition of `foo` is sealed, meaning it is marked as `[irreducible]`. This command is particularly useful in contexts where you want to prevent the reduction of `foo` in proofs.

In terms of functionality, `seal foo` is equivalent to `attribute [local irreducible] foo`. This attribute specifies that `foo` should be treated as irreducible only within the local scope, which helps in maintaining the desired abstraction level without affecting global settings.

```
command ::= ...
    | seal ident ident*
```

syntax

Local Reducibility

The `unseal foo` command ensures that the definition of `foo` is unsealed, meaning it is marked as `[semireducible]`, the default reducibility setting. This command is useful when you need to allow some level of reduction of `foo` in proofs.

Functionally, `unseal foo` is equivalent to `attribute [local semireducible] foo`. Applying this attribute makes `foo` semireducible only within the local scope.

```
command ::= ...
    | unseal ident ident*
```

7.6.6.3. Options

For performance, the elaborator and many tactics construct indices and caches. Many of these take reducibility into account, and there's no way to invalidate and regenerate them if reducibility changes globally. Unsafe changes to reducibility settings that could have unpredictable results are disallowed by default, but they can be enabled by using the `allowUnsafeReducibility` option.

option

`allowUnsafeReducibility`

Default value: `false`

enables users to modify the reducibility settings for declarations even when such changes are deemed potentially hazardous. For example, `simp` and type class resolution maintain term indices where reducible declarations are expanded.

8. Axioms

Axioms are postulated constants. While the axiom's type must itself be a type (that is, it must have type `Sort u`), there are no further requirements. Axioms do not reduce to other terms.

Axioms can be used to experiment with the consequences of an idea before investing the time required to construct a model or prove a theorem. They can also be used to adopt reasoning principles that can't otherwise be accessed in Lean's type theory; Lean itself provides three such axioms that are known to be consistent. However, axioms should be used with caution: axioms that are inconsistent with one another, or just false, undermine the very foundations of proofs. Lean automatically tracks the axioms that each proof depends on so that they can be audited.

8.1. Axiom Declarations

Axioms declarations include a name and a type:

syntax

Axiom Declarations

```
axiom ::= ...
    | axiom declId declSig
```

Axioms declarations may be modified with all possible declaration modifiers. Documentation comments, attributes, `private`, and `protected` have the same meaning as for other declarations. The modifiers `partial`, `nonrec`, `noncomputable` and `unsafe` have no effect.

8.2. Consistency

Using axioms is risky. Because they introduce a new constant of any type, and an inhabitant of a type that is a proposition counts as a proof of the proposition, axioms can be used to prove even false propositions. Any proof that relies on an axiom can be trusted only to the extent that the axiom is both true and consistent with the other axioms used. By their very nature, Lean cannot check whether new axioms are consistent; please exercise care when adding axioms.

▼ *Example: Inconsistencies From Axioms*

Axioms may introduce inconsistency, either alone or in combination with other axioms.

Assuming a false statement allows any statement at all to be proved:

```
axiom false_is_true : False

theorem two_eq_five : 2 = 5 := false_is_true.elim
```

Inconsistency may also arise from axioms that are incompatible with other properties of Lean. For example, parametricity is a powerful reasoning technique when used in languages that support it, but it is not compatible with Lean's standard axioms. If parametricity held, then the “free theorem” from the introduction to Wadler's *Theorems for Free* (1989), which describes a technique for using parametricity to derive theorems about polymorphic functions, would be true. As an axiom, it reads:

```
axiom List.free_theorem {α β}
  (f : {α : _} → List α → List α) (g : α → β) :
  f ∘ (List.map g) = (List.map g) ∘ f
```

However, a consequence of excluded middle is that all propositions are decidable; this means that a function can *check* whether they are true or false. This function can't be compiled, but it still exists. This can be used to define polymorphic functions that are not parametric:

```
open Classical in
noncomputable def nonParametric
    {α : _} (xs : List α) :
```

```
    List α :=
  if α = Nat then [] else xs
```

The existence of this function contradicts the “free theorem”:

```
theorem unit_not_nat : Unit ≠ Nat := by
  intro eq
  have ⟨allEq⟩ := eq ▸ inferInstanceAs (Subsingleton Unit)
  specialize allEq 0 1
  contradiction

example : False := by
  have := List.free_theorem nonParametric (fun () => 42)

  unfold nonParametric at this
  simp [unit_not_nat] at this

  have := congrFun this [()]
  contradiction
```

8.3. Reduction

Even consistent axioms can cause difficulties. Definitional equality identifies terms modulo reduction rules. The ι-reduction rule specifies the interaction of recursors and constructors; because axioms are not constructors, it does not apply to them. Ordinarily, terms without free variables reduce to applications of constructors, but axioms can cause them to get “stuck,” resulting in large terms.

▼ *Example: Axioms and Stuck Reduction*

Adding an addtional `0` to `Nat` with an axiom results in some definitional reductions getting stuck. In this example, two `Nat.succ` constructors are successfully moved to the outside of the term by reduction, but `Nat.rec` is unable to make further progress after encountering `Nat.otherZero`.

```
axiom Nat.otherZero : Nat

#reduce 4 + (Nat.otherZero + 2)
```

```
((Nat.rec ⟨fun x => x, PUnit.unit⟩ (fun n n_ih => ⟨fun x =>
(n_ih.1 x).succ, n_ih⟩) Nat.otherZero).1 4).succ.succ
```

Furthermore, the Lean compiler is not able to generate code for axioms. At runtime, Lean values must be represented by concrete data in memory, but axioms do not have a concrete representation. Definitions that contain non-proof code that relies on axioms must be marked `noncomputable` and can't be compiled.

▼ *Example: Axioms and Compilation*

Adding an addtional `0` to `Nat` with an axiom makes it so functions that use it can't be compiled. In particular, `List.length'` returns the axiom `Nat.otherZero` instead of `Nat.zero`. The function is marked `partial` to avoid further error messages from termination checking.

```
axiom Nat.otherZero : Nat

partial def List.length' : List α → Nat
  | [] => Nat.otherZero
  | _ :: xs => xs.length' + 1
```

```
axiom 'Nat.otherZero' not supported by code generator;
consider marking definition as 'noncomputable'
```

Axioms used in proofs rather than programs do not prevent a function from being compiled. The compiler does not generate code for proofs, so axioms in proofs are no problem. `nextOdd` computes the next odd number from a `Nat`, which may be the number itself or one greater:

```
def nextOdd (k : Nat) :
    { n : Nat // n % 2 = 1 ∧ (n = k ∨ n = k + 1) } where
  val := if k % 2 = 1 then k else k + 1
  property := by
    by_cases k % 2 = 1 <;>
    simp [*] <;> omega
```

The `by_cases` tactic uses the Law of the Excluded Middle, which is proved in the standard library using the Axiom of Choice. Indeed, this definition uses three axioms:

```
#print axioms nextOdd
```

```
'nextOdd' depends on axioms: [propext, Classical.choice,
Quot.sound]
```

Because they occur only in a proof, the compiler has no problem generating code:

```
#eval (nextOdd 4, nextOdd 5)
```

```
(5, 5)
```

8.4. Displaying Axiom Dependencies

The command `#print axioms` displays all the axioms that a declaration transitively relies on. In other words, if a proof uses another proof, which itself uses an axiom, then the axiom is reported by `#print axioms` for both. This can be used to audit the assumptions made by a proof. Together with `#guard_msgs`, it can also ensure that updates to libraries from other projects don't silently introduce unwanted dependencies on axioms.

8.5. Standard Axioms

Lean includes the following mathematical axioms:

- `propext {a b : Prop} : (a ↔ b) → a = b`
- `Classical.choice.{u} {α : Sort u} : Nonempty α → α`
- `Quot.sound.{u} {α : Sort u}`
  `{r : α → α → Prop} {a b : α} :`
  `r a b → Quot.mk r a = Quot.mk r b`

Three additional axioms allow the Lean kernel to invoke code generated by the compiler, rather than using its internal reduction engine. This can greatly improve performance of implementations of proof by reflection. Rather than using these axioms directly, they are usually invoked via the **`native_decide`** tactic. Both `Lean.reduceBool` and `Lean.reduceNat` contain references to `Lean.trustCompiler`, which ensures that the fact that a proof trusts the correctness of the compiler is tracked.

These axioms do not truly exist for their *mathematical* content; after all, `Lean.reduceBool` and `Lean.reduceNat` are essentially the identity function. However, they allow the use of compiled code in proofs to be carefully controlled, and tracking them as axioms allows `#print axioms` to be used to audit theorems.

- `Lean.trustCompiler : True`
- `Lean.ofReduceBool (a b : Bool) : Lean.reduceBool a = b → a = b`
- `Lean.ofReduceNat (a b : Nat) : Lean.reduceNat a = b → a = b`

Finally, the axiom `sorryAx` is used as part of the implementation of the **`sorry`** tactic and **`sorry`** term. Uses of this axiom are not intended to occur in finished proofs, and this can be confirmed using `#print axioms`.

9. Attributes

Attributes are an extensible set of compile-time annotations on declarations. They can be added as a declaration modifier or using the `attribute` command.

Attributes can associate information with declarations in compile-time tables (including custom simp sets, macros, and instances), impose additional requirements on definitions (e.g. rejecting them if their type is not a type class), or generate additional code. As with macros and custom elaborators for terms, commands, and tactics, the syntax category `attr` of attributes is designed to be extended, and there is a table that maps each extension to a compile-time program that interprets it.

Attributes are applied as *attribute instances* that pair a scope indicator with an attribute. These may occur either in attributes as declaration modifiers or the stand-alone `attribute` command.

syntax

Attribute Instances

```
attrInstance ::= ...
    | attrKind attr
```

An `attrKind` is the optional attribute scope keywords **`local`** or **`scoped`**. These control the visibility of the attribute's effects. The attribute itself is anything from the extensible syntax category `attr`.

The attribute system is very powerful: attributes can associate arbitrary information with declarations and generate any number of helpers. This imposes some design trade-offs: storing this information takes space, and retrieving it takes time. As a result, some attributes can only be applied to a declaration in the module where the declaration is defined. This allows lookups to be much faster in large projects, because they don't need to examine data for all modules. Each attribute determines how to store its own metadata and what the appropriate tradeoff between flexibility and performance is for a given use case.

9.1. Attributes as Modifiers

Attributes can be added to declarations as a declaration modifier. They are placed between the documentation comment and the visibility modifiers.

syntax

Attributes

```
attributes ::=
    @[attrInstance,*]
```

9.2. The `attribute` Command

The **`attribute`** command can be used to modify a declaration's attributes. Some example uses include:

- registering a pre-existing declaration as an instance in the local scope by adding **`instance`**,
- marking a pre-existing theorem as a simp lemma or an extensionality lemma, using **`simp`** or **`ext`**, and
- temporarily removing a simp lemma from the default simp set.

syntax

Attribute Modification

The `attribute` command adds or removes attributes from an existing declaration. The identifier is the name whose attributes are being modified.

```
command ::= ...
    | attribute [(eraseAttr | attrInstance),*] ident
```

In addition to attribute instances that add an attribute to an existing declaration, some attributes can be removed; this is called *erasing* the attribute. Attributes can be erased by preceding their name with −. Not all attributes support erasure, however.

syntax

Erasing Attributes

Attributes are erased by preceding their name with a −.

```
eraseAttr ::= ...
    | -ident
```

9.3. Scoped Attributes

Many attributes can be applied in a particular scope. This determines whether the attribute's effect is visible only in the current section scope, in namespaces that open the current namespace, or everywhere. These scope indications are also used to control syntax extensions and type class instances. Each attribute is responsible for defining precisely what these terms mean for its particular effect.

syntax

Attribute Scopes

Globally-scoped declarations (the default) are in effect whenever the module in which they're established is transitively imported. They are indicated by the absence of another scope modifier.

```
attrKind ::=
```

Locally-scoped declarations are in effect only for the extent of the section scope in which they are established.

```
attrKind ::= ...
    | local
```

Scoped declarations are in effect whenever the namespace in which they are established is opened.

```
attrKind ::= ...
    | scoped
```

10. Terms

Terms are the principal means of writing mathematics and programs in Lean. The elaborator translates them to Lean's minimal core language, which is then checked by the kernel and compiled for execution. The syntax of terms is arbitrarily extensible; this chapter documents the term syntax that Lean provides out-of-the-box.

10.1. Identifiers

syntax

Identifiers

```
$x:ident
```

An identifier term is a reference to a name. The specific lexical syntax of identifiers is described in the section on Lean's concrete syntax. Identifiers also occur in contexts where they bind names, such as `let` and `fun`; however, these binding occurrences are not complete terms in and of themselves. The mapping from identifiers to names is not trivial: at any point in a module, some number of namespaces will be open, there may be section variables, and there may be local bindings. Furthermore, identifiers may contain multiple dot-separated atomic identifiers; the dot both separates namespaces from their contents and variables from fields or functions that use field notation. This creates ambiguity, because an identifier `A.B.C.D.e.f` could refer to any of the following:

- A name `f` in the namespace `A.B.C.D.e` (for instance, a function defined in `e`'s `where` block).
- An application of `T.f` to `A.B.C.D.e` if `A.B.C.D.e` has type `T`
- A projection of field `f` from a structure named `A.B.C.D.e`
- A series of field projections `B.C.D.e` from structure value `A`, followed by an application of `f` using field notation
- If namespace `Q` is opened, it could be a reference to any of the above with a `Q` prefix, such as a name `f` in the namespace `Q.A.B.C.D.e`

This list is not exhaustive. Given an identifier, the elaborator must discover which name or names an identifier refers to, and whether any of the trailing components are fields or functions applied via field notation. This is called *resolving* the name.

Some declarations in the global environment are lazily created the first time they are referenced. Resolving an identifier in a way that both creates one of these declarations and results in a reference to it is called *realizing* the name. The rules for resolving and realizing a name are the same, so even

though this section refers only to resolving names, it applies to both.

Name resolution is affected by the following:

- Pre-resolved names attached to the identifier
- The macro scopes attached to the identifier
- The local bindings in scope, including auxiliary definitions created as part of the elaboration of `let rec`.
- Aliases created with `export` in modules transitively imported by the current module
- The current section scope, in particular the current namespace, opened namespaces, and section variables

Any prefix of an identifier can resolve to a set of names. The suffix that was not included in the resolution process is then treated as field projections or field notation. Resolutions of longer prefixes take precedence over resolutions of shorter prefixes; in other words, as few components as of the identifier as possible are treated as field notation. An identifier prefix may refer to any of the following, with earlier items taking precedence over later ones:

1. A locally-bound variable whose name is identical to the identifier prefix, including macro scopes, with closer local bindings taking precedence over outer local bindings.
2. A local auxiliary definition whose name is identical to the identifier prefix
3. A section variable whose name is identical to the identifier prefix
4. A global name that is identical to a prefix of the current namespace appended to the identifier prefix, or for which an alias exists in a prefix of the current namespace, with longer prefixes of the current namespace taking precedence over shorter ones
5. A global name that has been brought into scope via `open` commands that is identical to the identifier prefix

If an identifier resolves to multiple names, then the elaborator attempts to use all of them. If exactly one of them succeeds, then it is used as the meaning of the identifier. It is an error if more than one succeed or if all fail.

▼ *Example: Local Names Take Precedence*

Local bindings take precedence over global bindings:

```
def x := "global"

#eval
  let x := "local"
  x
```

```
"local"
```

The innermost local binding of a name takes precedence over others:

```
#eval
  let x := "outer"
  let x := "inner"
  x
```

```
"inner"
```

▼ *Example: Longer Prefixes of Current Namespace Take Precedence*

The namespaces A, B, and C are nested, and A and C each contain a definition of x.

```
namespace A
def x := "A.x"
namespace B
namespace C
def x := "A.B.C.x"
```

When the current namespace is A.B.C, x resolves to A.B.C.x.

```
#eval x
```

```
"A.B.C.x"
```

When the current namespace is A.B, x resolves to A.x.

```
end C
#eval x
```

```
"A.x"
```

▼ *Example: Longer Identifier Prefixes Take Precedence*

When an identifier could refer to different projections from names, the one with the longest name takes precedence:

```
structure A where
  y : String
deriving Repr

structure B where
  y : A
deriving Repr

def y : B := ⟨⟨"shorter"⟩⟩
def y.y : A := ⟨"longer"⟩
```

Given the above declarations, `y.y.y` could in principle refer either to the `y` field of the `y` field of `y`, or to the `y` field of `y.y`. It refers to the `y` field of `y.y`, because the name `y.y` is a longer prefix of `y.y.y` than the name `y`:

```
#eval y.y.y
```

```
"longer"
```

▼ *Example: Current Namespace Contents Take Precedence Over Opened Namespaces*

When an identifier could refer either to a name defined in a prefix of the current namespace or to an opened namespace, the former takes precedence.

```
namespace A
def x := "A.x"
end A

namespace B
def x := "B.x"
namespace C
open A
#eval x
```

Even though `A` was opened more recently than the declaration of `B.x`, the identifier `x` resolves to `B.x` rather than `A.x` because `B` is a prefix of the current namespace `B.C`.

```
#eval x
```

```
"B.x"
```

▼ *Example: Ambiguous Identifiers*

In this example, `x` could refer either to `A.x` or `B.x`, and neither takes precedence. Because both have the same type, it is an error.

```
def A.x := "A.x"
def B.x := "B.x"
open A
open B
#eval x
```

```
Ambiguous term
  x
Possible interpretations:
  B.x : String

  A.x : String
```

▼ *Example: Disambiguation via Typing*

When otherwise-ambiguous names have different types, the types are used to disambiguate:

```
def C.x := "C.x"
def D.x := 3
open C
open D
#eval (x : String)
```

```
"C.x"
```

10.1.1. Leading `.`

When an identifier beings with a dot (`.`), the type that the elaborator expects for the expression is used to resolve it, rather than the current namespace and set of open namespaces. Generalized field notation is related: leading dot notation uses the expect type of the identifier to resolve it to a name, while field notation uses the inferred type of the term immediately prior to the dot.

Identifiers with a leading `.` are to be looked up in the *expected type's namespace*. If the type expected for a term is a constant applied to zero or more arguments, then its namespace is the constant's name. If the type is not an application of a constant (e.g., a function, a metavariable, or a universe) then it doesn't have a namespace.

If the name is not found in the expected type's namespace, but the constant can be unfolded to yield another constant, then its namespace is consulted. This process is repeated until something other than an application of a constant is encountered, or until the constant can't be unfolded.

▼ *Example: Leading `.`*

The expected type for `.replicate` is `List Unit`. This type's namespace is `List`, so `.replicate` resolves to `List.replicate`.

```
#eval show List Unit from .replicate 3 ()
```

```
[(), (), ()]
```

▼ *Example: Leading `.` and Unfolding Definitions*

The expected type for `.replicate` is `MyList Unit`. This type's namespace is `MyList`, but there is no definition `MyList.replicate`. Unfolding `MyList Unit` yields `List Unit`, so `.replicate` resolves to `List.replicate`.

```
def MyList α := List α
#eval show MyList Unit from .replicate 3 ()
```

```
[(), (), ()]
```

10.2. Function Types

Lean's function types describe more than just the function's domain and codomain. They also provide instructions for elaborating application sites by indicating that some parameters are to be discovered automatically via unification or type class synthesis, that others are optional with default values, and that yet others should be synthesized using a custom tactic script. Furthermore, their syntax contains support for abbreviating curried functions.

syntax

Function types

Dependent function types include an explicit name:

```
term ::= ...
    | (ident : term) → term
```

Non-dependent function types do not:

```
term ::= ...
    | term → term
```

syntax

Curried Function Types

Dependent function types may include multiple parameters that have the same type in a single set of parentheses:

```
term ::= ...
    | (ident* : term) → term
```

This is equivalent to repeating the type annotation for each parameter name in a nested function type.

syntax

Implicit, Optional, and Auto Parameters

Function types can describe functions that take implicit, instance implicit, optional, and automatic parameters. All but instance implicit parameters require one or more names.

```
term ::= ...
    | (ident* : term := term) → term
```

```
term ::= ...
    | (ident* : term := by tacticSeq) → term

term ::= ...
    | {ident* : term} → term

term ::= ...
    | [term] → term

term ::= ...
    | [ident : term] → term

term ::= ...
    | ⦃ident* : term⦄ → term
```

▼ *Example: Multiple Parameters, Same Type*

The type of Nat.add can be written in the following ways:

- Nat → Nat → Nat
- (a : Nat) → (b : Nat) → Nat
- (a b : Nat) → Nat

The last two types allow the function to be used with named arguments; aside from this, all three are equivalent.

10.3. Functions

Terms with function types can be created via abstractions, introduced with the `fun` keyword. (i) In various communities, function abstractions are also known as *lambdas*, due to Alonzo Church's notation for them, or *anonymous functions* because they don't need to be defined with a name in the global environment. While abstractions in the core type theory only allow a single variable to be bound, function terms are quite flexible in the high-level Lean syntax.

syntax

Function Abstraction

The most basic function abstraction introduces a variable to stand for the function's parameter:

```
term ::= ...
    | fun ident => term
```

At elaboration time, Lean must be able to determine the function's domain. A type ascription is one way to provide this information:

```
term ::= ...
    | fun ident : term => term
```

Function definitions defined with keywords such as `def` desugar to `fun`. Inductive type declarations, on the other hand, introduce new values with function types (constructors and type constructors) that cannot themselves be implemented using just `fun`.

syntax

Curried Functions

Multiple parameter names are accepted after after `fun`:

```
term ::= ...
    | fun ident ident* => term

term ::= ...
    | fun ident ident* : term => term
```

Different type annotations for multiple parameters requires parentheses:

```
term ::= ...
    | fun (ident* : term) =>term
```

These are equivalent to writing nested `fun` terms.

The => may be replaced by ↦ in all of the syntax described in this section.

Function abstractions may also use pattern matching syntax as part of their parameter specification, avoiding the need to introduce a local variable that is immediately destructured. This syntax is described in the section on pattern matching.

10.3.1. Implicit Parameters

Lean supports implicit parameters to functions. This means that Lean itself can supply arguments to functions, rather than requiring users to supply all needed arguments. Implicit parameters come in three varieties:

Ordinary implicit parameters

Ordinary implicit parameters are function parameters that Lean should determine values for via unification. In other words, each call site should have exactly one potential argument value that would cause the function call as a whole to be well-typed. The Lean elaborator attempts to find values for all implicit arguments at each occurrence of a function. Ordinary implicit parameters are written in curly braces ({ and }).

Strict implicit parameters

Strict implicit parameters are identical to ordinary implicit parameters, except Lean will only attempt to find argument values when subsequent explicit arguments are provided at a call site. Strict implicit parameters are written in double curly braces (`{⦃` and `⦄`, or `{{` and `}}`).

Instance implicit parameters

Arguments for *instance implicit* parameters are found via type class synthesis. Instance implicit parameters are written in square brackets (`[` and `]`). Unlike the other kinds of implicit parameter, instance implicit parameters that are written without a `:` specify the parameter's type rather than providing a name. Furthermore, only a single name is allowed. Most instance implicit parameters omit the parameter name because instances synthesized as parameters to functions are already available in the functions' bodies, even without being named explicitly.

▼ *Example: Ordinary vs Strict Implicit Parameters*

The difference between the functions `f` and `g` is that `α` is strictly implicit in `f`:

```
def f ⦃α : Type⦄ : α → α := fun x => x
def g {α : Type} : α → α := fun x => x
```

These functions are elaborated identically when applied to concrete arguments:

```
example : f 2 = g 2 := rfl
```

However, when the explicit argument is not provided, uses of `f` do not require the implicit `α` to be solved:

```
example := f
```

However, uses of `g` do require it to be solved, and fail to elaborate if there is insufficient information available:

```
example := g
```

```
don't know how to synthesize implicit argument 'α'
  @g ?m.6
context:
⊢ Type
```

syntax

Functions with Varying Binders

The most general syntax for `fun` accepts a sequence of binders:

```
term ::= ...
    | fun funBinder funBinder* => term
```

syntax

Function Binders

Function binders may be identifiers:

```
funBinder ::= ...
    | ident
```

parenthesized sequences of identifiers:

```
funBinder ::= ...
    | (ident ident*)
```

sequences of identifiers with a type ascription:

```
funBinder ::= ...
    | (ident ident* : term)
```

implicit parameters, with or without a type ascription:

```
funBinder ::= ...
    | {ident ident*}
```

```
funBinder ::= ...
    | {ident ident* : term}
```

instance implicits, anonymous or named:

```
funBinder ::= ...
    | [term]
```

```
funBinder ::= ...
    | [ident : term]
```

or strict implicit parameters, with or without a type ascription:

```
funBinder ::= ...
    | ⦃ident ident*⦄
```

```
funBinder ::= ...
    | ⦃ident* : term⦄
```

As usual, an _ may be used instead of an identifier to create an anonymous parameter, and ⦃ and ⦄ may alternatively be written using {{ and }}, respectively.

Lean's core language does not distinguish between implicit, instance, and explicit parameters: the various kinds of function and function type are definitionally equal. The differences can be observed only during elaboration.

If the expected type of a function includes implicit parameters, but its binders do not, then the resulting function may end up with more parameters than the binders indicated in the code. This is because the implicit parameters are added automatically.

▼ *Example: Implicit Parameters from Types*

The identity function can be written with a single explicit parameter. As long as its type is known, the implicit type parameter is added automatically.

```
#check (fun x => x : {α : Type} → α → α)
```

```
fun {α} x => x : {α : Type} → α → α
```

The following are all equivalent:

```
#check (fun {α} x => x : {α : Type} → α → α)
```

```
fun {α} x => x : {α : Type} → α → α
```

```
#check (fun {α} (x : α) => x : {α : Type} → α → α)
```

```
fun {α} x => x : {α : Type} → α → α
```

```
#check (fun {α : Type} (x : α) => x : {α : Type} → α → α)
```

```
fun {α} x => x : {α : Type} → α → α
```

10.4. Function Application

Ordinarily, function application is written using juxtaposition: the argument is placed after the function, with at least one space between them. In Lean's type theory, all functions take exactly one argument and produce exactly one value. All function applications combine a single function with a single argument. Multiple arguments are represented via currying.

The high-level term language treats a function together with one or more arguments as a single unit, and supports additional features such as implicit, optional, and by-name arguments along with ordinary positional arguments. The elaborator converts these to the simpler model of the core type theory.

syntax

Function Application

A function application consists of a term followed by one or more arguments, or by zero or more arguments and a final ellipsis.

```
term ::= ...
    | term argument+
```

```
    | term argument* ..
```

syntax

Arguments

Function arguments are either terms or named arguments.

```
argument ::= ...
    | term
    | ((ident | _:ident) :=term)
```

The function's core-language type determines the placement of the arguments in the final expression. Function types include names for their expected parameters. In Lean's core language, non-dependent function types are encoded as dependent function types in which the parameter name does not occur in the body. Furthermore, they are chosen internally such that they cannot be written as the name of a named argument; this is important to prevent accidental capture.

Each parameter expected by the function has a name. Recurring over the function's argument types, arguments are selected from the sequence of arguments as follows:

- If the parameter's name matches the name provided for a named argument, then that argument is selected.
- If the parameter is implicit, a fresh metavariable is created with the parameter's type and selected.
- If the parameter is instance implicit, a fresh instance metavariable is created with the parameter's type and inserted. Instance metavariables are scheduled for later synthesis.
- If the parameter is a strict implicit parameter and there are any named or positional arguments that have not yet been selected, a fresh metavariable is created with the parameter's type and selected.
- If the parameter is explicit, then the next positional argument is selected and elaborated. If there are no positional arguments:
    - If the parameter is declared as an optional parameter, then its default value is selected as the argument.
    - If the parameter is an automatic parameter then its associated tactic script is executed to construct the argument.
    - If the parameter is neither optional nor automatic, and no ellipsis is present, then a fresh variable is selected as the argument. If there is an ellipsis, a fresh metavariable is selected as if the argument were implicit.

As a special case, when the function application occurs in a pattern and there is an ellipsis, optional and automatic arguments become universal patterns (_) instead of being inserted.

It is an error if the type is not a function type and arguments remain. After all arguments have been inserted and there is an ellipsis, then the missing arguments are all set to fresh metavariables, just as if they were implicit arguments. If any fresh variables were created for missing explicit positional arguments, the entire application is wrapped in a `fun` term that binds them. Finally, instance synthesis is invoked and as many metavariables as possible are solved:

1. A type is inferred for the entire function application. This may cause some metavariables to be solved due to unification that occurs during type inference.
2. The instance metavariables are synthesized. Default instances are only used if the inferred type is a metavariable that is the output parameter of one of the instances.
3. If there is an expected type, it is unified with the inferred type; however, errors resulting from this unification are discarded. If the expected and inferred types can be equal, unification can solve leftover implicit argument metavariables. If they can't be equal, an error is not thrown because a surrounding elaborator may be able to insert coercions or monad lifts.

▼ *Example: Named Arguments*

The `#check` command can be used to inspect the arguments that were inserted for a function call.

The function `sum3` takes three explicit `Nat` parameters, named `x`, `y`, and `z`.

```
def sum3 (x y z : Nat) : Nat := x + y + z
```

All three arguments can be provided positionally.

```
#check sum3 1 3 8
```

```
sum3 1 3 8 : Nat
```

They can also be provided by name.

```
#check sum3 (x := 1) (y := 3) (z := 8)
```

```
sum3 1 3 8 : Nat
```

When arguments are provided by name, it can be in any order.

```
#check sum3 (y := 3) (z := 8) (x := 1)
```

```
sum3 1 3 8 : Nat
```

Named and positional arguments may be freely intermixed.

```
#check sum3 1 (z := 8) (y := 3)
```

```
sum3 1 3 8 : Nat
```

Named and positional arguments may be freely intermixed. If an argument is provided by name, it is used, even if it occurs after a positional argument that could have been used.

```
#check sum3 1 (x := 8) (y := 3)
```

```
sum3 8 3 1 : Nat
```

If a named argument is to be inserted after arguments that aren't provided, a function is created in which the provided argument is filled out.

```
#check sum3 (z := 8)
```

```
fun x y => sum3 x y 8 : Nat → Nat → Nat
```

Behind the scenes, the names of the arguments are preserved in the function type. This means that the remaining arguments can again be passed by name.

```
#check (sum3 (z := 8)) (y := 1)
```

```
fun x => (fun x y => sum3 x y 8) x 1 : Nat → Nat
```

Optional and automatic parameters are not part of Lean's core type theory. They are encoded using the `optParam` and `autoParam` gadgets.

def

```
optParam.{u} (α : Sort u) (default : α) : Sort u
```

Gadget for optional parameter support.

A binder like `(x : α := default)` in a declaration is syntax sugar for `x : optParam α default`, and triggers the elaborator to attempt to use `default` to supply the argument if it is not supplied.

def

```
autoParam.{u} (α : Sort u) (tactic : Lean.Syntax) : Sort u
```

Gadget for automatic parameter support. This is similar to the `optParam` gadget, but it uses the given tactic. Like `optParam`, this gadget only affects elaboration. For example, the tactic will *not* be invoked during type class resolution.

10.4.1. Generalized Field Notation

The section on structure fields describes the notation for projecting a field from a term whose type is a structure. Generalized field notation consists of a term followed by a dot (`.`) and an identifier, not separated by spaces.

syntax

Field Notation

```
term ::= ...
    | term.ident
```

If a term's type is a constant applied to zero or more arguments, then field notation can be used to apply a function to it, regardless of whether the term is a structure or type class instance that has fields. The use of field notation to apply other functions is called *generalized field notation*.

The identifier after the dot is looked up in the namespace of the term's type, which is the constant's name. If the type is not an application of a constant (e.g., a function, a metavariable, or a universe) then it doesn't have a namespace and generalized field notation cannot be used. If the field is not found, but the constant can be unfolded to yield a further type which is a constant or application of a constant, then the process is repeated with the new constant.

When a function is found, the term before the dot becomes an argument to the function. Specifically, it becomes the first explicit argument that would not be a type error. Aside from that, the application is elaborated as usual.

▼ *Example: Generalized Field Notation*

The type `Username` is a constant, so functions in the `Username` namespace can be applied to terms with type `Username` with generalized field notation.

```
def Username := String
```

One such function is `Username.validate`, which checks that a username contains no leading whitespace and that only a small set of acceptable characters are used. In its definition, generalized field notation is used to call the functions `String.isPrefixOf`, `String.any`, `Char.isAlpha`, and `Char.isDigit`. In the case of `String.isPrefixOf`, which takes two `String` arguments, `" "` is used as the first because it's the term before the dot. `String.any` can be called on *name* using generalized field notation even though it has type `Username` because `Username.any` is not defined and `Username` unfolds to `String`.

```
def Username.validate (name : Username) : Except String Unit :=
do
  if " ".isPrefixOf name then
    throw "Unexpected leading whitespace"
  if name.any notOk then
    throw "Unexpected character"
  return ()
where
  notOk (c : Char) : Bool :=
    !c.isAlpha &&
    !c.isDigit &&
    !c ∈ ['_', ' ']

def root : Username := "root"
```

However, `Username.validate` can't be called on `"root"` using field notation, because `String` does not unfold to `Username`.

```
#eval "root".validate
```

```
Invalid field `validate`: The environment does not contain
`String.validate`
  "root"
has type
  String
```

`root`, on the other hand, has type `Username`:

```
#eval root.validate
```

```
Except.ok ()
```

option

pp.fieldNotation

Default value: `true`

(pretty printer) use field notation when pretty printing, including for structure projections, unless '@[pp_nodot]' is applied

attribute

Controlling Field Notation

The **pp_nodot** attribute causes Lean's pretty printer to not use field notation when printing a function.

```
attr ::= ...
    | pp_nodot
```

▼ *Example: Turning Off Field Notation*

`Nat.half` is printed using field notation by default.

```
def Nat.half : Nat → Nat
  | 0 | 1 => 0
  | n + 2 => n.half + 1
```

```
#check Nat.half Nat.zero
```

```
Nat.zero.half : Nat
```

Adding **pp_nodot** to Nat.half causes ordinary function application syntax to be used instead when displaying the term.

```
attribute [pp_nodot] Nat.half

#check Nat.half Nat.zero
```

```
Nat.half Nat.zero : Nat
```

10.4.2. Pipeline Syntax

Pipeline syntax provides alternative ways to write function applications. Repeated pipelines use parsing precedence instead of nested parentheses to nest applications of functions to positional arguments.

syntax

Pipelines

Right pipe notation applies the term to the right of the pipe to the one on its left.

```
term ::= ...
    | term |> term
```

Left pipe notation applies the term on the left of the pipe to the one on its right.

```
term ::= ...
    | term <| term
```

The intuition behind right pipeline notation is that the values on the left are being fed to the first function, its results are fed to the second one, and so forth. In left pipeline notation, values on the right are fed leftwards.

▼ *Example: Right pipeline notation*

Right pipelines can be used to call a series of functions on a term. For readers, they tend to emphasize the data that's being transformed.

```
#eval "Hello!" |> String.toList |> List.reverse |> List.head!
```

```
'!'
```

▼ *Example: Left pipeline notation*

Left pipelines can be used to call a series of functions on a term. They tend to emphasize the functions over the data.

```
#eval List.head! <| List.reverse <| String.toList <| "Hello!"
```

```
'!'
```

syntax

Pipeline Fields

There is a version of pipeline notation that's used for generalized field notation.

```
term ::= ...
    | term |>.ident

term ::= ...
    | term |>.fieldIdx
```

`e |>.f arg` is an alternative syntax for `(e).f arg`.

▼ *Example: Pipeline Fields*

Some functions are inconvenient to use with pipelines because their argument order is not conducive. For example, `Array.push` takes an array as its first argument, not a `Nat`, leading to this error:

```
#eval #[1, 2, 3] |> Array.push 4
```

```
failed to synthesize
  OfNat (Array ?m.4) 4
numerals are polymorphic in Lean, but the numeral `4` cannot
be used in a context where the expected type is
  Array ?m.4
due to the absence of the instance above

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Using pipeline field notation causes the array to be inserted at the first type-correct position:

```
#eval #[1, 2, 3] |>.push 4
```

```
#[1, 2, 3, 4]
```

This process can be iterated:

```
#eval #[1, 2, 3] |>.push 4 |>.reverse |>.push 0 |>.reverse
```

```
#[0, 1, 2, 3, 4]
```

10.5. Literals

There are two kinds of numeric literal: natural number literals and scientific literals. Both are overloaded via type classes.

10.5.1. Natural Numbers

When Lean encounters a natural number literal *n*, it interprets it via the overloaded method `OfNat.ofNat` *n*. A default instance of `OfNat Nat` *n* ensures that the type `Nat` can be inferred when no other type information is present.

type class

```
OfNat.{u} (α : Type u) : Nat → Type u
```

The class `OfNat` *α* *n* powers the numeric literal parser. If you write `37 : α`, Lean will attempt to synthesize `OfNat` *α* `37`, and will generate the term (`OfNat.ofNat` `37 :` *α*).

There is a bit of infinite regress here since the desugaring apparently still contains a literal `37` in it. The type of expressions contains a primitive constructor for "raw natural number literals", which you can directly access using the macro **`nat_lit`** `37`. Raw number literals are always of type `Nat`. So it would be more correct to say that Lean looks for an instance of `OfNat` *α* (**`nat_lit`** `37`), and it generates the term (`OfNat.ofNat` (**`nat_lit`** `37`) `:` *α*).

Instance Constructor

`OfNat.mk.{u}`

Methods

`ofNat : α`

The `OfNat.ofNat` function is automatically inserted by the parser when the user writes a numeric literal like `1 : α`. Implementations of this typeclass can therefore customize the behavior of *n* `:` *α* based on *n* and *α*.

▼ *Example: Custom Natural Number Literals*

The structure `NatInterval` represents an interval of natural numbers.

```
structure NatInterval where
  low : Nat
```

```
  high : Nat
  low_le_high : low ≤ high

instance : Add NatInterval where
  add
    | ⟨lo1, hi1, le1⟩, ⟨lo2, hi2, le2⟩ => ⟨lo1 + lo2, hi1 + hi2,
by omega⟩
```

An `OfNat` instance allows natural number literals to be used to represent intervals:

```
instance : OfNat NatInterval n where
  ofNat := ⟨n, n, by omega⟩

#eval (8 : NatInterval)
```

```
{ low := 8, high := 8, low_le_high := _ }
```

There are no separate integer literals. Terms such as `-5` consist of a prefix negation (which can be overloaded via the `Neg` type class) applied to a natural number literal.

10.5.2. Scientific Numbers

Scientific number literals consist of a sequence of digits followed by an optional period and decimal part and an optional exponent. If no period or exponent is present, then the term is instead a natural number literal. Scientific numbers are overloaded via the `OfScientific` type class.

type class

```
OfScientific.{u} (α : Type u) : Type u
```

For decimal and scientific numbers (e.g., `1.23`, `3.12e10`). Examples:

- `1.23` is syntax for `OfScientific.ofScientific (nat_lit 123) true (nat_lit 2)`
- `121e100` is syntax for `OfScientific.ofScientific (nat_lit 121) false (nat_lit 100)`

Note the use of **`nat_lit`**; there is no wrapping `OfNat.ofNat` in the resulting term.

Instance Constructor

`OfScientific.mk.{u}`

Methods

`ofScientific : Nat → Bool → Nat → α`

Produces a value from the given mantissa, exponent sign, and decimal exponent. For the exponent sign, `true` indicates a negative exponent.

Examples:

- `1.23` is syntax for `OfScientific.ofScientific (nat_lit 123) true (nat_lit 2)`
- `121e100` is syntax for `OfScientific.ofScientific (nat_lit 121) false (nat_lit 100)`

Note the use of `nat_lit`; there is no wrapping `OfNat.ofNat` in the resulting term.

There are an `OfScientific` instances for `Float` and `Float32`, but no separate floating-point literals.

10.5.3. Strings

String literals are described in the chapter on strings.

10.5.4. Lists and Arrays

List and array literals contain comma-separated sequences of elements inside of brackets, with arrays prefixed by a hash mark (#). Array literals are interpreted as list literals wrapped in a call to a conversion. For performance reasons, very large list and array literals are converted to sequences of local definitions, rather than just iterated applications of the list constructor.

syntax

List Literals

```
term ::= ...
    | [term,*]
```

syntax

Array Literals

```
term ::= ...
    | #[term,*]
```

▼ *Example: Long List Literals*

This list contains 32 elements. The generated code is an iterated application of `List.cons`:

```
#check
  [1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1]
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] : List Nat
```

With 33 elements, the list literal becomes a sequence of local definitions:

```
#check
  [1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,
   1,1,1,1,1,1,1,1,
   1]
```

```
let y :=
  let y :=
    let y := [1, 1, 1, 1, 1];
    1 :: 1 :: 1 :: 1 :: y;
  let y := 1 :: 1 :: 1 :: 1 :: y;
  1 :: 1 :: 1 :: 1 :: y;
let y :=
  let y := 1 :: 1 :: 1 :: 1 :: y;
  1 :: 1 :: 1 :: 1 :: y;
let y := 1 :: 1 :: 1 :: 1 :: y;
1 :: 1 :: 1 :: 1 :: y : List Nat
```

10.6. Structures and Constructors

Anonymous constructors and structure instance syntax are described in their respective sections.

10.7. Conditionals

The conditional expression is used to check whether a proposition is true or false. (i) Despite their syntactic similarity, the `if` used in the tactic language and the `if` used in do-notation are separate syntactic forms, documented in their own sections. This requires that the proposition has a `Decidable` instance, because it's not possible to check whether *arbitrary* propositions are true or false. There is also a coercion from `Bool` to `Prop` that results in a decidable proposition (namely, that the `Bool` in question is equal to `true`), described in the section on decidability.

There are two versions of the conditional expression: one simply performs a case distinction, while the other additionally adds an assumption about the proposition's truth or falsity to the local context. This allows run-time checks to generate compile-time evidence that can be used to statically rule out errors.

syntax

Conditionals

Without a name annotation, the conditional expression expresses only control flow.

```
term ::= ...
    | if term then
        term
      else
        term
```

With the name annotation, the branches of the `if` have access to a local assumption that the proposition is respectively true or false.

```
term ::= ...
    | if binderIdent : term then
        term
      else
        term
```

▼ *Example: Checking Array Bounds*

Array indexing requires evidence that the index in question is within the bounds of the array, so `getThird` does not elaborate.

```
def getThird (xs : Array α) : α := xs[2]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed,
and 'Panic' error message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that
index is valid
α : Type ?u.7
xs : Array α
⊢ 2 < xs.size
```

Relaxing the return type to `Option` and adding a bounds check results the same error. This is because the proof that the index is in bounds was not added to the local context.

```
def getThird (xs : Array α) : Option α :=
  if xs.size ≤ 2 then none
  else xs[2]
```

```
failed to prove index is valid, possible solutions:
  - Use `have`-expressions to prove the index is valid
  - Use `a[i]!` notation instead, runtime check is performed,
and 'Panic' error message is produced if index is not valid
  - Use `a[i]?` notation instead, result is an `Option` type
  - Use `a[i]'h` notation instead, where `h` is a proof that
index is valid
α : Type ?u.7
```

```
xs : Array α
⊢ 2 < xs.size
```

Naming the proof h is sufficient to enable the tactics that perform bounds checking to succeed, even though it does not occur explicitly in the text of the program.

```
def getThird (xs : Array α) : Option α :=
  if h : xs.size ≤ 2 then none
  else xs[2]
```

There is also a pattern-matching version of `if`. If the pattern matches, then it takes the first branch, binding the pattern variables. If the pattern does not match, then it takes the second branch.

syntax

Pattern-Matching Conditionals

```
term ::= ...
    | if let term := term then
        term
      else
        term
```

If a `Bool`-only conditional statement is ever needed, the `bif` variant can be used.

syntax

Boolean-Only Conditional

```
term ::= ...
    | bif term then
        term
      else
        term
```

10.8. Pattern Matching

Pattern matching is a way to recognize and destructure values using a syntax of *patterns* that are a subset of the terms. A pattern that recognizes and destructures a value is similar to the syntax that would be used to construct the value. One or more *match discriminants* are simultaneously compared to a series of *match alternatives*. Discriminants may be named. Each alternative contains one or more comma-separated sequences of patterns; all pattern sequences must contain the same number of patterns as there are discriminants. When a pattern sequence matches all of the

discriminants, the term following the corresponding `=>` is evaluated in an environment extended with values for each pattern variable as well as an equality hypothesis for each named discriminant. This term is called the *right-hand side* of the match alternative.

syntax

Pattern Matching

```
term ::= ...
    | match
        ((generalizing := (trueVal | falseVal)))?
        ((motive := term))?
        matchDiscr,*
      with
      (| (term,*)|* => term)*
```

syntax

Match Discriminants

```
matchDiscr ::=
    term

matchDiscr ::= ...
    | ident : term
```

Pattern matching expressions may alternatively use quasiquotations as patterns, matching the corresponding `Lean.Syntax` values and treating the contents of antiquotations as ordinary patterns. Quotation patterns are compiled differently than other patterns, so if one pattern in a `match` is syntax, then all of them must be. Quotation patterns are described in the section on quotations.

Patterns are a subset of the terms. They consist of the following:

Catch-All Patterns

The hole syntax _ is a pattern that matches any value and binds no pattern variables. Catch-all patterns are not entirely equivalent to unused pattern variables. They can be used in positions where the pattern's typing would otherwise require a more specific inaccessible pattern, while variables cannot be used in these positions.

Identifiers

If an identifier is not bound in the current scope and is not applied to arguments, then it represents a pattern variable. *Pattern variables* match any value, and the values thus matched are bound to the pattern variable in the local environment in which the right-hand side is evaluated. If the identifier is bound, it is a pattern if it is bound to the constructor of an inductive type or if its definition has the **`match_pattern`** attribute.

Applications

Function applications are patterns if the function being applied is an identifier that is bound to a constructor or that has the **`match_pattern`** attribute and if all arguments are also patterns. If the identifier is a constructor, the pattern matches values built with that constructor if the argument patterns match the constructor's arguments. If it is a function with the **`match_pattern`** attribute, then the function application is unfolded and the resulting term's normal form is used as the pattern. Default arguments are inserted as usual, and their normal forms are used as patterns. Ellipses, however, result in all further arguments being treated as universal patterns, even those with associated default values or tactics.

Literals

Character literals and string literals are patterns that match the corresponding character or string. Raw string literals are allowed as patterns, but interpolated strings are not. Natural number literals in patterns are interpreted by synthesizing the corresponding `OfNat` instance and reducing the resulting term to normal form, which must be a pattern. Similarly, scientific literals are interpreted via the corresponding `OfScientific` instance. While `Float` has such an instance, `Float`s cannot be used as patterns because the instance relies on an opaque function that can't be reduced to a valid pattern.

Structure Instances

Structure instances may be used as patterns. They are interpreted as the corresponding structure constructor.

Quoted names

Quoted names, such as `` `x `` and ``` ``none ```, match the corresponding `Lean.Name` value.

Macros

Macros in patterns are expanded. They are patterns if the resulting expansions are patterns.

Inaccessible patterns

Inaccessible patterns are patterns that are forced to have a particular value by later typing constraints. Any term may be used as an inaccessible term. Inaccessible terms are parenthesized, with a preceding period (`.`).

syntax

Inaccessible Patterns

```
term ::= ...
    | .(term)
```

▼ *Example: Inaccessible Patterns*

A number's *parity* is whether it's even or odd:

```
inductive Parity : Nat → Type where
  | even (h : Nat) : Parity (h + h)
```

```
    | odd (h : Nat) : Parity ((h + h) + 1)

def Nat.parity (n : Nat) : Parity n :=
  match n with
  | 0 => .even 0
  | n' + 1 =>
    match n'.parity with
    | .even h => .odd h
    | .odd h =>
      have eq : (h + 1) + (h + 1) = (h + h + 1 + 1) :=
        by omega
      eq ▸ .even (h + 1)
```

Because a value of type Parity contains half of a number (rounded down) as part of its representation of evenness or oddness, division by two can be implemented (in an unconventional manner) by finding a parity and then extracting the number.

```
def half (n : Nat) : Nat :=
  match n, n.parity with
  | .(h + h),     .even h => h
  | .(h + h + 1), .odd h  => h
```

Because the index structure of Parity.even and Parity.odd force the number to have a certain form that is not otherwise a valid pattern, patterns that match on it must use inaccessible patterns for the number being divided.

Patterns may additionally be named. Named patterns associate a name with a pattern; in subsequent patterns and on the right-hand side of the match alternative, the name refers to the part of the value that was matched by the given pattern. Named patterns are written with an @ between the name and the pattern. Just like discriminants, named patterns may also be provided with names for equality assumptions.

syntax

Named Patterns

```
term ::= ...
    | ident@term

term ::= ...
    | ident@ident:term
```

10.8.1. Types

Each discriminant must be well typed. Because patterns are a subset of terms, their types can also be checked. Each pattern that matches a given discriminant must have the same type as the corresponding discriminant.

The right-hand side of each match alternative should have the same type as the overall `match` term. To support dependent types, matching a discriminant against a pattern refines the types that are expected within the scope of the pattern. In both subsequent patterns in the same match alternative and the right-hand side's type, occurrences of the discriminant are replaced by the pattern that it was matched against.

▼ *Example: Type Refinement*

This indexed family describes mostly-balanced trees, with the depth encoded in the type.

```
inductive BalancedTree (α : Type u) : Nat → Type u where
  | empty : BalancedTree α 0
  | branch
    (left : BalancedTree α n)
    (val : α)
    (right : BalancedTree α n) :
    BalancedTree α (n + 1)
  | lbranch
    (left : BalancedTree α (n + 1))
    (val : α)
    (right : BalancedTree α n) :
    BalancedTree α (n + 2)
  | rbranch
    (left : BalancedTree α n)
    (val : α)
    (right : BalancedTree α (n + 1)) :
    BalancedTree α (n + 2)
```

To begin the implementation of a function to construct a perfectly balanced tree with some initial element and a given depth, a hole can be used for the definition.

```
def BalancedTree.filledWith (x : α) (depth : Nat) : BalancedTree
α depth := _
```

The error message demonstrates that the tree should have the indicated depth.

```
don't know how to synthesize placeholder
context:
α : Type u
x : α
depth : Nat
⊢ BalancedTree α depth
```

Matching on the expected depth and inserting holes results in an error message for each hole. These messages demonstrate that the expected type has been refined, with `depth` replaced by the matched values.

```
def BalancedTree.filledWith
    (x : α) (depth : Nat) :
```

```
    BalancedTree α depth :=
  match depth with
  | 0 => _
  | n + 1 => _
```

The first hole yields the following message:

```
don't know how to synthesize placeholder
context:
α : Type u
x : α
depth : Nat
⊢ BalancedTree α 0
```

The second hole yields the following message:

```
don't know how to synthesize placeholder
context:
α : Type u
x : α
depth n : Nat
⊢ BalancedTree α (n + 1)
```

Matching on the depth of a tree and the tree itself leads to a refinement of the tree's type according to the depth's pattern. This means that certain combinations are not well-typed, such as `0` and `branch`, because refining the second discriminant's type yields `BalancedTree α 0` which does not match the constructor's type.

```
def BalancedTree.isPerfectlyBalanced
    (n : Nat) (t : BalancedTree α n) : Bool :=
  match n, t with
  | 0, .empty => true
  | 0, .branch left val right =>
    isPerfectlyBalanced left &&
    isPerfectlyBalanced right
  | _, _ => false
```

```
type mismatch
  left.branch val right
has type
  BalancedTree ?m.54 (?m.51 + 1) : Type ?u.47
but is expected to have type
  BalancedTree α 0 : Type u
```

10.8.1.1. Pattern Equality Proofs

When a discriminant is named, `match` generates a proof that the pattern and discriminant are equal, binding it to the provided name in the right-hand side. This is useful to bridge the gap between dependent pattern matching on indexed families and APIs that expect explicit propositional arguments, and it can help tactics that make use of assumptions to succeed.

▼ *Example: Pattern Equality Proofs*

The function `last?`, which either throws an exception or returns the last element of its argument, uses the standard library function `List.getLast`. This function expects a proof that the list in question is nonempty. Naming the match on `xs` ensures that there's an assumption in scope that states that `xs` is equal to `_ :: _`, which **`simp_all`** uses to discharge the goal.

```
def last? (xs : List α) : Except String α :=
  match h : xs with
  | [] =>
    .error "Can't take first element of empty list"
  | _ :: _ =>
    .ok <| xs.getLast (show xs ≠ [] by intro h'; simp_all)
```

Without the name, **`simp_all`** is unable to find the contradiction.

```
def last?' (xs : List α) : Except String α :=
  match xs with
  | [] =>
    .error "Can't take first element of empty list"
  | _ :: _ =>
    .ok <| xs.getLast (show xs ≠ [] by intro h'; simp_all)
```

```
simp_all made no progress
```

10.8.1.2. Explicit Motives

Pattern matching is not a built-in primitive of Lean. Instead, it is translated to applications of recursors via auxiliary matching functions. Both require a *motive* that explains the relationship between the discriminant and the resulting type. Generally, the **`match`** elaborator is capable of synthesizing an appropriate motive, and the refinement of types that occurs during pattern matching is a result of the motive that was selected. In some specialized circumstances, a different motive may be needed and may be provided explicitly using the `(motive := …)` syntax of **`match`**. This motive should be a function type that expects at least as many parameters as there are discriminants. The type that results from applying a function with this type to the discriminants in order is the type of the entire **`match`** term, and the type that results from applying a function with this type to all patterns in each alternative is the type of that alternative's right-hand side.

▼ *Example: Matching with an Explicit Motive*

An explicit motive can be used to provide type information that is otherwise unavailable from the surrounding context. Attempting to match on a number and a proof that it is in fact 5 is an error, because there's no reason to connect the number to the proof:

```
#eval
  match 5, rfl with
```

```
  | 5, rfl => "ok"
```

```
Invalid match expression: This pattern contains
metavariables:
  Eq.refl ?m.76
```

An explicit motive explains the relationship between the discriminants:

```
#eval
  match (motive := (n : Nat) → n = 5 → String) 5, rfl with
  | 5, rfl => "ok"
```

```
"ok"
```

10.8.1.3. Discriminant Refinement

When matching on an indexed family, the indices must also be discriminants. Otherwise, the pattern would not be well typed: it is a type error if an index is just a variable but the type of a constructor requires a more specific value. However, a process called discriminant refinement automatically adds indices as additional discriminants.

▼ *Example: Discriminant Refinement*

In the definition of `f`, the equality proof is the only discriminant. However, equality is an indexed family, and the match is only valid when n is an additional discriminant.

```
def f (n : Nat) (p : n = 3) : String :=
  match p with
  | rfl => "ok"
```

Using `#print` demonstrates that the additional discriminant was added automatically.

```
#print f
```

```
def f : (n : Nat) → n = 3 → String :=
fun n p =>
  match 3, p with
  | .(n), ⋯ => "ok"
```

10.8.1.4. Generalization

The pattern match elaborator automatically determines the motive by finding occurrences of the discriminants in the expected type, generalizing them in the types of subsequent discriminants so that the appropriate pattern can be substituted. Additionally, occurrences of the discriminants in the types of variables in the context are generalized and substituted by default. This latter behavior can be turned off by passing the `(generalizing := false)` flag to `match`.

▼ *Example: Matching, With and Without Generalization*

In this definition of `boolCases`, the assumption *b* is generalized in the type of h and then replaced with the actual pattern. This means that *ifTrue* and *ifFalse* have the types `true = true → α` and `false = false → α` in their respective cases, but h's type mentions the original discriminant.

```
def boolCases (b : Bool)
    (ifTrue : b = true → α)
    (ifFalse : b = false → α) :
    α :=
  match h : b with
  | true  => ifTrue h
  | false => ifFalse h
```

The error for the first case is typical of both:

```
Application type mismatch: In the application
  ifTrue h
the argument
  h
has type
  b = true : Prop
but is expected to have type
  true = true : Prop
```

Turning off generalization allows type checking to succeed, because *b* remains in the types of *ifTrue* and *ifFalse*.

```
def boolCases (b : Bool)
    (ifTrue : b = true → α)
    (ifFalse : b = false → α) :
    α :=
  match (generalizing := false) h : b with
  | true  => ifTrue h
  | false => ifFalse h
```

In the generalized version, `rfl` could have been used as the proof arguments as an alternative.

10.8.2. Custom Pattern Functions

In patterns, defined constants with the **`match_pattern`** attribute are unfolded and normalized rather than rejected. This allows a more convenient syntax to be used for many patterns. In the standard library, `Nat.add`, `HAdd.hAdd`, `Add.add`, and `Neg.neg` all have this attribute, which allows patterns like `n + 1` instead of `Nat.succ n`. Similarly, `Unit` and `Unit.unit` are definitions that set the respective universe parameters of `PUnit` and `PUnit.unit` to 0; the **`match_pattern`** attribute on `Unit.unit` allows it to be used in patterns, where it expands to `PUnit.unit.{0}`.

attribute

Attribute for Match Patterns

The **match_pattern** attribute indicates that a definition should be unfolded, rather than rejected, in a pattern.

```
attr ::= ...
    | match_pattern
```

▼ *Example: Match Patterns Follow Reduction*

The following function can't be compiled:

```
def nonzero (n : Nat) : Bool :=
  match n with
  | 0 => false
  | 1 + k => true
```

The error message on the pattern `1 + _` is:

```
Invalid pattern(s): `k` is an explicit pattern variable, but
it only occurs in positions that are inaccessible to pattern
matching:
  .(Nat.add 1 k)
```

This is because `Nat.add` is defined by recursion on its second parameter, equivalently to:

```
def add : Nat → Nat → Nat
  | a, Nat.zero   => a
  | a, Nat.succ b => Nat.succ (Nat.add a b)
```

No ι-reduction is possible, because the value being matched is a variable, not a constructor. `1 + k` gets stuck as `Nat.add 1 k`, which is not a valid pattern.

In the case of `k + 1`, that is, `Nat.add k (.succ .zero)`, the second pattern matches, so it reduces to `Nat.succ (Nat.add k .zero)`. The second pattern now matches, yielding `Nat.succ k`, which is a valid pattern.

10.8.3. Pattern Matching Functions

syntax

Pattern-Matching Functions

Functions may be specified via pattern matching by writing a sequence of patterns after `fun`, each preceded by a vertical bar (`|`).

```
term ::= ...
    | fun
        (| term,* => term)*
```

This desugars to a function that immediately pattern-matches on its arguments.

▼ *Example: Pattern-Matching Functions*

`isZero` is defined using a pattern-matching function abstraction, while `isZero'` is defined using a pattern match expression:

```
def isZero : Nat → Bool :=
  fun
    | 0 => true
    | _ => false

def isZero' : Nat → Bool :=
  fun n =>
    match n with
    | 0 => true
    | _ => false
```

Because the former is syntactic sugar for the latter, they are definitionally equal:

```
example : isZero = isZero' := rfl
```

The desugaring is visible in the output of `#print`:

```
#print isZero
```

outputs

```
def isZero : Nat → Bool :=
fun x =>
  match x with
  | 0 => true
  | x => false
```

while

```
#print isZero'
```

outputs

```
def isZero' : Nat → Bool :=
fun n =>
  match n with
  | 0 => true
  | x => false
```

10.8.4. Other Pattern Matching Operators

In addition to `match` and `if let`, there are a few other operators that perform pattern matching.

syntax

The `matches` Operator

The `matches` operator returns `true` if the term on the left matches the pattern on the right.

```
term ::= ...
    | term matches term
```

When branching on the result of `matches`, it's usually better to use `if let`, which can bind pattern variables in addition to checking whether a pattern matches.

If there are no constructor patterns that could match a discriminant or sequence of discriminants, then the code in question is unreachable, as there must be a false assumption in the local context. The `nomatch` expression is a match with zero cases that can have any type whatsoever, so long as there are no possible cases that could match the discriminants.

syntax

Caseless Pattern Matches

```
term ::= ...
    | nomatch term,*
```

▼ *Example: Inconsistent Indices*

There are no constructor patterns that can match both proofs in this example:

```
example (p1 : x = "Hello") (p2 : x = "world") : False :=
  nomatch p1, p2
```

This is because they separately refine the value of x to unequal strings. Thus, the `nomatch` operator allows the example's body to prove `False` (or any other proposition or type).

When the expected type is a function type, `nofun` is shorthand for a function that takes as many parameters as the type indicates in which the body is `nomatch` applied to all of the parameters.

syntax

Caseless Functions

```
term ::= ...
```

```
    | nofun
```

▼ *Example: Impossible Functions*

Instead of introducing arguments for both equality proofs and then using both in a `nomatch`, it is possible to use `nofun`.

```
example : x = "Hello" → x = "world" → False := nofun
```

10.9. Holes

A *hole* or *placeholder term* is a term that indicates the absence of instructions to the elaborator. In terms, holes can be automatically filled when the surrounding context would only allow one type-correct term to be written where the hole is. Otherwise, a hole is an error. In patterns, holes represent universal patterns that can match anything.

syntax

Holes

Holes are written with underscores.

```
term ::= ...
    | _
```

▼ *Example: Filling Holes with Unification*

The function `the` can be used similarly to `show` or a type ascription.

```
def the (α : Sort u) (x : α) : α := x
```

If the second parameter's type can be inferred, then the first parameter can be a hole. Both of these commands are equivalent:

```
#check the String "Hello!"
```

```
#check the _ "Hello"
```

When writing proofs, it can be convenient to explicitly introduce unknown values. This is done via *synthetic holes*, which are never solved by unification and may occur in multiple positions. They are primarily useful in tactic proofs, and are described in the section on metavariables in proofs.

syntax

Synthetic Holes

```
term ::= ...
    | ?ident

term ::= ...
    | ?_
```

10.10. Type Ascription

Type ascriptions explicitly annotate terms with their types. They are a way to provide Lean with the expected type for a term. This type must be definitionally equal to the type that is expected based on the term's context. Type ascriptions are useful for more than just documenting a program:

- There may not be sufficient information in the program text to derive a type for a term. Ascriptions are one way to provide the type.
- An inferred type may not be the one that was desired for a term.
- The expected type of a term is used to drive the insertion of coercions, and ascriptions are one way to control where coercions are inserted.

syntax

Postfix Type Ascriptions

Type ascriptions must be surrounded by parentheses. They indicate that the first term's type is the second term.

```
term ::= ...
    | (term : term)
```

In cases where the term that requires a type ascription is long, such as a tactic proof or a do block, the postfix type ascription with its mandatory parentheses can be difficult to read. Additionally, for both proofs and do blocks, the term's type is essential to its interpretation. In these cases, the prefix versions can be easier to read.

syntax

Prefix Type Ascriptions

```
term ::= ...
    | show term from term
```

When the term in the body of `show` is a tactic proof, the keyword `from` may be omitted.

```
term ::= ...
    | show term by tacticSeq
```

▼ *Example: Ascribing Statements to Proofs*

This example is unable to execute the tactic proof because the desired proposition is not known. As part of running the earlier tactics, the proposition is automatically refined to be one that the tactics could prove. However, their default cases fill it out incorrectly, leading to a proof that fails.

```
example (n : Nat) := by
  induction n
  next => rfl
  next n' ih =>
    simp only [HAdd.hAdd, Add.add, Nat.add] at *
    rewrite [ih]
    rfl
```

```
tactic 'rewrite' failed, equality or iff proof expected
  0 ≺ n'
n' : Nat
ih : 0 ≺ n'
⊢ 0 ≺ n'.succ
```

A prefix type ascription with `show` can be used to provide the proposition being proved. This can be useful in syntactic contexts where adding it as a local definition would be inconvenient.

```
example (n : Nat) := show 0 + n = n by
  induction n
  next => rfl
  next n' ih =>
    simp only [HAdd.hAdd, Add.add, Nat.add] at *
    rewrite [ih]
    rfl
```

▼ *Example: Ascribing Types to `do` Blocks*

This example lacks sufficient type information to synthesize the `Pure` instance.

```
example := do
  return 5
```

```
typeclass instance problem is stuck, it is often due to
metavariables
  Pure ?m.53
```

A prefix type ascription with `show`, together with a hole, can be used to indicate the monad. The default `OfNat _ 5` instance provides enough type information to fill the hole with `Nat`.

```
example := show StateM String _ from do
  return 5
```

10.11. Quotation and Antiquotation

Quotation terms are described in the section on quotation.

10.12. do-Notation

`do`-notation is described in the chapter on monads.

10.13. Proofs

The syntax for invoking tactics (`by`) is described in the section on proofs.

11. Type Classes

An operation is *polymorphic* if it can be used with multiple types. In Lean, polymorphism comes in three varieties:

1. universe polymorphism, where the sorts in a definition can be instantiated in various ways,
2. functions that take types as (potentially implicit) parameters, allowing a single body of code to work with any type, and
3. *ad-hoc polymorphism*, implemented with type classes, in which operations to be overloaded may have different implementations for different types.

Because Lean does not allow case analysis of types, polymorphic functions implement operations that are uniform for any choice of type argument; for example, `List.map` does not suddenly compute differently depending on whether the input list contains `String`s or `Nat`s. Ad-hoc polymorphic operations are useful when there is no "uniform" way to implement an operation; the canonical use case is for overloading arithmetic operators so that they work with `Nat`, `Int`, `Float`, and any other type that has a sensible notion of addition. Ad-hoc polymorphism may also involve multiple types; looking up a value at a given index in a collection involves the collection type, the index type, and the type of member elements to be extracted. A *type class* (i) Type classes were first described in Philip Wadler and Stephen Blott, 1980. "How to make ad-hoc polymorphism less ad hoc". In *Proceedings of the 16th Symposium on Principles of Programming Languages.* describes a collection of overloaded operations (called *methods*) together with the involved types.

Type classes are very flexible. Overloading may involve multiple types; operations like indexing into a data structure can be overloaded for a specific choice of data structure, index type, element type, and even a predicate that asserts the presence of the key in the structure. Due to Lean's expressive type system, overloading operations is not restricted only to types; type classes may be parameterized by ordinary values, by families of types, and even by predicates or propositions. All of these possibilities are used in practice:

Natural number literals

The `OfNat` type class is used to interpret natural number literals. Instances may depend not only on the type being instantiated, but also on the number literal itself.

Computational effects

Type classes such as `Monad`, whose parameter is a function from one type to another, are used to provide special syntax for programs with side effects. The “type” for which operations are overloaded is actually a type-level function, such as `Option`, `IO`, or `Except`.

Predicates and propositions

The `Decidable` type class allows a decision procedure for a proposition to be found automatically by Lean. This is used as the basis for `if`-expressions, which may branch on any decidable proposition.

While ordinary polymorphic definitions simply expect instantiation with arbitrary parameters, the operators overloaded with type classes are to be instantiated with *instances* that define the overloaded operation for some specific set of parameters. These instance-implicit parameters are indicated in square brackets. At invocation sites, Lean either *synthesizes* a suitable instance from the available candidates or signals an error. Because instances may themselves have instance parameters, this search process may be recursive and result in a final composite instance value that combines code from a variety of instances. Thus, type class instance synthesis is also a means of constructing programs in a type-directed manner.

Here are some typical use cases for type classes:

- Type classes may represent overloaded operators, such as arithmetic that can be used with a variety of types of numbers or a membership predicate that can be used for a variety of data structures. There is often a single canonical choice of operator for a given type—after all, there is no sensible alternative definition of addition for `Nat`—but this is not an essential property, and libraries may provide alternative instances if needed.
- Type classes can represent an algebraic structure, providing both the extra structure and the axioms required by the structure. For example, a type class that represents an Abelian group may contain methods for a binary operator, a unary inverse operator, an identity element, as well as proofs that the binary operator is associative and commutative, that the identity is an identity, and that the inverse operator yields the identity element on both sides of the operator. Here, there may not be a canonical choice of structure, and a library may provide many ways to instantiate a given set of axioms; there are two equally canonical monoid structures over the integers.

- A type class can represent a relation between two types that allows them to be used together in some novel way by a library. The `Coe` class represents automatically-inserted coercions from one type to another, and `MonadLift` represents a way to run operations with one kind of effect in a context that expects another kind.
- Type classes can represent a framework of type-driven code generation, where instances for polymorphic types each contribute some portion of a final program. The `Repr` class defines a canonical pretty printer for a type, and polymorphic types end up with polymorphic `Repr` instances. When pretty printing is finally invoked on an expression with a known concrete type, such as `List (Nat × (String ⊕ Int))`, the resulting pretty printer contains code assembled from the `Repr` instances for `List`, `Prod`, `Nat`, `Sum`, `String`, and `Int`.

11.1. Class Declarations

Type classes are declared with the `class` keyword.

syntax

Type Class Declarations

```
command ::= ...
    | declModifiers
      class declId bracketedBinder* (: term)?
        (extends (ident : )?term,*)?
        where
        (declModifiers ident ::)?
        structFields
      (deriving ident,*)?
```

Declares a new type class.

The `class` declaration creates a new single-constructor inductive type, just as if the `structure` command had been used instead. In fact, the results of the `class` and `structure` commands are almost identical, and features such as default values may be used the same way in both. Please refer to the documentation for structures for more information about default values, inheritance, and other features of structures. The differences between structure and class declarations are:

Methods instead of fields

Instead of creating field projections that take a value of the structure type as an explicit parameter, methods are created. Each method takes the corresponding instance as an instance-implicit parameter.

Instance-implicit parent classes

The constructor of a class that extends other classes takes its class parents' instances as instance-implicit parameters, rather than explicit parameters. When instances of this class are defined, instance synthesis is used to find the values of inherited fields. Parents that are not classes are still explicit parameters to the underlying constructor.

Parent projections via instance synthesis

Structure field projections make use of inheritance information to project parent structure fields from child structure values. Classes instead use instance synthesis: given a child class instance, synthesis will construct the parent; thus, methods are not added to child classes in the same way that projections are added to child structures.

Registered as class

The resulting inductive type is registered as a type class, for which instances may be defined and that may be used as the type of instance-implicit arguments.

Out and semi-out parameters are considered

The `outParam` and `semiOutParam` gadgets have no meaning in structure definitions, but they are used in class definitions to control instance search.

While `deriving` clauses are allowed for class definitions to maintain the parallel between class and structure elaboration, they are not frequently used and should be considered an advanced feature.

▼ *Example: No Instances of Non-Classes*

Lean rejects instance-implicit parameters of types that are not classes:

```
def f [n : Nat] : n = n := rfl
```

```
invalid binder annotation, type is not a class instance
  Nat

Note: Use the command `set_option checkBinderAnnotations
false` to disable the check
```

▼ *Example: Class vs Structure Constructors*

A very small algebraic hierarchy can be represented either as structures (`S.Magma`, `S.Semigroup`, and `S.Monoid` below), a mix of structures and classes (`C1.Monoid`), or only using classes (`C2.Magma`, `C2.Semigroup`, and `C2.Monoid`):

```
namespace S
structure Magma (α : Type u) where
  op : α → α → α

structure Semigroup (α : Type u) extends Magma α where
  op_assoc : ∀ x y z, op (op x y) z = op x (op y z)

structure Monoid (α : Type u) extends Semigroup α where
  ident : α
  ident_left : ∀ x, op ident x = x
  ident_right : ∀ x, op x ident = x
end S
```

```
namespace C1
class Monoid (α : Type u) extends S.Semigroup α where
  ident : α
  ident_left : ∀ x, op ident x = x
  ident_right : ∀ x, op x ident = x
end C1

namespace C2
class Magma (α : Type u) where
  op : α → α → α

class Semigroup (α : Type u) extends Magma α where
  op_assoc : ∀ x y z, op (op x y) z = op x (op y z)

class Monoid (α : Type u) extends Semigroup α where
  ident : α
  ident_left : ∀ x, op ident x = x
  ident_right : ∀ x, op x ident = x
end C2
```

`S.Monoid.mk` and `C1.Monoid.mk` have identical signatures, because the parent of the class `C1.Monoid` is not itself a class:

```
S.Monoid.mk.{u} {α : Type u}
  (toSemigroup : S.Semigroup α)
  (ident : α)
  (ident_left : ∀ (x : α), toSemigroup.op ident x = x)
  (ident_right : ∀ (x : α), toSemigroup.op x ident = x) :
  S.Monoid α

C1.Monoid.mk.{u} {α : Type u}
  (toSemigroup : S.Semigroup α)
  (ident : α)
  (ident_left : ∀ (x : α), toSemigroup.op ident x = x)
  (ident_right : ∀ (x : α), toSemigroup.op x ident = x) :
  C1.Monoid α
```

Similarly, because neither `S.Magma` nor `C2.Magma` inherits from another structure or class, their constructors are identical:

```
S.Magma.mk.{u} {α : Type u} (op : α → α → α) : S.Magma α

C2.Magma.mk.{u} {α : Type u} (op : α → α → α) : C2.Magma α
```

`S.Semigroup.mk`, however, takes its parent as an ordinary parameter, while `C2.Semigroup.mk` takes its parent as an instance implicit parameter:

```
S.Semigroup.mk.{u} {α : Type u}
  (toMagma : S.Magma α)
```

```
  (op_assoc : ∀ (x y z : α),
    toMagma.op (toMagma.op x y) z = toMagma.op x (toMagma.op y
z)) :
  S.Semigroup α

C2.Semigroup.mk.{u} {α : Type u} [toMagma : C2.Magma α]
  (op_assoc : ∀ (x y z : α),
    toMagma.op (toMagma.op x y) z = toMagma.op x (toMagma.op y
z)) :
  C2.Semigroup α
```

Finally, `C2.Monoid.mk` takes its semigroup parent as an instance implicit parameter. The references to op become references to the method `C2.Magma.op`, relying on instance synthesis to recover the implementation from the `C2.Semigroup` instance-implicit parameter via its parent projection:

```
C2.Monoid.mk.{u} {α : Type u}
  [toSemigroup : C2.Semigroup α]
  (ident : α)
  (ident_left : ∀ (x : α), C2.Magma.op ident x = x)
  (ident_right : ∀ (x : α), C2.Magma.op x ident = x) :
  C2.Monoid α
```

Parameters to type classes may be marked with *gadgets*, which are special versions of the identity function that cause the elaborator to treat a value differently. Gadgets never change the *meaning* of a term, but they may cause it to be treated differently in elaboration-time search procedures. The gadgets `outParam` and `semiOutParam` affect instance synthesis, so they are documented in that section.

Whether a type is a class or not has no effect on definitional equality. Two instances of the same class with the same parameters are not necessarily identical and may in fact be very different.

▼ *Example: Instances are Not Unique*

This implementation of binary heap insertion is buggy:

```
structure Heap (α : Type u) where
  contents : Array α
deriving Repr

def Heap.bubbleUp [Ord α] (i : Nat) (xs : Heap α) : Heap α :=
  if h : i = 0 then xs
  else if h : i ≥ xs.contents.size then xs
  else
    let j := i / 2
    if Ord.compare xs.contents[i] xs.contents[j] == .lt then
      Heap.bubbleUp j { xs with contents := xs.contents.swap i j
```

```
}
    else xs

def Heap.insert [Ord α] (x : α) (xs : Heap α) : Heap α :=
  let i := xs.contents.size
  {xs with contents := xs.contents.push x}.bubbleUp i
```

The problem is that a heap constructed with one `Ord` instance may later be used with another, leading to the breaking of the heap invariant.

One way to correct this is to making the heap type depend on the selected `Ord` instance:

```
structure Heap' (α : Type u) [Ord α] where
  contents : Array α

def Heap'.bubbleUp [inst : Ord α]
    (i : Nat) (xs : @Heap' α inst) :
    @Heap' α inst :=
  if h : i = 0 then xs
  else if h : i ≥ xs.contents.size then xs
  else
    let j := i / 2
    if inst.compare xs.contents[i] xs.contents[j] == .lt then
      Heap'.bubbleUp j {xs with contents := xs.contents.swap i j}
    else xs

def Heap'.insert [Ord α] (x : α) (xs : Heap' α) : Heap' α :=
  let i := xs.contents.size
  {xs with contents := xs.contents.push x}.bubbleUp i
```

In the improved definitions, `Heap'.bubbleUp` is needlessly explicit; the instance does not need to be explicitly named here because Lean would select the indicated instances nonetheless, but it does bring the correctness invariant front and center for readers.

11.1.1. Sum Types as Classes

Most type classes follow the paradigm of a set of overloaded methods from which clients may choose freely. This is naturally modeled by a product type, from which the overloaded methods are projections. Some classes, however, are sum types: they require that the recipient of the synthesized instance first check *which* of the available instance constructors was provided. To account for these classes, a class declaration may consist of an arbitrary inductive type, not just an extended form of structure declaration.

syntax

Class Inductive Type Declarations

```
command ::= ...
    | declModifiers
```

```
class inductive declId optDeclSig where
  (| declModifiers ident optDeclSig)*
(deriving ident,*)?
```

Class inductive types are just like other inductive types, except they may participate in instance synthesis. The paradigmatic example of a class inductive is `Decidable`: synthesizing an instance in a context with free variables amounts to synthesizing the decision procedure, but if there are no free variables, then the truth of the proposition can be established by instance synthesis alone (as is done by the **`decide`** tactic).

11.1.2. Class Abbreviations

In some cases, many related type classes may co-occur throughout a codebase. Rather than writing all the names repeatedly, it would be possible to define a class that extends all the classes in question, contributing no new methods itself. However, this new class has a disadvantage: its instances must be declared explicitly.

The **`class abbrev`** command allows the creation of *class abbreviations* in which one name is short for a number of other class parameters. Behind the scenes, a class abbreviation is represented by a class that extends all the others. Its constructor is additionally declared to be an instance so the new class can be constructed by instance synthesis alone.

▼ *Example: Class Abbreviations*

Both `plusTimes1` and `plusTimes2` require that their parameters' type have `Add` and `Mul` instances:

```
class abbrev AddMul (α : Type u) := Add α, Mul α

def plusTimes1 [AddMul α] (x y z : α) := x + y * z

class AddMul' (α : Type u) extends Add α, Mul α

def plusTimes2 [AddMul' α] (x y z : α) := x + y * z
```

Because `AddMul` is a **`class abbrev`**, no additional declarations are necessary to use `plusTimes1` with `Nat`:

```
#eval plusTimes1 2 5 7
```

```
37
```

However, `plusTimes2` fails, because there is no `AddMul' Nat` instance—no instances whatsoever have yet been declared:

```
#eval plusTimes2 2 5 7
```

```
failed to synthesize
  AddMul' ?m.24

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Declaring an very general instance takes care of the problem for `Nat` and every other type:

```
instance [Add α] [Mul α] : AddMul' α where

#eval plusTimes2 2 5 7
```

```
37
```

11.2. Instance Declarations

The syntax of instance declarations is almost identical to that of definitions. The only syntactic differences are that the keyword `def` is replaced by `instance` and the name is optional:

syntax

Instance Declarations

Most instances define each method using `where` syntax:

```
instance ::= ...
    | instance ((priority := prio))? declId? declSig where
        structInstField*
```

However, type classes are inductive types, so instances can be constructed using any expression with an appropriate type:

```
instance ::= ...
    | instance ((priority := prio))? declId? declSig :=
        term
```

Instances may also be defined by cases; however, this feature is rarely used outside of `Decidable` instances:

```
instance ::= ...
    | instance ((priority := prio))? declId? declSig
        (| term => term)*
```

Instances defined with explicit terms often consist of either anonymous constructors (`⟨...⟩`) wrapping method implementations or of invocations of `inferInstanceAs` on definitionally equal types.

Elaboration of instances is almost identical to the elaboration of ordinary definitions, with the exception of the caveats documented below. If no name is provided, then one is created automatically. It is possible to refer to this generated name directly, but the algorithm used to generate the names has changed in the past and may change in the future. It's better to explicitly name instances that will be referred to directly. After elaboration, the new instance is registered as a candidate for instance search. Adding the attribute **`instance`** to a name can be used to mark any other defined name as a candidate.

▼ *Example: Instance Name Generation*

Following these declarations:

```
structure NatWrapper where
  val : Nat

instance : BEq NatWrapper where
  beq
    | ⟨x⟩, ⟨y⟩ => x == y
```

the name `instBEqNatWrapper` refers to the new instance.

▼ *Example: Variations in Instance Definitions*

Given this structure type:

```
structure NatWrapper where
  val : Nat
```

all of the following ways of defining a BEq instance are equivalent:

```
instance : BEq NatWrapper where
  beq
    | ⟨x⟩, ⟨y⟩ => x == y

instance : BEq NatWrapper :=
  ⟨fun x y => x.val == y.val⟩

instance : BEq NatWrapper :=
  ⟨fun ⟨x⟩ ⟨y⟩ => x == y⟩
```

Aside from introducing different names into the environment, the following are also equivalent:

```
@[instance]
def instBeqNatWrapper : BEq NatWrapper where
  beq
    | ⟨x⟩, ⟨y⟩ => x == y

instance : BEq NatWrapper :=
```

```
  ⟨fun x y => x.val == y.val⟩

instance : BEq NatWrapper :=
  ⟨fun ⟨x⟩ ⟨y⟩ => x == y⟩
```

11.2.1. Recursive Instances

Functions defined in `where` structure definition syntax are not recursive. Because instance declaration is a version of structure definition, type class methods are also not recursive by default. Instances for recursive inductive types are common, however. There is a standard idiom to work around this limitation: define a recursive function independently of the instance, and then refer to it in the instance definition. By convention, these recursive functions have the name of the corresponding method, but are defined in the type's namespace.

▼ *Example: Instances are not recursive*

Given this definition of NatTree:

```
inductive NatTree where
  | leaf
  | branch (left : NatTree) (val : Nat) (right : NatTree)
```

the following BEq instance fails:

```
instance : BEq NatTree where
  beq
    | .leaf, .leaf => true
    | .branch l1 v1 r1, .branch l2 v2 r2 => l1 == l2 && v1 == v2
&& r1 == r2
    | _, _ => false
```

with errors in both the left and right recursive calls that read:

```
failed to synthesize
  BEq NatTree

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Given a suitable recursive function, such as NatTree.beq:

```
def NatTree.beq : NatTree → NatTree → Bool
  | .leaf, .leaf => true
  | .branch l1 v1 r1, .branch l2 v2 r2 => l1 == l2 && v1 == v2 &&
r1 == r2
  | _, _ => false
```

the instance can be created in a second step:

```
instance : BEq NatTree where
  beq := NatTree.beq
```

or, equivalently, using anonymous constructor syntax:

```
instance : BEq NatTree := ⟨NatTree.beq⟩
```

Furthermore, instances are not available for instance synthesis during their own definitions. They are first marked as being available for instance synthesis after they are defined. Nested inductive types, in which the recursive occurrence of the type occurs as a parameter to some other inductive type, may require an instance to be available to write even the recursive function. The standard idiom to work around this limitation is to create a local instance in a recursively-defined function that includes a reference to the function being defined, taking advantage of the fact that instance synthesis may use every binding in the local context with the right type.

▼ *Example: Instances for nested types*

In this definition of `NatRoseTree`, the type being defined occurs nested under another inductive type constructor (`Array`):

```
inductive NatRoseTree where
  | node (val : Nat) (children : Array NatRoseTree)
```

Checking the equality of rose trees requires checking equality of arrays. However, instances are not typically available for instance synthesis during their own definitions, so the following definition fails, even though `NatRoseTree.beq` is a recursive function and is in scope in its own definition.

```
def NatRoseTree.beq : (tree1 tree2 : NatRoseTree) → Bool
  | .node val1 children1, .node val2 children2 =>
    val1 == val2 &&
    children1 == children2
```

```
failed to synthesize
  BEq (Array NatRoseTree)

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

To solve this, a local `BEq NatRoseTree` instance may be `let`-bound:

```
partial def NatRoseTree.beq : (tree1 tree2 : NatRoseTree) → Bool
  | .node val1 children1, .node val2 children2 =>
    let _ : BEq NatRoseTree := ⟨NatRoseTree.beq⟩
    val1 == val2 &&
    children1 == children2
```

The use of array equality on the children finds the let-bound instance during instance synthesis.

11.2.2. Instances of `class inductive`s

Many instances have function types: any instance that itself recursively invokes instance search is a function, as is any instance with implicit parameters. While most instances only project method implementations from their own instance parameters, instances of class inductive types typically pattern-match one or more of their arguments, allowing the instance to select the appropriate constructor. This is done using ordinary Lean function syntax. Just as with other instances, the function in question is not available for instance synthesis in its own definition.

▼ *Example: An instance for a sum class*

Because `DecidableEq α` is an abbreviation for `(a b : α) → Decidable (Eq a b)`, its arguments can be used directly, as in this example:

```
inductive ThreeChoices where
  | yes | no | maybe

instance : DecidableEq ThreeChoices
  | .yes,   .yes   =>
    .isTrue rfl
  | .no,    .no    =>
    .isTrue rfl
  | .maybe, .maybe =>
    .isTrue rfl
  | .yes,   .maybe | .yes,   .no
  | .maybe, .yes   | .maybe, .no
  | .no,    .yes   | .no,    .maybe =>
    .isFalse nofun
```

▼ *Example: A recursive instance for a sum class*

The type `StringList` represents monomorphic lists of strings:

```
inductive StringList where
  | nil
  | cons (hd : String) (tl : StringList)
```

In the following attempt at defining a `DecidableEq` instance, instance synthesis invoked while elaborating the inner `if` fails because the instance is not available for instance synthesis in its own definition:

```
instance : DecidableEq StringList
  | .nil, .nil => .isTrue rfl
```

```
  | .cons h1 t1, .cons h2 t2 =>
    if h : h1 = h2 then
      if h' : t1 = t2 then
        .isTrue (by simp [*])
      else
        .isFalse (by intro hEq; cases hEq; trivial)
    else
      .isFalse (by intro hEq; cases hEq; trivial)
  | .nil, .cons _ _ | .cons _ _, .nil => .isFalse nofun
```

```
failed to synthesize
  Decidable (t1 = t2)

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

However, because it is an ordinary Lean function, it can recursively refer to its own explicitly-provided name:

```
instance instDecidableEqStringList : DecidableEq StringList
  | .nil, .nil => .isTrue rfl
  | .cons h1 t1, .cons h2 t2 =>
    let _ : Decidable (t1 = t2) :=
      instDecidableEqStringList t1 t2
    if h : h1 = h2 then
      if h' : t1 = t2 then
        .isTrue (by simp [*])
      else
        .isFalse (by intro hEq; cases hEq; trivial)
    else
      .isFalse (by intro hEq; cases hEq; trivial)
  | .nil, .cons _ _ | .cons _ _, .nil => .isFalse nofun
```

11.2.3. Instance Priorities

Instances may be assigned *priorities*. During instance synthesis, higher-priority instances are preferred; see the section on instance synthesis for details of instance synthesis.

syntax

Instance Priorities

Priorities may be numeric:

```
prio ::=
    num
```

If no priority is specified, the default priority that corresponds to 1000 is used:

```
prio ::= ...
    | default
```

Three named priorities are available when numeric values are too fine-grained, corresponding to 100, 500, and 10000 respectively. The `mid` priority is lower than `default`.

```
prio ::= ...
    | low

prio ::= ...
    | mid

prio ::= ...
    | high
```

Finally, priorities can be added and subtracted, so `default + 2` is a valid priority, corresponding to 1002:

```
prio ::= ...
    | (prio)

prio ::= ...
    | prio + prio

prio ::= ...
    | prio - prio
```

11.2.4. Default Instances

The **`default_instance`** attribute specifies that an instance should be used as a fallback in situations where there is not enough information to select it otherwise. If no priority is specified, then the default priority `default` is used.

attribute

The `default_instance` Attribute

```
attr ::= ...
    | default_instance prio?
```

▼ *Example: Default Instances*

A default instance of `OfNat Nat` is used to select `Nat` for natural number literals in the absence of other type information. It is declared in the Lean standard library with priority 100. Given this representation of even numbers, in which an even number is represented by half of it:

```
structure Even where
  half : Nat
```

the following instances allow numeric literals to be used for small `Even` values (a limit on the depth of type class instance search prevents them from being used for arbitrarily large literals):

```
instance ofNatEven0 : OfNat Even 0 where
  ofNat := ⟨0⟩

instance ofNatEvenPlusTwo [OfNat Even n] : OfNat Even (n + 2)
where
  ofNat := ⟨(OfNat.ofNat n : Even).half + 1⟩

#eval (0 : Even)
#eval (34 : Even)
#eval (254 : Even)
```

```
{ half := 0 }
```

```
{ half := 17 }
```

```
{ half := 127 }
```

Specifying them as default instances with a priority greater than or equal to 100 causes them to be used instead of `Nat`:

```
attribute [default_instance 100] ofNatEven0
attribute [default_instance 100] ofNatEvenPlusTwo

#eval 0
#eval 34
```

```
{ half := 0 }
```

```
{ half := 17 }
```

Non-even numerals still use the `OfNat Nat` instance:

```
#eval 5
```

```
5
```

11.2.5. The Instance Attribute

The **`instance`** attribute declares a name to be an instance, with the specified priority. Like other attributes, **`instance`** can be applied globally, locally, or only when the current namespace is opened. The **`instance`** declaration is a form of definition that automatically applies the **`instance`** attribute.

attribute

The `instance` Attribute

Declares the definition to which it is applied to be an instance. If no priority is provided, then the default priority `default` is used.

```
attr ::= ...
    | instance prio?
```

11.3. Instance Synthesis

Instance synthesis is a recursive search procedure that either finds an instance for a given type class or fails. In other words, given a type that is registered as a type class, instance synthesis attempts to construct a term with said type. It respects reducibility: semireducible or irreducible definitions are not unfolded, so instances for a definition are not automatically treated as instances for its unfolding unless it is reducible. There may be multiple possible instances for a given class; in this case, declared priorities and order of declaration are used as tiebreakers, in that order, with more recent instances taking precedence over earlier ones with the same priority.

This search procedure is efficient in the presence of diamonds and does not loop indefinitely when there are cycles. *Diamonds* occur when there is more than one route to a given goal, and *cycles* are situations when two instances each could be solved if the other were solved. Diamonds occur regularly in practice when encoding mathematical concepts using type classes, and Lean's coercion feature naturally leads to cycles, e.g. between finite sets and finite multisets.

Instance synthesis can be tested using the `#synth` command. Additionally, `inferInstance` and `inferInstanceAs` can be used to synthesize an instance in a position where the instance itself is needed.

def

```
inferInstance.{u} {α : Sort u} [i : α] : α
```

`inferInstance` synthesizes a value of any target type by typeclass inference. This function has the same type signature as the identity function, but the square brackets on the `[i : α]` argument means that it will attempt to construct this argument by typeclass inference. (This will fail if `α` is not a **`class`**.) Example:

```
#check (inferInstance : Inhabited Nat) -- Inhabited Nat

def foo : Inhabited (Nat × Nat) :=
  inferInstance

example : foo.default = (default, default) :=
  rfl
```

def

```
inferInstanceAs.{u} (α : Sort u) [i : α] : α
```

`inferInstanceAs` α synthesizes a value of any target type by typeclass inference. This is just like `inferInstance` except that α is given explicitly instead of being inferred from the target type. It is especially useful when the target type is some `α'` which is definitionally equal to α, but the instance we are looking for is only registered for α (because typeclass search does not unfold most definitions, but definitional equality does.) Example:

```
#check inferInstanceAs (Inhabited Nat) -- Inhabited Nat
```

11.3.1. Instance Search Summary

Generally speaking, instance synthesis is a recursive search procedure that may, in general, backtrack arbitrarily. Synthesis may *succeed* with an instance term, *fail* if no such term can be found, or get *stuck* if there is insufficient information. A detailed description of the instance synthesis algorithm is available in Selsam, Ullrich, and de Moura (2020) (i Daniel Selsam, Sebastian Ullrich, and Leonardo de Moura, 2020. "Tabled typeclass resolution". arXiv:2001.04301). An instance search problem is given by a type class applied to concrete arguments; these argument values may or may not be known. Instance search attempts every locally-bound variable whose type is a class, as well as each registered instance, in order of priority and definition. When candidate instances themselves have instance-implicit parameters, they impose further synthesis tasks.

A problem is only attempted when all of the input parameters to the type class are known. When a problem cannot yet be attempted, then that branch is stuck; progress in other subproblems may result in the problem becoming solvable. Output or semi-output parameters may be either known or unknown at the start of instance search. Output parameters are ignored when checking whether an instance matches the problem, while semi-output parameters are considered.

Every candidate solution for a given problem is saved in a table; this prevents infinite regress in case of cycles as well as exponential search overheads in the presence of diamonds (that is, multiple paths by which the same goal can be achieved). A branch of the search fails when any of the following occur:

- All potential instances have been attempted, and the search space is exhausted.
- The instance size limit specified by the option `synthInstance.maxSize` is reached.
- The synthesized value of an output parameter does not match the specified value in the search problem. Failed branches are not retried.

If search would otherwise fail or get stuck, the search process attempts to use matching default instances in order of priority. For default instances, the input parameters do not need to be fully known, and may be instantiated by the instances parameter values. Default instances may take instance-implicit parameters, which induce further recursive search.

Successful branches in which the problem is fully known (that is, in which there are no unsolved metavariables) are pruned, and further potentially-successful instances are not attempted, because no later instance could cause the previously-succeeding branch to fail.

11.3.2. Instance Search Problems

Instance search occurs during the elaboration of (potentially nullary) function applications. Some of the implicit parameters' values are forced by others; for instance, an implicit type parameter may be solved using the type of a later value argument that is explicitly provided. Implicit parameters may also be solved using information from the expected type at that point in the program. The search for instance implicit arguments may make use of the implicit argument values that have been found, and may additionally solve others.

Instance synthesis begins with the type of the instance-implicit parameter. This type must be the application of a type class to zero or more arguments; these argument values may be known or unknown when search begins. If an argument to a class is unknown, the search process will not instantiate it unless the corresponding parameter is marked as an output parameter, explicitly making it an output of the instance synthesis routine.

Search may succeed, fail, or get stuck; a stuck search may occur when an unknown argument value becoming known might enable progress to be made. Stuck searches may be re-invoked when the elaborator has discovered one of the previously-unknown implicit arguments. If this does not occur, stuck searches become failures.

11.3.3. Candidate Instances

Instance synthesis uses both local and global instances in its search. *Local instances* are those available in the local context; they may be either parameters to a function or locally defined with `let`. Local instances do not need to be indicated specially; any local variable whose type is a type class is a candidate for instance synthesis. *Global instances* are those available in the global environment; every global instance is a defined name with the **`instance`** attribute applied. (i) **`instance`** declarations automatically apply the **`instance`** attribute.

▼ *Example: Local Instances*

In this example, `addPairs` contains a locally-defined instance of `Add NatPair`:

```
structure NatPair where
  x : Nat
  y : Nat

def addPairs (p1 p2 : NatPair) : NatPair :=
  let _ : Add NatPair :=
    ⟨fun ⟨x1, y1⟩ ⟨x2, y2⟩ => ⟨x1 + x2, y1 + y2⟩⟩
  p1 + p2
```

The local instance is used for the addition, having been found by instance synthesis.

▼ *Example: Local Instances Have Priority*

Here, `addPairs` contains a locally-defined instance of `Add NatPair`, even though there is a global instance:

```
structure NatPair where
  x : Nat
  y : Nat

instance : Add NatPair where
  add
    | ⟨x1, y1⟩, ⟨x2, y2⟩ => ⟨x1 + x2, y1 + y2⟩

def addPairs (p1 p2 : NatPair) : NatPair :=
  let _ : Add NatPair :=
    ⟨fun _ _ => ⟨0, 0⟩⟩
  p1 + p2
```

The local instance is selected instead of the global one:

```
#eval addPairs ⟨1, 2⟩ ⟨5, 2⟩
```

```
{ x := 0, y := 0 }
```

11.3.4. Instance Parameters and Synthesis

The search process for instances is largely governed by class parameters. Type classes take a certain number of parameters, and instances are tried during the search when their choice of parameters is *compatible* with those in the class type for which the instance is being synthesized.

Instances themselves may also take parameters, but the role of instances' parameters in instance synthesis is very different. Instances' parameters represent either variables that may be instantiated by instance synthesis or further synthesis work to be done before the instance can be used. In particular, parameters to instances may be explicit, implicit, or instance-implicit. If they are instance implicit, then they induce further recursive instance searching, while explicit or implicit parameters must be solved by unification.

▼ *Example: Implicit and Explicit Parameters to Instances*

While instances typically take parameters either implicitly or instance-implicitly, explicit parameters may be filled out as if they were implicit during instance synthesis. In this example, `aNonemptySumInstance` is found by synthesis, applied explicitly to `Nat`, which is needed to make it type-correct.

```
instance aNonemptySumInstance (α : Type) {β : Type} [inst :
Nonempty α] : Nonempty (α ⊕ β) :=
  let ⟨x⟩ := inst
```

```
  ⟨.inl x⟩

set_option pp.explicit true in
#synth Nonempty (Nat ⊕ Empty)
```

In the output, both the explicit argument `Nat` and the implicit argument `Empty` were found by unification with the search goal, while the `Nonempty Nat` instance was found via recursive instance synthesis.

```
@aNonemptySumInstance Nat Empty (@instNonemptyOfInhabited Nat
instInhabitedNat)
```

11.3.5. Output Parameters

By default, the parameters of a type class are considered to be *inputs* to the search process. If the parameters are not known, then the search process gets stuck, because choosing an instance would require the parameters to have values that match those in the instance, which cannot be determined on the basis of incomplete information. In most cases, guessing instances would make instance synthesis unpredictable.

In some cases, however, the choice of one parameter should cause an automatic choice of another. For example, the overloaded membership predicate type class `Membership` treats the type of elements of a data structure as an output, so that the type of element can be determined by the type of data structure at a use site, instead of requiring that there be sufficient type annotations to determine *both* types prior to starting instance synthesis. An element of a `List Nat` can be concluded to be a `Nat` simply on the basis of its membership in the list.

Type class parameters can be declared as outputs by wrapping their types in the `outParam` gadget. When a class parameter is an *output parameter*, instance synthesis will not require that it be known; in fact, any existing value is ignored completely. The first instance that matches the input parameters is selected, and that instance's assignment of the output parameter becomes its value. If there was a pre-existing value, then it is compared with the assignment after synthesis is complete, and it is an error if they do not match.

def

```
outParam.{u} (α : Sort u) : Sort u
```

Gadget for marking output parameters in type classes.

For example, the `Membership` class is defined as:

```
class Membership (α : outParam (Type u)) (γ : Type v)
```

This means that whenever a typeclass goal of the form `Membership ?α ?γ` comes up, Lean will wait to solve it until `?γ` is known, but then it will run typeclass inference, and take the first solution it finds, for any value of `?α`, which thereby determines what `?α` should be.

This expresses that in a term like `a ∈ s`, `s` might be a `Set α` or `List α` or some other type with a membership operation, and in each case the "member" type `α` is determined by looking at the container type.

▼ *Example: Output Parameters and Stuck Search*

This serialization framework provides a way to convert values to some underlying storage type:

```
class Serialize (input output : Type) where
  ser : input → output
export Serialize (ser)

instance : Serialize Nat String where
  ser n := toString n

instance [Serialize α γ] [Serialize β γ] [Append γ] :
    Serialize (α × β) γ where
  ser
    | (x, y) => ser x ++ ser y
```

In this example, the output type is unknown.

```
example := ser (2, 3)
```

Instance synthesis can't select the `Serialize Nat String` instance, and thus the `Append String` instance, because that would require instantiating the output type as `String`, so the search gets stuck:

```
typeclass instance problem is stuck, it is often due to
metavariables
  Serialize (Nat × Nat) ?m.16
```

One way to fix the problem is to supply an expected type:

```
example : String := ser (2, 3)
```

The other is to make the output type into an output parameter:

```
class Serialize (input : Type) (output : outParam Type) where
  ser : input → output
export Serialize (ser)

instance : Serialize Nat String where
  ser n := toString n

instance [Serialize α γ] [Serialize β γ] [Append γ] :
    Serialize (α × β) γ where
  ser
    | (x, y) => ser x ++ ser y
```

Now, instance synthesis is free to select the `Serialize Nat String` instance, which solves the unknown implicit `output` parameter of `ser`:

```
example := ser (2, 3)
```

▼ *Example: Output Parameters with Pre-Existing Values*

The class `OneSmaller` represents a way to transform non-maximal elements of a type into elements of a type that has one fewer elements. There are two separate instances that can match an input type `Option Bool`, with different outputs:

```
class OneSmaller (α : Type) (β : outParam Type) where
  biggest : α
  shrink : (x : α) → x ≠ biggest → β

instance : OneSmaller (Option α) α where
  biggest := none
  shrink
    | some x, _ => x

instance : OneSmaller (Option Bool) (Option Unit) where
  biggest := some true
  shrink
    | none, _ => none
    | some false, _ => some ()

instance : OneSmaller Bool Unit where
  biggest := true
  shrink
    | false, _ => ()
```

Because instance synthesis selects the most recently defined instance, the following code is an error:

```
#check OneSmaller.shrink (β := Bool) (some false) sorry
```

```
failed to synthesize
  OneSmaller (Option Bool) Bool

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

The `OneSmaller (Option Bool) (Option Unit)` instance was selected during instance synthesis, without regard to the supplied value of β.

Semi-output parameters are like output parameters in that they are not required to be known prior to synthesis commencing; unlike output parameters, their values are taken into account when selecting instances.

def

`semiOutParam.{u} (α : Sort u) : Sort u`

Gadget for marking semi output parameters in type classes.

Semi-output parameters influence the order in which arguments to type class instances are processed. Lean determines an order where all non-(semi-)output parameters to the instance argument have to be figured out before attempting to synthesize an argument (that is, they do not contain assignable metavariables created during TC synthesis). This rules out instances such as `[Mul β] : Add α` (because β could be anything). Marking a parameter as semi-output is a promise that instances of the type class will always fill in a value for that parameter.

For example, the `Coe` class is defined as:

```
class Coe (α : semiOutParam (Sort u)) (β : Sort v)
```

This means that all `Coe` instances should provide a concrete value for α (i.e., not an assignable metavariable). An instance like `Coe Nat Int` or `Coe α (Option α)` is fine, but `Coe α Nat` is not since it does not provide a value for α.

Semi-output parameters impose a requirement on instances: each instance of a class with semi-output parameters should determine the values of its semi-output parameters.

▼ *Example: Semi-Output Parameters with Pre-Existing Values*

The class `OneSmaller` represents a way to transform non-maximal elements of a type into elements of a type that one fewer elements. It has two separate instances that can match an input type `Option Bool`, with different outputs:

```
class OneSmaller (α : Type) (β : semiOutParam Type) where
  biggest : α
  shrink : (x : α) → x ≠ biggest → β

instance : OneSmaller (Option α) α where
  biggest := none
  shrink
    | some x, _ => x

instance : OneSmaller (Option Bool) (Option Unit) where
  biggest := some true
  shrink
    | none, _ => none
    | some false, _ => some ()
```

```
instance : OneSmaller Bool Unit where
  biggest := true
  shrink
    | false, _ => ()
```

Because instance synthesis takes semi-output parameters into account when selecting instances, the `OneSmaller (Option Bool) (Option Unit)` instance is passed over due to the supplied value for β:

```
#check OneSmaller.shrink (β := Bool) (some false) sorry
```

```
OneSmaller.shrink (some false) ⋯ : Bool
```

11.3.6. Default Instances

When instance synthesis would otherwise fail, having not selected an instance, the *default instances* specified using the **`default_instance`** attribute are attempted in order of priority. When priorities are equal, more recently-defined default instances are chosen before earlier ones. The first default instance that causes the search to succeed is chosen.

Default instances may induce further recursive instance search if the default instances themselves have instance-implicit parameters. If the recursive search fails, the search process backtracks and the next default instance is tried.

11.3.7. “Morally Canonical” Instances

During instance synthesis, if a goal is fully known (that is, contains no metavariables) and search succeeds, no further instances will be attempted for that same goal. In other words, when search succeeds for a goal in a way that can't be refuted by a subsequent increase in information, the goal will not be attempted again, even if there are other instances that could potentially have been used. This optimization can prevent a failure in a later branch of an instance synthesis search from causing spurious backtracking that replaces a fast solution from an earlier branch with a slow exploration of a large state space.

The optimization relies on the assumption that instances are *morally canonical*. Even if there is more than one potential implementation of a given type class's overloaded operations, or more than one way to synthesize an instance due to diamonds, *any discovered instance should be considered as good as any other*. In other words, there's no need to consider *all* potential instances so long as one of them has been guaranteed to work. The optimization may be disabled with the backwards-compatibility option `backward.synthInstance.canonInstances`, which may be removed in a future version of Lean.

Code that uses instance-implicit parameters should be prepared to consider all instances as equivalent. In other words, it should be robust in the face of differences in synthesized instances. When the code relies on instances *in fact* being equivalent, it should either explicitly manipulate

instances (e.g. via local definitions, by saving them in structure fields, or having a structure inherit from the appropriate class) or it should make this dependency explicit in the type, so that different choices of instance lead to incompatible types.

11.3.8. Options

option

`backward.synthInstance.canonInstances`

Default value: `true`

use optimization that relies on 'morally canonical' instances during type class resolution

option

`synthInstance.maxHeartbeats`

Default value: `20000`

maximum amount of heartbeats per typeclass resolution problem. A heartbeat is number of (small) memory allocations (in thousands), 0 means no limit

option

`synthInstance.maxSize`

Default value: `128`

maximum number of instances used to construct a solution in the type class instance synthesis procedure

11.4. Deriving Instances

Lean can automatically generate instances for many classes, a process known as *deriving* instances. Instance deriving can be invoked either when defining a type or as a stand-alone command.

syntax

Instance Deriving (Optional)

As part of a command that creates a new inductive type, a **`deriving`** clause specifies a comma-separated list of class names for which instances should be generated:

```
optDeriving ::=
    (deriving ident,*)?
```

syntax

Stand-Alone Deriving of Instances

The stand-alone `deriving` command specifies a number of class names and subject names. Each of the specified classes are derived for each of the specified subjects.

```
command ::= ...
    | deriving instance ident,* for ident,*
```

▼ *Example: Deriving Multiple Classes*

After specifying multiple classes to derive for multiple types, as in this code:

```
structure A where
structure B where

deriving instance BEq, Repr for A, B
```

all the instances exist for all the types, so all four `#synth` commands succeed:

```
#synth BEq A
#synth BEq B
#synth Repr A
#synth Repr B
```

11.4.1. Deriving Handlers

Instance deriving uses a table of *deriving handlers* that maps type class names to metaprograms that derive instances for them. Deriving handlers may be added to the table using `registerDerivingHandler`, which should be called in an `initialize` block. Each deriving handler should have the type `Array Name → CommandElabM Bool`. When a user requests that an instance of a class be derived, its registered handlers are called one at a time. They are provided with all of the names in the mutual block for which the instance is to be derived, and should either correctly derive an instance and return `true` or have no effect and return `false`. When a handler returns `true`, no further handlers are called.

Lean includes deriving handlers for the following classes:

- `BEq`
- `DecidableEq`
- `Hashable`
- `Inhabited`
- `Nonempty`
- `Ord`

- Repr
- SizeOf
- TypeName

def

```
Lean.Elab.registerDerivingHandler (className : Name)
  (handler : DerivingHandler) : IO Unit
```

A `DerivingHandler` is called on the fully qualified names of all types it is running for as well as the syntax of a **`with`** argument, if present.

For example, **`deriving instance`** `Foo` **`with`** `fooArgs` **`for`** `Bar, Baz` invokes `fooHandler #[`Bar, `Baz] `(fooArgs)`.

▼ *Example: Deriving Handlers*

Instances of the IsEnum class demonstrate that a type is a finite enumeration by providing a bijection between the type and a suitably-sized Fin:

```
class IsEnum (α : Type) where
  size : Nat
  toIdx : α → Fin size
  fromIdx : Fin size → α
  to_from_id : ∀ (i : Fin size), toIdx (fromIdx i) = i
  from_to_id : ∀ (x : α), fromIdx (toIdx x) = x
```

For inductive types that are trivial enumerations, where no constructor expects any parameters, instances of this class are quite repetitive. The instance for `Bool` is typical:

```
instance : IsEnum Bool where
  size := 2
  toIdx
    | false => 0
    | true => 1
  fromIdx
    | 0 => false
    | 1 => true
  to_from_id
    | 0 => rfl
    | 1 => rfl
  from_to_id
    | false => rfl
    | true => rfl
```

The deriving handler programmatically constructs each pattern case, by analogy to the IsEnum Bool implementation:

```
open Lean Elab Parser Term Command

def deriveIsEnum (declNames : Array Name) : CommandElabM Bool :=
do
  if h : declNames.size = 1 then
    let env ← getEnv
    if let some (.inductInfo ind) := env.find? declNames[0] then
      let mut tos : Array (TSyntax ``matchAlt) := #[]
      let mut froms := #[]
      let mut to_froms := #[]
      let mut from_tos := #[]
      let mut i := 0

      for ctorName in ind.ctors do
        let c := mkIdent ctorName
        let n := Syntax.mkNumLit (toString i)

        tos      := tos.push      (← `(matchAltExpr| | $c => $n))
        from_tos := from_tos.push (← `(matchAltExpr| | $c =>
rfl))
        froms    := froms.push    (← `(matchAltExpr| | $n => $c))
        to_froms := to_froms.push (← `(matchAltExpr| | $n =>
rfl))

        i := i + 1

      let cmd ← `(instance : IsEnum $(mkIdent declNames[0]) where
                   size := $(quote ind.ctors.length)
                   toIdx $tos:matchAlt*
                   fromIdx $froms:matchAlt*
                   to_from_id $to_froms:matchAlt*
                   from_to_id $from_tos:matchAlt*)
      elabCommand cmd

      return true
  return false

initialize
  registerDerivingHandler ``IsEnum deriveIsEnum
```

11.5. Basic Classes

Many Lean type classes exist in order to allow built-in notations such as addition or array indexing to be overloaded.

11.5.1. Boolean Equality Tests

The Boolean equality operator == is overloaded by defining instances of BEq. The companion class Hashable specifies a hashing procedure for a type. When a type has both BEq and Hashable instances, then the hashes computed should respect the BEq instance: two values equated by BEq.bEq should always have the same hash.

type class

```
BEq.{u} (α : Type u) : Type u
```

BEq α is a typeclass for supplying a boolean-valued equality relation on α, notated as a == b. Unlike DecidableEq α (which uses a = b), this is Bool valued instead of Prop valued, and it also does not have any axioms like being reflexive or agreeing with =. It is mainly intended for programming applications. See LawfulBEq for a version that requires that == and = coincide.

Typically we prefer to put the "more variable" term on the left, and the "more constant" term on the right.

Instance Constructor

BEq.mk.{u}

Methods

beq : α → α → Bool

Boolean equality, notated as a == b.

Conventions for notations in identifiers:

- The recommended spelling of == in identifiers is *beq*.

type class

```
Hashable.{u} (α : Sort u) : Sort (max 1 u)
```

Types that can be hashed into a UInt64.

Instance Constructor

Hashable.mk.{u}

Methods

hash : α → UInt64

Hashes a value into a UInt64.

opaque

```
mixHash (u₁ u₂ : UInt64) : UInt64
```

An opaque hash mixing operation, used to implement hashing for products.

type class

```
LawfulBEq.{u} (α : Type u) [BEq α] : Prop
```

A Boolean equality test coincides with propositional equality.

In other words:

- a == b implies $a = b$.
- a == a is true.

Instance Constructor

LawfulBEq.mk.{u}

Extends

☑ ReflBEq α

Methods

rfl : ∀ {a : α}, (a == a) = true

Inherited from ReflBEq α

eq_of_beq : ∀ {a b : α}, (a == b) = true → a = b

If a == b evaluates to true, then a and b are equal in the logic.

type class

```
EquivBEq.{u_1} (α : Type u_1) [BEq α] : Prop
```

EquivBEq says that the BEq implementation is an equivalence relation.

Instance Constructor

EquivBEq.mk.{u_1}

Extends

☑ PartialEquivBEq α ☑ ReflBEq α

Methods

`symm : ∀ {a b : α}, (a == b) = true → (b == a) = true`

Inherited from `PartialEquivBEq α` > `ReflBEq α`

`trans : ∀ {a b c : α}, (a == b) = true → (b == c) = true → (a == c) = true`

Inherited from `PartialEquivBEq α` > `ReflBEq α`

`rfl : ∀ {a : α}, (a == a) = true`

Inherited from `PartialEquivBEq α` > `ReflBEq α`

type class

`LawfulHashable.{u} (α : Type u) [BEq α] [Hashable α] : Prop`

The `BEq α` and `Hashable α` instances on `α` are compatible. This means that that `a == b` implies `hash a = hash b`.

This is automatic if the `BEq` instance is lawful.

Instance Constructor

`LawfulHashable.mk.{u}`

Methods

`hash_eq : ∀ (a b : α), (a == b) = true → hash a = hash b`

If `a == b`, then `hash a = hash b`.

theorem

`hash_eq.{u_1} {α : Type u_1} [BEq α] [Hashable α] [LawfulHashable α] {a b : α} : (a == b) = true → hash a = hash b`

A lawful hash function respects its Boolean equality test.

11.5.2. Ordering

There are two primary ways to order the values of a type:

- The `Ord` type class provides a three-way comparison operator, `compare`, which can indicate that one value is less than, equal to, or greater than another. It returns an `Ordering`.

- The LT and LE classes provide canonical Prop-valued ordering relations for a type that do not need to be decidable. These relations are used to overload the < and ≤ operators.

type class

```
Ord.{u} (α : Type u) : Type u
```

Ord α provides a computable total order on α, in terms of the `compare : α → α → Ordering` function.

Typically instances will be transitive, reflexive, and antisymmetric, but this is not enforced by the typeclass.

There is a derive handler, so appending **`deriving Ord`** to an inductive type or structure will attempt to create an Ord instance.

Instance Constructor

Ord.mk.{u}

Methods

`compare : α → α → Ordering`

Compare two elements in α using the comparator contained in an [Ord α] instance.

The compare method is exported, so no explicit Ord namespace is required to use it.

def

```
compareOn.{u_1, u_2} {β : Type u_1} {α : Sort u_2} [ord : Ord β]
  (f : α → β) (x y : α) : Ordering
```

Compares two values by comparing the results of applying a function.

In particular, *x* is compared to *y* by comparing *f x* and *f y*.

Examples:

- compareOn (·.length) "apple" "banana" = .lt
- compareOn (· % 3) 5 6 = .gt
- compareOn (·.foldl max 0) [1, 2, 3] [3, 2, 1] = .eq

def

```
Ord.opposite.{u_1} {α : Type u_1} (ord : Ord α) : Ord α
```

Inverts the order of an Ord instance.

The result is an `Ord α` instance that returns `Ordering.lt` when *ord* would return `Ordering.gt` and that returns `Ordering.gt` when *ord* would return `Ordering.lt`.

inductive type

`Ordering : Type`

The result of a comparison according to a total order.

The relationship between the compared items may be:

- `Ordering.lt`: less than
- `Ordering.eq`: equal
- `Ordering.gt`: greater than

Constructors

| `lt : Ordering`

Less than.

| `eq : Ordering`

Equal.

| `gt : Ordering`

Greater than.

def

`Ordering.swap : Ordering → Ordering`

Swaps less-than and greater-than ordering results.

Examples:

- `Ordering.lt.swap = Ordering.gt`
- `Ordering.eq.swap = Ordering.eq`
- `Ordering.gt.swap = Ordering.lt`

def

`Ordering.then (a b : Ordering) : Ordering`

If *a* and *b* are `Ordering`, then *a*`.then` *b* returns *a* unless it is `.eq`, in which case it returns *b*. Additionally, it has “short-circuiting” behavior similar to boolean &&: if *a* is not `.eq` then the expression for *b* is not evaluated.

This is a useful primitive for constructing lexicographic comparator functions. The **`deriving`** `Ord` syntax on a structure uses the `Ord` instance to compare each field in order, combining the results equivalently to `Ordering.then`.

Use `compareLex` to lexicographically combine two comparison functions.

Examples:

```
structure Person where
  name : String
  age : Nat

-- Sort people first by name (in ascending order), and people
with the same name by age (in
-- descending order)
instance : Ord Person where
  compare a b := (compare a.name b.name).then (compare b.age
a.age)

#eval Ord.compare (⟨"Gert", 33⟩ : Person) ⟨"Dana", 50⟩

Ordering.gt

#eval Ord.compare (⟨"Gert", 33⟩ : Person) ⟨"Gert", 50⟩

Ordering.gt

#eval Ord.compare (⟨"Gert", 33⟩ : Person) ⟨"Gert", 20⟩

Ordering.lt
```

def

```
Ordering.isLT : Ordering → Bool
```

Checks whether the ordering is `lt`.

def

```
Ordering.isLE : Ordering → Bool
```

Checks whether the ordering is `lt` or `eq`.

def

```
Ordering.isEq : Ordering → Bool
```

Checks whether the ordering is `eq`.

def

```
Ordering.isNe : Ordering → Bool
```

Checks whether the ordering is not `eq`.

def

```
Ordering.isGE : Ordering → Bool
```

Checks whether the ordering is `gt` or `eq`.

def

```
Ordering.isGT : Ordering → Bool
```

Checks whether the ordering is `gt`.

def

```
compareOfLessAndEq.{u_1} {α : Type u_1} (x y : α) [LT α]
  [Decidable (x ≤ y)] [DecidableEq α] : Ordering
```

Uses decidable less-than and equality relations to find an `Ordering`.

In particular, if $x < y$ then the result is `Ordering.lt`. If $x = y$ then the result is `Ordering.eq`. Otherwise, it is `Ordering.gt`.

`compareOfLessAndBEq` uses `BEq` instead of `DecidableEq`.

def

```
compareOfLessAndBEq.{u_1} {α : Type u_1} (x y : α) [LT α]
  [Decidable (x ≤ y)] [BEq α] : Ordering
```

Uses a decidable less-than relation and Boolean equality to find an `Ordering`.

In particular, if $x < y$ then the result is `Ordering.lt`. If $x == y$ then the result is `Ordering.eq`. Otherwise, it is `Ordering.gt`.

`compareOfLessAndEq` uses `DecidableEq` instead of `BEq`.

def

```
compareLex.{u_1, u_2} {α : Sort u_1} {β : Sort u_2}
  (cmp₁ cmp₂ : α → β → Ordering) (a : α) (b : β) : Ordering
```

Compares *a* and *b* lexicographically by cmp_1 and cmp_2.

a and *b* are first compared by cmp_1. If this returns `Ordering.eq`, *a* and *b* are compared by cmp_2 to break the tie.

To lexicographically combine two `Ordering`s, use `Ordering.then`.

syntax

Ordering Operators

The less-than operator is overloaded in the `LT` class:

```
term ::= ...
    | term < term
```

The less-than-or-equal-to operator is overloaded in the `LE` class:

```
term ::= ...
    | term ≤ term
```

The greater-than and greater-than-or-equal-to operators are the reverse of the less-than and less-than-or-equal-to operators, and cannot be independently overloaded:

```
term ::= ...
    | term > term
```

```
term ::= ...
    | term ≥ term
```

type class

`LT.{u} (α : Type u) : Type u`

`LT α` is the typeclass which supports the notation `x < y` where `x y : α`.

Instance Constructor

`LT.mk.{u}`

Methods

`lt : α → α → Prop`

The less-than relation: `x < y`

Conventions for notations in identifiers:

- The recommended spelling of < in identifiers is *lt*.

type class

`LE.{u} (α : Type u) : Type u`

LE *α* is the typeclass which supports the notation `x ≤ y` where `x y : α`.

Instance Constructor

LE.mk.{u}

Methods

`le : α → α → Prop`

The less-equal relation: `x ≤ y`

Conventions for notations in identifiers:

- The recommended spelling of ≤ in identifiers is *le*.
- The recommended spelling of <= in identifiers is *le* (prefer ≤ over <=).

An Ord can be used to construct BEq, LT, and LE instances with the following helpers. They are not automatically instances because many types are better served by custom relations.

def

`ltOfOrd.{u_1} {α : Type u_1} [Ord α] : LT α`

Constructs an LT instance from an Ord instance that asserts that the result of compare is Ordering.lt.

def

`leOfOrd.{u_1} {α : Type u_1} [Ord α] : LE α`

Constructs an LT instance from an Ord instance that asserts that the result of compare satisfies Ordering.isLE.

def

`Ord.toBEq.{u_1} {α : Type u_1} (ord : Ord α) : BEq α`

Constructs a BEq instance from an Ord instance.

def

```
Ord.toLE.{u_1} {α : Type u_1} (ord : Ord α) : LE α
```

Constructs an `LE` instance from an `Ord` instance.

def

```
Ord.toLT.{u_1} {α : Type u_1} (ord : Ord α) : LT α
```

Constructs an `LT` instance from an `Ord` instance.

▼ *Example: Using `Ord` Instances for `LT` and `LE` Instances*

Lean can automatically derive an `Ord` instance. In this case, the `Ord Vegetable` instance compares vegetables lexicographically:

```
structure Vegetable where
  color : String
  size : Fin 5
deriving Ord

def broccoli : Vegetable where
  color := "green"
  size := 2

def sweetPotato : Vegetable where
  color := "orange"
  size := 3
```

Using the helpers `ltOfOrd` and `leOfOrd`, `LT Vegetable` and `LE Vegetable` instances can be defined. These instances compare the vegetables using `compare` and logically assert that the result is as expected.

```
instance : LT Vegetable := ltOfOrd
instance : LE Vegetable := leOfOrd
```

The resulting relations are decidable because equality is decidable for `Ordering`:

```
#eval broccoli < sweetPotato
```

```
true
```

```
#eval broccoli ≤ sweetPotato
```

```
true
```

```
#eval broccoli < broccoli
```

```
false
```

```
#eval broccoli ≤ broccoli
```

```
true
```

11.5.2.1. Instance Construction

def

```
Ord.lex.{u_1, u_2} {α : Type u_1} {β : Type u_2} :
  Ord α → Ord β → Ord (α × β)
```

Constructs the lexicographic order on products $\alpha \times \beta$ from orders for α and β.

def

```
Ord.lex'.{u_1} {α : Type u_1} (ord₁ ord₂ : Ord α) : Ord α
```

Constructs an `Ord` instance from two existing instances by combining them lexicographically.

The resulting instance compares elements first by ord_1 and then, if this returns `Ordering.eq`, by ord_2.

The function `compareLex` can be used to perform this comparison without constructing an intermediate `Ord` instance. `Ordering.then` can be used to lexicographically combine the results of comparisons.

def

```
Ord.on.{u_1, u_2} {β : Type u_1} {α : Type u_2} :
  Ord β → (f : α → β) → Ord α
```

Constructs an `Ord` instance that compares values according to the results of f.

In particular, `ord.on f` compares x and y by comparing $f\ x$ and $f\ y$ according to `ord`.

The function `compareOn` can be used to perform this comparison without constructing an intermediate `Ord` instance.

11.5.3. Minimum and Maximum Values

The classes `Max` and `Min` provide overloaded operators for choosing the greater or lesser of two values. These should be in agreement with `Ord`, `LT`, and `LE` instances, if they exist, but there is no mechanism to enforce this.

type class

```
Min.{u} (α : Type u) : Type u
```

An overloaded operation to find the lesser of two values of type α.

Instance Constructor

Min.mk.{u}

Methods

min : $\alpha \to \alpha \to \alpha$

Returns the lesser of its two arguments.

type class

Max.{u} (α : Type u) : Type u

An overloaded operation to find the greater of two values of type α.

Instance Constructor

Max.mk.{u}

Methods

max : $\alpha \to \alpha \to \alpha$

Returns the greater of its two arguments.

Given an LE α instance for which LE.le is decidable, the helpers minOfLe and maxOfLe can be used to create suitable Min α and Max α instances. They can be used as the right-hand side of an instance declaration.

def

minOfLe.{u_1} {α : Type u_1} [LE α] [DecidableRel LE.le] : Min α

Constructs a Min instance from a decidable ≤ operation.

def

maxOfLe.{u_1} {α : Type u_1} [LE α] [DecidableRel LE.le] : Max α

Constructs a Max instance from a decidable ≤ operation.

11.5.4. Decidability

A proposition is *decidable* if it can be checked algorithmically. The Law of the Excluded Middle means that every proposition is true or false, but it provides no way to check which of the two cases holds, which can often be useful. By default, only algorithmic `Decidable` instances for which code can be generated are in scope; opening the `Classical` namespace makes every proposition decidable.

inductive type

```
Decidable (p : Prop) : Type
```

Either a proof that *p* is true or a proof that *p* is false. This is equivalent to a `Bool` paired with a proof that the `Bool` is `true` if and only if *p* is true.

`Decidable` instances are primarily used via **`if`**-expressions and the tactic **`decide`**. In conditional expressions, the `Decidable` instance for the proposition is used to select a branch. At run time, this case distinction code is identical to that which would be generated for a `Bool`-based conditional. In proofs, the tactic **`decide`** synthesizes an instance of `Decidable` *p*, attempts to reduce it to `isTrue` *h*, and then succeeds with the proof *h* if it can.

Because `Decidable` carries data, when writing `@[simp]` lemmas which include a `Decidable` instance on the LHS, it is best to use `{_ : Decidable p}` rather than `[Decidable p]` so that non-canonical instances can be found via unification rather than instance synthesis.

Constructors

| `isFalse {p : Prop} (h : ¬p) : Decidable p`

  Proves that *p* is decidable by supplying a proof of ¬*p*

| `isTrue {p : Prop} (h : p) : Decidable p`

  Proves that *p* is decidable by supplying a proof of *p*

def

```
DecidablePred.{u} {α : Sort u} (r : α → Prop) : Sort (max 1 u)
```

A decidable predicate.

A predicate is decidable if the corresponding proposition is `Decidable` for each possible argument.

def

```
DecidableRel.{u, v} {α : Sort u} {β : Sort v} (r : α → β → Prop) :
  Sort (max (max 1 u) v)
```

A decidable relation.

A relation is decidable if the corresponding proposition is `Decidable` for all possible arguments.

def

```
DecidableEq.{u} (α : Sort u) : Sort (max 1 u)
```

Propositional equality is `Decidable` for all elements of a type.

In other words, an instance of `DecidableEq α` is a means of deciding the proposition `a = b` is for all `a b : α`.

def

```
DecidableLT.{u} (α : Type u) [LT α] : Type u
```

Abbreviation for `DecidableRel (· < · : α → α → Prop)`.

def

```
DecidableLE.{u} (α : Type u) [LE α] : Type u
```

Abbreviation for `DecidableRel (· ≤ · : α → α → Prop)`.

def

```
Decidable.decide (p : Prop) [h : Decidable p] : Bool
```

Converts a decidable proposition into a `Bool`.

If `p : Prop` is decidable, then `decide p : Bool` is the Boolean value that is `true` if `p` is true and `false` if `p` is false.

def

```
Decidable.byCases.{u} {p : Prop} {q : Sort u} [dec : Decidable p]
  (h1 : p → q) (h2 : ¬p → q) : q
```

Construct a `q` if some proposition `p` is decidable, and both the truth and falsity of `p` are sufficient to construct a `q`.

This is a synonym for `dite`, the dependent if-then-else operator.

▼ *Example: Excluded Middle and* `Decidable`

The equality of functions from `Nat` to `Nat` is not decidable:

```
example (f g : Nat → Nat) : Decidable (f = g) := inferInstance
```

```
failed to synthesize
  Decidable (f = g)

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Opening `Classical` makes every proposition decidable; however, declarations and examples that use this fact must be marked **`noncomputable`** to indicate that code should not be generated for them.

```
open Classical
noncomputable example (f g : Nat → Nat) : Decidable (f = g) :=
inferInstance
```

11.5.5. Inhabited Types

type class

`Inhabited.{u} (α : Sort u) : Sort (max 1 u)`

`Inhabited α` is a typeclass that says that α has a designated element, called `(default : α)`. This is sometimes referred to as a "pointed type".

This class is used by functions that need to return a value of the type when called "out of domain". For example, `Array.get! arr i : α` returns a value of type α when `arr : Array α`, but if i is not in range of the array, it reports a panic message, but this does not halt the program, so it must still return a value of type α (and in fact this is required for logical consistency), so in this case it returns `default`.

Instance Constructor

`Inhabited.mk.{u}`

Methods

`default : α`

`default` is a function that produces a "default" element of any `Inhabited` type. This element does not have any particular specified properties, but it is often an all-zeroes value.

inductive predicate

```
Nonempty.{u} (α : Sort u) : Prop
```

`Nonempty α` is a typeclass that says that `α` is not an empty type, that is, there exists an element in the type. It differs from `Inhabited α` in that `Nonempty α` is a `Prop`, which means that it does not actually carry an element of `α`, only a proof that *there exists* such an element. Given `Nonempty α`, you can construct an element of `α` *nonconstructively* using `Classical.choice`.

Constructors

```
| intro.{u} {α : Sort u} (val : α) : Nonempty α
```

If `val : α`, then `α` is nonempty.

11.5.6. Subsingleton Types

type class

```
Subsingleton.{u} (α : Sort u) : Prop
```

A *subsingleton* is a type with at most one element. It is either empty or has a unique element.

All propositions are subsingletons because of proof irrelevance: false propositions are empty, and all proofs of a true proposition are equal to one another. Some non-propositional types are also subsingletons.

Instance Constructor

`Subsingleton.intro.{u}`

Prove that `α` is a subsingleton by showing that any two elements are equal.

Methods

```
allEq : ∀ (a b : α), a = b
```

Any two elements of a subsingleton are equal.

theorem

```
Subsingleton.elim.{u} {α : Sort u} [h : Subsingleton α] (a b : α) :
  a = b
```

If a type is a subsingleton, then all of its elements are equal.

theorem

```
Subsingleton.helim.{u} {α β : Sort u} [h₁ : Subsingleton α] (h₂ : α = β)
  (a : α) (b : β) : a ≍ b
```

If two types are equal and one of them is a subsingleton, then all of their elements are heterogeneously equal.

11.5.7. Arithmetic and Bitwise Operators

type class

```
Zero.{u} (α : Type u) : Type u
```

A type with a zero element.

Instance Constructor

Zero.mk.{u}

Methods

zero : α

The zero element of the type.

type class

```
NeZero.{u_1} {R : Type u_1} [Zero R] (n : R) : Prop
```

A type-class version of $n \neq 0$.

Instance Constructor

NeZero.mk.{u_1}

Methods

out : n ≠ 0

The proposition that n is not zero.

type class

```
HAdd.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The notation typeclass for heterogeneous addition. This enables the notation `a + b : γ` where `a : α, b : β`.

Instance Constructor

`HAdd.mk.{u, v, w}`

Methods

`hAdd : α → β → γ`

`a + b` computes the sum of `a` and `b`. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of + in identifiers is add.

type class

`Add.{u} (α : Type u) : Type u`

The homogeneous version of HAdd: `a + b : α` where `a b : α`.

Instance Constructor

`Add.mk.{u}`

Methods

`add : α → α → α`

`a + b` computes the sum of `a` and `b`. See HAdd.

type class

```
HSub.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The notation typeclass for heterogeneous subtraction. This enables the notation `a - b : γ` where `a : α, b : β`.

Instance Constructor

`HSub.mk.{u, v, w}`

Methods

`hSub : α → β → γ`

a - b computes the difference of *a* and *b*. The meaning of this notation is type-dependent.

- For natural numbers, this operator saturates at 0: *a* - *b* = 0 when *a* ≤ *b*.

Conventions for notations in identifiers:

- The recommended spelling of − in identifiers is sub (when used as a binary operator).

type class

```
Sub.{u} (α : Type u) : Type u
```

The homogeneous version of HSub: a - b : α where a b : α.

Instance Constructor

Sub.mk.{u}

Methods

```
sub : α → α → α
```

a - b computes the difference of *a* and *b*. See HSub.

type class

```
HMul.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The notation typeclass for heterogeneous multiplication. This enables the notation a * b : γ where *a* : *α*, *b* : *β*.

Instance Constructor

HMul.mk.{u, v, w}

Methods

```
hMul : α → β → γ
```

a * b computes the product of *a* and *b*. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of * in identifiers is mul.

type class

```
Mul.{u} (α : Type u) : Type u
```

The homogeneous version of HMul: a * b : α where a b : α.

Instance Constructor

Mul.mk.{u}

Methods

mul : α → α → α

a * b computes the product of *a* and *b*. See HMul.

type class

HDiv.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
Type (max (max u v) w)

The notation typeclass for heterogeneous division. This enables the notation a / b : γ where *a* : α, *b* : β.

Instance Constructor

HDiv.mk.{u, v, w}

Methods

hDiv : α → β → γ

a / b computes the result of dividing *a* by *b*. The meaning of this notation is type-dependent.

- For most types like Nat, Int, Rat, Real, *a* / 0 is defined to be 0.
- For Nat, *a* / *b* rounds downwards.
- For Int, *a* / *b* rounds downwards if *b* is positive or upwards if *b* is negative. It is implemented as Int.ediv, the unique function satisfying *a* % *b* + *b* * (*a* / *b*) = *a* and 0 ≤ a % b < natAbs b for *b* ≠ 0. Other rounding conventions are available using the functions Int.fdiv (floor rounding) and Int.tdiv (truncation rounding).
- For Float, *a* / 0 follows the IEEE 754 semantics for division, usually resulting in inf or nan.

Conventions for notations in identifiers:

- The recommended spelling of / in identifiers is div.

type class

```
Div.{u} (α : Type u) : Type u
```

The homogeneous version of HDiv: a / b : α where a b : α.

Instance Constructor

Div.mk.{u}

Methods

```
div : α → α → α
```

a / b computes the result of dividing *a* by *b*. See HDiv.

type class

```
Dvd.{u_1} (α : Type u_1) : Type u_1
```

Notation typeclass for the ∣ operation (typed as \|), which represents divisibility.

Instance Constructor

Dvd.mk.{u_1}

Methods

```
dvd : α → α → Prop
```

Divisibility. a ∣ b (typed as \|) means that there is some *c* such that b = a * c.

Conventions for notations in identifiers:

- The recommended spelling of ∣ in identifiers is *dvd*.

type class

```
HMod.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The notation typeclass for heterogeneous modulo / remainder. This enables the notation a % b : γ where *a* : *α*, *b* : *β*.

Instance Constructor

HMod.mk.{u, v, w}

Methods

`hMod : α → β → γ`

a `%` b computes the remainder upon dividing *a* by *b*. The meaning of this notation is type-dependent.

- For `Nat` and `Int` it satisfies a `% b + b * (a / b) = a`, and *a* `% 0` is defined to be *a*.

Conventions for notations in identifiers:

- The recommended spelling of `%` in identifiers is `mod`.

type class

`Mod.{u} (α : Type u) : Type u`

The homogeneous version of `HMod`: a `% b : α` where a `b : α`.

Instance Constructor

`Mod.mk.{u}`

Methods

`mod : α → α → α`

a `%` b computes the remainder upon dividing *a* by *b*. See `HMod`.

type class

```
HPow.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The notation typeclass for heterogeneous exponentiation. This enables the notation a `^ b : γ` where a `: α`, *b* `: β`.

Instance Constructor

`HPow.mk.{u, v, w}`

Methods

`hPow : α → β → γ`

a `^` b computes *a* to the power of *b*. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of ^ in identifiers is pow.

type class

`Pow.{u, v} (α : Type u) (β : Type v) : Type (max u v)`

The homogeneous version of HPow: a ^ b : α where *a* : *α*, *b* : *β*. (The right argument is not the same as the left since we often want this even in the homogeneous case.)

Types can choose to subscribe to particular defaulting behavior by providing an instance to either NatPow or HomogeneousPow:

- NatPow is for types whose exponents is preferentially a Nat.
- HomogeneousPow is for types whose base and exponent are preferentially the same.

Instance Constructor

Pow.mk.{u, v}

Methods

pow : *α* → *β* → *α*

a ^ b computes *a* to the power of *b*. See HPow.

type class

`NatPow.{u} (α : Type u) : Type u`

The homogeneous version of Pow where the exponent is a Nat. The purpose of this class is that it provides a default Pow instance, which can be used to specialize the exponent to Nat during elaboration.

For example, if x ^ 2 should preferentially elaborate with 2 : Nat then *x*'s type should provide an instance for this class.

Instance Constructor

NatPow.mk.{u}

Methods

pow : *α* → Nat → *α*

a ^ n computes *a* to the power of *n* where *n* : Nat. See Pow.

type class

```
HomogeneousPow.{u} (α : Type u) : Type u
```

The completely homogeneous version of Pow where the exponent has the same type as the base. The purpose of this class is that it provides a default Pow instance, which can be used to specialize the exponent to have the same type as the base's type during elaboration. This is to say, a type should provide an instance for this class in case x ^ y should be elaborated with both *x* and *y* having the same type.

For example, the Float type provides an instance of this class, which causes expressions such as (2.2 ^ 2.2 : Float) to elaborate.

Instance Constructor

HomogeneousPow.mk.{u}

Methods

pow : α → α → α

a ^ b computes *a* to the power of *b* where *a* and *b* both have the same type.

type class

```
HShiftLeft.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type
w)) :
  Type (max (max u v) w)
```

The typeclass behind the notation a <<< b : γ where *a* : α, *b* : β.

Instance Constructor

HShiftLeft.mk.{u, v, w}

Methods

hShiftLeft : α → β → γ

a <<< b computes *a* shifted to the left by *b* places. The meaning of this notation is type-dependent.

- On Nat, this is equivalent to *a* * 2 ^ *b*.
- On UInt8 and other fixed width unsigned types, this is the same but truncated to the bit width.

Conventions for notations in identifiers:

- The recommended spelling of <<< in identifiers is `shiftLeft`.

type class

```
ShiftLeft.{u} (α : Type u) : Type u
```

The homogeneous version of HShiftLeft: a <<< b : α where a b : α.

Instance Constructor

ShiftLeft.mk.{u}

Methods

```
shiftLeft : α → α → α
```

The implementation of a <<< b : α. See HShiftLeft.

type class

```
HShiftRight.{u, v, w} (α : Type u) (β : Type v)
  (γ : outParam (Type w)) : Type (max (max u v) w)
```

The typeclass behind the notation a >>> b : γ where $a : \alpha, b : \beta$.

Instance Constructor

HShiftRight.mk.{u, v, w}

Methods

```
hShiftRight : α → β → γ
```

a >>> b computes a shifted to the right by b places. The meaning of this notation is type-dependent.

- On Nat and fixed width unsigned types like UInt8, this is equivalent to a / 2 ^ b.

Conventions for notations in identifiers:

- The recommended spelling of >>> in identifiers is `shiftRight`.

type class

```
ShiftRight.{u} (α : Type u) : Type u
```

The homogeneous version of HShiftRight: a >>> b : α where a b : α.

Instance Constructor

`ShiftRight.mk.{u}`

Methods

`shiftRight : α → α → α`

The implementation of a `>>>` b : α. See `HShiftRight`.

type class

`Neg.{u} (α : Type u) : Type u`

The notation typeclass for negation. This enables the notation −a : α where *a* : *α*.

Instance Constructor

`Neg.mk.{u}`

Methods

`neg : α → α`

−*a* computes the negative or opposite of *a*. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of − in identifiers is *neg* (when used as a unary operator).

type class

```
HAnd.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The typeclass behind the notation a `&&&` b : γ where *a* : *α*, *b* : *β*.

Instance Constructor

`HAnd.mk.{u, v, w}`

Methods

`hAnd : α → β → γ`

a `&&&` b computes the bitwise AND of *a* and *b*. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of &&& in identifiers is and.

type class

```
HOr.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The typeclass behind the notation a ||| b : γ where *a* : *α*, *b* : *β*.

Instance Constructor

HOr.mk.{u, v, w}

Methods

hOr : *α* → *β* → *γ*

a ||| b computes the bitwise OR of *a* and *b*. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of ||| in identifiers is or.

type class

```
HXor.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w)) :
  Type (max (max u v) w)
```

The typeclass behind the notation a ^^^ b : γ where *a* : *α*, *b* : *β*.

Instance Constructor

HXor.mk.{u, v, w}

Methods

hXor : *α* → *β* → *γ*

a ^^^ b computes the bitwise XOR of *a* and *b*. The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of ^^^ in identifiers is xor.

11.5.8. Append

type class

```
HAppend.{u, v, w} (α : Type u) (β : Type v) (γ : outParam (Type w))
:
  Type (max (max u v) w)
```

The notation typeclass for heterogeneous append. This enables the notation `a ++ b : γ` where `a : α, b : β`.

Instance Constructor

`HAppend.mk.{u, v, w}`

Methods

`hAppend : α → β → γ`

`a ++ b` is the result of concatenation of *a* and *b*, usually read "append". The meaning of this notation is type-dependent.

Conventions for notations in identifiers:

- The recommended spelling of ++ in identifiers is append.

type class

```
Append.{u} (α : Type u) : Type u
```

The homogeneous version of HAppend: `a ++ b : α` where `a b : α`.

Instance Constructor

`Append.mk.{u}`

Methods

`append : α → α → α`

`a ++ b` is the result of concatenation of *a* and *b*. See HAppend.

11.5.9. Data Lookups

type class

```
GetElem.{u, v, w} (coll : Type u) (idx : Type v)
  (elem : outParam (Type w)) (valid : outParam (coll → idx → Prop))
:
  Type (max (max u v) w)
```

The classes GetElem and GetElem? implement lookup notation, specifically `xs[i]`, `xs[i]?`, `xs[i]!`, and `xs[i]'p`.

Both classes are indexed by types *coll*, *idx*, and *elem* which are the collection, the index, and the element types. A single collection may support lookups with multiple index types. The relation *valid* determines when the index is guaranteed to be valid; lookups of valid indices are guaranteed not to fail.

For example, an instance for arrays looks like `GetElem (Array α) Nat α (fun xs i => i < xs.size)`. In other words, given an array *xs* and a natural number *i*, `xs[i]` will return an *α* when *valid xs i* holds, which is true when *i* is less than the size of the array. Array additionally supports indexing with USize instead of Nat. In either case, because the bounds are checked at compile time, no runtime check is required.

Given `xs[i]` with *xs : coll* and *i : idx*, Lean looks for an instance of *GetElem coll idx elem valid* and uses this to infer the type of the return value *elem* and side condition *valid* required to ensure *xs[i]* yields a valid value of type *elem*. The tactic **get_elem_tactic** is invoked to prove validity automatically. The *xs[i]'p* notation uses the proof *p* to satisfy the validity condition. If the proof *p* is long, it is often easier to place the proof in the context using **have**, because **get_elem_tactic** tries **assumption**.

The proof side-condition *valid xs i* is automatically dispatched by the **get_elem_tactic** tactic; this tactic can be extended by adding more clauses to **get_elem_tactic_extensible** using **macro_rules**.

`xs[i]?` and `xs[i]!` do not impose a proof obligation; the former returns an *Option elem*, with none signalling that the value isn't present, and the latter returns *elem* but panics if the value isn't there, returning `default : elem` based on the *Inhabited elem* instance. These are provided by the GetElem? class, for which there is a default instance generated from a GetElem class as long as *valid xs i* is always decidable.

Important instances include:

- `arr[i] : α` where *arr* : Array *α* and *i* : Nat or `i : USize`: does array indexing with no runtime bounds check and a proof side goal *i* < *arr*.size.
- `l[i] : α` where `l : List α` and *i* : Nat: index into a list, with proof side goal *i* < *l*.length.

Instance Constructor

GetElem.mk.{u, v, w}

Methods

getElem : (*xs* : *coll*) → (*i* : *idx*) → *valid xs i* → *elem*

The syntax `arr[i]` gets the *i*'th element of the collection *arr*. If there are proof side conditions to the application, they will be automatically inferred by the **get_elem_tactic** tactic.

Conventions for notations in identifiers:

- The recommended spelling of `xs[i]` in identifiers is `getElem`.
- The recommended spelling of `xs[i]'h` in identifiers is `getElem`.

type class

```
GetElem?.{u, v, w} (coll : Type u) (idx : Type v)
  (elem : outParam (Type w)) (valid : outParam (coll → idx → Prop))
:
  Type (max (max u v) w)
```

The classes `GetElem` and `GetElem?` implement lookup notation, specifically `xs[i]`, `xs[i]?`, `xs[i]!`, and `xs[i]'p`.

Both classes are indexed by types *coll*, *idx*, and *elem* which are the collection, the index, and the element types. A single collection may support lookups with multiple index types. The relation *valid* determines when the index is guaranteed to be valid; lookups of valid indices are guaranteed not to fail.

For example, an instance for arrays looks like `GetElem (Array α) Nat α (fun xs i => i < xs.size)`. In other words, given an array *xs* and a natural number *i*, `xs[i]` will return an *α* when *valid xs i* holds, which is true when *i* is less than the size of the array. `Array` additionally supports indexing with `USize` instead of `Nat`. In either case, because the bounds are checked at compile time, no runtime check is required.

Given `xs[i]` with *xs : coll* and *i : idx*, Lean looks for an instance of `GetElem coll idx elem valid` and uses this to infer the type of the return value *elem* and side condition *valid* required to ensure *xs*`[`*i*`]` yields a valid value of type *elem*. The tactic **get_elem_tactic** is invoked to prove validity automatically. The *xs*`[`*i*`]'`*p* notation uses the proof *p* to satisfy the validity condition. If the proof *p* is long, it is often easier to place the proof in the context using **have**, because **get_elem_tactic** tries **assumption**.

The proof side-condition *valid xs i* is automatically dispatched by the **get_elem_tactic** tactic; this tactic can be extended by adding more clauses to **get_elem_tactic_extensible** using **macro_rules**.

`xs[i]?` and `xs[i]!` do not impose a proof obligation; the former returns an `Option elem`, with `none` signalling that the value isn't present, and the latter returns *elem* but panics if the value isn't there, returning `default : elem` based on the `Inhabited elem` instance. These are provided by the `GetElem?` class, for which there is a default instance generated from a `GetElem` class as long as *valid xs i* is always decidable.

Important instances include:

- `arr[i] : α` where `arr : Array α` and `i : Nat` or `i : USize`: does array indexing with no runtime bounds check and a proof side goal `i < arr.size`.
- `l[i] : α` where `l : List α` and `i : Nat`: index into a list, with proof side goal `i < l.length`.

Instance Constructor

`GetElem?.mk.{u, v, w}`

Extends

☑ `GetElem coll idx elem valid`

Methods

`getElem : (xs : coll) → (i : idx) → valid xs i → elem`

Inherited from `GetElem coll idx elem valid`

`getElem? : coll → idx → Option elem`

The syntax `arr[i]?` gets the `i`'th element of the collection `arr`, if it is present (and wraps it in `some`), and otherwise returns `none`.

Conventions for notations in identifiers:

- The recommended spelling of `xs[i]?` in identifiers is `getElem?`.

`getElem! : [Inhabited elem] → coll → idx → elem`

The syntax `arr[i]!` gets the `i`'th element of the collection `arr`, if it is present, and otherwise panics at runtime and returns the `default` term from `Inhabited elem`.

Conventions for notations in identifiers:

- The recommended spelling of `xs[i]!` in identifiers is `getElem!`.

type class

```
LawfulGetElem.{u, v, w} (cont : Type u) (idx : Type v)
  (elem : outParam (Type w)) (dom : outParam (cont → idx → Prop))
  [ge : GetElem? cont idx elem dom] : Prop
```

Lawful `GetElem?` instances (which extend `GetElem`) are those for which the potentially-failing `GetElem?.getElem?` and `GetElem?.getElem!` operators succeed when the validity predicate is satisfied, and fail when it is not.

Instance Constructor

`LawfulGetElem.mk.{u, v, w}`

Methods

```
getElem?_def : ∀ (c : cont) (i : idx) [inst : Decidable (dom c
i)], c[i]? = if h : dom c i then some c[i] else none
```

GetElem?.getElem? succeeds when the validity predicate is satisfied and fails otherwise.

```
getElem!_def : ∀ [inst : Inhabited elem] (c : cont) (i : idx),
  c[i]! =
    match c[i]? with
    | some e => e
    | none => default
```

GetElem?.getElem! succeeds and fails when GetElem.getElem? succeeds and fails.

12. Coercions

When the Lean elaborator is expecting one type but produces a term with a different type, it attempts to automatically insert a *coercion*, which is a specially designated function from the term's type to the expected type. Coercions make it possible to use specific types to represent data while interacting with APIs that expect less-informative types. They also allow mathematical developments to follow the usual practice of “punning”, where the same symbol is used to stand for both an algebraic structure and its carrier set, with the precise meaning determined by context.

Lean's standard library and metaprogramming APIs define many coercions. Some examples include:

- A Nat may be used where an Int is expected.
- A Fin may be used where a Nat is expected.
- An α may be used where an Option α is expected. The coercion wraps the value in some.
- An α may be used where a Thunk α is expected. The coercion wraps the term in a function to delay its evaluation.
- When one syntax category *c1* embeds into another category *c2*, a coercion from TSyntax *c1* to TSyntax *c2* performs any necessary wrapping to construct a valid syntax tree.

Coercions are found using type class synthesis. The set of coercions can be extended by adding further instances of the appropriate type classes.

▼ *Example: Coercions*

All of the following examples rely on coercions:

```
example (n : Nat) : Int := n
example (n : Fin k) : Nat := n
example (x : α) : Option α := x
```

```
def th (f : Int → String) (x : Nat) : Thunk String := f x

open Lean in
example (n : Ident) : Term := n
```

In the case of `th`, using `#print` demonstrates that evaluation of the function application is delayed until the thunk's value is requested:

```
#print th
```

```
def th : (Int → String) → Nat → Thunk String :=
fun f x => { fn := fun x_1 => f ↑x }
```

Coercions are not used to resolve generalized field notation: only the inferred type of the term is considered. However, a type ascription can be used to trigger a coercion to the type that has the desired generalized field. Coercions are also not used to resolve `OfNat` instances: even though there is a default instance for `OfNat Nat`, a coercion from `Nat` to α does not allow natural number literals to be used for α.

▼ *Example: Coercions and Generalized Field Notation*

The name `Nat.bdiv` is not defined, but `Int.bdiv` exists. The coercion from `Nat` to `Int` is not considered when looking up the field `bdiv`:

```
example (n : Nat) := n.bdiv 2
```

```
Invalid field `bdiv`: The environment does not contain
`Nat.bdiv`
  n
has type
  Nat
```

This is because coercions are only inserted when there is an expected type that differs from an inferred type, and generalized fields are resolved based on the inferred type of the term before the dot. Coercions can be triggered by adding a type ascription, which additionally causes the inferred type of the entire ascription term to be `Int`, allowing the function `Int.bdiv` to be found.

```
example (n : Nat) := (n : Int).bdiv 2
```

▼ *Example: Coercions and `OfNat`*

`Bin` is an inductive type that represents binary numbers.

```
inductive Bin where
  | done
```

```
  | zero : Bin → Bin
  | one : Bin → Bin

def Bin.toString : Bin → String
  | .done => ""
  | .one b => b.toString ++ "1"
  | .zero b => b.toString ++ "0"

instance : ToString Bin where
  toString
    | .done => "0"
    | b => Bin.toString b
```

Binary numbers can be converted to natural numbers by repeatedly applying `Bin.succ`:

```
def Bin.succ (b : Bin) : Bin :=
  match b with
  | .done => Bin.done.one
  | .zero b => .one b
  | .one b => .zero b.succ

def Bin.ofNat (n : Nat) : Bin :=
  match n with
  | 0 => .done
  | n + 1 => (Bin.ofNat n).succ
```

Even if `Bin.ofNat` is registered as a coercion, natural number literals cannot be used for `Bin`:

```
attribute [coe] Bin.ofNat

instance : Coe Nat Bin where
  coe := Bin.ofNat

#eval (9 : Bin)
```

```
failed to synthesize
  OfNat Bin 9
numerals are polymorphic in Lean, but the numeral `9` cannot
be used in a context where the expected type is
  Bin
due to the absence of the instance above

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

This is because coercions are inserted in response to mismatched types, but a failure to synthesize an `OfNat` instance is not a type mismatch.

The coercion can be used in the definition of the `OfNat Bin` instance:

```
instance : OfNat Bin n where
  ofNat := n
```

```
#eval (10 : Bin)
```

```
1010
```

Most new coercions can be defined by declaring an instance of the Coe type class and applying the **coe** attribute to the function that performs the coercion. To enable more control over coercions or to enable them in more contexts, Lean provides further classes that can be implemented, described in the rest of this chapter.

▼ *Example: Defining Coercions: Decimal Numbers*

Decimal numbers can be defined as arrays of digits.

```
structure Decimal where
  digits : Array (Fin 10)
```

Adding a coercion allows them to be used in contexts that expect Nat, but also contexts that expect any type that Nat can be coerced to.

```
@[coe]
def Decimal.toNat (d : Decimal) : Nat :=
  d.digits.foldl (init := 0) fun n d => n * 10 + d.val

instance : Coe Decimal Nat where
  coe := Decimal.toNat
```

This can be demonstrated by treating a Decimal as an Int as well as a Nat:

```
def twoHundredThirteen : Decimal where
  digits := #[2, 1, 3]

def one : Decimal where
  digits := #[1]

#eval (one : Int) - (twoHundredThirteen : Nat)
```

```
-212
```

type class

```
Coe.{u, v} (α : semiOutParam (Sort u)) (β : Sort v) :
  Sort (max (max 1 u) v)
```

Coe α β is the typeclass for coercions from α to β. It can be transitively chained with other Coe instances, and coercion is automatically used when x has type α but it is used in a context where β is expected. You can use the ↑x operator to explicitly trigger coercion.

Instance Constructor

`Coe.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription `((x : α) : β)`.

12.1. Coercion Insertion

The process of searching for a coercion from one type to another is called *coercion insertion*. Coercion insertion is attempted in the following situations where an error would otherwise occur:

- The expected type for a term is not equal to the type found for the term.
- A type or proposition is expected, but the term's type is not a universe.
- A term is applied as though it were a function, but its type is not a function type.

Coercions are also inserted when they are explicitly requested. Each situation in which coercions may be inserted has a corresponding prefix operator that triggers the appropriate insertion.

Because coercions are inserted automatically, nested type ascriptions provide a way to precisely control the types involved in a coercion. If α and β are not the same type, `((e : α) : β)` arranges for e to have type α and then inserts a coercion from α to β.

When a coercion is discovered, the instances used to find it are unfolded and removed from the resulting term. To the extent possible, calls to `Coe.coe` and related functions do not occur in the final term. This process of unfolding makes terms more readable. Even more importantly, it means that coercions can control the evaluation of the coerced terms by wrapping them in functions.

▼ *Example: Controlling Evaluation with Coercions*

The structure `Later` represents a term that can be evaluated in the future, by calling the contained function.

```
structure Later (α : Type u) where
  get : Unit → α
```

A coercion from any value to a later value is performed by creating a function that wraps it.

```
instance : CoeTail α (Later α) where
  coe x := { get := fun () => x }
```

However, if coercion insertion resulted in an application of `CoeTail.coe`, then this coercion would not have the desired effect at runtime, because the coerced value would be evaluated and then saved in the function's closure. Because coercion implementations are unfolded, this instance is nonetheless useful.

```
def tomorrow : Later String :=
  (Nat.fold 10000
    (init := "")
    (fun _ _ s => s ++ "tomorrow") : String)
```

Printing the resulting definition shows that the computation is inside the function's body:

```
#print tomorrow
```

```
def tomorrow : Later String :=
{ get := fun x => Nat.fold 10000 (fun x x s => s ++
"tomorrow") "" }
```

▼ *Example: Duplicate Evaluation in Coercions*

Because the contents of `Coe` instances are unfolded during coercion insertion, coercions that use their argument more than once should be careful to ensure that evaluation occurs just once. This can be done by using a helper function that is not part of the instance, or by using `let` to evaluate the coerced term and then re-use its resulting value.

The structure `Twice` requires that both fields have the same value:

```
structure Twice (α : Type u) where
  first : α
  second : α
  first_eq_second : first = second
```

One way to define a coercion from α to `Twice α` is with a helper function `twice`. The `coe` attribute marks it as a coercion so it can be shown correctly in proof goals and error messages.

```
@[coe]
def twice (x : α) : Twice α where
  first := x
  second := x
  first_eq_second := rfl

instance : Coe α (Twice α) := ⟨twice⟩
```

When the `Coe` instance is unfolded, the call to `twice` remains, which causes its argument to be evaluated before the body of the function is executed. As a result, the `dbg_trace` is included in the resulting term just once:

```
#eval ((dbg_trace "hello"; 5 : Nat) : Twice Nat)
```

This used to demonstrate the effect:

```
hello
```

Inlining the helper into the Coe instance results in a term that duplicates the dbg_trace:

```
instance : Coe α (Twice α) where
  coe x := ⟨x, x, rfl⟩

#eval ((dbg_trace "hello"; 5 : Nat) : Twice Nat)
```

```
hello
hello
```

Introducing an intermediate name for the result of the evaluation prevents the duplication of dbg_trace:

```
instance : Coe α (Twice α) where
  coe x := let y := x; ⟨y, y, rfl⟩

#eval ((dbg_trace "hello"; 5 : Nat) : Twice Nat)
```

```
hello
```

12.2. Coercing Between Types

Coercions between types are inserted when the Lean elaborator successfully constructs a term, inferring its type, in a context where a term of some other type was expected. Before signaling an error, the elaborator attempts to insert a coercion from the inferred type to the expected type by synthesizing an instance of CoeT. There are two ways that this might succeed:

1. There could be a chain of coercions from the inferred type to the expected type through a number of intermediate types. These chained coercions are selected based on the inferred type and the expected type, but not the term being coerced.
2. There could be a single dependent coercion from the inferred type to the expected type. Dependent coercions take the term being coerced into account as well as the inferred and expected types, but they cannot be chained.

The simplest way to define a non-dependent coercion is by implementing a Coe instance, which is enough to synthesize a CoeT instance. This instance participates in chaining, and may be applied any number of times. The expected type of the expression is used to drive synthesis of Coe instances, rather than the inferred type. For instances that can be used at most once, or instances in which the inferred type should drive synthesis, one of the other coercion classes may be needed.

▼ *Example: Defining Coercions*

The type Even represents the even natural numbers.

```
structure Even where
  number : Nat
  isEven : number % 2 = 0
```

A coercion allows even numbers to be used where natural numbers are expected. The **coe** attribute marks the projection as a coercion so that it can be shown accordingly in proof states and error messages, as described in the section on implementing coercions.

```
attribute [coe] Even.number

instance : Coe Even Nat where
  coe := Even.number
```

With this coercion in place, even numbers can be used where natural numbers are expected.

```
def four : Even := ⟨4, by omega⟩

#eval (four : Nat) + 1
```

```
5
```

Due to coercion chaining, there is also a coercion from `Even` to `Int` formed by chaining the `Coe Even Nat` instance with the existing coercion from `Nat` to `Int`:

```
#eval (four : Int) - 5
```

```
-1
```

Dependent coercions are needed when the specific term being coerced is required in order to determine whether or how to coerce the term: for example, only decidable propositions can be coerced to `Bool`, so the proposition in question must occur as part of the instance's type so that it can require the `Decidable` instance. Non-dependent coercions are used whenever all values of the inferred type can be coerced to the target type.

▼ *Example: Defining Dependent Coercions*

The string "four" can be coerced into the natural number 4 with this instance declaration:

```
instance : CoeDep String "four" Nat where
  coe := 4

#eval ("four" : Nat)
```

```
4
```

Ordinary type errors are produced for other strings:

```
#eval ("three" : Nat)
```

```
type mismatch
  "three"
has type
  String : Type
but is expected to have type
  Nat : Type
```

Non-dependent coercions may be chained: if there is a coercion from α to β and from β to γ, then there is also a coercion from α to γ. The chain should be in the form `CoeHead`?`CoeOut`*`Coe`*`CoeTail`?, which is to say it may consist of:

- An optional instance of `CoeHead` α α', followed by
- Zero or more instances of `CoeOut` α' ..., ..., `CoeOut` ... α'', followed by
- Zero or more instances of `Coe` α'' ..., ..., `Coe` ... β', followed by
- An optional instance of `CoeTail` β' γ

Most coercions can be implemented as instances of `Coe`. `CoeHead`, `CoeOut`, and `CoeTail` are needed in certain special situations.

`CoeHead` and `CoeOut` instances are chained from the inferred type towards the expected type. In other words, information in the type found for the term is used to resolve a chain of instances. `Coe` and `CoeTail` instances are chained from the expected type towards the inferred type, so information in the expected type is used to resolve a chain of instances. If these chains meet in the middle, a coercion has been found. This is reflected in their type signatures: `CoeHead` and `CoeOut` use semi-output parameters for the coercion's target, while `Coe` and `CoeTail` use semi-output parameters for the coercions' source.

When an instance provides a value for a semi-output parameter, the value is used during instance synthesis. However, if no value is provided, then a value may be assigned by the synthesis algorithm. Consequently, every semi-output parameter should be assigned a type when an instance is selected. This means that `CoeOut` should be used when the variables that occur in the coercion's output are a subset of those in its input, and `Coe` should be used when the variables in the input are a subset of those in the output.

▼ *Example: `CoeOut` vs `Coe` instances*

A `Truthy` value is a value paired with an indication of whether it should be considered to be true or false. A `Decision` is either `yes`, `no`, or `maybe`, with the latter containing further data for consideration.

```
structure Truthy (α : Type) where
  val : α
  isTrue : Bool

inductive Decision (α : Type) where
  | yes
```

```
  | maybe (val : α)
  | no
```

"Truthy" values can be converted to Bools by forgetting the contained value. Bools can be converted to Decisions by discounting the maybe case.

```
@[coe]
def Truthy.toBool : Truthy α → Bool :=
  Truthy.isTrue

@[coe]
def Decision.ofBool : Bool → Decision α
  | true => .yes
  | false => .no
```

Truthy.toBool must be a CoeOut instance, because the target of the coercion contains fewer unknown type variables than the source, while Decision.ofBool must be a Coe instance, because the source of the coercion contains fewer variables than the target:

```
instance : CoeOut (Truthy α) Bool := ⟨Truthy.isTrue⟩

instance : Coe Bool (Decision α) := ⟨Decision.ofBool⟩
```

With these instances, coercion chaining works:

```
#eval ({ val := 1, isTrue := true : Truthy Nat } : Decision
String)
```

```
Decision.yes
```

Attempting to use the wrong class leads to an error:

```
instance : Coe (Truthy α) Bool := ⟨Truthy.isTrue⟩
```

```
instance does not provide concrete values for (semi-)out-
params
  Coe (Truthy ?α) Bool
```

type class

```
CoeHead.{u, v} (α : Sort u) (β : semiOutParam (Sort v)) :
  Sort (max (max 1 u) v)
```

CoeHead α β is for coercions that are applied from left-to-right at most once at beginning of the coercion chain.

Instance Constructor

CoeHead.mk.{u, v}

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription `((x : α) : β)`.

type class

```
CoeOut.{u, v} (α : Sort u) (β : semiOutParam (Sort v)) :
  Sort (max (max 1 u) v)
```

`CoeOut α β` is for coercions that are applied from left-to-right.

Instance Constructor

`CoeOut.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription `((x : α) : β)`.

type class

```
CoeTail.{u, v} (α : semiOutParam (Sort u)) (β : Sort v) :
  Sort (max (max 1 u) v)
```

`CoeTail α β` is for coercions that can only appear at the end of a sequence of coercions. That is, α can be further coerced via `Coe σ α` and `CoeHead τ σ` instances but β will only be the expected type of the expression.

Instance Constructor

`CoeTail.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription `((x : α) : β)`.

Instances of CoeT can be synthesized when an appropriate chain of instances exists, or when there is a single applicable CoeDep instance. (i) When coercing from Nat to another type, a NatCast instances also suffices. If both exist, then the CoeDep instance takes priority.

type class

```
CoeT.{u, v} (α : Sort u) : α → (β : Sort v) → Sort (max 1 v)
```

CoeT is the core typeclass which is invoked by Lean to resolve a type error. It can also be triggered explicitly with the notation ↑x or by double type ascription `((x : α) : β)`.

A CoeT chain has the grammar `CoeHead? CoeOut* Coe* CoeTail? | CoeDep`.

Instance Constructor

CoeT.mk.{u, v}

Methods

`coe : β`

The resulting value of type β. The input `x : α` is a parameter to the type class, so the value of type β may possibly depend on additional typeclasses on x.

Dependent coercions may not be chained. As an alternative to a chain of coercions, a term e of type α can be coerced to β using an instance of `CoeDep α e β`. Dependent coercions are useful in situations where only some of the values can be coerced; this mechanism is used to coerce only decidable propositions to Bool. They are also useful when the value itself occurs in the coercion's target type.

type class

```
CoeDep.{u, v} (α : Sort u) : α → (β : Sort v) → Sort (max 1 v)
```

`CoeDep α (x : α) β` is a typeclass for dependent coercions, that is, the type β can depend on x (or rather, the value of x is available to typeclass search so an instance that relates β to x is allowed).

Dependent coercions do not participate in the transitive chaining process of regular coercions: they must exactly match the type mismatch on both sides.

Instance Constructor

CoeDep.mk.{u, v}

Methods

`coe : β`

The resulting value of type β. The input $x : \alpha$ is a parameter to the type class, so the value of type β may possibly depend on additional typeclasses on x.

▼ *Example: Dependent Coercion*

A type of non-empty lists can be defined as a pair of a list and a proof that it is not empty. This type can be coerced to ordinary lists by applying the projection:

```
structure NonEmptyList (α : Type u) : Type u where
  contents : List α
  non_empty : contents ≠ []

instance : Coe (NonEmptyList α) (List α) where
  coe xs := xs.contents
```

The coercion works as expected:

```
def oneTwoThree : NonEmptyList Nat := ⟨[1, 2, 3], by simp⟩

#eval (oneTwoThree : List Nat) ++ [4]
```

Arbitrary lists cannot, however, be coerced to non-empty lists, because some arbitrarily-chosen lists may indeed be empty:

```
instance : Coe (List α) (NonEmptyList α) where
  coe xs := ⟨xs, _⟩
```

```
don't know how to synthesize placeholder for argument
'non_empty'
context:
α : Type u_1
xs : List α
⊢ xs ≠ []
```

A dependent coercion can restrict the domain of the coercion to only lists that are not empty:

```
instance : CoeDep (List α) (x :: xs) (NonEmptyList α) where
  coe := ⟨x :: xs, by simp⟩

#eval ([1, 2, 3] : NonEmptyList Nat)
```

```
{ contents := [1, 2, 3], non_empty := _ }
```

Dependent coercion insertion requires that the term to be coerced syntactically matches the term in the instance header. Lists that are known to be non-empty, but which are not syntactically instances of `(· :: ·)`, cannot be coerced with this instance.

```
#check
  fun (xs : List Nat) =>
    let ys : List Nat := xs ++ [4]
```

```
(ys : NonEmptyList Nat)
```

When coercion insertion fails, the original type error is reported:

```
type mismatch
  ys
has type
  List Nat : Type
but is expected to have type
  NonEmptyList Nat : Type
```

syntax

Coercions

```
term ::= ...
    | ↑term
```

Coercions can be explicitly placed using the prefix operator ↑.

Unlike using nested type ascriptions, the ↑ syntax for placing coercions does not require the involved types to be written explicitly.

▼ *Example: Controlling Coercion Insertion*

Instance synthesis and coercion insertion interact with one another. Synthesizing an instance may make type information known that later triggers coercion insertion. The specific placement of coercions may matter.

In this definition of sub, the Sub Int instance is synthesized based on the function's return type. This instance requires that the two parameters also be Ints, but they are Nats. Coercions are inserted around each argument to the subtraction operator. This can be seen in the output of #print.

```
def sub (n k : Nat) : Int := n - k

#print sub
```

```
def sub : Nat → Nat → Int :=
fun n k => ↑n - ↑k
```

Placing the coercion operator outside the subtraction causes the elaborator to attempt to infer a type for the subtraction and then insert a coercion. Because the arguments are both Nats, the Sub Nat instance is selected, leading to the difference being a Nat. The difference is then coerced to an Int.

```
def sub' (n k : Nat) : Int := ↑ (n - k)
```

```
#print sub'
```

These two functions are not equivalent because subtraction of natural numbers truncates at zero:

```
#eval sub 4 8
```

```
-4
```

```
#eval sub' 4 8
```

```
0
```

12.2.1. Implementing Coercions

The appropriate CoeHead, CoeOut, Coe, or CoeTail instance is sufficient to cause a desired coercion to be inserted. However, the implementation of the coercion should be registered as a coercion using the **coe** attribute. This causes Lean to display uses of the coercion with the ↑ operator. It also causes the **norm_cast** tactic to treat the coercion as a cast, rather than as an ordinary function.

attribute

Coercion Declarations

```
attr ::= ...
    | coe
```

The `@[coe]` attribute on a function (which should also appear in a `instance : Coe A B := ⟨myFn⟩` declaration) allows the delaborator to show applications of this function as ↑ when printing expressions.

▼ *Example: Implementing Coercions*

The enum inductive type Weekday represents the days of the week:

```
inductive Weekday where
  | mo | tu | we | th | fr | sa | su
```

As a seven-element type, it contains the same information as Fin 7. There is a bijection:

```
def Weekday.toFin : Weekday → Fin 7
  | mo => 0
  | tu => 1
  | we => 2
  | th => 3
  | fr => 4
  | sa => 5
```

```
  | su => 6

def Weekday.fromFin : Fin 7 → Weekday
  | 0 => mo
  | 1 => tu
  | 2 => we
  | 3 => th
  | 4 => fr
  | 5 => sa
  | 6 => su
```

Each type can be coerced to the other:

```
instance : Coe Weekday (Fin 7) where
  coe := Weekday.toFin

instance : Coe (Fin 7) Weekday where
  coe := Weekday.fromFin
```

While this works, instances of the coercions that occur in Lean's output are not presented using the coercion operator, which is what Lean users expect. Instead, the name `Weekday.fromFin` is used explicitly:

```
def wednesday : Weekday := (2 : Fin 7)

#print wednesday
```

```
def wednesday : Weekday :=
Weekday.fromFin 2
```

Adding the **coe** attribute to the definition of a coercion causes it to be displayed using the coercion operator:

```
attribute [coe] Weekday.fromFin
attribute [coe] Weekday.toFin

def friday : Weekday := (5 : Fin 7)

#print friday
```

```
def friday : Weekday :=
↑5
```

12.2.2. Coercions from Natural Numbers and Integers

The type classes `NatCast` and `IntCast` are special cases of `Coe` that are used to define a coercion from `Nat` or `Int` to some other type that is in some sense canonical. They exist to enable better integration with large libraries of mathematics, such as Mathlib, that make heavy use of coercions to

map from the natural numbers or integers to other structures (typically rings). Ideally, the coercion of a natural number or integer into these structures is a simp normal form, because it is a convenient way to denote them.

When the coercion application is expected to be the simp normal form for a type, it is important that *all* such coercions are definitionally equal in practice. Otherwise, the simp normal form would need to choose a single chained coercion path, but lemmas could accidentally stated using a different path. Because `simp`'s internal index is based on the underlying structure of the term, rather than its presentation in the surface syntax, these differences would cause the lemmas to not be applied where expected. `NatCast` and `IntCast` instances, on the other hand, should be defined such that they are always definitionally equal, avoiding the problem. The Lean standard library's instances are arranged such that `NatCast` or `IntCast` instances are chosen preferentially over chains of coercion instances during coercion insertion. They can also be used as `CoeOut` instances, allowing a graceful fallback to coercion chaining when needed.

type class

```
NatCast.{u} (R : Type u) : Type u
```

The canonical homomorphism `Nat` → *R*. In most use cases, the target type will have a (semi)ring structure, and this homomorphism should be a (semi)ring homomorphism.

`NatCast` and `IntCast` exist to allow different libraries with their own types that can be notated as natural numbers to have consistent `simp` normal forms without needing to create coercion simplification sets that are aware of all combinations. Libraries should make it easy to work with `NatCast` where possible. For instance, in Mathlib there will be such a homomorphism (and thus a `NatCast` *R* instance) whenever *R* is an additive monoid with a `1`.

The prototypical example is `Int.ofNat`.

Instance Constructor

`NatCast.mk.{u}`

Methods

`natCast : Nat → R`

The canonical map `Nat` → *R*.

def

```
Nat.cast.{u} {R : Type u} [NatCast R] : Nat → R
```

The canonical homomorphism `Nat` → *R*. In most use cases, the target type will have a (semi)ring structure, and this homomorphism should be a (semi)ring homomorphism.

NatCast and IntCast exist to allow different libraries with their own types that can be notated as natural numbers to have consistent simp normal forms without needing to create coercion simplification sets that are aware of all combinations. Libraries should make it easy to work with NatCast where possible. For instance, in Mathlib there will be such a homomorphism (and thus a NatCast R instance) whenever R is an additive monoid with a 1.

The prototypical example is Int.ofNat.

type class

```
IntCast.{u} (R : Type u) : Type u
```

The canonical homomorphism Int → R. In most use cases, the target type will have a ring structure, and this homomorphism should be a ring homomorphism.

IntCast and NatCast exist to allow different libraries with their own types that can be notated as natural numbers to have consistent simp normal forms without needing to create coercion simplification sets that are aware of all combinations. Libraries should make it easy to work with IntCast where possible. For instance, in Mathlib there will be such a homomorphism (and thus an IntCast R instance) whenever R is an additive group with a 1.

Instance Constructor

IntCast.mk.{u}

Methods

```
intCast : Int → R
```

The canonical map Int → R.

def

```
Int.cast.{u} {R : Type u} [IntCast R] : Int → R
```

The canonical homomorphism Int → R. In most use cases, the target type will have a ring structure, and this homomorphism should be a ring homomorphism.

IntCast and NatCast exist to allow different libraries with their own types that can be notated as natural numbers to have consistent simp normal forms without needing to create coercion simplification sets that are aware of all combinations. Libraries should make it easy to work with IntCast where possible. For instance, in Mathlib there will be such a homomorphism (and thus an IntCast R instance) whenever R is an additive group with a 1.

12.3. Coercing to Sorts

The Lean elaborator expects types in certain positions without necessarily being able to determine the type's universe ahead of time. For example, the term following the colon in a definition header might be a proposition or a type. The ordinary coercion mechanism is not applicable because it requires a specific expected type, and there's no way to express that the expected type could be *any* universe in the Coe class.

When a term is elaborated in a position where a proposition or type is expected, but the inferred type of the elaborated term is not a proposition or type, Lean attempts to recover from the error by synthesizing an instance of CoeSort. If the instance is found, and the resulting type is itself a type, then it the coercion is inserted and unfolded.

Not every situation in which the elaborator expects a universe requires CoeSort. In some cases, a particular universe is available as an expected type. In these situations, ordinary coercion insertion using CoeT is used. Instances of CoeSort can be used to synthesize instances of CoeOut, so no separate instance is needed to support this use case. In general, coercions to types should be implemented as CoeSort.

type class

```
CoeSort.{u, v} (α : Sort u) (β : outParam (Sort v)) :
  Sort (max (max 1 u) v)
```

CoeSort α β is a coercion to a sort. β must be a universe, and this is triggered when α : α appears in a place where a type is expected, like (x : α) or α → a. CoeSort instances apply to CoeOut as well.

Instance Constructor

CoeSort.mk.{u, v}

Methods

coe : α → β

Coerces a value of type α to β, which must be a universe.

syntax

Explicit Coercion to Sorts

```
term ::= ...
    | ↥ term
```

Coercions to sorts can be explicitly triggered using the ↥ prefix operator.

▼ *Example: Sort Coercions*

A monoid is a type equipped with an associative binary operation and an identity element. While monoid structure can be defined as a type class, it can also be defined as a structure that "bundles up" the structure with the type:

```
structure Monoid where
  Carrier : Type u
  op : Carrier → Carrier → Carrier
  id : Carrier
  op_assoc :
    ∀ (x y z : Carrier), op x (op y z) = op (op x y) z
  id_op_identity : ∀ (x : Carrier), op id x = x
  op_id_identity : ∀ (x : Carrier), op x id = x
```

The type `Monoid` does not indicate the carrier:

```
def StringMonoid : Monoid where
  Carrier := String
  op := (· ++ ·)
  id := ""
  op_assoc := by intros; simp [String.append_assoc]
  id_op_identity := by intros; simp
  op_id_identity := by intros; simp
```

However, a `CoeSort` instance can be implemented that applies the `Monoid.Carrier` projection when a monoid is used in a position where Lean would expect a type:

```
instance : CoeSort Monoid (Type u) where
  coe m := m.Carrier

example : StringMonoid := "hello"
```

▼ *Example: Sort Coercions as Ordinary Coercions*

The inductive type `NatOrBool` represents the types `Nat` and `Bool`. They can be coerced to the actual types `Nat` and `Bool`:

```
inductive NatOrBool where
  | nat | bool

@[coe]
abbrev NatOrBool.asType : NatOrBool → Type
  | .nat => Nat
  | .bool => Bool

instance : CoeSort NatOrBool Type where
  coe := NatOrBool.asType
```

```
open NatOrBool
```

The `CoeSort` instance is used when `nat` occurs to the right of a colon:

```
def x : nat := 5
```

When an expected type is available, ordinary coercion insertion is used. In this case, the `CoeSort` instance is used to synthesize a `CoeOut NatOrBool Type` instance, which chains with the `Coe Type (Option Type)` instance to recover from the type error.

```
def y : Option Type := bool
```

12.4. Coercing to Function Types

Another situation where an expected type is not generally available is the function position in a function application term. Dependent function types are common; together with implicit parameters, they cause information to flow from the elaboration of one argument to the elaboration of the others. Attempting to deduce the type required for the function from the expected type of the entire application term and individually-inferred types of arguments will often fail. In these situations, Lean uses the `CoeFun` type class to coerce a non-function in an application position into a function. Like `CoeSort`, `CoeFun` instances do not chain with other coercions while inserting a function coercion, but they can be used as `CoeOut` instances during ordinary coercion insertion.

The second parameter to `CoeFun` is an output parameter that determines the resulting function type. This output parameter is function that computes the function type from the term that's being coerced, rather than the function type itself. Unlike `CoeDep`, the term itself is not taken into account during instance synthesis; it can, however, be used to create dependently typed coercions where the function type is determined by the term.

type class

```
CoeFun.{u, v} (α : Sort u) (γ : outParam (α → Sort v)) :
  Sort (max (max 1 u) v)
```

`CoeFun α (γ : α → Sort v)` is a coercion to a function. `γ α` should be a (coercion-to-)function type, and this is triggered whenever an element `f : α` appears in an application like `f x`, which would not make sense since `f` does not have a function type. `CoeFun` instances apply to `CoeOut` as well.

Instance Constructor

`CoeFun.mk.{u, v}`

Methods

```
coe : (f : α) → γ f
```

Coerces a value `f : α` to type `γ f`, which should be either be a function type or another CoeFun type, in order to resolve a mistyped application `f x`.

syntax

Explicit Coercion to Functions

```
term ::= ...
    | ↑ term
```

▼ *Example: Coercing Decorated Functions to Function Types*

The structure `NamedFun α β` pairs a function from `α` to `β` with a name.

```
structure NamedFun (α : Type u) (β : Type v) where
  function : α → β
  name : String
```

Existing functions can be named:

```
def succ : NamedFun Nat Nat where
  function n := n + 1
  name := "succ"

def asString [ToString α] : NamedFun α String where
  function := ToString.toString
  name := "asString"

def append : NamedFun (List α) (List α → List α) where
  function := (· ++ ·)
  name := "append"
```

Named functions can also be composed:

```
def NamedFun.comp
    (f : NamedFun β γ)
    (g : NamedFun α β) :
    NamedFun α γ where
  function := f.function ∘ g.function
  name := f.name ++ " ∘ " ++ g.name
```

Unlike ordinary functions, named functions have a reasonable representation as a string:

```
instance : ToString (NamedFun α α'') where
  toString f := s!"#<{f.name}>"

#eval asString.comp succ
```

```
#<asString ∘ succ>
```

A CoeFun instance allows them to be applied just like ordinary functions:

```
instance : CoeFun (NamedFun α α'') (fun _ => α → α'') where
  coe | ⟨f, _⟩ => f

#eval append [1, 2, 3] [4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

▼ *Example: Dependent Coercion to Functions*

Sometimes, the type of the resulting function depends on the specific value that is being coerced. A Writer represents a means of appending a representation of some value to a string:

```
structure Writer where
  Writes : Type u
  write : Writes → String → String

def natWriter : Writer where
  Writes := Nat
  write n out := out ++ toString n

def stringWriter : Writer where
  Writes := String
  write s out := out ++ s
```

Because the type of the parameter expected by the inner function depend on the Writer.Writes field, the CoeFun instance extracts the field:

```
instance :
    CoeFun Writer (·.Writes → String → String) where
  coe w := w.write
```

With this instance, concrete Writers can be used as functions:

```
#eval "" |> natWriter (5 : Nat) |> stringWriter " hello"
```

```
"5 hello"
```

▼ *Example: Coercing to Function Types*

A well-typed interpreter is an interpreter for a programming language that uses indexed families to rule out run-time type errors. Functions written in the interpreted language can be interpreted as Lean functions, but their underlying source code can also be inspected.

The first step in the well-typed interpreter is to select the subset of Lean types that can be used. These types are represented by an inductive type of codes `Ty` and a function that maps these codes to actual types.

```
inductive Ty where
  | nat
  | arr (dom cod : Ty)

abbrev Ty.interp : Ty → Type
  | .nat => Nat
  | .arr t t' => t.interp → t'.interp
```

The language itself is represented by an indexed family over variable contexts and result types. Variables are represented by de Bruijn indices.

```
inductive Tm : List Ty → Ty → Type where
  | zero : Tm Γ .nat
  | succ (n : Tm Γ .nat) : Tm Γ .nat
  | rep (n : Tm Γ .nat)
    (start : Tm Γ t)
    (f : Tm Γ (.arr .nat (.arr t t))) :
    Tm Γ t
  | lam (body : Tm (t :: Γ) t') : Tm Γ (.arr t t')
  | app (f : Tm Γ (.arr t t')) (arg : Tm Γ t) : Tm Γ t'
  | var (i : Fin Γ.length) : Tm Γ Γ[i]
deriving Repr
```

Because the `OfNat` instance for `Fin` requires that the upper bound be non-zero, `Tm.var` can be inconvenient to use with numeric literals. The helper `Tm.v` can be used to avoid the need for type annotations in these cases.

```
def Tm.v
    (i : Fin (Γ.length + 1)) :
    Tm (t :: Γ) (t :: Γ)[i] :=
  .var (Γ := t :: Γ) i
```

A function that adds two natural numbers uses the `rep` operation to apply the successor `Tm.succ` repeatedly.

```
def plus : Tm [] (.arr .nat (.arr .nat .nat)) :=
  .lam <| .lam <| .rep (.v 1) (.v 0) (.lam (.lam (.succ (.v 0))))
```

Each typing context can be interpreted as a type of run-time environments that provide a value for each variable in the context:

```
def Env : List Ty → Type
  | [] => Unit
  | t :: Γ => t.interp × Env Γ

def Env.empty : Env [] := ()
```

```
def Env.extend (ρ : Env Γ) (v : t.interp) : Env (t :: Γ) :=
  (v, ρ)

def Env.get (i : Fin Γ.length) (ρ : Env Γ) : Γ[i].interp :=
  match Γ, ρ, i with
  | _::_, (v, _), ⟨0, _⟩ => v
  | _::_, (_, ρ'), ⟨i+1, _⟩ => ρ'.get ⟨i, by simp_all⟩
```

Finally, the interpreter is a recursive function over the term:

```
def Tm.interp (ρ : Env α'') : Tm α'' t → t.interp
  | .zero => 0
  | .succ n => n.interp ρ + 1
  | .rep n start f =>
    let f' := f.interp ρ
    (n.interp ρ).fold (fun n _ x => f' n x) (start.interp ρ)
  | .lam body => fun x => body.interp (ρ.extend x)
  | .app f arg => f.interp ρ (arg.interp ρ)
  | .var i => ρ.get i
```

Coercing a `Tm` to a function consists of calling the interpreter.

```
instance : CoeFun (Tm [] α'') (fun _ => α''.interp) where
  coe f := f.interp .empty
```

Because functions are represented by a first-order inductive type, their code can be inspected:

```
#eval plus
```

```
Tm.lam (Tm.lam (Tm.rep (Tm.var 1) (Tm.var 0) (Tm.lam (Tm.lam
(Tm.succ (Tm.var 0))))))
```

At the same time, due to the coercion, they can be applied just like native Lean functions:

```
#eval plus 3 5
```

```
8
```

12.5. Implementation Details

Only ordinary coercion insertion uses chaining. Inserting coercions to a sort or a function uses ordinary instance synthesis. Similarly, dependent coercions are not chained.

12.5.1. Unfolding Coercions

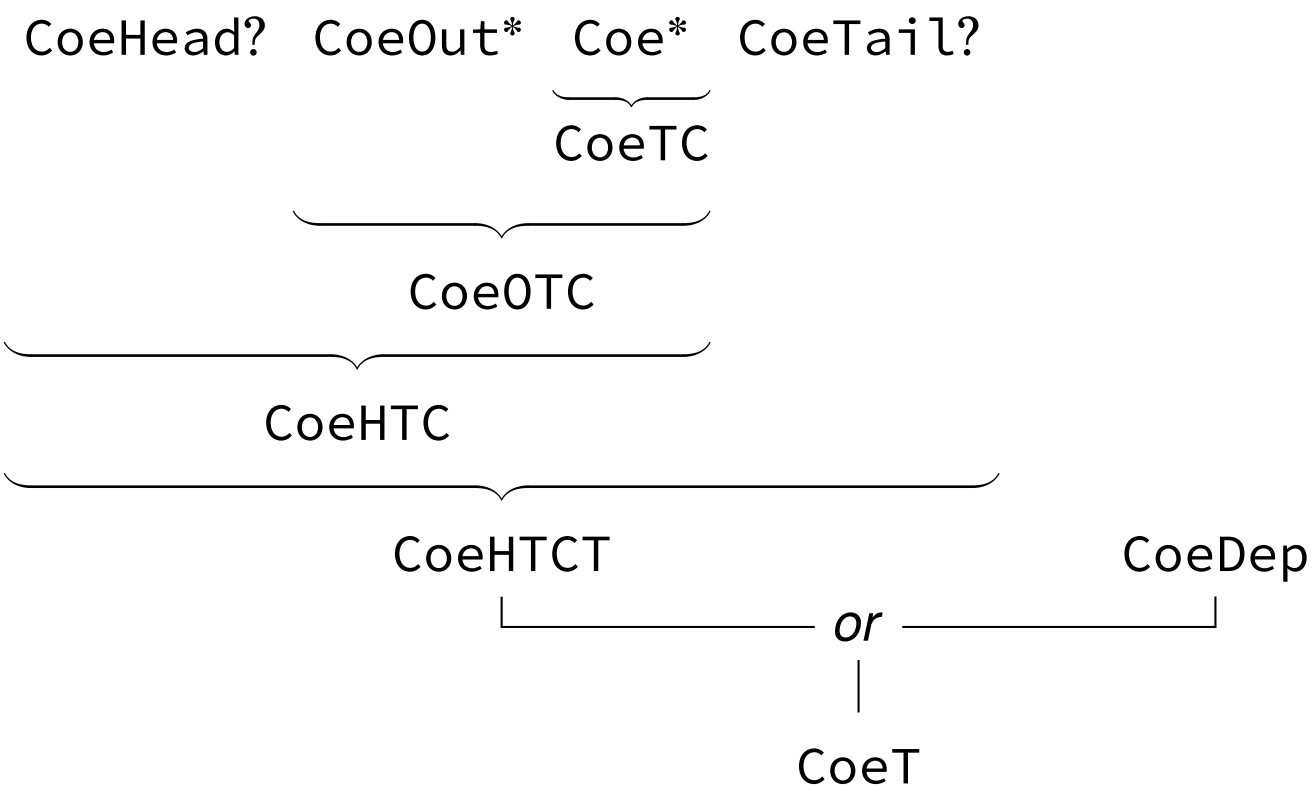
The coercion insertion mechanism unfolds applications of coercions, which allows them to control the specific shape of the resulting term. This is important both to ensure readable proof goals and to control evaluation of the coerced term in compiled code. Unfolding coercions is controlled by the

`coe_decl` attribute, which is applied to each coercion method (e.g. `Coe.coe`). This attribute should be considered part of the internals of the coercion mechanism, rather than part of the public coercion API.

12.5.2. Coercion Chaining

Coercion chaining is implemented through a collection of auxiliary type classes. Users should not write instances of these classes directly, but knowledge of their structure can be useful when diagnosing the reason why a coercion was not inserted as expected. The specific rules governing the ordering of instances in the chain (namely, that it should match `CoeHead`? `CoeOut`*`Coe`*`CoeTail`?) are implemented by the following type classes:

- `CoeTC` is the transitive closure of `Coe` instances.
- `CoeOTC` is the middle of the chain, consisting of the transitive closure of `CoeOut` instances followed by `CoeTC`.
- `CoeHTC` is the start of the chain, consisting of at most one `CoeHead` instance followed by `CoeOTC`.
- `CoeHTCT` is the whole chain, consisting of CoeHTC followed by at most one `CoeTail` instance. Alternatively, it might be a `NatCast` instance.
- `CoeT` represents the entire chain: it is either a `CoeHTCT` chain or a single `CoeDep` instance.



Auxiliary Classes for Coercions

type class

```
CoeHTCT.{u, v} (α : Sort u) (β : Sort v) : Sort (max (max 1 u) v)
```

Auxiliary class implementing `CoeHead* Coe* CoeTail?`. Users should generally not implement this directly.

Instance Constructor

`CoeHTCT.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription $((x : \alpha) : \beta)$.

type class

`CoeHTC.{u, v} (α : Sort u) (β : Sort v) : Sort (max (max 1 u) v)`

Auxiliary class implementing `CoeHead CoeOut* Coe*`. Users should generally not implement this directly.

Instance Constructor

`CoeHTC.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription $((x : \alpha) : \beta)$.

type class

`CoeOTC.{u, v} (α : Sort u) (β : Sort v) : Sort (max (max 1 u) v)`

Auxiliary class implementing `CoeOut* Coe*`. Users should generally not implement this directly.

Instance Constructor

`CoeOTC.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription `((x : α) : β)`.

type class

`CoeTC.{u, v} (α : Sort u) (β : Sort v) : Sort (max (max 1 u) v)`

Auxiliary class implementing `Coe*`. Users should generally not implement this directly.

Instance Constructor

`CoeTC.mk.{u, v}`

Methods

`coe : α → β`

Coerces a value of type α to type β. Accessible by the notation ↑x, or by double type ascription `((x : α) : β)`.

13. Tactic Proofs

The tactic language is a special-purpose programming language for constructing proofs. In Lean, propositions are represented by types, and proofs are terms that inhabit these types. The section on propositions describes propositions in more detail. While terms are designed to make it convenient to indicate a specific inhabitant of a type, tactics are designed to make it convenient to demonstrate that a type is inhabited. This distinction exists because it's important that definitions pick out the precise objects of interest and that programs return the intended results, but proof irrelevance means that there's no *technical* reason to prefer one proof term over another. For example, given two assumptions of a given type, a program must be carefully written to use the correct one, while a proof may use either without consequence.

Tactics are imperative programs that modify a *proof state*. A proof state consists of an ordered sequence of *goals*, which are contexts of local assumptions together with types to be inhabited; a tactic may either *succeed* with a possibly-empty sequence of further goals (called *subgoals*) or *fail* if it cannot make progress. If a tactic succeeds with no subgoals, then the proof is complete. If it succeeds with one or more subgoals, then its goal or goals will be proved when those subgoals have been proved. The first goal in the proof state is called the *main goal*. While most tactics affect only the main goal, operators such as `<;>` and `all_goals` can be used to apply a tactic to many goals, and operators such as bullets, `next` or `case` can narrow the focus of subsequent tactics to only a single goal in the proof state.

Behind the scenes, tactics construct proof terms. Proof terms are independently checkable evidence of a theorem's truth, written in Lean's type theory. Each proof is checked in the kernel, and can be verified with independently-implemented external checkers, so the worst outcome from a bug in a

tactic is a confusing error message, rather than an incorrect proof. Each goal in a tactic proof corresponds to an incomplete portion of a proof term.

13.1. Running Tactics

syntax

Tactic Proofs with `by`

Tactics are included in terms using `by`, which is followed by a sequence of tactics in which each has the same indentation:

```
term ::= ...
    | by
      tacticSeq
```

Alternatively, explicit braces and semicolons may be used:

```
term ::= ...
    | by { tactic* }
```

Tactics are invoked using the `by` term. When the elaborator encounters `by`, it invokes the tactic interpreter to construct the resulting term. Tactic proofs may be embedded via `by` in any context in which a term can occur.

13.2. Reading Proof States

The goals in a proof state are displayed in order, with the main goal on top. Goals may be either named or anonymous. Named goals are indicated with `case` at the top (called a *case label*), while anonymous goals have no such indicator. Tactics assign goal names, typically on the basis of constructor names, parameter names, structure field names, or the nature of the reasoning step implemented by the tactic.

▼ *Example: Named goals*

This proof state contains four goals, all of which are named. This is part of a proof that the `Monad Option` instance is lawful (that is, to provide the `LawfulMonad Option` instance), and the case names (highlighted below) come from the names of the fields of `LawfulMonad`.

```
  case bind_pure_comp
α : Type ?u.2428
β : Type ?u.2428
f : α → β
x : Option α
⊢ (do let a ← x pure (f a)) = f <$> x
```

```
  case bind_map
α : Type ?u.2428
β : Type ?u.2428
f : Option (α → β)
x : Option α
⊢ (do let x_1 ← f x_1 <$> x) = f <*> x

  case pure_bind
α : Type ?u.2428
β : Type ?u.2428
x : α
f : α → Option β
⊢ pure x >>= f = f x

  case bind_assoc
α : Type ?u.2428
β : Type ?u.2428
γ : Type ?u.2428
x : Option α
f : α → Option β
g : β → Option γ
⊢ x >>= f >>= g = x >>= fun x => f x >>= g
```

▼ *Example: Anonymous Goals*

This proof state contains a single anonymous goal.

```
n : Nat
k : Nat
⊢ n + k = k + n
```

The **`case`** and **`case'`** tactics can be used to select a new main goal using the desired goal's name. When names are assigned in the context of a goal which itself has a name, the new goals' names are appended to the main goal's name with a dot (`'.'`, `Unicode FULL STOP (0x2e)`) between them.

▼ *Example: Hierarchical Goal Names*

In the course of an attempt to prove `∀ (n k : Nat), n + k = k + n`, this proof state can occur:

```
  case zero
k : Nat
⊢ 0 + k = k + 0
```

```
  case succ
  k : Nat
n✝ : Nat
a✝ : n✝ + k = k + n✝
⊢ n✝ + 1 + k = k + (n✝ + 1)
```

After `induction k`, the two new cases' names have `zero` as a prefix, because they were created in a goal named `zero`:

```
  case zero.zero
⊢ 0 + 0 = 0 + 0
```

```
  case zero.succ
n✝ : Nat
a✝ : 0 + n✝ = n✝ + 0
⊢ 0 + (n✝ + 1) = n✝ + 1 + 0
```

```
  case succ
  k : Nat
n✝ : Nat
a✝ : n✝ + k = k + n✝
⊢ n✝ + 1 + k = k + (n✝ + 1)
```

Each goal consists of a sequence of assumptions and a desired conclusion. Each assumption has a name and a type; the conclusion is a type. Assumptions are either arbitrary elements of some type or statements that are presumed true.

▼ *Example: Assumption Names and Conclusion*

This goal has four assumptions:

```
  case cons
  α : Type ?u.4498
  x : α
xs : List α
ih : xs ++ [] = xs
⊢ x :: xs ++ [] = x :: xs
```

They are:

- α, an arbitrary type
- x, an arbitrary α
- xs, an arbitrary `List` α

- `ih`, an induction hypothesis that asserts that appending the empty list to `xs` is equal to `xs`.

The conclusion is the statement that prepending x to both sides of the equality in the induction hypothesis results in equal lists.

Some assumptions are *inaccessible*, which means that they cannot be referred to explicitly by name. Inaccessible assumptions occur when an assumption is created without a specified name or when the assumption's name is shadowed by a later assumption. Inaccessible assumptions should be regarded as anonymous; they are presented as if they had names because they may be referred to in later assumptions or in the conclusion, and displaying a name allows these references to be distinguished from one another. In particular, inaccessible assumptions are presented with daggers (†) after their names.

▼ *Example: Accessible Assumption Names*

In this proof state, all assumptions are accessible.

```
  case bind_pure_comp
α : Type ?u.5444
β : Type ?u.5444
f : α → β
x : Option α
⊢ (do let a ← x pure (f a)) = f <$> x
```

▼ *Example: Inaccessible Assumption Names*

In this proof state, only the first and third assumptions are accessible. The second and fourth are inaccessible, and their names include a dagger to indicate that they cannot be referenced.

```
  case bind_pure_comp
  α : Type ?u.5787
β† : Type ?u.5787
  f : α → β†
x† : Option α
⊢ (do let a ← x† pure (f a)) = f <$> x†
```

Inaccessible assumptions can still be used. Tactics such as **`assumption`** or **`simp`** can scan the entire list of assumptions, finding one that is useful, and **`contradiction`** can eliminate the current goal by finding an impossible assumption without naming it. Other tactics, such as **`rename_i`** and

`next`, can be used to name inaccessible assumptions, making them accessible. Additionally, assumptions can be referred to by their type, by writing the type in single guillemets.

syntax

Assumptions by Type

Single guillemets around a term represent a reference to some term in scope with that type.

```
term ::= ...
    | ‹term›
```

This can be used to refer to local lemmas by their theorem statement rather than by name, or to refer to assumptions regardless of whether they have explicit names.

▼ *Example: Assumptions by Type*

In the following proof, `cases` is repeatedly used to analyze a number. At the beginning of the proof, the number is named `x`, but `cases` generates an inaccessible name for subsequent numbers. Rather than providing names, the proof takes advantage of the fact that there is a single assumption of type `Nat` at any given time and uses `‹Nat›` to refer to it. After the iteration, there is an assumption that `n + 3 < 3`, which `contradiction` can use to remove the goal from consideration.

```
example : x < 3 → x ∈ [0, 1, 2] := by
  intros
  iterate 3
    cases ‹Nat›
    . decide
  contradiction
```

▼ *Example: Assumptions by Type, Outside Proofs*

Single-guillemet syntax also works outside of proofs:

```
#eval
  let x := 1
  let y := 2
  ‹Nat›
```

```
2
```

This is generally not a good idea for non-propositions, however—when it matters *which* element of a type is selected, it's better to select it explicitly.

13.2.1. Hiding Proofs and Large Terms

Terms in proof states can be quite big, and there may be many assumptions. Because of definitional proof irrelevance, proof terms typically give little useful information. By default, they are not shown in goals in proof states unless they are *atomic*, meaning that they contain no subterms. Hiding proofs is controlled by two options: `pp.proofs` turns the feature on and off, while `pp.proofs.threshold` determines a size threshold for proof hiding.

▼ *Example: Hiding Proof Terms*

In this proof state, the proof that `0 < n` is hidden.

```
 n : Nat
 i : Fin n
gt : ↑i > 5
⊢ ⟨0, ⋯⟩ ≤ i
```

option

`pp.proofs`

Default value: `false`

(pretty printer) display proofs when true, and replace proofs appearing within expressions by ⋯ when false

option

`pp.proofs.threshold`

Default value: `0`

(pretty printer) when `pp.proofs` is false, controls the complexity of proofs at which they begin being replaced with ⋯

Additionally, non-proof terms may be hidden when they are too large. In particular, Lean will hide terms that are below a configurable depth threshold, and it will hide the remainder of a term once a certain amount in total has been printed. Showing deep terms can enabled or disabled with the option `pp.deepTerms`, and the depth threshold can be configured with the option `pp.deepTerms.threshold`. The maximum number of pretty printer steps can be configured with the option `pp.maxSteps`. Printing very large terms can lead to slowdowns or even stack overflows in tooling; please be conservative when adjusting these options' values.

option

`pp.deepTerms`

Default value: `false`

(pretty printer) display deeply nested terms, replacing them with ⋯ if set to false

option

`pp.deepTerms.threshold`

Default value: `50`

(pretty printer) when `pp.deepTerms` is false, the depth at which terms start being replaced with ⋯

option

`pp.maxSteps`

Default value: `5000`

(pretty printer) maximum number of expressions to visit, after which terms will pretty print as …

13.2.2. Metavariables

Terms that begin with a question mark are *metavariables* that correspond to an unknown value. They may stand for either universe levels or for terms. Some metavariables arise as part of Lean's elaboration process, when not enough information is yet available to determine a value. These metavariables' names have a numeric component at the end, such as `?m.392` or `?u.498`. Other metavariables come into existence as a result of tactics or synthetic holes. These metavariables' names do not have a numeric component. Metavariables that result from tactics frequently appear as goals whose case labels match the name of the metavariable.

▼ *Example: Universe Level Metavariables*

In this proof state, the universe level of α is unknown:

```
   α : Type ?u.9759
   x : α
  xs : List α
elem : x ∈ xs
⊢ xs.length > 0
```

▼ *Example: Type Metavariables*

In this proof state, the type of list elements is unknown. The metavariable is repeated because the unknown type must be the same in both positions.

```
   x : ?m.10119
  xs : List ?m.10119
elem : x ∈ xs
⊢ xs.length > 0
```

▼ *Example: Metavariables in Proofs*

In this proof state,

```
 i : Nat
 j : Nat
 k : Nat
h1 : i ≤ j
h2 : j ≤ k
⊢ i ≤ k
```

applying the tactic `apply Nat.lt_trans` results in the following proof state, in which the middle value of the transitivity step `?m` is unknown:

```
  case h₁
  i : Nat
  j : Nat
  k : Nat
 h1 : i ≤ j
 h2 : j ≤ k
 ⊢ i ≤ ?m

  case a
  i : Nat
  j : Nat
  k : Nat
 h1 : i ≤ j
 h2 : j ≤ k
 ⊢ ?m ≤ k

  case m
  i : Nat
  j : Nat
  k : Nat
 h1 : i ≤ j
```

```
h2 : j ≤ k
⊢ Nat
```

▼ *Example: Explicitly-Created Metavariables*

Explicit named holes are represented by metavariables, and additionally give rise to proof goals. In this proof state,

```
i : Nat
j : Nat
k : Nat
h1 : i ≤ j
h2 : j ≤ k
⊢ i ≤ k
```

applying the tactic `apply @Nat.lt_trans i ?middle k ?p1 ?p2` results in the following proof state, in which the middle value of the transitivity step `?middle` is unknown and goals have been created for each of the named holes in the term:

```
case middle
i : Nat
j : Nat
k : Nat
h1 : i ≤ j
h2 : j ≤ k
⊢ Nat

case p1
i : Nat
j : Nat
k : Nat
h1 : i ≤ j
h2 : j ≤ k
⊢ i ≤ ?middle

case p2
i : Nat
j : Nat
k : Nat
h1 : i ≤ j
h2 : j ≤ k
⊢ ?middle ≤ k
```

The display of metavariable numbers can be disabled using the `pp.mvars`. This can be useful when using features such as `#guard_msgs` that match Lean's output against a desired string, which is very useful when writing tests for custom tactics.

option

`pp.mvars`

Default value: `true`

(pretty printer) display names of metavariables when true, and otherwise display them as '?' *(for expression metavariables) and as* '' (for universe level metavariables)

13.3. The Tactic Language

A tactic script consists of a sequence of tactics, separated either by semicolons or newlines. When separated by newlines, tactics must be indented to the same level. Explicit curly braces and semicolons may be used instead of indentation. Tactic sequences may be grouped by parentheses. This allows a sequence of tactics to be used in a position where a single tactic would otherwise be grammatically expected.

Generally, execution proceeds from top to bottom, with each tactic running in the proof state left behind by the prior tactic. The tactic language contains a number of control structures that can modify this flow.

Each tactic is a syntax extension in the `tactic` category. This means that tactics are free to define their own concrete syntax and parsing rules. However, with a few exceptions, the majority of tactics can be identified by a leading keyword; the exceptions are typically frequently-used built-in control structures such as `<;>`.

13.3.1. Control Structures

Strictly speaking, there is no fundamental distinction between control structures and other tactics. Any tactic is free to take others as arguments and arrange for their execution in any context that it sees fit. Even if a distinction is arbitrary, however, it can still be useful. The tactics in this section are those that resemble traditional control structures from programming, or those that *only* recombine other tactics rather than making progress themselves.

13.3.1.1. Success and Failure

When run in a proof state, every tactic either succeeds or fails. Tactic failure is akin to exceptions: failures typically "bubble up" until handled. Unlike exceptions, there is no operator to distinguish between reasons for failure; **`first`** simply takes the first branch that succeeds.

tactic

`fail`

`fail msg` is a tactic that always fails, and produces an error using the given message.

tactic

fail_if_success

`fail_if_success t` fails if the tactic `t` succeeds.

tactic

try

`try` `tac` runs `tac` and succeeds even if `tac` failed.

tactic

first

`first` | `tac` | `...` runs each `tac` until one succeeds, or else fails.

13.3.1.2. Branching

Tactic proofs may use pattern matching and conditionals. However, their meaning is not quite the same as it is in terms. While terms are expected to be executed once the values of their variables are known, proofs are executed with their variables left abstract and should consider *all* cases simultaneously. Thus, when **`if`** and **`match`** are used in tactics, their meaning is reasoning by cases rather than selection of a concrete branch. All of their branches are executed, and the condition or pattern match is used to refine the main goal with more information in each branch, rather than to select a single branch.

tactic

if ... then ... else ...

In tactic mode, **`if`** *`t`* **`then`** *`tac1`* **`else`** *`tac2`* is alternative syntax for:

```
by_cases t
· tac1
· tac2
```

It performs case distinction on `h† : t` or `h† : ¬t`, where `h†` is an anonymous hypothesis, and `tac1` and `tac2` are the subproofs. (It doesn't actually use nondependent **`if`**, since this wouldn't add anything to the context and hence would be useless for proving theorems. To actually insert an `ite` application use **`refine if t then ?_ else ?_`**.)

tactic

if h : ... then ... else ...

In tactic mode, `if h : t then tac1 else tac2` can be used as alternative syntax for:

```
by_cases h : t
· tac1
· tac2
```

It performs case distinction on `h : t` or `h : ¬t` and `tac1` and `tac2` are the subproofs.

You can use `?_` or `_` for either subproof to delay the goal to after the tactic, but if a tactic sequence is provided for `tac1` or `tac2` then it will require the goal to be closed by the end of the block.

▼ *Example: Reasoning by cases with `if`*

In each branch of the `if`, an assumption is added that reflects whether `n = 0`.

```
example (n : Nat) : if n = 0 then n < 1 else n > 0 := by
  if n = 0 then
    simp [*]
  else
    simp only [↓reduceIte, gt_iff_lt, *]
    omega
```

tactic

`match`

`match` performs case analysis on one or more expressions. See Induction and Recursion. The syntax for the **`match`** tactic is the same as term-mode **`match`**, except that the match arms are tactics instead of expressions.

```
example (n : Nat) : n = n := by
  match n with
  | 0 => rfl
  | i+1 => simp
```

When pattern matching, instances of the scrutinee in the goal are replaced with the patterns that match them in each branch. Each branch must then prove the refined goal. Compared to the `cases` tactic, using `match` can allow a greater degree of flexibility in the cases analysis being performed, but the requirement that each branch solve its goal completely makes it more difficult to incorporate into larger automation scripts.

▼ *Example: Reasoning by cases with `match`*

In each branch of the `match`, the scrutinee `n` has been replaced by either `0` or `k + 1`.

```
example (n : Nat) : if n = 0 then n < 1 else n > 0 := by
  match n with
  | 0 =>
    simp
  | k + 1 =>
    simp
```

13.3.1.3. Goal Selection

Most tactics affect the main goal. Goal selection tactics provide a way to treat a different goal as the main one, rearranging the sequence of goals in the proof state.

tactic

case

- **case** `tag => tac` focuses on the goal with case name `tag` and solves it using `tac`, or else fails.
- **case** `tag` x_1 `...` x_n `=> tac` additionally renames the n most recent hypotheses with inaccessible names to the given names.
- **case** tag_1 `|` tag_2 `=> tac` is equivalent to (**case** tag_1 `=> tac`); (**case** tag_2 `=> tac`).

tactic

case'

case' is similar to the **case** `tag => tac` tactic, but does not ensure the goal has been solved after applying `tac`, nor admits the goal if `tac` failed. Recall that **case** closes the goal using **sorry** when `tac` fails, and the tactic execution is not interrupted.

tactic

rotate_left

`rotate_left n` rotates goals to the left by n. That is, **rotate_left** `1` takes the main goal and puts it to the back of the subgoal list. If n is omitted, it defaults to `1`.

tactic

rotate_right

Rotate the goals to the right by n. That is, take the goal at the back and push it to the front n times. If n is omitted, it defaults to `1`.

13.3.1.3.1. Sequencing

In addition to running tactics one after the other, each being used to solve the main goal, the tactic language supports sequencing tactics according to the way in which goals are produced. The `<;>` tactic combinator allows a tactic to be applied to *every* subgoal produced by some other tactic. If no new goals are produced, then the second tactic is not run.

tactic

`<;>`

`tac <;> tac'` runs `tac` on the main goal and `tac'` on each produced goal, concatenating all goals produced by `tac'`.

If the tactic fails on any of the subgoals, then the whole `<;>` tactic fails.

▼ *Example: Subgoal Sequencing*

In a this proof state:

```
x : Nat
h : x = 1 ∨ x = 2
⊢ x ≤ 3
```

the tactic `cases h` yields the following two goals:

```
case inl
x : Nat
h✝ : x = 1
⊢ x ≤ 3

case inr
x : Nat
h✝ : x = 2
⊢ x ≤ 3
```

Running `cases h ; simp [*]` causes `simp` to solve the first goal, leaving the second behind:

```
case inr
x : Nat
h✝ : x = 2
⊢ x ≤ 3
```

Replacing the `;` with `<;>` and running `cases h <;> simp [*]` solves **both** of the new goals with `simp`:

All goals completed! 🐙

13.3.1.3.2. Working on Multiple Goals

The tactics `all_goals` and `any_goals` allow a tactic to be applied to every goal in the proof state. The difference between them is that if the tactic fails for in any of the goals, `all_goals` itself fails, while `any_goals` fails only if the tactic fails in all of the goals.

tactic

`all_goals`

`all_goals tac` runs `tac` on each goal, concatenating the resulting goals. If the tactic fails on any goal, the entire *`all_goals`* tactic fails.

See also `any_goals tac`.

tactic

`any_goals`

`any_goals tac` applies the tactic `tac` to every goal, concatenating the resulting goals for successful tactic applications. If the tactic fails on all of the goals, the entire *`any_goals`* tactic fails.

This tactic is like `all_goals try tac` except that it fails if none of the applications of `tac` succeeds.

13.3.1.4. Focusing

Focusing tactics remove some subset of the proof goals (typically leaving only the main goal) from the consideration of some further tactics. In addition to the tactics described here, the `case` and `case'` tactics focus on the selected goal.

tactic

`·`

- `tac` focuses on the main goal and tries to solve it using `tac`, or else fails.

It is generally considered good Lean style to use bullets whenever a tactic line results in more than one new subgoal. This makes it easier to read and maintain proofs, because the connections between steps of reasoning are more clear and any change in the number of subgoals while editing the proof will have a localized effect.

tactic

`next`

`next` `=> tac` focuses on the next goal and solves it using `tac`, or else fails. **`next`** x_1 `...` x_n `=> tac` additionally renames the n most recent hypotheses with inaccessible names to the given names.

tactic

`focus`

`focus` *tac* focuses on the main goal, suppressing all other goals, and runs *tac* on it. Usually `· tac`, which enforces that the goal is closed by *tac*, should be preferred.

13.3.1.5. Repetition and Iteration

tactic

`iterate`

`iterate n tac` runs `tac` exactly n times. `iterate tac` runs `tac` repeatedly until failure.

`iterate`'s argument is a tactic sequence, so multiple tactics can be run using **`iterate`** `n` (tac_1`; `tac_2`; ⋯`) or

```
iterate n
  tac₁
  tac₂
  …
```

tactic

`repeat`

`repeat` `tac` repeatedly applies `tac` so long as it succeeds. The tactic `tac` may be a tactic sequence, and if `tac` fails at any point in its execution, **`repeat`** will revert any partial changes that `tac` made to the tactic state.

The tactic `tac` should eventually fail, otherwise **`repeat`** `tac` will run indefinitely.

See also:

- **`try`** `tac` is like **`repeat`** `tac` but will apply `tac` at most once.
- `repeat' tac` recursively applies `tac` to each goal.
- **`first`** `| tac1 | tac2` implements the backtracking used by **`repeat`**

tactic

`repeat'`

`repeat' tac` recursively applies `tac` on all of the goals so long as it succeeds. That is to say, if `tac` produces multiple subgoals, then `repeat' tac` is applied to each of them.

See also:

- **`repeat`** `tac` simply repeatedly applies `tac`.
- `repeat1' tac` is `repeat' tac` but requires that `tac` succeed for some goal at least once.

tactic

`repeat1'`

`repeat1' tac` recursively applies to `tac` on all of the goals so long as it succeeds, but `repeat1' tac` fails if `tac` succeeds on none of the initial goals.

See also:

- **`repeat`** `tac` simply applies `tac` repeatedly.
- `repeat' tac` is like `repeat1' tac` but it does not require that `tac` succeed at least once.

13.3.2. Names and Hygiene

Behind the scenes, tactics generate proof terms. These proof terms exist in a local context, because assumptions in proof states correspond to local binders in terms. Uses of assumptions correspond to variable references. It is very important that the naming of assumptions be predictable; otherwise, small changes to the internal implementation of a tactic could either lead to variable capture or to a broken reference if they cause different names to be selected.

Lean's tactic language is *hygienic*. This means that the tactic language respects lexical scope: names that occur in a tactic refer to the enclosing binding in the source code, rather than being determined by the generated code, and the tactic framework is responsible for maintaining this property. Variable references in tactic scripts refer either to names that were in scope at the beginning of the script or to bindings that were explicitly introduced as part of the tactics, rather than to the names chosen for use in the proof term behind the scenes.

A consequence of hygienic tactics is that the only way to refer to an assumption is to explicitly name it. Tactics cannot assign assumption names themselves, but must rather accept names from users; users are correspondingly obligated to provide names for assumptions that they wish to refer to. When an assumption does not have a user-provided name, it is shown in the proof state with a dagger (`'†', DAGGER 0x2020`). The dagger indicates that the name is *inaccessible* and cannot be explicitly referred to.

Hygiene can be disabled by setting the option `tactic.hygienic` to `false`. This is not recommended, as many tactics rely on the hygiene system to prevent capture and thus do not incur the overhead of careful manual name selection.

option

`tactic.hygienic`

Default value: `true`

make sure tactics are hygienic

▼ *Example: Tactic hygiene: inaccessible assumptions*

When proving that `∀ (n : Nat), 0 + n = n`, the initial proof state is:

```
⊢ ∀ (n : Nat), 0 + n = n
```

The tactic `intro` results in a proof state with an inaccessible assumption:

```
n✝ : Nat
⊢ 0 + n✝ = n✝
```

▼ *Example: Tactic hygiene: accessible assumptions*

When proving that `∀ (n : Nat), 0 + n = n`, the initial proof state is:

```
⊢ ∀ (n : Nat), 0 + n = n
```

The tactic `intro n`, with the explicit name n, results in a proof state with an accessibly-named assumption:

```
n : Nat
⊢ 0 + n = n
```

13.3.2.1. Accessing Assumptions

Many tactics provide a means of specifying names for the assumptions that they introduce. For example, `intro` and `intros` take assumption names as arguments, and `induction`'s `with`-form allows simultaneous case selection, assumption naming, and focusing. When an assumption does not have a name, one can be assigned using `next`, `case`, or `rename_i`.

tactic

`rename_i`

`rename_i x_1 ... x_n` renames the last n inaccessible names using the given names.

13.3.3. Assumption Management

Larger proofs can benefit from management of proof states, removing irrelevant assumptions and making their names easier to understand. Along with these operators, `rename_i` allows inaccessible assumptions to be renamed, and `intro`, `intros` and `rintro` convert goals that are implications or universal quantification into goals with additional assumptions.

tactic

`rename`

`rename t => x` renames the most recent hypothesis whose type matches `t` (which may contain placeholders) to `x`, or fails if no such hypothesis could be found.

tactic

`revert`

`revert` `x...` is the inverse of **`intro`** `x...`: it moves the given hypotheses into the main goal's target type.

tactic

`clear`

`clear` `x...` removes the given hypotheses, or fails if there are remaining references to a hypothesis.

13.3.4. Local Definitions and Proofs

`have` and `let` both create local assumptions. Generally speaking, **`have`** should be used when proving an intermediate lemma; `let` should be reserved for local definitions.

tactic

`have`

The **`have`** tactic is for adding hypotheses to the local context of the main goal.

- **`have`** `h : t :=` e adds the hypothesis *`h : t`* if e is a term of type *`t`*.
- **`have`** `h :=` e uses the type of e for *`t`*.
- **`have : t :=`** e and **`have :=`** e use `this` for the name of the hypothesis.

- **have** pat := e for a pattern pat is equivalent to **match** e **with** | pat => _, where _ stands for the tactics that follow this one. It is convenient for types that have only one applicable constructor. For example, given h : p ∧ q ∧ r, **have** ⟨h₁, h₂, h₃⟩ := h produces the hypotheses $h_1 : p$, $h_2 : q$, and $h_3 : r$.

tactic

have

The **have** tactic is for adding hypotheses to the local context of the main goal.

- **have** h : t := e adds the hypothesis $h : t$ if e is a term of type t.
- **have** h := e uses the type of e for t.
- **have** : t := e and **have** := e use this for the name of the hypothesis.
- **have** pat := e for a pattern pat is equivalent to **match** e **with** | pat => _, where _ stands for the tactics that follow this one. It is convenient for types that have only one applicable constructor. For example, given h : p ∧ q ∧ r, **have** ⟨h₁, h₂, h₃⟩ := h produces the hypotheses $h_1 : p$, $h_2 : q$, and $h_3 : r$.

tactic

have'

Similar to **have**, but using **refine'**

tactic

let

The **let** tactic is for adding definitions to the local context of the main goal.

- **let** x : t := e adds the definition x : t := e if e is a term of type t.
- **let** x := e uses the type of e for t.
- **let** : t := e and **let** := e use this for the name of the hypothesis.
- **let** pat := e for a pattern pat is equivalent to **match** e **with** | pat => _, where _ stands for the tactics that follow this one. It is convenient for types that let only one applicable constructor. For example, given $p : \alpha \times \beta \times \gamma$, **let** ⟨x, y, z⟩ := p produces the local variables $x : \alpha$, $y : \beta$, and $z : \gamma$.

tactic

let rec

`let rec f : t := e` adds a recursive definition `f` to the current goal. The syntax is the same as term-mode **`let rec`**.

tactic

`letI`

`letI` behaves like **`let`**, but inlines the value instead of producing a **`let`** term.

tactic

`let'`

Similar to **`let`**, but using **`refine'`**

13.3.5. Configuration

Many tactics are configurable. By convention, tactics share a configuration syntax, described using *optConfig*. The specific options available to each tactic are described in the tactic's documentation.

syntax

Tactic Configuration

A tactic configuration consists of zero or more configuration items:

```
optConfig ::=
    configItem*
```

syntax

Tactic Configuration Items

Each configuration item has a name that corresponds to an underlying tactic option. Boolean options may be enabled or disabled using prefix + and −:

```
configItem ::=
    +ident

configItem ::= ...
    | -ident
```

Options may be assigned specific values using a syntax similar to that for named function arguments:

```
configItem ::= ...
    | (ident := term)
```

Finally, the name `config` is reserved; it is used to pass an entire set of options as a data structure. The specific type expected depends on the tactic.

```
configItem ::= ...
    | (config := term)
```

13.3.6. Namespace and Option Management

Namespaces and options can be adjusted in tactic scripts using the same syntax as in terms.

tactic

set_option

`set_option` `opt val` **`in`** `tacs` (the tactic) acts like **`set_option`** `opt val` at the command level, but it sets the option only within the tactics `tacs`.

tactic

open

`open` `Foo` **`in`** `tacs` (the tactic) acts like **`open`** `Foo` at command level, but it opens a namespace only within the tactics `tacs`.

13.3.6.1. Controlling Unfolding

By default, only definitions marked reducible are unfolded, except when checking definitional equality. These operators allow this default to be adjusted for some part of a tactic script.

tactic

`with_reducible_and_instances`

`with_reducible_and_instances tacs` executes `tacs` using the `.instances` transparency setting. In this setting only definitions tagged as `[reducible]` or type class instances are unfolded.

tactic

`with_reducible`

`with_reducible tacs` executes `tacs` using the reducible transparency setting. In this setting only definitions tagged as `[reducible]` are unfolded.

tactic

`with_unfolding_all`

`with_unfolding_all tacs` executes `tacs` using the `.all` transparency setting. In this setting all definitions that are not opaque are unfolded.

13.4. Options

These options affect the meaning of tactics.

option

`tactic.customEliminators`

Default value: `true`

enable using custom eliminators in the 'induction' and 'cases' tactics defined using the '@[induction_eliminator]' and '@[cases_eliminator]' attributes

option

`tactic.skipAssignedInstances`

Default value: `true`

in the **`rw`** and **`simp`** tactics, if an instance implicit argument is assigned, do not try to synthesize instance.

option

`tactic.simp.trace`

Default value: `false`

When tracing is enabled, calls to **`simp`** or **`dsimp`** will print an equivalent **`simp only`** call.

13.5. Tactic Reference

13.5.1. Classical Logic

tactic

`classical`

`classical tacs` runs `tacs` in a scope where `Classical.propDecidable` is a low priority local instance.

Note that **classical** is a scoping tactic: it adds the instance only within the scope of the tactic.

13.5.2. Assumptions

tactic

assumption

assumption tries to solve the main goal using a hypothesis of compatible type, or else fails. Note also the ‹t› term notation, which is a shorthand for **show** *t* **by assumption**.

tactic

apply_assumption

apply_assumption looks for an assumption of the form `... → ∀ _, ... → head` where `head` matches the current goal.

You can specify additional rules to apply using **apply_assumption** `[...]`. By default **apply_assumption** will also try `rfl`, `trivial`, `congrFun`, and `congrArg`. If you don't want these, or don't want to use all hypotheses, use **apply_assumption** `only [...]`. You can use **apply_assumption** `[-h]` to omit a local hypothesis. You can use **apply_assumption using** $[a_1, \ldots]$ to use all lemmas which have been labelled with the attributes a_i (these attributes must be created using **register_label_attr**).

apply_assumption will use consequences of local hypotheses obtained via **symm**.

If **apply_assumption** fails, it will call **exfalso** and try again. Thus if there is an assumption of the form $P \to \neg Q$, the new tactic state will have two goals, P and Q.

You can pass a further configuration via the syntax **apply_rules** `(config := {...}) lemmas`. The options supported are the same as for **solve_by_elim** (and include all the options for **apply**).

13.5.3. Quantifiers

tactic

exists

exists $e_1, e_2, \ldots$ is shorthand for **refine** $\langle e_1, e_2, \ldots \rangle$; **try trivial**. It is useful for existential goals.

tactic

intro

Introduces one or more hypotheses, optionally naming and/or pattern-matching them. For each hypothesis to be introduced, the remaining main goal's target type must be a `let` or function type.

- *intro* by itself introduces one anonymous hypothesis, which can be accessed by e.g. `assumption`.
- `intro x y` introduces two hypotheses and names them. Individual hypotheses can be anonymized via _, or matched against a pattern:

```
intro (a, b)
-- ..., a : α, b : β ⊢ ...
```

- Alternatively, *intro* can be combined with pattern matching much like **fun**:

```
intro
| n + 1, 0 => tac
| ...
```

tactic

intros

Introduces zero or more hypotheses, optionally naming them.

- `intros` is equivalent to repeatedly applying `intro` until the goal is not an obvious candidate for `intro`, which is to say that so long as the goal is a `let` or a pi type (e.g. an implication, function, or universal quantifier), the `intros` tactic will introduce an anonymous hypothesis. This tactic does not unfold definitions.
- **intros** `x y ...` is equivalent to **intro** `x y ...`, introducing hypotheses for each supplied argument and unfolding definitions as necessary. Each argument can be either an identifier or a _. An identifier indicates a name to use for the corresponding introduced hypothesis, and a _ indicates that the hypotheses should be introduced anonymously.

Examples

Basic properties:

```
def AllEven (f : Nat → Nat) := ∀ n, f n % 2 = 0

-- Introduces the two obvious hypotheses automatically
example : ∀ (f : Nat → Nat), AllEven f → AllEven (fun k => f (k +
1)) := by
  intros
  /- Tactic state
     f✝ : Nat → Nat
     a✝ : AllEven f✝
     ⊢ AllEven fun k => f✝ (k + 1) -/
  sorry
```

```
-- Introduces exactly two hypotheses, naming only the first
example : ∀ (f : Nat → Nat), AllEven f → AllEven (fun k => f (k +
1)) := by
  intros g _
  /- Tactic state
     g : Nat → Nat
     a✝ : AllEven g
     ⊢ AllEven fun k => g (k + 1) -/
  sorry

-- Introduces exactly three hypotheses, which requires unfolding
`AllEven`
example : ∀ (f : Nat → Nat), AllEven f → AllEven (fun k => f (k +
1)) := by
  intros f h n
  /- Tactic state
     f : Nat → Nat
     h : AllEven f
     n : Nat
     ⊢ (fun k => f (k + 1)) n % 2 = 0 -/
  apply h
```

Implications:

```
example (p q : Prop) : p → q → p := by
  intros
  /- Tactic state
     a✝¹ : p
     a✝ : q
     ⊢ p      -/
  assumption
```

Let bindings:

```
example : let n := 1; let k := 2; n + k = 3 := by
  intros
  /- n✝ : Nat := 1
     k✝ : Nat := 2
     ⊢ n✝ + k✝ = 3 -/
  rfl
```

tactic

intro |_... => ..._ | _... => ...

The tactic

```
intro
| pat1 => tac1
| pat2 => tac2
```

is the same as:

```
intro x
match x with
| pat1 => tac1
| pat2 => tac2
```

That is, `intro` can be followed by match arms and it introduces the values while doing a pattern match. This is equivalent to **`fun`** with match arms in term mode.

tactic

`rintro`

The `rintro` tactic is a combination of the `intros` tactic with `rcases` to allow for destructuring patterns while introducing variables. See `rcases` for a description of supported patterns. For example, `rintro (a | ⟨b, c⟩) ⟨d, e⟩` will introduce two variables, and then do case splits on both of them producing two subgoals, one with variables `a d e` and the other with `b c d e`.

`rintro`, unlike `rcases`, also supports the form `(x y : ty)` for introducing and type-ascripting multiple variables at once, similar to binders.

13.5.4. Relations

tactic

`rfl`

This tactic applies to a goal whose target has the form `x ~ x`, where ~ is equality, heterogeneous equality or any relation that has a reflexivity lemma tagged with the attribute @[refl].

tactic

`rfl'`

`rfl'` is similar to `rfl`, but disables smart unfolding and unfolds all kinds of definitions, theorems included (relevant for declarations defined by well-founded recursion).

tactic

`apply_rfl`

The same as `rfl`, but without trying `eq_refl` at the end.

attribute

Reflexive Relations

The **`refl`** attribute marks a lemma as a proof of reflexivity for some relation. These lemmas are used by the **`rfl`**, **`rfl'`**, and **`apply_rfl`** tactics.

```
attr ::= ...
    | refl
```

tactic

symm

- **`symm`** applies to a goal whose target has the form `t ~ u` where ~ is a symmetric relation, that is, a relation which has a symmetry lemma tagged with the attribute [symm]. It replaces the target with `u ~ t`.
- **`symm at`** h will rewrite a hypothesis `h : t ~ u` to `h : u ~ t`.

tactic

symm_saturate

For every hypothesis `h : a ~ b` where a `@[symm]` lemma is available, add a hypothesis `h_symm : b ~ a`.

attribute

Symmetric Relations

The **`symm`** attribute marks a lemma as a proof that a relation is symmetric. These lemmas are used by the **`symm`** and **`symm_saturate`** tactics.

```
attr ::= ...
    | symm
```

tactic

calc

Step-wise reasoning over transitive relations.

```
calc
  a = b := pab
  b = c := pbc
  ...
  y = z := pyz
```

proves `a = z` from the given step-wise proofs. = can be replaced with any relation implementing the typeclass `Trans`. Instead of repeating the right- hand sides, subsequent left-hand sides can be replaced with _.

```
calc
  a = b := pab
  _ = c := pbc
  ...
  _ = z := pyz
```

It is also possible to write the *first* relation as `<lhs>\n  _ = <rhs> := <proof>`. This is useful for aligning relation symbols, especially on longer: identifiers:

```
calc abc
  _ = bce := pabce
  _ = cef := pbcef
  ...
  _ = xyz := pwxyz
```

`calc` works as a term, as a tactic or as a `conv` tactic.

See Theorem Proving in Lean 4 for more information.

type class

```
Trans.{u, v, w, u_1, u_2, u_3} {α : Sort u_1} {β : Sort u_2}
  {γ : Sort u_3} (r : α → β → Sort u) (s : β → γ → Sort v)
  (t : outParam (α → γ → Sort w)) :
  Sort (max (max (max (max (max (max 1 u) u_1) u_2) u_3) v) w)
```

Transitive chaining of proofs, used e.g. by **`calc`**.

It takes two relations `r` and `s` as "input", and produces an "output" relation `t`, with the property that `r a b` and `s b c` implies `t a c`. The **`calc`** tactic uses this so that when it sees a chain with `a ≤ b` and `b < c` it knows that this should be a proof of `a < c` because there is an instance `Trans (·≤·) (·<·) (·<·)`.

Instance Constructor

`Trans.mk.{u, v, w, u_1, u_2, u_3}`

Methods

`trans : {a : α} → {b : β} → {c : γ} → r a b → s b c → t a c`

Compose two proofs by transitivity, generalized over the relations involved.

13.5.4.1. Equality

tactic

subst

subst x... substitutes each hypothesis x with a definition found in the local context, then eliminates the hypothesis.

- If *x* is a local definition, then its definition is used.
- Otherwise, if there is a hypothesis of the form *x* = *e* or *e* = *x*, then *e* is used for the definition of *x*.

If *h* : *a* = *b*, then **subst** h may be used if either *a* or *b* unfolds to a local hypothesis. This is similar to the **cases** h tactic.

See also: **subst_vars** for substituting all local hypotheses that have a defining equation.

tactic

subst_eqs

subst_eq repeatedly substitutes according to the equality proof hypotheses in the context, replacing the left side of the equality with the right, until no more progress can be made.

tactic

subst_vars

Applies **subst** to all hypotheses of the form *h* : *x* = *t* or h : t = x.

tactic

congr

Apply congruence (recursively) to goals of the form ⊢ f as = f bs and ⊢ f as ≍ f bs. The optional parameter is the depth of the recursive applications. This is useful when congr is too aggressive in breaking down the goal. For example, given ⊢ f (g (x + y)) = f (g (y + x)), congr produces the goals ⊢ x = y and ⊢ y = x, while **congr** 2 produces the intended ⊢ x + y = y + x.

tactic

eq_refl

eq_refl is equivalent to **exact** rfl, but has a few optimizations.

tactic

ac_rfl

ac_rfl proves equalities up to application of an associative and commutative operator.

```
instance : Associative (α := Nat) (.+.) := ⟨Nat.add_assoc⟩
instance : Commutative (α := Nat) (.+.) := ⟨Nat.add_comm⟩

example (a b c d : Nat) : a + b + c + d = d + (b + c) + a := by
ac_rfl
```

13.5.5. Associativity and Commutativity

tactic

ac_nf

ac_nf normalizes equalities up to application of an associative and commutative operator.

- **ac_nf** normalizes all hypotheses and the goal target of the goal.
- **ac_nf at** `l` normalizes at location(s) `l`, where `l` is either `*` or a list of hypotheses in the local context. In the latter case, a turnstile ⊢ or `|-` can also be used, to signify the target of the goal.

```
instance : Associative (α := Nat) (.+.) := ⟨Nat.add_assoc⟩
instance : Commutative (α := Nat) (.+.) := ⟨Nat.add_comm⟩

example (a b c d : Nat) : a + b + c + d = d + (b + c) + a := by
 ac_nf
 -- goal: a + (b + (c + d)) = a + (b + (c + d))
```

tactic

ac_nf0

Implementation of **ac_nf** (the full **ac_nf** calls `trivial` afterwards).

13.5.6. Lemmas

tactic

exact

exact e closes the main goal if its target type matches that of e.

tactic

apply

apply e tries to match the current goal against the conclusion of e's type. If it succeeds, then the tactic returns as many subgoals as the number of premises that have not been fixed by type inference or type class resolution. Non-dependent premises are added before dependent ones.

The **apply** tactic uses higher-order pattern matching, type class resolution, and first-order unification with dependent types.

tactic

refine

refine e behaves like **exact** e, except that named (?x) or unnamed (?_) holes in e that are not solved by unification with the main goal's target type are converted into new goals, using the hole's name, if any, as the goal case name.

tactic

refine'

refine' e behaves like **refine** e, except that unsolved placeholders (_) and implicit parameters are also converted into new goals.

tactic

solve_by_elim

solve_by_elim calls **apply** on the main goal to find an assumption whose head matches and then repeatedly calls **apply** on the generated subgoals until no subgoals remain, performing at most `maxDepth` (defaults to 6) recursive steps.

solve_by_elim discharges the current goal or fails.

solve_by_elim performs backtracking if subgoals can not be solved.

By default, the assumptions passed to **apply** are the local context, `rfl`, `trivial`, `congrFun` and `congrArg`.

The assumptions can be modified with similar syntax as for **simp**:

- **solve_by_elim** `[h₁, h₂, ..., hᵣ]` also applies the given expressions.
- **solve_by_elim** `only [h₁, h₂, ..., hᵣ]` does not include the local context, `rfl`, `trivial`, `congrFun`, or `congrArg` unless they are explicitly included.
- **solve_by_elim** `[-h₁, ... -hₙ]` removes the given local hypotheses.
- **solve_by_elim using** `[a₁, ...]` uses all lemmas which have been labelled with the attributes `aᵢ` (these attributes must be created using **register_label_attr**).

solve_by_elim* tries to solve all goals together, using backtracking if a solution for one goal makes other goals impossible. (Adding or removing local hypotheses may not be well-behaved when starting with multiple goals.)

Optional arguments passed via a configuration argument as **solve_by_elim** `(config := { ... })`

- `maxDepth`: number of attempts at discharging generated subgoals
- **symm**: adds all hypotheses derived by **symm** (defaults to `true`).
- **exfalso**: allow calling **exfalso** and trying again if **solve_by_elim** fails (defaults to `true`).
- `transparency`: change the transparency mode when calling **apply**. Defaults to `.default`, but it is often useful to change to `.reducible`, so semireducible definitions will not be unfolded when trying to apply a lemma.

See also the doc-comment for `Lean.Meta.Tactic.Backtrack.BacktrackConfig` for the options `proc`, `suspend`, and `discharge` which allow further customization of **solve_by_elim**. Both **apply_assumption** and **apply_rules** are implemented via these hooks.

tactic

apply_rules

apply_rules `[l₁, l₂, ...]` tries to solve the main goal by iteratively applying the list of lemmas `[l₁, l₂, ...]` or by applying a local hypothesis. If **apply** generates new goals, **apply_rules** iteratively tries to solve those goals. You can use **apply_rules** `[-h]` to omit a local hypothesis.

apply_rules will also use `rfl`, `trivial`, congrFun and congrArg. These can be disabled, as can local hypotheses, by using **apply_rules** `only [...]`.

You can use **apply_rules using** `[a₁, ...]` to use all lemmas which have been labelled with the attributes a_i (these attributes must be created using **register_label_attr**).

You can pass a further configuration via the syntax **apply_rules** `(config := {...})`. The options supported are the same as for **solve_by_elim** (and include all the options for **apply**).

apply_rules will try calling **symm** on hypotheses and **exfalso** on the goal as needed. This can be disabled with **apply_rules** `(config := {symm := false, exfalso := false})`.

You can bound the iteration depth using the syntax **apply_rules** `(config := {maxDepth := n})`.

Unlike `solve_by_elim`, `apply_rules` does not perform backtracking, and greedily applies a lemma from the list until it gets stuck.

tactic

`as_aux_lemma`

`as_aux_lemma => tac` does the same as `tac`, except that it wraps the resulting expression into an auxiliary lemma. In some cases, this significantly reduces the size of expressions because the proof term is not duplicated.

13.5.7. Falsehood

tactic

`exfalso`

`exfalso` converts a goal `⊢ tgt` into `⊢ False` by applying `False.elim`.

tactic

`contradiction`

`contradiction` closes the main goal if its hypotheses are "trivially contradictory".

- Inductive type/family with no applicable constructors

```
example (h : False) : p := by contradiction
```

- Injectivity of constructors

```
example (h : none = some true) : p := by contradiction -- ok
```

- Decidable false proposition

```
example (h : 2 + 2 = 3) : p := by contradiction
```

- Contradictory hypotheses

```
example (h : p) (h' : ¬ p) : q := by contradiction
```

- Other simple contradictions such as

```
example (x : Nat) (h : x ≠ x) : p := by contradiction
```

tactic

`false_or_by_contra`

Changes the goal to `False`, retaining as much information as possible:

- If the goal is `False`, do nothing.
- If the goal is an implication or a function type, introduce the argument and restart. (In particular, if the goal is $x \neq y$, introduce $x = y$.)
- Otherwise, for a propositional goal P, replace it with $\neg \neg P$ (attempting to find a `Decidable` instance, but otherwise falling back to working classically) and introduce $\neg P$.
- For a non-propositional goal use `False.elim`.

13.5.8. Goal Management

tactic

`suffices`

Given a main goal `ctx ⊢ t`, **`suffices`** `h : t' from e` replaces the main goal with `ctx ⊢ t'`, `e` must have type `t` in the context `ctx, h : t'`.

The variant **`suffices`** `h : t' by tac` is a shorthand for **`suffices`** `h : t' from by tac`. If `h :` is omitted, the name `this` is used.

tactic

`change`

- **`change`** `tgt'` will change the goal from `tgt` to `tgt'`, assuming these are definitionally equal.
- **`change`** `t' at` h will change hypothesis $h : t$ to have type `t'`, assuming assuming t and `t'` are definitionally equal.

tactic

`change ... with ...`

- **`change`** a **`with`** b will change occurrences of a to b in the goal, assuming a and b are definitionally equal.
- **`change`** a **`with`** b **`at`** h similarly changes a to b in the type of hypothesis h.

tactic

`generalize`

- **generalize** `([h :] e = x),+` replaces all occurrences *e*s in the main goal with a fresh hypothesis *x*s. If *h* is given, *h* `: e = x` is introduced as well.
- **generalize** `e = x` **at** h_1 `...` h_n also generalizes occurrences of *e* inside h_1, ..., h_n.
- **generalize** `e = x` **at** `*` will generalize occurrences of *e* everywhere.

tactic

specialize

The tactic **specialize** `h` a_1 `...` a_n works on local hypothesis h. The premises of this hypothesis, either universal quantifications or non-dependent implications, are instantiated by concrete terms coming from arguments a_1 ... a_n. The tactic adds a new hypothesis with the same name `h := h` a_1 `...` a_n and tries to clear the previous one.

tactic

obtain

The **obtain** tactic is a combination of **have** and **rcases**. See **rcases** for a description of supported patterns.

```
obtain ⟨patt⟩ : type := proof
```

is equivalent to

```
have h : type := proof
rcases h with ⟨patt⟩
```

If `⟨patt⟩` is omitted, **rcases** will try to infer the pattern.

If `type` is omitted, `:= proof` is required.

tactic

show

show `t` finds the first goal whose target unifies with `t`. It makes that the main goal, performs the unification, and replaces the target with the unified version of `t`.

tactic

show_term

show_term *tac* runs *tac*, then prints the generated term in the form "exact X Y Z" or "refine X ?_ Z" (prefixed by **expose_names** if necessary) if there are remaining subgoals.

(For some tactics, the printed term will not be human readable.)

13.5.9. Cast Management

The tactics in this section make it easier avoid getting stuck on *casts*, which are functions that coerce data from one type to another, such as converting a natural number to the corresponding integer. They are described in more detail by Lewis and Madelaine (2020) (i) Robert Y. Lewis and Paul-Nicolas Madelaine, 2020. "Simplifying Casts and Coercions". arXiv:2001.10594.

tactic

norm_cast

The **norm_cast** family of tactics is used to normalize certain coercions (*casts*) in expressions.

- **norm_cast** normalizes casts in the target.
- **norm_cast at** h normalizes casts in hypothesis h.

The tactic is basically a version of **simp** with a specific set of lemmas to move casts upwards in the expression. Therefore even in situations where non-terminal **simp** calls are discouraged (because of fragility), **norm_cast** is considered to be safe. It also has special handling of numerals.

For instance, given an assumption

```
a b : ℤ
h : ↑a + ↑b < (10 : ℚ)
```

writing **norm_cast at** h will turn h into

```
h : a + b < 10
```

There are also variants of basic tactics that use **norm_cast** to normalize expressions during their operation, to make them more flexible about the expressions they accept (we say that it is a tactic *modulo* the effects of **norm_cast**):

- **exact_mod_cast** for **exact** and **apply_mod_cast** for **apply**. Writing **exact_mod_cast** h and **apply_mod_cast** h will normalize casts in the goal and h before using **exact** h or **apply** h.
- **rw_mod_cast** for **rw**. It applies **norm_cast** between rewrites.
- **assumption_mod_cast** for **assumption**. This is effectively **norm_cast at *; assumption**, but more efficient. It normalizes casts in the goal and, for every hypothesis h in the context, it will try to normalize casts in h and use **exact** h.

See also **push_cast**, which moves casts inwards rather than lifting them outwards.

tactic

push_cast

push_cast rewrites the goal to move certain coercions (*casts*) inward, toward the leaf nodes. This uses **norm_cast** lemmas in the forward direction. For example, `↑(a + b)` will be written to `↑a + ↑b`.

- **push_cast** moves casts inward in the goal.
- **push_cast at** h moves casts inward in the hypothesis h. It can be used with extra simp lemmas with, for example, **push_cast** `[Int.add_zero]`.

Example:

```
example (a b : Nat)
    (h1 : ((a + b : Nat) : Int) = 10)
    (h2 : ((a + b + 0 : Nat) : Int) = 10) :
    ((a + b : Nat) : Int) = 10 := by
  /-
  h1 : ↑(a + b) = 10
  h2 : ↑(a + b + 0) = 10
  ⊢ ↑(a + b) = 10
  -/
  push_cast
  /- Now
  ⊢ ↑a + ↑b = 10
  -/
  push_cast at h1
  push_cast [Int.add_zero] at h2
  /- Now
  h1 h2 : ↑a + ↑b = 10
  -/
  exact h1
```

See also **norm_cast**.

tactic

exact_mod_cast

Normalize casts in the goal and the given expression, then close the goal with **exact**.

tactic

apply_mod_cast

Normalize casts in the goal and the given expression, then **apply** the expression to the goal.

tactic

rw_mod_cast

Rewrites with the given rules, normalizing casts prior to each step.

tactic

`assumption_mod_cast`

`assumption_mod_cast` is a variant of `assumption` that solves the goal using a hypothesis. Unlike `assumption`, it first pre-processes the goal and each hypothesis to move casts as far outwards as possible, so it can be used in more situations.

Concretely, it runs `norm_cast` on the goal. For each local hypothesis h, it also normalizes h with `norm_cast` and tries to use that to close the goal.

13.5.10. Extensionality

tactic

`ext`

Applies extensionality lemmas that are registered with the `@[ext]` attribute.

- `ext pat*` applies extensionality theorems as much as possible, using the patterns `pat*` to introduce the variables in extensionality theorems using `rintro`. For example, the patterns are used to name the variables introduced by lemmas such as `funext`.
- Without patterns, `ext` applies extensionality lemmas as much as possible but introduces anonymous hypotheses whenever needed.
- `ext pat* : n` applies ext theorems only up to depth n.

The `ext1 pat*` tactic is like `ext pat*` except that it only applies a single extensionality theorem.

Unused patterns will generate warning. Patterns that don't match the variables will typically result in the introduction of anonymous hypotheses.

tactic

`ext1`

`ext1 pat*` is like `ext pat*` except that it only applies a single extensionality theorem rather than recursively applying as many extensionality theorems as possible.

The `pat*` patterns are processed using the `rintro` tactic. If no patterns are supplied, then variables are introduced anonymously using the `intros` tactic.

tactic

`apply_ext_theorem`

Apply a single extensionality theorem to the current goal.

tactic

funext

Apply function extensionality and introduce new hypotheses. The tactic **funext** will keep applying the `funext` lemma until the goal target is not reducible to

```
|-  ((fun x => ...) = (fun x => ...))
```

The variant **funext** h_1 ... h_n applies `funext` n times, and uses the given identifiers to name the new hypotheses. Patterns can be used like in the **intro** tactic. Example, given a goal

```
|-  ((fun x : Nat × Bool => ...) = (fun x => ...))
```

funext `(a, b)` applies `funext` once and performs pattern matching on the newly introduced pair.

13.5.11. Simplification

The simplifier is described in greater detail in its dedicated chapter.

tactic

simp

The **simp** tactic uses lemmas and hypotheses to simplify the main goal target or non-dependent hypotheses. It has many variants:

- **simp** simplifies the main goal target using lemmas tagged with the attribute `[`**simp**`]`.
- **simp** `[`h_1`, `h_2`, ..., `h_n`]` simplifies the main goal target using the lemmas tagged with the attribute `[`**simp**`]` and the given h_i's, where the h_i's are expressions.-
- If an h_i is a defined constant `f`, then `f` is unfolded. If `f` has equational lemmas associated with it (and is not a projection or a `reducible` definition), these are used to rewrite with `f`.
- **simp** `[*]` simplifies the main goal target using the lemmas tagged with the attribute `[`**simp**`]` and all hypotheses.
- **simp** `only [`h_1`, `h_2`, ..., `h_n`]` is like **simp** `[`h_1`, `h_2`, ..., `h_n`]` but does not use `[`**simp**`]` lemmas.
- **simp** `[-`id_1`, ..., -`id_n`]` simplifies the main goal target using the lemmas tagged with the attribute `[`**simp**`]`, but removes the ones named id_i.
- **simp at** h_1 h_2 ... h_n simplifies the hypotheses $h_1 : T_1 \ldots h_n : T_n$. If the target or another hypothesis depends on h_i, a new simplified hypothesis h_i is introduced, but the old one remains in the local context.

- **simp at *** simplifies all the hypotheses and the target.
- **simp [*] at *** simplifies target and all (propositional) hypotheses using the other hypotheses.

tactic

simp!

simp! is shorthand for **simp** with `autoUnfold := true`. This will rewrite with all equation lemmas, which can be used to partially evaluate many definitions.

tactic

simp?

simp? takes the same arguments as **simp**, but reports an equivalent call to **simp only** that would be sufficient to close the goal. This is useful for reducing the size of the simp set in a local invocation to speed up processing.

```
example (x : Nat) : (if True then x + 2 else 3) = x + 2 := by
  simp? -- prints "Try this: simp only [ite_true]"
```

This command can also be used in **simp_all** and **dsimp**.

tactic

simp?!

simp? takes the same arguments as **simp**, but reports an equivalent call to **simp only** that would be sufficient to close the goal. This is useful for reducing the size of the simp set in a local invocation to speed up processing.

```
example (x : Nat) : (if True then x + 2 else 3) = x + 2 := by
  simp? -- prints "Try this: simp only [ite_true]"
```

This command can also be used in **simp_all** and **dsimp**.

tactic

simp_arith

simp_arith has been deprecated. It was a shorthand for **simp** `+arith +decide`. Note that +**decide** is not needed for reducing arithmetic terms since simprocs have been added to Lean.

tactic

simp_arith!

simp_arith! has been deprecated. It was a shorthand for **simp!** `+arith +decide`. Note that `+`**decide** is not needed for reducing arithmetic terms since simprocs have been added to Lean.

tactic

dsimp

The **dsimp** tactic is the definitional simplifier. It is similar to **simp** but only applies theorems that hold by reflexivity. Thus, the result is guaranteed to be definitionally equal to the input.

tactic

dsimp!

dsimp! is shorthand for **dsimp** with `autoUnfold := true`. This will rewrite with all equation lemmas, which can be used to partially evaluate many definitions.

tactic

dsimp?

simp? takes the same arguments as **simp**, but reports an equivalent call to **simp only** that would be sufficient to close the goal. This is useful for reducing the size of the simp set in a local invocation to speed up processing.

```
example (x : Nat) : (if True then x + 2 else 3) = x + 2 := by
  simp? -- prints "Try this: simp only [ite_true]"
```

This command can also be used in **simp_all** and **dsimp**.

tactic

dsimp?!

simp? takes the same arguments as **simp**, but reports an equivalent call to **simp only** that would be sufficient to close the goal. This is useful for reducing the size of the simp set in a local invocation to speed up processing.

```
example (x : Nat) : (if True then x + 2 else 3) = x + 2 := by
  simp? -- prints "Try this: simp only [ite_true]"
```

This command can also be used in **simp_all** and **dsimp**.

tactic

simp_all

simp_all is a stronger version of **simp** `[*]` **at** `*` where the hypotheses and target are simplified multiple times until no simplification is applicable. Only non-dependent propositional hypotheses are considered.

tactic

simp_all!

simp_all! is shorthand for **simp_all** with `autoUnfold := true`. This will rewrite with all equation lemmas, which can be used to partially evaluate many definitions.

tactic

simp_all?

simp? takes the same arguments as **simp**, but reports an equivalent call to **simp only** that would be sufficient to close the goal. This is useful for reducing the size of the simp set in a local invocation to speed up processing.

```
example (x : Nat) : (if True then x + 2 else 3) = x + 2 := by
  simp? -- prints "Try this: simp only [ite_true]"
```

This command can also be used in **simp_all** and **dsimp**.

tactic

simp_all?!

simp? takes the same arguments as **simp**, but reports an equivalent call to **simp only** that would be sufficient to close the goal. This is useful for reducing the size of the simp set in a local invocation to speed up processing.

```
example (x : Nat) : (if True then x + 2 else 3) = x + 2 := by
  simp? -- prints "Try this: simp only [ite_true]"
```

This command can also be used in **simp_all** and **dsimp**.

tactic

simp_all_arith

simp_all_arith has been deprecated. It was a shorthand for **simp_all** `+arith +decide`. Note that `+`**decide** is not needed for reducing arithmetic terms since simprocs have been added to Lean.

tactic

simp_all_arith!

simp_all_arith! has been deprecated. It was a shorthand for **simp_all!** `+arith +decide`. Note that `+`**decide** is not needed for reducing arithmetic terms since simprocs have been added to Lean.

tactic

simpa

This is a "finishing" tactic modification of **simp**. It has two forms.

- **simpa** `[rules, ⋯]` **using** `e` will simplify the goal and the type of `e` using `rules`, then try to close the goal using `e`.

Simplifying the type of `e` makes it more likely to match the goal (which has also been simplified). This construction also tends to be more robust under changes to the simp lemma set.

- **simpa** `[rules, ⋯]` will simplify the goal and the type of a hypothesis `this` if present in the context, then try to close the goal using the **assumption** tactic.

tactic

simpa!

This is a "finishing" tactic modification of **simp**. It has two forms.

- **simpa** `[rules, ⋯]` **using** `e` will simplify the goal and the type of `e` using `rules`, then try to close the goal using `e`.

Simplifying the type of `e` makes it more likely to match the goal (which has also been simplified). This construction also tends to be more robust under changes to the simp lemma set.

- **simpa** `[rules, ⋯]` will simplify the goal and the type of a hypothesis `this` if present in the context, then try to close the goal using the **assumption** tactic.

tactic

simpa?

This is a "finishing" tactic modification of **simp**. It has two forms.

- **simpa** `[rules, ⋯]` **using** `e` will simplify the goal and the type of `e` using `rules`, then try to close the goal using `e`.

Simplifying the type of `e` makes it more likely to match the goal (which has also been simplified). This construction also tends to be more robust under changes to the simp lemma set.

- **simpa** [rules, ⋯] will simplify the goal and the type of a hypothesis `this` if present in the context, then try to close the goal using the **assumption** tactic.

tactic

simpa?!

This is a "finishing" tactic modification of **simp**. It has two forms.

- **simpa** [rules, ⋯] **using** e will simplify the goal and the type of e using rules, then try to close the goal using e.

Simplifying the type of e makes it more likely to match the goal (which has also been simplified). This construction also tends to be more robust under changes to the simp lemma set.

- **simpa** [rules, ⋯] will simplify the goal and the type of a hypothesis `this` if present in the context, then try to close the goal using the **assumption** tactic.

tactic

simp_wf

Unfold definitions commonly used in well founded relation definitions.

Since Lean 4.12, Lean unfolds these definitions automatically before presenting the goal to the user, and this tactic should no longer be necessary. Calls to **simp_wf** can be removed or replaced by plain calls to **simp**.

13.5.12. Rewriting

tactic

rw

rw is like **rewrite**, but also tries to close the goal by "cheap" (reducible) rfl afterwards.

tactic

rewrite

rewrite [e] applies identity e as a rewrite rule to the target of the main goal. If e is preceded by left arrow (← or <−), the rewrite is applied in the reverse direction. If e is a defined constant, then the equational theorems associated with e are used. This provides a convenient way to unfold e.

- **rewrite** [e_1, ..., e_n] applies the given rules sequentially.

- **rewrite** `[e] at l` rewrites `e` at location(s) `l`, where `l` is either `*` or a list of hypotheses in the local context. In the latter case, a turnstile `⊢` or `|-` can also be used, to signify the target of the goal.

Using **rw** `(occs := .pos L) [e]`, where *L* `: List Nat`, you can control which "occurrences" are rewritten. (This option applies to each rule, so usually this will only be used with a single rule.) Occurrences count from `1`. At each allowed occurrence, arguments of the rewrite rule `e` may be instantiated, restricting which later rewrites can be found. (Disallowed occurrences do not result in instantiation.) `(occs := .neg L)` allows skipping specified occurrences.

tactic

erw

erw `[rules]` is a shorthand for **rw** `(transparency := .default) [rules]`. This does rewriting up to unfolding of regular definitions (by comparison to regular **rw** which only unfolds `@[reducible]` definitions).

tactic

rwa

rwa is short-hand for **`rw; assumption`**.

structure

`Lean.Meta.Rewrite.Config : Type`

Configures the behavior of the **rewrite** and **rw** tactics.

Constructor

Lean.Meta.Rewrite.Config.mk

Fields

`transparency : Lean.Meta.TransparencyMode`

The transparency mode to use for unfolding

`offsetCnstrs : Bool`

Whether to support offset constraints such as `?x + 1 =?= e`

`occs : Lean.Meta.Occurrences`

Which occurrences to rewrite

`newGoals : Lean.Meta.Rewrite.NewGoals`

How to convert the resulting metavariables into new goals

inductive type

`Lean.Meta.Occurrences : Type`

Configuration for which occurrences that match an expression should be rewritten.

Constructors

| `all : Lean.Meta.Occurrences`

All occurrences should be rewritten.

| `pos (idxs : List Nat) : Lean.Meta.Occurrences`

A list of indices for which occurrences should be rewritten.

| `neg (idxs : List Nat) : Lean.Meta.Occurrences`

A list of indices for which occurrences should not be rewritten.

inductive type

`Lean.Meta.TransparencyMode : Type`

Which constants should be unfolded?

Constructors

| `all : Lean.Meta.TransparencyMode`

Unfolds all constants, even those tagged as `@[irreducible]`.

| `default : Lean.Meta.TransparencyMode`

Unfolds all constants except those tagged as `@[irreducible]`.

| `reducible : Lean.Meta.TransparencyMode`

Unfolds only constants tagged with the `@[reducible]` attribute.

| `instances : Lean.Meta.TransparencyMode`

Unfolds reducible constants and constants tagged with the `@[instance]` attribute.

def

`Lean.Meta.Rewrite.NewGoals : Type`

Controls which new mvars are turned in to goals by the **`apply`** tactic.

- `nonDependentFirst` mvars that don't depend on other goals appear first in the goal list.
- `nonDependentOnly` only mvars that don't depend on other goals are added to goal list.
- `all` all unassigned mvars are added to the goal list.

tactic

`unfold`

- **`unfold`** `id` unfolds all occurrences of definition `id` in the target.
- **`unfold`** `id1 id2 ...` is equivalent to **`unfold`** `id1; unfold id2; ...`.
- **`unfold`** `id` **`at`** `h` unfolds at the hypothesis h.

Definitions can be either global or local definitions.

For non-recursive global definitions, this tactic is identical to **`delta`**. For recursive global definitions, it uses the "unfolding lemma" `id.eq_def`, which is generated for each recursive definition, to unfold according to the recursive definition given by the user. Only one level of unfolding is performed, in contrast to **`simp only`** `[id]`, which unfolds definition `id` recursively.

Implemented by `Lean.Elab.Tactic.evalUnfold`.

tactic

`replace`

Acts like **`have`**, but removes a hypothesis with the same name as this one if possible. For example, if the state is:

```
f : α → β
h : α
⊢ goal
```

Then after **`replace`** `h := f h` the state will be:

```
f : α → β
h : β
⊢ goal
```

whereas **`have`** `h := f h` would result in:

```
f : α → β
h† : α
h : β
⊢ goal
```

This can be used to simulate the **`specialize`** and **`apply at`** tactics of Coq.

tactic

`delta`

`delta` `id1 id2 ...` delta-expands the definitions `id1`, `id2`, ... This is a low-level tactic, it will expose how recursive definitions have been compiled by Lean.

13.5.13. Inductive Types

13.5.13.1. Introduction

tactic

`constructor`

If the main goal's target type is an inductive type, **`constructor`** solves it with the first matching constructor, or else fails.

tactic

`injection`

The **`injection`** tactic is based on the fact that constructors of inductive data types are injections. That means that if *c* is a constructor of an inductive datatype, and if `(c t₁)` and `(c t₂)` are two terms that are equal then t_1 and t_2 are equal too. If q is a proof of a statement of conclusion $t_1 = t_2$, then injection applies injectivity to derive the equality of all arguments of t_1 and t_2 placed in the same positions. For example, from $(a::b) = (c::d)$ we derive *a*=*c* and *b*=*d*. To use this tactic t_1 and t_2 should be constructor applications of the same constructor. Given *h* `: a::b = c::d`, the tactic **`injection`** h adds two new hypothesis with types *a* `=` *c* and *b* `=` *d* to the main goal. The tactic **`injection`** h **`with`** h_1 h_2 uses the names h_1 and h_2 to name the new hypotheses.

tactic

`injections`

`injections` applies **`injection`** to all hypotheses recursively (since **`injection`** can produce new hypotheses). Useful for destructing nested constructor equalities like $(a::b::c) = (d::e::f)$.

tactic

left

Applies the first constructor when the goal is an inductive type with exactly two constructors, or fails otherwise.

```
example : True ∨ False := by
  left
  trivial
```

tactic

right

Applies the second constructor when the goal is an inductive type with exactly two constructors, or fails otherwise.

```
example {p q : Prop} (h : q) : p ∨ q := by
  right
  exact h
```

13.5.13.2. Elimination

Elimination tactics use recursors and the automatically-derived `casesOn` helper to implement induction and case splitting. The subgoals that result from these tactics are determined by the types of the minor premises of the eliminators, and using different eliminators with the `using` option results in different subgoals.

▼ *Example: Choosing Eliminators*

When attempting to prove that `∀(i : Fin (n + 1)), 0 + i = i`, after introducing the hypotheses the tactic `induction i` results in:

```
case mk
   n : Nat
 val✝ : Nat
isLt✝ : val✝ < n + 1
⊢ 0 + ⟨val✝, isLt✝⟩ = ⟨val✝, isLt✝⟩
```

This is because `Fin` is a structure with a single non-recursive constructor. Its recursor has a single minor premise for this constructor:

```
Fin.rec.{u} {n : Nat} {motive : Fin n → Sort u}
  (mk : (val : Nat) →
    (isLt : val < n) →
    motive ⟨val, isLt⟩)
  (t : Fin n) : motive t
```

Using the tactic `induction i using Fin.induction` instead results in:

```
  case zero
n : Nat
⊢ 0 + 0 = 0

  case succ
  n : Nat
i✝ : Fin n
a✝ : 0 + i✝.castSucc = i✝.castSucc
⊢ 0 + i✝.succ = i✝.succ
```

`Fin.induction` is an alternative eliminator that implements induction on the underlying `Nat`:

```
Fin.induction.{u} {n : Nat}
  {motive : Fin (n + 1) → Sort u}
  (zero : motive 0)
  (succ : (i : Fin n) →
    motive i.castSucc →
    motive i.succ)
  (i : Fin (n + 1)) : motive i
```

Custom eliminators can be registered using the **`induction_eliminator`** and **`cases_eliminator`** attributes. The eliminator is registered for its explicit targets (i.e. those that are explicit, rather than implicit, parameters to the eliminator function) and will be applied when **`induction`** or **`cases`** is used on targets of those types. When present, custom eliminators take precedence over recursors. Setting `tactic.customEliminators` to `false` disables the use of custom eliminators.

attribute

Custom Eliminators

The **`induction_eliminator`** attribute registers an eliminator for use by the **`induction`** tactic.

```
attr ::= ...
    | induction_eliminator
```

The **`induction_eliminator`** attribute registers an eliminator for use by the **`cases`** tactic.

```
attr ::= ...
    | cases_eliminator
```

tactic

`cases`

Assuming x is a variable in the local context with an inductive type, **cases** x splits the main goal, producing one goal for each constructor of the inductive type, in which the target is replaced by a general instance of that constructor. If the type of an element in the local context depends on x, that element is reverted and reintroduced afterward, so that the case split affects that hypothesis as well. **cases** detects unreachable cases and closes them automatically.

For example, given *n* `:` Nat and a goal with a hypothesis `h : P n` and target `Q n`, **cases** `n` produces one goal with hypothesis `h : P 0` and target `Q 0`, and one goal with hypothesis `h : P (Nat.succ a)` and target `Q (Nat.succ a)`. Here the name `a` is chosen automatically and is not accessible. You can use **with** to provide the variables names for each constructor.

- **cases** e, where e is an expression instead of a variable, generalizes e in the goal, and then cases on the resulting variable.
- Given *as* `:` List *α*, **cases** as **with** `| nil => tac₁ | cons a as' => tac₂`, uses tactic $\texttt{tac}_1$ for the `nil` case, and $\texttt{tac}_2$ for the `cons` case, and `a` and `as'` are used as names for the new variables introduced.
- **cases** `h :` e, where e is a variable or an expression, performs cases on e as above, but also adds a hypothesis `h : e = ...` to each hypothesis, where `...` is the constructor instance for that particular case.

tactic

rcases

rcases is a tactic that will perform **cases** recursively, according to a pattern. It is used to destructure hypotheses or expressions composed of inductive types like *h1* `:` *a* ∧ *b* ∧ *c* ∨ *d* or `h2 : ∃ x y, trans_rel R x y`. Usual usage might be **rcases** `h1` **with** `⟨ha, hb, hc⟩ | hd` or **rcases** `h2` **with** `⟨x, y, _ | ⟨z, hxz, hzy⟩⟩` for these examples.

Each element of an **rcases** pattern is matched against a particular local hypothesis (most of which are generated during the execution of **rcases** and represent individual elements destructured from the input expression). An **rcases** pattern has the following grammar:

- A name like x, which names the active hypothesis as x.
- A blank _, which does nothing (letting the automatic naming system used by **cases** name the hypothesis).
- A hyphen −, which clears the active hypothesis and any dependents.
- The keyword rfl, which expects the hypothesis to be *h* `:` *a* = *b*, and calls **subst** on the hypothesis (which has the effect of replacing *b* with *a* everywhere or vice versa).
- A type ascription *p* `:` *ty*, which sets the type of the hypothesis to *ty* and then matches it against *p*. (Of course, *ty* must unify with the actual type of *h* for this to work.)

- A tuple pattern `⟨p1, p2, p3⟩`, which matches a constructor with many arguments, or a series of nested conjunctions or existentials. For example if the active hypothesis is $a \wedge b \wedge c$, then the conjunction will be destructured, and `p1` will be matched against a, `p2` against b and so on.
- A `@` before a tuple pattern as in `@⟨p1, p2, p3⟩` will bind all arguments in the constructor, while leaving the `@` off will only use the patterns on the explicit arguments.
- An alternation pattern `p1 | p2 | p3`, which matches an inductive type with multiple constructors, or a nested disjunction like $a \vee b \vee c$.

A pattern like `⟨a, b, c⟩ | ⟨d, e⟩` will do a split over the inductive datatype, naming the first three parameters of the first constructor as `a`,`b`,`c` and the first two of the second constructor `d`,`e`. If the list is not as long as the number of arguments to the constructor or the number of constructors, the remaining variables will be automatically named. If there are nested brackets such as `⟨⟨a⟩, b | c⟩ | d` then these will cause more case splits as necessary. If there are too many arguments, such as `⟨a, b, c⟩` for splitting on `∃ x, ∃ y, p x`, then it will be treated as `⟨a, ⟨b, c⟩⟩`, splitting the last parameter as necessary.

rcases also has special support for quotient types: quotient induction into Prop works like matching on the constructor `quot.mk`.

rcases `h : e` **with** PAT will do the same as **rcases** `e` **with** PAT with the exception that an assumption h `: e =` PAT will be added to the context.

tactic

fun_cases

The `fun_cases` tactic is a convenience wrapper of the **cases** tactic when using a functional cases principle.

The tactic invocation

```
fun_cases f x ... y ...`
```

is equivalent to

```
cases y, ... using f.fun_cases_unfolding x ...
```

where the arguments of `f` are used as arguments to `f.fun_cases_unfolding` or targets of the case analysis, as appropriate.

The form

```
fun_cases f
```

(with no arguments to `f`) searches the goal for an unique eligible application of `f`, and uses these arguments. An application of `f` is eligible if it is saturated and the arguments that will become targets are free variables.

The form **fun_cases** `f x y` **with** | `case1 => tac₁` | `case2 x' ih => tac₂` works like with **cases**.

Under **set_option** `tactic.fun_induction.unfolding true` (the default), **fun_induction** uses the `f.fun_cases_unfolding` theorem, which will try to automatically unfold the call to `f` in the goal. With **set_option** `tactic.fun_induction.unfolding false`, it uses `f.fun_cases` instead.

tactic

induction

Assuming `x` is a variable in the local context with an inductive type, **induction** `x` applies induction on `x` to the main goal, producing one goal for each constructor of the inductive type, in which the target is replaced by a general instance of that constructor and an inductive hypothesis is added for each recursive argument to the constructor. If the type of an element in the local context depends on `x`, that element is reverted and reintroduced afterward, so that the inductive hypothesis incorporates that hypothesis as well.

For example, given *n* `:` Nat and a goal with a hypothesis `h : P n` and target `Q n`, **induction** `n` produces one goal with hypothesis `h : P 0` and target `Q 0`, and one goal with hypotheses `h : P (Nat.succ a)` and ih_1 `: P a → Q a` and target `Q (Nat.succ a)`. Here the names `a` and ih_1 are chosen automatically and are not accessible. You can use **with** to provide the variables names for each constructor.

- **induction** `e`, where `e` is an expression instead of a variable, generalizes `e` in the goal, and then performs induction on the resulting variable.
- **induction** `e` **using** `r` allows the user to specify the principle of induction that should be used. Here `r` should be a term whose result type must be of the form `C t`, where *C* is a bound variable and `t` is a (possibly empty) sequence of bound variables
- **induction** `e` **generalizing** z_1 `...` z_n, where z_1 `...` z_n are variables in the local context, generalizes over z_1 `...` z_n before applying the induction but then introduces them in each goal. In other words, the net effect is that each inductive hypothesis is generalized.
- Given *x* `:` Nat, **induction** `x` **with** | `zero => tac₁` | `succ x' ih => tac₂` uses tactic tac_1 for the `zero` case, and tac_2 for the `succ` case.

tactic

fun_induction

The **fun_induction** tactic is a convenience wrapper around the **induction** tactic to use the the functional induction principle.

The tactic invocation

```
fun_induction f x₁ ... xₙ y₁ ... yₘ
```

where `f` is a function defined by non-mutual structural or well-founded recursion, is equivalent to

```
induction y₁, ... yₘ using f.induct_unfolding x₁ ... xₙ
```

where the arguments of `f` are used as arguments to `f.induct_unfolding` or targets of the induction, as appropriate.

The form

```
fun_induction f
```

(with no arguments to `f`) searches the goal for an unique eligible application of `f`, and uses these arguments. An application of `f` is eligible if it is saturated and the arguments that will become targets are free variables.

The forms **`fun_induction`** `f x y` **`generalizing`** `z₁ ... zₙ` and **`fun_induction`** `f x y` **`with`** `| case1 => tac₁ | case2 x' ih => tac₂` work like with **`induction`**.

Under **`set_option`** `tactic.fun_induction.unfolding true` (the default), **`fun_induction`** uses the `f.induct_unfolding` induction principle, which will try to automatically unfold the call to `f` in the goal. With **`set_option`** `tactic.fun_induction.unfolding false`, it uses `f.induct` instead.

tactic

`nofun`

The tactic **`nofun`** is shorthand for **`exact nofun`**: it introduces the assumptions, then performs an empty pattern match, closing the goal if the introduced pattern is impossible.

tactic

`nomatch`

The tactic **`nomatch`** h is shorthand for **`exact nomatch`** h.

13.5.14. Library Search

The library search tactics are intended for interactive use. When run, they search the Lean library for lemmas or rewrite rules that could be applicable in the current situation, and suggests a new tactic. These tactics should not be left in a proof; rather, their suggestions should be incorporated.

tactic

`exact?`

Searches environment for definitions or theorems that can solve the goal using **exact** with conditions resolved by **solve_by_elim**.

The optional **using** clause provides identifiers in the local context that must be used by **exact?** when closing the goal. This is most useful if there are multiple ways to resolve the goal, and one wants to guide which lemma is used.

tactic

apply?

Searches environment for definitions or theorems that can refine the goal using **apply** with conditions resolved when possible with **solve_by_elim**.

The optional **using** clause provides identifiers in the local context that must be used when closing the goal.

In this proof state:

```
i : Nat
j : Nat
k : Nat
h1 : i ≤ j
h2 : j ≤ k
⊢ i ≤ k
```

invoking **apply?** suggests:

```
Try this: exact Nat.lt_trans h1 h2
```

tactic

rw?

rw? tries to find a lemma which can rewrite the goal.

rw? should not be left in proofs; it is a search tool, like **apply?**.

Suggestions are printed as **rw** `[h]` or **rw** `[← h]`.

You can use **rw?** `[-my_lemma, -my_theorem]` to prevent **rw?** using the named lemmas.

13.5.15. Case Analysis

tactic

split

The **split** tactic is useful for breaking nested if-then-else and **match** expressions into separate cases. For a **match** expression with n cases, the **split** tactic generates at most n subgoals.

For example, given *n* `: Nat`, and a target **if** *n* `= 0` **then** *Q* **else** *R*, **split** will generate one goal with hypothesis *n* `= 0` and target *Q*, and a second goal with hypothesis ¬*n* `= 0` and target *R*. Note that the introduced hypothesis is unnamed, and is commonly renamed used the **case** or **next** tactics.

- **split** will split the goal (target).
- **split at** h will split the hypothesis h.

tactic

by_cases

by_cases `(h :)?` p splits the main goal into two cases, assuming *h* `:` *p* in the first branch, and h `: ¬` p in the second branch.

13.5.16. Decision Procedures

tactic

decide

decide attempts to prove the main goal (with target type *p*) by synthesizing an instance of `Decidable` *p* and then reducing that instance to evaluate the truth value of *p*. If it reduces to `isTrue` *h*, then *h* is a proof of *p* that closes the goal.

The target is not allowed to contain local variables or metavariables. If there are local variables, you can first try using the **revert** tactic with these local variables to move them into the target, or you can use the `+`**revert** option, described below.

Options:

- **decide** `+revert` begins by reverting local variables that the target depends on, after cleaning up the local context of irrelevant variables. A variable is *relevant* if it appears in the target, if it appears in a relevant variable, or if it is a proposition that refers to a relevant variable.
- **decide** `+kernel` uses kernel for reduction instead of the elaborator. It has two key properties: (1) since it uses the kernel, it ignores transparency and can unfold everything, and (2) it reduces the `Decidable` instance only once instead of twice.
- **decide** `+native` uses the native code compiler (**#eval**) to evaluate the `Decidable` instance, admitting the result via the `Lean.ofReduceBool` axiom. This can be significantly more efficient than using reduction, but it is at the cost of increasing the size

of the trusted code base. Namely, it depends on the correctness of the Lean compiler and all definitions with an `@[implemented_by]` attribute. Like with `+kernel`, the `Decidable` instance is evaluated only once.

Limitation: In the default mode or `+kernel` mode, since `decide` uses reduction to evaluate the term, `Decidable` instances defined by well-founded recursion might not work because evaluating them requires reducing proofs. Reduction can also get stuck on `Decidable` instances with `Eq.rec` terms. These can appear in instances defined using tactics (such as `rw` and `simp`). To avoid this, create such instances using definitions such as `decidable_of_iff` instead.

Examples

Proving inequalities:

```
example : 2 + 2 ≠ 5 := by decide
```

Trying to prove a false proposition:

```
example : 1 ≠ 1 := by decide
/-
tactic 'decide' proved that the proposition
  1 ≠ 1
is false
-/
```

Trying to prove a proposition whose `Decidable` instance fails to reduce

```
opaque unknownProp : Prop

open scoped Classical in
example : unknownProp := by decide
/-
tactic 'decide' failed for proposition
  unknownProp
since its 'Decidable' instance reduced to
  Classical.choice ⋯
rather than to the 'isTrue' constructor.
-/
```

Properties and relations

For equality goals for types with decidable equality, usually `rfl` can be used in place of `decide`.

```
example : 1 + 1 = 2 := by decide
example : 1 + 1 = 2 := by rfl
```

tactic

`native_decide`

native_decide is a synonym for **decide** +native. It will attempt to prove a goal of type *p* by synthesizing an instance of Decidable *p* and then evaluating it to isTrue .... Unlike decide, this uses **#eval** to evaluate the decidability instance.

This should be used with care because it adds the entire lean compiler to the trusted part, and the axiom Lean.ofReduceBool will show up in **#print** axioms for theorems using this method or anything that transitively depends on them. Nevertheless, because it is compiled, this can be significantly more efficient than using decide, and for very large computations this is one way to run external programs and trust the result.

```
example : (List.range 1000).length = 1000 := by native_decide
```

tactic

omega

The **omega** tactic, for resolving integer and natural linear arithmetic problems.

It is not yet a full decision procedure (no "dark" or "grey" shadows), but should be effective on many problems.

We handle hypotheses of the form *x* = *y*, x < y, x ≤ y, and k ∣ x for x y in Nat or Int (and *k* a literal), along with negations of these statements.

We decompose the sides of the inequalities as linear combinations of atoms.

If we encounter x / k or x % k for literal integers *k* we introduce new auxiliary variables and the relevant inequalities.

On the first pass, we do not perform case splits on natural subtraction. If **omega** fails, we recursively perform a case split on a natural subtraction appearing in a hypothesis, and try again.

The options

```
omega +splitDisjunctions +splitNatSub +splitNatAbs +splitMinMax
```

can be used to:

- splitDisjunctions: split any disjunctions found in the context, if the problem is not otherwise solvable.
- splitNatSub: for each appearance of ((*a* - *b* : Nat) : Int), split on *a* ≤ *b* if necessary.
- splitNatAbs: for each appearance of Int.natAbs *a*, split on 0 ≤ *a* if necessary.
- splitMinMax: for each occurrence of min *a* *b*, split on min *a* *b* = *a* ∨ min *a* *b* = *b* Currently, all of these are on by default.

tactic

bv_omega

bv_omega is **omega** with an additional preprocessor that turns statements about BitVec into statements about Nat. Currently the preprocessor is implemented as **try simp only** [bitvec_to_nat] **at** *. bitvec_to_nat is a @[**simp**] attribute that you can (cautiously) add to more theorems.

13.5.16.1. SAT Solver Integration

tactic

bv_decide

Close fixed-width BitVec and Bool goals by obtaining a proof from an external SAT solver and verifying it inside Lean. The solvable goals are currently limited to

- the Lean equivalent of QF_BV
- automatically splitting up **structure**s that contain information about BitVec or Bool

```
example : ∀ (a b : BitVec 64), (a &&& b) + (a ^^^ b) = a ||| b :=
by
  intros
  bv_decide
```

If **bv_decide** encounters an unknown definition it will be treated like an unconstrained BitVec variable. Sometimes this enables solving goals despite not understanding the definition because the precise properties of the definition do not matter in the specific proof.

If **bv_decide** fails to close a goal it provides a counter-example, containing assignments for all terms that were considered as variables.

In order to avoid calling a SAT solver every time, the proof can be cached with **bv_decide?**.

If solving your problem relies inherently on using associativity or commutativity, consider enabling the bv.ac_nf option.

Note: **bv_decide** uses ofReduceBool and thus trusts the correctness of the code generator.

Note: include **import** Std.Tactic.BVDecide

tactic

bv_normalize

Run the normalization procedure of **bv_decide** only. Sometimes this is enough to solve basic BitVec goals already.

Note: include `import Std.Tactic.BVDecide`

tactic

bv_check

This tactic works just like **bv_decide** but skips calling a SAT solver by using a proof that is already stored on disk. It is called with the name of an LRAT file in the same directory as the current Lean file:

```
bv_check "proof.lrat"
```

tactic

bv_decide?

Suggest a proof script for a **bv_decide** tactic call. Useful for caching LRAT proofs.

Note: include `import Std.Tactic.BVDecide`

13.5.17. Controlling Reduction

tactic

with_reducible

`with_reducible tacs` executes `tacs` using the reducible transparency setting. In this setting only definitions tagged as `[reducible]` are unfolded.

tactic

with_reducible_and_instances

`with_reducible_and_instances tacs` executes `tacs` using the `.instances` transparency setting. In this setting only definitions tagged as `[reducible]` or type class instances are unfolded.

tactic

with_unfolding_all

`with_unfolding_all tacs` executes `tacs` using the `.all` transparency setting. In this setting all definitions that are not opaque are unfolded.

13.5.18. Control Flow

tactic

skip

skip does nothing.

tactic

guard_hyp

Tactic to check that a named hypothesis has a given type and/or value.

- guard_hyp h : t checks the type up to reducible defeq,
- guard_hyp h :~ t checks the type up to default defeq,
- guard_hyp h :$_s$ t checks the type up to syntactic equality,
- guard_hyp h :$_a$ t checks the type up to alpha equality.
- guard_hyp h := v checks value up to reducible defeq,
- guard_hyp h :=~ v checks value up to default defeq,
- guard_hyp h :=$_s$ v checks value up to syntactic equality,
- guard_hyp h :=$_a$ v checks the value up to alpha equality.

The value v is elaborated using the type of h as the expected type.

tactic

guard_target

Tactic to check that the target agrees with a given expression.

- guard_target = e checks that the target is defeq at reducible transparency to e.
- guard_target =~ e checks that the target is defeq at default transparency to e.
- guard_target =$_s$ e checks that the target is syntactically equal to e.
- guard_target =$_a$ e checks that the target is alpha-equivalent to e.

The term e is elaborated with the type of the goal as the expected type, which is mostly useful within conv mode.

tactic

guard_expr

Tactic to check equality of two expressions.

- `guard_expr e = e'` checks that `e` and `e'` are defeq at reducible transparency.
- `guard_expr e =~ e'` checks that `e` and `e'` are defeq at default transparency.
- `guard_expr e =ₛ e'` checks that `e` and `e'` are syntactically equal.
- `guard_expr e =ₐ e'` checks that `e` and `e'` are alpha-equivalent.

Both `e` and `e'` are elaborated then have their metavariables instantiated before the equality check. Their types are unified (using `isDefEqGuarded`) before synthetic metavariables are processed, which helps with default instance handling.

tactic

`done`

`done` succeeds iff there are no remaining goals.

tactic

`sleep`

The tactic `sleep ms` sleeps for `ms` milliseconds and does nothing. It is used for debugging purposes only.

tactic

`stop`

`stop` is a helper tactic for "discarding" the rest of a proof: it is defined as `repeat sorry`. It is useful when working on the middle of a complex proofs, and less messy than commenting the remainder of the proof.

13.5.19. Term Elaboration Backends

These tactics are used during elaboration of terms to satisfy obligations that arise.

tactic

`decreasing_with`

Constructs a proof of decreasing along a well founded relation, by simplifying, then applying lexicographic order lemmas and finally using `ts` to solve the base case. If it fails, it prints a message to help the user diagnose an ill-founded recursive definition.

tactic

`get_elem_tactic`

get_elem_tactic is the tactic automatically called by the notation `arr[i]` to prove any side conditions that arise when constructing the term (e.g. the index is in bounds of the array). It just delegates to **get_elem_tactic_extensible** and gives a diagnostic error message otherwise; users are encouraged to extend **get_elem_tactic_extensible** instead of this tactic.

tactic

get_elem_tactic_trivial

get_elem_tactic_trivial has been deprecated in favour of **get_elem_tactic_extensible**.

13.5.20. Debugging Utilities

tactic

sorry

The **sorry** tactic is a temporary placeholder for an incomplete tactic proof, closing the main goal using **exact sorry**.

This is intended for stubbing-out incomplete parts of a proof while still having a syntactically correct proof skeleton. Lean will give a warning whenever a proof uses **sorry**, so you aren't likely to miss it, but you can double check if a theorem depends on **sorry** by looking for `sorryAx` in the output of the **#print axioms my_thm** command, the axiom used by the implementation of **sorry**.

tactic

admit

admit is a synonym for **sorry**.

tactic

dbg_trace

dbg_trace `"foo"` prints `foo` when elaborated. Useful for debugging tactic control flow:

```
example : False ∨ True := by
  first
  | apply Or.inl; trivial; dbg_trace "left"
  | apply Or.inr; trivial; dbg_trace "right"
```

tactic

trace_state

trace_state displays the current state in the info view.

tactic

trace

`trace msg` displays `msg` in the info view.

13.5.21. Other

tactic

trivial

`trivial` tries different simple tactics (e.g., `rfl`, **contradiction**, ...) to close the current goal. You can use the command **macro_rules** to extend the set of tactics used. Example:

```
macro_rules | `(tactic| trivial) => `(tactic| simp)
```

tactic

solve

Similar to **first**, but succeeds only if one the given tactics solves the current goal.

tactic

and_intros

and_intros applies `And.intro` until it does not make progress.

tactic

infer_instance

infer_instance is an abbreviation for **exact** `inferInstance`. It synthesizes a value of any target type by typeclass inference.

tactic

expose_names

expose_names renames all inaccessible variables with accessible names, making them available for reference in generated tactics. However, this renaming introduces machine-generated names that are not fully under user control. **expose_names** is primarily intended as

a preamble for auto-generated end-game tactic scripts. It is also useful as an alternative to `set_option tactic.hygienic false`. If explicit control over renaming is needed in the middle of a tactic script, consider using structured tactic scripts with **`match .. with`**, **`induction .. with`**, or intro with explicit user-defined names, as well as tactics such as next, case, and rename_i.

tactic

unhygienic

`unhygienic tacs` runs `tacs` with name hygiene disabled. This means that tactics that would normally create inaccessible names will instead make regular variables. **Warning**: Tactics may change their variable naming strategies at any time, so code that depends on autogenerated names is brittle. Users should try not to use *unhygienic* if possible.

```
example : ∀ x : Nat, x = x := by unhygienic
  intro             -- x would normally be intro'd as
inaccessible
  exact Eq.refl x   -- refer to x
```

tactic

run_tac

The run_tac doSeq tactic executes code in `TacticM Unit`.

13.6. Targeted Rewriting with conv

The conv, or conversion, tactic allows targeted rewriting within a goal. The argument to conv is written in a separate language that interoperates with the main tactic language; it features commands to navigate to specific subterms within the goal along with commands that allow these subterms to be rewritten. conv is useful when rewrites should only be applied in part of a goal (e.g. only on one side of an equality), rather than across the board, or when rewrites should be applied underneath a binder that prevents tactics like rw from accessing the term.

The conversion tactic language is very similar to the main tactic language: it uses the same proof states, tactics work primarily on the main goal and may either fail or succeed with a sequence of new goals, and macro expansion is interleaved with tactic execution. Unlike the main tactic language, in which tactics are intended to eventually solve goals, the conv tactic is used to *change* a goal so that it becomes amenable to further processing in the main tactic language. Goals that are intended to be rewritten with conv are shown with a vertical bar instead of a turnstile.

tactic

conv

`conv => ...` allows the user to perform targeted rewriting on a goal or hypothesis, by focusing on particular subexpressions.

See https://lean-lang.org/theorem_proving_in_lean4/conv.html for more details.

Basic forms:

- `conv => cs` will rewrite the goal with conv tactics `cs`.
- `conv at h => cs` will rewrite hypothesis `h`.
- `conv in pat => cs` will rewrite the first subexpression matching `pat` (see `pattern`).

▼ *Example: Navigation and Rewriting with* **conv**

In this example, there are multiple instances of addition, and **rw** would by default rewrite the first instance that it encounters. Using **conv** to navigate to the specific subterm before rewriting leaves **rw** no choice but to rewrite the correct term.

```
example (x y z : Nat) : x + (y + z) = (x + z) + y := by
  conv =>
    lhs
    arg 2
    rw [Nat.add_comm]
  rw [Nat.add_assoc]
```

▼ *Example: Rewriting Under Binders with* **conv**

In this example, addition occurs under binders, so **rw** can't be used. However, after using **conv** to navigate to the function body, it succeeds. The nested use of **conv** causes control to return to the current position in the term after performing further conversions on one of its subterms. Because the goal is a reflexive equation after rewriting, **conv** automatically closes it.

```
example :
    (fun (x y z : Nat) =>
      x + (y + z))
    =
    (fun x y z =>
      (z + x) + y)
  := by
  conv =>
    lhs
    intro x y z
    conv =>
      arg 2
      rw [Nat.add_comm]
```

```
rw [← Nat.add_assoc]
arg 1
rw [Nat.add_comm]
```

13.6.1. Control Structures

conv tactic

`first`

`first | conv | ...` runs each `conv` until one succeeds, or else fails.

conv tactic

`try`

`try` `tac` runs `tac` and succeeds even if `tac` failed.

conv tactic

`<;>`

`tac <;> tac'` runs `tac` on the main goal and `tac'` on each produced goal, concatenating all goals produced by `tac'`.

conv tactic

`repeat`

`repeat` `convs` runs the sequence `convs` repeatedly until it fails to apply.

conv tactic

`skip`

`skip` does nothing.

conv tactic

`{ ... }`

`{ convs }` runs the list of *convs* on the current target, and any subgoals that remain are trivially closed by **`skip`**.

conv tactic

`( ... )`

(*convs*) runs the *convs* in sequence on the current list of targets. This is pure grouping with no added effects.

conv tactic

done

done succeeds iff there are no goals remaining.

13.6.2. Goal Selection

conv tactic

all_goals

`all_goals tac` runs `tac` on each goal, concatenating the resulting goals, if any.

conv tactic

any_goals

`any_goals tac` applies the tactic `tac` to every goal, and succeeds if at least one application succeeds.

conv tactic

case ... => ...

- **case** `tag =>` `tac` focuses on the goal with case name `tag` and solves it using `tac`, or else fails.
- **case** `tag` x_1 `...` x_n `=>` `tac` additionally renames the n most recent hypotheses with inaccessible names to the given names.
- **case** tag_1 `|` tag_2 `=>` `tac` is equivalent to (**case** tag_1 `=>` `tac`)**;** (**case** tag_2 `=>` `tac`).

conv tactic

case' ... => ...

case' is similar to the **case** `tag =>` `tac` tactic, but does not ensure the goal has been solved after applying `tac`, nor admits the goal if `tac` failed. Recall that **case** closes the goal using **sorry** when `tac` fails, and the tactic execution is not interrupted.

conv tactic

next ... => ...

`next => tac` focuses on the next goal and solves it using `tac`, or else fails. **`next x₁ ... xₙ => tac`** additionally renames the n most recent hypotheses with inaccessible names to the given names.

conv tactic

`focus`

`focus tac` focuses on the main goal, suppressing all other goals, and runs `tac` on it. Usually `· tac`, which enforces that the goal is closed by `tac`, should be preferred.

conv tactic

`· ...`

`· conv` focuses on the main conv goal and tries to solve it using `s`.

conv tactic

`· ...`

`· conv` focuses on the main conv goal and tries to solve it using `s`.

conv tactic

`fail_if_success`

`fail_if_success t` fails if the tactic `t` succeeds.

13.6.3. Navigation

conv tactic

`lhs`

Traverses into the left subterm of a binary operator.

In general, for an n-ary operator, it traverses into the second to last argument. It is a synonym for *`arg -2`*.

conv tactic

`rhs`

Traverses into the right subterm of a binary operator.

In general, for an n-ary operator, it traverses into the last argument. It is a synonym for `arg -1`.

conv tactic

`fun`

Traverses into the function of a (unary) function application. For example, `| f a b` turns into `| f a`. (Use `arg 0` to traverse into `f`.)

conv tactic

`congr`

Performs one step of "congruence", which takes a term and produces subgoals for all the function arguments. For example, if the target is `f x y` then `congr` produces two subgoals, one for `x` and one for `y`.

conv tactic

`arg [@]i`

- `arg i` traverses into the `i`'th argument of the target. For example if the target is `f a b c d` then `arg 1` traverses to `a` and `arg 3` traverses to `c`. The index may be negative; `arg -1` traverses into the last argument, `arg -2` into the second-to-last argument, and so on.
- `arg @i` is the same as `arg i` but it counts all arguments instead of just the explicit arguments.
- `arg 0` traverses into the function. If the target is `f a b c d`, `arg 0` traverses into `f`.

syntax

Arguments to **`enter`**

```
enterArg ::= ...
    | num

enterArg ::= ...
    | @num

enterArg ::= ...
    | ident
```

conv tactic

`enter`

`enter [arg, ...]` is a compact way to describe a path to a subterm. It is a shorthand for other conv tactics as follows:

- `enter [i]` is equivalent to `arg i`.
- `enter [@i]` is equivalent to `arg @i`.
- `enter [x]` (where x is an identifier) is equivalent to `ext x`. For example, given the target `f (g a (fun x => x b))`, `enter [1, 2, x, 1]` will traverse to the subterm b.

conv tactic

`pattern`

- `pattern pat` traverses to the first subterm of the target that matches `pat`.
- `pattern (occs := *) pat` traverses to every subterm of the target that matches `pat` which is not contained in another match of `pat`. It generates one subgoal for each matching subterm.
- `pattern (occs := 1 2 4) pat` matches occurrences `1, 2, 4` of `pat` and produces three subgoals. Occurrences are numbered left to right from the outside in.

Note that skipping an occurrence of `pat` will traverse inside that subexpression, which means it may find more matches and this can affect the numbering of subsequent pattern matches. For example, if we are searching for `f _` in `f (f a) = f b`:

- `occs := 1 2` (and `occs := *`) returns `| f (f a)` and `| f b`
- `occs := 2` returns `| f a`
- `occs := 2 3` returns `| f a` and `| f b`
- `occs := 1 3` is an error, because after skipping `f b` there is no third match.

conv tactic

`ext`

`ext` x traverses into a binder (a `fun` *x* `=>` *e* or `∀` *x*`,` *e* expression) to target *e*, introducing name x in the process.

conv tactic

`args`

`args` traverses into all arguments. Synonym for congr.

conv tactic

left

left traverses into the left argument. Synonym for `lhs`.

conv tactic

right

right traverses into the right argument. Synonym for `rhs`.

conv tactic

intro

intro traverses into binders. Synonym for **ext**.

13.6.4. Changing the Goal

13.6.4.1. Reduction

conv tactic

whnf

Reduces the target to Weak Head Normal Form. This reduces definitions in "head position" until a constructor is exposed. For example, `List.map f [a, b, c]` weak head normalizes to `f a :: List.map f [b, c]`.

conv tactic

reduce

Puts term in normal form, this tactic is meant for debugging purposes only.

conv tactic

zeta

Expands let-declarations and let-variables.

conv tactic

delta

delta id1 id2 ... unfolds all occurrences of id1, id2, ... in the target. Like the **delta** tactic, this ignores any definitional equations and uses primitive delta-reduction instead, which may result in leaking implementation details. Users should prefer **unfold** for unfolding definitions.

conv tactic

unfold

- **unfold** id unfolds all occurrences of definition id in the target.
- **unfold** id1 id2 ... is equivalent to **unfold** id1; **unfold** id2; ....

Definitions can be either global or local definitions.

For non-recursive global definitions, this tactic is identical to **delta**. For recursive global definitions, it uses the "unfolding lemma" id.eq_def, which is generated for each recursive definition, to unfold according to the recursive definition given by the user. Only one level of unfolding is performed, in contrast to **simp only** [id], which unfolds definition id recursively.

This is the conv version of the **unfold** tactic.

13.6.4.2. Simplification

conv tactic

simp

simp [thm] performs simplification using thm and marked @[**simp**] lemmas. See the **simp** tactic for more information.

conv tactic

dsimp

dsimp is the definitional simplifier in conv-mode. It differs from **simp** in that it only applies theorems that hold by reflexivity.

Examples:

```
example (a : Nat): (0 + 0) = a - a := by
  conv =>
    lhs
    dsimp
    rw [← Nat.sub_self a]
```

conv tactic

`simp_match`

`simp_match` simplifies match expressions. For example,

```
match [a, b] with
| [] => 0
| hd :: tl => hd
```

simplifies to a.

13.6.4.3. Rewriting

conv tactic

`change`

`change t'` replaces the target `t` with `t'`, assuming `t` and `t'` are definitionally equal.

conv tactic

`rewrite`

`rw [thm]` rewrites the target using `thm`. See the `rw` tactic for more information.

conv tactic

`rw`

`rw [rules]` applies the given list of rewrite rules to the target. See the `rw` tactic for more information.

conv tactic

`erw`

`erw [rules]` is a shorthand for `rw (transparency := .default) [rules]`. This does rewriting up to unfolding of regular definitions (by comparison to regular `rw` which only unfolds `@[reducible]` definitions).

conv tactic

`apply`

The `apply thm` conv tactic is the same as `apply thm` the tactic. There are no restrictions on `thm`, but strange results may occur if `thm` cannot be reasonably interpreted as proving one equality from a list of others.

13.6.5. Nested Tactics

tactic

`conv'`

Executes the given conv block without converting regular goal into a `conv` goal.

conv tactic

`tactic`

Focuses, converts the `conv` goal `⊢ lhs` into a regular goal `⊢ lhs = rhs`, and then executes the given tactic block.

conv tactic

`tactic'`

Executes the given tactic block without converting `conv` goal into a regular goal.

tactic

`conv'`

Executes the given conv block without converting regular goal into a `conv` goal.

conv tactic

`conv => ...`

`conv => cs` runs `cs` in sequence on the target `t`, resulting in `t'`, which becomes the new target subgoal.

13.6.6. Debugging Utilities

conv tactic

`trace_state`

`trace_state` prints the current goal state.

13.6.7. Other

conv tactic

`rfl`

`rfl` closes one conv goal "trivially", by using reflexivity (that is, no rewriting).

conv tactic

`norm_cast`

`norm_cast` tactic in conv mode.

13.7. Naming Bound Variables

When the simplifier or the `rw` tactic introduce new binding forms such as function parameters, they select a name for the bound variable based on the one in the statement of the rewrite rule being applied. This name is made unique if necessary. In some situations, such as preprocessing definitions for termination proofs that use well-founded recursion, the names that appear in termination proof obligations should be the corresponding names written in the original function definition.

The `binderNameHint` gadget can be used to indicate that a bound variable should be named according to the variables bound in some other term. By convention, the term `()` is used to indicate that a name should *not* be taken from the original definition.

def

```
binderNameHint.{u, v, w} {α : Sort u} {β : Sort v} {γ : Sort w} (v
: α)
  (binder : β) (e : γ) : γ
```

The expression `binderNameHint v binder e` defined to be *e*.

If it is used on the right-hand side of an equation that is used for rewriting by `rw` or `simp`, and *v* is a local variable, and *binder* is an expression that (after beta-reduction) is a binder (`fun w => …` or `∀ w, …`), then it will rename *v* to the name used in that binder, and remove the `binderNameHint`.

A typical use of this gadget would be as follows; the gadget ensures that after rewriting, the local variable is still `name`, and not `x`:

```
theorem all_eq_not_any_not (l : List α) (p : α → Bool) :
    l.all p = !l.any fun x => binderNameHint x p (!p x) := sorry

example (names : List String) : names.all (fun name =>
"Waldo".isPrefixOf name) = true := by
  rw [all_eq_not_any_not]
  -- ⊢ (!names.any fun name => !"Waldo".isPrefixOf name) = true
```

If *binder* is not a binder, then the name of *v* attains a macro scope. This only matters when the resulting term is used in a non-hygienic way, e.g. in termination proofs for well-founded recursion.

This gadget is supported by

- **simp**, **dsimp** and **rw** in the right-hand-side of an equation
- **simp** in the assumptions of congruence rules

It is ineffective in other positions (hyptheses of rewrite rules) or when used by other tactics (e.g. **apply**).

13.8. Custom Tactics

Tactics are productions in the syntax category `tactic`. Given the syntax of a tactic, the tactic interpreter is responsible for carrying out actions in the tactic monad `TacticM`, which is a wrapper around Lean's term elaborator that keeps track of the additional state needed to execute tactics. A custom tactic consists of an extension to the `tactic` category along with either:

- a macro that translates the new syntax into existing syntax, or
- an elaborator that carries out `TacticM` actions to implement the tactic.

13.8.1. Tactic Macros

The easiest way to define a new tactic is as a macro that expands into already-existing tactics. Macro expansion is interleaved with tactic execution. The tactic interpreter first expands tactic macros just before they are to be interpreted. Because tactic macros are not fully expanded prior to running a tactic script, they can use recursion; as long as the recursive occurrence of the macro syntax is beneath a tactic that can be executed, there will not be an infinite chain of expansion.

▼ *Example: Recursive tactic macro*

This recursive implementation of a tactic akin to **repeat** is defined via macro expansion. When the argument `$t` fails, the recursive occurrence of **rep** is never invoked, and is thus never macro expanded.

```
syntax "rep" tactic : tactic
macro_rules
  | `(tactic|rep $t) =>
  `(tactic|
    first
      | $t; rep $t
      | skip)

example : 0 ≤ 4 := by
  rep (apply Nat.le.step)
```

```
  apply Nat.le.refl
```

Like other Lean macros, tactic macros are hygienic. References to global names are resolved when the macro is defined, and names introduced by the tactic macro cannot capture names from its invocation site.

When defining a tactic macro, it's important to specify that the syntax being matched or constructed is for the syntax category `tactic`. Otherwise, the syntax will be interpreted as that of a term, which will match against or construct an incorrect AST for tactics.

13.8.1.1. Extensible Tactic Macros

Because macro expansion can fail, multiple macros can match the same syntax, allowing backtracking. Tactic macros take this further: even if a tactic macro expands successfully, if the expansion fails when interpreted, the tactic interpreter will attempt the next expansion. This is used to make a number of Lean's built-in tactics extensible—new behavior can be added to a tactic by adding a `macro_rules` declaration.

▼ *Example: Extending* `trivial`

The `trivial`, which is used by many other tactics to quickly dispatch subgoals that are not worth bothering the user with, is designed to be extended through new macro expansions. Lean's default `trivial` can't solve `IsEmpty []` goals:

```
def IsEmpty (xs : List α) : Prop :=
  ¬ xs ≠ []

example (α : Type u) : IsEmpty (α := α) [] := by trivial
```

The error message is an artifact of `trivial` trying `assumption` last. Adding another expansion allows `trivial` to take care of these goals:

```
def emptyIsEmpty : IsEmpty (α := α) [] := by simp [IsEmpty]

macro_rules | `(tactic|trivial) => `(tactic|exact emptyIsEmpty)

example (α : Type u) : IsEmpty (α := α) [] := by
  trivial
```

▼ *Example: Expansion Backtracking*

Macro expansion can induce backtracking when the failure arises from any part of the expanded syntax. An infix version of `first` can be defined by providing multiple expansions in separate `macro_rules` declarations:

```
syntax tactic "<|||>" tactic : tactic
macro_rules
  | `(tactic|$t1 <|||> $t2) => pure t1
macro_rules
  | `(tactic|$t1 <|||> $t2) => pure t2

example : 2 = 2 := by
  rfl <|||> apply And.intro

example : 2 = 2 := by
  apply And.intro <|||> rfl
```

Multiple `macro_rules` declarations are needed because each defines a pattern-matching function that will always take the first matching alternative. Backtracking is at the granularity of `macro_rules` declarations, not their individual cases.

14. Functors, Monads and do-Notation

The type classes `Functor`, `Applicative`, and `Monad` provide fundamental tools for functional programming. An introduction to programming with these abstractions is available in *Functional Programming in Lean*. While they are inspired by the concepts of functors and monads in category theory, the versions used for programming are more limited. The type classes in Lean's standard library represent the concepts as used for programming, rather than the general mathematical definition.

Instances of `Functor` allow an operation to be applied consistently throughout some polymorphic context. Examples include transforming each element of a list by applying a function and creating new `IO` actions by arranging for a pure function to be applied to the result of an existing `IO` action. Instances of `Monad` allow side effects with data dependencies to be encoded; examples include using a tuple to simulate mutable state, a sum type to simulate exceptions, and representing actual side effects with `IO`. `Applicative` functors occupy a middle ground: like monads, they allow functions computed with effects to be applied to arguments that are computed with effects, but they do not allow sequential data dependencies where the output of an effect forms an input into another effectful operation.

The additional type classes `Pure`, `Bind`, `SeqLeft`, `SeqRight`, and `Seq` capture individual operations from `Applicative` and `Monad`, allowing them to be overloaded and used with types that are not necessarily `Applicative` functors or `Monad`s. The `Alternative` type class describes applicative functors that additionally have some notion of failure and recovery.

type class

```
Functor.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)
```

A functor in the sense used in functional programming, which means a function `f : Type u → Type v` has a way of mapping a function over its contents. This `map` operator is written `<$>`, and overloaded via `Functor` instances.

This map function should respect identity and function composition. In other words, for all terms `v : f α`, it should be the case that:

- `id <$> v = v`
- For all functions `h : β → γ` and `g : α → β`, `(h ∘ g) <$> v = h <$> g <$> v`

While all Functor instances should live up to these requirements, they are not required to *prove* that they do. Proofs may be required or provided via the LawfulFunctor class.

Assuming that instances are lawful, this definition corresponds to the category-theoretic notion of functor in the special case where the category is the category of types and functions between them.

Instance Constructor

Functor.mk.{u, v}

Methods

`map : {α β : Type u} → (α → β) → f α → f β`

Applies a function inside a functor. This is used to overload the <$> operator.

When mapping a constant function, use Functor.mapConst instead, because it may be more efficient.

Conventions for notations in identifiers:

- The recommended spelling of <$> in identifiers is *map*.

`mapConst : {α β : Type u} → α → f β → f α`

Mapping a constant function.

Given `a : α` and `v : f α`, `mapConst a v` is equivalent to `Function.const _ a <$> v`. For some functors, this can be implemented more efficiently; for all other functors, the default implementation may be used.

type class

`Pure.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)`

The pure function is overloaded via Pure instances.

Pure is typically accessed via Monad or Applicative instances.

Instance Constructor

`Pure.mk.{u, v}`

Methods

`pure : {α : Type u} → α → f α`

Given `a : α`, then `pure a : f α` represents an action that does nothing and returns `a`.

Examples:

- `(pure "hello" : Option String) = some "hello"`
- `(pure "hello" : Except (Array String) String) = Except.ok "hello"`
- `(pure "hello" : StateM Nat String).run 105 = ("hello", 105)`

type class

`Seq.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)`

The `<*>` operator is overloaded using the function `Seq.seq`.

While `<$>` from the class `Functor` allows an ordinary function to be mapped over its contents, `<*>` allows a function that's "inside" the functor to be applied. When thinking about `f` as possible side effects, this captures evaluation order: `seq` arranges for the effects that produce the function to occur prior to those that produce the argument value.

For most applications, `Applicative` or `Monad` should be used rather than `Seq` itself.

Instance Constructor

`Seq.mk.{u, v}`

Methods

`seq : {α β : Type u} → f (α → β) → (Unit → f α) → f β`

The implementation of the `<*>` operator.

In a monad, `mf <*> mx` is the same as `do let f ← mf; x ← mx; pure (f x)`: it evaluates the function first, then the argument, and applies one to the other.

To avoid surprising evaluation semantics, `mx` is taken "lazily", using a `Unit → f α` function.

Conventions for notations in identifiers:

- The recommended spelling of <*> in identifiers is *seq*.

type class

`SeqLeft.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)`

The <* operator is overloaded using `seqLeft`.

When thinking about *f* as potential side effects, <* evaluates first the left and then the right argument for their side effects, discarding the value of the right argument and returning the value of the left argument.

For most applications, `Applicative` or `Monad` should be used rather than `SeqLeft` itself.

Instance Constructor

`SeqLeft.mk.{u, v}`

Methods

`seqLeft : {α β : Type u} → f α → (Unit → f β) → f α`

Sequences the effects of two terms, discarding the value of the second. This function is usually invoked via the <* operator.

Given *x* : *f α* and *y* : *f β*, *x* <* *y* runs *x*, then runs *y*, and finally returns the result of *x*.

The evaluation of the second argument is delayed by wrapping it in a function, enabling "short-circuiting" behavior from *f*.

Conventions for notations in identifiers:

- The recommended spelling of <* in identifiers is *seqLeft*.

type class

`SeqRight.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)`

The *> operator is overloaded using `seqRight`.

When thinking about *f* as potential side effects, *> evaluates first the left and then the right argument for their side effects, discarding the value of the left argument and returning the value of the right argument.

For most applications, `Applicative` or `Monad` should be used rather than `SeqLeft` itself.

Instance Constructor

`SeqRight.mk.{u, v}`

Methods

`seqRight : {α β : Type u} → f α → (Unit → f β) → f β`

Sequences the effects of two terms, discarding the value of the first. This function is usually invoked via the `*>` operator.

Given $x : f\ \alpha$ and $y : f\ \beta$, $x\ *>\ y$ runs x, then runs y, and finally returns the result of y.

The evaluation of the second argument is delayed by wrapping it in a function, enabling "short-circuiting" behavior from f.

Conventions for notations in identifiers:

- The recommended spelling of `*>` in identifiers is *seqRight*.

type class

`Applicative.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)`

An applicative functor is more powerful than a `Functor`, but less powerful than a `Monad`.

Applicative functors capture sequencing of effects with the `<*>` operator, overloaded as `seq`, but not data-dependent effects. The results of earlier computations cannot be used to control later effects.

Applicative functors should satisfy four laws. Instances of `Applicative` are not required to prove that they satisfy these laws, which are part of the `LawfulApplicative` class.

Instance Constructor

`Applicative.mk.{u, v}`

Extends

☑ `Functor f` ☑ `Pure f` ☑ `Seq f` ☑ `SeqLeft f` ☑ `SeqRight f`

Methods

`map : {α β : Type u} → (α → β) → f α → f β`

Inherited from `Functor f` > `Pure f` > `Seq f` > `SeqLeft f` > `SeqRight f`

`mapConst : {α β : Type u} → α → f β → f α`

Inherited from Functor f > Pure f > Seq f > SeqLeft f > SeqRight f

```
pure : {α : Type u} → α → f α
```

Inherited from Functor f > Pure f > Seq f > SeqLeft f > SeqRight f

```
seq : {α β : Type u} → f (α → β) → (Unit → f α) → f β
```

Inherited from Functor f > Pure f > Seq f > SeqLeft f > SeqRight f

```
seqLeft : {α β : Type u} → f α → (Unit → f β) → f α
```

Inherited from Functor f > Pure f > Seq f > SeqLeft f > SeqRight f

```
seqRight : {α β : Type u} → f α → (Unit → f β) → f β
```

Inherited from Functor f > Pure f > Seq f > SeqLeft f > SeqRight f

▼ *Example: Lists with Lengths as Applicative Functors*

The structure LenList pairs a list with a proof that it has the desired length. As a consequence, its zipWith operator doesn't require a fallback in case the lengths of its inputs differ.

```
structure LenList (length : Nat) (α : Type u) where
  list : List α
  lengthOk : list.length = length

def LenList.head (xs : LenList (n + 1) α) : α :=
  xs.list.head <| by
    intro h
    cases xs
    simp_all
    subst_eqs

def LenList.tail (xs : LenList (n + 1) α) : LenList n α :=
  match xs with
  | ⟨_ :: xs', _⟩ => ⟨xs', by simp_all⟩

def LenList.map (f : α → β) (xs : LenList n α) : LenList n β
where
  list := xs.list.map f
  lengthOk := by
    cases xs
    simp [List.length_map, *]

def LenList.zipWith (f : α → β → γ)
    (xs : LenList n α) (ys : LenList n β) :
    LenList n γ where
  list := xs.list.zipWith f ys.list
  lengthOk := by
```

```
    cases xs; cases ys
    simp [List.length_zipWith, *]
```

The well-behaved Applicative instance applies functions to arguments element-wise. Because Applicative extends Functor, a separate Functor instance is not necessary, and map can be defined as part of the Applicative instance.

```
instance : Applicative (LenList n) where
  map := LenList.map
  pure x := {
    list := List.replicate n x
    lengthOk := List.length_replicate
  }
  seq {α β} fs xs := fs.zipWith (· ·) (xs ())
```

The well-behaved Monad instance takes the diagonal of the results of applying the function:

```
@[simp]
theorem LenList.list_length_eq (xs : LenList n α) :
    xs.list.length = n := by
  cases xs
  simp [*]

def LenList.diagonal (square : LenList n (LenList n α)) : LenList
n α :=
  match n with
  | 0 => ⟨[], rfl⟩
  | n' + 1 => {
    list :=
      square.head.head :: (square.tail.map
(·.tail)).diagonal.list
    lengthOk := by simp
  }
```

type class

```
Alternative.{u, v} (f : Type u → Type v) : Type (max (u + 1) v)
```

An Alternative functor is an Applicative functor that can "fail" or be "empty" and a binary operation <|> that “collects values” or finds the “left-most success”.

Important instances include

- Option, where `failure := none` and <|> returns the left-most some.
- Parser combinators typically provide an Applicative instance for error-handling and backtracking.

Error recovery and state can interact subtly. For example, the implementation of `Alternative` for `OptionT (StateT σ Id)` keeps modifications made to the state while recovering from failure, while `StateT σ (OptionT Id)` discards them.

Instance Constructor

`Alternative.mk.{u, v}`

Extends

- ☑ `Applicative f`

Methods

`map : {α β : Type u} → (α → β) → f α → f β`

Inherited from `Applicative f`

`mapConst : {α β : Type u} → α → f β → f α`

Inherited from `Applicative f`

`pure : {α : Type u} → α → f α`

Inherited from `Applicative f`

`seq : {α β : Type u} → f (α → β) → (Unit → f α) → f β`

Inherited from `Applicative f`

`seqLeft : {α β : Type u} → f α → (Unit → f β) → f α`

Inherited from `Applicative f`

`seqRight : {α β : Type u} → f α → (Unit → f β) → f β`

Inherited from `Applicative f`

`failure : {α : Type u} → f α`

Produces an empty collection or recoverable failure. The < | > operator collects values or recovers from failures. See `Alternative` for more details.

`orElse : {α : Type u} → f α → (Unit → f α) → f α`

Depending on the `Alternative` instance, collects values or recovers from `failures` by returning the leftmost success. Can be written using the < | > operator syntax.

type class

`Bind.{u, v} (m : Type u → Type v) : Type (max (u + 1) v)`

The >>= operator is overloaded via instances of `bind`.

`Bind` is typically used via `Monad`, which extends it.

Instance Constructor

`Bind.mk.{u, v}`

Methods

`bind : {α β : Type u} → m α → (α → m β) → m β`

Sequences two computations, allowing the second to depend on the value computed by the first.

If `x : m α` and `f : α → m β`, then `x >>= f : m β` represents the result of executing `x` to get a value of type `α` and then passing it to `f`.

Conventions for notations in identifiers:

- The recommended spelling of >>= in identifiers is `bind`.

type class

`Monad.{u, v} (m : Type u → Type v) : Type (max (u + 1) v)`

Monads are an abstraction of sequential control flow and side effects used in functional programming. Monads allow both sequencing of effects and data-dependent effects: the values that result from an early step may influence the effects carried out in a later step.

The `Monad` API may be used directly. However, it is most commonly accessed through **do**-notation.

Most `Monad` instances provide implementations of `pure` and `bind`, and use default implementations for the other methods inherited from `Applicative`. Monads should satisfy certain laws, but instances are not required to prove this. An instance of `LawfulMonad` expresses that a given monad's operations are lawful.

Instance Constructor

`Monad.mk.{u, v}`

Extends

☑ `Applicative m` ☑ `Bind m`

Methods

`map : {α β : Type u} → (α → β) → m α → m β`

Inherited from `Applicative m` > `Bind m`

`mapConst : {α β : Type u} → α → m β → m α`

Inherited from Applicative m > Bind m

```
pure : {α : Type u} → α → m α
```

Inherited from Applicative m > Bind m

```
seq : {α β : Type u} → m (α → β) → (Unit → m α) → m β
```

Inherited from Applicative m > Bind m

```
seqLeft : {α β : Type u} → m α → (Unit → m β) → m α
```

Inherited from Applicative m > Bind m

```
seqRight : {α β : Type u} → m α → (Unit → m β) → m β
```

Inherited from Applicative m > Bind m

```
bind : {α β : Type u} → m α → (α → m β) → m β
```

Inherited from Applicative m > Bind m

14.1. Laws

Having map, pure, seq, and bind operators with the appropriate types is not really sufficient to have a functor, applicative functor, or monad. These operators must additionally satisfy certain axioms, which are often called the *laws* of the type class.

For a functor, the map operation must preserve identity and function composition. In other words, given a purported Functor f, for all x : f α:

- id <$> x = x, and
- for all function g and h, (h ∘ g) <$> x = h <$> g <$> x.

Instances that violate these assumptions can be very surprising! Additionally, because Functor includes mapConst to enable instances to provide a more efficient implementation, a lawful functor's mapConst should be equivalent to its default implementation.

The Lean standard library does not require proofs of these properties in every instance of Functor. Nonetheless, if an instance violates them, then it should be considered a bug. When proofs of these properties are necessary, an instance implicit parameter of type LawfulFunctor f can be used. The LawfulFunctor class includes the necessary proofs.

type class

```
LawfulFunctor.{u, v} (f : Type u → Type v) [Functor f] : Prop
```

A functor satisfies the functor laws.

The Functor class contains the operations of a functor, but does not require that instances prove they satisfy the laws of a functor. A LawfulFunctor instance includes proofs that the laws are satisfied. Because Functor instances may provide optimized implementations of mapConst, LawfulFunctor instances must also prove that the optimized implementation is equivalent to the standard implementation.

Instance Constructor

LawfulFunctor.mk.{u, v}

Methods

map_const : ∀ {α β : Type u}, Functor.mapConst = Functor.map ∘ Function.const β

The mapConst implementation is equivalent to the default implementation.

id_map : ∀ {α : Type u} (x : f α), id <$> x = x

The map implementation preserves identity.

comp_map : ∀ {α β γ : Type u} (g : α → β) (h : β → γ) (x : f α), (h ∘ g) <$> x = h <$> g <$> x

The map implementation preserves function composition.

In addition to proving that the potentially-optimized SeqLeft.seqLeft and SeqRight.seqRight operations are equivalent to their default implementations, Applicative functors f must satisfy four laws.

type class

LawfulApplicative.{u, v} (f : Type u → Type v) [Applicative f] : Prop

An applicative functor satisfies the laws of an applicative functor.

The Applicative class contains the operations of an applicative functor, but does not require that instances prove they satisfy the laws of an applicative functor. A LawfulApplicative instance includes proofs that the laws are satisfied.

Because Applicative instances may provide optimized implementations of seqLeft and seqRight, LawfulApplicative instances must also prove that the optimized implementation is equivalent to the standard implementation.

Instance Constructor

LawfulApplicative.mk.{u, v}

Extends

☑ LawfulFunctor *f*

Methods

map_const : ∀ {α β : Type u}, Functor.mapConst = Functor.map ∘ Function.const β

Inherited from LawfulFunctor *f*

id_map : ∀ {α : Type u} (x : f α), id <$> x = x

Inherited from LawfulFunctor *f*

comp_map : ∀ {α β γ : Type u} (g : α → β) (h : β → γ) (x : f α), (h ∘ g) <$> x = h <$> g <$> x

Inherited from LawfulFunctor *f*

seqLeft_eq : ∀ {α β : Type u} (x : f α) (y : f β), x <* y = Function.const β <$> x <*> y

seqLeft is equivalent to the default implementation.

seqRight_eq : ∀ {α β : Type u} (x : f α) (y : f β), x *> y = Function.const α id <$> x <*> y

seqRight is equivalent to the default implementation.

pure_seq : ∀ {α β : Type u} (g : α → β) (x : f α), pure g <*> x = g <$> x

pure before seq is equivalent to Functor.map.

This means that pure really is pure when occurring immediately prior to seq.

map_pure : ∀ {α β : Type u} (g : α → β) (x : α), g <$> pure x = pure (g x)

Mapping a function over the result of pure is equivalent to applying the function under pure.

This means that pure really is pure with respect to Functor.map.

seq_pure : ∀ {α β : Type u} (g : f (α → β)) (x : α), g <*> pure x = (fun h => h x) <$> g

pure after seq is equivalent to Functor.map.

This means that pure really is pure when occurring just after seq.

seq_assoc : ∀ {α β γ : Type u} (x : f α) (g : f (α → β)) (h : f (β → γ)), h <*> (g <*> x) = Function.comp <$> h <*> g <*> x

`seq` is associative.

Changing the nesting of `seq` calls while maintaining the order of computations results in an equivalent computation. This means that `seq` is not doing any more than sequencing.

The monad laws specify that `pure` followed by `bind` should be equivalent to function application (that is, `pure` has no effects), that `bind` followed by `pure` around a function application is equivalent to `map`, and that `bind` is associative.

type class

```
LawfulMonad.{u, v} (m : Type u → Type v) [Monad m] : Prop
```

Lawful monads are those that satisfy a certain behavioral specification. While all instances of `Monad` should satisfy these laws, not all implementations are required to prove this.

`LawfulMonad.mk'` is an alternative constructor that contains useful defaults for many fields.

Instance Constructor

`LawfulMonad.mk.{u, v}`

Extends

- ☑ `LawfulApplicative m`

Methods

`map_const : ∀ {α β : Type u}, Functor.mapConst = Functor.map ∘ Function.const β`

Inherited from `LawfulApplicative m`

`id_map : ∀ {α : Type u} (x : m α), id <$> x = x`

Inherited from `LawfulApplicative m`

`comp_map : ∀ {α β γ : Type u} (g : α → β) (h : β → γ) (x : m α), (h ∘ g) <$> x = h <$> g <$> x`

Inherited from `LawfulApplicative m`

`seqLeft_eq : ∀ {α β : Type u} (x : m α) (y : m β), x <* y = Function.const β <$> x <*> y`

Inherited from `LawfulApplicative m`

`seqRight_eq : ∀ {α β : Type u} (x : m α) (y : m β), x *> y = Function.const α id <$> x <*> y`

Inherited from `LawfulApplicative m`

`pure_seq : ∀ {α β : Type u} (g : α → β) (x : m α), pure g <*> x = g <$> x`

Inherited from `LawfulApplicative m`

```
map_pure : ∀ {α β : Type u} (g : α → β) (x : α), g <$> pure x = pure (g x)
```

Inherited from LawfulApplicative m

```
seq_pure : ∀ {α β : Type u} (g : m (α → β)) (x : α), g <*> pure x = (fun h => h x) <$> g
```

Inherited from LawfulApplicative m

```
seq_assoc : ∀ {α β γ : Type u} (x : m α) (g : m (α → β)) (h : m (β → γ)), h <*> (g <*> x) = Function.comp <$> h <*> g <*> x
```

Inherited from LawfulApplicative m

```
bind_pure_comp : ∀ {α β : Type u} (f : α → β) (x : m α),
  (do
      let a ← x
      pure (f a)) =
    f <$> x
```

A bind followed by pure composed with a function is equivalent to a functorial map.

This means that pure really is pure after a bind and has no effects.

```
bind_map : ∀ {α β : Type u} (f : m (α → β)) (x : m α),
  (do
      let x_1 ← f
      x_1 <$> x) =
    f <*> x
```

A bind followed by a functorial map is equivalent to Applicative sequencing.

This means that the effect sequencing from Monad and Applicative are the same.

```
pure_bind : ∀ {α β : Type u} (x : α) (f : α → m β), pure x >>= f = f x
```

pure followed by bind is equivalent to function application.

This means that pure really is pure before a bind and has no effects.

```
bind_assoc : ∀ {α β γ : Type u} (x : m α) (f : α → m β) (g : β → m γ), x >>= f >>= g = x >>= fun x => f x >>= g
```

bind is associative.

Changing the nesting of bind calls while maintaining the order of computations results in an equivalent computation. This means that bind is not doing more than data-dependent sequencing.

theorem

```
LawfulMonad.mk'.{u, v} (m : Type u → Type v) [Monad m]
  (id_map : ∀ {α : Type u} (x : m α), id <$> x = x)
  (pure_bind :
```

```
  ∀ {α β : Type u} (x : α) (f : α → m β), pure x >>= f = f x)
(bind_assoc :
  ∀ {α β γ : Type u} (x : m α) (f : α → m β) (g : β → m γ),
    x >>= f >>= g = x >>= fun x => f x >>= g)
(map_const :
  ∀ {α β : Type u} (x : α) (y : m β),
    Functor.mapConst x y = Function.const β x <$> y := by
  intros; rfl)
(seqLeft_eq :
  ∀ {α β : Type u} (x : m α) (y : m β),
    x <* y = do
      let a ← x
      let _ ← y
      pure a := by
  intros; rfl)
(seqRight_eq :
  ∀ {α β : Type u} (x : m α) (y : m β),
    x *> y = do
      let _ ← x
      y := by
  intros; rfl)
(bind_pure_comp :
  ∀ {α β : Type u} (f : α → β) (x : m α),
    (do
        let y ← x
        pure (f y)) =
      f <$> x := by
  intros; rfl)
(bind_map :
  ∀ {α β : Type u} (f : m (α → β)) (x : m α),
    (do
        let x_1 ← f
        x_1 <$> x) =
      f <*> x := by
  intros; rfl) :
LawfulMonad m
```

An alternative constructor for LawfulMonad which has more defaultable fields in the common case.

14.2. Lifting Monads

When one monad is at least as capable as another, then actions from the latter monad can be used in a context that expects actions from the former. This is called *lifting* the action from one monad to another. Lean automatically inserts lifts when they are available; lifts are defined in the MonadLift type class. Automatic monad lifting is attempted before the general coercion mechanism.

type class

```
MonadLift.{u, v, w} (m : semiOutParam (Type u → Type v))
  (n : Type u → Type w) : Type (max (max (u + 1) v) w)
```

Computations in the monad *m* can be run in the monad *n*. These translations are inserted automatically by the compiler.

Usually, *n* consists of some number of monad transformers applied to *m*, but this is not mandatory.

New instances should use this class, MonadLift. Clients that require one monad to be liftable into another should instead request MonadLiftT, which is the reflexive, transitive closure of MonadLift.

Instance Constructor

MonadLift.mk.{u, v, w}

Methods

monadLift : {α : Type u} → m α → n α

Translates an action from monad *m* into monad *n*.

Lifting between monads is reflexive and transitive:

- Any monad can run its own actions.
- Lifts from *m* to *m'* and from *m'* to *n* can be composed to yield a lift from *m* to *n*. The utility type class MonadLiftT constructs lifts via the reflexive and transitive closure of MonadLift instances. Users should not define new instances of MonadLiftT, but it is useful as an instance implicit parameter to a polymorphic function that needs to run actions from multiple monads in some user-provided monad.

type class

```
MonadLiftT.{u, v, w} (m : Type u → Type v) (n : Type u → Type w) :
  Type (max (max (u + 1) v) w)
```

Computations in the monad *m* can be run in the monad *n*. These translations are inserted automatically by the compiler.

Usually, *n* consists of some number of monad transformers applied to *m*, but this is not mandatory.

This is the reflexive, transitive closure of MonadLift. Clients that require one monad to be liftable into another should request an instance of MonadLiftT. New instances should instead be defined for MonadLift itself.

Instance Constructor

`MonadLiftT.mk.{u, v, w}`

Methods

```
monadLift : {α : Type u} → m α → n α
```

Translates an action from monad *m* into monad *n*.

▼ *Example: Monad Lifts in Function Signatures*

The function `IO.withStdin` has the following signature:

```
IO.withStdin.{u} {m : Type → Type u} {α : Type}
  [Monad m] [MonadFinally m] [MonadLiftT BaseIO m]
  (h : IO.FS.Stream) (x : m α) :
  m α
```

Because it doesn't require its parameter to precisely be in `IO`, it can be used in many monads, and the body does not need to restrict itself to `IO`. The instance implicit parameter `MonadLiftT BaseIO m` allows the reflexive transitive closure of `MonadLift` to be used to assemble the lift.

When a term of type $n\ \beta$ is expected, but the provided term has type $m\ \alpha$, and the two types are not definitionally equal, Lean attempts to insert lifts and coercions before reporting an error. There are the following possibilities:

1. If *m* and *n* can be unified to the same monad, then α and β are not the same. In this case, no monad lifts are necessary, but the value in the monad must be coerced. If the appropriate coercion is found, then a call to `Lean.Internal.coeM` is inserted, which has the following signature:

   ```
   Lean.Internal.coeM.{u, v} {m : Type u → Type v} {α β : Type u}
     [(a : α) → CoeT α a β] [Monad m]
     (x : m α) :
     m β
   ```

2. If α and β can be unified, then the monads differ. In this case, a monad lift is necessary to transform an expression with type $m\ \alpha$ to $n\ \alpha$. If *m* can be lifted to *n* (that is, there is an instance of `MonadLiftT m n`) then a call to `liftM`, which is an alias for `MonadLiftT.monadLift`, is inserted.

   ```
   liftM.{u, v, w}
     {m : Type u → Type v} {n : Type u → Type w}
     [self : MonadLiftT m n] {α : Type u} :
     m α → n α
   ```

3. If neither m and n nor α and β can be unified, but m can be lifted into n and α can be coerced to β, then a lift and a coercion can be combined. This is done by inserting a call to `Lean.Internal.liftCoeM`:

```
Lean.Internal.liftCoeM.{u, v, w}
  {m : Type u → Type v} {n : Type u → Type w}
  {α β : Type u}
  [MonadLiftT m n] [(a : α) → CoeT α a β] [Monad n]
  (x : m α) :
  n β
```

As their names suggest, `Lean.Internal.coeM` and `Lean.Internal.liftCoeM` are implementation details, not part of the public API. In the resulting terms, occurrences of `Lean.Internal.coeM`, `Lean.Internal.liftCoeM`, and coercions are unfolded.

▼ *Example: Lifting `IO` Monads*

There is an instance of `MonadLift BaseIO IO`, so any `BaseIO` action can be run in `IO` as well:

```
def fromBaseIO (act : BaseIO α) : IO α := act
```

Behind the scenes, `liftM` is inserted:

```
#check fun {α} (act : BaseIO α) => (act : IO α)
```

```
fun {α} act => liftM act : {α : Type} → BaseIO α → EIO
IO.Error α
```

▼ *Example: Lifting Transformed Monads*

There are also instances of `MonadLift` for most of the standard library's monad transformers, so base monad actions can be used in transformed monads without additional work. For example, state monad actions can be lifted across reader and exception transformers, allowing compatible monads to be intermixed freely:

```
def incrBy (n : Nat) : StateM Nat Unit := modify (· + n)

def incrOrFail : ReaderT Nat (ExceptT String (StateM Nat)) Unit
:= do
  if (← read) > 5 then throw "Too much!"
  incrBy (← read)
```

Disabling lifting causes an error:

```
set_option autoLift false

def incrBy (n : Nat) : StateM Nat Unit := modify (. + n)
```

```
def incrOrFail : ReaderT Nat (ExceptT String (StateM Nat)) Unit
:= do
  if (← read) > 5 then throw "Too much!"
  incrBy (← read)
```

```
type mismatch
  incrBy __do_lift✝
has type
  StateM Nat Unit : Type
but is expected to have type
  ReaderT Nat (ExceptT String (StateM Nat)) Unit : Type
```

Automatic lifting can be disabled by setting `autoLift` to `false`.

option

`autoLift`

Default value: `true`

Insert monadic lifts (i.e., `liftM` and coercions) when needed.

14.2.1. Reversing Lifts

Monad lifting is not always sufficient to combine monads. Many operations provided by monads are higher order, taking an action *in the same monad* as a parameter. Even if these operations are lifted to some more powerful monad, their arguments are still restricted to the original monad.

There are two type classes that support this kind of "reverse lifting": `MonadFunctor` and `MonadControl`. An instance of `MonadFunctor` *m n* explains how to interpret a fully-polymorphic function in *m* into *n*. This polymorphic function must work for *all* types *α*: it has type `{α : Type u} → m α → m α`. Such a function can be thought of as one that may have effects, but can't do so based on specific values that are provided. An instance of `MonadControl` *m n* explains how to interpret an arbitrary action from *m* into *n*, while at the same time providing a "reverse interpreter" that allows the *m* action to run *n* actions.

14.2.1.1. Monad Functors

type class

```
MonadFunctor.{u, v, w} (m : semiOutParam (Type u → Type v))
  (n : Type u → Type w) : Type (max (max (u + 1) v) w)
```

A way to interpret a fully-polymorphic function in *m* into *n*. Such a function can be thought of as one that may change the effects in *m*, but can't do so based on specific values that are provided.

Clients of MonadFunctor should typically use MonadFunctorT, which is the reflexive, transitive closure of MonadFunctor. New instances should be defined for MonadFunctor.

Instance Constructor

MonadFunctor.mk.{u, v, w}

Methods

monadMap : {α : Type u} → ({β : Type u} → m β → m β) → n α → n α

Lifts a fully-polymorphic transformation of *m* into *n*.

type class

MonadFunctorT.{u, v, w} (m : Type u → Type v) (n : Type u → Type w)
:
Type (max (max (u + 1) v) w)

A way to interpret a fully-polymorphic function in *m* into *n*. Such a function can be thought of as one that may change the effects in *m*, but can't do so based on specific values that are provided.

This is the reflexive, transitive closure of MonadFunctor. It automatically chains together MonadFunctor instances as needed. Clients of MonadFunctor should typically use MonadFunctorT, but new instances should be defined for MonadFunctor.

Instance Constructor

MonadFunctorT.mk.{u, v, w}

Methods

monadMap : {α : Type u} → ({β : Type u} → m β → m β) → n α → n α

Lifts a fully-polymorphic transformation of *m* into *n*.

14.2.1.2. Reversible Lifting with `MonadControl`

type class

MonadControl.{u, v, w} (m : semiOutParam (Type u → Type v))
(n : Type u → Type w) : Type (max (max (u + 1) v) w)

A way to lift a computation from one monad to another while providing the lifted computation with a means of interpreting computations from the outer monad. This provides a means of lifting higher-order operations automatically.

Clients should typically use `control` or `controlAt`, which request an instance of `MonadControlT`: the reflexive, transitive closure of `MonadControl`. New instances should be defined for `MonadControl` itself.

Instance Constructor

`MonadControl.mk.{u, v, w}`

Methods

```
stM : Type u → Type u
```

A type that can be used to reconstruct both a returned value and any state used by the outer monad.

```
liftWith : {α : Type u} → (({β : Type u} → n β → m
(MonadControl.stM m n β)) → m α) → n α
```

Lifts an action from the inner monad *m* to the outer monad *n*. The inner monad has access to a reverse lifting operator that can run an *n* action, returning a value and state together.

```
restoreM : {α : Type u} → m (MonadControl.stM m n α) → n α
```

Lifts a monadic action that returns a state and a value in the inner monad to an action in the outer monad. The extra state information is used to restore the results of effects from the reverse lift passed to `liftWith`'s parameter.

type class

```
MonadControlT.{u, v, w} (m : Type u → Type v) (n : Type u → Type w)
:
  Type (max (max (u + 1) v) w)
```

A way to lift a computation from one monad to another while providing the lifted computation with a means of interpreting computations from the outer monad. This provides a means of lifting higher-order operations automatically.

Clients should typically use `control` or `controlAt`, which request an instance of `MonadControlT`: the reflexive, transitive closure of `MonadControl`. New instances should be defined for `MonadControl` itself.

Instance Constructor

`MonadControlT.mk.{u, v, w}`

Methods

```
stM : Type u → Type u
```

A type that can be used to reconstruct both a returned value and any state used by the outer monad.

```
liftWith : {α : Type u} → (({β : Type u} → n β → m (stM m n β)) → m α) → n α
```

Lifts an action from the inner monad *m* to the outer monad *n*. The inner monad has access to a reverse lifting operator that can run an *n* action, returning a value and state together.

```
restoreM : {α : Type u} → stM m n α → n α
```

Lifts a monadic action that returns a state and a value in the inner monad to an action in the outer monad. The extra state information is used to restore the results of effects from the reverse lift passed to `liftWith`'s parameter.

def

```
control.{u, v, w} {m : Type u → Type v} {n : Type u → Type w}
  [MonadControlT m n] [Bind n] {α : Type u}
  (f : ({β : Type u} → n β → m (stM m n β)) → m (stM m n α)) : n α
```

Lifts an operation from an inner monad to an outer monad, providing it with a reverse lifting operator that allows outer monad computations to be run in the inner monad. The lifted operation is required to return extra information that is required in order to reconstruct the reverse lift's effects in the outer monad; this extra information is determined by `stM`.

This function takes the inner monad as an implicit parameter. Use `controlAt` to specify it explicitly.

def

```
controlAt.{u, v, w} (m : Type u → Type v) {n : Type u → Type w}
  [MonadControlT m n] [Bind n] {α : Type u}
  (f : ({β : Type u} → n β → m (stM m n β)) → m (stM m n α)) : n α
```

Lifts an operation from an inner monad to an outer monad, providing it with a reverse lifting operator that allows outer monad computations to be run in the inner monad. The lifted operation is required to return extra information that is required in order to reconstruct the reverse lift's effects in the outer monad; this extra information is determined by `stM`.

This function takes the inner monad as an explicit parameter. Use `control` to infer the monad.

▼ *Example: Exceptions and Lifting*

One example is `Except.tryCatch`:

```
Except.tryCatch.{u, v} {ε : Type u} {α : Type v}
  (ma : Except ε α) (handle : ε → Except ε α) :
```

```
Except ε α
```

Both of its parameters are in `Except ε`. `MonadLift` can lift the entire application of the handler. The function `getBytes`, which extracts the single bytes from an array of `Nat`s using state and exceptions, is written without `do`-notation or automatic lifting in order to make its structure explicit.

```
set_option autoLift false

def getByte (n : Nat) : Except String UInt8 :=
  if n < 256 then
    pure n.toUInt8
  else throw s!"Out of range: {n}"

def getBytes (input : Array Nat) :
    StateT (Array UInt8) (Except String) Unit := do
  input.forM fun i =>
    liftM (Except.tryCatch (some <$> getByte i) fun _ => pure
none) >>=
      fun
        | some b => modify (·.push b)
        | none => pure ()

#eval getBytes #[1, 58, 255, 300, 2, 1000000] |>.run #[] |>.map
(·.2)
```

```
Except.ok #[1, 58, 255, 2]
```

`getBytes` uses an `Option` returned from the lifted action to signal the desired state updates. This quickly becomes unwieldy if there is more than one way to react to the inner action, such as saving handled exceptions. Ideally, state updates would be performed within the `tryCatch` call directly.

Attempting to save bytes and handled exceptions does not work, however, because the arguments to `Except.tryCatch` have type `Except String Unit`:

```
def getBytes' (input : Array Nat) :
    StateT (Array String)
      (StateT (Array UInt8)
        (Except String)) Unit := do
  input.forM fun i =>
    liftM
      (Except.tryCatch
        (getByte i >>= fun b =>
         modifyThe (Array UInt8) (·.push b))
        fun e =>
          modifyThe (Array String) (·.push e))
```

```
failed to synthesize
  MonadStateOf (Array String) (Except String)
```

```
Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Because StateT has a MonadControl instance, control can be used instead of liftM. It provides the inner action with an interpreter for the outer monad. In the case of StateT, this interpreter expects that the inner monad returns a tuple that includes the updated state, and takes care of providing the initial state and extracting the updated state from the tuple.

```
def getBytes' (input : Array Nat) :
    StateT (Array String)
      (StateT (Array UInt8)
        (Except String)) Unit := do
  input.forM fun i =>
    control fun run =>
      (Except.tryCatch
        (getByte i >>= fun b =>
         run (modifyThe (Array UInt8) (·.push b))))
        fun e =>
          run (modifyThe (Array String) (·.push e))

#eval
  getBytes' #[1, 58, 255, 300, 2, 1000000]
  |>.run #[] |>.run #[]
  |>.map (fun (((), bytes), errs) => (bytes, errs))
```

```
Except.ok (#["Out of range: 300", "Out of range: 1000000"], #
[1, 58, 255, 2])
```

14.3. Syntax

Lean supports programming with functors, applicative functors, and monads via special syntax:

- Infix operators are provided for the most common operations.
- An embedded language called do-notation allows the use of imperative syntax when writing programs in a monad.

14.3.1. Infix Operators

Infix operators are primarily useful in smaller expressions, or when there is no Monad instance.

14.3.1.1. Functors

There are two infix operators for Functor.map.

syntax

Functor Operators

g `<$>` *x* is short for Functor.map *g* *x*.

```
term ::= ...
    | term <$> term
```

`x <&> g` is short for `Functor.map g x`.

```
term ::= ...
    | term <&> term
```

14.3.1.2. Applicative Functors

syntax

Applicative Operators

`g <*> x` is short for `Seq.seq g (fun () => x)`. The function is inserted to delay evaluation because control might not reach the argument.

```
term ::= ...
    | term <*> term
```

`e1 *> e2` is short for `SeqRight.seqRight e1 (fun () => e2)`.

```
term ::= ...
    | term *> term
```

`e1 <* e2` is short for `SeqLeft.seqLeft e1 (fun () => e2)`.

```
term ::= ...
    | term <* term
```

Many applicative functors also support failure and recovery via the `Alternative` type class. This class also has an infix operator.

syntax

Alternative Operators

`e <|> e'` is short for `OrElse.orElse e (fun () => e')`. The function is inserted to delay evaluation because control might not reach the argument.

```
term ::= ...
    | term <|> term
```

```
structure User where
  name : String
  favoriteNat : Nat
def main : IO Unit := pure ()
```

▼ *Example: Infix `Functor` and `Applicative` Operators*

A common functional programming idiom is to use a pure function in some context with effects by applying it via `Functor.map` and `Seq.seq`. The function is applied to its sequence of arguments using `<$>`, and the arguments are separated by `<*>`.

In this example, the constructor `User.mk` is applied via this idiom in the body of `main`.

```
def getName : IO String := do
  IO.println "What is your name?"
  return (← (← IO.getStdin).getLine).trimRight

partial def getFavoriteNat : IO Nat := do
  IO.println "What is your favorite natural number?"
  let line ← (← IO.getStdin).getLine
  if let some n := line.trim.toNat? then
    return n
  else
    IO.println "Let's try again."
    getFavoriteNat

structure User where
  name : String
  favoriteNat : Nat
deriving Repr

def main : IO Unit := do
  let user ← User.mk <$> getName <*> getFavoriteNat
  IO.println (repr user)
```

When run with this input:

```
stdin
  1 A. Lean User
  2 None
  3 42
```

it produces this output:

```
stdout
  1 What is your name?
  2 What is your favorite natural number?
  3 Let's try again.
  4 What is your favorite natural number?
  5 { name := "A. Lean User", favoriteNat := 42 }
```

14.3.1.3. Monads

Monads are primarily used via do-notation. However, it can sometimes be convenient to describe monadic computations via operators.

syntax

Monad Operators

act `>>=` *f* is syntax for `Bind.bind` *act f*.

```
term ::= ...
    | term >>= term
```

Similarly, the reversed operator *f* `=<<` *act* is syntax for `Bind.bind` *act f*.

```
term ::= ...
    | term =<< term
```

The Kleisli composition operators `Bind.kleisliRight` and `Bind.kleisliLeft` also have infix operators.

```
term ::= ...
    | term >=> term
```

```
term ::= ...
    | term <=< term
```

14.3.2. do-Notation

Monads are primarily used via `do`-notation, which is an embedded language for programming in an imperative style. It provides familiar syntax for sequencing effectful operations, early return, local mutable variables, loops, and exception handling. All of these features are translated to the operations of the `Monad` type class, with a few of them requiring addition instances of classes such as `ForIn` that specify iteration over containers. For more details about the design of `do`-notation, please consult Ullrich and de Moura (2022) (i) Sebastian Ullrich and Leonardo de Moura, 2022. "do Unchained: Embracing Local Imperativity in a Purely Functional Language". In *Proceedings of the ACM on Programming Languages: ICFP 2022.* . A `do` term consists of the keyword `do` followed by a sequence of `do` *items*.

syntax

do-Notation

```
term ::= ...
    | do doSeqItem*
```

The items in a `do` may be separated by semicolons; otherwise, each should be on its own line and they should have equal indentation.

14.3.2.1. Sequential Computations

One form of do item is a term.

syntax

Terms in do-Notation

```
doSeqItem ::= ...
    | term
```

A term followed by a sequence of items is translated to a use of `bind`; in particular, `do e1; es` is translated to `e1 >>= fun () => do es`.

`do` Item	Desugaring

`do`
`e1`
`es`
	`e1 >>= fun () => do es`

The result of the term's computation may also be named, allowing it to be used in subsequent steps. This is done using `let`.

syntax

Data Dependence in do-Notation

There are two forms of monadic `let`-binding in a `do` block. The first binds an identifier to the result, with an optional type annotation:

```
doSeqItem ::= ...
    | let ident(:term)? ← term
```

The second binds a pattern to the result. The fallback clause, beginning with `|`, specifies the behavior when the pattern does not match the result.

```
doSeqItem ::= ...
    | let term ← term
        (| doSeq)?
```

This syntax is also translated to a use of bind. `do let x ← e1; es` is translated to *e1* `>>= fun` *x* `=> do` *es*, and fallback clauses are translated to default pattern matches. `let` may also be used with the standard definition syntax `:=` instead of ←. This indicates a pure, rather than monadic, definition:

syntax

Local Definitions in do-Notation

```
doSeqItem ::= ...
    | let (ident | hole) := term
```

`do let x := e; es` is translated to `let` *x* `:=` *e*`; do es`.

do Item	Desugaring

`do`
`let x ← e1`
`es`
	e1 `>>= fun` *x* `=>`
	`  do` *es*

`do`
`let some x ← e1?`
`  | fallback`
`es`
	e1? `>>= fun`
	`  | some` *x* `=> do`
	`  es`
	`  | _ =>` *fallback*

`do`
`let x := e`
`es`
	`let` *x* `:=` *e*
	`do es`

Within a `do` block, ← may be used as a prefix operator. The expression to which it is applied is replaced with a fresh variable, which is bound using `bind` just before the current step. This allows monadic effects to be used in positions that otherwise might expect a pure value, while still maintaining the distinction between *describing* an effectful computation and actually *executing* its effects. Multiple occurrences of ← are processed from left to right, inside to outside.

Example do Item	Desugaring

`do`
`f (← e1) (← e2)`
`es`
	`do`
	`let x ← e1`
	`let y ← e2`
	`f x y`
	`es`

`do`
`let x := g (← h (← e1))`
`es`
	`do`
	`let y ← e1`
	`let z ← h y`
	`let x := g z`
	`es`

Example Nested Action Desugarings

In addition to convenient support for sequential computations with data dependencies, do-notation also supports the local addition of a variety of effects, including early return, local mutable state, and loops with early termination. These effects are implemented via transformations of the entire do block in a manner akin to monad transformers, rather than via a local desugaring.

14.3.2.2. Early Return

Early return terminates a computation immediately with a given value. The value is returned from the closest containing do block; however, this may not be the closest do keyword. The rules for determining the extent of a do block are described in their own section.

syntax

Early Return

```
doSeqItem ::= ...
    | return term

doSeqItem ::= ...
    | return
```

Not all monads include early return. Thus, when a do block contains `return`, the code needs to be rewritten to simulate the effect. A program that uses early return to compute a value of type α in a monad m can be thought of as a program in the monad `ExceptT α m α`: early-returned values take the exception pathway, while ordinary returns do not. Then, an outer handler can return the value from either code paths. Internally, the do elaborator performs a translation very much like this one.

On its own, `return` is short for `return ()`.

14.3.2.3. Local Mutable State

Local mutable state is mutable state that cannot escape the do block in which it is defined. The `let mut` binder introduces a locally-mutable binding.

syntax

Local Mutability

Mutable bindings may be initialized either with pure computations or with monadic computations:

```
doSeqItem ::= ...
    | let mut (ident | hole) := term

doSeqItem ::= ...
    | let mut ident ← doElem
```

Similarly, they can be mutated either with pure values or the results of monad computations:

```
doElem ::= ...
    | ident(: term)?  := term

doElem ::= ...
    | term(: term)? := term

doElem ::= ...
    | ident(: term)? ← term

doElem ::= ...
    | term ← term
        (| doSeq)?
```

These locally-mutable bindings are less powerful than a state monad because they are not mutable outside their lexical scope; this also makes them easier to reason about. When do blocks contain mutable bindings, the do elaborator transforms the expression similarly to the way that `StateT` would, constructing a new monad and initializing it with the correct values.

14.3.2.4. Control Structures

There are do items that correspond to most of Lean's term-level control structures. When they occur as a step in a do block, they are interpreted as do items rather than terms. Each branch of the control structures is a sequence of do items, rather than a term, and some of them are more syntactically flexible than their corresponding terms.

syntax

Conditionals

In a do block, if statements may omit their else branch. Omitting an else branch is equivalent to using pure () as the contents of the branch.

```
doSeqItem ::= ...
    | if (ident :)? term then
        doSeqItem*
      (else
        doSeqItem*)?
```

Syntactically, the then branch cannot be omitted. For these cases, unless only executes its body when the condition is false. The do in unless is part of its syntax and does not induce a nested do block.

syntax

Reverse Conditionals

```
doSeqItem ::= ...
    | unless term do
        doSeqItem*
```

When match is used in a do block, each branch is considered to be part of the same block. Otherwise, it is equivalent to the match term.

syntax

Pattern Matching

```
doSeqItem ::= ...
    | match (((ident | hole) :)? term),* with
        (| term,* => doSeqItem*)*
```

14.3.2.5. Iteration

Within a do block, for…in loops allow iteration over a data structure. The body of the loop is part of the containing do block, so local effects such as early return and mutable variables may be used.

syntax

Iteration over Collections

```
doSeqItem ::= ...
    | for ((ident :)? term in term),* do
        doSeqItem*
```

A `for…in` loop requires at least one clause that specifies the iteration to be performed, which consists of an optional membership proof name followed by a colon (`:`), a pattern to bind, the keyword `in`, and a collection term. The pattern, which may just be an identifier, must match any element of the collection; patterns in this position cannot be used as implicit filters. Further clauses may be provided by separating them with commas. Each collection is iterated over at the same time, and iteration stops when any of the collections runs out of elements.

▼ *Example: Iteration Over Multiple Collections*

When iterating over multiple collections, iteration stops when any of the collections runs out of elements.

```
#eval Id.run do
  let mut v := #[]
  for x in [0:43], y in ['a', 'b'] do
    v := v.push (x, y)
  return v
```

```
#[(0, 'a'), (1, 'b')]
```

▼ *Example: Iteration over Array Indices with* `for`

When iterating over the valid indices for an array with `for`, naming the membership proof allows the tactic that searches for proofs that array indices are in bounds to succeed.

```
def satisfyingIndices
    (p : α → Prop) [DecidablePred p]
    (xs : Array α) : Array Nat := Id.run do
  let mut out := #[]
  for h : i in [0:xs.size] do
    if p xs[i] then out := out.push i
  return out
```

Omitting the hypothesis name causes the array lookup to fail, because no proof is available in the context that the iteration variable is within the specified range.

Iteration with `for`-loops is translated into uses of `ForIn.forIn`, which is an analogue of `ForM.forM` with added support for local mutations and early termination. `ForIn.forIn` receives an initial value for the local mutable state and a monadic action as parameters, along with the collection being iterated over. The monadic action passed to `ForIn.forIn` takes a current state as a parameter and, after carrying out actions in the monad *m*, returns either `ForInStep.yield` to indicate that iteration should continue with an updated set of local mutable values, or `ForInStep.done` to indicate that `break` or `return` was executed. When iteration is complete, `ForIn.forIn` returns the final values of the local mutable values.

The specific desugaring of a loop depends on how state and early termination are used in its body. Here are some examples:

`do` Item	Desugaring

`do`
`let mut b := …`
`for x in xs do`
`  b ← f x b`
`es`
	`do`
	`let b := …`
	`let b ← ForIn.forIn xs b fun x b => do`
	`  let b ← f x b`
	`  return ForInStep.yield b`
	`es`

`do`
`let mut b := …`
`for x in xs do`
`  b ← f x b`
`  break`
`es`
	`do`
	`let b := …`
	`let b ← ForIn.forIn xs b fun x b => do`
	`  let b ← f x b`
	`  return ForInStep.done b`
	`es`

`do`
`let mut b := …`
`for h : x in xs do`
`  b ← f' x h b`
`es`
	`do`
	`let b := …`
	`let b ← ForIn'.forIn' xs b fun x h b => do`
	`  let b ← f' x h b`
	`  return ForInStep.yield b`
	`es`

`do`
`let mut b := …`
`for h : x in xs do`
`  b ← f' x h b`
`  break`
`es`
	`do`
	`let b := …`
	`let b ← ForIn'.forIn' xs b fun x h b => do`
	`  let b ← f' x h b`
	`  return ForInStep.done b`
	`es`

The body of a `while` loop is repeated while the condition remains true. It is possible to write infinite loops using them in functions that are not marked `partial`. This is because the `partial` modifier only applies to non-termination or infinite regress induced by the function being defined, and not by those that it calls. The translation of `while` loops relies on a separate helper.

syntax

Conditional Loops

```
doSeqItem ::= ...
    | while term do
```

```
        doSeqItem*

doSeqItem ::= ...
    | while ident : term do
        doSeqItem*
```

The body of a `repeat`-`until` loop is always executed at least once. After each iteration, the condition is checked, and the loop is repeated when the condition is **false**. When the condition becomes true, iteration stops.

syntax

Post-Tested Loops

```
doSeqItem ::= ...
    | repeat
        doSeqItem*
      until term
```

The body of a `repeat` loop is repeated until a `break` statement is executed. Just like `while` loops, these loops can be used in functions that are not marked `partial`.

syntax

Unconditional Loops

```
doSeqItem ::= ...
    | repeat
        doSeqItem*
```

The `continue` statement skips the rest of the body of the closest enclosing `repeat`, `while`, or `for` loop, moving on to the next iteration. The `break` statement terminates the closest enclosing `repeat`, `while`, or `for` loop, stopping iteration.

syntax

Loop Control Statements

```
doSeqItem ::= ...
    | continue

doSeqItem ::= ...
    | break
```

In addition to `break`, loops can always be terminated by effects in the current monad. Throwing an exception from a loop terminates the loop.

▼ *Example: Terminating Loops in the `Option` Monad*

The `failure` method from the `Alternative` class can be used to terminate an otherwise-infinite loop in the `Option` monad.

```
#eval show Option Nat from do
  let mut i := 0
  repeat
    if i > 1000 then failure
    else i := 2 * (i + 1)
  return i
```

```
none
```

14.3.2.6. Identifying do Blocks

Many features of `do`-notation have an effect on the current `do` block. In particular, early return aborts the current block, causing it to evaluate to the returned value, and mutable bindings can only be mutated in the block in which they are defined. Understanding these features requires a precise definition of what it means to be in the “same” block.

Empirically, this can be checked using the Lean language server. When the cursor is on a `return` statement, the corresponding `do` keyword is highlighted. Attempting to mutate a mutable binding outside of the same `do` block results in an error message.

```
def find? (p : α → Option β) (xs : List α) : Option β := do
  for x in xs do
    if let some y := p x then
      return y
  failure

def find? (p : α → Option β) (xs : List α) : Option β := do
  xs.forM fun x => do
    if let some y := p x then
      return y
```

Highlighting `do`

The rules are as follows:

- Each item immediately nested under the `do` keyword that begins a block belongs to that block.
- Each item immediately nested under the `do` keyword that is an item in a containing `do` block belongs to the outer block.
- Items in the branches of an `if`, `match`, or `unless` item belong to the same `do` block as the control structure that contains them. The `do` keyword that is part of the syntax of `unless` does not introduce a new `do` block.
- Items in the body of `repeat`, `while`, and `for` belong to the same `do` block as the loop that contains them. The `do` keyword that is part of the syntax of `while` and `for` does not introduce a new `do` block.

▼ *Example: Nested do and Branches*

The following example outputs 6 rather than 7:

```
def test : StateM Nat Unit := do
  set 5
  if true then
    set 6
    do return
  set 7
  return

#eval test.run 0
```

```
((), 6)
```

This is because the `return` statement under the `if` belongs to the same `do` as its immediate parent, which itself belongs to the same `do` as the `if`. If `do` blocks that occurred as items in other `do` blocks instead created new blocks, then the example would output 7.

14.3.2.7. Type Classes for Iteration

To be used with `for` loops without membership proofs, collections must implement the `ForIn` type class. Implementing `ForIn'` additionally allows the use of `for` loops with membership proofs.

type class

```
ForIn.{u, v, u₁, u₂} (m : Type u₁ → Type u₂) (ρ : Type u)
  (α : outParam (Type v)) : Type (max (max (max u (u₁ + 1)) u₂) v)
```

Monadic iteration in **do**-blocks, using the `for x in xs` notation.

The parameter m is the monad of the **do**-block in which iteration is performed, ρ is the type of the collection being iterated over, and α is the type of elements.

Instance Constructor

`ForIn.mk.{u, v, u₁, u₂}`

Methods

`forIn : {β : Type u₁} → [Monad m] → ρ → β → (α → β → m (ForInStep β)) → m β`

Monadically iterates over the contents of a collection `xs`, with a local state b and the possibility of early termination.

Because a **do** block supports local mutable bindings along with **`return`**, and **`break`**, the monadic action passed to `ForIn.forIn` takes a starting state in addition to the current element of the collection and returns an updated state together with an indication of whether iteration should continue or terminate. If the action returns `ForInStep.done`, then `ForIn.forIn` should stop iteration and return the updated state. If the action returns `ForInStep.yield`, then `ForIn.forIn` should continue iterating if there are further elements, passing the updated state to the action.

More information about the translation of **`for`** loops into `ForIn.forIn` is available in the Lean reference manual.

type class

```
ForIn'.{u, v, u₁, u₂} (m : Type u₁ → Type u₂) (ρ : Type u)
  (α : outParam (Type v)) (d : outParam (Membership α ρ)) :
  Type (max (max (max u (u₁ + 1)) u₂) v)
```

Monadic iteration in **do**-blocks with a membership proof, using the **`for`** `h : x` **`in`** `xs` notation.

The parameter *m* is the monad of the **do**-block in which iteration is performed, *ρ* is the type of the collection being iterated over, *α* is the type of elements, and *d* is the specific membership predicate to provide.

Instance Constructor

`ForIn'.mk.{u, v, u₁, u₂}`

Methods

`forIn' : {β : Type u₁} → [Monad m] → (x : ρ) → β → ((a : α) → a ∈ x → β → m (ForInStep β)) → m β`

Monadically iterates over the contents of a collection `xs`, with a local state b and the possibility of early termination. At each iteration, the body of the loop is provided with a proof that the current element is in the collection.

Because a **do** block supports local mutable bindings along with **return**, and **break**, the monadic action passed to `ForIn'.forIn'` takes a starting state in addition to the current element of the collection with its membership proof. The action returns an updated state together with an indication of whether iteration should continue or terminate. If the action returns `ForInStep.done`, then `ForIn'.forIn'` should stop iteration and return the updated state. If the action returns `ForInStep.yield`, then `ForIn'.forIn'` should continue iterating if there are further elements, passing the updated state to the action.

More information about the translation of **for** loops into `ForIn'.forIn'` is available in the Lean reference manual.

inductive type

```
ForInStep.{u} (α : Type u) : Type u
```

An indication of whether a loop's body terminated early that's used to compile the **for** `x` **in** `xs` notation.

A collection's `ForIn` or `ForIn'` instance describe's how to iterate over its elements. The monadic action that represents the body of the loop returns a `ForInStep α`, where α is the local state used to implement features such as **let mut**.

Constructors

| `done.{u} {α : Type u} : α → ForInStep α`

The loop should terminate early.

`ForInStep.done` is produced by uses of **break** or **return** in the loop body.

| `yield.{u} {α : Type u} : α → ForInStep α`

The loop should continue with the next iteration, using the returned state.

`ForInStep.yield` is produced by **continue** and by reaching the bottom of the loop body.

def

```
ForInStep.value.{u_1} {α : Type u_1} (x : ForInStep α) : α
```

Extracts the value from a `ForInStep`, ignoring whether it is `ForInStep.done` or `ForInStep.yield`.

type class

```
ForM.{u, v, w₁, w₂} (m : Type u → Type v) (γ : Type w₁)
  (α : outParam (Type w₂)) : Type (max (max (max (u + 1) v) w₁) w₂)
```

Overloaded monadic iteration over some container type.

An instance of `ForM m γ α` describes how to iterate a monadic operator over a container of type `γ` with elements of type `α` in the monad `m`. The element type should be uniquely determined by the monad and the container.

Use `ForM.forIn` to construct a `ForIn` instance from a `ForM` instance, thus enabling the use of the **`for`** operator in **`do`**-notation.

Instance Constructor

`ForM.mk.{u, v, w₁, w₂}`

Methods

`forM : [Monad m] → γ → (α → m PUnit) → m PUnit`

Runs the monadic action `f` on each element of the collection `coll`.

def

```
ForM.forIn.{u_1, u_2, u_3, u_4} {m : Type u_1 → Type u_2} {β : Type
u_1}
  {ρ : Type u_3} {α : Type u_4} [Monad m]
  [ForM (StateT β (ExceptT β m)) ρ α] (x : ρ) (b : β)
  (f : α → β → m (ForInStep β)) : m β
```

Creates a suitable implementation of `ForIn.forIn` from a `ForM` instance.

14.4. API Reference

In addition to the general functions described here, there are some functions that are conventionally defined as part of the API of in the namespace of each collection type:

- `mapM` maps a monadic function.
- `forM` maps a monadic function, throwing away the result.
- `filterM` filters using a monadic predicate, returning the values that satisfy it.

▼ *Example: Monadic Collection Operations*

`Array.filterM` can be used to write a filter that depends on a side effect.

```
def values := #[1, 2, 3, 5, 8]
def main : IO Unit := do
  let filtered ← values.filterM fun v => do
```

```
    repeat
      IO.println s!"Keep {v}? [y/n]"
      let answer := (← (← IO.getStdin).getLine).trim
      if answer == "y" then return true
      if answer == "n" then return false
    return false
  IO.println "These values were kept:"
  for v in filtered do
    IO.println s!" * {v}"
```

```
stdin
  1 y
  2 n
  3 oops
  4 y
  5 n
  6 y
```

```
stdout
  1 Keep 1? [y/n]
  2 Keep 2? [y/n]
  3 Keep 3? [y/n]
  4 Keep 3? [y/n]
  5 Keep 5? [y/n]
  6 Keep 8? [y/n]
  7 These values were kept:
  8  * 1
  9  * 3
 10  * 8
```

14.4.1. Discarding Results

The `discard` function is especially useful when using an action that returns a value only for its side effects.

def

```
Functor.discard.{u, v} {f : Type u → Type v} {α : Type u} [Functor f]
  (x : f α) : f PUnit
```

Discards the value in a functor, retaining the functor's structure.

Discarding values is especially useful when using `Applicative` functors or `Monad`s to implement effects, and some operation should be carried out only for its effects. In **do**-notation, statements whose values are discarded must return `Unit`, and `discard` can be used to explicitly discard their values.

14.4.2. Control Flow

def

```
guard.{v} {f : Type → Type v} [Alternative f] (p : Prop) [Decidable
p] :
  f Unit
```

If the proposition *p* is true, does nothing, else fails (using `failure`).

def

```
optional.{u, v} {f : Type u → Type v} [Alternative f] {α : Type u}
  (x : f α) : f (Option α)
```

Returns some *x* if *f* succeeds with value *x*, else returns none.

14.4.3. Lifting Boolean Operations

def

```
andM.{u, v} {m : Type u → Type v} {β : Type u} [Monad m] [ToBool β]
  (x y : m β) : m β
```

Converts the result of the monadic action *x* to a Bool. If it is `true`, returns *y*; otherwise, returns the original result of *x*.

This a monadic counterpart to the short-circuiting && operator, usually accessed via the <&&> operator.

Conventions for notations in identifiers:

- The recommended spelling of <&&> in identifiers is andM.

def

```
orM.{u, v} {m : Type u → Type v} {β : Type u} [Monad m] [ToBool β]
  (x y : m β) : m β
```

Converts the result of the monadic action *x* to a Bool. If it is `true`, returns it and ignores *y*; otherwise, runs *y* and returns its result.

This a monadic counterpart to the short-circuiting || operator, usually accessed via the <||> operator.

Conventions for notations in identifiers:

- The recommended spelling of <||> in identifiers is orM.

def

```
notM.{v} {m : Type → Type v} [Functor m] (x : m Bool) : m Bool
```

Runs a monadic action and returns the negation of its result.

14.4.4. Kleisli Composition

Kleisli composition is the composition of monadic functions, analogous to `Function.comp` for ordinary functions.

def

```
Bind.kleisliRight.{u, u_1, u_2} {α : Type u} {m : Type u_1 → Type
u_2}
  {β γ : Type u_1} [Bind m] (f₁ : α → m β) (f₂ : β → m γ) (a : α) :
m γ
```

Left-to-right composition of Kleisli arrows.

Conventions for notations in identifiers:

- The recommended spelling of >=> in identifiers is `kleisliRight`.

def

```
Bind.kleisliLeft.{u, u_1, u_2} {α : Type u} {m : Type u_1 → Type
u_2}
  {β γ : Type u_1} [Bind m] (f₂ : β → m γ) (f₁ : α → m β) (a : α) :
m γ
```

Right-to-left composition of Kleisli arrows.

Conventions for notations in identifiers:

- The recommended spelling of <=< in identifiers is `kleisliLeft`.

14.4.5. Re-Ordered Operations

Sometimes, it can be convenient to partially apply a function to its second argument. These functions reverse the order of arguments, making it this easier.

def

```
Functor.mapRev.{u, v} {f : Type u → Type v} [Functor f] {α β : Type
u} :
  f α → (α → β) → f β
```

Maps a function over a functor, with parameters swapped so that the function comes last.

This function is `Functor.map` with the parameters reversed, typically used via the <&> operator.

Conventions for notations in identifiers:

- The recommended spelling of <&> in identifiers is `mapRev`.

def

```
Bind.bindLeft.{u, u_1} {α : Type u} {m : Type u → Type u_1} {β :
Type u}
  [Bind m] (f : α → m β) (ma : m α) : m β
```

Same as `Bind.bind` but with arguments swapped.

Conventions for notations in identifiers:

- The recommended spelling of =<< in identifiers is `bindLeft`.

14.5. Varieties of Monads

The `IO` monad has many, many effects, and is used for writing programs that need to interact with the world. It is described in its own section. Programs that use `IO` are essentially black boxes: they are typically not particularly amenable to verification.

Many algorithms are easiest to express with a much smaller set of effects. These effects can often be simulated; for example, mutable state can be simulated by passing around a tuple that contains both the program's value and the state. These simulated effects are easier to reason formally about, because they are defined using ordinary code rather than new language primitives.

The standard library provides abstractions for working with commonly-used effects. Many frequently-used effects fall into a small number of categories:

State monads have mutable state

Computations that have access to some data that may be modified by other parts of the computation use *mutable state*. State can be implemented in a variety of ways, described in the section on state monads and captured in the `MonadState` type class.

Reader monads are parameterized computations

Computations that can read the value of some parameter provided by a context exist in most programming languages, but many languages that feature state and exceptions as first-class features do not have built-in facilities for defining new parameterized computations. Typically, these computations are provided with a parameter value when invoked, and sometimes they can locally override it. Parameter values have *dynamic extent*: the value provided most recently in the call stack is the one that is used. They can be simulated by passing a value unchanged through a sequence of function calls; however, this technique can make code harder to read and introduces a risk that the values may be passed incorrectly to further calls by mistake.

They can also be simulated using mutable state with a careful discipline surrounding the modification of the state. Monads that maintain a parameter, potentially allowing it to be overridden in a section of the call stack, are called *reader monads*. Reader monads are captured in the `MonadReader` type class. Additionally, reader monads that allow the parameter value to be locally overridden are captured in the `MonadWithReader` type class.

Exception monads have exceptions

Computations that may terminate early with an exceptional value use *exceptions*. They are typically modeled with a sum type that has a constructor for ordinary termination and a constructor for early termination with errors. Exception monads are described in the section on exception monads, and captured in the `MonadExcept` type class.

14.5.1. Monad Type Classes

Using type classes like `MonadState` and `MonadExcept` allow client code to be polymorphic with respect to monads. Together with automatic lifting, this allows programs to be re-usable in many different monads and makes them more robust to refactoring.

It's important to be aware that effects in a monad may not interact in only one way. For example, a monad with state and exceptions may or may not roll back state changes when an exception is thrown. If this matters for the correctness of a function, then it should use a more specific signature.

▼ *Example: Effect Ordering*

The function `sumNonFives` adds the contents of a list using a state monad, terminating early if it encounters a 5.

```
def sumNonFives {m}
    [Monad m] [MonadState Nat m] [MonadExcept String m]
    (xs : List Nat) :
    m Unit := do
  for x in xs do
    if x == 5 then
      throw "Five was encountered"
    else
      modify (· + x)
```

Running it in one monad returns the state at the time that 5 was encountered:

```
#eval
  sumNonFives (m := ExceptT String (StateM Nat))
    [1, 2, 3, 4, 5, 6] |>.run |>.run 0
```

```
(Except.error "Five was encountered", 10)
```

In another, the state is discarded:

```
#eval
  sumNonFives (m := StateT Nat (Except String))
    [1, 2, 3, 4, 5, 6] |>.run 0
```

```
Except.error "Five was encountered"
```

In the second case, an exception handler would roll back the state to its value at the start of the `try`. The following function is thus incorrect:

```
/-- Computes the sum of the non-5 prefix of a list. -/
def sumUntilFive {m}
    [Monad m] [MonadState Nat m] [MonadExcept String m]
    (xs : List Nat) :
    m Nat := do
  MonadState.set 0
  try
    sumNonFives xs
  catch _ =>
    pure ()
  get
```

In one monad, the answer is correct:

```
#eval
  sumUntilFive (m := ExceptT String (StateM Nat))
    [1, 2, 3, 4, 5, 6] |>.run |>.run' 0
```

```
Except.ok 10
```

In the other, it is not:

```
#eval
  sumUntilFive (m := StateT Nat (Except String))
    [1, 2, 3, 4, 5, 6] |>.run' 0
```

```
Except.ok 0
```

A single monad may support multiple version of the same effect. For example, there might be a mutable `Nat` and a mutable `String` or two separate reader parameters. As long as they have different types, it should be convenient to access both. In typical use, some monadic operations that are overloaded in type classes have type information available for instance synthesis, while others do not. For example, the argument passed to `set` determines the type of the state to be used, while `get` takes no such argument. The type information present in applications of `set` can be used to pick the correct instance when multiple states are available, which suggests that the type of the mutable state should be an input parameter or semi-output parameter so that it can be used to select instances. The lack of type information present in uses of `get`, on the other hand, suggests that the type of the mutable state should be an output parameter in `MonadState`, so type class synthesis determines the state's type from the monad itself.

This dichotomy is solved by having two versions of many of the effect type classes. The version with a semi-output parameter has the suffix `-Of`, and its operations take types explicitly as needed. Examples include `MonadStateOf`, `MonadReaderOf`, and `MonadExceptOf`. The operations with explicit type parameters have names ending in `-The`, such as `getThe`, `readThe`, and

`tryCatchThe`. The name of the version with an output parameter is undecorated. The standard library exports a mix of operations from the `-Of` and undecorated versions of each type class, based on what has good inference behavior in typical use cases.

Operation	From Class	Notes
`get`	`MonadState`	Output parameter improves type inference
`set`	`MonadStateOf`	Semi-output parameter uses type information from `set`'s argument
`modify`	`MonadState`	Output parameter is needed to allow functions without annotations
`modifyGet`	`MonadState`	Output parameter is needed to allow functions without annotations
`read`	`MonadReader`	Output parameter is needed due to lack of type information from arguments
`readThe`	`MonadReaderOf`	Semi-output parameter uses the provided type to guide synthesis
`withReader`	`MonadWithReader`	Output parameter avoids the need for type annotations on the function
`withTheReader`	`MonadWithReaderOf`	Semi-output parameter uses provided type to guide synthesis
`throw`	`MonadExcept`	Output parameter enables the use of constructor dot notation for the exception
`throwThe`	`MonadExceptOf`	Semi-output parameter uses provided type to guide synthesis
`tryCatch`	`MonadExcept`	Output parameter enables the use of constructor dot notation for the exception
`tryCatchThe`	`MonadExceptOf`	Semi-output parameter uses provided type to guide synthesis

▼ *Example: State Types*

The state monad `M` has two separate states: a `Nat` and a `String`.

```
abbrev M := StateT Nat (StateM String)
```

Because `get` is an alias for `MonadState.get`, the state type is an output parameter. This means that Lean selects a state type automatically, in this case the one from the outermost monad transformer:

```
#check (get : M _)
```

```
get : M Nat
```

Only the outermost may be used, because the type of the state is an output parameter.

```
#check (get : M String)
```

```
failed to synthesize
  MonadState String M

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

Providing the state type explicitly using `getThe` from `MonadStateOf` allows both states to be read.

```
#check ((getThe String, getThe Nat) : M String × M Nat)
```

```
(getThe String, getThe Nat) : M String × M Nat
```

Setting a state works for either type, because the state type is a semi-output parameter on `MonadStateOf`.

```
#check (set 4 : M Unit)
```

```
set 4 : M PUnit
```

```
#check (set "Four" : M Unit)
```

```
set "Four" : M PUnit
```

14.5.2. Monad Transformers

A *monad transformer* is a function that, when provided with a monad, gives back a new monad. Typically, this new monad has all the effects of the original monad along with some additional ones.

A monad transformer consists of the following:

- A function T that constructs the new monad's type from an existing monad
- A `run` function that adapts a $T\ m\ \alpha$ into some variant of m, often requiring additional parameters and returning a more specific type under m
- An instance of `[Monad m] → Monad (T m)` that allows the transformed monad to be used as a monad

- An instance of `MonadLift` that allows the original monad's code to be used in the transformed monad
- If possible, an instance of `MonadControl m (T m)` that allows actions from the transformed monad to be used in the original monad

Typically, a monad transformer also provides instances of one or more type classes that describe the effects that it introduces. The transformer's `Monad` and `MonadLift` instances make it practical to write code in the transformed monad, while the type class instances allow the transformed monad to be used with polymorphic functions.

▼ *Example: The Identity Monad Transformer*

The identity monad transformer neither adds nor removes capabilities to the transformed monad. Its definition is the identity function, suitably specialized:

```
def IdT (m : Type u → Type v) : Type u → Type v := m
```

Similarly, the `run` function requires no additional arguments and just returns an `m α`:

```
def IdT.run (act : IdT m α) : m α := act
```

The monad instance relies on the monad instance for the transformed monad, selecting it via type ascriptions:

```
instance [Monad m] : Monad (IdT m) where
  pure x := (pure x : m _)
  bind x f := (x >>= f : m _)
```

Because `IdT m` is definitionally equal to `m`, the `MonadLift m (IdT m)` instance doesn't need to modify the action being lifted:

```
instance : MonadLift m (IdT m) where
  monadLift x := x
```

The `MonadControl` instance is similarly simple.

```
instance [Monad m] : MonadControl m (IdT m) where
  stM α := α
  liftWith f := f (fun x => Id.run <| pure x)
  restoreM v := v
```

The Lean standard library provides transformer versions of many different monads, including `ReaderT`, `ExceptT`, and `StateT`, along with variants using other representations such as `StateCpsT`, `StateRefT`, and `ExceptCpsT`. Additionally, the `EStateM` monad is equivalent to combining `ExceptT` and `StateT`, but it can use a more specialized representation to improve performance.

14.5.3. Identity

The identity monad `Id` has no effects whatsoever. Both `Id` and the corresponding implementation of `pure` are the identity function, and `bind` is reversed function application. The identity monad has two primary use cases:

1. It can be the type of a `do` block that implements a pure function with local effects.
2. It can be placed at the bottom of a stack of monad transformers.

def

```
Id.{u} (type : Type u) : Type u
```

The identity function on types, used primarily for its `Monad` instance.

The identity monad is useful together with monad transformers to construct monads for particular purposes. Additionally, it can be used with **do**-notation in order to use control structures such as local mutability, **for**-loops, and early returns in code that does not otherwise use monads.

Examples:

```
def containsFive (xs : List Nat) : Bool := Id.run do
  for x in xs do
    if x == 5 then return true
  return false

#eval containsFive [1, 3, 5, 7]
```

```
true
```

def

```
Id.run.{u_1} {α : Type u_1} (x : Id α) : α
```

Runs a computation in the identity monad.

This function is the identity function. Because its parameter has type `Id α`, it causes **do**-notation in its arguments to use the `Monad Id` instance.

▼ *Example: Local Effects with the Identity Monad*

This code block implements a countdown procedure by using simulated local mutability in the identity monad.

```
#eval Id.run do
  let mut xs := []
  for x in [0:10] do
    xs := x :: xs
```

```
  pure xs
```

```
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

14.5.4. State

State monads provide access to a mutable value. The underlying implementation may use a tuple to simulate mutability, or it may use something like `ST.Ref` to ensure mutation. Even those implementations that use a tuple may in fact use mutation at run-time due to Lean's use of mutation when there are unique references to values, but this requires a programming style that prefers `modify` and `modifyGet` over `get` and `set`.

14.5.4.1. General State API

type class

```
MonadState.{u, v} (σ : outParam (Type u)) (m : Type u → Type v) :
  Type (max (u + 1) v)
```

State monads provide a value of a given type (the *state*) that can be retrieved or replaced. Instances may implement these operations by passing state values around, by using a mutable reference cell (e.g. `ST.Ref σ`), or in other ways.

In this class, σ is an `outParam`, which means that it is inferred from m. `MonadStateOf σ` provides the same operations, but allows σ to influence instance synthesis.

The mutable state of a state monad is visible between multiple **do**-blocks or functions, unlike local mutable state in **do**-notation.

Instance Constructor

`MonadState.mk.{u, v}`

Methods

`get : m σ`

Retrieves the current value of the monad's mutable state.

`set : σ → m PUnit`

Replaces the current value of the mutable state with a new one.

`modifyGet : {α : Type u} → (σ → α × σ) → m α`

Applies a function to the current state that both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to `do let (a, s) := f (← get); set s; pure a`. However, using `modifyGet` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

def

```
MonadState.get.{u, v} {σ : outParam (Type u)} {m : Type u → Type v}
  [self : MonadState σ m] : m σ
```

Retrieves the current value of the monad's mutable state.

def

```
modify.{u, v} {σ : Type u} {m : Type u → Type v} [MonadState σ m]
  (f : σ → σ) : m PUnit
```

Mutates the current state, replacing its value with the result of applying f to it.

Use `modifyThe` to explicitly select a state type to modify.

It is equivalent to `do set (f (← get))`. However, using `modify` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

def

```
MonadState.modifyGet.{u, v} {σ : outParam (Type u)}
  {m : Type u → Type v} [self : MonadState σ m] {α : Type u} :
  (σ → α × σ) → m α
```

Applies a function to the current state that both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to `do let (a, s) := f (← get); set s; pure a`. However, using `modifyGet` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

def

```
getModify.{u, v} {σ : Type u} {m : Type u → Type v} [MonadState σ
m]
  (f : σ → σ) : m σ
```

Replaces the state with the result of applying f to it. Returns the old value of the state.

It is equivalent to `get <* modify f` but may be more efficient.

type class

```
MonadStateOf.{u, v} (σ : semiOutParam (Type u)) (m : Type u → Type v) :
  Type (max (u + 1) v)
```

State monads provide a value of a given type (the *state*) that can be retrieved or replaced. Instances may implement these operations by passing state values around, by using a mutable reference cell (e.g. `ST.Ref σ`), or in other ways.

In this class, σ is a `semiOutParam`, which means that it can influence the choice of instance. `MonadState σ` provides the same operations, but requires that σ be inferable from m.

The mutable state of a state monad is visible between multiple **do**-blocks or functions, unlike local mutable state in **do**-notation.

Instance Constructor

`MonadStateOf.mk.{u, v}`

Methods

`get : m σ`

Retrieves the current value of the monad's mutable state.

`set : σ → m PUnit`

Replaces the current value of the mutable state with a new one.

`modifyGet : {α : Type u} → (σ → α × σ) → m α`

Applies a function to the current state that both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to `do let (a, s) := f (← get); set s; pure a`. However, using `modifyGet` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

def

```
getThe.{u, v} (σ : Type u) {m : Type u → Type v} [MonadStateOf σ m] :
  m σ
```

Gets the current state that has the explicitly-provided type σ. When the current monad has multiple state types available, this function selects one of them.

def

```
modifyThe.{u, v} (σ : Type u) {m : Type u → Type v} [MonadStateOf σ m]
  (f : σ → σ) : m PUnit
```

Mutates the current state that has the explicitly-provided type σ, replacing its value with the result of applying f to it. When the current monad has multiple state types available, this function selects one of them.

It is equivalent to `do set (f (← get))`. However, using `modify` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

def

```
modifyGetThe.{u, v} {α : Type u} (σ : Type u) {m : Type u → Type v}
  [MonadStateOf σ m] (f : σ → α × σ) : m α
```

Applies a function to the current state that has the explicitly-provided type σ. The function both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to `do let (a, s) := f (← getThe σ); set s; pure a`. However, using `modifyGetThe` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

14.5.4.2. Tuple-Based State Monads

The tuple-based state monads represent a computation with states that have type σ yielding values of type α as functions that take a starting state and yield a value paired with a final state, e.g. $\sigma \to \alpha \times \sigma$. The `Monad` operations thread the state correctly through the computation.

def

```
StateM.{u} (σ α : Type u) : Type u
```

A tuple-based state monad.

Actions in `StateM σ` are functions that take an initial state and return a value paired with a final state.

def

```
StateT.{u, v} (σ : Type u) (m : Type u → Type v) (α : Type u) :
  Type (max u v)
```

Adds a mutable state of type σ to a monad.

Actions in the resulting monad are functions that take an initial state and return, in *m*, a tuple of a value and a state.

def

```
StateT.run.{u, v} {σ : Type u} {m : Type u → Type v} {α : Type u}
  (x : StateT σ m α) (s : σ) : m (α × σ)
```

Executes an action from a monad with added state in the underlying monad *m*. Given an initial state, it returns a value paired with the final state.

def

```
StateT.get.{u, v} {σ : Type u} {m : Type u → Type v} [Monad m] :
  StateT σ m σ
```

Retrieves the current value of the monad's mutable state.

This increments the reference count of the state, which may inhibit in-place updates.

def

```
StateT.set.{u, v} {σ : Type u} {m : Type u → Type v} [Monad m] :
  σ → StateT σ m PUnit
```

Replaces the mutable state with a new value.

def

```
StateT.orElse.{u, v} {σ : Type u} {m : Type u → Type v}
[Alternative m]
  {α : Type u} (x₁ : StateT σ m α) (x₂ : Unit → StateT σ m α) :
  StateT σ m α
```

Recovers from errors. The state is rolled back on error recovery. Typically used via the <|> operator.

def

```
StateT.failure.{u, v} {σ : Type u} {m : Type u → Type v}
[Alternative m]
  {α : Type u} : StateT σ m α
```

Fails with a recoverable error. The state is rolled back on error recovery.

def

```
StateT.run'.{u, v} {σ : Type u} {m : Type u → Type v} [Functor m]
  {α : Type u} (x : StateT σ m α) (s : σ) : m α
```

Executes an action from a monad with added state in the underlying monad m. Given an initial state, it returns a value, discarding the final state.

def

```
StateT.bind.{u, v} {σ : Type u} {m : Type u → Type v} [Monad m]
  {α β : Type u} (x : StateT σ m α) (f : α → StateT σ m β) :
  StateT σ m β
```

Sequences two actions. Typically used via the >>= operator.

def

```
StateT.modifyGet.{u, v} {σ : Type u} {m : Type u → Type v} [Monad
m]
  {α : Type u} (f : σ → α × σ) : StateT σ m α
```

Applies a function to the current state that both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to `do let (a, s) := f (← StateT.get); StateT.set s; pure a`. However, using `StateT.modifyGet` may lead to better performance because it doesn't add a new reference to the state value, and additional references can inhibit in-place updates of data.

def

```
StateT.lift.{u, v} {σ : Type u} {m : Type u → Type v} [Monad m]
  {α : Type u} (t : m α) : StateT σ m α
```

Runs an action from the underlying monad in the monad with state. The state is not modified.

This function is typically implicitly accessed via a `MonadLiftT` instance as part of automatic lifting.

def

```
StateT.map.{u, v} {σ : Type u} {m : Type u → Type v} [Monad m]
  {α β : Type u} (f : α → β) (x : StateT σ m α) : StateT σ m β
```

Modifies the value returned by a computation. Typically used via the <$> operator.

def

```
StateT.pure.{u, v} {σ : Type u} {m : Type u → Type v} [Monad m]
  {α : Type u} (a : α) : StateT σ m α
```

Returns the given value without modifying the state. Typically used via Pure.pure.

14.5.4.3. State Monads in Continuation Passing Style

Continuation-passing-style state monads represent stateful computations as functions that, for any type whatsoever, take an initial state and a continuation (modeled as a function) that accepts a value and an updated state. An example of such a type is `(δ : Type u) → σ → (α → σ → δ) → δ`, though StateCpsT is a transformer that can be applied to any monad. State monads in continuation passing style have different performance characteristics than tuple-based state monads; for some applications, it may be worth benchmarking them.

def

```
StateCpsT.{u, v} (σ : Type u) (m : Type u → Type v) (α : Type u) :
  Type (max (u + 1) v)
```

An alternative implementation of a state monad transformer that internally uses continuation passing style instead of tuples.

def

```
StateCpsT.lift.{u, v} {α σ : Type u} {m : Type u → Type v} [Monad
m]
  (x : m α) : StateCpsT σ m α
```

Runs an action from the underlying monad in the monad with state. The state is not modified.

This function is typically implicitly accessed via a MonadLiftT instance as part of automatic lifting.

def

```
StateCpsT.runK.{u, v} {α σ : Type u} {m : Type u → Type v} {β :
Type u}
  (x : StateCpsT σ m α) (s : σ) (k : α → σ → m β) : m β
```

Runs a stateful computation that's represented using continuation passing style by providing it with an initial state and a continuation.

def

```
StateCpsT.run'.{u, v} {α σ : Type u} {m : Type u → Type v} [Monad
m]
  (x : StateCpsT σ m α) (s : σ) : m α
```

Executes an action from a monad with added state in the underlying monad *m*. Given an initial state, it returns a value, discarding the final state.

def

```
StateCpsT.run.{u, v} {α σ : Type u} {m : Type u → Type v} [Monad m]
  (x : StateCpsT σ m α) (s : σ) : m (α × σ)
```

Executes an action from a monad with added state in the underlying monad *m*. Given an initial state, it returns a value paired with the final state.

While the state is internally represented in continuation passing style, the resulting value is the same as for a non-CPS state monad.

14.5.4.4. State Monads from Mutable References

The monad **`StateRefT`** *σ m* is a specialized state monad transformer that can be used when *m* is a monad to which `ST` computations can be lifted. It implements the operations of `MonadState` using an `ST.Ref`, rather than pure functions. This ensures that mutation is actually used at run time.

`ST` and `EST` require a phantom type parameter that's used together with `runST`'s polymorphic function argument to encapsulate mutability. Rather than require this as a parameter to the transformer, an auxiliary type class `STWorld` is used to propagate it directly from *m*.

The transformer itself is defined as a syntax extension and an elaborator, rather than an ordinary function. This is because `STWorld` has no methods: it exists only to propagate information from the inner monad to the transformed monad. Nonetheless, its instances are terms; keeping them around could lead to unnecessarily large types.

type class

```
STWorld (σ : outParam Type) (m : Type → Type) : Type
```

An auxiliary class used to infer the “state” of `EST` and `ST` monads.

Instance Constructor

`STWorld.mk`

syntax

```
StateRefT
```

The syntax for **`StateRefT`** *σ m* accepts two arguments:

```
term ::= ...
    | StateRefT term (macroDollarArg
```

```
        | term)
```

Its elaborator synthesizes an instance of `STWorld ω m` to ensure that `m` supports mutable references. Having discovered the value of `ω`, it then produces the term `StateRefT' ω σ m`, discarding the synthesized instance.

def

```
StateRefT' (ω σ : Type) (m : Type → Type) (α : Type) : Type
```

A state monad that uses an actual mutable reference cell (i.e. an `ST.Ref ω σ`).

The macro **`StateRefT`** `σ m α` infers `ω` from `m`. It should normally be used instead.

def

```
StateRefT'.get {ω σ : Type} {m : Type → Type} [MonadLiftT (ST ω) m]
:
  StateRefT' ω σ m σ
```

Retrieves the current value of the monad's mutable state.

This increments the reference count of the state, which may inhibit in-place updates.

def

```
StateRefT'.set {ω σ : Type} {m : Type → Type} [MonadLiftT (ST ω) m]
  (s : σ) : StateRefT' ω σ m PUnit
```

Replaces the mutable state with a new value.

def

```
StateRefT'.modifyGet {ω σ : Type} {m : Type → Type} {α : Type}
  [MonadLiftT (ST ω) m] (f : σ → α × σ) : StateRefT' ω σ m α
```

Applies a function to the current state that both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to a `get` followed by a `set`. However, using `modifyGet` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

def

```
StateRefT'.run {ω σ : Type} {m : Type → Type} [Monad m]
  [MonadLiftT (ST ω) m] {α : Type} (x : StateRefT' ω σ m α) (s : σ)
```

```
:
  m (α × σ)
```

Executes an action from a monad with added state in the underlying monad `m`. Given an initial state, it returns a value paired with the final state.

The monad `m` must support ST effects in order to create and mutate reference cells.

def

```
StateRefT'.run' {ω σ : Type} {m : Type → Type} [Monad m]
  [MonadLiftT (ST ω) m] {α : Type} (x : StateRefT' ω σ m α) (s : σ)
:
  m α
```

Executes an action from a monad with added state in the underlying monad `m`. Given an initial state, it returns a value, discarding the final state.

The monad `m` must support ST effects in order to create and mutate reference cells.

def

```
StateRefT'.lift {ω σ : Type} {m : Type → Type} {α : Type} (x : m α)
:
  StateRefT' ω σ m α
```

Runs an action from the underlying monad in the monad with state. The state is not modified.

This function is typically implicitly accessed via a `MonadLiftT` instance as part of automatic lifting.

14.5.5. Reader

type class

```
MonadReader.{u, v} (ρ : outParam (Type u)) (m : Type u → Type v) :
  Type v
```

Reader monads provide the ability to implicitly thread a value through a computation. The value can be read, but not written. A `MonadWithReader ρ` instance additionally allows the value to be locally overridden for a sub-computation.

In this class, `ρ` is an `outParam`, which means that it is inferred from `m`. `MonadReaderOf ρ` provides the same operations, but allows `ρ` to influence instance synthesis.

Instance Constructor

`MonadReader.mk.{u, v}`

Methods

`read : m ρ`

Retrieves the local value.

Use `readThe` to explicitly specify a type when more than one value is available.

type class

```
MonadReaderOf.{u, v} (ρ : semiOutParam (Type u)) (m : Type u → Type v) :
  Type v
```

Reader monads provide the ability to implicitly thread a value through a computation. The value can be read, but not written. A `MonadWithReader ρ` instance additionally allows the value to be locally overridden for a sub-computation.

In this class, ρ is a `semiOutParam`, which means that it can influence the choice of instance. `MonadReader ρ` provides the same operations, but requires that ρ be inferable from m.

Instance Constructor

`MonadReaderOf.mk.{u, v}`

Methods

`read : m ρ`

Retrieves the local value.

def

```
readThe.{u, v} (ρ : Type u) {m : Type u → Type v} [MonadReaderOf ρ m] :
  m ρ
```

Retrieves the local value whose type is ρ. This is useful when a monad supports reading more that one type of value.

Use `read` for a version that expects the type ρ to be inferred from m.

type class

```
MonadWithReader.{u, v} (ρ : outParam (Type u)) (m : Type u → Type v) :
  Type (max (u + 1) v)
```

A reader monad that additionally allows the value to be locally overridden.

In this class, ρ is an outParam, which means that it is inferred from m. MonadWithReaderOf ρ provides the same operations, but allows ρ to influence instance synthesis.

Instance Constructor

MonadWithReader.mk.{u, v}

Methods

```
withReader : {α : Type u} → (ρ → ρ) → m α → m α
```

Locally modifies the reader monad's value while running an action.

During the inner action x, reading the value returns f applied to the original value. After control returns from x, the reader monad's value is restored.

type class

```
MonadWithReaderOf.{u, v} (ρ : semiOutParam (Type u))
  (m : Type u → Type v) : Type (max (u + 1) v)
```

A reader monad that additionally allows the value to be locally overridden.

In this class, ρ is a semiOutParam, which means that it can influence the choice of instance. MonadWithReader ρ provides the same operations, but requires that ρ be inferable from m.

Instance Constructor

MonadWithReaderOf.mk.{u, v}

Methods

```
withReader : {α : Type u} → (ρ → ρ) → m α → m α
```

Locally modifies the reader monad's value while running an action.

During the inner action x, reading the value returns f applied to the original value. After control returns from x, the reader monad's value is restored.

def

```
withTheReader.{u, v} (ρ : Type u) {m : Type u → Type v}
  [MonadWithReaderOf ρ m] {α : Type u} (f : ρ → ρ) (x : m α) : m α
```

Locally modifies the reader monad's value while running an action, with the reader monad's local value type specified explicitly. This is useful when a monad supports reading more than one type of value.

During the inner action *x*, reading the value returns *f* applied to the original value. After control returns from *x*, the reader monad's value is restored.

Use withReader for a version that expects the local value's type to be inferred from *m*.

def

```
ReaderT.{u, v} (ρ : Type u) (m : Type u → Type v) (α : Type u) :
  Type (max u v)
```

Adds the ability to access a read-only value of type *ρ* to a monad. The value can be locally overridden by withReader, but it cannot be mutated.

Actions in the resulting monad are functions that take the local value as a parameter, returning ordinary actions in *m*.

def

```
ReaderM.{u} (ρ α : Type u) : Type u
```

A monad with access to a read-only value of type *ρ*. The value can be locally overridden by withReader, but it cannot be mutated.

def

```
ReaderT.run.{u, v} {ρ : Type u} {m : Type u → Type v} {α : Type u}
  (x : ReaderT ρ m α) (r : ρ) : m α
```

Executes an action from a monad with a read-only value in the underlying monad *m*.

def

```
ReaderT.read.{u, v} {ρ : Type u} {m : Type u → Type v} [Monad m] :
  ReaderT ρ m ρ
```

Retrieves the reader monad's local value. Typically accessed via read, or via readThe when more than one local value is available.

def

```
ReaderT.adapt.{u, v} {ρ : Type u} {m : Type u → Type v} {ρ' α :
Type u}
  (f : ρ' → ρ) : ReaderT ρ m α → ReaderT ρ' m α
```

Modifies a reader monad's local value with f. The resulting computation applies f to the incoming local value and passes the result to the inner computation.

def

```
ReaderT.pure.{u, v} {ρ : Type u} {m : Type u → Type v} [Monad m]
  {α : Type u} (a : α) : ReaderT ρ m α
```

Returns the provided value a, ignoring the reader monad's local value. Typically used via Pure.pure.

def

```
ReaderT.bind.{u, v} {ρ : Type u} {m : Type u → Type v} [Monad m]
  {α β : Type u} (x : ReaderT ρ m α) (f : α → ReaderT ρ m β) :
  ReaderT ρ m β
```

Sequences two reader monad computations. Both are provided with the local value, and the second is passed the value of the first. Typically used via the >>= operator.

def

```
ReaderT.orElse.{u_1, u_2} {m : Type u_1 → Type u_2} {ρ α : Type
u_1}
  [Alternative m] (x₁ : ReaderT ρ m α) (x₂ : Unit → ReaderT ρ m α)
:
  ReaderT ρ m α
```

Recovers from errors. The same local value is provided to both branches. Typically used via the <|> operator.

def

```
ReaderT.failure.{u_1, u_2} {m : Type u_1 → Type u_2} {ρ α : Type
u_1}
  [Alternative m] : ReaderT ρ m α
```

Fails with a recoverable error.

14.5.6. Option

Ordinarily, Option is thought of as data, similarly to a nullable type. It can also be considered as a monad, and thus a way of performing computations. The Option monad and its transformer OptionT can be understood as describing computations that may terminate early, discarding the

results. Callers can check for early termination and invoke a fallback if desired using `OrElse.orElse` or by treating it as a `MonadExcept Unit`.

def

```
OptionT.{u, v} (m : Type u → Type v) (α : Type u) : Type v
```

Adds the ability to fail to a monad. Unlike ordinary exceptions, there is no way to signal why a failure occurred.

def

```
OptionT.run.{u, v} {m : Type u → Type v} {α : Type u}
  (x : OptionT m α) : m (Option α)
```

Executes an action that might fail in the underlying monad `m`, returning `none` in case of failure.

def

```
OptionT.lift.{u, v} {m : Type u → Type v} [Monad m] {α : Type u}
  (x : m α) : OptionT m α
```

Converts a computation from the underlying monad into one that could fail, even though it does not.

This function is typically implicitly accessed via a `MonadLiftT` instance as part of automatic lifting.

def

```
OptionT.mk.{u, v} {m : Type u → Type v} {α : Type u}
  (x : m (Option α)) : OptionT m α
```

Converts an action that returns an `Option` into one that might fail, with `none` indicating failure.

def

```
OptionT.pure.{u, v} {m : Type u → Type v} [Monad m] {α : Type u}
  (a : α) : OptionT m α
```

Succeeds with the provided value.

def

```
OptionT.bind.{u, v} {m : Type u → Type v} [Monad m] {α β : Type u}
  (x : OptionT m α) (f : α → OptionT m β) : OptionT m β
```

Sequences two potentially-failing actions. The second action is run only if the first succeeds.

def

```
OptionT.fail.{u, v} {m : Type u → Type v} [Monad m] {α : Type u} :
  OptionT m α
```

A recoverable failure.

def

```
OptionT.orElse.{u, v} {m : Type u → Type v} [Monad m] {α : Type u}
  (x : OptionT m α) (y : Unit → OptionT m α) : OptionT m α
```

Recovers from failures. Typically used via the <|> operator.

def

```
OptionT.tryCatch.{u, v} {m : Type u → Type v} [Monad m] {α : Type
u}
  (x : OptionT m α) (handle : Unit → OptionT m α) : OptionT m α
```

Handles failures by treating them as exceptions of type `Unit`.

14.5.7. Exceptions

Exception monads describe computations that terminate early (fail). Failing computations provide their caller with an *exception* value that describes *why* they failed. In other words, computations either return a value or an exception. The inductive type `Except` captures this pattern, and is itself a monad.

14.5.7.1. Exceptions

inductive type

```
Except.{u, v} (ε : Type u) (α : Type v) : Type (max u v)
```

`Except ε α` is a type which represents either an error of type ε or a successful result with a value of type α.

`Except ε : Type u → Type v` is a `Monad` that represents computations that may throw exceptions: the `pure` operation is `Except.ok` and the `bind` operation returns the first encountered `Except.error`.

Constructors

```
| error.{u, v} {ε : Type u} {α : Type v} : ε → Except ε α
```

A failure value of type ε

```
| ok.{u, v} {ε : Type u} {α : Type v} : α → Except ε α
```

A success value of type α

def

```
Except.pure.{u, u_1} {ε : Type u} {α : Type u_1} (a : α) : Except ε α
```

A successful computation in the `Except ε` monad: a is returned, and no exception is thrown.

def

```
Except.bind.{u, u_1, u_2} {ε : Type u} {α : Type u_1} {β : Type u_2}
  (ma : Except ε α) (f : α → Except ε β) : Except ε β
```

Sequences two operations that may throw exceptions, allowing the second to depend on the value returned by the first.

If the first operation throws an exception, then it is the result of the computation. If the first succeeds but the second throws an exception, then that exception is the result. If both succeed, then the result is the result of the second computation.

This is the implementation of the >>= operator for `Except ε`.

def

```
Except.map.{u, u_1, u_2} {ε : Type u} {α : Type u_1} {β : Type u_2}
  (f : α → β) : Except ε α → Except ε β
```

Transforms a successful result with a function, doing nothing when an exception is thrown.

Examples:

- `(pure 2 : Except String Nat).map toString = pure 2`
- `(throw "Error" : Except String Nat).map toString = throw "Error"`

def

```
Except.mapError.{u, u_1, u_2} {ε : Type u} {ε' : Type u_1}
  {α : Type u_2} (f : ε → ε') : Except ε α → Except ε' α
```

Transforms exceptions with a function, doing nothing on successful results.

Examples:

- `(pure 2 : Except String Nat).mapError (·.length) = pure 2`
- `(throw "Error" : Except String Nat).mapError (·.length) = throw 5`

def

```
Except.tryCatch.{u, u_1} {ε : Type u} {α : Type u_1} (ma : Except ε α)
  (handle : ε → Except ε α) : Except ε α
```

Handles exceptions thrown in the `Except ε` monad.

If *ma* is successful, its result is returned. If it throws an exception, then *handle* is invoked on the exception's value.

Examples:

- `(pure 2 : Except String Nat).tryCatch (pure ·.length) = pure 2`
- `(throw "Error" : Except String Nat).tryCatch (pure ·.length) = pure 5`
- `(throw "Error" : Except String Nat).tryCatch (fun x => throw ("E: " ++ x)) = throw "E: Error"`

def

```
Except.orElseLazy.{u, u_1} {ε : Type u} {α : Type u_1} (x : Except ε α)
  (y : Unit → Except ε α) : Except ε α
```

Recovers from exceptions thrown in the `Except ε` monad. Typically used via the <|> operator.

`Except.tryCatch` is a related operator that allows the recovery procedure to depend on *which* exception was thrown.

def

```
Except.isOk.{u, u_1} {ε : Type u} {α : Type u_1} : Except ε α → Bool
```

Returns `true` if the value is `Except.ok`, `false` otherwise.

def

```
Except.toOption.{u, u_1} {ε : Type u} {α : Type u_1} :
  Except ε α → Option α
```

Returns `none` if an exception was thrown, or `some` around the value on success.

Examples:

- `(pure 10 : Except String Nat).toOption = some 10`
- `(throw "Failure" : Except String Nat).toOption = none`

def

```
Except.toBool.{u, u_1} {ε : Type u} {α : Type u_1} : Except ε α →
Bool
```

Returns `true` if the value is `Except.ok`, `false` otherwise.

14.5.7.2. Type Class

type class

```
MonadExcept.{u, v, w} (ε : outParam (Type u)) (m : Type v → Type w)
:
  Type (max (max u (v + 1)) w)
```

Exception monads provide the ability to throw errors and handle errors.

In this class, ε is an `outParam`, which means that it is inferred from m. `MonadExceptOf ε` provides the same operations, but allows ε to influence instance synthesis.

`MonadExcept.tryCatch` is used to desugar **`try ... catch ...`** steps inside **`do`**-blocks when the handlers do not have exception type annotations.

Instance Constructor

`MonadExcept.mk.{u, v, w}`

Methods

`throw : {α : Type v} → ε → m α`

Throws an exception of type ε to the nearest enclosing handler.

`tryCatch : {α : Type v} → m α → (ε → m α) → m α`

Catches errors thrown in `body`, passing them to `handler`. Errors in `handler` are not caught.

def

```
MonadExcept.ofExcept.{u_1, u_2, u_3} {m : Type u_1 → Type u_2}
  {ε : Type u_3} {α : Type u_1} [Monad m] [MonadExcept ε m] :
  Except ε α → m α
```

Re-interprets an `Except ε` action in an exception monad `m`, succeeding if it succeeds and throwing an exception if it throws an exception.

def

```
MonadExcept.orElse.{u, v, w} {ε : Type u} {m : Type v → Type w}
  [MonadExcept ε m] {α : Type v} (t₁ : m α) (t₂ : Unit → m α) : m α
```

Unconditional error recovery that ignores which exception was thrown. Usually used via the <|> operator.

If both computations throw exceptions, then the result is the second exception.

def

```
MonadExcept.orelse'.{u, v, w} {ε : Type u} {m : Type v → Type w}
  [MonadExcept ε m] {α : Type v} (t₁ t₂ : m α)
  (useFirstEx : Bool := true) : m α
```

An alternative unconditional error recovery operator that allows callers to specify which exception to throw in cases where both operations throw exceptions.

By default, the first is thrown, because the <|> operator throws the second.

type class

```
MonadExceptOf.{u, v, w} (ε : semiOutParam (Type u))
  (m : Type v → Type w) : Type (max (max u (v + 1)) w)
```

Exception monads provide the ability to throw errors and handle errors.

In this class, `ε` is a `semiOutParam`, which means that it can influence the choice of instance. `MonadExcept ε` provides the same operations, but requires that `ε` be inferable from `m`.

`tryCatchThe`, which takes an explicit exception type, is used to desugar `try ... catch ...` steps inside **do**-blocks when the handlers have type annotations.

Instance Constructor

`MonadExceptOf.mk.{u, v, w}`

Methods

```
throw : {α : Type v} → ε → m α
```

Throws an exception of type ε to the nearest enclosing **`catch`**.

```
tryCatch : {α : Type v} → m α → (ε → m α) → m α
```

Catches errors thrown in `body`, passing them to `handler`. Errors in `handler` are not caught.

def

```
throwThe.{u, v, w} (ε : Type u) {m : Type v → Type w}
  [MonadExceptOf ε m] {α : Type v} (e : ε) : m α
```

Throws an exception, with the exception type specified explicitly. This is useful when a monad supports throwing more than one type of exception.

Use `throw` for a version that expects the exception type to be inferred from m.

def

```
tryCatchThe.{u, v, w} (ε : Type u) {m : Type v → Type w}
  [MonadExceptOf ε m] {α : Type v} (x : m α) (handle : ε → m α) : m
α
```

Catches errors, recovering using *`handle`*. The exception type is specified explicitly. This is useful when a monad supports throwing or handling more than one type of exception.

Use `tryCatch`, for a version that expects the exception type to be inferred from m.

14.5.7.3. “Finally” Computations

type class

```
MonadFinally.{u, v} (m : Type u → Type v) : Type (max (u + 1) v)
```

Monads that provide the ability to ensure an action happens, regardless of exceptions or other failures.

`MonadFinally.tryFinally'` is used to desugar **`try ... finally ...`** syntax.

Instance Constructor

`MonadFinally.mk.{u, v}`

Methods

`tryFinally' : {α β : Type u} → m α → (Option α → m β) → m (α × β)`

Runs an action, ensuring that some other action always happens afterward.

More specifically, `tryFinally' x f` runs *x* and then the "finally" computation *f*. If *x* succeeds with some value `a : α`, `f (some a)` is returned. If *x* fails for *m*'s definition of failure, `f none` is returned.

`tryFinally'` can be thought of as performing the same role as a **`finally`** block in an imperative programming language.

14.5.7.4. Transformer

def

```
ExceptT.{u, v} (ε : Type u) (m : Type u → Type v) (α : Type u) :
Type v
```

Adds exceptions of type *ε* to a monad *m*.

def

```
ExceptT.lift.{u, v} {ε : Type u} {m : Type u → Type v} [Monad m]
  {α : Type u} (t : m α) : ExceptT ε m α
```

Runs a computation from an underlying monad in the transformed monad with exceptions.

def

```
ExceptT.run.{u, v} {ε : Type u} {m : Type u → Type v} {α : Type u}
  (x : ExceptT ε m α) : m (Except ε α)
```

Use a monadic action that may throw an exception as an action that may return an exception's value.

This is the inverse of `ExceptT.mk`.

def

```
ExceptT.pure.{u, v} {ε : Type u} {m : Type u → Type v} [Monad m]
  {α : Type u} (a : α) : ExceptT ε m α
```

Returns the value *a* without throwing exceptions or having any other effect.

def

```
ExceptT.bind.{u, v} {ε : Type u} {m : Type u → Type v} [Monad m]
  {α β : Type u} (ma : ExceptT ε m α) (f : α → ExceptT ε m β) :
  ExceptT ε m β
```

Sequences two actions that may throw exceptions. Typically used via **do**-notation or the >>= operator.

def

```
ExceptT.bindCont.{u, v} {ε : Type u} {m : Type u → Type v} [Monad
m]
  {α β : Type u} (f : α → ExceptT ε m β) : Except ε α → m (Except ε
β)
```

Handles exceptions thrown by an action that can have no effects *other* than throwing exceptions.

def

```
ExceptT.tryCatch.{u, v} {ε : Type u} {m : Type u → Type v} [Monad
m]
  {α : Type u} (ma : ExceptT ε m α) (handle : ε → ExceptT ε m α) :
  ExceptT ε m α
```

Handles exceptions produced in the `ExceptT ε` transformer.

def

```
ExceptT.mk.{u, v} {ε : Type u} {m : Type u → Type v} {α : Type u}
  (x : m (Except ε α)) : ExceptT ε m α
```

Use a monadic action that may return an exception's value as an action in the transformed monad that may throw the corresponding exception.

This is the inverse of `ExceptT.run`.

def

```
ExceptT.map.{u, v} {ε : Type u} {m : Type u → Type v} [Monad m]
  {α β : Type u} (f : α → β) (x : ExceptT ε m α) : ExceptT ε m β
```

Transforms a successful computation's value using `f`. Typically used via the <$> operator.

def

```
ExceptT.adapt.{u, v} {ε : Type u} {m : Type u → Type v} [Monad m]
  {ε' α : Type u} (f : ε → ε') : ExceptT ε m α → ExceptT ε' m α
```

Transforms exceptions using the function f.

This is the ExceptT version of Except.mapError.

14.5.7.5. Exception Monads in Continuation Passing Style

Continuation-passing-style exception monads represent potentially-failing computations as functions that take success and failure continuations, both of which return the same type, returning that type. They must work for *any* return type. An example of such a type is $(\beta : \text{Type u}) \to (\alpha \to \beta) \to (\varepsilon \to \beta) \to \beta$. ExceptCpsT is a transformer that can be applied to any monad, so ExceptCpsT ε m α is actually defined as $(\beta : \text{Type u}) \to (\alpha \to m\ \beta) \to (\varepsilon \to m\ \beta) \to m\ \beta$. Exception monads in continuation passing style have different performance characteristics than Except-based state monads; for some applications, it may be worth benchmarking them.

def

```
ExceptCpsT.{u, v} (ε : Type u) (m : Type u → Type v) (α : Type u) :
  Type (max (u + 1) v)
```

Adds exceptions of type ε to a monad m.

Instead of using Except ε to model exceptions, this implementation uses continuation passing style. This has different performance characteristics from ExceptT ε.

def

```
ExceptCpsT.runCatch.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type
u_1}
  [Monad m] (x : ExceptCpsT α m α) : m α
```

Returns the value of a computation, forgetting whether it was an exception or a success.

This corresponds to early return.

def

```
ExceptCpsT.runK.{u, u_1} {m : Type u → Type u_1} {β ε α : Type u}
  (x : ExceptCpsT ε m α) (s : ε) (ok : α → m β) (error : ε → m β) :
m β
```

Use a monadic action that may throw an exception by providing explicit success and failure continuations.

def

```
ExceptCpsT.run.{u, u_1} {m : Type u → Type u_1} {ε α : Type u}
[Monad m]
```

```
  (x : ExceptCpsT ε m α) : m (Except ε α)
```

Use a monadic action that may throw an exception as an action that may return an exception's value.

def

```
ExceptCpsT.lift.{u_1, u_2} {m : Type u_1 → Type u_2} {α ε : Type
u_1}
  [Monad m] (x : m α) : ExceptCpsT ε m α
```

Run an action from the transformed monad in the exception monad.

14.5.8. Combined Error and State Monads

The EStateM monad has both exceptions and mutable state. EStateM ε σ α is logically equivalent to ExceptT ε (StateM σ) α. While ExceptT ε (StateM σ) evaluates to the type σ → Except ε α × σ, the type EStateM ε σ α evaluates to σ → EStateM.Result ε σ α. EStateM.Result is an inductive type that's very similar to Except, except both constructors have an additional field for the state. In compiled code, this representation removes one level of indirection from each monadic bind.

def

```
EStateM.{u} (ε σ α : Type u) : Type u
```

A combined state and exception monad in which exceptions do not automatically roll back the state.

Instances of EStateM.Backtrackable provide a way to roll back some part of the state if needed.

EStateM ε σ is equivalent to ExceptT ε (StateM σ), but it is more efficient.

inductive type

```
EStateM.Result.{u} (ε σ α : Type u) : Type u
```

The value returned from a combined state and exception monad in which exceptions do not automatically roll back the state.

Result ε σ α is equivalent to Except ε α × σ, but using a single combined inductive type yields a more efficient data representation.

Constructors

|

```
ok.{u} {ε σ α : Type u} : α → σ → EStateM.Result ε σ α
```

A success value of type α and a new state σ.

```
| error.{u} {ε σ α : Type u} : ε → σ → EStateM.Result ε σ α
```

An exception of type ε and a new state σ.

def

```
EStateM.run.{u} {ε σ α : Type u} (x : EStateM ε σ α) (s : σ) :
  EStateM.Result ε σ α
```

Executes an EStateM action with the initial state s. The returned value includes the final state and indicates whether an exception was thrown or a value was returned.

def

```
EStateM.run'.{u} {ε σ α : Type u} (x : EStateM ε σ α) (s : σ) :
Option α
```

Executes an EStateM with the initial state s for the returned value α, discarding the final state. Returns none if an unhandled exception was thrown.

def

```
EStateM.adaptExcept.{u} {ε σ α ε' : Type u} (f : ε → ε')
  (x : EStateM ε σ α) : EStateM ε' σ α
```

Transforms exceptions with a function, doing nothing on successful results.

def

```
EStateM.fromStateM {ε σ α : Type} (x : StateM σ α) : EStateM ε σ α
```

Converts a state monad action into a state monad action with exceptions.

The resulting action does not throw an exception.

14.5.8.1. State Rollback

Composing StateT and ExceptT in different orders causes exceptions to interact differently with state. In one ordering, state changes are rolled back when exceptions are caught; in the other, they persist. The latter option matches the semantics of most imperative programming languages, but the former is very useful for search-based problems. Often, some but not all state should be rolled back; this can be achieved by “sandwiching” ExceptT between two separate uses of StateT.

To avoid yet another layer of indirection via the use of `StateT σ (EStateM ε σ') α`, `EStateM` offers the `EStateM.Backtrackable` type class. This class specifies some part of the state that can be saved and restored. `EStateM` then arranges for the saving and restoring to take place around error handling.

type class

```
EStateM.Backtrackable.{u} (δ : outParam (Type u)) (σ : Type u) :
Type u
```

Exception handlers in `EStateM` save some part of the state, determined by δ, and restore it if an exception is caught. By default, δ is `Unit`, and no information is saved.

Instance Constructor

`EStateM.Backtrackable.mk.{u}`

Methods

`save : σ → δ`

Extracts the information in the state that should be rolled back if an exception is handled.

`restore : σ → δ → σ`

Updates the current state with the saved information that should be rolled back. This updated state becomes the current state when an exception is handled.

There is a universally-applicable instance of `Backtrackable` that neither saves nor restores anything. Because instance synthesis chooses the most recent instance first, the universal instance is used only if no other instance has been defined.

def

```
EStateM.nonBacktrackable.{u} {σ : Type u} :
  EStateM.Backtrackable PUnit σ
```

A fallback `Backtrackable` instance that saves no information from a state. This allows every type to be used as a state in `EStateM`, with no rollback.

Because this is the first declared instance of `Backtrackable _ σ`, it will be picked only if there are no other `Backtrackable _ σ` instances registered.

14.5.8.2. Implementations

These functions are typically not called directly, but rather are accessed through their corresponding type classes.

def

```
EStateM.map.{u} {ε σ α β : Type u} (f : α → β) (x : EStateM ε σ α)
:
  EStateM ε σ β
```

Transforms the value returned from an `EStateM ε σ` action using a function.

def

```
EStateM.pure.{u} {ε σ α : Type u} (a : α) : EStateM ε σ α
```

Returns a value without modifying the state or throwing an exception.

def

```
EStateM.bind.{u} {ε σ α β : Type u} (x : EStateM ε σ α)
  (f : α → EStateM ε σ β) : EStateM ε σ β
```

Sequences two `EStateM ε σ` actions, passing the returned value from the first into the second.

def

```
EStateM.orElse.{u} {ε σ α δ : Type u} [EStateM.Backtrackable δ σ]
  (x₁ : EStateM ε σ α) (x₂ : Unit → EStateM ε σ α) : EStateM ε σ α
```

Failure handling that does not depend on specific exception values.

The `Backtrackable δ σ` instance is used to save a snapshot of part of the state prior to running x_1. If an exception is caught, the state is updated with the saved snapshot, rolling back part of the state. If no instance of *`Backtrackable`* is provided, a fallback instance in which δ is `Unit` is used, and no information is rolled back.

def

```
EStateM.orElse'.{u} {ε σ α δ : Type u} [EStateM.Backtrackable δ σ]
  (x₁ x₂ : EStateM ε σ α) (useFirstEx : Bool := true) : EStateM ε σ
α
```

Alternative orElse operator that allows callers to select which exception should be used when both operations fail. The default is to use the first exception since the standard `orElse` uses the second.

def

```
EStateM.seqRight.{u} {ε σ α β : Type u} (x : EStateM ε σ α)
  (y : Unit → EStateM ε σ β) : EStateM ε σ β
```

Sequences two `EStateM ε σ` actions, running *x* before *y*. The first action's return value is ignored.

def

```
EStateM.tryCatch.{u} {ε σ δ : Type u} [EStateM.Backtrackable δ σ]
  {α : Type u} (x : EStateM ε σ α) (handle : ε → EStateM ε σ α) :
  EStateM ε σ α
```

Handles exceptions thrown in the combined error and state monad.

The `Backtrackable δ σ` instance is used to save a snapshot of part of the state prior to running *x*. If an exception is caught, the state is updated with the saved snapshot, rolling back part of the state. If no instance of *`Backtrackable`* is provided, a fallback instance in which *δ* is `Unit` is used, and no information is rolled back.

def

```
EStateM.throw.{u} {ε σ α : Type u} (e : ε) : EStateM ε σ α
```

Throws an exception of type *ε* to the nearest enclosing handler.

def

```
EStateM.get.{u} {ε σ : Type u} : EStateM ε σ σ
```

Retrieves the current value of the monad's mutable state.

def

```
EStateM.set.{u} {ε σ : Type u} (s : σ) : EStateM ε σ PUnit
```

Replaces the current value of the mutable state with a new one.

def

```
EStateM.modifyGet.{u} {ε σ α : Type u} (f : σ → α × σ) : EStateM ε
σ α
```

Applies a function to the current state that both computes a new state and a value. The new state replaces the current state, and the value is returned.

It is equivalent to `do let (a, s) := f (← get); set s; pure a`. However, using `modifyGet` may lead to higher performance because it doesn't add a new reference to the state value. Additional references can inhibit in-place updates of data.

15. IO

Lean is a pure functional programming language. While Lean code is strictly evaluated at run time, the order of evaluation that is used during type checking, especially while checking definitional equality, is formally unspecified and makes use of a number of heuristics that improve performance but are subject to change. This means that simply adding operations that perform side effects (such as file I/O, exceptions, or mutable references) would lead to programs in which the order of effects is unspecified. During type checking, even terms with free variables are reduced; this would make side effects even more difficult to predict. Finally, a basic principle of Lean's logic is that functions are *functions* that map each element of the domain to a unique element of the range. Including side effects such as console I/O, arbitrary mutable state, or random number generation would violate this principle.

Programs that may have side effects have a type (typically `IO α`) that distinguishes them from pure functions. Logically speaking, `IO` describes the sequencing and data dependencies of side effects. Many of the basic side effects, such as reading from files, are opaque constants from the perspective of Lean's logic. Others are specified by code that is logically equivalent to the run-time version. At run time, the compiler produces ordinary code.

15.1. Logical Model

Conceptually, Lean distinguishes evaluation or reduction of terms from *execution* of side effects. Term reduction is specified by rules such as β and δ, which may occur anywhere at any time. Side effects, which must be executed in the correct order, are abstractly described in Lean's logic. When programs are run, the Lean runtime system is responsible for actually carrying out the described effects.

The type `IO α` is a description of a process that, by performing side effects, should either return a value of type α or throw an error. It can be thought of as a state monad in which the state is the entire world. Just as a value of type `StateM Nat Bool` computes a `Bool` while having the ability to mutate a natural number, a value of type `IO Bool` computes a `Bool` while potentially changing the world. Error handling is accomplished by layering an appropriate exception monad transformer on top of this.

Because the entire world can't be represented in memory, the actual implementation uses an abstract token that stands for its state. The Lean runtime system is responsible for providing the initial token when the program is run, and each primitive action accepts a token that represents the world and returns another when finished. This ensures that effects occur in the proper order, and it clearly separates the execution of side effects from the reduction semantics of Lean terms.

Non-termination via general recursion is treated separately from the effects described by `IO`. Programs that may not terminate due to infinite loops must be defined as `partial` functions. From the logical perspective, they are treated as arbitrary constants; `IO` is not needed.

A very important property of `IO` is that there is no way for values to “escape”. Without using one of a few clearly-marked unsafe operators, programs have no way to extract a pure `Nat` from an `IO Nat`. This ensures that the correct ordering of side effects is preserved, and it ensures that programs that have side effects are clearly marked as such.

15.1.1. The `IO`, `EIO` and `BaseIO` Monads

There are two monads that are typically used for programs that interact with the real world:

- Actions in `IO` may throw exceptions of type `IO.Error` or modify the world.
- Actions in `BaseIO` can't throw exceptions, but they can modify the world.

The distinction makes it possible to tell whether exceptions are possible by looking at an action's type signature. `BaseIO` actions are automatically promoted to `IO` as necessary.

def

```
IO : Type → Type
```

A monad that supports arbitrary side effects and throwing exceptions of type `IO.Error`.

def

```
BaseIO : Type → Type
```

An `IO` monad that cannot throw exceptions.

Both `IO` and `BaseIO` are instances of `EIO`, in which the type of errors is a parameter. `IO` is defined as `EIO IO.Error`, while `BaseIO` is defined as `EIO Empty`. In some circumstances, such as bindings to non-Lean libraries, it can be convenient to use `EIO` with a custom error type, which ensures that errors are handled at the boundaries between these and other `IO` actions.

def

```
EIO (ε : Type) : Type → Type
```

A monad that can have side effects on the external world or throw exceptions of type ε.

`BaseIO` is a version of this monad that cannot throw exceptions. `IO` sets the exception type to `IO.Error`.

def

```
IO.lazyPure {α : Type} (fn : Unit → α) : IO α
```

Creates an IO action that will invoke fn if and when it is executed, returning the result.

def

```
BaseIO.toIO {α : Type} (act : BaseIO α) : IO α
```

Runs a `BaseIO` action, which cannot throw an exception, as an `IO` action.

This function is usually used implicitly via automatic monadic lifting rather than being called explicitly.

def

```
BaseIO.toEIO {α ε : Type} (act : BaseIO α) : EIO ε α
```

Runs a `BaseIO` action, which cannot throw an exception, in any other `EIO` monad.

This function is usually used implicitly via automatic monadic lifting rather being than called explicitly.

def

```
EIO.toBaseIO {ε α : Type} (act : EIO ε α) : BaseIO (Except ε α)
```

Converts an `EIO ε` action that might throw an exception of type ε into an exception-free `BaseIO` action that returns an `Except` value.

def

```
EIO.toIO {ε α : Type} (f : ε → IO.Error) (act : EIO ε α) : IO α
```

Converts an `EIO ε` action into an `IO` action by translating any exceptions that it throws into `IO.Error`s using f.

def

```
EIO.toIO' {ε α : Type} (act : EIO ε α) : IO (Except ε α)
```

Converts an `EIO ε` action that might throw an exception of type ε into an exception-free `IO` action that returns an `Except` value.

def

```
IO.toEIO {ε α : Type} (f : IO.Error → ε) (act : IO α) : EIO ε α
```

Runs an `IO` action in some other `EIO` monad, using f to translate `IO` exceptions.

15.1.2. Errors and Error Handling in IO

Error handling in the IO monad uses the same facilities as any other exception monad. In particular, throwing and catching exceptions uses the methods of the MonadExceptOf type class. The exceptions thrown in IO have the type IO.Error. The constructors of this type represent the low-level errors that occur on most operating systems, such as files not existing. The most-used constructor is userError, which covers all other cases and includes a string that describes the problem.

inductive type

```
IO.Error : Type
```

Exceptions that may be thrown in the IO monad.

Many of the constructors of IO.Error correspond to POSIX error numbers. In these cases, the documentation string lists POSIX standard error macros that correspond to the error. This list is not necessarily exhaustive, and these constructor includes a field for the underlying error number.

Constructors

| `alreadyExists (filename : Option String) (osCode : UInt32) (details : String) : IO.Error`

The operation failed because a file already exists.

This corresponds to POSIX errors EEXIST, EINPROGRESS, and EISCONN.

| `otherError (osCode : UInt32) (details : String) : IO.Error`

Some error not covered by the other constructors of IO.Error occurred.

This also includes POSIX error EFAULT.

| `resourceBusy (osCode : UInt32) (details : String) : IO.Error`

A necessary resource was busy.

This corresponds to POSIX errors EADDRINUSE, EBUSY, EDEADLK, and ETXTBSY.

| `resourceVanished (osCode : UInt32) (details : String) : IO.Error`

A necessary resource is no longer available.

This corresponds to POSIX errors ECONNRESET, EIDRM, ENETDOWN, ENETRESET, ENOLINK, and EPIPE.

|

```
unsupportedOperation (osCode : UInt32) (details : String) :
  IO.Error
```

An operation was not supported.

This corresponds to POSIX errors EADDRNOTAVAIL, EAFNOSUPPORT, ENODEV, ENOPROTOOPT ENOSYS, EOPNOTSUPP, ERANGE, ESPIPE, and EXDEV.

```
| hardwareFault (osCode : UInt32) (details : String) :
    IO.Error
```

The operation failed due to a hardware problem, such as an I/O error.

This corresponds to the POSIX error EIO.

```
| unsatisfiedConstraints (osCode : UInt32)
    (details : String) : IO.Error
```

A constraint required by an operation was not satisfied (e.g. a directory was not empty).

This corresponds to the POSIX error ENOTEMPTY.

```
| illegalOperation (osCode : UInt32) (details : String) :
    IO.Error
```

An inappropriate I/O control operation was attempted.

This corresponds to the POSIX error ENOTTY.

```
| protocolError (osCode : UInt32) (details : String) :
    IO.Error
```

A protocol error occurred.

This corresponds to the POSIX errors EPROTO, EPROTONOSUPPORT, and EPROTOTYPE.

```
| timeExpired (osCode : UInt32) (details : String) : IO.Error
```

An operation timed out.

This corresponds to the POSIX errors ETIME, and ETIMEDOUT.

```
| interrupted (filename : String) (osCode : UInt32)
    (details : String) : IO.Error
```

The operation was interrupted.

This corresponds to the POSIX error EINTR.

```
|
```

```
noFileOrDirectory (filename : String) (osCode : UInt32)
  (details : String) : IO.Error
```

No such file or directory.

This corresponds to the POSIX error ENOENT.

```
| invalidArgument (filename : Option String) (osCode : UInt32)
    (details : String) : IO.Error
```

An argument to an I/O operation was invalid.

This corresponds to the POSIX errors ELOOP, ENAMETOOLONG, EDESTADDRREQ, EILSEQ, EINVAL, EDOM, EBADF ENOEXEC, ENOSTR, ENOTCONN, and ENOTSOCK.

```
| permissionDenied (filename : Option String)
    (osCode : UInt32) (details : String) : IO.Error
```

An operation failed due to insufficient permissions.

This corresponds to the POSIX errors EACCES, EROFS, ECONNABORTED, EFBIG, and EPERM.

```
| resourceExhausted (filename : Option String)
    (osCode : UInt32) (details : String) : IO.Error
```

A resource was exhausted.

This corresponds to the POSIX errors EMFILE, ENFILE, ENOSPC, E2BIG, EAGAIN, EMLINK, EMSGSIZE, ENOBUFS, ENOLCK, ENOMEM, and ENOSR.

```
| inappropriateType (filename : Option String)
    (osCode : UInt32) (details : String) : IO.Error
```

An argument was the wrong type (e.g. a directory when a file was required).

This corresponds to the POSIX errors EISDIR, EBADMSG, and ENOTDIR.

```
| noSuchThing (filename : Option String) (osCode : UInt32)
    (details : String) : IO.Error
```

A required resource does not exist.

This corresponds to the POSIX errors ENXIO, EHOSTUNREACH, ENETUNREACH, ECHILD, ECONNREFUSED, ENODATA, ENOMSG, and ESRCH.

```
| unexpectedEof : IO.Error
```

An unexpected end-of-file marker was encountered.

```
| userError (msg : String) : IO.Error
```

Some other error occurred.

def

```
IO.Error.toString : IO.Error → String
```

Converts an `IO.Error` to a descriptive string.

`IO.Error.userError` is converted to its embedded message. The other constructors are converted in a way that preserves structured information, such as error codes and filenames, that can help diagnose the issue.

def

```
IO.ofExcept.{u_1} {ε : Type u_1} {α : Type} [ToString ε]
  (e : Except ε α) : IO α
```

Converts an `Except ε` action into an `IO` action.

If the `Except ε` action throws an exception, then the exception type's `ToString` instance is used to convert it into an `IO.Error`, which is thrown. Otherwise, the value is returned.

def

```
EIO.catchExceptions {ε α : Type} (act : EIO ε α) (h : ε → BaseIO α)
:
  BaseIO α
```

Handles any exception that might be thrown by an `EIO ε` action, transforming it into an exception-free `BaseIO` action.

def

```
IO.userError (s : String) : IO.Error
```

Constructs an `IO.Error` from a string.

`IO.Error` is the type of exceptions thrown by the `IO` monad.

▼ *Example: Throwing and Catching Errors*

This program repeatedly demands a password, using exceptions for control flow. The syntax used for exceptions is available in all exception monads, not just `IO`. When an incorrect password is provided, an exception is thrown, which is caught by the loop that repeats the

password check. A correct password allows control to proceed past the check, terminating the loop, and any other exceptions are re-thrown.

```
def accessControl : IO Unit := do
  IO.println "What is the password?"
  let password ← (← IO.getStdin).getLine
  if password.trim != "secret" then
    throw (.userError "Incorrect password")
  else return

def repeatAccessControl : IO Unit := do
  repeat
    try
      accessControl
      break
    catch
      | .userError "Incorrect password" =>
        continue
      | other =>
        throw other

def main : IO Unit := do
  repeatAccessControl
  IO.println "Access granted!"
```

When run with this input:

```
stdin
  1 publicinfo
  2 secondtry
  3 secret
```

the program emits:

```
stdout
  1 What is the password?
  2 What is the password?
  3 What is the password?
  4 Access granted!
```

15.2. Control Structures

Normally, programs written in `IO` use the same control structures as those written in other monads. There is one specific `IO` helper.

opaque

```
IO.iterate {α β : Type} (a : α) (f : α → IO (α ⊕ β)) : IO β
```

Iterates an `IO` action. Starting with an initial state, the action is applied repeatedly until it returns a final value in `Sum.inr`. Each time it returns `Sum.inl`, the returned value is treated as a new state.

15.3. Console Output

Lean includes convenience functions for writing to standard output and standard error. All make use of `ToString` instances, and the varieties whose names end in `-ln` add a newline after the output. These convenience functions only expose a part of the functionality available using the standard I/O streams. In particular, to read a line from standard input, use a combination of `IO.getStdin` and `IO.FS.Stream.getLine`.

def

```
IO.print.{u_1} {α : Type u_1} [ToString α] (s : α) : IO Unit
```

Converts `s` to a string using its `ToString α` instance, and prints it to the current standard output (as determined by `IO.getStdout`).

def

```
IO.println.{u_1} {α : Type u_1} [ToString α] (s : α) : IO Unit
```

Converts `s` to a string using its `ToString α` instance, and prints it with a trailing newline to the current standard output (as determined by `IO.getStdout`).

def

```
IO.eprint.{u_1} {α : Type u_1} [ToString α] (s : α) : IO Unit
```

Converts `s` to a string using its `ToString α` instance, and prints it to the current standard error (as determined by `IO.getStderr`).

def

```
IO.eprintln.{u_1} {α : Type u_1} [ToString α] (s : α) : IO Unit
```

Converts `s` to a string using its `ToString α` instance, and prints it with a trailing newline to the current standard error (as determined by `IO.getStderr`).

▼ *Example: Printing*

This program demonstrates all four convenience functions for console I/O.

```
def main : IO Unit := do
  IO.print "This is the "
  IO.print "Lean"
  IO.println " language reference."
  IO.println "Thank you for reading it!"
  IO.eprint "Please report any "
  IO.eprint "errors"
  IO.eprintln " so they can be corrected."
```

It outputs the following to the standard output:

```
stdout
  1 This is the Lean language reference.
  2 Thank you for reading it!
```

and the following to the standard error:

```
stderr
  1 Please report any errors so they can be corrected.
```

15.4. Mutable References

While ordinary state monads encode stateful computations with tuples that track the contents of the state along with the computation's value, Lean's runtime system also provides mutable references that are always backed by mutable memory cells. Mutable references have a type `IO.Ref` that indicates that a cell is mutable, and reads and writes must be explicit. `IO.Ref` is implemented using `ST.Ref`, so the entire `ST.Ref` API may also be used with `IO.Ref`.

def

```
IO.Ref (α : Type) : Type
```

Mutable reference cells that contain values of type *α*. These cells can read from and mutated in the `IO` monad.

def

```
IO.mkRef {α : Type} (a : α) : BaseIO (IO.Ref α)
```

Creates a new mutable reference cell that contains *a*.

15.4.1. State Transformers

Mutable references are often useful in contexts where arbitrary side effects are undesired. They can give a significant speedup when Lean is unable to optimize pure operations into mutation, and some algorithms are more easily expressed using mutable references than with state monads. Additionally, it has a property that other side effects do not have: if all of the mutable references used by a piece of code are created during its execution, and no mutable references from the code escape to other code, then the result of evaluation is deterministic.

The `ST` monad is a restricted version of `IO` in which mutable state is the only side effect, and mutable references cannot escape. ⓘ `ST` was first described by John Launchbury and Simon L Peyton Jones, 1994. “Lazy functional state threads”. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation.*. `ST` takes a type parameter that is never used to classify any terms. The `runST` function, which allow escape from `ST`, requires that the `ST` action that is passed to it can instantiate this type parameter with *any* type. This unknown type does not exist except as a parameter to a function, which means that values whose types are “marked” by it cannot escape its scope.

def

```
ST (σ : Type) : Type → Type
```

A restricted version of `IO` in which mutable state is the only side effect.

It is possible to run `ST` computations in a non-monadic context using `runST`.

def

```
runST {α : Type} (x : (σ : Type) → ST σ α) : α
```

Runs an `ST` computation, in which mutable state via `ST.Ref` is the only side effect.

As with `IO` and `EIO`, there is also a variation of `ST` that takes a custom error type as a parameter. Here, `ST` is analogous to `BaseIO` rather than `IO`, because `ST` cannot result in errors being thrown.

def

```
EST (ε σ : Type) : Type → Type
```

A restricted version of `IO` in which mutable state and exceptions are the only side effects.

It is possible to run `EST` computations in a non-monadic context using `runEST`.

def

```
runEST {ε α : Type} (x : (σ : Type) → EST ε σ α) : Except ε α
```

Runs an EST computation, in which mutable state and exceptions are the only side effects.

structure

```
ST.Ref (σ α : Type) : Type
```

Mutable reference cells that contain values of type *α*. These cells can read from and mutated in the ST *σ* monad.

Constructor

ST.Ref.mk

def

```
ST.mkRef {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m] {α :
Type}
  (a : α) : m (ST.Ref σ α)
```

Creates a new mutable reference that contains the provided value *a*.

15.4.1.1. Reading and Writing

def

```
ST.Ref.get {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m] {α :
Type}
  (r : ST.Ref σ α) : m α
```

Reads the value of a mutable reference.

def

```
ST.Ref.set {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m] {α :
Type}
  (r : ST.Ref σ α) (a : α) : m Unit
```

Replaces the value of a mutable reference.

▼ *Example: Data races with* `get` *and* `set`

```
def main : IO Unit := do
  let balance ← IO.mkRef (100 : Int)

  let mut orders := #[]
```

```
  IO.println "Sending out orders..."
  for _ in [0:100] do
    let o ← IO.asTask (prio := .dedicated) do
      let cost ← IO.rand 1 100
      IO.sleep (← IO.rand 10 100).toUInt32
      if cost < (← balance.get) then
        IO.sleep (← IO.rand 10 100).toUInt32
        balance.set ((← balance.get) - cost)
    orders := orders.push o

  -- Wait until all orders are completed
  for o in orders do
    match o.get with
    | .ok () => pure ()
    | .error e => throw e

  if (← balance.get) < 0 then
    IO.eprintln "Final balance is negative!"
  else
    IO.println "Final balance is zero or positive."
```

```
stdout
  1 Sending out orders...
```

```
stderr
  1 Final balance is negative!
```

def

```
ST.Ref.modify {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m]
  {α : Type} (r : ST.Ref σ α) (f : α → α) : m Unit
```

Atomically modifies a mutable reference cell by replacing its contents with the result of a function call.

▼ *Example: Avoiding data races with* `modify`

This program launches 100 threads. Each thread simulates a purchase attempt: it generates a random price, and if the account balance is sufficient, it decrements it by the price. The balance check and the computation of the new value occur in an atomic call to `ST.Ref.modify`.

```
def main : IO Unit := do
  let balance ← IO.mkRef (100 : Int)

  let mut orders := #[]
  IO.println "Sending out orders..."
  for _ in [0:100] do
```

```
    let o ← IO.asTask (prio := .dedicated) do
      let cost ← IO.rand 1 100
      IO.sleep (← IO.rand 10 100).toUInt32
      balance.modify fun b =>
        if cost < b then
          b - cost
        else b
      orders := orders.push o

  -- Wait until all orders are completed
  for o in orders do
    match o.get with
    | .ok () => pure ()
    | .error e => throw e

  if (← balance.get) < 0 then
    IO.eprintln "Final balance negative!"
  else
    IO.println "Final balance is zero or positive."
```

```
stdout
  1 Sending out orders...
  2 Final balance is zero or positive.
```

```
stderr
    <empty>
```

def

```
ST.Ref.modifyGet {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m]
  {α β : Type} (r : ST.Ref σ α) (f : α → β × α) : m β
```

Atomically modifies a mutable reference cell by replacing its contents with the result of a function call that simultaneously computes a value to return.

def

```
ST.Ref.swap {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m]
  {α : Type} (r : ST.Ref σ α) (a : α) : m α
```

Atomically swaps the value of a mutable reference cell with another value. The reference cell's original value is returned.

15.4.1.2. Comparisons

def

```
ST.Ref.ptrEq {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m]
  {α : Type} (r1 r2 : ST.Ref σ α) : m Bool
```

Checks whether two reference cells are in fact aliases for the same cell.

Even if they contain the same value, two references allocated by different executions of `IO.mkRef` or `ST.mkRef` are distinct. Modifying one has no effect on the other. Likewise, a single reference cell may be aliased, and modifications to one alias also modify the other.

15.4.1.3. ST-Backed State Monads

def

```
ST.Ref.toMonadStateOf {σ : Type} {m : Type → Type} [MonadLiftT (ST
σ) m]
  {α : Type} (r : ST.Ref σ α) : MonadStateOf α m
```

Creates a `MonadStateOf` instance from a reference cell.

This allows programs written against the state monad API to be executed using a mutable reference cell to track the state.

15.4.2. Concurrency

Mutable references can be used as a locking mechanism. *Taking* the contents of the reference causes attempts to take it or to read from it to block until it is `set` again. This is a low-level feature that can be used to implement other synchronization mechanisms; it's usually better to rely on higher-level abstractions when possible.

unsafe def

```
ST.Ref.take {σ : Type} {m : Type → Type} [MonadLiftT (ST σ) m]
  {α : Type} (r : ST.Ref σ α) : m α
```

Reads the value of a mutable reference cell, removing it.

This causes subsequent attempts to read from or take the reference cell to block until a new value is written using `ST.Ref.set`.

▼ *Example: Reference Cells as Locks*

This program launches 100 threads. Each thread simulates a purchase attempt: it generates a random price, and if the account balance is sufficient, it decrements it by the price. If the balance is not sufficient, then it is not decremented. Because each thread takes the balance cell prior to checking it and only returns it when it is finished, the cell acts as a lock. Unlike using ST.Ref.modify, which atomically modifies the contents of the cell using a pure function, other IO actions may occur in the critical section This program's `main` function is marked `unsafe` because take itself is unsafe.

```
unsafe def main : IO Unit := do
  let balance ← IO.mkRef (100 : Int)
  let validationUsed ← IO.mkRef false

  let mut orders := #[]

  IO.println "Sending out orders..."
  for _ in [0:100] do
    let o ← IO.asTask (prio := .dedicated) do
      let cost ← IO.rand 1 100
      IO.sleep (← IO.rand 10 100).toUInt32
      let b ← balance.take
      if cost ≤ b then
        balance.set (b - cost)
      else
        balance.set b
        validationUsed.set true
    orders := orders.push o

  -- Wait until all orders are completed
  for o in orders do
    match o.get with
    | .ok () => pure ()
    | .error e => throw e

  if (← validationUsed.get) then
    IO.println "Validation prevented a negative balance."

  if (← balance.get) < 0 then
    IO.eprintln "Final balance negative!"
  else
    IO.println "Final balance is zero or positive."
```

The program's output is:

```
stdout
  1 Sending out orders...
  2 Validation prevented a negative balance.
```

```
3 Final balance is zero or positive.
```

15.5. Files, File Handles, and Streams

Lean provides a consistent filesystem API on all supported platforms. These are the key concepts:

Files

Files are an abstraction provided by operating systems that provide random access to persistently-stored data, organized hierarchically into directories.

Directories

Directories, also known as *folders*, may contain files or other directories. Fundamentally, a directory maps names to the files and/or directories that it contains.

File Handles

File handles (`Handle`) are abstract references to files that have been opened for reading and/or writing. A file handle maintains a mode that determines whether reading and/or writing are allowed, along with a cursor that points at a specific location in the file. Reading from or writing to a file handle advances the cursor. File handles may be buffered, which means that reading from a file handle may not return the current contents of the persistent data, and writing to a file handle may not modify them immediately.

Paths

Files are primarily accessed via *paths* (`System.FilePath`). A path is a sequence of directory names, potentially terminated by a file name. They are represented by strings in which separator characters (i The current platform's separator characters are listed in `System.FilePath.pathSeparators`.) delimit the names.

The details of paths are platform-specific. Absolute paths begin in a *root directory*; some operating systems have a single root, while others may have multiple root directories. Relative paths do not begin in a root directory and require that some other directory be taken as a starting point. In addition to directories, paths may contain the special directory names `.`, which refers to the directory in which it is found, and `..`, which refers to prior directory in the path.

Filenames, and thus paths, may end in one or more *extensions* that identify the file's type. Extensions are delimited by the character `System.FilePath.extSeparator`. On some platforms, executable files have a special extension (`System.FilePath.exeExtension`).

Streams

Streams are a higher-level abstraction over files, both providing additional functionality and hiding some details of files. While file handles are essentially a thin wrapper around the operating system's representation, streams are implemented in Lean as a structure called `IO.FS.Stream`. Because streams are implemented in Lean, user code can create additional streams, which can be used seamlessly together with those provided in the standard library.

15.5.1. Low-Level File API

At the lowest level, files are explicitly opened using `Handle.mk`. When the last reference to the handle object is dropped, the file is closed. There is no explicit way to close a file handle other than by ensuring that there are no references to it.

opaque

```
IO.FS.Handle : Type
```

A reference to an opened file.

File handles wrap the underlying operating system's file descriptors. There is no explicit operation to close a file: when the last reference to a file handle is dropped, the file is closed automatically.

Handles have an associated read/write cursor that determines the where reads and writes occur in the file.

opaque

```
IO.FS.Handle.mk (fn : System.FilePath) (mode : IO.FS.Mode) :
  IO IO.FS.Handle
```

Opens the file at *fn* with the given *mode*.

An exception is thrown if the file cannot be opened.

inductive type

```
IO.FS.Mode : Type
```

Whether a file should be opened for reading, writing, creation and writing, or appending.

At the operating system level, this translates to the mode of a file handle (i.e., a set of **open** flags and an `fdopen` mode).

None of the modes represented by this datatype translate line endings (i.e. `O_BINARY` on Windows). Furthermore, they are not inherited across process creation (i.e. `O_NOINHERIT` on Windows and `O_CLOEXEC` elsewhere).

Operating System Specifics:

- Windows: `_open`, `_fdopen`
- Linux: **`open`**, `fdopen`

Constructors

```
| read : IO.FS.Mode
```

The file should be opened for reading.

The read/write cursor is positioned at the beginning of the file. It is an error if the file does not exist.

- **open** flags: `O_RDONLY`
- `fdopen` mode: `r`

`write : IO.FS.Mode`

The file should be opened for writing.

If the file already exists, it is truncated to zero length. Otherwise, a new file is created. The read/write cursor is positioned at the beginning of the file.

- **open** flags: `O_WRONLY | O_CREAT | O_TRUNC`
- `fdopen` mode: `w`

`writeNew : IO.FS.Mode`

A new file should be created for writing.

It is an error if the file already exists. A new file is created, with the read/write cursor positioned at the start.

- **open** flags: `O_WRONLY | O_CREAT | O_TRUNC | O_EXCL`
- `fdopen` mode: `w`

`readWrite : IO.FS.Mode`

The file should be opened for both reading and writing.

It is an error if the file does not already exist. The read/write cursor is positioned at the start of the file.

- **open** flags: `O_RDWR`
- `fdopen` mode: `r+`

`append : IO.FS.Mode`

The file should be opened for writing.

If the file does not already exist, it is created. If the file already exists, it is opened, and the read/write cursor is positioned at the end of the file.

- **open** flags: `O_WRONLY | O_CREAT | O_APPEND`
- `fdopen` mode: `a`

opaque

```
IO.FS.Handle.read (h : IO.FS.Handle) (bytes : USize) : IO ByteArray
```

Reads up to the given number of bytes from the handle. If the returned array is empty, an end-of-file marker (EOF) has been reached.

Encountering an EOF does not close a handle. Subsequent reads may block and return more data.

def

```
IO.FS.Handle.readToEnd (h : IO.FS.Handle) : IO String
```

Reads the entire remaining contents of the file handle as a UTF-8-encoded string. An exception is thrown if the contents are not valid UTF-8.

The underlying file is not automatically closed, and subsequent reads from the handle may block and/or return data.

def

```
IO.FS.Handle.readBinToEnd (h : IO.FS.Handle) : IO ByteArray
```

Reads the entire remaining contents of the file handle until an end-of-file marker (EOF) is encountered.

The underlying file is not automatically closed upon encountering an EOF, and subsequent reads from the handle may block and/or return data.

def

```
IO.FS.Handle.readBinToEndInto (h : IO.FS.Handle) (buf : ByteArray) :
  IO ByteArray
```

Reads the entire remaining contents of the file handle until an end-of-file marker (EOF) is encountered.

The underlying file is not automatically closed upon encountering an EOF, and subsequent reads from the handle may block and/or return data.

opaque

```
IO.FS.Handle.getLine (h : IO.FS.Handle) : IO String
```

Reads UTF-8-encoded text up to and including the next line break from the handle. If the returned string is empty, an end-of-file marker (EOF) has been reached.

Encountering an EOF does not close a handle. Subsequent reads may block and return more data.

opaque

```
IO.FS.Handle.write (h : IO.FS.Handle) (buffer : ByteArray) : IO Unit
```

Writes the provided bytes to the the handle.

Writing to a handle is typically buffered, and may not immediately modify the file on disk. Use `IO.FS.Handle.flush` to write changes to buffers to the associated device.

opaque

```
IO.FS.Handle.putStr (h : IO.FS.Handle) (s : String) : IO Unit
```

Writes the provided string to the file handle using the UTF-8 encoding.

Writing to a handle is typically buffered, and may not immediately modify the file on disk. Use `IO.FS.Handle.flush` to write changes to buffers to the associated device.

def

```
IO.FS.Handle.putStrLn (h : IO.FS.Handle) (s : String) : IO Unit
```

Writes the contents of the string to the handle, followed by a newline. Uses UTF-8.

opaque

```
IO.FS.Handle.flush (h : IO.FS.Handle) : IO Unit
```

Flushes the output buffer associated with the handle, writing any unwritten data to the associated output device.

opaque

```
IO.FS.Handle.rewind (h : IO.FS.Handle) : IO Unit
```

Rewinds the read/write cursor to the beginning of the handle's file.

opaque

```
IO.FS.Handle.truncate (h : IO.FS.Handle) : IO Unit
```

Truncates the handle to its read/write cursor.

This operation does not automatically flush output buffers, so the contents of the output device may not reflect the change immediately. This does not usually lead to problems because the read/write cursor includes buffered writes. However, buffered writes followed by `IO.FS.Handle.rewind`, then `IO.FS.Handle.truncate`, and then closing the file may lead to a non-empty file. If unsure, call `IO.FS.Handle.flush` before truncating.

opaque

```
IO.FS.Handle.isTty (h : IO.FS.Handle) : BaseIO Bool
```

Returns `true` if a handle refers to a Windows console or a Unix terminal.

opaque

```
IO.FS.Handle.lock (h : IO.FS.Handle) (exclusive : Bool := true) :
  IO Unit
```

Acquires an exclusive or shared lock on the handle. Blocks to wait for the lock if necessary.

Acquiring a exclusive lock while already possessing a shared lock will **not** reliably succeed: it works on Unix-like systems but not on Windows.

opaque

```
IO.FS.Handle.tryLock (h : IO.FS.Handle) (exclusive : Bool := true) :
  IO Bool
```

Tries to acquire an exclusive or shared lock on the handle and returns `true` if successful. Will not block if the lock cannot be acquired, but instead returns `false`.

Acquiring a exclusive lock while already possessing a shared lock will **not** reliably succeed: it works on Unix-like systems but not on Windows.

opaque

```
IO.FS.Handle.unlock (h : IO.FS.Handle) : IO Unit
```

Releases any previously-acquired lock on the handle. Succeeds even if no lock has been acquired.

▼ *Example: One File, Multiple Handles*

This program has two handles to the same file. Because file I/O may be buffered independently for each handle, `Handle.flush` should be called when the buffers need to be synchronized with the file's actual contents. Here, the two handles proceed in lock-step through the file, with one of them a single byte ahead of the other. The first handle is used to count the number of

occurrences of `'A'`, while the second is used to replace each `'A'` with `'!'`. The second handle is opened in `readWrite` mode rather than `write` mode because opening an existing file in `write` mode replaces it with an empty file. In this case, the buffers don't need to be flushed during execution because modifications occur only to parts of the file that will not be read again, but the write handle should be flushed after the loop has completed.

```
open IO.FS (Handle)

def main : IO Unit := do
  IO.println s!"Starting contents: '{(← IO.FS.readFile
"data").trim}'"

  let h ← Handle.mk "data" .read
  let h' ← Handle.mk "data" .readWrite
  h'.rewind

  let mut count := 0
  let mut buf : ByteArray ← h.read 1
  while ok : buf.size = 1 do
    if Char.ofUInt8 buf[0] == 'A' then
      count := count + 1
      h'.write (ByteArray.empty.push '!'.toUInt8)
    else
      h'.write buf
    buf ← h.read 1

  h'.flush

  IO.println s!"Count: {count}"
  IO.println s!"Contents: '{(← IO.FS.readFile "data").trim}'"
```

When run on this file:

Input: `data`
```
AABAABCDAB
```

the program outputs:

`stdout`
```
Starting contents: 'AABAABCDAB'
Count: 5
Contents: '!!B!!BCD!B'
```

Afterwards, the file contains:

Output: `data`
```
!!B!!BCD!B
```

15.5.2. Streams

structure

`IO.FS.Stream : Type`

A pure-Lean abstraction of POSIX streams. These streams may represent an underlying POSIX stream or be implemented by Lean code.

Because standard input, standard output, and standard error are all `IO.FS.Stream`s that can be overridden, Lean code may capture and redirect input and output.

Constructor

`IO.FS.Stream.mk`

Fields

`flush : IO Unit`

Flushes the stream's output buffers.

`read : USize → IO ByteArray`

Reads up to the given number of bytes from the stream.

If the returned array is empty, an end-of-file marker (EOF) has been reached. An EOF does not actually close a stream, so further reads may block and return more data.

`write : ByteArray → IO Unit`

Writes the provided bytes to the stream.

If the stream represents a physical output device such as a file on disk, then the results may be buffered. Call `FS.Stream.flush` to synchronize their contents.

`getLine : IO String`

Reads text up to and including the next newline from the stream.

If the returned string is empty, an end-of-file marker (EOF) has been reached. An EOF does not actually close a stream, so further reads may block and return more data.

`putStr : String → IO Unit`

Writes the provided string to the stream.

`isTty : BaseIO Bool`

Returns `true` if a stream refers to a Windows console or Unix terminal.

def

```
IO.FS.Stream.ofBuffer (r : IO.Ref IO.FS.Stream.Buffer) :
IO.FS.Stream
```

Creates a stream from a mutable reference to a buffer.

The resulting stream simulates a file, mutating the contents of the reference in response to writes and reading from it in response to reads. These streams can be used with `IO.withStdin`, `IO.setStdin`, and the corresponding operators for standard output and standard error to redirect input and output.

def

```
IO.FS.Stream.ofHandle (h : IO.FS.Handle) : IO.FS.Stream
```

Creates a Lean stream from a file handle. Each stream operation is implemented by the corresponding file handle operation.

def

```
IO.FS.Stream.putStrLn (strm : IO.FS.Stream) (s : String) : IO Unit
```

Writes the contents of the string to the stream, followed by a newline.

structure

```
IO.FS.Stream.Buffer : Type
```

A byte buffer that can simulate a file in memory.

Use `IO.FS.Stream.ofBuffer` to create a stream from a buffer.

Constructor

`IO.FS.Stream.Buffer.mk`

Fields

`data : ByteArray`

The contents of the buffer.

`pos : Nat`

The read/write cursor's position in the buffer.

15.5.3. Paths

Paths are represented by strings. Different platforms have different conventions for paths: some use slashes (`/`) as directory separators, others use backslashes (`\`). Some are case-sensitive, others are not. Different Unicode encodings and normal forms may be used to represent filenames, and some platforms consider filenames to be byte sequences rather than strings. A string that represents an absolute path on one system may not even be a valid path on another system.

To write Lean code that is as compatible as possible with multiple systems, it can be helpful to use Lean's path manipulation primitives instead of raw string manipulation. Helpers such as `System.FilePath.join` take platform-specific rules for absolute paths into account, `System.FilePath.pathSeparator` contains the appropriate path separator for the current platform, and `System.FilePath.exeExtension` contains any necessary extension for executable files. Avoid hard-coding these rules.

There is an instance of the `Div` type class for `FilePath` which allows the slash operator to be used to concatenate paths.

structure

```
System.FilePath : Type
```

A path on the file system.

Paths consist of a sequence of directories followed by the name of a file or directory. They are delimited by a platform-dependent separator character (see `System.FilePath.pathSeparator`).

Constructor

`System.FilePath.mk`

Fields

`toString : String`

The string representation of the path.

def

```
System.mkFilePath (parts : List String) : System.FilePath
```

Constructs a path from a list of file names by interspersing them with the current platform's path separator.

def

```
System.FilePath.join (p sub : System.FilePath) : System.FilePath
```

Appends two paths, taking absolute paths into account. This operation is also accessible via the `/` operator.

If *sub* is an absolute path, then *p* is discarded and *sub* is returned. If *sub* is a relative path, then it is attached to *p* with the platform-specific path separator.

def

```
System.FilePath.normalize (p : System.FilePath) : System.FilePath
```

Normalizes a path, returning an equivalent path that may better follow platform conventions.

In particular:

- On Windows, drive letters are made uppercase.
- On platforms that support multiple path separators (that is, where `System.FilePath.pathSeparators` has length greater than one), alternative path separators are replaced with the preferred path separator.

There is no guarantee that two equivalent paths normalize to the same path.

def

```
System.FilePath.isAbsolute (p : System.FilePath) : Bool
```

An absolute path starts at the root directory or a drive letter. Accessing files through an absolute path does not depend on the current working directory.

def

```
System.FilePath.isRelative (p : System.FilePath) : Bool
```

A relative path is one that depends on the current working directory for interpretation. Relative paths do not start with the root directory or a drive letter.

def

```
System.FilePath.parent (p : System.FilePath) : Option System.FilePath
```

Returns the parent directory of a path, if there is one.

If the path is that of the root directory or the root of a drive letter, `none` is returned. Otherwise, the path's parent directory is returned.

def

```
System.FilePath.components (p : System.FilePath) : List String
```

Splits a path into a list of individual file names at the platform-specific path separator.

def

```
System.FilePath.fileName (p : System.FilePath) : Option String
```

Extracts the last element of a path if it is a file or directory name.

Returns `none` if the last entry is a special name (such as `.` or `..`) or if the path is the root directory.

def

```
System.FilePath.fileStem (p : System.FilePath) : Option String
```

Extracts the stem (non-extension) part of *p*`.fileName`.

If the filename contains multiple extensions, then only the last one is removed. Returns `none` if there is no file name at the end of the path.

Examples:

- `("app.exe" : System.FilePath).fileStem = some "app"`
- `("file.tar.gz" : System.FilePath).fileStem = some "file.tar"`
- `("files/" : System.FilePath).fileStem = none`
- `("files/picture.jpg" : System.FilePath).fileStem = some "picture"`

def

```
System.FilePath.extension (p : System.FilePath) : Option String
```

Extracts the extension part of *p*`.fileName`.

If the filename contains multiple extensions, then only the last one is extracted. Returns `none` if there is no file name at the end of the path.

Examples:

- `("app.exe" : System.FilePath).extension = some "exe"`
- `("file.tar.gz" : System.FilePath).extension = some "gz"`
- `("files/" : System.FilePath).extension = none`

- `("files/picture.jpg" : System.FilePath).extension = some "jpg"`

def

```
System.FilePath.addExtension (p : System.FilePath) (ext : String) :
  System.FilePath
```

Appends the extension *ext* to a path *p*.

ext should not have leading `.`, as this function adds one. If *ext* is the empty string, no `.` is added.

Unlike System.FilePath.withExtension, this does not remove any existing extension.

def

```
System.FilePath.withExtension (p : System.FilePath) (ext : String) :
  System.FilePath
```

Replaces the current extension in a path *p* with *ext*, adding it if there is no extension. If the path has multiple file extensions, only the last one is replaced. If the path has no filename, or if *ext* is the empty string, then the filename is returned unmodified.

ext should not have a leading `.`, as this function adds one.

Examples:

- `("files/picture.jpeg" : System.FilePath).withExtension "jpg" = ⟨"files/picture.jpg"⟩`
- `("files/" : System.FilePath).withExtension "zip" = ⟨"files/"⟩`
- `("files" : System.FilePath).withExtension "zip" = ⟨"files.zip"⟩`
- `("files/archive.tar.gz" : System.FilePath).withExtension "xz" = ⟨"files.tar.xz"⟩`

def

```
System.FilePath.withFileName (p : System.FilePath) (fname : String)
:
  System.FilePath
```

Replaces the file name at the end of the path *p* with *fname*, placing *fname* in the parent directory of *p*.

If *p* has no parent directory, then *fname* is returned unmodified.

def

```
System.FilePath.pathSeparator : Char
```

The character that separates directories.

On platforms that support multiple separators, `System.FilePath.pathSeparator` is the "ideal" one expected by users on the platform. `System.FilePath.pathSeparators` lists all supported separators.

def

```
System.FilePath.pathSeparators : List Char
```

The list of all path separator characters supported on the current platform.

On platforms that support multiple separators, `System.FilePath.pathSeparator` is the "ideal" one expected by users on the platform.

def

```
System.FilePath.extSeparator : Char
```

The character that separates file extensions from file names.

def

```
System.FilePath.exeExtension : String
```

The file extension expected for executable binaries on the current platform, or `""` if there is no such extension.

15.5.4. Interacting with the Filesystem

Some operations on paths consult the filesystem.

structure

```
IO.FS.Metadata : Type
```

File metadata.

The metadata for a file can be accessed with `System.FilePath.metadata`/ `System.FilePath.symlinkMetadata`.

Constructor

`IO.FS.Metadata.mk`

Fields

`accessed : IO.FS.SystemTime`

File access time.

`modified : IO.FS.SystemTime`

File modification time.

`byteSize : UInt64`

The size of the file in bytes.

`type : IO.FS.FileType`

Whether the file is an ordinary file, a directory, a symbolic link, or some other kind of file.

opaque

`System.FilePath.metadata : System.FilePath → IO IO.FS.Metadata`

Returns metadata for the indicated file, following symlinks. Throws an exception if the file does not exist or the metadata cannot be accessed.

def

`System.FilePath.pathExists (p : System.FilePath) : BaseIO Bool`

Checks whether the indicated path points to a file that exists. This function will traverse symlinks.

def

`System.FilePath.isDir (p : System.FilePath) : BaseIO Bool`

Checks whether the indicated path can be read and is a directory. This function will traverse symlinks.

structure

`IO.FS.DirEntry : Type`

An entry in a directory on a filesystem.

Constructor

`IO.FS.DirEntry.mk`

Fields

`root : System.FilePath`

The directory in which the entry is found.

`fileName : String`

The name of the entry.

def

```
IO.FS.DirEntry.path (entry : IO.FS.DirEntry) : System.FilePath
```

The path of the file indicated by the directory entry.

opaque

```
System.FilePath.readDir : System.FilePath → IO (Array
IO.FS.DirEntry)
```

Returns the contents of the indicated directory. Throws an exception if the file does not exist or is not a directory.

def

```
System.FilePath.walkDir (p : System.FilePath)
  (enter : System.FilePath → IO Bool := fun x => pure true) :
  IO (Array System.FilePath)
```

Traverses a filesystem starting at the path *p* and exploring directories that satisfy `enter`, returning the paths visited.

The traversal is a preorder traversal, in which parent directories occur prior to any of their children. Symbolic links are followed.

structure

```
IO.AccessRight : Type
```

POSIX-style file permissions.

The `FileRight` structure describes these permissions for a file's owner, members of it's designated group, and all others.

Constructor

`IO.AccessRight.mk`

Fields

`read : Bool`

The file can be read.

`write : Bool`

The file can be written to.

`execution : Bool`

The file can be executed.

def

`IO.AccessRight.flags (acc : IO.AccessRight) : UInt32`

Converts individual POSIX-style file permissions to their conventional three-bit representation.

This is the bitwise or of the following:

- If the file can be read, `0x4`, otherwise `0`.
- If the file can be written, `0x2`, otherwise `0`.
- If the file can be executed, `0x1`, otherwise `0`.

Examples:

- `{read := true : AccessRight}.flags = 4`
- `{read := true, write := true : AccessRight}.flags = 6`
- `{read := true, execution := true : AccessRight}.flags = 5`

structure

`IO.FileRight : Type`

POSIX-style file permissions that describe access rights for a file's owner, members of its assigned group, and all others.

Constructor

`IO.FileRight.mk`

Fields

`user : IO.AccessRight`

The owner's permissions to access the file.

`group : IO.AccessRight`

The assigned group's permissions to access the file.

`other : IO.AccessRight`

The permissions that all others have to access the file.

def

```
IO.FileRight.flags (acc : IO.FileRight) : UInt32
```

Converts POSIX-style file permissions to their numeric representation, with three bits each for the owner's permissions, the group's permissions, and others' permissions.

def

```
IO.setAccessRights (filename : System.FilePath) (mode :
IO.FileRight) :
  IO Unit
```

Sets the POSIX-style permissions for a file.

opaque

```
IO.FS.removeFile (fname : System.FilePath) : IO Unit
```

Removes (deletes) a file from the filesystem.

To remove a directory, use `IO.FS.removeDir` or `IO.FS.removeDirAll` instead.

opaque

```
IO.FS.rename (old new : System.FilePath) : IO Unit
```

Moves a file or directory `old` to the new location *new*.

This function coincides with the POSIX `rename` function.

opaque

```
IO.FS.removeDir : System.FilePath → IO Unit
```

Removes (deletes) a directory.

Removing a directory fails if the directory is not empty. Use `IO.FS.removeDirAll` to remove directories along with their contents.

def

```
IO.FS.lines (fname : System.FilePath) : IO (Array String)
```

Returns the contents of a UTF-8-encoded text file as an array of lines.

Newline markers are not included in the lines.

def

```
IO.FS.withTempFile.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  [MonadFinally m] [MonadLiftT IO m]
  (f : IO.FS.Handle → System.FilePath → m α) : m α
```

Creates a temporary file in the most secure manner possible and calls f with both a `Handle` to the already-opened file and its path. Afterwards, the temporary file is deleted.

There are no race conditions in the file's creation. The file is readable and writable only by the creating user ID. Additionally on UNIX style platforms the file is executable by nobody.

Use `IO.FS.createTempFile` to avoid the automatic deletion of the temporary file.

def

```
IO.FS.withTempDir.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  [MonadFinally m] [MonadLiftT IO m] (f : System.FilePath → m α) :
m α
```

Creates a temporary directory in the most secure manner possible, providing a its path to an `IO` action. Afterwards, all files in the temporary directory are recursively deleted, regardless of how or when they were created.

There are no race conditions in the directory's creation. The directory is readable and writable only by the creating user ID. Use `IO.FS.createTempDir` to avoid the automatic deletion of the directory's contents.

opaque

```
IO.FS.createDirAll (p : System.FilePath) : IO Unit
```

Creates a directory at the specified path, creating all missing parents as directories.

def

```
IO.FS.writeBinFile (fname : System.FilePath) (content : ByteArray) :
  IO Unit
```

Write the provided bytes to a binary file at the specified path.

def

```
IO.FS.withFile {α : Type} (fn : System.FilePath) (mode :
IO.FS.Mode)
  (f : IO.FS.Handle → IO α) : IO α
```

Opens the file *fn* with the specified *mode* and passes the resulting file handle to *f*.

The file handle is closed when the last reference to it is dropped. If references escape *f*, then the file remains open even after `IO.FS.withFile` has finished.

opaque

```
IO.FS.removeDirAll (p : System.FilePath) : IO Unit
```

Fully remove given directory by deleting all contained files and directories in an unspecified order. Symlinks are deleted but not followed. Fails if any contained entry cannot be deleted or was newly created during execution.

opaque

```
IO.FS.createTempFile : IO (IO.FS.Handle × System.FilePath)
```

Creates a temporary file in the most secure manner possible, returning both a `Handle` to the already-opened file and its path.

There are no race conditions in the file's creation. The file is readable and writable only by the creating user ID. Additionally on UNIX style platforms the file is executable by nobody.

It is the caller's job to remove the file after use. Use `withTempFile` to ensure that the temporary file is removed.

opaque

```
IO.FS.createTempDir : IO System.FilePath
```

Creates a temporary directory in the most secure manner possible, returning the new directory's path. There are no race conditions in the directory's creation. The directory is readable and writable only by the creating user ID.

It is the caller's job to remove the directory after use. Use `withTempDir` to ensure that the temporary directory is removed.

def

```
IO.FS.readFile (fname : System.FilePath) : IO String
```

Reads the entire contents of the UTF-8-encoded file at the given path as a `String`.

An exception is thrown if the contents of the file are not valid UTF-8. This is in addition to exceptions that may always be thrown as a result of failing to read files.

opaque

```
IO.FS.realPath (fname : System.FilePath) : IO System.FilePath
```

Resolves a path to an absolute path that contains no '.', '..', or symbolic links.

This function coincides with the POSIX `realpath` function.

def

```
IO.FS.writeFile (fname : System.FilePath) (content : String) : IO Unit
```

Write contents of a string to a file at the specified path using UTF-8 encoding.

def

```
IO.FS.readBinFile (fname : System.FilePath) : IO ByteArray
```

Reads the entire contents of the binary file at the given path as an array of bytes.

opaque

```
IO.FS.createDir : System.FilePath → IO Unit
```

Creates a directory at the specified path. The parent directory must already exist.

Throws an exception if the directory cannot be created.

15.5.5. Standard I/O

On operating systems that are derived from or inspired by Unix, *standard input*, *standard output*, and *standard error* are the names of three streams that are available in each process. Generally, programs are expected to read from standard input, write ordinary output to the standard output, and error messages to the standard error. By default, standard input receives input from the console, while standard output and standard error output to the console, but all three are often redirected to or from pipes or files.

Rather than providing direct access to the operating system's standard I/O facilities, Lean wraps them in `Stream`s. Additionally, the `IO` monad contains special support for replacing or locally overriding them. This extra level of indirection makes it possible to redirect input and output within a Lean program.

opaque

```
IO.getStdin : BaseIO IO.FS.Stream
```

Returns the current thread's standard input stream.

Use `IO.setStdin` to replace the current thread's standard input stream.

▼ *Example: Reading from Standard Input*

In this example, `IO.getStdin` and `IO.getStdout` are used to get the current standard input and output, respectively. These can be read from and written to.

```
def main : IO Unit := do
  let stdin ← IO.getStdin
  let stdout ← IO.getStdout
  stdout.putStrLn "Who is it?"
  let name ← stdin.getLine
  stdout.putStr "Hello, "
  stdout.putStrLn name
```

With this standard input:

```
stdin
  1 Lean user
```

the standard output is:

```
stdout
  1 Who is it?
  2 Hello, Lean user
```

opaque

```
IO.setStdin : IO.FS.Stream → BaseIO IO.FS.Stream
```

Replaces the standard input stream of the current thread and returns its previous value.

Use `IO.getStdin` to get the current standard input stream.

def

```
IO.withStdin.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  [MonadFinally m] [MonadLiftT BaseIO m] (h : IO.FS.Stream) (x : m
α) :
  m α
```

Runs an action with the specified stream *h* as standard input, restoring the original standard input stream afterwards.

opaque

```
IO.getStdout : BaseIO IO.FS.Stream
```

Returns the current thread's standard output stream.

Use IO.setStdout to replace the current thread's standard output stream.

opaque

```
IO.setStdout : IO.FS.Stream → BaseIO IO.FS.Stream
```

Replaces the standard output stream of the current thread and returns its previous value.

Use IO.getStdout to get the current standard output stream.

def

```
IO.withStdout.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  [MonadFinally m] [MonadLiftT BaseIO m] (h : IO.FS.Stream) (x : m
α) :
  m α
```

Runs an action with the specified stream *h* as standard output, restoring the original standard output stream afterwards.

opaque

```
IO.getStderr : BaseIO IO.FS.Stream
```

Returns the current thread's standard error stream.

Use IO.setStderr to replace the current thread's standard error stream.

opaque

```
IO.setStderr : IO.FS.Stream → BaseIO IO.FS.Stream
```

Replaces the standard error stream of the current thread and returns its previous value.

Use IO.getStderr to get the current standard error stream.

def

```
IO.withStderr.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  [MonadFinally m] [MonadLiftT BaseIO m] (h : IO.FS.Stream) (x : m
α) :
  m α
```

Runs an action with the specified stream *h* as standard error, restoring the original standard error stream afterwards.

def

```
IO.FS.withIsolatedStreams.{u_1} {m : Type → Type u_1} {α : Type}
  [Monad m] [MonadFinally m] [MonadLiftT BaseIO m] (x : m α)
  (isolateStderr : Bool := true) : m (String × α)
```

Runs an action with `stdin` emptied and `stdout` and `stderr` captured into a `String`. If *`isolateStderr`* is `false`, only `stdout` is captured.

▼ *Example: Redirecting Standard I/O to Strings*

The `countdown` function counts down from a specified number, writing its progress to standard output. Using `IO.FS.withIsolatedStreams`, this output can be redirected to a string.

```
def countdown : Nat → IO Unit
  | 0 =>
    IO.println "Blastoff!"
  | n + 1 => do
    IO.println s!"{n + 1}"
    countdown n

def runCountdown : IO String := do
  let (output, ()) ← IO.FS.withIsolatedStreams (countdown 10)
  return output

#eval runCountdown
```

Running `countdown` yields a string that contains the output:

```
"10\n9\n8\n7\n6\n5\n4\n3\n2\n1\nBlastoff!\n"
```

15.5.6. Files and Directories

opaque

```
IO.currentDir : IO System.FilePath
```

Returns the current working directory of the executing process.

opaque

```
IO.appPath : IO System.FilePath
```

Returns the file name of the currently-running executable.

def

```
IO.appDir : IO System.FilePath
```

Returns the directory that the current executable is located in.

15.6. System and Platform Information

def

```
System.Platform.numBits : Nat
```

The word size of the current platform, which may be 64 or 32 bits.

def

```
System.Platform.target : String
```

The LLVM target triple of the current platform. Empty if missing when Lean was compiled.

def

```
System.Platform.isWindows : Bool
```

Is the current platform Windows?

def

```
System.Platform.isOSX : Bool
```

Is the current platform macOS?

def

```
System.Platform.isEmscripten : Bool
```

Is the current platform Emscripten?

15.7. Environment Variables

opaque

```
IO.getEnv (var : String) : BaseIO (Option String)
```

Returns the value of the environment variable *var*, or none if it is not present in the environment.

15.8. Timing

def

```
IO.sleep (ms : UInt32) : BaseIO Unit
```

Pauses execution for the specified number of milliseconds.

opaque

```
IO.monoNanosNow : BaseIO Nat
```

Monotonically increasing time since an unspecified past point in nanoseconds. There is no relation to wall clock time.

opaque

```
IO.monoMsNow : BaseIO Nat
```

Monotonically increasing time since an unspecified past point in milliseconds. There is no relation to wall clock time.

opaque

```
IO.getNumHeartbeats : BaseIO Nat
```

Returns the number of *heartbeats* that have occurred during the current thread's execution. The heartbeat count is the number of “small” memory allocations performed in a thread.

Heartbeats used to implement timeouts that are more deterministic across different hardware.

def

```
IO.addHeartbeats (count : Nat) : BaseIO Unit
```

Adjusts the heartbeat counter of the current thread by the given amount. This can be useful to give allocation-avoiding code additional “weight” and is also used to adjust the counter after resuming from a snapshot.

Heartbeats are a means of implementing “deterministic” timeouts. The heartbeat counter is the number of “small” memory allocations performed on the current execution thread.

15.9. Processes

15.9.1. Current Process

opaque

```
IO.Process.getCurrentDir : IO System.FilePath
```

Returns the current working directory of the calling process.

opaque

```
IO.Process.setCurrentDir (path : System.FilePath) : IO Unit
```

Sets the current working directory of the calling process.

opaque

```
IO.Process.exit {α : Type} : UInt8 → IO α
```

Terminates the current process with the provided exit code. `0` indicates success, all other values indicate failure.

opaque

```
IO.Process.getPID : BaseIO UInt32
```

Returns the process ID of the calling process.

15.9.2. Running Processes

There are three primary ways to run other programs from Lean:

1. `IO.Process.run` synchronously executes another program, returning its standard output as a string. It throws an error if the process exits with an error code other than `0`.
2. `IO.Process.output` synchronously executes another program with an empty standard input, capturing its standard output, standard error, and exit code. No error is thrown if the process terminates unsuccessfully.
3. `IO.Process.spawn` starts another program asynchronously and returns a data structure that can be used to access the process's standard input, output, and error streams.

def

```
IO.Process.run (args : IO.Process.SpawnArgs) : IO String
```

Runs a process to completion, blocking until it terminates. If the child process terminates successfully with exit code 0, its standard output is returned. An exception is thrown if it terminates with any other exit code.

▼ *Example: Running a Program*

When run, this program concatenates its own source code with itself twice using the Unix tool `cat`.

```
-- Main.lean begins here
def main : IO Unit := do
  let src2 ← IO.Process.run {cmd := "cat", args := #["Main.lean",
"Main.lean"]}
  IO.println src2
-- Main.lean ends here
```

Its output is:

```
stdout
  1 -- Main.lean begins here
  2 def main : IO Unit := do
  3   let src2 ← IO.Process.run {cmd := "cat", args := #["Main.l
  4   IO.println src2
  5 -- Main.lean ends here
  6 -- Main.lean begins here
  7 def main : IO Unit := do
  8   let src2 ← IO.Process.run {cmd := "cat", args := #["Main.l
  9   IO.println src2
 10 -- Main.lean ends here
```

▼ *Example: Running a Program on a File*

This program uses the Unix utility `grep` as a filter to find four-digit palindromes. It creates a file that contains all numbers from `0` through `9999`, and then invokes `grep` on it, reading the result from its standard output.

```
def main : IO Unit := do
  -- Feed the input to the subprocess
  IO.FS.withFile "numbers.txt" .write fun h =>
    for i in [0:10000] do
```

```
      h.putStrLn (toString i)

  let palindromes ← IO.Process.run {
    cmd := "grep",
    args := #[r#"^\([0-9]\)\([0-9]\)\2\1$"#, "numbers.txt"]
  }

  let count := palindromes.trim.splitOn "\n" |>.length

  IO.println s!"There are {count} four-digit palindromes."
```

Its output is:

```
stdout
  1 There are 90 four-digit palindromes.
```

def

```
IO.Process.output (args : IO.Process.SpawnArgs) : IO
IO.Process.Output
```

Runs a process to completion and captures its output and exit code. The child process is run with a null standard input, and the current process blocks until it has run to completion.

The specifications of standard input, output, and error handles in *args* are ignored.

▼ *Example: Checking Exit Codes*

When run, this program first invokes `cat` on a nonexistent file and displays the resulting error code. It then concatenates its own source code with itself twice using the Unix tool `cat`.

```
-- Main.lean begins here
def main : IO UInt32 := do
  let src1 ← IO.Process.output {cmd := "cat", args := #
["Nonexistent.lean"]}
  IO.println s!"Exit code from failed process: {src1.exitCode}"

  let src2 ← IO.Process.output {cmd := "cat", args := #
["Main.lean", "Main.lean"]}
  if src2.exitCode == 0 then
    IO.println src2.stdout
  else
    IO.eprintln "Concatenation failed"
    return 1

  return 0
-- Main.lean ends here
```

Its output is:

```
stdout
Exit code from failed process: 1
-- Main.lean begins here
def main : IO UInt32 := do
  let src1 ← IO.Process.output {cmd := "cat", args := #["Non
  IO.println s!"Exit code from failed process: {src1.exitCod
  let src2 ← IO.Process.output {cmd := "cat", args := #["Mai
  if src2.exitCode == 0 then
    IO.println src2.stdout
  else
    IO.eprintln "Concatenation failed"
    return 1

  return 0
-- Main.lean ends here
-- Main.lean begins here
def main : IO UInt32 := do
  let src1 ← IO.Process.output {cmd := "cat", args := #["Non
  IO.println s!"Exit code from failed process: {src1.exitCod
  let src2 ← IO.Process.output {cmd := "cat", args := #["Mai
  if src2.exitCode == 0 then
    IO.println src2.stdout
  else
    IO.eprintln "Concatenation failed"
    return 1

  return 0
-- Main.lean ends here

```

opaque

```
IO.Process.spawn (args : IO.Process.SpawnArgs) :
  IO (IO.Process.Child args.toStdioConfig)
```

Starts a child process with the provided configuration. The child process is spawned using operating system primitives, and it can be written in any language.

The child process runs in parallel with the parent.

If the child process's standard input is a pipe, use `IO.Process.Child.takeStdin` to make it possible to close the child's standard input before the process terminates, which provides the child with an end-of-file marker.

▼ *Example: Asynchronous Subprocesses*

This program uses the Unix utility `grep` as a filter to find four-digit palindromes. It feeds all numbers from `0` through `9999` to the `grep` process and then reads its result. This code is only correct when `grep` is sufficiently fast and when the output pipe is large enough to contain all 90 four-digit palindromes.

```
def main : IO Unit := do
  let grep ← IO.Process.spawn {
    cmd := "grep",
    args := #[r#"^\([0-9]\)\([0-9]\)\2\1$"#],
    stdin := .piped,
    stdout := .piped,
    stderr := .null
  }

  -- Feed the input to the subprocess
  for i in [0:10000] do
    grep.stdin.putStrLn (toString i)

  -- Consume its output, after waiting 100ms for grep to process
the data.
  IO.sleep 100
  let count := (← grep.stdout.readToEnd).trim.splitOn "\n"
|>.length

  IO.println s!"There are {count} four-digit palindromes."
```

Its output is:

```
stdout
  1 There are 90 four-digit palindromes.
```

structure

`IO.Process.SpawnArgs : Type`

Configuration for a child process to be spawned.

Use IO.Process.spawn to start the child process. IO.Process.output and IO.Process.run can be used when the child process should be run to completion, with its output and/or error code captured.

Constructor

IO.Process.SpawnArgs.mk

Extends

- ☑ IO.Process.StdioConfig

Fields

stdin : IO.Process.Stdio

Inherited from IO.Process.StdioConfig

stdout : IO.Process.Stdio

Inherited from IO.Process.StdioConfig

stderr : IO.Process.Stdio

Inherited from IO.Process.StdioConfig

cmd : String

Command name.

args : Array String

Arguments for the command.

cwd : Option System.FilePath

The child process's working directory. Inherited from the parent current process if none.

env : Array (String × Option String)

Add or remove environment variables for the child process.

The child process inherits the parent's environment, as modified by *env*. Keys in the array are the names of environment variables. A none, causes the entry to be removed from the environment, and some sets the variable to the new value, adding it if necessary. Variables are processed from left to right.

inheritEnv : Bool

Inherit environment variables from the spawning process.

setsid : Bool

Starts the child process in a new session and process group using *setsid*. Currently a no-op on non-POSIX platforms.

structure

```
IO.Process.StdioConfig : Type
```

Configuration for the standard input, output, and error handles of a child process.

Constructor

IO.Process.StdioConfig.mk

Fields

stdin : IO.Process.Stdio

Configuration for the process' stdin handle.

stdout : IO.Process.Stdio

Configuration for the process' stdout handle.

stderr : IO.Process.Stdio

Configuration for the process' stderr handle.

inductive type

```
IO.Process.Stdio : Type
```

Whether the standard input, output, and error handles of a child process should be attached to pipes, inherited from the parent, or null.

If the stream is a pipe, then the parent process can use it to communicate with the child.

Constructors

| piped : IO.Process.Stdio

The stream should be attached to a pipe.

| inherit : IO.Process.Stdio

The stream should be inherited from the parent process.

| null : IO.Process.Stdio

The stream should be empty.

def

```
IO.Process.Stdio.toHandleType : IO.Process.Stdio → Type
```

The type of handles that can be used to communicate with a child process on its standard input, output, or error streams.

For `IO.Process.Stdio.piped`, this type is `IO.FS.Handle`. Otherwise, it is `Unit`, because no communication is possible.

structure

```
IO.Process.Child (cfg : IO.Process.StdioConfig) : Type
```

A child process that was spawned with configuration `cfg`.

The configuration determines whether the child process's standard input, standard output, and standard error are `IO.FS.Handle`s or `Unit`.

Fields

`stdin : cfg.stdin.toHandleType`

The child process's standard input handle, if it was configured as `IO.Process.Stdio.piped`, or `()` otherwise.

`stdout : cfg.stdout.toHandleType`

The child process's standard output handle, if it was configured as `IO.Process.Stdio.piped`, or `()` otherwise.

`stderr : cfg.stderr.toHandleType`

The child process's standard error handle, if it was configured as `IO.Process.Stdio.piped`, or `()` otherwise.

opaque

```
IO.Process.Child.wait {cfg : IO.Process.StdioConfig} :
  IO.Process.Child cfg → IO UInt32
```

Blocks until the child process has exited and return its exit code.

opaque

```
IO.Process.Child.tryWait {cfg : IO.Process.StdioConfig} :
  IO.Process.Child cfg → IO (Option UInt32)
```

Checks whether the child has exited. Returns `none` if the process has not exited, or its exit code if it has.

opaque

```
IO.Process.Child.kill {cfg : IO.Process.StdioConfig} :
  IO.Process.Child cfg → IO Unit
```

Terminates the child process using the `SIGTERM` signal or a platform analogue.

If the process was started using `SpawnArgs.setsid`, terminates the entire process group instead.

opaque

```
IO.Process.Child.takeStdin {cfg : IO.Process.StdioConfig} :
  IO.Process.Child cfg →
    IO
      (cfg.stdin.toHandleType ×
        IO.Process.Child
          { stdin := IO.Process.Stdio.null, stdout := cfg.stdout,
            stderr := cfg.stderr })
```

Extracts the `stdin` field from a `Child` object, allowing the handle to be closed while maintaining a reference to the child process.

File handles are closed when the last reference to them is dropped. Closing the child's standard input causes an end-of-file marker. Because the `Child` object has a reference to the standard input, this operation is necessary in order to close the stream while the process is running (e.g. to extract its exit code after calling `Child.wait`). Many processes do not terminate until their standard input is exhausted.

▼ *Example: Closing a Subprocess's Standard Input*

This program uses the Unix utility `grep` as a filter to find four-digit palindromes, ensuring that the subprocess terminates successfully. It feeds all numbers from `0` through `9999` to the `grep` process, then closes the process's standard input, which causes it to terminate. After checking `grep`'s exit code, the program extracts its result.

```
def main : IO UInt32 := do
  let grep ← do
    let (stdin, child) ← (← IO.Process.spawn {
      cmd := "grep",
      args := #[r#"^\([0-9]\)\([0-9]\)\2\1$"#],
      stdin := .piped,
      stdout := .piped,
      stderr := .null
    }).takeStdin

    -- Feed the input to the subprocess
    for i in [0:10000] do
      stdin.putStrLn (toString i)
```

```
    -- Return the child without its stdin handle.
    -- This closes the handle, because there are
    -- no more references to it.
    pure child

  -- Wait for grep to terminate
  if (← grep.wait) != 0 then
    IO.eprintln s!"grep terminated unsuccessfully"
    return 1

  -- Consume its output
  let count := (← grep.stdout.readToEnd).trim.splitOn "\n"
|>.length

  IO.println s!"There are {count} four-digit palindromes."
  return 0
```

Its output is:

```
stdout
  1 There are 90 four-digit palindromes.
```

structure

IO.Process.Output : Type

The result of running a process to completion.

Constructor

IO.Process.Output.mk

Fields

exitCode : UInt32

The process's exit code.

stdout : String

Everything that was written to the process's standard output.

stderr : String

Everything that was written to the process's standard error.

15.10. Random Numbers

def

```
IO.setRandSeed (n : Nat) : BaseIO Unit
```

Seeds the random number generator state used by `IO.rand`.

def

```
IO.rand (lo hi : Nat) : BaseIO Nat
```

Returns a pseudorandom number between *lo* and *hi*, using and updating a saved random generator state.

This state can be seeded using `IO.setRandSeed`.

def

```
randBool.{u} {gen : Type u} [RandomGen gen] (g : gen) : Bool × gen
```

Generates a random Boolean.

def

```
randNat.{u} {gen : Type u} [RandomGen gen] (g : gen) (lo hi : Nat)
:
  Nat × gen
```

Generates a random natural number in the interval [lo, hi].

15.10.1. Random Generators

type class

```
RandomGen.{u} (g : Type u) : Type u
```

Interface for random number generators.

Instance Constructor

`RandomGen.mk.{u}`

Methods

```
range : g → Nat × Nat
```

range returns the range of values returned by the generator.

```
next : g → Nat × g
```

next operation returns a natural number that is uniformly distributed the range returned by *range* (including both end points), and a new generator.

```
split : g → g × g
```

The 'split' operation allows one to obtain two distinct random number generators. This is very useful in functional programs (for example, when passing a random number generator down to recursive calls).

structure

```
StdGen : Type
```

"Standard" random number generator.

def

```
stdRange : Nat × Nat
```

The range of values returned by `StdGen`

def

```
stdNext : StdGen → Nat × StdGen
```

The next value from a `StdGen`, paired with an updated generator state.

def

```
stdSplit : StdGen → StdGen × StdGen
```

Splits a `StdGen` into two separate states.

def

```
mkStdGen (s : Nat := 0) : StdGen
```

Returns a standard number generator.

15.10.2. System Randomness

opaque

```
IO.getRandomBytes (nBytes : USize) : IO ByteArray
```

Reads bytes from a system entropy source. It is not guaranteed to be cryptographically secure.

If *nBytes* is 0, returns immediately with an empty buffer.

15.11. Tasks and Threads

Tasks are the fundamental primitive for writing multi-threaded code. A `Task α` represents a computation that, at some point, will *resolve* to a value of type α; it may be computed on a separate thread. When a task has resolved, its value can be read; attempting to get the value of a task before it resolves causes the current thread to block until the task has resolved. Tasks are similar to promises in JavaScript, `JoinHandle` in Rust, and `Future` in Scala.

Tasks may either carry out pure computations or `IO` actions. The API of pure tasks resembles that of thunks: `Task.spawn` creates a `Task α` from a function in `Unit → α`, and `Task.get` waits until the function's value has been computed and then returns it. The value is cached, so subsequent requests do not need to recompute it. The key difference lies in when the computation occurs: while the values of thunks are not computed until they are forced, tasks execute opportunistically in a separate thread.

Tasks in `IO` are created using `IO.asTask`. Similarly, `BaseIO.asTask` and `EIO.asTask` create tasks in other `IO` monads. These tasks may have side effects, and can communicate with other tasks.

When the last reference to a task is dropped it is *cancelled*. Pure tasks created with `Task.spawn` are terminated upon cancellation. Tasks spawned with `IO.asTask`, `EIO.asTask`, or `BaseIO.asTask` continue executing and must explicitly check for cancellation using `IO.checkCanceled`. Tasks may be explicitly cancelled using `IO.cancel`.

The Lean runtime maintains a thread pool for running tasks. The size of the thread pool is determined by the environment variable `LEAN_NUM_THREADS` if it is set, or by the number of logical processors on the current machine otherwise. The size of the thread pool is not a hard limit; in certain situations it may be exceeded to avoid deadlocks. By default, these threads are used to run tasks; each task has a *priority* (`Task.Priority`), and higher-priority tasks take precedence over lower-priority tasks. Tasks may also be assigned to dedicated threads by spawning them with a sufficiently high priority.

type

```
Task.{u} (α : Type u) : Type u
```

`Task α` is a primitive for asynchronous computation. It represents a computation that will resolve to a value of type *α*, possibly being computed on another thread. This is similar to `Future` in Scala, `Promise` in Javascript, and `JoinHandle` in Rust.

The tasks have an overridden representation in the runtime.

15.11.1. Creating Tasks

Pure tasks should typically be created with `Task.spawn`, as `Task.pure` is a task that's already been resolved with the provided value. Impure tasks are created by one of the `asTask` actions.

15.11.1.1. Pure Tasks

Pure tasks may be created outside the `IO` family of monads. They are terminated when the last reference to them is dropped.

def

```
Task.spawn.{u} {α : Type u} (fn : Unit → α)
  (prio : Task.Priority := Task.Priority.default) : Task α
```

`spawn fn : Task α` constructs and immediately launches a new task for evaluating the function `fn () : α` asynchronously.

`prio`, if provided, is the priority of the task.

constructor of Task

```
Task.pure.{u} {α : Type u} (get : α) : Task α
```

`Task.pure (a : α)` constructs a task that is already resolved with value `a`.

15.11.1.2. Impure Tasks

When spawning a task with side effects using one of the `asTask` functions, it's important to actually execute the resulting `IO` action. A task is spawned each time the resulting action is executed, not when `asTask` is called. Impure tasks continue running even when there are no references to them, though this does result in cancellation being requested. Cancellation may also be explicitly requested using `IO.cancel`. The impure task must check for cancellation using `IO.checkCanceled`.

opaque

```
BaseIO.asTask {α : Type} (act : BaseIO α)
  (prio : Task.Priority := Task.Priority.default) : BaseIO (Task α)
```

Runs `act` in a separate `Task`, with priority `prio`.

Running the resulting `BaseIO` action causes the task to be started eagerly. Pure accesses to the `Task` do not influence the impure `act`.

Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being

dropped.

def

```
EIO.asTask {ε α : Type} (act : EIO ε α)
  (prio : Task.Priority := Task.Priority.default) :
  BaseIO (Task (Except ε α))
```

Runs *act* in a separate Task, with priority *prio*. Because EIO *ε* actions may throw an exception of type *ε*, the result of the task is an Except *ε α*.

Running the resulting IO action causes the task to be started eagerly. Pure accesses to the Task do not influence the impure *act*.

Unlike pure tasks created by Task.spawn, tasks created by this function will run even if the last reference to the task is dropped. The *act* should explicitly check for cancellation via IO.checkCanceled if it should be terminated or otherwise react to the last reference being dropped.

def

```
IO.asTask {α : Type} (act : IO α)
  (prio : Task.Priority := Task.Priority.default) :
  BaseIO (Task (Except IO.Error α))
```

Runs *act* in a separate Task, with priority *prio*. Because IO actions may throw an exception of type IO.Error, the result of the task is an Except IO.Error *α*.

Running the resulting BaseIO action causes the task to be started eagerly. Pure accesses to the Task do not influence the impure *act*. Because IO actions may throw an exception of type IO.Error, the result of the task is an Except IO.Error *α*.

Unlike pure tasks created by Task.spawn, tasks created by this function will run even if the last reference to the task is dropped. The *act* should explicitly check for cancellation via IO.checkCanceled if it should be terminated or otherwise react to the last reference being dropped.

15.11.1.3. Priorities

Task priorities are used by the thread scheduler to assign tasks to threads. Within the priority range default–max, higher-priority tasks always take precedence over lower-priority tasks. Tasks spawned with priority dedicated are assigned their own dedicated threads and do not contend with other tasks for the threads in the thread pool.

def

```
Task.Priority : Type
```

Task priority.

Tasks with higher priority will always be scheduled before tasks with lower priority. Tasks with a priority greater than `Task.Priority.max` are scheduled on dedicated threads.

def

```
Task.Priority.default : Task.Priority
```

The default priority for spawned tasks, also the lowest priority: `0`.

def

```
Task.Priority.max : Task.Priority
```

The highest regular priority for spawned tasks: `8`.

Spawning a task with a priority higher than `Task.Priority.max` is not an error but will spawn a dedicated worker for the task. This is indicated using `Task.Priority.dedicated`. Regular priority tasks are placed in a thread pool and worked on according to their priority order.

def

```
Task.Priority.dedicated : Task.Priority
```

Indicates that a task should be scheduled on a dedicated thread.

Any priority higher than `Task.Priority.max` will result in the task being scheduled immediately on a dedicated thread. This is particularly useful for long-running and/or I/O-bound tasks since Lean will, by default, allocate no more non-dedicated workers than the number of cores to reduce context switches.

15.11.2. Task Results

def

```
Task.get.{u} {α : Type u} (self : Task α) : α
```

Blocks the current thread until the given task has finished execution, and then returns the result of the task. If the current thread is itself executing a (non-dedicated) task, the maximum threadpool size is temporarily increased by one while waiting so as to ensure the process cannot be deadlocked by threadpool starvation. Note that when the current thread is unblocked, more tasks than the configured threadpool size may temporarily be running at the same time until sufficiently many tasks have finished.

Task.map and Task.bind should be preferred over Task.get for setting up task dependencies where possible as they do not require temporarily growing the threadpool in this way. In particular, calling Task.get in a task continuation with (sync := true) will panic as the continuation is decidedly not "cheap" in this case and deadlocks may otherwise occur. The waited-upon task should instead be returned and unwrapped using Task.bind/IO.bindTask.

opaque

```
IO.wait {α : Type} (t : Task α) : BaseIO α
```

Waits for the task to finish, then returns its result.

opaque

```
IO.waitAny {α : Type} (tasks : List (Task α))
  (h : tasks.length > 0 := by exact Nat.zero_lt_succ _) : BaseIO α
```

Waits until any of the tasks in the list has finished, then return its result.

15.11.3. Sequencing Tasks

These operators create new tasks from old ones. When possible, it's good to use Task.map or Task.bind instead of manually calling Task.get in a new task because they don't temporarily increase the size of the thread pool.

def

```
Task.map.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α → β)
  (x : Task α) (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : Task β
```

map f x maps function *f* over the task *x*: that is, it constructs (and immediately launches) a new task which will wait for the value of *x* to be available and then calls *f* on the result.

prio, if provided, is the priority of the task. If *sync* is set to true, *f* is executed on the current thread if *x* has already finished and otherwise on the thread that *x* finished on. *prio* is ignored in this case. This should only be done when executing *f* is cheap and non-blocking.

def

```
Task.bind.{u_1, u_2} {α : Type u_1} {β : Type u_2} (x : Task α)
  (f : α → Task β) (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : Task β
```

`bind x f` does a monad "bind" operation on the task *x* with function *f*: that is, it constructs (and immediately launches) a new task which will wait for the value of *x* to be available and then calls *f* on the result, resulting in a new task which is then run for a result.

prio, if provided, is the priority of the task. If *sync* is set to true, *f* is executed on the current thread if *x* has already finished and otherwise on the thread that *x* finished on. *prio* is ignored in this case. This should only be done when executing *f* is cheap and non-blocking.

def

```
Task.mapList.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : List α → β)
  (tasks : List (Task α))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : Task β
```

Creates a task that, when all *tasks* have finished, computes the result of *f* applied to their results.

opaque

```
BaseIO.mapTask.{u_1} {α : Type u_1} {β : Type} (f : α → BaseIO β)
  (t : Task α) (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task β)
```

Creates a new task that waits for *t* to complete and then runs the `BaseIO` action *f* on its result. This new task has priority *prio*.

Running the resulting `BaseIO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped.

def

```
EIO.mapTask.{u_1} {α : Type u_1} {ε β : Type} (f : α → EIO ε β)
  (t : Task α) (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task (Except ε β))
```

Creates a new task that waits for *t* to complete and then runs the `IO` action *f* on its result. This new task has priority *prio*.

Running the resulting `BaseIO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it

should be terminated or otherwise react to the last reference being dropped. Because `EIO ε` actions may throw an exception of type ε, the result of the task is an `Except ε α`.

def

```
IO.mapTask.{u_1} {α : Type u_1} {β : Type} (f : α → IO β) (t : Task
α)
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task (Except IO.Error β))
```

Creates a new task that waits for t to complete and then runs the `IO` action f on its result. This new task has priority $prio$.

Running the resulting `BaseIO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped. Because `IO` actions may throw an exception of type `IO.Error`, the result of the task is an `Except IO.Error α`.

def

```
BaseIO.mapTasks.{u_1} {α : Type u_1} {β : Type} (f : List α →
BaseIO β)
  (tasks : List (Task α))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task β)
```

Creates a new task that waits for all the tasks in the list $tasks$ to complete, and then runs the `IO` action f on their results. This new task has priority $prio$.

Running the resulting `BaseIO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped.

def

```
EIO.mapTasks.{u_1} {α : Type u_1} {ε β : Type} (f : List α → EIO ε
β)
  (tasks : List (Task α))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task (Except ε β))
```

Creates a new task that waits for all the tasks in the list $tasks$ to complete, and then runs the `EIO ε` action f on their results. This new task has priority $prio$.

Running the resulting `BaseIO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped.

def

```
IO.mapTasks.{u_1} {α : Type u_1} {β : Type} (f : List α → IO β)
  (tasks : List (Task α))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task (Except IO.Error β))
```

`IO` specialization of `EIO.mapTasks`.

opaque

```
BaseIO.bindTask.{u_1} {α : Type u_1} {β : Type} (t : Task α)
  (f : α → BaseIO (Task β))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task β)
```

Creates a new task that waits for t to complete, runs the `IO` action f on its result, and then continues as the resulting task. This new task has priority $prio$.

Running the resulting `BaseIO` action causes this new task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped.

def

```
EIO.bindTask.{u_1} {α : Type u_1} {ε β : Type} (t : Task α)
  (f : α → EIO ε (Task (Except ε β)))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task (Except ε β))
```

Creates a new task that waits for t to complete, runs the `EIO` ε action f on its result, and then continues as the resulting task. This new task has priority $prio$.

Running the resulting `BaseIO` action causes this new task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped. Because `EIO` ε actions may throw an exception of type ε, the result of the task is an `Except` ε α.

def

```
IO.bindTask.{u_1} {α : Type u_1} {β : Type} (t : Task α)
  (f : α → IO (Task (Except IO.Error β)))
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO (Task (Except IO.Error β))
```

Creates a new task that waits for `t` to complete, runs the `IO` action `f` on its result, and then continues as the resulting task. This new task has priority `prio`.

Running the resulting `BaseIO` action causes this new task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped. Because `IO` actions may throw an exception of type `IO.Error`, the result of the task is an `Except IO.Error α`.

def

```
BaseIO.chainTask.{u_1} {α : Type u_1} (t : Task α) (f : α → BaseIO
Unit)
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : BaseIO Unit
```

Creates a new task that waits for `t` to complete and then runs the `IO` action `f` on its result. This new task has priority `prio`.

This is a version of `BaseIO.mapTask` that ignores the result value.

Running the resulting `BaseIO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped.

def

```
EIO.chainTask.{u_1} {α : Type u_1} {ε : Type} (t : Task α)
  (f : α → EIO ε Unit) (prio : Task.Priority :=
Task.Priority.default)
  (sync : Bool := false) : EIO ε Unit
```

Creates a new task that waits for `t` to complete and then runs the `EIO ε` action `f` on its result. This new task has priority `prio`.

This is a version of `EIO.mapTask` that ignores the result value.

Running the resulting `EIO ε` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The `act` should explicitly check for cancellation via `IO.checkCanceled` if it

should be terminated or otherwise react to the last reference being dropped.

def

```
IO.chainTask.{u_1} {α : Type u_1} (t : Task α) (f : α → IO Unit)
  (prio : Task.Priority := Task.Priority.default)
  (sync : Bool := false) : IO Unit
```

Creates a new task that waits for `t` to complete and then runs the `IO` action `f` on its result. This new task has priority `prio`.

This is a version of `IO.mapTask` that ignores the result value.

Running the resulting `IO` action causes the task to be started eagerly. Unlike pure tasks created by `Task.spawn`, tasks created by this function will run even if the last reference to the task is dropped. The act should explicitly check for cancellation via `IO.checkCanceled` if it should be terminated or otherwise react to the last reference being dropped.

15.11.4. Cancellation and Status

Impure tasks should use `IO.checkCanceled` to react to cancellation, which occurs either as a result of `IO.cancel` or when the last reference to the task is dropped. Pure tasks are terminated automatically upon cancellation.

opaque

```
IO.cancel.{u_1} {α : Type u_1} : Task α → BaseIO Unit
```

Requests cooperative cancellation of the task. The task must explicitly call `IO.checkCanceled` to react to the cancellation.

opaque

```
IO.checkCanceled : BaseIO Bool
```

Checks whether the current task's cancellation flag has been set by calling `IO.cancel` or by dropping the last reference to the task.

def

```
IO.hasFinished.{u_1} {α : Type u_1} (task : Task α) : BaseIO Bool
```

Checks whether the task has finished execution, at which point calling `Task.get` will return immediately.

opaque

```
IO.getTaskState.{u_1} {α : Type u_1} : Task α → BaseIO IO.TaskState
```

Returns the current state of a task in the Lean runtime's task manager.

For tasks derived from `Promise`s, the states `waiting` and `running` should be considered equivalent.

inductive type

`IO.TaskState : Type`

The current state of a `Task` in the Lean runtime's task manager.

Constructors

| `waiting : IO.TaskState`

The `Task` is waiting to be run.

It can be waiting for dependencies to complete or sitting in the task manager queue waiting for a thread to run on.

| `running : IO.TaskState`

The `Task` is actively running on a thread or, in the case of a `Promise`, waiting for a call to `IO.Promise.resolve`.

| `finished : IO.TaskState`

The `Task` has finished running and its result is available. Calling `Task.get` or `IO.wait` on the task will not block.

opaque

`IO.getTID : BaseIO UInt64`

Returns the thread ID of the calling thread.

15.11.5. Promises

Promises represent a value that will be supplied in the future. Supplying the value is called *resolving* the promise. Once created, a promise can be stored in a data structure or passed around like any other value, and attempts to read from it will block until it is resolved.

def

`IO.Promise (α : Type) : Type`

`Promise α` allows you to create a `Task α` whose value is provided later by calling `resolve`.

Typical usage is as follows:

1. `let promise ← Promise.new` creates a promise
2. `promise.result? : Task (Option α)` can now be passed around
3. `promise.result?.get` blocks until the promise is resolved
4. `promise.resolve a` resolves the promise
5. `promise.result?.get` now returns `some a`

If the promise is dropped without ever being resolved, `promise.result?.get` will return `none`. See `Promise.result!`/*`resultD`* for other ways to handle this case.

opaque

`IO.Promise.new {α : Type} [Nonempty α] : BaseIO (IO.Promise α)`

Creates a new `Promise`.

def

`IO.Promise.isResolved {α : Type} (promise : IO.Promise α) : BaseIO Bool`

Checks whether the promise has already been resolved, i.e. whether access to `result*` will return immediately.

opaque

`IO.Promise.result? {α : Type} (promise : IO.Promise α) : Task (Option α)`

Like `Promise.result`, but resolves to `none` if the promise is dropped without ever being resolved.

def

`IO.Promise.result! {α : Type} (promise : IO.Promise α) : Task α`

The result task of a `Promise`.

The task blocks until `Promise.resolve` is called. If the promise is dropped without ever being resolved, evaluating the task will panic and, when not using fatal panics, block forever. Use `Promise.result?` to handle this case explicitly.

def

```
IO.Promise.resultD {α : Type} (promise : IO.Promise α) (dflt : α) :
  Task α
```

Like `Promise.result`, but resolves to `dflt` if the promise is dropped without ever being resolved.

opaque

```
IO.Promise.resolve {α : Type} (value : α) (promise : IO.Promise α)
:
  BaseIO Unit
```

Resolves a `Promise`.

Only the first call to this function has an effect.

15.11.6. Communication Between Tasks

In addition to the types and operations described in this section, `IO.Ref` can be used as a lock. Taking the reference (using `take`) causes other threads to block when reading until the reference is `set` again. This pattern is described in the section on reference cells.

15.11.6.1. Channels

The types and functions in this section are available after importing `Std.Sync.Channel`.

structure

```
Std.Channel (α : Type) : Type
```

A multi-producer multi-consumer FIFO channel that offers both bounded and unbounded buffering and an asynchronous API, to switch into synchronous mode use `Channel.sync`.

If a channel needs to be closed to indicate some sort of completion event use `Std.CloseableChannel` instead. Note that `Std.CloseableChannel` introduces a need for error handling in some cases, thus `Std.Channel` is usually easier to use if applicable.

def

```
Std.Channel.new {α : Type} (capacity : Option Nat := none) :
  BaseIO (Std.Channel α)
```

Create a new channel, if:

- `capacity` is `none` it will be unbounded (the default)
- `capacity` is `some 0` it will always force a rendezvous between sender and receiver

- *capacity* is some *n* with n > 0 it will use a buffer of size *n* and begin blocking once it is filled

def

```
Std.Channel.send {α : Type} (ch : Std.Channel α) (v : α) :
  BaseIO (Task Unit)
```

Send a value through the channel, returning a task that will resolve once the transmission could be completed.

def

```
Std.Channel.recv {α : Type} [Inhabited α] (ch : Std.Channel α) :
  BaseIO (Task α)
```

Receive a value from the channel, returning a task that will resolve once the transmission could be completed. Note that the task may resolve to none if the channel was closed before it could be completed.

opaque

```
Std.Channel.forAsync {α : Type} [Inhabited α] (f : α → BaseIO Unit)
  (ch : Std.Channel α) (prio : Task.Priority :=
Task.Priority.default) :
  BaseIO (Task Unit)
```

ch.forAsync *f* calls *f* for every message received on *ch*.

Note that if this function is called twice, each message will only arrive at exactly one invocation.

def

```
Std.Channel.sync {α : Type} (ch : Std.Channel α) : Std.Channel.Sync
α
```

This function is a no-op and just a convenient way to expose the synchronous API of the channel.

def

```
Std.Channel.Sync (α : Type) : Type
```

A multi-producer multi-consumer FIFO channel that offers both bounded and unbounded buffering and a synchronous API. This type acts as a convenient layer to use a channel in a blocking fashion and is not actually different from the original channel.

If a channel needs to be closed to indicate some sort of completion event use `Std.CloseableChannel.Sync` instead. Note that `Std.CloseableChannel.Sync` introduces a need for error handling in some cases, thus `Std.Channel.Sync` is usually easier to use if applicable.

def

```
Std.CloseableChannel (α : Type) : Type
```

A multi-producer multi-consumer FIFO channel that offers both bounded and unbounded buffering and an asynchronous API, to switch into synchronous mode use `CloseableChannel.sync`.

Additionally `Std.CloseableChannel` can be closed if necessary, unlike `Std.Channel`. This introduces a need for error handling in some cases, thus it is usually easier to use `Std.Channel` if applicable.

def

```
Std.CloseableChannel.new {α : Type} (capacity : Option Nat := none)
:
  BaseIO (Std.CloseableChannel α)
```

Create a new channel, if:

- *capacity* is `none` it will be unbounded (the default)
- *capacity* is `some 0` it will always force a rendezvous between sender and receiver
- *capacity* is `some n` with `n > 0` it will use a buffer of size *n* and begin blocking once it is filled

Synchronous channels can also be read using `for` loops. In particular, there is an instance of type `ForIn m (Std.Channel.Sync α) α` for every monad *m* with a `MonadLiftT BaseIO m` instance and *α* with an `Inhabited α` instance.

15.11.6.2. Mutexes

The types and functions in this section are available after importing `Std.Sync.Mutex`.

type

```
Std.Mutex (α : Type) : Type
```

Mutual exclusion primitive (lock) guarding shared state of type *α*.

The type `Mutex α` is similar to `IO.Ref α`, except that concurrent accesses are guarded by a mutex instead of atomic pointer operations and busy-waiting.

def

```
Std.Mutex.new {α : Type} (a : α) : BaseIO (Std.Mutex α)
```

Creates a new mutex.

def

```
Std.Mutex.atomically {m : Type → Type} {α β : Type} [Monad m]
  [MonadLiftT BaseIO m] [MonadFinally m] (mutex : Std.Mutex α)
  (k : Std.AtomicT α m β) : m β
```

mutex`.atomically` *k* runs *k* with access to the mutex's state while locking the mutex.

Calling `mutex.atomically` while already holding the underlying `BaseMutex` in the same thread is undefined behavior. If this is unavoidable in your code, consider using `RecursiveMutex`.

def

```
Std.Mutex.atomicallyOnce {m : Type → Type} {α β : Type} [Monad m]
  [MonadLiftT BaseIO m] [MonadFinally m] (mutex : Std.Mutex α)
  (condvar : Std.Condvar) (pred : Std.AtomicT α m Bool)
  (k : Std.AtomicT α m β) : m β
```

mutex`.atomicallyOnce` *condvar pred k* runs *k*, waiting on *condvar* until *pred* returns true. Both *k* and *pred* have access to the mutex's state.

Calling `mutex.atomicallyOnce` while already holding the underlying `BaseMutex` in the same thread is undefined behavior. If this is unavoidable in your code, consider using `RecursiveMutex`.

def

```
Std.AtomicT (σ : Type) (m : Type → Type) (α : Type) : Type
```

`AtomicT α m` is the monad that can be atomically executed inside mutual exclusion primitives like `Mutex α` with outside monad *m*. The action has access to the state *α* of the mutex (via `get` and `set`).

15.11.6.3. Condition Variables

The types and functions in this section are available after importing `Std.Sync.Mutex`.

def

```
Std.Condvar : Type
```

Condition variable, a synchronization primitive to be used with a `BaseMutex` or `Mutex`.

The thread that wants to modify the shared variable must:

1. Lock the `BaseMutex` or `Mutex`
2. Work on the shared variable
3. Call `Condvar.notifyOne` or `Condvar.notifyAll` after it is done. Note that this may be done before or after the mutex is unlocked.

If working with a `Mutex` the thread that waits on the `Condvar` can use `Mutex.atomicallyOnce` to wait until a condition is true. If working with a `BaseMutex` it must:

1. Lock the `BaseMutex`.
2. Do one of the following:

- Use `Condvar.waitUntil` to (potentially repeatedly wait) on the condition variable until the condition is true.
- Implement the waiting manually by:
    1. Checking the condition
    2. Calling `Condvar.wait` which releases the `BaseMutex` and suspends execution until the condition variable is notified.
    3. Check the condition and resume waiting if not satisfied.

opaque

```
Std.Condvar.new : BaseIO Std.Condvar
```

Creates a new condition variable.

opaque

```
Std.Condvar.wait (condvar : Std.Condvar) (mutex : Std.BaseMutex) :
  BaseIO Unit
```

Waits until another thread calls `notifyOne` or `notifyAll`.

opaque

```
Std.Condvar.notifyOne (condvar : Std.Condvar) : BaseIO Unit
```

Wakes up a single other thread executing `wait`.

opaque

```
Std.Condvar.notifyAll (condvar : Std.Condvar) : BaseIO Unit
```

Wakes up all other threads executing `wait`.

def

```
Std.Condvar.waitUntil.{u_1} {m : Type → Type u_1} [Monad m]
  [MonadLiftT BaseIO m] (condvar : Std.Condvar) (mutex :
Std.BaseMutex)
  (pred : m Bool) : m Unit
```

Waits on the condition variable until the predicate is true.

16. The Simplifier

The simplifier is one of the most-used features of Lean. It performs inside-out rewriting of terms based on a database of simplification rules. The simplifier is highly configurable, and a number of tactics use it in different ways.

16.1. Invoking the Simplifier

Lean's simplifier can be invoked in a variety of ways. The most common patterns are captured in a set of tactics. The tactic reference contains a complete list of simplification tactics.

Simplification tactics all contain `simp` in their name. Aside from that, they are named according to a system of prefixes and suffixes that describe their functionality:

`-!` suffix
: Sets the `autoUnfold` configuration option to `true`, causing the simplifier unfold all definitions

`-?` suffix
: Causes the simplifier to keep track of which rules it employed during simplification and suggest a minimal simp set as an edit to the tactic script

`-_arith` suffix
: Enables the use of linear arithmetic simplification rules

`d-` prefix
: Causes the simplifier to simplify only with rewrites that hold definitionally

`-_all` suffix
: Causes the simplifier to repeatedly simplify all assumptions and the conclusion of the goal, taking as many hypotheses into account as possible, until no further simplification is possible

There are two further simplification tactics, `simpa` and `simpa!`, which are used to simultaneously simplify a goal and either a proof term or an assumption before discharging the goal. This simultaneous simplification makes proofs more robust to changes in the simp set.

16.1.1. Parameters

The simplification tactics have the following grammar:

syntax

Simplification Tactics

```
tactic ::= ...
    | simp optConfig only? ([ (simpStar | simpErase | simpLemma),*
] )? (at (term | locationType)*)?
```

In other words, an invocation of a simplification tactic takes the following modifiers, in order, all of which are optional:

- A set of configuration options, which should include the fields of `Lean.Meta.Simp.Config` or `Lean.Meta.DSimp.Config`, depending on whether the simplifier being invoked is a version of `simp` or a version of `dsimp`.
- The `only` modifier excludes the default simp set, instead beginning with an empty simp set. (i) Technically, the simp set always includes `eq_self` and `iff_self` in order to discharge reflexive cases.
- The lemma list adds or removes lemmas from the simp set. There are three ways to specify lemmas in the lemma list:
  - `*`, which adds all assumptions in the proof state to the simp set
  - `-` followed by a lemma, which removes the lemma from the simp set
  - A lemma specifier, consisting of the following in sequence:
    - An optional ↓ or ↑, which respectively cause the lemma to be applied before or after entering a subterm (↑ is the default). The simplifier typically simplifies subterms before attempting to simplify parent terms, as simplified arguments often make more rules applicable; ↓ causes the parent term to be simplified with the rule prior to the simplification of subterms.
    - An optional ←, which causes equational lemmas to be used from right to left rather than from left to right.
    - A mandatory lemma, which can be a simp set name, a lemma name, or a term. Terms are treated as if they were named lemmas with fresh names.
- A location specifier, preceded by `at`, which consists of a sequence of locations. Locations may be:

- The name of an assumption, indicating that its type should be simplified
- An asterisk `*`, indicating that all assumptions and the conclusion should be simplified
- A turnstile ⊢, indicating that the conclusion should be simplified

By default, only the conclusion is simplified.

▼ *Example: Location specifiers for* **`simp`**

In this proof state,

```
  p : Nat → Prop
  x : Nat
  h : p (x + 5 + 2)
h' : p (3 + x + 9)
⊢ p (6 + x + 1)
```

the tactic **`simp`** `+arith` simplifies only the goal:

```
  p : Nat → Prop
  x : Nat
  h : p (x + 5 + 2)
h' : p (3 + x + 9)
⊢ p (x + 7)
```

Invoking **`simp`** `+arith` **`at`** *h* yields a goal in which the hypothesis h has been simplified:

```
  p : Nat → Prop
  x : Nat
h' : p (3 + x + 9)
  h : p (x + 7)
⊢ p (6 + x + 1)
```

The conclusion can be additionally simplified by adding ⊢, that is, **`simp`** `+arith` **`at`** *h* ⊢:

```
  p : Nat → Prop
  x : Nat
h' : p (3 + x + 9)
  h : p (x + 7)
⊢ p (x + 7)
```

Using **`simp`** `+arith` **`at`** `*` simplifies all assumptions together with the conclusion:

```
  p : Nat → Prop
  x : Nat
  h : p (x + 7)
h' : p (x + 12)
```

```
⊢ p (x + 7)
```

16.2. Rewrite Rules

The simplifier has three kinds of rewrite rules:

Declarations to unfold

The simplifier will only unfold reducible definitions by default. However, a rewrite rule can be added for any semireducible or irreducible definition that causes the simplifier to unfold it as well. When the simplifier is running in definitional mode (`dsimp` and its variants), definition unfolding only replaces the defined name with its value; otherwise, it also uses the equational lemmas produced by the equation compiler.

Equational lemmas

The simplifier can treat equality proofs as rewrite rules, in which case the left side of the equality will be replaced with the right. These equational lemmas may have any number of parameters. The simplifier instantiates parameters to make the left side of the equality match the goal, and it performs a proof search to instantiate any additional parameters.

Simplification procedures

The simplifier supports simplification procedures, known as *simprocs*, that use Lean metaprogramming to perform rewrites that can't be efficiently specified using equations. Lean includes simprocs for the most important operations on built-in types.

Due to propositional extensionality, equational lemmas can rewrite propositions to simpler, logically equivalent propositions. When the simplifier rewrites a proof goal to `True`, it automatically closes it. As a special case of equational lemmas, propositions other than equality can be tagged as rewrite rules They are preprocessed into rules that rewrite the proposition to `True`.

▼ *Example: Rewriting Propositions*

When asked to simplify an equality of pairs:

```
α : Type
β : Type
w : α
y : α
x : β
z : β
⊢ (w, x) = (y, z)
```

`simp` yields a conjunction of equalities:

```
α : Type
β : Type
```

```
w : α
y : α
x : β
z : β
⊢ w ≍ y ∧ x ≍ z
```

The default simp set contains `Prod.mk.injEq`, which shows the equivalence of the two statements:

```
Prod.mk.injEq.{u, v} {α : Type u} {β : Type v} (fst : α) (snd :
β) :
  ∀ (fst_1 : α) (snd_1 : β),
    ((fst, snd) = (fst_1, snd_1)) = (fst = fst_1 ∧ snd = snd_1)
```

In addition to rewrite rules, **`simp`** has a number of built-in reduction rules, controlled by the `config` parameter. Even when the simp set is empty, **`simp`** can replace `let`-bound variables with their values, reduce **`match`** expressions whose scrutinees are constructor applications, reduce structure projections applied to constructors, or apply lambdas to their arguments.

16.3. Simp sets

A collection of rules used by the simplifier is called a *simp set*. A simp set is specified in terms of modifications from a *default simp set*. These modifications can include adding rules, removing rules, or adding a set of rules. The `only` modifier to the **`simp`** tactic causes it to start with an empty simp set, rather than the default one. Rules are added to the default simp set using the **`simp`** attribute.

attribute

Registering `simp` Lemmas

The **`simp`** attribute adds a declaration to the default simp set. If the declaration is a definition, the definition is marked for unfolding; if it is a theorem, then the theorem is registered as a rewrite rule.

```
attr ::= ...
    | simp
```

```
attr ::= ...
    | simp ↑ (← ?
       | <- ?)
```

```
attr ::= ...
    | simp ↓ (← ?
       | <- ?)
```

```
attr ::= ...
    | simp prio
```

Custom simp sets are created with `registerSimpAttr`, which must be run during initialization by placing it in an `initialize` block. As a side effect, it creates a new attribute with the same interface as **`simp`** that adds rules to the custom simp set. The returned value is a `SimpExtension`, which can be used to programmatically access the contents of the custom simp set. The **`simp`** tactics can be instructed to use the new simp set by including its attribute name in the rule list.

def

```
Lean.Meta.registerSimpAttr (attrName : Lean.Name) (attrDescr :
String)
  (ref : Lean.Name := by exact decl_name%) : IO
Lean.Meta.SimpExtension
```

Registers the given name as a custom simp set. Applying the name as an attribute to a name adds it to the simp set, and using the name as a parameter to the **`simp`** tactic causes **`simp`** to use the included lemmas.

Custom simp sets must be registered during initialization.

The description should be a short, singular noun phrase that describes the contents of the custom simp set.

def

```
Lean.Meta.SimpExtension : Type
```

The environment extension that contains a simp set, returned by `Lean.Meta.registerSimpAttr`.

Use the simp set's attribute or `Lean.Meta.addSimpTheorem` to add theorems to the simp set. Use `Lean.Meta.SimpExtension.getTheorems` to get the contents.

16.4. Simp Normal Forms

The default simp set contains all the theorems and simplification procedures marked with the **`simp`** attribute. The *simp normal form* of an expression is the result of applying the default simp set via the **`simp`** tactic until no more rules can be applied. When an expression is in simp normal form, it is as reduced as possible according to the default simp set, often making it easier to work with in proofs.

The **`simp`** tactic **does not guarantee confluence**, which means that the simp normal form of an expression may depend on the order in which the elements of the default simp set are applied. The order in which the rules are applied can be changed by assigning priorities when setting the **`simp`** attribute.

When designing a Lean library, it's important to think about what the appropriate simp normal form for the various combinations of the library's operators is. This can serve as a guide when selecting which rules the library should add to the default simp set. In particular, the right-hand side of simp lemmas should be in simp normal form; this helps ensure that simplification terminates. Additionally, each concept in the library should be expressed through one simp normal form, even if there are multiple equivalent ways to state it. If a concept is stated in two different ways in different simp lemmas, then some desired simplifications may not occur because the simplifier does not connect them.

Even though simplification doesn't need to be confluent, striving for confluence is helpful because it makes the library more predictable and tends to reveal missing or poorly chosen simp lemmas. The default simp set is as much a part of a library's interface as the type signatures of the constants that it exports.

Libraries should not add rules to the default simp set that don't mention at least one constant defined in the library. Otherwise, importing a library could change the behavior of `simp` for some unrelated library. If a library relies on additional simplification rules for definitions or declarations from other libraries, please create a custom simp set and either instruct users to use it or provide a dedicated tactic.

16.5. Terminal vs Non-Terminal Positions

To write maintainable proofs, avoid using `simp` without `only` unless it closes the goal. Such uses of `simp` that do not close a goal are referred to as *non-terminal simps*. This is because additions to the default simp set may make `simp` more powerful or just cause it to select a different sequence of rewrites and arrive at a different simp normal form. When `only` is specified, additional lemmas will not affect that invocation of the tactic. In practice, terminal uses of `simp` are not nearly as likely to be broken by the addition of new simp lemmas, and when they are, it's easier to understand the issue and fix it.

When working in non-terminal positions, `simp?` (or one of the other simplification tactics with `?` in their names) can be used to generate an appropriate invocation with `only`. Just as `apply?` or `rw?` suggest the use of relevant lemmas, `simp?` suggests an invocation of `simp` with a minimal simp set that was used to reach the normal form.

▼ *Example: Using* `simp?`

The non-terminal `simp?` in this proof suggests a smaller `simp` with `only`:

```
example (xs : Array Unit) : xs.size = 2 → xs = #[(), ()] := by
  intros
  ext
  simp?
  assumption
```

The suggested rewrite is:

```
Try this: simp only [List.size_toArray, List.length_cons,
List.length_nil, Nat.zero_add, Nat.reduceAdd]
```

which results in the more maintainable proof:

```
example (xs : Array Unit) : xs.size = 2 → xs = #[(), ()] := by
  intros
  ext

simp only [List.size_toArray, List.length_cons, List.length_nil,
Nat.zero_add, Nat.reduceAdd]
  assumption
```

16.6. Configuring Simplification

`simp` is primarily configured via a configuration parameter, passed as a named argument called `config`.

structure

`Lean.Meta.Simp.Config : Type`

The configuration for `simp`. Passed to `simp` using, for example, the `simp (config := {contextual := true})` syntax.

See also `Lean.Meta.Simp.neutralConfig` and `Lean.Meta.DSimp.Config`.

Constructor

`Lean.Meta.Simp.Config.mk`

Fields

`maxSteps : Nat`

The maximum number of subexpressions to visit when performing simplification. The default is 100000.

`maxDischargeDepth : Nat`

When simp discharges side conditions for conditional lemmas, it can recursively apply simplification. The *maxDischargeDepth* (default: 2) is the maximum recursion depth when recursively applying simplification to side conditions.

`contextual : Bool`

When *contextual* is true (default: `false`) and simplification encounters an implication *p* → *q* it includes *p* as an additional simp lemma when simplifying *q*.

`memoize : Bool`

When true (default: `true`) then the simplifier caches the result of simplifying each subexpression, if possible.

`singlePass : Bool`

When *`singlePass`* is `true` (default: `false`), the simplifier runs through a single round of simplification, which consists of running pre-methods, recursing using congruence lemmas, and then running post-methods. Otherwise, when it is `false`, it iteratively applies this simplification procedure.

`zeta : Bool`

When `true` (default: `true`), performs zeta reduction of **`let`** and **`have`** expressions. That is, **`let`** `x := v; e[x]` reduces to `e[v]`. If *`zetaHave`* is `false` then **`have`** expressions are not zeta reduced. See also *`zetaDelta`*.

`beta : Bool`

When `true` (default: `true`), performs beta reduction of applications of **`fun`** expressions. That is, (**`fun`** *`x => e[x])`* *`v`* reduces to `e[v]`.

`eta : Bool`

TODO (currently unimplemented). When `true` (default: `true`), performs eta reduction for **`fun`** expressions. That is, (**`fun`** *`x`* `=> f x)` reduces to *`f`*.

`etaStruct : Lean.Meta.EtaStructMode`

Configures how to determine definitional equality between two structure instances. See documentation for `Lean.Meta.EtaStructMode`.

`iota : Bool`

When `true` (default: `true`), reduces **`match`** expressions applied to constructors.

`proj : Bool`

When `true` (default: `true`), reduces projections of structure constructors.

`decide : Bool`

When `true` (default: `false`), rewrites a proposition *p* to `True` or `False` by inferring a `Decidable` *p* instance and reducing it.

`arith : Bool`

When `true` (default: `false`), simplifies simple arithmetic expressions.

`autoUnfold : Bool`

When `true` (default: `false`), unfolds definitions. This can be enabled using the **`simp!`** syntax.

`dsimp : Bool`

When `true` (default: `true`) then switches to *`dsimp`* on dependent arguments if there is no congruence theorem that would allow **`simp`** to visit them. When *`dsimp`* is `false`, then the argument is not visited.

`failIfUnchanged : Bool`

If *`failIfUnchanged`* is `true` (default: `true`), then calls to **`simp`**, *`dsimp`*, or **`simp_all`** will fail if they do not make progress.

`ground : Bool`

If *`ground`* is `true` (default: `false`), then ground terms are reduced. A term is ground when it does not contain free or meta variables. Reduction is interrupted at a function application `f ...` if `f` is marked to not be unfolded. Ground term reduction applies `@[seval]` lemmas.

`unfoldPartialApp : Bool`

If *`unfoldPartialApp`* is `true` (default: `false`), then calls to **`simp`**, *`dsimp`*, or **`simp_all`** will unfold even partial applications of `f` when we request `f` to be unfolded.

`zetaDelta : Bool`

When `true` (default: `false`), local definitions are unfolded. That is, given a local context containing `x : t := e`, then the free variable `x` reduces to `e`. Otherwise, `x` must be provided as a **`simp`** argument.

`index : Bool`

When `index` (default : `true`) is `false`, **`simp`** will only use the root symbol to find candidate **`simp`** theorems. It approximates Lean 3 **`simp`** behavior.

`implicitDefEqProofs : Bool`

If `implicitDefEqProofs := true`, **`simp`** does not create proof terms when the input and output terms are definitionally equal.

`zetaUnused : Bool`

When `true` (default : `true`), then **`simp`** will remove unused **`let`** and **`have`** expressions: **`let`** *`x := v; e`* simplifies to *`e`* when `x` does not occur in *`e`*. This option takes precedence over *`zeta`* and *`zetaHave`*.

`catchRuntime : Bool`

When `true` (default : `true`), then simps will catch runtime exceptions and convert them into **`simp`** exceptions.

`zetaHave : Bool`

When `false` (default: `true`), then disables zeta reduction of **have** expressions. If *`zeta`* is `false`, then this option has no effect. Unused **have**s are still removed if *`zeta`* or *`zetaUnused`* are true.

`letToHave : Bool`

When `true` (default : `true`), then **`simp`** will attempt to transform **`let`**s into **have**s if they are non-dependent. This only applies when `zeta := false`.

def

`Lean.Meta.Simp.neutralConfig : Lean.Meta.Simp.Config`

A neutral configuration for **`simp`**, turning off all reductions and other built-in simplifications.

structure

`Lean.Meta.DSimp.Config : Type`

The configuration for **`dsimp`**. Passed to **`dsimp`** using, for example, the **`dsimp`** `(config := {zeta := false})` syntax.

Implementation note: this structure is only used for processing the `(config := ...)` syntax, and it is not used internally. It is immediately converted to `Lean.Meta.Simp.Config` by `Lean.Elab.Tactic.elabSimpConfig`.

Constructor

`Lean.Meta.DSimp.Config.mk`

Fields

`zeta : Bool`

When `true` (default: `true`), performs zeta reduction of **`let`** and **have** expressions. That is, **`let`** `x := v; e[x]` reduces to `e[v]`. If *`zetaHave`* is `false` then **have** expressions are not zeta reduced. See also *`zetaDelta`*.

`beta : Bool`

When `true` (default: `true`), performs beta reduction of applications of **fun** expressions. That is, `(`**`fun`** *`x`* `=> e[x])` *`v`* reduces to `e[v]`.

`eta : Bool`

TODO (currently unimplemented). When `true` (default: `true`), performs eta reduction for **fun** expressions. That is, `(`**`fun`** *`x`* `=> f x)` reduces to *`f`*.

`etaStruct : Lean.Meta.EtaStructMode`

Configures how to determine definitional equality between two structure instances. See documentation for `Lean.Meta.EtaStructMode`.

`iota : Bool`

When `true` (default: `true`), reduces **`match`** expressions applied to constructors.

`proj : Bool`

When `true` (default: `true`), reduces projections of structure constructors.

`decide : Bool`

When `true` (default: `false`), rewrites a proposition *p* to `True` or `False` by inferring a `Decidable` *p* instance and reducing it.

`autoUnfold : Bool`

When `true` (default: `false`), unfolds definitions. This can be enabled using the **`simp!`** syntax.

`failIfUnchanged : Bool`

If *`failIfUnchanged`* is `true` (default: `true`), then calls to **`simp`**, **`dsimp`**, or **`simp_all`** will fail if they do not make progress.

`unfoldPartialApp : Bool`

If *`unfoldPartialApp`* is `true` (default: `false`), then calls to **`simp`**, **`dsimp`**, or **`simp_all`** will unfold even partial applications of `f` when we request `f` to be unfolded.

`zetaDelta : Bool`

When `true` (default: `false`), local definitions are unfolded. That is, given a local context containing `x : t := e`, then the free variable `x` reduces to `e`. Otherwise, `x` must be provided as a **`simp`** argument.

`index : Bool`

When `index` (default : `true`) is `false`, **`simp`** will only use the root symbol to find candidate **`simp`** theorems. It approximates Lean 3 **`simp`** behavior.

`zetaUnused : Bool`

When `true` (default : `true`), then **`simp`** will remove unused **`let`** and **`have`** expressions: **`let`** *`x := v; e`* simplifies to *e* when `x` does not occur in *e*.

`zetaHave : Bool`

When `false` (default: `true`), then disables zeta reduction of **have** expressions. If *zeta* is `false`, then this option has no effect. Unused **have**s are still removed if *zeta* or *zetaUnused* are true.

16.6.1. Options

Some global options affect **simp**:

option

`simprocs`

Default value: `true`

Enable/disable **simproc**s (simplification procedures).

option

`tactic.simp.trace`

Default value: `false`

When tracing is enabled, calls to **simp** or **dsimp** will print an equivalent **simp only** call.

option

`linter.unnecessarySimpa`

Default value: `true`

enable the 'unnecessary simpa' linter

option

`trace.Meta.Tactic.simp.rewrite`

Default value: `false`

enable/disable tracing for the given module and submodules

option

`trace.Meta.Tactic.simp.discharge`

Default value: `false`

enable/disable tracing for the given module and submodules

16.7. Simplification vs Rewriting

Both `simp` and `rw`/`rewrite` use equational lemmas to replace parts of terms with equivalent alternatives. Their intended uses and their rewriting strategies differ, however. Tactics in the `simp` family are primarily used to reformulate a problem in a standardized way, making it more amenable to both human understanding and further automation. In particular, simplification should never render an otherwise-provable goal impossible. Tactics in the `rw` family are primarily used to apply hand-selected transformations that do not always preserve provability nor place terms in standardized forms. These different emphases are reflected in the differences of behavior between the two families of tactics.

The `simp` tactics primarily rewrite from the inside out. The smallest possible expressions are simplified first so that they can unlock further simplification opportunities for the surrounding expressions. The `rw` tactics select the leftmost outermost subterm that matches the pattern, rewriting it a single time. Both tactics allow their strategy to be overridden: when adding a lemma to a simp set, the ↓ modifier causes it to be applied prior to the simplification of subterms, and the `occs` field of `rw`'s configuration parameter allows a different occurrence to be selected, either via a whitelist or a blacklist.

17. The `grind` tactic

17.1. Quick Start

- **Availability** – `grind` ships with Lean 4 (no extra installation) and is usable in any Lean file—just write `by grind`. No extra `import` is required beyond what your own definitions already need.
- **Library support** – Lean's standard library is already annotated with `@[grind]` attributes, so common lemmas are discovered automatically. Mathlib will be annotated gradually, starting with its most frequently used theories.
- **First proof**

  ```
  example (a b c : Nat) (h₁ : a = b) (h₂ : b = c) :
      a = c := by
    grind
  ```

  This succeeds instantly using congruence closure.
- **Power examples** – showcasing `grind`'s satellite solvers:
  - **Algebraic reasoning** (commutative-ring solver):

    ```
    example [CommRing α] [NoNatZeroDivisors α] (a b c : α)
        : a + b + c = 3 →
          a^2 + b^2 + c^2 = 5 →
          a^3 + b^3 + c^3 = 7 →
          a^4 + b^4 = 9 - c^4 := by
    ```

```
  grind
```

- **Finite-field style reasoning** (works in `Fin 11`):

```
example (x y : Fin 11) :
    x^2*y = 1 → x*y^2 = y → y*x = 1 := by
  grind
```

- **Linear integer arithmetic with case analysis**:

```
example (x y : Int) :
    27 ≤ 11*x + 13*y →
    11*x + 13*y ≤ 45 →
    -10 ≤ 7*x - 9*y →
    7*x - 9*y ≤ 4 → False := by
  grind
```

- **Useful flags**

  - `by grind (splits := 3) (ematch := 2)` – limit case splits / E-matching rounds.

17.2. What is `grind`?

A proof-automation tactic inspired by modern SMT solvers.

Picture a virtual white-board: every time **`grind`** discovers a new equality, inequality, or Boolean literal it writes that fact on the board, merges equivalent terms into buckets, and invites each engine to read from—and add back to—the shared white-board. The cooperating engines are:

- congruence closure,
- constraint propagation,
- E-matching,
- guided case analysis, and
- a suite of satellite theory solvers (linear integer arithmetic, commutative rings, …).

Lean supports dependent types and a powerful type-class system, and **`grind`** produces ordinary Lean proof terms for every fact it adds.

17.3. What `grind` is not.

`grind` is **not** designed for goals whose search space explodes combinatorially—think large-n pigeonhole instances, graph-coloring reductions, high-order N-queens boards, or a 200-variable Sudoku encoded as Boolean constraints. Such encodings require thousands (or millions) of case-splits that overwhelm **`grind`**'s branching search.

- **Bit-level or pure Boolean combinatorial problems** → use `by_decide`. The `by_decide` tactic calls a state-of-the-art SAT solver (e.g. CaDiCaL or Kissat) and then returns a **compact, machine-checkable certificate**. All heavy search happens outside Lean; the certificate is replayed and verified inside Lean, so trust is preserved (verification time scales with certificate size).
- **Full SMT problems that need substantial case analysis across multiple theories** (arrays, bit-vectors, rich arithmetic, quantifiers, …) → use the forthcoming **`lean-smt`** tactic—a tight Lean front-end for CVC5 that replays unsat cores or models inside Lean.

17.4. Congruence Closure

17.4.1. What is congruence closure?

Congruence closure maintains **equivalence classes of terms** under the reflexive–symmetric–transitive closure of "is equal to" *and* the rule that equal arguments yield equal function results. Formally, if `a = a'` and `b = b'`, then `f a b = f a' b'` is added. The algorithm merges classes until a fixed point is reached.

Think of a **shared white-board**:

1. Every hypothesis `h : t₁ = t₂` writes a line connecting `t₁` and `t₂`.
2. Each merge paints both terms the same color. Soon whole constellations (`f a`, `g (f a)`, …) share the color.
3. If `True` and `False` ever land in the same color—or likewise two different constructors of the *same inductive type* such as `none` and `some 1`—the goal is closed by contradiction.

17.4.2. How it differs from `simp`

- **`simp`** *rewrites* a goal, replacing occurrences of `t₁` with `t₂` as soon as it sees `h : t₁ = t₂`. The rewrite is directional and destructive.
- **`grind`** *accumulates* equalities bidirectionally. No term is rewritten; instead, both representatives live in the same class. All other engines (E-matching, theory solvers, propagation) can query these classes and add new facts, then the closure updates incrementally.

This makes congruence closure especially robust in the presence of symmetrical reasoning, mutual recursion, and large nestings of constructors where rewriting would duplicate work.

17.4.3. Minimal examples

```
example {α} (f g : α → α) (x y : α)
    (h₁ : x = y) (h₂ : f y = g y) :
    f x = g x := by
  -- After `h₁`, `x` and `y` share a class,
  -- `h₂` adds `f y = g y`, and
```

```
  -- closure bridges to `f x = g x`
  grind

example (a b c : Nat) (h : a = b) : (a, c) = (b, c) := by
  -- Pair constructor obeys congruence,
  -- so once `a = b` the tuples are equal
  grind
```

17.5. Debugging tip

When **`grind` fails** it prints the remaining subgoal **followed by all equivalence classes**. The two largest classes are shown as **True propositions** and **False propositions**, listing every literal currently known to be provable or refutable. Inspect these lists to spot missing facts or contradictory assumptions.

17.6. Constraint Propagation

Constraint propagation works on the **True** and **False** buckets of the white-board. Whenever a literal is added to one of those buckets, **`grind`** fires dozens of small *forward rules* to push its logical consequences:

- Boolean connectives — e.g. if A is `True`, mark `A ∨ B` as `True`; if `A ∧ B` is `True`, mark both A and B as `True`; if `A ∧ B` is `False`, at least one of A, B becomes `False`.
- Inductive datatypes — two different constructors (`none` vs `some _`) collapsing into the same class yields a contradiction; equal tuples yield equal components.
- Projections and casts — from `h : (x, y) = (x', y')` we derive `x = x'` and `y = y'`; any term `cast h a` is merged with `a` immediately (using a heterogeneous equality) so both live in the same class.
- Structural eta and definitional equalities — `⟨a, b⟩.1` propagates to `a`, etc.

Below is a *representative slice* of the propagators so you can see the style they follow. Each follows the same skeleton: inspect the truth-value of sub-expressions, push equalities (`pushEq`) or truth-values (`pushEqTrue` / `pushEqFalse`), and optionally close the goal if a contradiction (`closeGoal`) arises. A few high-signal examples:

```
/-- Propagate equalities *upwards* for conjunctions. -/
builtin_grind_propagator propagateAndUp ↑And := fun e => do
  let_expr And a b := e | return ()
  if (← isEqTrue a) then
    -- a = True  ⇒  (a ∧ b) = b
    pushEq e b <|
      mkApp3 (mkConst ``Grind.and_eq_of_eq_true_left)
        a b (← mkEqTrueProof a)
  else if (← isEqTrue b) then
    pushEq e a <|
      mkApp3 (mkConst ``Grind.and_eq_of_eq_true_right)
        a b (← mkEqTrueProof b)
```

```
  else if (← isEqFalse a) then
    pushEqFalse e <|
      mkApp3 (mkConst ``Grind.and_eq_of_eq_false_left)
        a b (← mkEqFalseProof a)
  else if (← isEqFalse b) then
    pushEqFalse e <|
      mkApp3 (mkConst ``Grind.and_eq_of_eq_false_right)
        a b (← mkEqFalseProof b)

/--
Truth flows *down* when the whole `And` is proven `True`.
-/
builtin_grind_propagator propagateAndDown ↓And :=
  fun e => do
  if (← isEqTrue e) then
    let_expr And a b := e | return ()
    let h ← mkEqTrueProof e
    pushEqTrue a <| mkApp3
      (mkConst ``Grind.eq_true_of_and_eq_true_left) a b h
    pushEqTrue b <| mkApp3
      (mkConst ``Grind.eq_true_of_and_eq_true_right) a b h
```

Other frequently-triggered propagators follow the same pattern:

Propagator	Handles	Notes
`propagateOrUp` / `propagateOrDown`	`a ∨ b`	True/False pushes for disjunctions
`propagateNotUp` / `propagateNotDown`	`¬ a`	Links `¬ a` with the Boolean of `a`
`propagateEqUp` / `propagateEqDown`	`a = b`	Bridges Booleans, detects constructor clash
`propagateIte` / `propagateDIte`	`ite` / `dite`	Replaces chosen branch once condition is fixed
`propagateEtaStruct`	structures tagged `[grind ext]`	Generates η-expansion `a = ⟨a.1, …⟩`

Many specialized variants for `Bool` mirror these rules exactly (e.g. `propagateBoolAndUp`).

17.6.1. Propagation-only examples

These goals are closed **purely** by constraint propagation—no case splits, no theory solvers:

```
example (a : Bool) : (a && !a) = false := by
  grind
```

```
-- Conditional (ite):
-- once the condition is true, ite picks the 'then' branch.
example (c : Bool) (t e : Nat) (h : c = true) :
    (if c then t else e) = t := by
  grind

-- Negation propagates truth downwards.
example (a : Bool) (h : (!a) = true) : a = false := by
  grind
```

These snippets run instantly because the relevant propagators (`propagateBoolAndUp`, `propagateIte`, `propagateBoolNotDown`) fire as soon as the hypotheses are internalized.

> **Note** If you toggle `set_option trace.grind.eqc true`, **`grind`** will print a line every time two equivalence classes merge—handy for seeing propagation in action.

Implementation tip **`grind`** is still under active development. Until the API has stabilized we recommend *refraining from custom elaborators or satellite solvers*. If you really need a project-local propagator, use the user-facing `grind_propagator` command rather than `builtin_grind_propagator` (the latter is reserved for Lean's own code). When adding new propagators keep them **small and orthogonal**—they should fire in ≤1 µs and either push one fact or close the goal. This keeps the propagation phase predictable and easy to debug.

We continuously expand and refine the rule set—expect the **Info View** to show increasingly rich `True`/`False` buckets over time. The full equivalence classes are displayed automatically *only when* **`grind`** *fails*, and only for the first subgoal it could not close—use this output to inspect missing facts and understand why the subgoal remains open.

17.7. Case Analysis

17.7.1. Selection heuristics

`grind` decides which sub-term to split on by combining three sources of signal:

1. **Structural flags** — quick Booleans that enable whole syntactic classes:
   - `splitIte` (default `true`) → split every `if … then … else …` term.
   - `splitMatch` (default `true`) → split on all `match` expressions (the **`grind`** analogue of Lean's **`split`** tactic, just like `splitIte`).
   - `splitImp` (default `false`) → when `true` splits on any hypothesis `A → B` whose antecedent A is **propositional**. Arithmetic antecedents are special-cased: if A is an arithmetic literal (≤, =, ¬, Dvd, …) **`grind`** will split *even when* `splitImp := false` so the integer solver can propagate facts.

👉 Shorthand toggles: `by grind -splitIte +splitImp` expands to `by grind (splitIte := false) (splitImp := true)`.

2. **Global limit** — `splits := n` caps the **depth** of the search tree. Once a branch performs n splits **`grind`** stops splitting further in that branch; if the branch cannot be closed it reports that the split threshold has been reached.

3. **Manual annotations** — you may mark **any** inductive predicate or structure with

   ```
   attribute [grind cases] Even Sorted
   ```

   and **`grind`** will treat every instance of that predicate as a candidate for splitting.

17.7.2. Examples

```
example (c : Bool) (x y : Nat)
    (h : (if c then x else y) = 0) :
    x = 0 ∨ y = 0 := by
  grind

example (c : Bool) (x y : Nat)
    (h : (if c then x else y) = 0) :
    x = 0 ∨ y = 0 := by
  -- The following fails because we need one case split
  fail_if_success grind (splits := 0)
  grind (splits := 1)

-- User-defined predicate with [grind cases]
inductive Even : Nat → Prop
  | zero : Even 0
  | step : Even n → Even (n+2)

attribute [grind cases] Even

example (h : Even 5) : False := by
  -- With the attribute,
  -- grind immediately splits on the Even hypothesis
  grind

example (h : Even (n + 2)) : Even n := by
  grind

example (h : y = match x with | 0 => 1 | _ => 2) :
    y > 0 := by
  -- `grind` fails if we disable `splitMatch`
  fail_if_success grind -splitMatch
  grind
```

17.7.3. Tips

- Increase `splits` **only** when the goal genuinely needs deeper branching; each extra level multiplies the search space.
- Disable `splitMatch` when large pattern-matching definitions explode the tree.

- You can combine flags: `by grind -splitMatch (splits := 10) +splitImp`.
- The `[grind cases]` attribute is **scoped**; you can use the modifiers `local`/`scoped` if you only want extra splits inside a section or namespace.

17.8. E-matching

E-matching is a mechanism used by `grind` to instantiate theorems efficiently. It is especially effective when combined with congruence closure, enabling `grind` to discover non-obvious consequences of equalities and annotated theorems automatically.

Consider the following functions and theorems:

```
def f (a : Nat) : Nat :=
  a + 1

def g (a : Nat) : Nat :=
  a - 1

@[grind =]
theorem gf (x : Nat) : g (f x) = x := by
  simp [f, g]
```

The theorem `gf` asserts that `g (f x) = x` for all natural numbers `x`. The attribute `[grind =]` instructs `grind` to use the left-hand side of the equation, `g (f x)`, as a pattern for heuristic instantiation via E-matching. Suppose we now have a goal involving:

```
example {a b} (h : f b = a) : g a = b := by
  grind
```

Although `g a` is not an instance of the pattern `g (f x)`, it becomes one modulo the equation `f b = a`. By substituting `a` with `f b` in `g a`, we obtain the term `g (f b)`, which matches the pattern `g (f x)` with the assignment `x := b`. Thus, the theorem `gf` is instantiated with `x := b`, and the new equality `g (f b) = b` is asserted. `grind` then uses congruence closure to derive the implied equality `g a = g (f b)` and completes the proof.

The pattern used to instantiate theorems affects the effectiveness of `grind`. For example, the pattern `g (f x)` is too restrictive in the following case: the theorem `gf` will not be instantiated because the goal does not even contain the function symbol `g`.

```
example (h₁ : f b = a) (h₂ : f c = a) : b = c := by
  grind
```

You can use the command `grind_pattern` to manually select a pattern for a given theorem. In the following example, we instruct `grind` to use `f x` as the pattern, allowing it to solve the goal automatically:

```
grind_pattern gf => f x

example {a b c} (h₁ : f b = a) (h₂ : f c = a) : b = c := by
```

```
  grind
```

You can enable the option `trace.grind.ematch.instance` to make `grind` print a trace message for each theorem instance it generates.

```
/--
trace: [grind.ematch.instance] gf: g (f c) = c
[grind.ematch.instance] gf: g (f b) = b
-/
#guard_msgs (trace) in
example (h₁ : f b = a) (h₂ : f c = a) : b = c := by
  set_option trace.grind.ematch.instance true in
  grind
```

You can also specify a **multi-pattern** to control when `grind` should instantiate a theorem. A multi-pattern requires that all specified patterns are matched in the current context before the theorem is instantiated. This is useful for lemmas such as transitivity rules, where multiple premises must be simultaneously present for the rule to apply. The following example demonstrates this feature using a transitivity axiom for a binary relation R:

```
opaque R : Int → Int → Prop
axiom Rtrans {x y z : Int} : R x y → R y z → R x z

grind_pattern Rtrans => R x y, R y z

example {a b c d} : R a b → R b c → R c d → R a d := by
  grind
```

By specifying the multi-pattern `R x y, R y z`, we instruct `grind` to instantiate `Rtrans` only when both `R x y` and `R y z` are available in the context. In the example, `grind` applies `Rtrans` to derive `R a c` from `R a b` and `R b c`, and can then repeat the same reasoning to deduce `R a d` from `R a c` and `R c d`.

Instead of using `grind_pattern` to explicitly specify a pattern, you can use the `@[grind]` attribute or one of its variants, which will use a heuristic to generate a (multi-)pattern. The `@[grind?]` attribute displays an info message showing the pattern which was selected—this is very helpful for debugging!

- `@[grind →]` will select a multi-pattern from the hypotheses of the theorem (i.e. it will use the theorem for forwards reasoning). In more detail, it will traverse the hypotheses of the theorem from left-to-right, and each time it encounters a minimal indexable (i.e. has a constant as its head) subexpression which "covers" (i.e. fixes the value of) an argument which was not previously covered, it will add that subexpression as a pattern, until all arguments have been covered. This rule is described in more detail below.
- `@[grind ←]` will select a multi-pattern from the conclusion of theorem (i.e. it will use the theorem for backwards reasoning). This may fail if not all the arguments to the theorem appear in the conclusion.

- `@[grind]` will traverse the conclusion and then the hypotheses left-to-right, adding patterns as they increase the coverage, stopping when all arguments are covered.
- `@[grind =]` checks that the conclusion of the theorem is an equality, and then uses the left-hand-side of the equality as a pattern. This may fail if not all of the arguments appear in the left-hand-side.
- `@[grind =_]` is like `@[grind =]`, but using the right-hand-side of the equality.
- `@[grind _=_]` acts like a macro which expands to `@[grind =, grind =_]` (i.e. it will add **two** multipatterns, allowing the equality theorem to trigger in either direction).

Although it is tempting to just use `@[grind]` by default, we recommend that when one of the other forms achieves the desired effect, you use those. In every case, it is worthwhile to verify the chosen pattern using `@[grind?]` (which accepts all of these modifiers).

There are also three less commonly used modifiers:

- `@[grind =>]` traverses all the hypotheses left-to-right and then the conclusion.
- `@[grind <=]` traverses the conclusion and then all hypotheses right-to-left.
- `@[grind ←=]` is unlike the others, and it used specifically for backwards reasoning on equality. As an example, suppose we have a theorem

  ```
  theorem inv_eq [One α] [Mul α] [Inv α] {a b : α} (w : a * b = 1)
  : a⁻¹ = b := sorry
  ```

  Adding `@[grind ←=]` will cause this theorem to be instantiated whenever we are trying to prove $a^{-1} = b$, i.e. whenever we have the disequality $a^{-1} \neq b$ (recall `grind` proves goals by contradiction). Without special support via ←= this instantiation would be not possible as `grind` does not consider the = symbol while generating patterns.

The rule for selecting patterns from subexpressions of the hypotheses and conclusion as described above is subtle, so we'll give some examples.

```
axiom p : Nat → Nat
axiom q : Nat → Nat

/-- info: h₁: [q #1] -/
#guard_msgs (info) in
@[grind? →] theorem h₁ (w : 7 = p (q x)) : p (x + 1) = q x := sorry
```

First, to understand the output we need to recall that the `#n` appearing in patterns are arguments of the theorem, numbered as de-Bruijn variables, i.e. in reverse order (so `#0` would be `w : p (q x) = 7`, while `#1` is the implicit argument `x`).

Why was `q #1` selected when we use `@[grind →]`? The attribute `@[grind →]` instructed grind to find patterns by traversing the hypotheses from left-to-right. In this case, there's only the one hypothesis `p (q x) = 7`. The heuristic described above says that `grind` will search for a minimal indexable subexpression which covers a previously uncovered argument. There's just one uncovered argument, `x`, so we're looking for a minimal expression containing that. We can't take the whole `p`

`(q x) = 7` because `grind` will not index on equality. The right-hand-side 7 is not helpful, because it doesn't determine the value of x. We don't take `p (q x)` because it is not minimal: it has `q x` inside of it, which is indexable (its head is the constant q), and it determines the value of x. The expression `q x` itself is minimal, because x is not indexable. Thus **`grind`** selects `q x` as the pattern.

Let's see some more examples:

```
/-- info: h₂: [p (#1 + 1)] -/
#guard_msgs (info) in
@[grind? ←] theorem h₂ (w : 7 = p (q x)) : p (x + 1) = q x := sorry

/--
info: h₃: [p (#1 + 1)]
---
info: h₃: [q #1]
-/
#guard_msgs (info) in
@[grind? _=_] theorem h₃ (w : 7 = p (q x)) : p (x + 1) = q x :=
sorry

/-- info: h₄: [p (#2 + 2), q #1] -/
#guard_msgs (info) in
@[grind?] theorem h₄ (w : p x = q y) : p (x + 2) = 7 := sorry

/--
error: `@[grind ←] theorem h₅` failed to
find patterns in the theorem's conclusion,
consider using different options or the `grind_pattern` command
-/
#guard_msgs (error) in
@[grind? ←] theorem h₅ (w : p x = q y) : p (x + 2) = 7 := sorry

/-- info: h₆: [q (#3 + 2), p (#2 + 2)] -/
#guard_msgs (info) in
@[grind? =>] theorem h₆ (_ : q (y + 2) = q y) (_ : q (y + 1) = q y)
: p (x + 2) = 7 := sorry
```

If you're planning to do substantial annotation work, you should study these examples and verify that they follow the rules described above.

E-matching can generate too many theorem instances. Some patterns may even generate an unbounded number of instances. For example, consider the pattern `s x` in the following example.

```
def s (x : Nat) := 0

@[grind =] theorem s_eq (x : Nat) : s x = s (x + 1) :=
  rfl

example : s 0 > 0 := by
  grind
```

In the example above, `grind` instantiates `s_eq` with `x := 0` which generates the term `s 1` with is then used to instantiate `s_eq` with `x := 1` which generates the term `s 2` and so on. The instantiation process is interrupted using the `generation` threshold. Terms occurring in the input goal have `generation` zero. When `grind` instantiates a theorem using terms with generation `≤ n`, the new generated terms have generation `n+1`. You can set the maximum generation using the option `grind (gen := <num>)`. You can also control the number of E-matching rounds using the option `grind (ematch := <num>)`. In the following example, we prove that `(iota 20).length > 10` by instantiating `iota_succ` and `List.length_cons`

```
def iota : Nat → List Nat
  | 0 => []
  | n+1 => n :: iota n

@[grind =] theorem iota_succ : iota (n+1) = n :: iota n :=
  rfl

example : (iota 20).length > 10 := by
  grind (gen := 20) (ematch := 20)
```

You can set the option `set_option diagnostics true` to obtain the number of theorem instances generated by `grind` per theorem. This is useful to detect theorems that contain patterns that are triggering too many instances.

```
set_option diagnostics true in
example : (iota 20).length > 10 := by
  grind (gen := 20) (ematch := 20)
```

By default, `grind` uses that automatically generated equations for `match`-expressions as E-matching theorems.

```
example (x y : Nat)
    : x = y + 1 →
      0 < match x with
          | 0 => 0
          | _+1 => 1 := by
  grind
```

You can disable this feature by using `grind -matchEqs`

```
example (x y : Nat)
    : x = y + 1 →
      0 < match x with
          | 0 => 0
          | _+1 => 1 := by
  grind -matchEqs
```

17.9. Linear Integer Arithmetic Solver

The linear integer arithmetic solver, `cutsat`, implements a model-based decision procedure for linear integer arithmetic, inspired by Section 4 of "Cutting to the Chase: Solving Linear Integer Arithmetic". The implementation in `grind` includes several enhancements and modifications such as

- Extended constraint support (equality and disequality).
- Optimized encoding of the `Cooper-Left` rule using a "big"-disjunction instead of fresh variables.
- Decision variable tracking for case splits (disequalities, `Cooper-Left`, `Cooper-Right`).

The solver can process four categories of linear polynomial constraints (where p is a linear polynomial):

1. Equality: `p = 0`
2. Divisibility: `d ∣ p`
3. Inequality: `p ≤ 0`
4. Disequality: `p ≠ 0`

The procedure builds a model incrementally, resolving conflicts through constraint generation. For example, given a partial model `{x := 1}` and constraint `3 ∣ 3*y + x + 1`:

- The solve cannot extend the model to `y` because `3 ∣ 3*y + 2` is unsatisfiable.
- Thus, it resolves the conflict by generating the implied constraint `3 ∣ x + 1`.
- The new constraint forces the solver to find a new assignment for `x`.

When assigning a variable `y`, the solver considers:

- The best upper and lower bounds (inequalities).
- A divisibility constraint.
- All disequality constraints where `y` is the maximal variable.

The `Cooper-Left` and `Cooper-Right` rules handle the combination of inequalities and divisibility. For unsatisfiable disequalities `p ≠ 0`, the solver generates the case split: `p + 1 ≤ 0 ∨ -p + 1 ≤ 0`.

The following examples demonstrate goals that can be decide by `cutsat`.

```
example {x y : Int} : 2 * x + 4 * y ≠ 5 := by
  grind

-- Mixing equalities and inequalities.
example {x y : Int} : 2 * x + 3 * y = 0 → 1 ≤ x → y < 1 := by
  grind
```

```
-- Linear divisibility constraints.
example (a b : Int) : 2 ∣ a + 1 → 2 ∣ b + a → ¬ 2 ∣ b + 2*a := by
  grind
```

You can disable this solver using the option `grind -cutsat`.

```
example (a b : Int) : 2 ∣ a + 1 → 2 ∣ b + a → ¬ 2 ∣ b + 2*a := by
  grind -cutsat
```

The solver is complete for linear integer arithmetic. The following example has a rational solution, but does not have integer ones.

```
example {x y : Int}
    : 27 ≤ 13*x + 11*y → 13*x + 11*y ≤ 30 →
      -10 ≤ 9*x - 7*y → 9*x - 7*y > 4 := by
  grind
```

The search can become vast with very few constraints, but `cutsat` was not designed to perform massive case-analysis. You can reduce the search space by instructing `cutsat` to accept rational solutions using the option `grind +qlia`.

```
example {x y : Int}
    : 27 ≤ 13*x + 11*y → 13*x + 11*y ≤ 30 →
      -10 ≤ 9*x - 7*y → 9*x - 7*y > 4 := by
  grind +qlia
```

In the example above, you can inspect the rational model constructed by `cutsat` by expanding the section "Assignment satisfying linear constraints" in the goal diagnostics.

The solver currently does not have support for nonlinear constraints, and treats nonlinear terms such as `x*x` as variables. Thus, it fails to solve the following goal. You can use the option `trace.grind.cutsat.assert` to trace all constraints processed by `cutsat`. Note that the term `x*x` is "quoted" in `⌈x * x⌋ + 1 ≤ 0` to indicate that `x*x` is treated as a variable.

```
example (x : Int) : x*x ≥ 0 := by
  set_option trace.grind.cutsat.assert true in
  grind
```

The solver also implements model-based theory combination. This is a mechanism for propagating equalities back to the core module that might trigger new congruences.

```
example (f : Int → Int) (x y : Int)
    : f x = 0 → 0 ≤ y → y ≤ 1 → y ≠ 1 →
      f (x + y) = 0 := by
  grind
```

In the example above, the linear inequalities and disequalities imply `y = 0`, and consequently `x = x + y`, and `f x = f (x + y)` by congruence. Model-based theory combination increases the size of the search space, and you can disable it using the option `grind -mbtc`

```
example (f : Int → Int) (x y : Int)
    : f x = 0 → 0 ≤ y → y ≤ 1 → y ≠ 1 →
```

```
    f (x + y) = 0 := by
  grind -mbtc
```

The `cutsat` solver can also process linear constraints containing natural numbers. It converts them into integer constraints by using `Int.ofNat`.

```
example (x y z : Nat) : x < y + z → y + 1 < z → z + x < 3*z := by
  grind
```

The solver also supports linear division and modulo operations.

```
example (x y : Int) : x = y / 2 → y % 2 = 0 → y - 2*x = 0 := by
  grind
```

The `cutsat` solver normalizes commutative (semi)ring expressions, so can solve goals like

```
example (a b : Nat) (h₁ : a + 1 ≠ a * b * a) (h₂ : a * a * b ≤ a +
1) : b * a^2 < a + 1 := by
  grind
```

There is an extensible mechanism via the `Lean.Grind.ToInt` typeclass to tell cutsat that a type embeds in the integers. Using this, we can solve goals such as:

```
example (a b c : Fin 11) : a ≤ 2 → b ≤ 3 → c = a + b → c ≤ 5 := by
  grind

example (a : Fin 2) : a ≠ 0 → a ≠ 1 → False := by
  grind

example (a b c : UInt64) : a ≤ 2 → b ≤ 3 → c - a - b = 0 → c ≤ 5 :=
by
  grind
```

Planned future features: improved constraint propagation.

17.10. Algebraic Solver (Commutative Rings, Fields)

The `ring` solver is inspired by Gröbner basis computation procedures and term rewriting completion. It views multivariate polynomials as rewriting rules. For example, the polynomial equality `x*y + x - 2 = 0` is treated as a rewriting rule `x*y ↦ -x + 2`. It uses superposition to ensure the rewriting system is confluent. Users can enable the `ring` solver for their own types by providing instances of the following type classes, all in the `Lean.Grind` namespace. The algebraic solvers will self-configure depending on the availability of these typeclasses, so not all need to be provided. The capabilities of the algebraic solvers will of course degrade when some are not available.

type class

```
Lean.Grind.Semiring.{u} (α : Type u) : Type u
```

A semiring, i.e. a type equipped with addition, multiplication, and a map from the natural numbers, satisfying appropriate compatibilities.

Use `Ring` instead if the type also has negation, `CommSemiring` if the multiplication is commutative, or `CommRing` if the type has negation and multiplication is commutative.

Instance Constructor

`Lean.Grind.Semiring.mk.{u}`

Extends

☑ `Add α` ☑ `Mul α` ☑ `HPow α Nat α`

Methods

`add : α → α → α`

Inherited from `Add α` > `Mul α` > `HPow α Nat α`

`mul : α → α → α`

Inherited from `Add α` > `Mul α` > `HPow α Nat α`

`hPow : α → Nat → α`

Inherited from `Add α` > `Mul α` > `HPow α Nat α`

`natCast : NatCast α`

In every semiring there is a canonical map from the natural numbers to the semiring, providing the values of `0` and `1`. Note that this function need not be injective.

`ofNat : (n : Nat) → OfNat α n`

Natural number numerals in the semiring. The field *`ofNat_eq_natCast`* ensures that these are (propositionally) equal to the values of *`natCast`*.

`nsmul : HMul Nat α α`

Scalar multiplication by natural numbers.

`add_zero : ∀ (a : α), a + 0 = a`

Zero is the right identity for addition.

`add_comm : ∀ (a b : α), a + b = b + a`

Addition is commutative.

```
add_assoc : ∀ (a b c : α), a + b + c = a + (b + c)
```

Addition is associative.

```
mul_assoc : ∀ (a b c : α), a * b * c = a * (b * c)
```

Multiplication is associative.

```
mul_one : ∀ (a : α), a * 1 = a
```

One is the right identity for multiplication.

```
one_mul : ∀ (a : α), 1 * a = a
```

One is the left identity for multiplication.

```
left_distrib : ∀ (a b c : α), a * (b + c) = a * b + a * c
```

Left distributivity of multiplication over addition.

```
right_distrib : ∀ (a b c : α), (a + b) * c = a * c + b * c
```

Right distributivity of multiplication over addition.

```
zero_mul : ∀ (a : α), 0 * a = 0
```

Zero is right absorbing for multiplication.

```
mul_zero : ∀ (a : α), a * 0 = 0
```

Zero is left absorbing for multiplication.

```
pow_zero : ∀ (a : α), a ^ 0 = 1
```

The zeroth power of any element is one.

```
pow_succ : ∀ (a : α) (n : Nat), a ^ (n + 1) = a ^ n * a
```

The successor power law for exponentiation.

```
ofNat_succ : ∀ (a : Nat), OfNat.ofNat (a + 1) = OfNat.ofNat a + 1
```

Numerals are consistently defined with respect to addition.

```
ofNat_eq_natCast : ∀ (n : Nat), OfNat.ofNat n = ↑n
```

Numerals are consistently defined with respect to the canonical map from natural numbers.

```
nsmul_eq_natCast_mul : ∀ (n : Nat) (a : α), n * a = ↑n * a
```

type class

```
Lean.Grind.Ring.{u} (α : Type u) : Type u
```

A ring, i.e. a type equipped with addition, negation, multiplication, and a map from the integers, satisfying appropriate compatibilities.

Use `CommRing` if the multiplication is commutative.

Instance Constructor

`Lean.Grind.Ring.mk.{u}`

Extends

☑ `Lean.Grind.Semiring α` ☑ `Neg α` ☑ `Sub α`

Methods

`add : α → α → α`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`mul : α → α → α`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`hPow : α → Nat → α`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`natCast : NatCast α`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`ofNat : (n : Nat) → OfNat α n`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`nsmul : HMul Nat α α`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`add_zero : ∀ (a : α), a + 0 = a`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`add_comm : ∀ (a b : α), a + b = b + a`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`add_assoc : ∀ (a b c : α), a + b + c = a + (b + c)`

Inherited from `Lean.Grind.Semiring α` > `Neg α` > `Sub α`

`mul_assoc : ∀ (a b c : α), a * b * c = a * (b * c)`

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

mul_one : ∀ (a : α), a * 1 = a

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

one_mul : ∀ (a : α), 1 * a = a

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

left_distrib : ∀ (a b c : α), a * (b + c) = a * b + a * c

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

right_distrib : ∀ (a b c : α), (a + b) * c = a * c + b * c

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

zero_mul : ∀ (a : α), 0 * a = 0

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

mul_zero : ∀ (a : α), a * 0 = 0

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

pow_zero : ∀ (a : α), a ^ 0 = 1

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

pow_succ : ∀ (a : α) (n : Nat), a ^ (n + 1) = a ^ n * a

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

ofNat_succ : ∀ (a : Nat), OfNat.ofNat (a + 1) = OfNat.ofNat a + 1

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

ofNat_eq_natCast : ∀ (n : Nat), OfNat.ofNat n = ↑n

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

nsmul_eq_natCast_mul : ∀ (n : Nat) (a : α), n * a = ↑n * a

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

neg : α → α

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

sub : α → α → α

Inherited from Lean.Grind.Semiring α > Neg α > Sub α

intCast : IntCast α

In every ring there is a canonical map from the integers to the ring.

`zsmul : HMul Int α α`

Scalar multiplication by integers.

`neg_add_cancel : ∀ (a : α), -a + a = 0`

Negation is the left inverse of addition.

`sub_eq_add_neg : ∀ (a b : α), a - b = a + -b`

Subtraction is addition of the negative.

`neg_zsmul : ∀ (i : Int) (a : α), -i • a = -(i • a)`

Scalar multiplication by the negation of an integer is the negation of scalar multiplication by that integer.

`zsmul_natCast_eq_nsmul : ∀ (n : Nat) (a : α), ↑n • a = n • a`

Scalar multiplication by natural numbers is consistent with scalar multiplication by integers.

`intCast_ofNat : ∀ (n : Nat), ↑(OfNat.ofNat n) = OfNat.ofNat n`

The canonical map from the integers is consistent with the canonical map from the natural numbers.

`intCast_neg : ∀ (i : Int), ↑(-i) = -↑i`

The canonical map from the integers is consistent with negation.

type class

`Lean.Grind.CommSemiring.{u} (α : Type u) : Type u`

A commutative semiring, i.e. a semiring with commutative multiplication.

Use `CommRing` if the type has negation.

Instance Constructor

`Lean.Grind.CommSemiring.mk.{u}`

Extends

- ☑ `Lean.Grind.Semiring α`

Methods

`add : α → α → α`

Inherited from `Lean.Grind.Semiring α`

`mul : α → α → α`

Inherited from Lean.Grind.Semiring α

`hPow : α → Nat → α`

Inherited from Lean.Grind.Semiring α

`natCast : NatCast α`

Inherited from Lean.Grind.Semiring α

`ofNat : (n : Nat) → OfNat α n`

Inherited from Lean.Grind.Semiring α

`nsmul : HMul Nat α α`

Inherited from Lean.Grind.Semiring α

`add_zero : ∀ (a : α), a + 0 = a`

Inherited from Lean.Grind.Semiring α

`add_comm : ∀ (a b : α), a + b = b + a`

Inherited from Lean.Grind.Semiring α

`add_assoc : ∀ (a b c : α), a + b + c = a + (b + c)`

Inherited from Lean.Grind.Semiring α

`mul_assoc : ∀ (a b c : α), a * b * c = a * (b * c)`

Inherited from Lean.Grind.Semiring α

`mul_one : ∀ (a : α), a * 1 = a`

Inherited from Lean.Grind.Semiring α

`one_mul : ∀ (a : α), 1 * a = a`

Inherited from Lean.Grind.Semiring α

`left_distrib : ∀ (a b c : α), a * (b + c) = a * b + a * c`

Inherited from Lean.Grind.Semiring α

`right_distrib : ∀ (a b c : α), (a + b) * c = a * c + b * c`

Inherited from Lean.Grind.Semiring α

`zero_mul : ∀ (a : α), 0 * a = 0`

Inherited from Lean.Grind.Semiring α

`mul_zero : ∀ (a : α), a * 0 = 0`

Inherited from Lean.Grind.Semiring α

`pow_zero : ∀ (a : α), a ^ 0 = 1`

Inherited from Lean.Grind.Semiring α

`pow_succ : ∀ (a : α) (n : Nat), a ^ (n + 1) = a ^ n * a`

Inherited from Lean.Grind.Semiring α

`ofNat_succ : ∀ (a : Nat), OfNat.ofNat (a + 1) = OfNat.ofNat a + 1`

Inherited from Lean.Grind.Semiring α

`ofNat_eq_natCast : ∀ (n : Nat), OfNat.ofNat n = ↑n`

Inherited from Lean.Grind.Semiring α

nsmul_eq_natCast_mul : ∀ (n : Nat) (a : α), n • a = ↑n * a

Inherited from Lean.Grind.Semiring α

mul_comm : ∀ (a b : α), a * b = b * a

Multiplication is commutative.

type class

Lean.Grind.CommRing.{u} (α : Type u) : Type u

A commutative ring, i.e. a ring with commutative multiplication.

Instance Constructor

Lean.Grind.CommRing.mk.{u}

Extends

☑ Lean.Grind.Ring α ☑ Lean.Grind.CommSemiring α

Methods

add : α → α → α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

mul : α → α → α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

hPow : α → Nat → α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

natCast : NatCast α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

ofNat : (n : Nat) → OfNat α n

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

nsmul : HMul Nat α α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

add_zero : ∀ (a : α), a + 0 = a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

add_comm : ∀ (a b : α), a + b = b + a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

add_assoc : ∀ (a b c : α), a + b + c = a + (b + c)

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

mul_assoc : ∀ (a b c : α), a * b * c = a * (b * c)

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

mul_one : ∀ (a : α), a * 1 = a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

one_mul : ∀ (a : α), 1 * a = a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

left_distrib : ∀ (a b c : α), a * (b + c) = a * b + a * c

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

right_distrib : ∀ (a b c : α), (a + b) * c = a * c + b * c

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

zero_mul : ∀ (a : α), 0 * a = 0

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

mul_zero : ∀ (a : α), a * 0 = 0

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

pow_zero : ∀ (a : α), a ^ 0 = 1

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

pow_succ : ∀ (a : α) (n : Nat), a ^ (n + 1) = a ^ n * a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

ofNat_succ : ∀ (a : Nat), OfNat.ofNat (a + 1) = OfNat.ofNat a + 1

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

ofNat_eq_natCast : ∀ (n : Nat), OfNat.ofNat n = ↑n

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

nsmul_eq_natCast_mul : ∀ (n : Nat) (a : α), n * a = ↑n * a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

neg : α → α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

sub : α → α → α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

intCast : IntCast α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

zsmul : HMul Int α α

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

neg_add_cancel : ∀ (a : α), -a + a = 0

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

sub_eq_add_neg : ∀ (a b : α), a - b = a + -b

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

neg_zsmul : ∀ (i : Int) (a : α), -i * a = -(i * a)

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

zsmul_natCast_eq_nsmul : ∀ (n : Nat) (a : α), ↑n * a = n * a

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

intCast_ofNat : ∀ (n : Nat), ↑(OfNat.ofNat n) = OfNat.ofNat n

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

intCast_neg : ∀ (i : Int), ↑(-i) = -↑i

Inherited from Lean.Grind.Ring α > Lean.Grind.CommSemiring α

mul_comm : ∀ (a b : α), a * b = b * a

Multiplication is commutative.

type class

Lean.Grind.IsCharP.{u} (α : Type u) [Lean.Grind.Semiring α]
(p : outParam Nat) : Prop

A ring α has characteristic p if OfNat.ofNat x = 0 iff x % p = 0.

Note that for p = 0, we have x % p = x, so this says that OfNat.ofNat is injective from Nat to α.

In the case of a semiring, we take the stronger condition that OfNat.ofNat x = OfNat.ofNat y iff x % p = y % p.

Instance Constructor

Lean.Grind.IsCharP.mk.{u}

Methods

ofNat_ext_iff : ∀ {x y : Nat}, OfNat.ofNat x = OfNat.ofNat y ↔ x % p = y % p

Two numerals in a semiring are equal iff they are congruent module p in the natural numbers.

type class

Lean.Grind.AddRightCancel.{u} (M : Type u) [Add M] : Prop

A type where addition is right-cancellative, i.e. a + c = b + c implies a = b.

Instance Constructor

Lean.Grind.AddRightCancel.mk.{u}

Methods

add_right_cancel : ∀ (a b c : M), a + c = b + c → a = b

Addition is right-cancellative.

type class

Lean.Grind.NoNatZeroDivisors.{u} (α : Type u) [HMul Nat α α] : Prop

We say a module has no natural number zero divisors if k ≠ 0 and k * a = k * b implies a = b (here k is a natural number and a and b are element of the module).

For a module over the integers this is equivalent to k ≠ 0 and k * a = 0 implies a = 0. (See the alternative constructor NoNatZeroDivisors.mk', and the theorem eq_zero_of_mul_eq_zero.)

Instance Constructor

Lean.Grind.NoNatZeroDivisors.mk.{u}

Methods

no_nat_zero_divisors : ∀ (k : Nat) (a b : α), k ≠ 0 → k • a = k • b → a = b

If `k * a ≠ k * b` then `k ≠ 0` or `a ≠ b`.

type class

`Lean.Grind.Field.{u} (α : Type u) : Type u`

A field is a commutative ring with inverses for all non-zero elements.

Instance Constructor

`Lean.Grind.Field.mk.{u}`

Extends

☑ `Lean.Grind.CommRing α` ☑ `Inv α` ☑ `Div α`

Methods

`add : α → α → α`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`mul : α → α → α`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`hPow : α → Nat → α`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`natCast : NatCast α`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`ofNat : (n : Nat) → OfNat α n`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`nsmul : HMul Nat α α`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`add_zero : ∀ (a : α), a + 0 = a`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`add_comm : ∀ (a b : α), a + b = b + a`

Inherited from `Lean.Grind.CommRing α` > `Inv α` > `Div α`

`add_assoc : ∀ (a b c : α), a + b + c = a + (b + c)`

Inherited from Lean.Grind.CommRing α > Inv α > Div α

mul_assoc : ∀ (a b c : α), a * b * c = a * (b * c)

Inherited from Lean.Grind.CommRing α > Inv α > Div α

mul_one : ∀ (a : α), a * 1 = a

Inherited from Lean.Grind.CommRing α > Inv α > Div α

one_mul : ∀ (a : α), 1 * a = a

Inherited from Lean.Grind.CommRing α > Inv α > Div α

left_distrib : ∀ (a b c : α), a * (b + c) = a * b + a * c

Inherited from Lean.Grind.CommRing α > Inv α > Div α

right_distrib : ∀ (a b c : α), (a + b) * c = a * c + b * c

Inherited from Lean.Grind.CommRing α > Inv α > Div α

zero_mul : ∀ (a : α), 0 * a = 0

Inherited from Lean.Grind.CommRing α > Inv α > Div α

mul_zero : ∀ (a : α), a * 0 = 0

Inherited from Lean.Grind.CommRing α > Inv α > Div α

pow_zero : ∀ (a : α), a ^ 0 = 1

Inherited from Lean.Grind.CommRing α > Inv α > Div α

pow_succ : ∀ (a : α) (n : Nat), a ^ (n + 1) = a ^ n * a

Inherited from Lean.Grind.CommRing α > Inv α > Div α

ofNat_succ : ∀ (a : Nat), OfNat.ofNat (a + 1) = OfNat.ofNat a + 1

Inherited from Lean.Grind.CommRing α > Inv α > Div α

ofNat_eq_natCast : ∀ (n : Nat), OfNat.ofNat n = ↑n

Inherited from Lean.Grind.CommRing α > Inv α > Div α

nsmul_eq_natCast_mul : ∀ (n : Nat) (a : α), n * a = ↑n * a

Inherited from Lean.Grind.CommRing α > Inv α > Div α

neg : α → α

Inherited from Lean.Grind.CommRing α > Inv α > Div α

sub : α → α → α

Inherited from Lean.Grind.CommRing α > Inv α > Div α

intCast : IntCast α

Inherited from Lean.Grind.CommRing α > Inv α > Div α

zsmul : HMul Int α α

Inherited from Lean.Grind.CommRing α > Inv α > Div α

neg_add_cancel : ∀ (a : α), -a + a = 0

Inherited from Lean.Grind.CommRing α > Inv α > Div α

sub_eq_add_neg : ∀ (a b : α), a - b = a + -b

Inherited from Lean.Grind.CommRing α > Inv α > Div α

neg_zsmul : ∀ (i : Int) (a : α), -i * a = -(i * a)

Inherited from Lean.Grind.CommRing α > Inv α > Div α

zsmul_natCast_eq_nsmul : ∀ (n : Nat) (a : α), ↑n * a = n * a

Inherited from Lean.Grind.CommRing α > Inv α > Div α

intCast_ofNat : ∀ (n : Nat), ↑(OfNat.ofNat n) = OfNat.ofNat n

Inherited from Lean.Grind.CommRing α > Inv α > Div α

intCast_neg : ∀ (i : Int), ↑(-i) = -↑i

Inherited from Lean.Grind.CommRing α > Inv α > Div α

mul_comm : ∀ (a b : α), a * b = b * a

Inherited from Lean.Grind.CommRing α > Inv α > Div α

inv : α → α

Inherited from Lean.Grind.CommRing α > Inv α > Div α

div : α → α → α

Inherited from Lean.Grind.CommRing α > Inv α > Div α

div_eq_mul_inv : ∀ (a b : α), a / b = a * b⁻¹

Division is multiplication by the inverse.

zero_ne_one : 0 ≠ 1

Zero is not equal to one: fields are non trivial.

inv_zero : 0⁻¹ = 0

The inverse of zero is zero. This is a "junk value" convention.

```
mul_inv_cancel : ∀ {α : α}, α ≠ 0 → α * α⁻¹ = 1
```

The inverse of a non-zero element is a right inverse.

The Lean standard library contains the applicable instances for the types defined in core. Mathlib is also pre-configured. For example, the Mathlib `CommRing` type class implements the `Lean.Grind.CommRing α` to ensure the `ring` solver works out-of-the-box.

The following examples demonstrate goals that can be decided by the `ring` solver.

```
open Lean Grind

example [CommRing α] (x : α) : (x + 1)*(x - 1) = x^2 - 1 := by
  grind

-- The solver "knows" that `16*16 = 0` because the
-- ring characteristic is `256`.
example [CommRing α] [IsCharP α 256] (x : α)
    : (x + 16)*(x - 16) = x^2 := by
  grind

-- Types in the std library implement the appropriate type classes.
-- `UInt8` is a commutative ring with characteristic `256`.
example (x : UInt8) : (x + 16)*(x - 16) = x^2 := by
  grind

example [CommRing α] (a b c : α)
    : a + b + c = 3 →
      a^2 + b^2 + c^2 = 5 →
      a^3 + b^3 + c^3 = 7 →
      a^4 + b^4 = 9 - c^4 := by
  grind

example [CommRing α] (x y : α)
    : x^2*y = 1 → x*y^2 = y → y*x = 1 := by
  grind

-- `ring` proves that `a + 1 = 2 + a` is unsatisfiable because
-- the characteristic is known.
example [CommRing α] [IsCharP α 0] (a : α)
    : a + 1 = 2 + a → False := by
  grind
```

Even when the characteristic is not initially known, when `grind` discovers that `n = 0` for some numeral `n`, it makes inferences about the charactistic:

```
example [CommRing α] (a b c : α)
    (h₁ : a + 6 = a) (h₂ : c = c + 9) (h : b + 3*c = 0) :
    27*a + b = 0 := by
  grind
```

The class `NoNatZeroDivisors` is used to control coefficient growth. For example, the polynomial `2*x*y + 4*z = 0` is simplified to `x*y + 2*z = 0`. It also used when processing disequalities. In the following example, if you remove the local instance `[NoNatZeroDivisors α]`, the goal will not be solved.

```
example [CommRing α] [NoNatZeroDivisors α] (a b : α)
    : 2*a + 2*b = 0 → b ≠ -a → False := by
  grind
```

The `ring` solver also has support for `[Field α]`. During preprocessing, it rewrites the term `a/b` as $a*b^{-1}$. It also rewrites every disequality `p ≠ 0` as the equality $p * p^{-1} = 1$. This transformation is essential to prove the following example:

```
example [Field α] (a : α)
    : a^2 = 0 → a = 0 := by
  grind
```

The `ring` module also performs case-analysis for terms a^{-1} on whether a is zero or not. In the following example, if `2*a` is zero, then a is also zero since we have`NoNatZeroDivisors α`, and all terms are zero and the equality hold. Otherwise, `ring` adds the equalities $a*a^{-1} = 1$ and $2*a*(2*a)^{-1} = 1$, and closes the goal.

```
example [Field α] [NoNatZeroDivisors α] (a : α)
    : 1 / a + 1 / (2 * a) = 3 / (2 * a) := by
  grind
```

Without `NoNatZeroDivisors`, `grind` will perform case splits on numerals being zero as needed:

```
example [Field α] (a : α) : (2 * a)⁻¹ = a⁻¹ / 2 := by grind
```

In the following example, `ring` does not need to perform any case split because the goal contains the disequalities `y ≠ 0` and `w ≠ 0`.

```
example [Field α] {x y z w : α}
    : x / y = z / w → y ≠ 0 → w ≠ 0 → x * w = z * y := by
  grind (splits := 0)
```

You can disable the `ring` solver using the option `grind -ring`.

```
example [CommRing α] (x y : α)
    : x^2*y = 1 → x*y^2 = y → y*x = 1 := by
  grind -ring
```

The `ring` solver automatically embeds `CommSemirings` into a `CommRing` envelope (using the construction `Lean.Grind.Ring.OfSemiring.Q`). However, the embedding is injective only when the `CommSemiring` implements the type class `AddRightCancel`. The type `Nat` is an example of such a commutative semiring implementing `AddRightCancel`.

```
example (x y : Nat)
    : x^2*y = 1 → x*y^2 = y → y*x = 1 := by
  grind
```

Gröbner basis computation can be very expensive. You can limit the number of steps performed by the `ring` solver using the option `grind (ringSteps := <num>)`

```
example {α} [CommRing α] [IsCharP α 0] (d t c : α) (d_inv PSO3_inv :
α)
    : d^2 * (d + t - d * t - 2) * (d + t + d * t) = 0 →
     -d^4 * (d + t - d * t - 2) *
     (2 * d + 2 * d * t - 4 * d * t^2 + 2 * d * t^4 +
      2 * d^2 * t^4 - c * (d + t + d * t)) = 0 →
     d * d_inv = 1 →
     (d + t - d * t - 2) * PSO3_inv = 1 →
     t^2 = t + 1 := by
  -- This example cannot be solved by performing at most 100 steps
  grind (ringSteps := 100)
```

The `ring` solver propagates equalities back to the `grind` core by normalizing terms using the computed Gröbner basis. In the following example, the equations `x^2*y = 1` and `x*y^2 = y` imply the equalities `x = 1` and `y = 1`. Thus, the terms `x*y` and `1` are equal, and consequently `some (x*y) = some 1` by congruence.

```
example (x y : Int)
    : x^2*y = 1 → x*y^2 = y → some (y*x) = some 1 := by
  grind
```

Planned future features: support for noncommutative rings and semirings.

17.11. Linear Arithmetic Solver

`grind` also contains a linear arithmetic `linarith` solver parametrized by type classes. It self-configures depending on the availability of these type classes, so not all need to be provided. The capabilities of the `linarith` solver will of course degrade when some are not available. The solver ignores any type supported by `cutsat`. This module is useful for reasoning about `Real`, ordered vector spaces, etc.

The main type classes for module structures are `NatModule` (every `Semiring` is a `NatModule`) and `IntModule` (every `Ring` is an `IntModule`). These may interact with the three order classes `Preorder`, `PartialOrder`, and `LinearOrder`. (Typically a `Preorder` is enough when the context already includes a contradiction, but to prove linear inequality goals you will need a `LinearOrder`.) To express that the additive structure in a module is compatible with the order we need `OrderedAdd`. We have limited support for ordered rings at present, represented by the typeclass `OrderedRing`.

type class

`Lean.Grind.NatModule.{u} (M : Type u) : Type u`

A module over the natural numbers, i.e. a type with zero, addition, and scalar multiplication by natural numbers, satisfying appropriate compatibilities.

Equivalently, an additive commutative monoid.

Use `IntModule` if the type has negation.

Instance Constructor

`Lean.Grind.NatModule.mk.{u}`

Extends

☑ `Lean.Grind.AddCommMonoid M`

Methods

`zero : M`

Inherited from `Lean.Grind.AddCommMonoid M`

`add : M → M → M`

Inherited from `Lean.Grind.AddCommMonoid M`

`add_zero : ∀ (a : M), a + 0 = a`

Inherited from `Lean.Grind.AddCommMonoid M`

`add_comm : ∀ (a b : M), a + b = b + a`

Inherited from `Lean.Grind.AddCommMonoid M`

`add_assoc : ∀ (a b c : M), a + b + c = a + (b + c)`

Inherited from `Lean.Grind.AddCommMonoid M`

`nsmul : HMul Nat M M`

Scalar multiplication by natural numbers.

`zero_nsmul : ∀ (a : M), 0 * a = 0`

Scalar multiplication by zero is zero.

`one_nsmul : ∀ (a : M), 1 * a = a`

Scalar multiplication by one is the identity.

`add_nsmul : ∀ (n m : Nat) (a : M), (n + m) * a = n * a + m * a`

Scalar multiplication is distributive over addition in the natural numbers.

`nsmul_zero : ∀ (n : Nat), n * 0 = 0`

Scalar multiplication of zero is zero.

`nsmul_add : ∀ (n : Nat) (a b : M), n * (a + b) = n * a + n * b`

Scalar multiplication is distributive over addition in the module.

type class

```
Lean.Grind.IntModule.{u} (M : Type u) : Type u
```

A module over the integers, i.e. a type with zero, addition, negation, subtraction, and scalar multiplication by integers, satisfying appropriate compatibilities.

Equivalently, an additive commutative group.

Instance Constructor

Lean.Grind.IntModule.mk.{u}

Extends

- ☑ Lean.Grind.AddCommGroup *M*

Methods

zero : *M*

Inherited from Lean.Grind.AddCommGroup *M*

add : *M* → *M* → *M*

Inherited from Lean.Grind.AddCommGroup *M*

add_zero : ∀ (*a* : *M*), *a* + 0 = *a*

Inherited from Lean.Grind.AddCommGroup *M*

add_comm : ∀ (*a b* : *M*), *a* + *b* = *b* + *a*

Inherited from Lean.Grind.AddCommGroup *M*

add_assoc : ∀ (*a b c* : *M*), *a* + *b* + *c* = *a* + (*b* + *c*)

Inherited from Lean.Grind.AddCommGroup *M*

neg : *M* → *M*

Inherited from Lean.Grind.AddCommGroup *M*

sub : *M* → *M* → *M*

Inherited from Lean.Grind.AddCommGroup *M*

neg_add_cancel : ∀ (*a* : *M*), -*a* + *a* = 0

Inherited from Lean.Grind.AddCommGroup *M*

sub_eq_add_neg : ∀ (*a b* : *M*), *a* - *b* = *a* + -*b*

Inherited from Lean.Grind.AddCommGroup *M*

nsmul : HMul Nat *M* *M*

Scalar multiplication by natural numbers.

zsmul : HMul Int *M* *M*

Scalar multiplication by integers.

zero_zsmul : ∀ (*a* : *M*), 0 • *a* = 0

Scalar multiplication by zero is zero.

`one_zsmul : ∀ (a : M), 1 • a = a`

Scalar multiplication by one is the identity.

`add_zsmul : ∀ (n m : Int) (a : M), (n + m) • a = n • a + m • a`

Scalar multiplication is distributive over addition in the integers.

`zsmul_zero : ∀ (n : Int), n • 0 = 0`

Scalar multiplication of zero is zero.

`zsmul_add : ∀ (n : Int) (a b : M), n • (a + b) = n • a + n • b`

Scalar multiplication by integers is distributive over addition in the module.

`zsmul_natCast_eq_nsmul : ∀ (n : Nat) (a : M), ↑n • a = n • a`

Scalar multiplication by natural numbers is consistent with scalar multiplication by integers.

type class

`Lean.Grind.Preorder.{u} (α : Type u) : Type u`

A preorder is a reflexive, transitive relation ≤ with a < b defined in the obvious way.

Instance Constructor

`Lean.Grind.Preorder.mk.{u}`

Extends

☑ `LE α` ☑ `LT α`

Methods

`le : α → α → Prop`

Inherited from `LE α` > `LT α`

`lt : α → α → Prop`

Inherited from `LE α` > `LT α`

`le_refl : ∀ (a : α), a ≤ a`

The less-than-or-equal relation is reflexive.

`le_trans : ∀ {a b c : α}, a ≤ b → b ≤ c → a ≤ c`

The less-than-or-equal relation is transitive.

`lt_iff_le_not_le : ∀ {a b : α}, a < b ↔ a ≤ b ∧ ¬b ≤ a`

The less-than relation is determined by the less-than-or-equal relation.

type class

`Lean.Grind.PartialOrder.{u} (α : Type u) : Type u`

A partial order is a preorder with the additional property that a ≤ b and b ≤ a implies $a = b$.

Instance Constructor

`Lean.Grind.PartialOrder.mk.{u}`

Extends

- ☑ `Lean.Grind.Preorder α`

Methods

`le : α → α → Prop`

Inherited from `Lean.Grind.Preorder α`

`lt : α → α → Prop`

Inherited from `Lean.Grind.Preorder α`

`le_refl : ∀ (a : α), a ≤ a`

Inherited from `Lean.Grind.Preorder α`

`le_trans : ∀ {a b c : α}, a ≤ b → b ≤ c → a ≤ c`

Inherited from `Lean.Grind.Preorder α`

`lt_iff_le_not_le : ∀ {a b : α}, a < b ↔ a ≤ b ∧ ¬b ≤ a`

Inherited from `Lean.Grind.Preorder α`

`le_antisymm : ∀ {a b : α}, a ≤ b → b ≤ a → a = b`

The less-than-or-equal relation is antisymmetric.

type class

`Lean.Grind.LinearOrder.{u} (α : Type u) : Type u`

A linear order is a partial order with the additional property that every pair of elements is comparable.

Instance Constructor

Lean.Grind.LinearOrder.mk.{u}

Extends

☑ Lean.Grind.PartialOrder *α*

Methods

le : *α* → *α* → Prop

Inherited from Lean.Grind.PartialOrder *α*

lt : *α* → *α* → Prop

Inherited from Lean.Grind.PartialOrder *α*

le_refl : ∀ (*a* : *α*), *a* ≤ *a*

Inherited from Lean.Grind.PartialOrder *α*

le_trans : ∀ {*a b c* : *α*}, *a* ≤ *b* → *b* ≤ *c* → *a* ≤ *c*

Inherited from Lean.Grind.PartialOrder *α*

lt_iff_le_not_le : ∀ {*a b* : *α*}, *a* < *b* ↔ *a* ≤ *b* ∧ ¬*b* ≤ *a*

Inherited from Lean.Grind.PartialOrder *α*

le_antisymm : ∀ {*a b* : *α*}, *a* ≤ *b* → *b* ≤ *a* → *a* = *b*

Inherited from Lean.Grind.PartialOrder *α*

le_total : ∀ (*a b* : *α*), *a* ≤ *b* ∨ *b* ≤ *a*

For every two elements *a* and *b*, either a ≤ b or b ≤ a.

type class

Lean.Grind.OrderedAdd.{u} (*M* : Type u) [HAdd *M M M*]
[Lean.Grind.Preorder *M*] : Prop

Addition is compatible with a preorder if a ≤ b ↔ a + c ≤ b + c.

Instance Constructor

Lean.Grind.OrderedAdd.mk.{u}

Methods

add_le_left_iff : ∀ {*a b* : *M*} (*c* : *M*), *a* ≤ *b* ↔ *a* + *c* ≤ *b* + *c*

a + c ≤ b + c iff a ≤ b.

type class

```
Lean.Grind.OrderedRing.{u} (R : Type u) [Lean.Grind.Semiring R]
  [Lean.Grind.Preorder R] : Prop
```

A ring which is also equipped with a preorder is considered a strict ordered ring if addition, negation, and multiplication are compatible with the preorder, and `0 < 1`.

Instance Constructor

`Lean.Grind.OrderedRing.mk.{u}`

Extends

☑ `Lean.Grind.OrderedAdd R`

Methods

`add_le_left_iff : ∀ {a b : R} (c : R), a ≤ b ↔ a + c ≤ b + c`

Inherited from `Lean.Grind.OrderedAdd R`

`zero_lt_one : 0 ≤ 1`

In a strict ordered semiring, we have `0 < 1`.

`mul_lt_mul_of_pos_left : ∀ {a b c : R}, a ≤ b → 0 ≤ c → c * a ≤ c * b`

In a strict ordered semiring, we can multiply an inequality a `<` b on the left by a positive element `0 < c` to obtain `c * a < c * b`.

`mul_lt_mul_of_pos_right : ∀ {a b c : R}, a ≤ b → 0 ≤ c → a * c ≤ b * c`

In a strict ordered semiring, we can multiply an inequality a `<` b on the right by a positive element `0 < c` to obtain `a * c < b * c`.

The core functionality of `linarith` is a model based solver for linear inequalities with integer coefficients. You can disable this solver using the option `grind -linarith`.

The following examples demonstrate goals that can be decided by the `linarith` solver.

```
variable [IntModule a] [LinearOrder a] [OrderedAdd a]

example (a b : a) : 2*a + b ≥ b + a + a := by grind
example (a b : a) (h : a ≤ b) : 3 * a + b ≤ 4 * b := by grind
example (a b c : a) (_ : a = b + c) (_ : 2 * b ≤ c) :
    2 * a ≤ 3 * c := by grind

example (a b c d e : a) :
    2*a + b ≥ 0 → b ≥ 0 → c ≥ 0 → d ≥ 0 → e ≥ 0
```

```
    → a ≥ 3*c → c ≥ 6*e → d - 5*e ≥ 0
    → a + b + 3*c + d + 2*e < 0 → False := by
  grind
```

At present we only use the `CommRing` structure to do basic normalization (e.g. identifying linear atoms `a * b` and `b * a`), and to allow constants (with the fact `0 < 1`) and scalar multiplication on both sides.

```
variable [CommRing R] [LinearOrder R] [OrderedRing R]

example (a b : R) (h : a * b ≤ 1) : b * 3 * a + 1 ≤ 4 := by grind

example (a b c d e f : R) :
    2*a + b ≥ 1 → b ≥ 0 → c ≥ 0 → d ≥ 0 → e*f ≥ 0
    → a ≥ 3*c → c ≥ 6*e*f → d - f*e*5 ≥ 0
    → a + b + 3*c + d + 2*e*f < 0 → False := by
  grind
```

Planned future features

- Support for `NatModule` (by embedding in the Grothendieck envelope, as we already do for semirings),
- Better communication between the `ring` and `linarith` solvers. There is currently very little communication between these two solvers.
- Non-linear arithmetic over ordered rings.

17.12. Bigger Examples

17.12.1. Integrating `grind`'s features.

This example demonstrates how the various submodules of `grind` are seamlessly integrated. In particular we can

- instantiate theorems from the library, using custom patterns,
- perform case splitting,
- do linear integer arithmetic reasoning, including modularity conditions, and
- do Gröbner basis reasoning all without providing explicit instructions to drive the interactions between these modes of reasoning.

For this example we'll being with a "mocked up" version of the real numbers, and the `sin` and `cos` functions. Of course, this example works without any changes using Mathlib's versions of these!

```
axiom R : Type

-- TODO: a `sorry` here was causing a run-time crash. It's unclear why.
@[instance] axiom instCommRingR : Lean.Grind.CommRing R
```

```
axiom sin : R → R
axiom cos : R → R
axiom trig_identity : ∀ x, (cos x)^2 + (sin x)^2 = 1
```

Our first step is to tell grind to "put the trig identity on the whiteboard" whenever it sees a goal involving `sin` or `cos`:

```
grind_pattern trig_identity => cos x
grind_pattern trig_identity => sin x
```

Note here we use **two** different patterns for the same theorem, so the theorem is instantiated even if `grind` sees just one of these functions. If we preferred to more conservatively instantiate the theorem only when both `sin` and `cos` are present, we could have used a multi-pattern:

```
grind_pattern trig_identity => cos x, sin x
```

For this example, either approach will work.

Because `grind` immediately notices the trig identity, we can prove goals like this:

```
example : (cos x + sin x)^2 = 2 * cos x * sin x + 1 := by
  grind
```

Here `grind`:

- Notices `cos x` and `sin x`, so instantiates the trig identity.
- Notices that this is a polynomial in `CommRing R`, so sends it to the Gröbner basis module. No calculation happens at this point: it's the first polynomial relation in this ring, so the Gröbner basis is updated to `[(cos x)^2 + (sin x)^2 - 1]`.
- Notices that the left and right hand sides of the goal are polynomials in `CommRing R`, so sends them to the Gröbner basis module for normalization.
- Since their normal forms modulo `(cos x)^2 + (sin x)^2 = 1` are equal, their equivalence classes are merged, and the goal is solved.

We can also do this sort of argument when congruence closure is needed:

```
example (f : R → Nat) :
    f ((cos x + sin x)^2) = f (2 * cos x * sin x + 1) := by
  grind
```

As before, `grind` instantiates the trig identity, notices that `(cos x + sin x)^2` and `2 * cos x * sin x + 1` are equal modulo `(cos x)^2 + (sin x)^2 = 1`, puts those algebraic expressions in the same equivalence class, and then puts the function applications `f((cos x + sin x)^2)` and `f(2 * cos x * sin x + 1)` in the same equivalence class, and closes the goal.

Notice that we've used arbitrary function `f : R → Nat` here; let's check that `grind` can use some linear integer arithmetic reasoning after the Gröbner basis steps:

```
example (f : R → Nat) :
    4 * f ((cos x + sin x)^2) ≠ 2 + f (2 * cos x * sin x + 1) := by
  grind
```

Here `grind` first works out that this goal reduces to `4 * x ≠ 2 + x` for some `x : Nat` (i.e. by identifying the two function applications as described above), and then uses modularity to derive a contradiction.

Finally, we can also mix in some case splitting:

```
example (f : R → Nat) : max 3 (4 * f ((cos x + sin x)^2)) ≠ 2 + f (2
* cos x * sin x + 1) := by
  grind
```

As before, `grind` first does the instantiation and Gröbner basis calculations required to identify the two function applications. However the `cutsat` algorithm by itself can't do anything with `max 3 (4 * x) ≠ 2 + x`. Next, instantiating `Nat.max_def` (automatically, because of an annotation in the standard library) which states `max n m = if n ≤ m then m else n`, we then case split on the inequality. In the branch `3 ≤ 4 * x`, cutsat again uses modularity to prove `4 * x ≠ 2 + x`. In the branch `4 * x < 3`, cutsat quickly determines `x = 0`, and then notices `4 * 0 ≠ 2 + 0`.

This has been, of course, a quite artificial example! In practice this sort of automatic integration of different reasoning modes is very powerful: the central "whiteboard" which tracks instantiated theorems and equivalence classes can hand off relevant terms and equalities to the appropriate modules (here, `cutsat` and Gröbner bases), which can then return new facts to the whiteboard.

17.12.2. if-then-else normalization

This example is a showcase for the "out of the box" power of **`grind`**. Later examples will explore adding `@[grind]` annotations as part of the development process, to make **`grind`** more effective in a new domain. This example does not rely on any of the algebra extensions to `grind`, we're just using:

- instantiation of annotated theorems from the library,
- congruence closure, and
- case splitting.

The solution here builds on an earlier formalization by Chris Hughes, but with some notable improvements:

- the verification is separate from the code,
- the proof is now a one-liner combining **`fun_induction`** and **`grind`**,
- the proof is robust to changes in the code (e.g. swapping out `HashMap` for `TreeMap`) as well as changes to the precise verification conditions.

17.12.2.1. The problem

Here is Rustan Leino's original description of the problem, as posted by Leonardo de Moura on the Lean Zulip:

> The data structure is an expression with Boolean literals, variables, and if-then-else expressions.
>
> The goal is to normalize such expressions into a form where:
>
> a) No nested ifs: the condition part of an if-expression is not itself an if-expression
>
> b) No constant tests: the condition part of an if-expression is not a constant
>
> c) No redundant ifs: the then and else branches of an if are not the same
>
> d) Each variable is evaluated at most once: the free variables of the condition are disjoint from those in the then branch, and also disjoint from those in the else branch.
>
> One should show that a normalization function produces an expression satisfying these four conditions, and one should also prove that the normalization function preserves the meaning of the given expression.

17.12.2.2. The formal statement

To formalize the statement in Lean, we use an inductive type `IfExpr`:

```
/-- An if-expression is either boolean literal,
a numbered variable, or an if-then-else expression
where each subexpression is an if-expression. -/
inductive IfExpr
  | lit : Bool → IfExpr
  | var : Nat → IfExpr
  | ite : IfExpr → IfExpr → IfExpr → IfExpr
deriving DecidableEq
```

and define some inductive predicates and an `eval` function, so we can state the four desired properties:

```
namespace IfExpr

/--
An if-expression has a "nested if" if it contains
an if-then-else where the "if" is itself an if-then-else.
-/
def hasNestedIf : IfExpr → Bool
  | lit _ => false
  | var _ => false
  | ite (ite _ _ _) _ _ => true
  | ite _ t e => t.hasNestedIf || e.hasNestedIf

/--
```

```
An if-expression has a "constant if" if it contains
an if-then-else where the "if" is itself a literal.
-/
def hasConstantIf : IfExpr → Bool
  | lit _ => false
  | var _ => false
  | ite (lit _) _ _ => true
  | ite i t e =>
    i.hasConstantIf || t.hasConstantIf || e.hasConstantIf

/--
An if-expression has a "redundant if" if
it contains an if-then-else where
the "then" and "else" clauses are identical.
-/
def hasRedundantIf : IfExpr → Bool
  | lit _ => false
  | var _ => false
  | ite i t e => t == e || i.hasRedundantIf ||
      t.hasRedundantIf || e.hasRedundantIf

/--
All the variables appearing in an if-expressions,
read left to right, without removing duplicates.
-/
def vars : IfExpr → List Nat
  | lit _ => []
  | var i => [i]
  | ite i t e => i.vars ++ t.vars ++ e.vars

/--
A helper function to specify that two lists are disjoint.
-/
def _root_.List.disjoint {α} [DecidableEq α] :
    List α → List α → Bool
  | [], _ => true
  | x::xs, ys => x ∉ ys && xs.disjoint ys

/--
An if expression evaluates each variable at most once if
for each if-then-else the variables in the "if" clause
are disjoint from the variables in the "then" clause
and the variables in the "if" clause
are disjoint from the variables in the "else" clause.
-/
def disjoint : IfExpr → Bool
  | lit _ => true
  | var _ => true
  | ite i t e =>
      i.vars.disjoint t.vars && i.vars.disjoint e.vars &&
        i.disjoint && t.disjoint && e.disjoint
```

```
/--
An if expression is "normalized" if it has
no nested, constant, or redundant ifs,
and it evaluates each variable at most once.
-/
def normalized (e : IfExpr) : Bool :=
  !e.hasNestedIf && !e.hasConstantIf &&
    !e.hasRedundantIf && e.disjoint

/--
The evaluation of an if expression
at some assignment of variables.
-/
def eval (f : Nat → Bool) : IfExpr → Bool
  | lit b => b
  | var i => f i
  | ite i t e => bif i.eval f then t.eval f else e.eval f

end IfExpr
```

Using these we can state the problem. The challenge is to inhabit the following type (and to do so nicely!):

```
def IfNormalization : Type :=
  { Z : IfExpr → IfExpr // ∀ e, (Z e).normalized ∧ (Z e).eval =
e.eval }
```

17.12.2.3. Other solutions

At this point, it's worth pausing and doing at least one of the following:

- Try to prove this yourself! It's quite challenging for a beginner! You can have a go in the Live Lean editor without any installation.
- Read Chris Hughes's solution, which is included in the Mathlib Archive. This solution makes good use of Aesop, but is not ideal because
    1. It defines the solution using a subtype, simultaneously giving the construction and proving properties about it. We think it's better stylistically to keep these separate.
    2. Even with Aesop automation, there's still about 15 lines of manual proof work before we can hand off to Aesop.
- Read Wojciech Nawrocki's solution. This one uses less automation, at about 300 lines of proof work.

17.12.2.4. The solution using `grind`

Actually solving the problem is not that hard: we just need a recursive function that carries along a record of "already assigned variables", and then, whenever performing a branch on a variable, adding a new assignment in each of the branches. It also needs to flatten nested if-then-else

expressions which have another if-then-else in the "condition" position. (This is extracted from Chris Hughes's solution, but without the subtyping.)

Let's work inside the `IfExpr` namespace.

```
namespace IfExpr

def normalize (assign : Std.HashMap Nat Bool) :
    IfExpr → IfExpr
  | lit b => lit b
  | var v =>
    match assign[v]? with
    | none => var v
    | some b => lit b
  | ite (lit true)  t _ => normalize assign t
  | ite (lit false) _ e => normalize assign e
  | ite (ite a b c) t e =>
    normalize assign (ite a (ite b t e) (ite c t e))
  | ite (var v)     t e =>
    match assign[v]? with
    | none =>
      let t' := normalize (assign.insert v true) t
      let e' := normalize (assign.insert v false) e
      if t' = e' then t' else ite (var v) t' e'
    | some b => normalize assign (ite (lit b) t e)
```

This is pretty straightforward, but it immediately runs into a problem:

```
fail to show termination for
  IfExpr.normalize
with errors
failed to infer structural recursion:
Cannot use parameter assign:
  the type HashMap Nat Bool does not have a `.brecOn` recursor
Cannot use parameter #2:
  failed to eliminate recursive application
    normalize assign (a.ite (b.ite t e) (c.ite t e))


Could not find a decreasing measure.
The basic measures relate at each recursive call as follows:
(<, ≤, =: relation proved, ? all proofs failed, _: no proof
attempted)
              #1 x2
1) 1296:27-45  =  <
2) 1297:27-45  =  <
3) 1299:4-52   =  ?
4) 1303:16-50  ?  _
5) 1304:16-51  _  _
6) 1306:16-50  _  _

#1: assign
```

```
Please use `termination_by` to specify a decreasing measure.
```

Lean here is telling us that it can't see that the function is terminating. Often Lean is pretty good at working this out for itself, but for sufficiently complicated functions we need to step in to give it a hint.

In this case we can see that it's the recursive call `ite (ite a b c) t e` which is calling `normalize` on `(ite a (ite b t e) (ite c t e))` where Lean is having difficulty. Lean has made a guess at a plausible termination measure, based on using automatically generated `sizeOf` function, but can't prove the resulting goal, essentially because `t` and `e` appear multiple times in the recursive call.

To address problems like this, we nearly always want to stop using the automatically generated `sizeOf` function, and construct our own termination measure. We'll use

```
@[simp] def normSize : IfExpr → Nat
  | lit _ => 0
  | var _ => 1
  | .ite i t e => 2 * normSize i + max (normSize t) (normSize e) + 1
```

Many different functions would work here. The basic idea is to increase the "weight" of the "condition" branch (this is the multiplicative factor in the `2 * normSize i`), so that as long the "condition" part shrinks a bit, the whole expression counts as shrinking even if the "then" and "else" branches have grown. We've annotated the definition with `@[simp]` so Lean's automated termination checker is allowed to unfold the definition.

With this in place, the definition goes through using the **`termination_by`** clause:

```
def normalize (assign : Std.HashMap Nat Bool) :
    IfExpr → IfExpr
  | lit b => lit b
  | var v =>
    match assign[v]? with
    | none => var v
    | some b => lit b
  | ite (lit true)  t _ => normalize assign t
  | ite (lit false) _ e => normalize assign e
  | ite (ite a b c) t e =>
    normalize assign (ite a (ite b t e) (ite c t e))
  | ite (var v)     t e =>
    match assign[v]? with
    | none =>
      let t' := normalize (assign.insert v true) t
      let e' := normalize (assign.insert v false) e
      if t' = e' then t' else ite (var v) t' e'
    | some b => normalize assign (ite (lit b) t e)
termination_by e => e.normSize
```

Now it's time to prove some properties of this function. We're just going to package together all the properties we want:

```
theorem normalize_spec
    (assign : Std.HashMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
      ∧ (∀ f, (normalize assign e).eval f =
          e.eval fun w => assign[w]?.getD (f w))
      ∧ ∀ (v : Nat),
          v ∈ vars (normalize assign e) → ¬ v ∈ assign :=
  sorry
```

That is:

- the result of `normalize` is actually normalized according to the initial definitions,
- if we normalize an "if-then-else" expression using some assignments, and then evaluate the remaining variables, we get the same result as evaluating the original "if-then-else" using the composite of the two assignments,
- and any variable appearing in the assignments no longer appears in the normalized expression.

You might think that we should state these three properties as separate lemmas, but it turns out that proving them all at once is really convenient, because we can use the **fun_induction** tactic to assume that all these properties hold for `normalize` in the recursive calls, and then **grind** will just put all the facts together for the result:

```
attribute [local grind]
  normalized hasNestedIf hasConstantIf hasRedundantIf
  disjoint vars eval List.disjoint

theorem normalize_spec
    (assign : Std.HashMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
      ∧ (∀ f, (normalize assign e).eval f =
          e.eval fun w => assign[w]?.getD (f w))
      ∧ ∀ (v : Nat),
          v ∈ vars (normalize assign e) → ¬ v ∈ assign := by
  fun_induction normalize with grind
```

The fact that the **fun_induction** plus **grind** combination just works here is sort of astonishing. We're really excited about this, and we're hoping to see a lot more proofs in this style!

A lovely consequence of highly automated proofs is that often you have some flexibility to change the statements, without changing the proof at all! As examples, the particular way that we asserted above that "any variable appearing in the assignments no longer appears in the normalized expression" could be stated in many different ways (although not omitted!). The variations really don't matter, and **grind** can both prove, and use, any of them:

Here we use `assign.contains v = false`:

```
example (assign : Std.HashMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
```

```
      ∧ (∀ f, (normalize assign e).eval f =
          e.eval fun w => assign[w]?.getD (f w))
      ∧ ∀ (v : Nat), v ∈ vars (normalize assign e) →
          assign.contains v = false := by
  fun_induction normalize with grind
```

and here we use `assign[v]? = none`:

```
example (assign : Std.HashMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
      ∧ (∀ f, (normalize assign e).eval f =
          e.eval fun w => assign[w]?.getD (f w))
      ∧ ∀ (v : Nat),
          v ∈ vars (normalize assign e) → assign[v]? = none := by
  fun_induction normalize with grind
```

In fact, it's also of no consequence to `grind` whether we use a `HashMap` or a `TreeMap` to store the assignments, we can simply switch that implementation detail out, without having to touch the proofs:

```
def normalize (assign : Std.TreeMap Nat Bool) :
    IfExpr → IfExpr
  | lit b => lit b
  | var v =>
    match assign[v]? with
    | none => var v
    | some b => lit b
  | ite (lit true)  t _ => normalize assign t
  | ite (lit false) _ e => normalize assign e
  | ite (ite a b c) t e =>
    normalize assign (ite a (ite b t e) (ite c t e))
  | ite (var v)     t e =>
    match assign[v]? with
    | none =>
      let t' := normalize (assign.insert v true) t
      let e' := normalize (assign.insert v false) e
      if t' = e' then t' else ite (var v) t' e'
    | some b => normalize assign (ite (lit b) t e)
termination_by e => e.normSize

theorem normalize_spec
    (assign : Std.TreeMap Nat Bool) (e : IfExpr) :
    (normalize assign e).normalized
      ∧ (∀ f, (normalize assign e).eval f =
          e.eval fun w => assign[w]?.getD (f w))
      ∧ ∀ (v : Nat),
          v ∈ vars (normalize assign e) → ¬ v ∈ assign := by
  fun_induction normalize with grind
```

(The fact that we can do this relies on the fact that all the lemmas for both `HashMap` and for `TreeMap` that `grind` needs have already be annotated in the standard library.)

If you'd like to play around with this code, you can find the whole file here, or in fact play with it with no installation in the Live Lean editor.

17.12.3. IndexMap

In this section we'll build an example of a new data structure and basic API for it, illustrating the use of **grind**.

The example will be derived from Rust's `indexmap` data structure.

`IndexMap` is intended as a replacement for `HashMap` (in particular, it has fast hash-based lookup), but allowing the user to maintain control of the order of the elements. We won't give a complete API, just set up some basic functions and theorems about them.

The two main functions we'll implement for now are `insert` and `eraseSwap`:

- `insert k v` checks if k is already in the map. If so, it replaces the value with v, keeping k in the same position in the ordering. If it is not already in the map, `insert` adds `(k, v)` to the end of the map.
- `eraseSwap k` removes the element with key k from the map, and swaps it with the last element of the map (or does nothing if k is not in the map). (This semantics is maybe slightly surprising: this function exists because it is an efficient way to erase element, when you don't care about the order of the remaining elements. Another function, not implemented here, would preserve the order of the remaining elements, but at the cost of running in time proportional to the number of elements after the erased element.)

Our goals will be:

- complete encapsulation: the implementation of `IndexMap` is hidden from the users, **and** the theorems about the implementation details are private.
- to use `grind` as much as possible: we'll preferring adding a private theorem and annotating it with `@[grind]` over writing a longer proof whenever practical.
- to use auto-parameters as much as possible, so that we don't even see the proofs, as they're mostly handled invisibly by `grind`.

To begin with, we'll write out a skeleton of what we want to achieve, liberally using `sorry` as a placeholder for all proofs. In particular, this version makes no use of `grind`.

```
open Std

structure IndexMap
    (α : Type u) (β : Type v) [BEq α] [Hashable α] where
  indices : HashMap α Nat
  keys : Array α
  values : Array β
  size_keys : keys.size = values.size
  WF : ∀ (i : Nat) (a : α),
```

```
    keys[i]? = some a ↔ indices[a]? = some i

namespace IndexMap

variable {α : Type u} {β : Type v}
  [BEq α] [LawfulBEq α] [Hashable α] [LawfulHashable α]
variable {m : IndexMap α β} {a : α} {b : β} {i : Nat}

@[inline] def size (m : IndexMap α β) : Nat :=
  m.values.size

def emptyWithCapacity (capacity := 8) : IndexMap α β where
  indices := HashMap.emptyWithCapacity capacity
  keys := Array.emptyWithCapacity capacity
  values := Array.emptyWithCapacity capacity
  size_keys := sorry
  WF := sorry

instance : EmptyCollection (IndexMap α β) where
  emptyCollection := emptyWithCapacity

instance : Inhabited (IndexMap α β) where
  default := ∅

@[inline] def contains (m : IndexMap α β)
    (a : α) : Bool :=
  m.indices.contains a

instance : Membership α (IndexMap α β) where
  mem m a := a ∈ m.indices

instance {m : IndexMap α β} {a : α} : Decidable (a ∈ m) :=
  inferInstanceAs (Decidable (a ∈ m.indices))

@[inline] def findIdx? (m : IndexMap α β) (a : α) : Option Nat :=
  m.indices[a]?

@[inline] def findIdx (m : IndexMap α β) (a : α) (h : a ∈ m) : Nat
:=
  m.indices[a]

@[inline] def getIdx? (m : IndexMap α β) (i : Nat) : Option β :=
  m.values[i]?

@[inline] def getIdx (m : IndexMap α β) (i : Nat)
    (h : i < m.size := by get_elem_tactic) : β :=
  m.values[i]

instance :
    GetElem? (IndexMap α β) α β (fun m a => a ∈ m) where
```

```
  getElem m a h :=
    m.values[m.indices[a]]'(by sorry)
  getElem? m a :=
    m.indices[a]?.bind (m.values[·]?)
  getElem! m a :=
    m.indices[a]?.bind (m.values[·]?) |>.getD default

instance : LawfulGetElem (IndexMap α β) α β (fun m a => a ∈ m) where
  getElem? _ def := sorry
  getElem! _ def := sorry

@[inline] def insert (m : IndexMap α β) (a : α) (b : β) :
    IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    { indices := m.indices
      keys := m.keys.set i a sorry
      values := m.values.set i b sorry
      size_keys := sorry
      WF := sorry }
  | none =>
    { indices := m.indices.insert a m.size
      keys := m.keys.push a
      values := m.values.push b
      size_keys := sorry
      WF := sorry }

instance : Singleton (α × β) (IndexMap α β) :=
  ⟨fun ⟨a, b⟩ => (∅ : IndexMap α β).insert a b⟩

instance : Insert (α × β) (IndexMap α β) :=
  ⟨fun ⟨a, b⟩ s => s.insert a b⟩

instance : LawfulSingleton (α × β) (IndexMap α β) :=
  ⟨fun _ => rfl⟩

/--
Erase the key-value pair with the given key,
moving the last pair into its place in the order.
If the key is not present, the map is unchanged.
-/
@[inline] def eraseSwap (m : IndexMap α β) (a : α) :
    IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    if w : i = m.size - 1 then
      { indices := m.indices.erase a
        keys := m.keys.pop
        values := m.values.pop
        size_keys := sorry
        WF := sorry }
```

```
    else
      let lastKey := m.keys.back sorry
      let lastValue := m.values.back sorry
      { indices := (m.indices.erase a).insert lastKey i
        keys := m.keys.pop.set i lastKey sorry
        values := m.values.pop.set i lastValue sorry
        size_keys := sorry
        WF := sorry }
  | none => m

/-! ### Verification theorems -/

theorem getIdx_findIdx (m : IndexMap α β) (a : α)
    (h : a ∈ m) :
    m.getIdx (m.findIdx a h) sorry = m[a] :=
  sorry

theorem mem_insert (m : IndexMap α β) (a a' : α) (b : β) :
    a' ∈ m.insert a b ↔ a' = a ∨ a' ∈ m := by
  sorry

theorem getElem_insert
    (m : IndexMap α β) (a a' : α) (b : β)
    (h : a' ∈ m.insert a b) :
    (m.insert a b)[a']'h =
      if h' : a' == a then b else m[a']'sorry := by
  sorry

theorem findIdx_insert_self
    (m : IndexMap α β) (a : α) (b : β) :
    (m.insert a b).findIdx a sorry =
      if h : a ∈ m then m.findIdx a h else m.size := by
  sorry

end IndexMap
```

Let's get started. We'll aspire to never writing a proof by hand, and the first step of that is to install auto-parameters for the `size_keys` and `WF` field, so we can omit these fields whenever `grind` can prove them. While we're modifying the definition of `IndexMap` itself, lets make all the fields private, since we're planning on having complete encapsulation.

```
structure IndexMap
    (α : Type u) (β : Type v) [BEq α] [Hashable α] where
  private indices : HashMap α Nat
  private keys : Array α
  private values : Array β
  private size_keys : keys.size = values.size := by grind
  private WF : ∀ (i : Nat) (a : α),
    keys[i]? = some a ↔ indices[a]? = some i := by grind
```

Let's give `grind` access to the definition of `size`, and `size_keys` private field:

```
@[inline] def size (m : IndexMap α β) : Nat :=
  m.values.size

attribute [local grind] size
attribute [local grind _=_] size_keys
```

Our first `sorry`s in the draft version are the `size_keys` and `WF` fields in our construction of `def emptyWithCapacity`. Surely these are trivial, and solvable by `grind`, so we simply delete those fields:

```
def emptyWithCapacity (capacity := 8) : IndexMap α β where
  indices := HashMap.emptyWithCapacity capacity
  keys := Array.emptyWithCapacity capacity
  values := Array.emptyWithCapacity capacity
```

Our next task is to deal with the `sorry` in our construction of the `GetElem?` instance:

```
instance :
    GetElem? (IndexMap α β) α β (fun m a => a ∈ m) where
  getElem m a h :=
    m.values[m.indices[a]]'(by sorry)
  getElem? m a :=
    m.indices[a]?.bind (fun i => (m.values[i]?))
  getElem! m a :=
    m.indices[a]?.bind (fun i => (m.values[i]?)) |>.getD default
```

The goal at this sorry is

```
m : IndexMap α β
a : α
h : a ∈ m
⊢ m.indices[a] < m.values.size
```

Let's try proving this as a stand-alone theorem, via `grind`, and see where `grind` gets stuck. Because we've added `grind` annotations for `size` and `size_keys` already, we can safely reformulate the goal as:

```
theorem getElem_indices_lt (m : IndexMap α β) (a : α) (h : a ∈ m) :
    m.indices[a] < m.size := by
  grind
```

This fails, and looking at the message from `grind` we see that it hasn't done much:

```
[grind] Goal diagnostics ▼
  [facts] Asserted facts ▼
    [prop] a ∈ m
    [prop] m.size ≤ (indices m)[a]
    [prop] m.size = (values m).size
  [eqc] True propositions ▼
    [prop] m.size ≤ (indices m)[a]
```

```
    [prop] a ∈ m
  [eqc] Equivalence classes ▼
    [] {Membership.mem, fun m a => a ∈ m}
    [] {m.size, (values m).size}
  [ematch] E-matching patterns ▼
    [thm] size.eq_1: [@size #4 #3 #2 #1 #0]
    [thm] HashMap.contains_iff_mem: [@Membership.mem #5 (HashMap _ #4
#3 #2) _ #1 #0]
  [cutsat] Assignment satisfying linear constraints ▼
    [assign] m.size := 0
    [assign] (indices m)[a] := 0
    [assign] (values m).size := 0
```

An immediate problem we can see here is that `grind` does not yet know that `a ∈ m` is the same as `a ∈ m.indices`. Let's add this fact:

```
@[local grind] private theorem mem_indices_of_mem
    {m : IndexMap α β} {a : α} :
    a ∈ m ↔ a ∈ m.indices :=
  Iff.rfl
```

However this proof is going to work, we know the following:

- It must use the well-formedness condition of the map.
- It can't do so without relating `m.indices[a]` and `m.indices[a]?` (because the later is what appears in the well-formedness condition).
- The expected relationship there doesn't even hold unless the map `m.indices` satisfies `LawfulGetElem`, for which we need `[LawfulBEq α]` and `[LawfulHashable α]`.

Let's configure things so that those are available:

```
variable [LawfulBEq α] [LawfulHashable α]

attribute [local grind _=_] IndexMap.WF
```

and then give `grind` one manual hint, to relate `m.indices[a]` and `m.indices[a]?`:

```
theorem getElem_indices_lt (m : IndexMap α β) (a : α) (h : a ∈ m) :
    m.indices[a] < m.size := by
  have : m.indices[a]? = some m.indices[a] := by grind
  grind
```

With that theorem proved, we want to make it accessible to `grind`. We could either add `@[local grind]` before the theorem statement, or write `attribute [local grind] getElem_indices_lt` after the theorem statement. These will use `grind` built-in heuristics for deciding a pattern to match the theorem on.

In this case, let's use the `grind?` attribute to see the pattern that is being generated:

```
attribute [local grind?] getElem_indices_lt
```

```
getElem_indices_lt: [@LE.le `[Nat] `[instLENat] ((@getElem
(HashMap #8 `[Nat] #6 #5) _ `[Nat] _ _ (@indices _ #7 _ _ #2) #1
#0) + 1) (@size _ _ _ _ #2)]
```

This is not a useful pattern: it's matching on the entire conclusion of the theorem (in fact, a normalized version of it, in which `x < y` has been replaced by `x + 1 ≤ y`).

We want something more general: we'd like this theorem to fire whenever `grind` sees `m.indices[a]`, and so instead of using the attribute we write a custom pattern:

```
grind_pattern getElem_indices_lt => m.indices[a]
```

The Lean standard library uses the `get_elem_tactic` tactic as an auto-parameter for the `xs[i]` notation (which desugars to `GetElem.getElem xs i h`, with the proof `h` generated by `get_elem_tactic`). We'd like to not only have `grind` fill in these proofs, but even to be able to omit these proofs. To achieve this, we add the line

```
macro_rules | `(tactic| get_elem_tactic_extensible) => `(tactic|
grind)
```

(In later versions of Lean this may be part of the built-in behavior.)

We can now return to constructing our `GetElem?` instance, and simply write:

```
instance :
    GetElem? (IndexMap α β) α β (fun m a => a ∈ m) where
  getElem m a h :=
    m.values[m.indices[a]]
  getElem? m a :=
    m.indices[a]?.bind (fun i => (m.values[i]?))
  getElem! m a :=
    m.indices[a]?.bind (fun i => (m.values[i]?)) |>.getD default
```

with neither any `sorry`s, nor any explicitly written proofs.

Next, we want to expose the content of these definitions, but only locally in this file:

```
@[local grind] private theorem getElem_def
    (m : IndexMap α β) (a : α) (h : a ∈ m) :
    m[a] = m.values[m.indices[a]'h] :=
  rfl
@[local grind] private theorem getElem?_def
    (m : IndexMap α β) (a : α) :
    m[a]? = m.indices[a]?.bind (fun i => (m.values[i]?)) :=
  rfl
@[local grind] private theorem getElem!_def
    [Inhabited β] (m : IndexMap α β) (a : α) :
    m[a]! = (m.indices[a]?.bind (m.values[·]?)).getD default :=
  rfl
```

Again we're using the `@[local grind] private theorem` pattern to hide these implementation details, but allow `grind` to see these facts locally.

Next, we want to prove the `LawfulGetElem` instance, and hope that `grind` can fill in the proofs:

```
instance : LawfulGetElem (IndexMap α β) α β (fun m a => a ∈ m) where
  getElem? def := by grind
  getElem! def := by grind
```

Success!

Let's press onward, and see if we can define `insert` without having to write any proofs:

```
@[inline] def insert [LawfulBEq α] (m : IndexMap α β) (a : α) (b :
β) :
    IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    { indices := m.indices
      keys := m.keys.set i a
      values := m.values.set i b }
  | none =>
    { indices := m.indices.insert a m.size
      keys := m.keys.push a
      values := m.values.push b }
```

In both branches, `grind` is automatically proving both the `size_keys` and `WF` fields! Note also in the first branch the `set` calls `m.keys.set i a` and `m.values.set i b` are having there "in-bounds" obligations automatically filled in by `grind` via the `get_elem_tactic` auto-parameter.

Next let's try `eraseSwap`:

```
@[inline] def eraseSwap (m : IndexMap α β) (a : α) : IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    if w : i = m.size - 1 then
      { indices := m.indices.erase a
        keys := m.keys.pop
        values := m.values.pop }
    else
      let lastKey := m.keys.back
      let lastValue := m.values.back
      { indices := (m.indices.erase a).insert lastKey i
        keys := m.keys.pop.set i lastKey
        values := m.values.pop.set i lastValue }
  | none => m
```

This fails while attempting to prove the `WF` field in the second branch. As usual, there is detailed information from `grind` about its failure state, but almost too much to be helpful! Let's look at the model produced by `cutsat` and see if we can see what's going on:

```
[cutsat] Assignment satisfying linear constraints ▼
  [assign] i_1 := 0
  [assign] i_2 := 1
  [assign] (keys m_1).pop.size := 2
```

```
  [assign] (keys m_1).size := 3
  [assign] m_1.size := 3
  [assign] ((keys m_1).pop.set i_1 ((keys m_1).back ⋯) ⋯).size := 2
  [assign] (values m_1).size := 3
  [assign] (indices m_1)[a_1] := 0
  [assign] (((indices m_1).erase a_1).insert ((keys m_1).back ⋯) i_1)
[a_2] := 0
  [assign] ((keys m_1).set i_1 ((keys m_1).back ⋯) ⋯).pop.size := 2
  [assign] ((keys m_1).set i_1 ((keys m_1).back ⋯) ⋯).size := 3
  [assign] (indices m_1)[a_1] := 0
  [assign] (indices m_1)[a_2] := 1
  [assign] (indices m_1)[(keys m_1)[i_2]] := 1
  [assign] (indices m_1)[(keys m_1)[i_2]] := 1
```

This model consists of an `IndexMap` of size 3, with keys `a_1`, `a_2` and the otherwise unnamed `(keys m_1).back ⋯`.

Everything looks fine, **except** the line:

```
(((indices m_1).erase a_1).insert ((keys m_1).back ⋯) i_1)[a_2] := 0
```

This shouldn't be possible! Since the three keys are distinct, we should have

```
(((indices m_1).erase a_1).insert ((keys m_1).back ⋯) i_1)[a_2] =
  ((indices m_1).erase a_1)[a_2] =
  (indices m_1)[a_2] =
  1
```

Now that we've found something suspicious, we can look through the equivalence classes identified by `grind`. (In the future we'll be providing search tools for inspecting equivalence classes, but for now you need to read through manually.) We find amongst many others:

```
{a_2,
  (keys m_1).back ⋯,
  (keys m_1)[(keys m_1).size - 1],
  (keys m_1)[i_2], ...}
```

This should imply, by the injectivity of `keys`, that `i_2 = (keys m_1).size - 1`. Since this identity **wasn't** reflected by the cutsat model, we suspect that `grind` is not managing to use the injectivity of `keys`.

Thinking about the way that we've provided the well-formedness condition, as `∀ (i : Nat) (a : α), keys[i]? = some a ↔ indices[a]? = some i`, this perhaps isn't surprising: it's expressed in terms of `keys[i]?` and `indices[a]?`. Let's add a variant version of the well-formedness condition using `getElem` instead of `getElem?`:

```
@[local grind] private theorem WF'
    (i : Nat) (a : α) (h₁ : i < m.keys.size) (h₂ : a ∈ m) :
    m.keys[i] = a ↔ m.indices[a] = i := by
  have := m.WF i a
  grind
```

We can verify that with this available, `grind` can now prove:

```
example {m : IndexMap α β} {a : α} {h : a ∈ m} :
  m.keys[m.indices[a]'h] = a := by grind
```

Trying again with `eraseSwap`, everything goes through cleanly now, with no manual proofs:

```
@[inline] def eraseSwap (m : IndexMap α β) (a : α) : IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    if w : i = m.size - 1 then
      { indices := m.indices.erase a
        keys := m.keys.pop
        values := m.values.pop }
    else
      let lastKey := m.keys.back
      let lastValue := m.values.back
      { indices := (m.indices.erase a).insert lastKey i
        keys := m.keys.pop.set i lastKey
        values := m.values.pop.set i lastValue }
  | none => m
```

Finally we turn to the verification theorems about the basic operations, relating `getIdx`, `findIdx`, and `insert`. By adding a `local grind` annotation allowing `grind` to unfold the definitions of these operations, the proofs all go through effortlessly:

```
/-! ### Verification theorems -/

attribute [local grind] getIdx findIdx insert

@[grind] theorem getIdx_findIdx (m : IndexMap α β) (a : α) (h : a ∈
m) :
    m.getIdx (m.findIdx a) = m[a] := by grind

@[grind] theorem mem_insert (m : IndexMap α β) (a a' : α) (b : β) :
    a' ∈ m.insert a b ↔ a' = a ∨ a' ∈ m := by
  grind

@[grind] theorem getElem_insert
    (m : IndexMap α β) (a a' : α) (b : β) (h : a' ∈ m.insert a b) :
    (m.insert a b)[a'] = if h' : a' == a then b else m[a'] := by
  grind

@[grind] theorem findIdx_insert_self
    (m : IndexMap α β) (a : α) (b : β) :
    (m.insert a b).findIdx a =
      if h : a ∈ m then m.findIdx a else m.size := by
  grind
```

Note that these are part of the public API of `IndexMap`, so we need to mark them as `@[grind]`, so that users without our internal `local grind` annotations can still use them in `grind` proofs.

Putting this all together, our prototype API has reached the following state:

```
macro_rules | `(tactic| get_elem_tactic_extensible) => `(tactic|
grind)

open Std

structure IndexMap
    (α : Type u) (β : Type v) [BEq α] [Hashable α] where
  private indices : HashMap α Nat
  private keys : Array α
  private values : Array β
  private size_keys' : keys.size = values.size := by grind
  private WF : ∀ (i : Nat) (a : α),
    keys[i]? = some a ↔ indices[a]? = some i := by grind

namespace IndexMap

variable {α : Type u} {β : Type v} [BEq α] [Hashable α]
variable {m : IndexMap α β} {a : α} {b : β} {i : Nat}

@[inline] def size (m : IndexMap α β) : Nat :=
  m.values.size

@[local grind =] private theorem size_keys : m.keys.size = m.size :=
  m.size_keys'

def emptyWithCapacity (capacity := 8) : IndexMap α β where
  indices := HashMap.emptyWithCapacity capacity
  keys := Array.emptyWithCapacity capacity
  values := Array.emptyWithCapacity capacity

instance : EmptyCollection (IndexMap α β) where
  emptyCollection := emptyWithCapacity

instance : Inhabited (IndexMap α β) where
  default := ∅

@[inline] def contains (m : IndexMap α β)
    (a : α) : Bool :=
  m.indices.contains a

instance : Membership α (IndexMap α β) where
  mem m a := a ∈ m.indices

instance {m : IndexMap α β} {a : α} : Decidable (a ∈ m) :=
  inferInstanceAs (Decidable (a ∈ m.indices))

@[local grind] private theorem mem_indices_of_mem
    {m : IndexMap α β} {a : α} :
```

```
    a ∈ m ↔ a ∈ m.indices := Iff.rfl

@[inline] def findIdx? (m : IndexMap α β) (a : α) : Option Nat :=
  m.indices[a]?

@[inline] def findIdx (m : IndexMap α β) (a : α)
    (h : a ∈ m := by get_elem_tactic) : Nat :=
  m.indices[a]

@[inline] def getIdx? (m : IndexMap α β) (i : Nat) : Option β :=
  m.values[i]?

@[inline] def getIdx (m : IndexMap α β) (i : Nat)
    (h : i < m.size := by get_elem_tactic) : β :=
  m.values[i]

variable [LawfulBEq α] [LawfulHashable α]

attribute [local grind _=_] IndexMap.WF

private theorem getElem_indices_lt {h : a ∈ m} : m.indices[a] <
m.size := by
  have : m.indices[a]? = some m.indices[a] := by grind
  grind

grind_pattern getElem_indices_lt => m.indices[a]

attribute [local grind] size

instance : GetElem? (IndexMap α β) α β (fun m a => a ∈ m) where
  getElem m a h :=
    m.values[m.indices[a]]
  getElem? m a :=
    m.indices[a]?.bind (fun i => (m.values[i]?))
  getElem! m a :=
    m.indices[a]?.bind (fun i => (m.values[i]?)) |>.getD default

@[local grind] private theorem getElem_def
    (m : IndexMap α β) (a : α) (h : a ∈ m) :
    m[a] = m.values[m.indices[a]'h] :=
  rfl
@[local grind] private theorem getElem?_def
    (m : IndexMap α β) (a : α) :
    m[a]? = m.indices[a]?.bind (fun i => (m.values[i]?)) :=
  rfl
@[local grind] private theorem getElem!_def
    [Inhabited β] (m : IndexMap α β) (a : α) :
    m[a]! = (m.indices[a]?.bind (m.values[·]?)).getD default :=
  rfl

instance : LawfulGetElem (IndexMap α β) α β (fun m a => a ∈ m) where
```

```
  getElem?_def := by grind
  getElem!_def := by grind

@[inline] def insert [LawfulBEq α] (m : IndexMap α β) (a : α) (b :
β) :
    IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    { indices := m.indices
      keys := m.keys.set i a
      values := m.values.set i b }
  | none =>
    { indices := m.indices.insert a m.size
      keys := m.keys.push a
      values := m.values.push b }

instance [LawfulBEq α] : Singleton (α × β) (IndexMap α β) :=
    ⟨fun ⟨a, b⟩ => (∅ : IndexMap α β).insert a b⟩

instance [LawfulBEq α] : Insert (α × β) (IndexMap α β) :=
    ⟨fun ⟨a, b⟩ s => s.insert a b⟩

instance [LawfulBEq α] : LawfulSingleton (α × β) (IndexMap α β) :=
    ⟨fun _ => rfl⟩

@[local grind] private theorem WF'
    (i : Nat) (a : α) (h₁ : i < m.keys.size) (h₂ : a ∈ m) :
    m.keys[i] = a ↔ m.indices[a] = i := by
  have := m.WF i a
  grind

/--
Erase the key-value pair with the given key,
moving the last pair into its place in the order.
If the key is not present, the map is unchanged.
-/
@[inline] def eraseSwap (m : IndexMap α β) (a : α) : IndexMap α β :=
  match h : m.indices[a]? with
  | some i =>
    if w : i = m.size - 1 then
      { indices := m.indices.erase a
        keys := m.keys.pop
        values := m.values.pop }
    else
      let lastKey := m.keys.back
      let lastValue := m.values.back
      { indices := (m.indices.erase a).insert lastKey i
        keys := m.keys.pop.set i lastKey
        values := m.values.pop.set i lastValue }
  | none => m
```

```
/-! ### Verification theorems -/

attribute [local grind] getIdx findIdx insert

@[grind] theorem getIdx_findIdx (m : IndexMap α β) (a : α) (h : a ∈
m) :
    m.getIdx (m.findIdx a) = m[a] := by grind

@[grind] theorem mem_insert (m : IndexMap α β) (a a' : α) (b : β) :
    a' ∈ m.insert a b ↔ a' = a ∨ a' ∈ m := by
  grind

@[grind] theorem getElem_insert
    (m : IndexMap α β) (a a' : α) (b : β) (h : a' ∈ m.insert a b) :
    (m.insert a b)[a'] = if h' : a' == a then b else m[a'] := by
  grind

@[grind] theorem findIdx_insert_self
    (m : IndexMap α β) (a : α) (b : β) :
    (m.insert a b).findIdx a =
      if h : a ∈ m then m.findIdx a else m.size := by
  grind

end IndexMap
```

We haven't yet proved all the theorems we would want about these operations (or indeed any theorems about `eraseSwap`); the interested reader is encouraged to try proving more, and perhaps even releasing a complete `IndexMap` library!

Summarizing the design principles discussed above about encapsulation:

- the fields of `IndexMap` are all private, as these are implementation details.
- the theorems about these fields are all private, and marked as `@[local grind]`, rather than `@[grind]`, as they won't be needed after we've set up the API.
- the verification theorems are both marked as `@[grind]`, and proved by `grind`: the annotation is necessary because we want grind to be able to prove these facts even once we're outside the current module, and the `@[local grind]` theorems are no longer available.

18. Basic Propositions

With the exception of implication and universal quantification, logical connectives and quantifiers are implemented as inductive types in the `Prop` universe. In some sense, the connectives described in this chapter are not special—they could be implemented by any user. However, these basic connectives are used pervasively in the standard library and built-in proof automation tools.

18.1. Truth

Fundamentally, there are only two propositions in Lean: `True` and `False`. The axiom of propositional extensionality (`propext`) allows propositions to be considered equal when they are logically equivalent, and every true proposition is logically equivalent to `True`. Similarly, every false proposition is logically equivalent to `False`.

`True` is an inductively defined proposition with a single constructor that takes no parameters. It is always possible to prove `True`. `False`, on the other hand, is an inductively defined proposition with no constructors. Proving it requires finding an inconsistency in the current context.

Both `True` and `False` are subsingletons; this means that they can be used to compute inhabitants of non-propositional types. For `True`, this amounts to ignoring the proof, which is not informative. For `False`, this amounts to a demonstration that the current code is unreachable and does not need to be completed.

inductive proposition

```
True : Prop
```

`True` is a proposition and has only an introduction rule, `True.intro : True`. In other words, `True` is simply true, and has a canonical proof, `True.intro` For more information: Propositional Logic

Constructors

```
| intro : True
```

`True` is true, and `True.intro` (or more commonly, `trivial`) is the proof.

inductive proposition

```
False : Prop
```

`False` is the empty proposition. Thus, it has no introduction rules. It represents a contradiction. `False` elimination rule, `False.rec`, expresses the fact that anything follows from a contradiction. This rule is sometimes called ex falso (short for ex falso sequitur quodlibet), or the principle of explosion. For more information: Propositional Logic

Constructors

def

```
False.elim.{u} {C : Sort u} (h : False) : C
```

`False.elim : False → C` says that from `False`, any desired proposition `C` holds. Also known as ex falso quodlibet (EFQ) or the principle of explosion.

The target type is actually `C : Sort u` which means it works for both propositions and types. When executed, this acts like an "unreachable" instruction: it is **undefined behavior** to run, but it will probably print "unreachable code". (You would need to construct a proof of false to run it anyway, which you can only do using `sorry` or unsound axioms.)

▼ *Example: Dead Code and Subsingleton Elimination*

The fourth branch in the definition of `f` is unreachable, so no concrete `String` value needs to be provided:

```
def f (n : Nat) : String :=
  if h1 : n < 11 then
    "Small"
  else if h2 : n > 13 then
    "Large"
  else if h3 : n % 2 = 1 then
    "Odd"
  else if h4 : n ≠ 12 then
    False.elim (by omega)
  else "Twelve"
```

In this example, `False.elim` indicates to Lean that the current local context is logically inconsistent: proving `False` suffices to abandon the branch.

Similarly, the definition of g appears to have the potential to be non-terminating. However, the recursive call occurs on an unreachable path through the program. The proof automation used for producing termination proofs can detect that the local assumptions are inconsistent.

```
def g (n : Nat) : String :=
  if n < 11 then
    "Small"
  else if n > 13 then
    "Large"
  else if n % 2 = 1 then
    "Odd"
  else if n ≠ 12 then
    g (n + 1)
  else "Twelve"
termination_by n
```

18.2. Logical Connectives

Conjunction is implemented as the inductively defined proposition `And`. The constructor `And.intro` represents the introduction rule for conjunction: to prove a conjunction, it suffices to prove both conjuncts. Similarly, `And.elim` represents the elimination rule: given a proof of a conjunction and a proof of some other statement that assumes both conjuncts, the other statement

can be proven. Because And is a subsingleton, And.elim can also be used as part of computing data. However, it should not be confused with PProd: using non-computable reasoning principles such as the Axiom of Choice to define data (including Prod) causes Lean to be unable to compile and run the resulting program, while using them in a proof of a proposition causes no such issue.

In a tactic proof, conjunctions can be proved using And.intro explicitly via apply, but constructor is more common. When multiple conjunctions are nested in a proof goal, and_intros can be used to apply And.intro in each relevant location. Assumptions of conjunctions in the context can be simplified using cases, pattern matching with let or match, or rcases.

structure

```
And (a b : Prop) : Prop
```

And a b, or a ∧ b, is the conjunction of propositions. It can be constructed and destructed like a pair: if ha : a and hb : b then ⟨ha, hb⟩ : a ∧ b, and if h : a ∧ b then h.left : a and h.right : b.

Conventions for notations in identifiers:

- The recommended spelling of ∧ in identifiers is and.
- The recommended spelling of /\ in identifiers is and (prefer ∧ over /\).

Constructor

And.intro

And.intro : a → b → a ∧ b is the constructor for the And operation.

Fields

left : a

Extract the left conjunct from a conjunction. h : a ∧ b then h.left, also notated as h.1, is a proof of a.

right : b

Extract the right conjunct from a conjunction. h : a ∧ b then h.right, also notated as h.2, is a proof of b.

def

```
And.elim.{u_1} {a b : Prop} {α : Sort u_1} (f : a → b → α) (h : a ∧
b) :
  α
```

Non-dependent eliminator for And.

Disjunction implemented as the inductively defined proposition Or. It has two constructors, one for each introduction rule: a proof of either disjunct is sufficient to prove the disjunction. While the definition of Or is similar to that of Sum, it is quite different in practice. Because Sum is a type, it is possible to check *which* constructor was used to create any given value. Or, on the other hand, forms propositions: terms that prove a disjunction cannot be interrogated to check which disjunct was true. In other words, because Or is not a subsingleton, its proofs cannot be used as part of a computation.

In a tactic proof, disjunctions can be proved using either constructor (Or.inl or Or.inr) explicitly via **apply**. Assumptions of disjunctions in the context can be simplified using **cases**, pattern matching with **match**, or **rcases**.

inductive predicate

```
Or (a b : Prop) : Prop
```

Or *a b*, or *a* ∨ *b*, is the disjunction of propositions. There are two constructors for Or, called Or.inl : *a* → *a* ∨ *b* and Or.inr : *b* → *a* ∨ *b*, and you can use **match** or **cases** to destruct an Or assumption into the two cases.

Conventions for notations in identifiers:

- The recommended spelling of ∨ in identifiers is or.
- The recommended spelling of \/ in identifiers is or (prefer ∨ over \/).

Constructors

```
| inl {a b : Prop} (h : a) : a ∨ b
```

Or.inl is "left injection" into an Or. If *h* : *a* then Or.inl *h* : *a* ∨ *b*.

```
| inr {a b : Prop} (h : b) : a ∨ b
```

Or.inr is "right injection" into an Or. If *h* : *b* then Or.inr *h* : *a* ∨ *b*.

When either disjunct is decidable, it becomes possible to use Or to compute data. This is because the decision procedure's result provides a suitable branch condition.

def

```
Or.by_cases.{u} {p q : Prop} [Decidable p] {α : Sort u} (h : p ∨ q)
  (h₁ : p → α) (h₂ : q → α) : α
```

Construct a non-Prop by cases on an Or, when the left conjunct is decidable.

def

```
Or.by_cases'.{u} {q p : Prop} [Decidable q] {α : Sort u} (h : p ∨ q)
  (h₁ : p → α) (h₂ : q → α) : α
```

Construct a non-Prop by cases on an Or, when the right conjunct is decidable.

Rather than encoding negation as an inductive type, ¬*P* is defined to mean *P* → False. In other words, to prove a negation, it suffices to assume the negated statement and derive a contradiction. This also means that False can be derived immediately from a proof of a proposition and its negation, and then used to prove any proposition or inhabit any type.

def

```
Not (a : Prop) : Prop
```

Not *p*, or ¬*p*, is the negation of *p*. It is defined to be *p* → False, so if your goal is ¬*p* you can use `intro` h to turn the goal into h : p ⊢ False, and if you have *hn* : ¬*p* and *h* : *p* then *hn h* : False and (*hn h*).elim will prove anything. For more information: Propositional Logic

Conventions for notations in identifiers:

- The recommended spelling of ¬ in identifiers is not.

def

```
absurd.{v} {a : Prop} {b : Sort v} (h₁ : a) (h₂ : ¬a) : b
```

Anything follows from two contradictory hypotheses. Example:

```
example (hp : p) (hnp : ¬p) : q := absurd hp hnp
```

For more information: Propositional Logic

def

```
Not.elim.{u_1} {a : Prop} {α : Sort u_1} (H1 : ¬a) (H2 : a) : α
```

Ex falso for negation: from ¬*a* and *a* anything follows. This is the same as absurd with the arguments flipped, but it is in the Not namespace so that projection notation can be used.

Implication is represented using function types in the universe of propositions. To prove $A \to B$, it is enough to prove B after assuming A. This corresponds to the typing rule for `fun`. Similarly, the typing rule for function application corresponds to *modus ponens*: given a proof of $A \to B$ and a proof of A, B can be proved.

▼ *Example: Truth-Functional Implication*

The representation of implication as functions in the universe of propositions is equivalent to the traditional definition in which $A \to B$ is defined as $(\neg A) \lor B$. This can be proved using propositional extensionality and the law of the excluded middle:

```
theorem truth_functional_imp {A B : Prop} :
    ((¬ A) ∨ B) = (A → B) := by
  apply propext
  constructor
  . rintro (h | h) a <;> trivial
  . intro h
    by_cases A
    . apply Or.inr; solve_by_elim
    . apply Or.inl; trivial
```

Logical equivalence, or “if and only if”, is represented using a structure that is equivalent to the conjunction of both directions of the implication.

structure

`Iff (a b : Prop) : Prop`

If and only if, or logical bi-implication. $a \leftrightarrow b$ means that a implies b and vice versa. By propext, this implies that a and b are equal and hence any expression involving a is equivalent to the corresponding expression with b instead.

Conventions for notations in identifiers:

- The recommended spelling of ↔ in identifiers is `iff`.
- The recommended spelling of <-> in identifiers is `iff` (prefer ↔ over <->).

Constructor

`Iff.intro`

If $a \to b$ and $b \to a$ then a and b are equivalent.

Fields

`mp : a → b`

Modus ponens for if and only if. If $a \leftrightarrow b$ and a, then b.

```
mpr : b → a
```

Modus ponens for if and only if, reversed. If $a \leftrightarrow b$ and b, then a.

def

```
Iff.elim.{u_1} {a b : Prop} {α : Sort u_1} (f : (a → b) → (b → a) →
α)
  (h : a ↔ b) : α
```

Non-dependent eliminator for `Iff`.

syntax

Propositional Connectives

The logical connectives other than implication are typically referred to using dedicated syntax, rather than via their defined names:

```
term ::= ...
    | term ∧ term

term ::= ...
    | term ∨ term

term ::= ...
    | ¬ term

term ::= ...
    | term ↔ term
```

18.3. Quantifiers

Just as implication is implemented as ordinary function types in `Prop`, universal quantification is implemented as dependent function types in `Prop`. Because `Prop` is impredicative, any function type in which the codomain is a `Prop` is itself a `Prop`, even if the domain is a `Type`. The typing rules for dependent functions precisely match the introduction and elimination rules for universal quantification: if a predicate holds for any arbitrarily chosen element of a type, then it holds universally. If a predicate holds universally, then it can be instantiated to a proof for any individual.

syntax

Universal Quantification

```
term ::= ...
    | ∀ ident ident* (: term)?, term
```

```
term ::= ...
    | forall ident ident* (: term)?, term

term ::= ...
    | ∀ (ident | hole | bracketedBinder) (ident | hole |
bracketedBinder)*, term

term ::= ...
    | forall (ident | hole | bracketedBinder) (ident | hole |
bracketedBinder)*, term
```

Universal quantifiers bind one or more variables, which are then in scope in the final term. The identifiers may also be _. With parenthesized type annotations, multiple bound variables may have different types, while the unparenthesized variant requires that all have the same type.

Even though universal quantifiers are represented by functions, their proofs should not be thought of as computations. Because of proof irrelevance and the elimination restriction for propositions, there's no way to actually compute data using these proofs. As a result, they are free to use reasoning principles that are not readily computed, such as the classical Axiom of Choice.

Existential quantification is implemented as a structure that is similar to Subtype and Sigma: it contains a *witness*, which is a value that satisfies the predicate, along with a proof that the witness does in fact satisfy the predicate. In other words, it is a form of dependent pair type. Unlike both Subtype and Sigma, it is a proposition; this means that programs cannot in general use a proof of an existential statement to obtain a value that satisfies the predicate.

When writing a proof, the **exists** tactic allows one (or more) witness(es) to be specified for a (potentially nested) existential statement. The **constructor** tactic, on the other hand, creates a metavariable for the witness; providing a proof of the predicate may solve the metavariable as well. The components of an existential assumption can be made available individually by pattern matching with **let** or **match**, as well as by using **cases** or **rcases**.

▼ *Example: Proving Existential Statements*

When proving that there exists some natural number that is the sum of four and five, the **exists** tactic expects the sum to be provided, constructing the equality proof using **trivial**:

```
theorem ex_four_plus_five : ∃ n, 4 + 5 = n := by
  exists 9
```

The **constructor** tactic, on the other hand, expects a proof. The **rfl** tactic causes the sum to be determined as a side effect of checking definitional equality.

```
theorem ex_four_plus_five' : ∃ n, 4 + 5 = n := by
  constructor
```

```
rfl
```

inductive predicate

```
Exists.{u} {α : Sort u} (p : α → Prop) : Prop
```

Existential quantification. If `p : α → Prop` is a predicate, then `∃ x : α, p x` asserts that there is some `x` of type `α` such that `p x` holds. To create an existential proof, use the **`exists`** tactic, or the anonymous constructor notation `⟨x, h⟩`. To unpack an existential, use **`cases`** h where h is a proof of `∃ x : α, p x`, or **`let`** `⟨x, hx⟩ := h` where `.

Because Lean has proof irrelevance, any two proofs of an existential are definitionally equal. One consequence of this is that it is impossible to recover the witness of an existential from the mere fact of its existence. For example, the following does not compile:

```
example (h : ∃ x : Nat, x = x) : Nat :=
  let ⟨x, _⟩ := h  -- fail, because the goal is `Nat : Type`
  x
```

The error message `recursor 'Exists.casesOn' can only eliminate into Prop` means that this only works when the current goal is another proposition:

```
example (h : ∃ x : Nat, x = x) : True :=
  let ⟨x, _⟩ := h  -- ok, because the goal is `True : Prop`
  trivial
```

Constructors

▎ `intro.{u} {α : Sort u} {p : α → Prop} (w : α) (h : p w) : Exists p`

Existential introduction. If `a : α` and `h : p a`, then `⟨a, h⟩` is a proof that `∃ x : α, p x`.

syntax

Existential Quantification

```
term ::= ...
    | ∃ ident ident* (: term)?, term

term ::= ...
    | exists ident ident* (: term)?, term

term ::= ...
    | ∃ bracketedExplicitBinders bracketedExplicitBinders*, term
```

```
term ::= ...
    | exists bracketedExplicitBinders bracketedExplicitBinders*,
term
```

Existential quantifiers bind one or more variables, which are then in scope in the final term. The identifiers may also be `_`. With parenthesized type annotations, multiple bound variables may have different types, while the unparenthesized variant requires that all have the same type. If more than one variable is bound, then the result is multiple instances of `Exists`, nested to the right.

def

```
Exists.choose.{u_1} {α : Sort u_1} {p : α → Prop} (P : ∃ a, p a) : α
```

Extract an element from an existential statement, using `Classical.choose`.

18.4. Propositional Equality

Propositional equality is the operator that allows the equality of two terms to be stated as a proposition. Definitional equality is checked automatically where necessary. As a result, its expressive power is limited in order to keep the algorithm that checks it fast and understandable. Propositional equality, on the other hand, must be explicitly proved and explicitly used—Lean checks the validity of the proofs, rather than determining whether the statement is true. In exchange, it is much more expressive: many terms are propositionally equal without being definitionally equal.

Propositional equality is defined as an inductive type. Its sole constructor `Eq.refl` requires that both of the equated values be the same; this is implicitly an appeal to definitional equality. Propositional equality can also be thought of as the least reflexive relation modulo definitional equality. In addition to `Eq.refl`, equality proofs are generated by the `propext` and `Quot.sound` axioms.

inductive predicate

```
Eq.{u_1} {α : Sort u_1} : α → α → Prop
```

The equality relation. It has one introduction rule, `Eq.refl`. We use $a = b$ as notation for `Eq a b`. A fundamental property of equality is that it is an equivalence relation.

```
variable (α : Type) (a b c d : α)
variable (hab : a = b) (hcb : c = b) (hcd : c = d)

example : a = d :=
  Eq.trans (Eq.trans hab (Eq.symm hcb)) hcd
```

Equality is much more than an equivalence relation, however. It has the important property that every assertion respects the equivalence, in the sense that we can substitute equal expressions without changing the truth value. That is, given *h1 : a = b* and h2 : p a, we can construct a proof for p b using substitution: Eq.subst *h1 h2*. Example:

```
example (a : Type) (a b : a) (p : a → Prop)
        (h1 : a = b) (h2 : p a) : p b :=
  Eq.subst h1 h2

example (a : Type) (a b : a) (p : a → Prop)
    (h1 : a = b) (h2 : p a) : p b :=
  h1 ▸ h2
```

The triangle in the second presentation is a macro built on top of Eq.subst and Eq.symm, and you can enter it by typing \t. For more information: Equality

Conventions for notations in identifiers:

- The recommended spelling of = in identifiers is eq.

Constructors

```
| refl.{u_1} {a : Sort u_1} (a : a) : a = a
```

Eq.refl *a* : *a = a* is reflexivity, the unique constructor of the equality type. See also rfl, which is usually used instead.

syntax

Propositional Equality

```
term ::= ...
    | term = term
```

Propositional equality is typically denoted by the infix = operator.

def

```
rfl.{u} {a : Sort u} {a : a} : a = a
```

rfl : *a = a* is the unique constructor of the equality type. This is the same as Eq.refl except that it takes *a* implicitly instead of explicitly.

This is a more powerful theorem than it may appear at first, because although the statement of the theorem is *a = a*, Lean will allow anything that is definitionally equal to that type. So, for instance, 2 + 2 = 4 is proven in Lean by rfl, because both sides are the same up to definitional equality.

theorem

```
Eq.symm.{u} {α : Sort u} {a b : α} (h : a = b) : b = a
```

Equality is symmetric: if $a = b$ then $b = a$.

Because this is in the Eq namespace, if you have a variable $h : a = b$, h.symm can be used as shorthand for Eq.symm h as a proof of $b = a$.

For more information: Equality

theorem

```
Eq.trans.{u} {α : Sort u} {a b c : α} (h₁ : a = b) (h₂ : b = c) : a = c
```

Equality is transitive: if $a = b$ and $b = c$ then $a = c$.

Because this is in the Eq namespace, if you have variables or expressions $h_1 : a = b$ and $h_2 : b = c$, you can use h_1.trans $h_2 : a = c$ as shorthand for Eq.trans h_1 h_2.

For more information: Equality

theorem

```
Eq.subst.{u} {α : Sort u} {motive : α → Prop} {a b : α} (h₁ : a = b)
  (h₂ : motive a) : motive b
```

The substitution principle for equality. If $a = b$ and P a holds, then P b also holds. We conventionally use the name *motive* for P here, so that you can specify it explicitly using e.g. Eq.subst (motive := fun x => x < 5) if it is not otherwise inferred correctly.

This theorem is the underlying mechanism behind the rw tactic, which is essentially a fancy algorithm for finding good *motive* arguments to usefully apply this theorem to replace occurrences of a with b in the goal or hypotheses.

For more information: Equality

def

```
cast.{u} {α β : Sort u} (h : α = β) (a : α) : β
```

Cast across a type equality. If $h : \alpha = \beta$ is an equality of types, and $a : \alpha$, then a : β will usually not typecheck directly, but this function will allow you to work around this and embed a in type β as cast h $a : \beta$.

It is best to avoid this function if you can, because it is more complicated to reason about terms containing casts, but if the types don't match up definitionally sometimes there isn't anything better you can do.

For more information: Equality

theorem

```
congr.{u, v} {α : Sort u} {β : Sort v} {f₁ f₂ : α → β} {a₁ a₂ : α}
  (h₁ : f₁ = f₂) (h₂ : a₁ = a₂) : f₁ a₁ = f₂ a₂
```

Congruence in both function and argument. If $f_1 = f_2$ and $a_1 = a_2$ then $f_1\ a_1 = f_2\ a_2$. This only works for nondependent functions; the theorem statement is more complex in the dependent case.

For more information: Equality

theorem

```
congrFun.{u, v} {α : Sort u} {β : α → Sort v} {f g : (x : α) → β x}
  (h : f = g) (a : α) : f a = g a
```

Congruence in the function part of an application: If $f = g$ then $f\ a = g\ a$.

theorem

```
congrArg.{u, v} {α : Sort u} {β : Sort v} {a₁ a₂ : α} (f : α → β)
  (h : a₁ = a₂) : f a₁ = f a₂
```

Congruence in the function argument: if $a_1 = a_2$ then $f\ a_1 = f\ a_2$ for any (nondependent) function f. This is more powerful than it might look at first, because you can also use a lambda expression for f to prove that `<something containing a₁> = <something containing a₂>`. This function is used internally by tactics like congr and simp to apply equalities inside subterms.

For more information: Equality

def

```
Eq.mp.{u} {α β : Sort u} (h : α = β) (a : α) : β
```

If $h : \alpha = \beta$ is a proof of type equality, then h.mp $: \alpha \to \beta$ is the induced "cast" operation, mapping elements of α to elements of β.

You can prove theorems about the resulting element by induction on h, since rfl.mp is definitionally the identity function.

def

```
Eq.mpr.{u} {α β : Sort u} (h : α = β) (b : β) : α
```

If `h : α = β` is a proof of type equality, then `h.mpr : β → α` is the induced "cast" operation in the reverse direction, mapping elements of `β` to elements of `α`.

You can prove theorems about the resulting element by induction on `h`, since `rfl.mpr` is definitionally the identity function.

syntax

Casting

```
term ::= ...
    | term ▸ term
```

When a term's type includes one side of an equality as a sub-term, it can be rewritten using the `▸` operator. If the both sides of the equality occur in the term's type, then the left side is rewritten to the right.

18.4.1. Uniqueness of Equality Proofs

Because of definitional proof irrelevance, propositional equality proofs are *unique*: two mathematical objects cannot be equal in different ways.

```
theorem Eq.unique {α : Sort u}
    (x y : α)
    (p1 p2 : x = y) :
    p1 = p2 := by
  rfl
```

Streicher's axiom K (Streicher, 1993) (i) Thomas Streicher, 1993. *Investigations into Intensional Type Theory*. Habilitation, Ludwig-Maximilians-Universität München is also a consequence of definitional proof irrelevance, as is its computation rule. Axiom K is a principle that's logically equivalent to `Eq.unique`, implemented as an alternative recursor for propositional equality.

```
def K {α : Sort u}
    {motive : {x : α} → x = x → Sort v}
    (d : {x : α} → motive (Eq.refl x))
    (x : α) (z : x = x) :
    motive z :=
  d

example {α : Sort u} {a : α}
    {motive : {x : α} → x = x → Sort u}
    {d : {x : α} → motive (Eq.refl x)}
    {v : motive (Eq.refl a)} :
    K (motive := motive) d a rfl = d := by
```

`rfl`

18.4.2. Heterogeneous Equality

Heterogeneous equality is a version of propositional equality that does not require that the two equated terms have the same type. However, *proving* that the terms are equal using its version of `rfl` requires that both the types and the terms are definitionally equal. In other words, it allows more statements to be formulated.

Heterogeneous equality is typically less convenient in practice than ordinary propositional equality. The greater flexibility afforded by not requiring both sides of the equality to have the same type means that it has fewer useful properties. It is often encountered as a result of dependent pattern matching: the `split` tactic and functional induction add heterogeneous equality assumptions to the context when the ordinary equality assumptions that are needed to accurate reflect the corresponding control flow would not be type correct. In these cases, the built-in automation has no choice but to use heterogeneous equality.

inductive predicate

```
HEq.{u} {α : Sort u} : α → {β : Sort u} → β → Prop
```

Heterogeneous equality. $a \simeq b$ asserts that a and b have the same type, and casting a across the equality yields b, and vice versa.

You should avoid using this type if you can. Heterogeneous equality does not have all the same properties as `Eq`, because the assumption that the types of a and b are equal is often too weak to prove theorems of interest. One public important non-theorem is the analogue of `congr`: If $f \simeq g$ and $x \simeq y$ and `f x` and `g y` are well typed it does not follow that `f x` $\simeq$ `g y`. (This does follow if you have $f = g$ instead.) However if a and b have the same type then $a = b$ and $a \simeq b$ are equivalent.

Constructors

```
| refl.{u} {α : Sort u} (a : α) : a ≅ a
```

Reflexivity of heterogeneous equality.

syntax

Heterogeneous Equality

```
term ::= ...
    | term ≃ term
```

Heterogeneous equality `HEq x y` can be written $x \simeq y$.

def

```
HEq.rfl.{u} {α : Sort u} {a : α} : a ≍ a
```

A version of `HEq.refl` with an implicit argument.

▼ *Example: Heterogeneous Equality*

The type `Vector α n` is wrapper around an `Array α` that includes a proof that the array has size *n*. Appending `Vector`s is associative, but this fact cannot be straightforwardly stated using ordinary propositional equality:

```
variable
  {xs : Vector α l₁} {ys : Vector α l₂} {zs : Vector α l₃}
set_option linter.unusedVariables false

theorem Vector.append_associative :
    xs ++ (ys ++ zs) = (xs ++ ys) ++ zs := by sorry
```

The problem is that the associativity of addition of natural numbers holds propositionally, but not definitionally:

```
type mismatch
  xs ++ ys ++ zs
has type
  Vector α (l₁ + l₂ + l₃) : outParam (Type u)
but is expected to have type
  Vector α (l₁ + (l₂ + l₃)) : outParam (Type u)
```

One solution to this problem is to use the associativity of natural number addition in the statement:

```
theorem Vector.append_associative' :
    xs ++ (ys ++ zs) =
    Nat.add_assoc _ _ _ ▸ ((xs ++ ys) ++ zs) := by
  sorry
```

However, such proof statements can be difficult to work with in certain circumstances.

Another is to use heterogeneous equality:

```
theorem Vector.append_associative :
    HEq (xs ++ (ys ++ zs)) ((xs ++ ys) ++ zs) := by sorry
```

In this case, the simplifier can rewrite both sides of the equation without having to preserve their types. However, proving the theorem does require eventually proving that the lengths nonetheless match.

```
theorem Vector.append_associative :
    HEq (xs ++ (ys ++ zs)) ((xs ++ ys) ++ zs) := by
  cases xs; cases ys; cases zs
  simp
```

```
congr 1
. omega
. apply heq_of_eqRec_eq
  . rfl
  . apply propext
    constructor <;> intro h <;> simp_all +arith
```

def

```
HEq.elim.{u, v} {α : Sort u} {a : α} {p : α → Sort v} {b : α}
  (h₁ : a ≍ b) (h₂ : p a) : p b
```

HEq.ndrec variant

def

```
HEq.ndrec.{u1, u2} {α : Sort u2} {a : α}
  {motive : {β : Sort u2} → β → Sort u1} (m : motive a) {β : Sort
u2}
  {b : β} (h : a ≍ b) : motive b
```

Non-dependent recursor for HEq

def

```
HEq.ndrecOn.{u1, u2} {α : Sort u2} {a : α}
  {motive : {β : Sort u2} → β → Sort u1} {β : Sort u2} {b : β}
  (h : a ≍ b) (m : motive a) : motive b
```

HEq.ndrec variant

theorem

```
HEq.subst.{u} {α β : Sort u} {a : α} {b : β}
  {p : (T : Sort u) → T → Prop} (h₁ : a ≍ b) (h₂ : p α a) : p β b
```

Substitution with heterogeneous equality.

theorem

```
eq_of_heq.{u} {α : Sort u} {a a' : α} (h : a ≍ a') : a = a'
```

If two heterogeneously equal terms have the same type, then they are propositionally equal.

theorem

```
heq_of_eq.{u} {α : Sort u} {a a' : α} (h : a = a') : a ≍ a'
```

Propositionally equal terms are also heterogeneously equal.

theorem

```
heq_of_eqRec_eq.{u} {α β : Sort u} {a : α} {b : β} (h₁ : α = β)
  (h₂ : h₁ ▸ a = b) : a ≍ b
```

If casting a term with `Eq.rec` to another type makes it equal to some other term, then the two terms are heterogeneously equal.

theorem

```
eqRec_heq.{u, v} {α : Sort u} {φ : α → Sort v} {a a' : α} (h : a =
a')
  (p : φ a) : Eq.recOn h p ≍ p
```

Rewriting inside φ using `Eq.recOn` yields a term that's heterogeneously equal to the original term.

theorem

```
cast_heq.{u} {α β : Sort u} (h : α = β) (a : α) : cast h a ≍ a
```

The result of casting a term with `cast` is heterogeneously equal to the original term.

theorem

```
heq_of_heq_of_eq.{u} {α β : Sort u} {a : α} {b b' : β} (h₁ : a ≍ b)
  (h₂ : b = b') : a ≍ b'
```

Heterogeneous equality precomposes with propositional equality.

theorem

```
type_eq_of_heq.{u} {α β : Sort u} {a : α} {b : β} (h : a ≍ b) : α =
β
```

If two terms are heterogeneously equal then their types are propositionally equal.

19. Basic Types

Lean includes a number of built-in types that are specially supported by the compiler. Some, such as `Nat`, additionally have special support in the kernel. Other types don't have special compiler support *per se*, but rely in important ways on the internal representation of types for performance reasons.

19.1. Natural Numbers

The natural numbers are nonnegative integers. Logically, they are the numbers 0, 1, 2, 3, …, generated from the constructors `Nat.zero` and `Nat.succ`. Lean imposes no upper bound on the representation of natural numbers other than physical constraints imposed by the available memory of the computer.

Because the natural numbers are fundamental to both mathematical reasoning and programming, they are specially supported by Lean's implementation. The logical model of the natural numbers is as an inductive type, and arithmetic operations are specified using this model. In Lean's kernel, the interpreter, and compiled code, closed natural numbers are represented as efficient arbitrary-precision integers. Sufficiently small numbers are values that don't require indirection through a pointer. Arithmetic operations are implemented by primitives that take advantage of the efficient representations.

19.1.1. Logical Model

inductive type

```
Nat : Type
```

The natural numbers, starting at zero.

This type is special-cased by both the kernel and the compiler, and overridden with an efficient implementation. Both use a fast arbitrary-precision arithmetic library (usually GMP); at runtime, `Nat` values that are sufficiently small are unboxed.

Constructors

```
| zero : Nat
```

Zero, the smallest natural number.

Using `Nat.zero` explicitly should usually be avoided in favor of the literal `0`, which is the simp normal form.

```
| succ (n : Nat) : Nat
```

The successor of a natural number *n*.

Using `Nat.succ` *n* should usually be avoided in favor of *n* `+ 1`, which is the simp normal form.

▼ *Example: Proofs by Induction*

The natural numbers are an inductive type, so the **induction** tactic can be used to prove universally-quantified statements. A proof by induction requires a base case and an induction step. The base case is a proof that the statement is true for `0`. The induction step is a proof that the truth of the statement for some arbitrary number *i* implies its truth for *i* + 1.

This proof uses the lemma `Nat.succ_lt_succ` in its induction step.

```
example (n : Nat) : n < n + 1 := by
  induction n with
  | zero =>
    show 0 < 1
    decide
  | succ i ih => -- ih : i < i + 1
    show i + 1 < i + 1 + 1
    exact Nat.succ_lt_succ ih
```

19.1.1.1. Peano Axioms

The Peano axioms are a consequence of this definition. The induction principle generated for `Nat` is the one demanded by the axiom of induction:

```
Nat.rec.{u} {motive : Nat → Sort u}
  (zero : motive zero)
  (succ : (n : Nat) → motive n → motive n.succ)
  (t : Nat) :
  motive t
```

This induction principle also implements primitive recursion. The injectivity of `Nat.succ` and the disjointness of `Nat.succ` and `Nat.zero` are consequences of the induction principle, using a construction typically called “no confusion”:

```
def NoConfusion : Nat → Nat → Prop
  | 0, 0 => True
  | 0, _ + 1 | _ + 1, 0 => False
  | n + 1, k + 1 => n = k

theorem noConfusionDiagonal (n : Nat) :
    NoConfusion n n :=
  Nat.rec True.intro (fun _ _ => rfl) n

theorem noConfusion (n k : Nat) (eq : n = k) :
    NoConfusion n k :=
  eq ▸ noConfusionDiagonal n

theorem succ_injective : n + 1 = k + 1 → n = k :=
  noConfusion (n + 1) (k + 1)
```

```
theorem succ_not_zero : ¬n + 1 = 0 :=
  noConfusion (n + 1) 0
```

19.1.2. Run-Time Representation

The representation suggested by the declaration of `Nat` would be horrendously inefficient, as it's essentially a linked list. The length of the list would be the number. With this representation, addition would take time linear in the size of one of the addends, and numbers would take at least as many machine words as their magnitude in memory. Thus, natural numbers have special support in both the kernel and the compiler that avoids this overhead.

In the kernel, there are special `Nat` literal values that use a widely-trusted, efficient arbitrary-precision integer library (usually GMP). Basic functions such as addition are overridden by primitives that use this representation. Because they are part of the kernel, if these primitives did not correspond to their definitions as Lean functions, it could undermine soundness.

In compiled code, sufficiently-small natural numbers are represented without pointer indirections: the lowest-order bit in an object pointer is used to indicate that the value is not, in fact, a pointer, and the remaining bits are used to store the number. 31 bits are available on 32-bits architectures for pointer-free `Nat`s, while 63 bits are available on 64-bit architectures. In other words, natural numbers smaller than $2^{31} = 2,147,483,648$ or $2^{63} = 9,223,372,036,854,775,808$ do not require allocations. If an natural number is too large for this representation, it is instead allocated as an ordinary Lean object that consists of an object header and an arbitrary-precision integer value.

19.1.2.1. Performance Notes

Using Lean's built-in arithmetic operators, rather than redefining them, is essential. The logical model of `Nat` is essentially a linked list, so addition would take time linear in the size of one argument. Still worse, multiplication takes quadratic time in this model. While defining arithmetic from scratch can be a useful learning exercise, these redefined operations will not be nearly as fast.

19.1.3. Syntax

Natural number literals are overridden using the `OfNat` type class, which is described in the section on literal syntax.

19.1.4. API Reference

19.1.4.1. Arithmetic

def

```
Nat.pred : Nat → Nat
```

The predecessor of a natural number is one less than it. The predecessor of `0` is defined to be `0`.

This definition is overridden in the compiler with an efficient implementation. This definition is the logical model.

def

`Nat.add : Nat → Nat → Nat`

Addition of natural numbers, typically used via the + operator.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

def

`Nat.sub : Nat → Nat → Nat`

Subtraction of natural numbers, truncated at `0`. Usually used via the − operator.

If a result would be less than zero, then the result is zero.

This definition is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

Examples:

- `5 - 3 = 2`
- `8 - 2 = 6`
- `8 - 8 = 0`
- `8 - 20 = 0`

def

`Nat.mul : Nat → Nat → Nat`

Multiplication of natural numbers, usually accessed via the `*` operator.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

def

`Nat.div (x y : Nat) : Nat`

Division of natural numbers, discarding the remainder. Division by `0` returns `0`. Usually accessed via the `/` operator.

This operation is sometimes called “floor division.”

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `21 / 3 = 7`
- `21 / 5 = 4`
- `0 / 22 = 0`
- `5 / 0 = 0`

def

`Nat.mod : Nat → Nat → Nat`

The modulo operator, which computes the remainder when dividing one natural number by another. Usually accessed via the `%` operator. When the divisor is `0`, the result is the dividend rather than an error.

`Nat.mod` is a wrapper around `Nat.modCore` that special-cases two situations, giving better definitional reductions:

- `Nat.mod 0 m` should reduce to `m`, for all terms `m : Nat`.
- `Nat.mod n (m + n + 1)` should reduce to `n` for concrete `Nat` literals `n`.

These reductions help `Fin n` literals work well, because the `OfNat` instance for `Fin` uses `Nat.mod`. In particular, `(0 : Fin (n + 1)).val` should reduce definitionally to `0`. `Nat.modCore` can handle all numbers, but its definitional reductions are not as convenient.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `7 % 2 = 1`
- `9 % 3 = 0`
- `5 % 7 = 5`
- `5 % 0 = 5`
- `show ∀ (n : Nat), 0 % n = 0 from fun _ => rfl`
- `show ∀ (m : Nat), 5 % (m + 6) = 5 from fun _ => rfl`

def

`Nat.modCore (x y : Nat) : Nat`

The modulo operator, which computes the remainder when dividing one natural number by another. Usually accessed via the `%` operator. When the divisor is `0`, the result is the dividend rather than an error.

This is the core implementation of `Nat.mod`. It computes the correct result for any two closed natural numbers, but it does not have some convenient definitional reductions when the `Nat`s contain free variables. The wrapper `Nat.mod` handles those cases specially and then calls `Nat.modCore`.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

def

```
Nat.pow (m : Nat) : Nat → Nat
```

The power operation on natural numbers, usually accessed via the `^` operator.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

def

```
Nat.log2 (n : Nat) : Nat
```

Base-two logarithm of natural numbers. Returns $\lfloor \texttt{max 0 (log}_2\texttt{ n)} \rfloor$.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `Nat.log2 0 = 0`
- `Nat.log2 1 = 0`
- `Nat.log2 2 = 1`
- `Nat.log2 4 = 2`
- `Nat.log2 7 = 2`
- `Nat.log2 8 = 3`

19.1.4.1.1. Bitwise Operations

def

```
Nat.shiftLeft : Nat → Nat → Nat
```

Shifts the binary representation of a value left by the specified number of bits. Usually accessed via the <<< operator.

Examples:

- `1 <<< 2 = 4`
- `1 <<< 3 = 8`
- `0 <<< 3 = 0`
- `0xf1 <<< 4 = 0xf10`

def

```
Nat.shiftRight : Nat → Nat → Nat
```

Shifts the binary representation of a value right by the specified number of bits. Usually accessed via the >>> operator.

Examples:

- `4 >>> 2 = 1`
- `8 >>> 2 = 2`
- `8 >>> 3 = 1`
- `0 >>> 3 = 0`
- `0xf13a >>> 8 = 0xf1`

def

```
Nat.xor : Nat → Nat → Nat
```

Bitwise exclusive or. Usually accessed via the ^^^ operator.

Each bit of the resulting value is set if the corresponding bit is set in exactly one of the inputs.

def

```
Nat.lor : Nat → Nat → Nat
```

Bitwise or. Usually accessed via the ||| operator.

Each bit of the resulting value is set if the corresponding bit is set in at least one of the inputs.

def

```
Nat.land : Nat → Nat → Nat
```

Bitwise and. Usually accessed via the &&& operator.

Each bit of the resulting value is set if the corresponding bit is set in both of the inputs.

def

```
Nat.bitwise (f : Bool → Bool → Bool) (n m : Nat) : Nat
```

A helper for implementing bitwise operators on `Nat`.

Each bit of the resulting `Nat` is the result of applying *f* to the corresponding bits of the input `Nat`s, up to the position of the highest set bit in either input.

def

```
Nat.testBit (m n : Nat) : Bool
```

Returns `true` if the `(n+1)`th least significant bit is `1`, or `false` if it is `0`.

19.1.4.2. Minimum and Maximum

def

```
Nat.min (n m : Nat) : Nat
```

Returns the lesser of two natural numbers. Usually accessed via `Min.min`.

Returns *n* if $n \le m$, or *m* if $m \le n$.

Examples:

- `min 0 5 = 0`
- `min 4 5 = 4`
- `min 4 3 = 3`
- `min 8 8 = 8`

def

```
Nat.max (n m : Nat) : Nat
```

Returns the greater of two natural numbers. Usually accessed via `Max.max`.

Returns *m* if $n \le m$, or *n* if $m \le n$.

Examples:

- `max 0 5 = 5`

- `max 4 5 = 5`
- `max 4 3 = 4`
- `max 8 8 = 8`

19.1.4.3. GCD and LCM

def

```
Nat.gcd (m n : Nat) : Nat
```

Computes the greatest common divisor of two natural numbers. The GCD of two natural numbers is the largest natural number that evenly divides both.

In particular, the GCD of a number and `0` is the number itself.

This reference implementation via the Euclidean algorithm is overridden in both the kernel and the compiler to efficiently evaluate using arbitrary-precision arithmetic. The definition provided here is the logical model.

Examples:

- `Nat.gcd 10 15 = 5`
- `Nat.gcd 0 5 = 5`
- `Nat.gcd 7 0 = 7`

def

```
Nat.lcm (m n : Nat) : Nat
```

The least common multiple of *m* and *n* is the smallest natural number that's evenly divisible by both *m* and *n*. Returns `0` if either *m* or *n* is `0`.

Examples:

- `Nat.lcm 9 6 = 18`
- `Nat.lcm 9 3 = 9`
- `Nat.lcm 0 3 = 0`
- `Nat.lcm 3 0 = 0`

19.1.4.4. Powers of Two

def

```
Nat.isPowerOfTwo (n : Nat) : Prop
```

A natural number *n* is a power of two if there exists some *k* `:` Nat such that *n* `=` `2` `^` *k*.

def

`Nat.nextPowerOfTwo (`*n* `:` Nat`) :` Nat

Returns the least power of two that's greater than or equal to *n*.

Examples:

- Nat.nextPowerOfTwo `0 = 1`
- Nat.nextPowerOfTwo `1 = 1`
- Nat.nextPowerOfTwo `2 = 2`
- Nat.nextPowerOfTwo `3 = 4`
- Nat.nextPowerOfTwo `5 = 8`

19.1.4.5. Comparisons

19.1.4.5.1. Boolean Comparisons

def

`Nat.beq :` Nat `→` Nat `→` Bool

Boolean equality of natural numbers, usually accessed via the == operator.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

def

`Nat.ble :` Nat `→` Nat `→` Bool

The Boolean less-than-or-equal-to comparison on natural numbers.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

Examples:

- Nat.ble `2 5 =` true
- Nat.ble `5 2 =` false
- Nat.ble `5 5 =` true

def

```
Nat.blt (a b : Nat) : Bool
```

The Boolean less-than comparison on natural numbers.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

Examples:

- `Nat.blt 2 5 = true`
- `Nat.blt 5 2 = false`
- `Nat.blt 5 5 = false`

19.1.4.5.2. Decidable Equality

def

```
Nat.decEq (n m : Nat) : Decidable (n = m)
```

A decision procedure for equality of natural numbers, usually accessed via the `DecidableEq Nat` instance.

This function is overridden in both the kernel and the compiler to efficiently evaluate using the arbitrary-precision arithmetic library. The definition provided here is the logical model.

Examples:

- `Nat.decEq 5 5 = isTrue rfl`
- `(if 3 = 4 then "yes" else "no") = "no"`
- `show 12 = 12 by decide`

def

```
Nat.decLe (n m : Nat) : Decidable (n ≤ m)
```

A decision procedure for non-strict inequality of natural numbers, usually accessed via the `DecidableLE Nat` instance.

Examples:

- `(if 3 ≤ 4 then "yes" else "no") = "yes"`
- `(if 6 ≤ 4 then "yes" else "no") = "no"`
- `show 12 ≤ 12 by decide`
- `show 5 ≤ 12 by decide`

def

```
Nat.decLt (n m : Nat) : Decidable (n ≤ m)
```

A decision procedure for strict inequality of natural numbers, usually accessed via the `DecidableLT Nat` instance.

Examples:

- `(if 3 < 4 then "yes" else "no") = "yes"`
- `(if 4 < 4 then "yes" else "no") = "no"`
- `(if 6 < 4 then "yes" else "no") = "no"`
- `show 5 < 12 by decide`

19.1.4.5.3. Predicates

inductive predicate

```
Nat.le (n : Nat) : Nat → Prop
```

Non-strict, or weak, inequality of natural numbers, usually accessed via the ≤ operator.

Constructors

| `refl {n : Nat} : n.le n`

Non-strict inequality is reflexive: $n \leq n$

| `step {n m : Nat} : n.le m → n.le m.succ`

If $n \leq m$, then $n \leq m + 1$.

def

```
Nat.lt (n m : Nat) : Prop
```

Strict inequality of natural numbers, usually accessed via the < operator.

It is defined as `n < m = n + 1 ≤ m`.

19.1.4.6. Iteration

Many iteration operators come in two versions: a structurally recursive version and a tail-recursive version. The structurally recursive version is typically easier to use in contexts where definitional equality is important, as it will compute when only some prefix of a natural number is known.

def

```
Nat.repeat.{u} {α : Type u} (f : α → α) (n : Nat) (a : α) : α
```

Applies a function to a starting value the specified number of times.

In other words, *f* is iterated *n* times on *a*.

Examples:

- `Nat.repeat f 3 a = f <| f <| f <| a`
- `Nat.repeat (· ++ "!") 4 "Hello" = "Hello!!!!"`

def

```
Nat.repeatTR.{u} {α : Type u} (f : α → α) (n : Nat) (a : α) : α
```

Applies a function to a starting value the specified number of times.

In other words, *f* is iterated *n* times on *a*.

This is a tail-recursive version of `Nat.repeat` that's used at runtime.

Examples:

- `Nat.repeatTR f 3 a = f <| f <| f <| a`
- `Nat.repeatTR (· ++ "!") 4 "Hello" = "Hello!!!!"`

def

```
Nat.fold.{u} {α : Type u} (n : Nat) (f : (i : Nat) → i ≤ n → α → α)
  (init : α) : α
```

Iterates the application of a function *f* to a starting value *init*, *n* times. At each step, *f* is applied to the current value and to the next natural number less than *n*, in increasing order.

Examples:

- `Nat.fold 3 f init = (init |> f 0 (by simp) |> f 1 (by simp) |> f 2 (by simp))`
- `Nat.fold 4 (fun i _ xs => xs.push i) #[] = #[0, 1, 2, 3]`
- `Nat.fold 0 (fun i _ xs => xs.push i) #[] = #[]`

def

```
Nat.foldTR.{u} {α : Type u} (n : Nat) (f : (i : Nat) → i ≤ n → α →
α)
  (init : α) : α
```

Iterates the application of a function *f* to a starting value *init*, *n* times. At each step, *f* is applied to the current value and to the next natural number less than *n*, in increasing order.

This is a tail-recursive version of Nat.fold that's used at runtime.

Examples:

- Nat.foldTR 3 f init = (init |> f 0 (by simp) |> f 1 (by simp) |> f 2 (by simp))
- Nat.foldTR 4 (fun i _ xs => xs.push i) #[] = #[0, 1, 2, 3]
- Nat.foldTR 0 (fun i _ xs => xs.push i) #[] = #[]

def

```
Nat.foldM.{u, v} {α : Type u} {m : Type u → Type v} [Monad m] (n : Nat)
  (f : (i : Nat) → i ≤ n → α → m α) (init : α) : m α
```

Iterates the application of a monadic function *f* to a starting value *init*, *n* times. At each step, *f* is applied to the current value and to the next natural number less than *n*, in increasing order.

def

```
Nat.foldRev.{u} {α : Type u} (n : Nat) (f : (i : Nat) → i ≤ n → α → α)
  (init : α) : α
```

Iterates the application of a function *f* to a starting value *init*, *n* times. At each step, *f* is applied to the current value and to the next natural number less than *n*, in decreasing order.

Examples:

- Nat.foldRev 3 f init = (f 0 (by simp) <| f 1 (by simp) <| f 2 (by simp) init)
- Nat.foldRev 4 (fun i _ xs => xs.push i) #[] = #[3, 2, 1, 0]
- Nat.foldRev 0 (fun i _ xs => xs.push i) #[] = #[]

def

```
Nat.foldRevM.{u, v} {α : Type u} {m : Type u → Type v} [Monad m]
  (n : Nat) (f : (i : Nat) → i ≤ n → α → m α) (init : α) : m α
```

Iterates the application of a monadic function *f* to a starting value *init*, *n* times. At each step, *f* is applied to the current value and to the next natural number less than *n*, in decreasing order.

def

```
Nat.forM.{u_1} {m : Type → Type u_1} [Monad m] (n : Nat)
  (f : (i : Nat) → i ≤ n → m Unit) : m Unit
```

Executes a monadic action on all the numbers less than some bound, in increasing order.

Example:

```
#eval Nat.forM 5 fun i _ => IO.println i
```

```
0
1
2
3
4
```

def

```
Nat.forRevM.{u_1} {m : Type → Type u_1} [Monad m] (n : Nat)
  (f : (i : Nat) → i ≤ n → m Unit) : m Unit
```

Executes a monadic action on all the numbers less than some bound, in decreasing order.

Example:

```
#eval Nat.forRevM 5 fun i _ => IO.println i
```

```
4
3
2
1
0
```

def

```
Nat.all (n : Nat) (f : (i : Nat) → i ≤ n → Bool) : Bool
```

Checks whether *f* returns `true` for every number strictly less than a bound.

Examples:

- `Nat.all 4 (fun i _ => i < 5) = true`
- `Nat.all 7 (fun i _ => i < 5) = false`
- `Nat.all 7 (fun i _ => i % 2 = 0) = false`
- `Nat.all 1 (fun i _ => i % 2 = 0) = true`

def

```
Nat.allTR (n : Nat) (f : (i : Nat) → i ≤ n → Bool) : Bool
```

Checks whether *f* returns `true` for every number strictly less than a bound.

This is a tail-recursive equivalent of `Nat.all` that's used at runtime.

Examples:

- `Nat.allTR 4 (fun i _ => i < 5) = true`
- `Nat.allTR 7 (fun i _ => i < 5) = false`
- `Nat.allTR 7 (fun i _ => i % 2 = 0) = false`
- `Nat.allTR 1 (fun i _ => i % 2 = 0) = true`

def

```
Nat.any (n : Nat) (f : (i : Nat) → i ≤ n → Bool) : Bool
```

Checks whether there is some number less that the given bound for which *f* returns `true`.

Examples:

- `Nat.any 4 (fun i _ => i < 5) = true`
- `Nat.any 7 (fun i _ => i < 5) = true`
- `Nat.any 7 (fun i _ => i % 2 = 0) = true`
- `Nat.any 1 (fun i _ => i % 2 = 1) = false`

def

```
Nat.anyTR (n : Nat) (f : (i : Nat) → i ≤ n → Bool) : Bool
```

Checks whether there is some number less that the given bound for which *f* returns `true`.

This is a tail-recursive equivalent of `Nat.any` that's used at runtime.

Examples:

- `Nat.anyTR 4 (fun i _ => i < 5) = true`
- `Nat.anyTR 7 (fun i _ => i < 5) = true`
- `Nat.anyTR 7 (fun i _ => i % 2 = 0) = true`
- `Nat.anyTR 1 (fun i _ => i % 2 = 1) = false`

def

```
Nat.allM.{u_1} {m : Type → Type u_1} [Monad m] (n : Nat)
  (p : (i : Nat) → i ≤ n → m Bool) : m Bool
```

Checks whether the monadic predicate *p* returns `true` for all numbers less that the given bound. Numbers are checked in increasing order until *p* returns false, after which no further are checked.

def

```
Nat.anyM.{u_1} {m : Type → Type u_1} [Monad m] (n : Nat)
  (p : (i : Nat) → i ≤ n → m Bool) : m Bool
```

Checks whether there is some number less that the given bound for which the monadic predicate *p* returns `true`. Numbers are checked in increasing order until *p* returns true, after which no further are checked.

19.1.4.7. Conversion

def

```
Nat.toUInt8 (n : Nat) : UInt8
```

Converts a natural number to an 8-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Nat.toUInt8 5 = 5`
- `Nat.toUInt8 255 = 255`
- `Nat.toUInt8 256 = 0`
- `Nat.toUInt8 259 = 3`
- `Nat.toUInt8 32770 = 2`

def

```
Nat.toUInt16 (n : Nat) : UInt16
```

Converts a natural number to a 16-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Nat.toUInt16 5 = 5`
- `Nat.toUInt16 255 = 255`
- `Nat.toUInt16 32770 = 32770`

- `Nat.toUInt16 65537 = 1`

def

`Nat.toUInt32 (n : Nat) : UInt32`

Converts a natural number to a 32-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Nat.toUInt32 5 = 5`
- `Nat.toUInt32 65_539 = 65_539`
- `Nat.toUInt32 4_294_967_299 = 3`

def

`Nat.toUInt64 (n : Nat) : UInt64`

Converts a natural number to a 64-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Nat.toUInt64 5 = 5`
- `Nat.toUInt64 65539 = 65539`
- `Nat.toUInt64 4_294_967_299 = 4_294_967_299`
- `Nat.toUInt64 18_446_744_073_709_551_620 = 4`

def

`Nat.toUSize (n : Nat) : USize`

Converts an arbitrary-precision natural number to an unsigned word-sized integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

def

`Nat.toInt8 (n : Nat) : Int8`

Converts a natural number to an 8-bit signed integer, wrapping around to negative numbers on overflow.

Examples:

- `Nat.toInt8 53 = 53`
- `Nat.toInt8 127 = 127`
- `Nat.toInt8 128 = -128`
- `Nat.toInt8 255 = -1`

def

```
Nat.toInt16 (n : Nat) : Int16
```

Converts a natural number to a 16-bit signed integer, wrapping around to negative numbers on overflow.

Examples:

- `Nat.toInt16 127 = 127`
- `Nat.toInt16 32767 = 32767`
- `Nat.toInt16 32768 = -32768`
- `Nat.toInt16 32770 = -32766`

def

```
Nat.toInt32 (n : Nat) : Int32
```

Converts a natural number to a 32-bit signed integer, wrapping around to negative numbers on overflow.

Examples:

- `Nat.toInt32 127 = 127`
- `Nat.toInt32 32770 = 32770`
- `Nat.toInt32 2_147_483_647 = 2_147_483_647`
- `Nat.toInt32 2_147_483_648 = -2_147_483_648`

def

```
Nat.toInt64 (n : Nat) : Int64
```

Converts a natural number to a 64-bit signed integer, wrapping around to negative numbers on overflow.

Examples:

- `Nat.toInt64 127 = 127`

- `Nat.toInt64 2_147_483_648 = 2_147_483_648`
- `Nat.toInt64 9_223_372_036_854_775_807 = 9_223_372_036_854_775_807`
- `Nat.toInt64 9_223_372_036_854_775_808 = -9_223_372_036_854_775_808`
- `Nat.toInt64 18_446_744_073_709_551_618 = 0`

def

```
Nat.toISize (n : Nat) : ISize
```

Converts an arbitrary-precision natural number to a word-sized signed integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
Nat.toFloat (n : Nat) : Float
```

Converts a natural number into the closest-possible 64-bit floating-point number, or an infinite floating-point value if the range of `Float` is exceeded.

def

```
Nat.toFloat32 (n : Nat) : Float32
```

Converts a natural number into the closest-possible 32-bit floating-point number, or an infinite floating-point value if the range of `Float32` is exceeded.

def

```
Nat.isValidChar (n : Nat) : Prop
```

A `Nat` denotes a valid Unicode code point if it is less than `0x110000` and it is also not a surrogate code point (the range `0xd800` to `0xdfff` inclusive).

def

```
Nat.repr (n : Nat) : String
```

Converts a natural number to its decimal string representation.

def

```
Nat.toDigits (base n : Nat) : List Char
```

Returns the decimal representation of a natural number as a list of digit characters in the given base. If the base is greater than `16` then `'*'` is returned for digits greater than `0xf`.

Examples:

- `Nat.toDigits 10 0xff = ['2', '5', '5']`
- `Nat.toDigits 8 0xc = ['1', '4']`
- `Nat.toDigits 16 0xcafe = ['c', 'a', 'f', 'e']`
- `Nat.toDigits 80 200 = ['2', '*']`

def

```
Nat.digitChar (n : Nat) : Char
```

Returns a single digit representation of *n*, which is assumed to be in a base less than or equal to `16`. Returns `'*'` if *n* `> 15`.

Examples:

- `Nat.digitChar 5 = '5'`
- `Nat.digitChar 12 = 'c'`
- `Nat.digitChar 15 = 'f'`
- `Nat.digitChar 16 = '*'`
- `Nat.digitChar 85 = '*'`

def

```
Nat.toSubscriptString (n : Nat) : String
```

Converts a natural number to a string that contains the its decimal representation as Unicode subscript digit characters.

Examples:

- `Nat.toSubscriptString 0 = "₀"`
- `Nat.toSubscriptString 35 = "₃₅"`

def

```
Nat.toSuperscriptString (n : Nat) : String
```

Converts a natural number to a string that contains the its decimal representation as Unicode superscript digit characters.

Examples:

- `Nat.toSuperscriptString 0 = "⁰"`
- `Nat.toSuperscriptString 35 = "³⁵"`

def

`Nat.toSuperDigits (n : Nat) : List Char`

Converts a natural number to the list of Unicode superscript digit characters that corresponds to its decimal representation.

Examples:

- `Nat.toSuperDigits 0 = ['⁰']`
- `Nat.toSuperDigits 35 = ['³', '⁵']`

def

`Nat.toSubDigits (n : Nat) : List Char`

Converts a natural number to the list of Unicode subscript digit characters that corresponds to its decimal representation.

Examples:

- `Nat.toSubDigits 0 = ['₀']`
- `Nat.toSubDigits 35 = ['₃', '₅']`

def

`Nat.subDigitChar (n : Nat) : Char`

Converts a natural number less than `10` to the corresponding Unicode subscript digit character. Returns `'*'` for other numbers.

Examples:

- `Nat.subDigitChar 3 = '₃'`
- `Nat.subDigitChar 7 = '₇'`
- `Nat.subDigitChar 10 = '*'`

def

`Nat.superDigitChar (n : Nat) : Char`

Converts a natural number less than `10` to the corresponding Unicode superscript digit character. Returns `'*'` for other numbers.

Examples:

- `Nat.superDigitChar 3 = '³'`
- `Nat.superDigitChar 7 = '⁷'`
- `Nat.superDigitChar 10 = '*'`

19.1.4.8. Elimination

The recursion principle that is automatically generated for `Nat` results in proof goals that are phrased in terms of `Nat.zero` and `Nat.succ`. This is not particularly user-friendly, so an alternative logically-equivalent recursion principle is provided that results in goals that are phrased in terms of `0` and `n + 1`. Custom eliminators for the **`induction`** and **`cases`** tactics can be supplied using the **`induction_eliminator`** and **`cases_eliminator`** attributes.

def

```
Nat.recAux.{u} {motive : Nat → Sort u} (zero : motive 0)
  (succ : (n : Nat) → motive n → motive (n + 1)) (t : Nat) : motive
t
```

A recursor for `Nat` that uses the notations `0` for `Nat.zero` and `n + 1` for `Nat.succ`.

It is otherwise identical to the default recursor `Nat.rec`. It is used by the **`induction`** tactic by default for `Nat`.

def

```
Nat.casesAuxOn.{u} {motive : Nat → Sort u} (t : Nat) (zero : motive
0)
  (succ : (n : Nat) → motive (n + 1)) : motive t
```

A case analysis principle for `Nat` that uses the notations `0` for `Nat.zero` and `n + 1` for `Nat.succ`.

It is otherwise identical to the default recursor `Nat.casesOn`. It is used as the default `Nat` case analysis principle for `Nat` by the **`cases`** tactic.

19.1.4.8.1. Alternative Induction Principles

def

```
Nat.strongRecOn.{u} {motive : Nat → Sort u} (n : Nat)
  (ind : (n : Nat) → ((m : Nat) → m ≤ n → motive m) → motive n) :
  motive n
```

Strong induction on the natural numbers.

The induction hypothesis is that all numbers less than a given number satisfy the motive, which should be demonstrated for the given number.

def

```
Nat.caseStrongRecOn.{u} {motive : Nat → Sort u} (a : Nat)
  (zero : motive 0)
  (ind : (n : Nat) → ((m : Nat) → m ≤ n → motive m) → motive
n.succ) :
  motive a
```

Case analysis based on strong induction for the natural numbers.

def

```
Nat.div.inductionOn.{u} {motive : Nat → Nat → Sort u} (x y : Nat)
  (ind : (x y : Nat) → 0 < y ∧ y ≤ x → motive (x - y) y → motive x
y)
  (base : (x y : Nat) → ¬(0 < y ∧ y ≤ x) → motive x y) : motive x y
```

An induction principle customized for reasoning about the recursion pattern of natural number division by iterated subtraction.

def

```
Nat.div2Induction.{u} {motive : Nat → Sort u} (n : Nat)
  (ind : (n : Nat) → (n > 0 → motive (n / 2)) → motive n) : motive
n
```

An induction principle for the natural numbers with two cases:

- `n = 0`, and the motive is satisfied for `0`
- `n > 0`, and the motive should be satisfied for `n` on the assumption that it is satisfied for `n / 2`.

def

```
Nat.mod.inductionOn.{u} {motive : Nat → Nat → Sort u} (x y : Nat)
  (ind : (x y : Nat) → 0 < y ∧ y ≤ x → motive (x - y) y → motive x
y)
  (base : (x y : Nat) → ¬(0 < y ∧ y ≤ x) → motive x y) : motive x y
```

An induction principle customized for reasoning about the recursion pattern of `Nat.mod`.

19.2. Integers

The integers are whole numbers, both positive and negative. Integers are arbitrary-precision, limited only by the capability of the hardware on which Lean is running; for fixed-width integers that are used in programming and computer science, please see the section on fixed-precision integers.

Integers are specially supported by Lean's implementation. The logical model of the integers is based on the natural numbers: each integer is modeled as either a natural number or the negative successor of a natural number. Operations on the integers are specified using this model, which is used in the kernel and in interpreted code. In these contexts, integer code inherits the performance benefits of the natural numbers' special support. In compiled code, integers are represented as efficient arbitrary-precision integers, and sufficiently small numbers are stored as values that don't require indirection through a pointer. Arithmetic operations are implemented by primitives that take advantage of the efficient representations.

19.2.1. Logical Model

Integers are represented either as a natural number or as the negation of the successor of a natural number.

inductive type

```
Int : Type
```

The integers.

This type is special-cased by the compiler and overridden with an efficient implementation. The runtime has a special representation for `Int` that stores “small” signed numbers directly, while larger numbers use a fast arbitrary-precision arithmetic library (usually GMP). A “small number” is an integer that can be encoded with one fewer bits than the platform's pointer size (i.e. 63 bits on 64-bit architectures and 31 bits on 32-bit architectures).

Constructors

| `ofNat : Nat → Int`

A natural number is an integer.

This constructor covers the non-negative integers (from `0` to ∞).

| `negSucc : Nat → Int`

The negation of the successor of a natural number is an integer.

This constructor covers the negative integers (from `-1` to -∞).

This representation of the integers has a number of useful properties. It is relatively simple to use and to understand. Unlike a pair of a sign and a `Nat`, there is a unique representation for 0, which simplifies reasoning about equality. Integers can also be represented as a pair of natural numbers in which one is subtracted from the other, but this requires a quotient type to be well-behaved, and quotient types can be laborious to work with due to the need to prove that functions respect the equivalence relation.

19.2.2. Run-Time Representation

Like natural numbers, sufficiently-small integers are represented without pointers: the lowest-order bit in an object pointer is used to indicate that the value is not, in fact, a pointer. If an integer is too large to fit in the remaining bits, it is instead allocated as an ordinary Lean object that consists of an object header and an arbitrary-precision integer.

19.2.3. Syntax

The `OfNat Int` instance allows numerals to be used as literals, both in expression and in pattern contexts. (`OfNat.ofNat` *n* `: Int`) reduces to the constructor application `Int.ofNat` *n*. The `Neg Int` instance allows negation to be used as well.

On top of these instances, there is special syntax for the constructor `Int.negSucc` that is available when the `Int` namespace is opened. The notation `-[ n +1]` is suggestive of $-(n+1)$, which is the meaning of `Int.negSucc` *n*.

syntax

Negative Successor

`-[ n +1]` is notation for `Int.negSucc` *n*.

```
term ::= ...
    | -[ term +1]
```

19.2.4. API Reference

19.2.4.1. Properties

def

`Int.sign : Int → Int`

Returns the “sign” of the integer as another integer:

- `1` for positive numbers,
- `-1` for negative numbers, and
- `0` for `0`.

Examples:

- `Int.sign 34 = 1`
- `Int.sign 2 = 1`
- `Int.sign 0 = 0`
- `Int.sign -1 = -1`
- `Int.sign -362 = -1`

19.2.4.2. Conversions

def

`Int.natAbs (m : Int) : Nat`

The absolute value of an integer is its distance from `0`.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(7 : Int).natAbs = 7`
- `(0 : Int).natAbs = 0`
- `((-11 : Int).natAbs = 11`

def

`Int.toNat : Int → Nat`

Converts an integer into a natural number. Negative numbers are converted to `0`.

Examples:

- `(7 : Int).toNat = 7`
- `(0 : Int).toNat = 0`
- `(-7 : Int).toNat = 0`

def

`Int.toNat? : Int → Option Nat`

Converts an integer into a natural number. Returns `none` for negative numbers.

Examples:

- `(7 : Int).toNat? = some 7`

- `(0 : Int).toNat? = some 0`
- `(-7 : Int).toNat? = none`

def

```
Int.toISize (i : Int) : ISize
```

Converts an arbitrary-precision integer to a word-sized signed integer, wrapping around on over- or underflow.

This function is overridden at runtime with an efficient implementation.

def

```
Int.toInt8 (i : Int) : Int8
```

Converts an arbitrary-precision integer to an 8-bit integer, wrapping on overflow or underflow.

Examples:

- `Int.toInt8 48 = 48`
- `Int.toInt8 (-115) = -115`
- `Int.toInt8 (-129) = 127`
- `Int.toInt8 (128) = -128`

def

```
Int.toInt16 (i : Int) : Int16
```

Converts an arbitrary-precision integer to a 16-bit integer, wrapping on overflow or underflow.

Examples:

- `Int.toInt16 48 = 48`
- `Int.toInt16 (-129) = -129`
- `Int.toInt16 (128) = 128`
- `Int.toInt16 70000 = 4464`
- `Int.toInt16 (-40000) = 25536`

def

```
Int.toInt32 (i : Int) : Int32
```

Converts an arbitrary-precision integer to a 32-bit integer, wrapping on overflow or underflow.

Examples:

- `Int.toInt32 48 = 48`
- `Int.toInt32 (-129) = -129`
- `Int.toInt32 70000 = 70000`
- `Int.toInt32 (-40000) = -40000`
- `Int.toInt32 2147483648 = -2147483648`
- `Int.toInt32 (-2147483649) = 2147483647`

def

```
Int.toInt64 (i : Int) : Int64
```

Converts an arbitrary-precision integer to a 64-bit integer, wrapping on overflow or underflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int.toInt64 48 = 48`
- `Int.toInt64 (-40_000) = -40_000`
- `Int.toInt64 2_147_483_648 = 2_147_483_648`
- `Int.toInt64 (-2_147_483_649) = -2_147_483_649`
- `Int.toInt64 9_223_372_036_854_775_808 = -9_223_372_036_854_775_808`
- `Int.toInt64 (-9_223_372_036_854_775_809) = 9_223_372_036_854_775_807`

def

```
Int.repr : Int → String
```

Returns the decimal string representation of an integer.

19.2.4.3. Arithmetic

Typically, arithmetic operations on integers are accessed using Lean's overloaded arithmetic notation. In particular, the instances of `Add Int`, `Neg Int`, `Sub Int`, and `Mul Int` allow ordinary infix operators to be used. Division is somewhat more intricate, because there are multiple sensible notions of division on integers.

def

```
Int.add (m n : Int) : Int
```

Addition of integers, usually accessed via the + operator.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(7 : Int) + (6 : Int) = 13`
- `(6 : Int) + (-6 : Int) = 0`

def

`Int.sub (m n : Int) : Int`

Subtraction of integers, usually accessed via the − operator.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(63 : Int) - (6 : Int) = 57`
- `(7 : Int) - (0 : Int) = 7`
- `(0 : Int) - (7 : Int) = -7`

def

`Int.subNatNat (m n : Nat) : Int`

Non-truncating subtraction of two natural numbers.

Examples:

- `Int.subNatNat 5 2 = 3`
- `Int.subNatNat 2 5 = -3`
- `Int.subNatNat 0 13 = -13`

def

`Int.neg (n : Int) : Int`

Negation of integers, usually accessed via the − prefix operator.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `-(6 : Int) = -6`
- `-(-6 : Int) = 6`
- `(12 : Int).neg = -12`

def

`Int.negOfNat : Nat → Int`

Negation of natural numbers.

Examples:

- `Int.negOfNat 6 = -6`
- `Int.negOfNat 0 = 0`

def

`Int.mul (m n : Int) : Int`

Multiplication of integers, usually accessed via the `*` operator.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(63 : Int) * (6 : Int) = 378`
- `(6 : Int) * (-6 : Int) = -36`
- `(7 : Int) * (0 : Int) = 0`

def

`Int.pow (m : Int) : Nat → Int`

Power of an integer to a natural number, usually accessed via the `^` operator.

Examples:

- `(2 : Int) ^ 4 = 16`
- `(10 : Int) ^ 0 = 1`
- `(0 : Int) ^ 10 = 0`
- `(-7 : Int) ^ 3 = -343`

def

```
Int.gcd (m n : Int) : Nat
```

Computes the greatest common divisor of two integers as a natural number. The GCD of two integers is the largest natural number that evenly divides both. However, the GCD of a number and `0` is the number's absolute value.

This implementation uses `Nat.gcd`, which is overridden in both the kernel and the compiler to efficiently evaluate using arbitrary-precision arithmetic.

Examples:

- `Int.gcd 10 15 = 5`
- `Int.gcd 10 (-15) = 5`
- `Int.gcd (-6) (-9) = 3`
- `Int.gcd 0 5 = 5`
- `Int.gcd (-7) 0 = 7`

def

```
Int.lcm (m n : Int) : Nat
```

Computes the least common multiple of two integers as a natural number. The LCM of two integers is the smallest natural number that's evenly divisible by the absolute values of both.

Examples:

- `Int.lcm 9 6 = 18`
- `Int.lcm 9 (-6) = 18`
- `Int.lcm 9 3 = 9`
- `Int.lcm 9 (-3) = 9`
- `Int.lcm 0 3 = 0`
- `Int.lcm (-3) 0 = 0`

19.2.4.3.1. Division

The `Div Int` and `Mod Int` instances implement Euclidean division, described in the reference for `Int.ediv`. This is not, however, the only sensible convention for rounding and remainders in division. Four pairs of division and modulus functions are available, implementing various conventions.

▼ *Example: Division by 0*

In all integer division conventions, division by `0` is defined to be `0`:

```
#eval Int.ediv 5 0
#eval Int.ediv 0 0
#eval Int.ediv (-5) 0
#eval Int.bdiv 5 0
#eval Int.bdiv 0 0
#eval Int.bdiv (-5) 0
#eval Int.fdiv 5 0
#eval Int.fdiv 0 0
#eval Int.fdiv (-5) 0
#eval Int.tdiv 5 0
#eval Int.tdiv 0 0
#eval Int.tdiv (-5) 0
```

All evaluate to 0.

```
0
```

def

`Int.ediv : Int → Int → Int`

Integer division that uses the E-rounding convention. Usually accessed via the `/` operator. Division by zero is defined to be zero, rather than an error.

In the E-rounding convention (Euclidean division), `Int.emod x y` satisfies `0 ≤ Int.emod x y < Int.natAbs y` for $y \neq 0$ and `Int.ediv` is the unique function satisfying `Int.emod x y + (Int.ediv x y) * y = x` for $y \neq 0$.

This means that `Int.ediv x y` is $\lfloor x / y \rfloor$ when $y > 0$ and $\lceil x / y \rceil$ when $y < 0$.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(7 : Int) / (0 : Int) = 0`
- `(0 : Int) / (7 : Int) = 0`
- `(12 : Int) / (6 : Int) = 2`
- `(12 : Int) / (-6 : Int) = -2`
- `(-12 : Int) / (6 : Int) = -2`
- `(-12 : Int) / (-6 : Int) = 2`
- `(12 : Int) / (7 : Int) = 1`

- `(12 : Int) / (-7 : Int) = -1`
- `(-12 : Int) / (7 : Int) = -2`
- `(-12 : Int) / (-7 : Int) = 2`

def

`Int.emod : Int → Int → Int`

Integer modulus that uses the E-rounding convention. Usually accessed via the `%` operator.

In the E-rounding convention (Euclidean division), `Int.emod x y` satisfies `0 ≤ Int.emod x y < Int.natAbs y` for $y \neq 0$ and `Int.ediv` is the unique function satisfying `Int.emod x y + (Int.ediv x y) * y = x` for $y \neq 0$.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(7 : Int) % (0 : Int) = 7`
- `(0 : Int) % (7 : Int) = 0`
- `(12 : Int) % (6 : Int) = 0`
- `(12 : Int) % (-6 : Int) = 0`
- `(-12 : Int) % (6 : Int) = 0`
- `(-12 : Int) % (-6 : Int) = 0`
- `(12 : Int) % (7 : Int) = 5`
- `(12 : Int) % (-7 : Int) = 5`
- `(-12 : Int) % (7 : Int) = 2`
- `(-12 : Int) % (-7 : Int) = 2`

def

`Int.tdiv : Int → Int → Int`

Integer division using the T-rounding convention.

In the T-rounding convention (division with truncation), all rounding is towards zero. Division by 0 is defined to be 0. In this convention, `Int.tmod a b + b * (Int.tdiv a b) = a`.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(7 : Int).tdiv (0 : Int) = 0`
- `(0 : Int).tdiv (7 : Int) = 0`
- `(12 : Int).tdiv (6 : Int) = 2`
- `(12 : Int).tdiv (-6 : Int) = -2`
- `(-12 : Int).tdiv (6 : Int) = -2`
- `(-12 : Int).tdiv (-6 : Int) = 2`
- `(12 : Int).tdiv (7 : Int) = 1`
- `(12 : Int).tdiv (-7 : Int) = -1`
- `(-12 : Int).tdiv (7 : Int) = -1`
- `(-12 : Int).tdiv (-7 : Int) = 1`

def

```
Int.tmod : Int → Int → Int
```

Integer modulo using the T-rounding convention.

In the T-rounding convention (division with truncation), all rounding is towards zero. Division by 0 is defined to be 0 and `Int.tmod a 0 = a`.

In this convention, `Int.tmod a b + b * (Int.tdiv a b) = a`. Additionally, `Int.natAbs (Int.tmod a b) = Int.natAbs a % Int.natAbs b`, and when *b* does not divide *a*, `Int.tmod a b` has the same sign as *a*.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `(7 : Int).tmod (0 : Int) = 7`
- `(0 : Int).tmod (7 : Int) = 0`
- `(12 : Int).tmod (6 : Int) = 0`
- `(12 : Int).tmod (-6 : Int) = 0`
- `(-12 : Int).tmod (6 : Int) = 0`
- `(-12 : Int).tmod (-6 : Int) = 0`
- `(12 : Int).tmod (7 : Int) = 5`
- `(12 : Int).tmod (-7 : Int) = 5`
- `(-12 : Int).tmod (7 : Int) = -5`
- `(-12 : Int).tmod (-7 : Int) = -5`

def

```
Int.bdiv (x : Int) (m : Nat) : Int
```

Balanced division.

This returns the unique integer so that `b * (Int.bdiv a b) + Int.bmod a b = a`.

Examples:

- `(7 : Int).bdiv 0 = 0`
- `(0 : Int).bdiv 7 = 0`
- `(12 : Int).bdiv 6 = 2`
- `(12 : Int).bdiv 7 = 2`
- `(12 : Int).bdiv 8 = 2`
- `(12 : Int).bdiv 9 = 1`
- `(-12 : Int).bdiv 6 = -2`
- `(-12 : Int).bdiv 7 = -2`
- `(-12 : Int).bdiv 8 = -1`
- `(-12 : Int).bdiv 9 = -1`

def

```
Int.bmod (x : Int) (m : Nat) : Int
```

Balanced modulus.

This version of integer modulus uses the balanced rounding convention, which guarantees that `-m / 2 ≤ Int.bmod x m < m/2` for `m ≠ 0` and `Int.bmod x m` is congruent to `x` modulo `m`.

If `m = 0`, then `Int.bmod x m = x`.

Examples:

- `(7 : Int).bmod 0 = 7`
- `(0 : Int).bmod 7 = 0`
- `(12 : Int).bmod 6 = 0`
- `(12 : Int).bmod 7 = -2`
- `(12 : Int).bmod 8 = -4`
- `(12 : Int).bmod 9 = 3`
- `(-12 : Int).bmod 6 = 0`
- `(-12 : Int).bmod 7 = 2`

- `(-12 : Int).bmod 8 = -4`
- `(-12 : Int).bmod 9 = -3`

def

```
Int.fdiv : Int → Int → Int
```

Integer division using the F-rounding convention.

In the F-rounding convention (flooring division), `Int.fdiv x y` satisfies `Int.fdiv x y = ⌊x / y⌋` and `Int.fmod` is the unique function satisfying `Int.fmod x y + (Int.fdiv x y) * y = x`.

Examples:

- `(7 : Int).fdiv (0 : Int) = 0`
- `(0 : Int).fdiv (7 : Int) = 0`
- `(12 : Int).fdiv (6 : Int) = 2`
- `(12 : Int).fdiv (-6 : Int) = -2`
- `(-12 : Int).fdiv (6 : Int) = -2`
- `(-12 : Int).fdiv (-6 : Int) = 2`
- `(12 : Int).fdiv (7 : Int) = 1`
- `(12 : Int).fdiv (-7 : Int) = -2`
- `(-12 : Int).fdiv (7 : Int) = -2`
- `(-12 : Int).fdiv (-7 : Int) = 1`

def

```
Int.fmod : Int → Int → Int
```

Integer modulus using the F-rounding convention.

In the F-rounding convention (flooring division), `Int.fdiv x y` satisfies `Int.fdiv x y = ⌊x / y⌋` and `Int.fmod` is the unique function satisfying `Int.fmod x y + (Int.fdiv x y) * y = x`.

Examples:

- `(7 : Int).fmod (0 : Int) = 7`
- `(0 : Int).fmod (7 : Int) = 0`
- `(12 : Int).fmod (6 : Int) = 0`
- `(12 : Int).fmod (-6 : Int) = 0`

- `(-12 : Int).fmod (6 : Int) = 0`
- `(-12 : Int).fmod (-6 : Int) = 0`
- `(12 : Int).fmod (7 : Int) = 5`
- `(12 : Int).fmod (-7 : Int) = -2`
- `(-12 : Int).fmod (7 : Int) = 2`
- `(-12 : Int).fmod (-7 : Int) = -5`

19.2.4.4. Bitwise Operators

Bitwise operators on `Int` can be understood as bitwise operators on an infinite stream of bits that are the twos-complement representation of integers.

def

```
Int.not : Int → Int
```

Bitwise not, usually accessed via the ~~~ prefix operator.

Interprets the integer as an infinite sequence of bits in two's complement and complements each bit.

Examples:

- `~~~(0 : Int) = -1`
- `~~~(1 : Int) = -2`
- `~~~(-1 : Int) = 0`

def

```
Int.shiftRight : Int → Nat → Int
```

Bitwise right shift, usually accessed via the >>> operator.

Interprets the integer as an infinite sequence of bits in two's complement and shifts the value to the right.

Examples:

- `( 0b0111 : Int) >>> 1 =  0b0011`
- `( 0b1000 : Int) >>> 1 =  0b0100`
- `(-0b1000 : Int) >>> 1 = -0b0100`
- `(-0b0111 : Int) >>> 1 = -0b0100`

19.2.4.5. Comparisons

Equality and inequality tests on `Int` are typically performed using the decidability of its equality and ordering relations or using the `BEq Int` and `Ord Int` instances.

def

```
Int.le (a b : Int) : Prop
```

Non-strict inequality of integers, usually accessed via the ≤ operator.

a `≤` *b* is defined as *b* `-` *a* `≥ 0`, using `Int.NonNeg`.

def

```
Int.lt (a b : Int) : Prop
```

Strict inequality of integers, usually accessed via the < operator.

a `<` *b* when *a* `+ 1 ≤` *b*.

def

```
Int.decEq (a b : Int) : Decidable (a = b)
```

Decides whether two integers are equal. Usually accessed via the `DecidableEq Int` instance.

This function is overridden by the compiler with an efficient implementation. This definition is the logical model.

Examples:

- `show (7 : Int) = (3 : Int) + (4 : Int) by decide`
- `if (6 : Int) = (3 : Int) * (2 : Int) then "yes" else "no" = "yes"`
- `(¬ (6 : Int) = (3 : Int)) = true`

19.3. Finite Natural Numbers

For any natural number *n*, the `Fin n` is a type that contains all the natural numbers that are strictly less than *n*. In other words, `Fin n` has exactly *n* elements. It can be used to represent the valid indices into a list or array, or it can serve as a canonical *n*-element type.

structure

```
Fin (n : Nat) : Type
```

Natural numbers less than some upper bound.

In particular, a `Fin n` is a natural number `i` with the constraint that `i < n`. It is the canonical type with *n* elements.

Constructor

`Fin.mk`

Creates a `Fin n` from `i : Nat` and a proof that `i < n`.

Fields

`val : Nat`

The number that is strictly less than *n*.

`Fin.val` is a coercion, so any `Fin n` can be used in a position where a `Nat` is expected.

`isLt : ↑self ≤ n`

The number `val` is strictly less than the bound *n*.

`Fin` is closely related to `UInt8`, `UInt16`, `UInt32`, `UInt64`, and `USize`, which also represent finite non-negative integral types. However, these types are backed by bitvectors rather than by natural numbers, and they have fixed bounds. `Fin` is comparatively more flexible, but also less convenient for low-level reasoning. In particular, using bitvectors rather than proofs that a number is less than some power of two avoids needing to take care to avoid evaluating the concrete bound.

19.3.1. Run-Time Characteristics

Because `Fin n` is a structure in which only a single field is not a proof, it is a trivial wrapper. This means that it is represented identically to the underlying natural number in compiled code.

19.3.2. Coercions and Literals

There is a coercion from `Fin n` to `Nat` that discards the proof that the number is less than the bound. In particular, this coercion is precisely the projection `Fin.val`. One consequence of this is that uses of `Fin.val` are displayed as coercions rather than explicit projections in proof states.

▼ *Example: Coercing from `Fin` to `Nat`*

A `Fin n` can be used where a `Nat` is expected:

```
#eval let one : Fin 3 := ⟨1, by omega⟩; (one : Nat)
```

```
1
```

Uses of `Fin.val` show up as coercions in proof states:

```
n : Nat
i : Fin n
⊢ ↑i ≤ n
```

Natural number literals may be used for `Fin` types, implemented as usual via an `OfNat` instance. The `OfNat` instance for `Fin n` requires that the upper bound *n* is not zero, but does not check that the literal is less than *n*. If the literal is larger than the type can represent, the remainder when dividing it by *n* is used.

▼ *Example: Numeric Literals for* `Fin`

If `n > 0`, then natural number literals can be used for `Fin n`:

```
example : Fin 5 := 3
example : Fin 20 := 19
```

When the literal is greater than or equal to *n*, the remainder when dividing by *n* is used:

```
#eval (5 : Fin 3)
```

```
2
```

```
#eval ([0, 1, 2, 3, 4, 5, 6] : List (Fin 3))
```

```
[0, 1, 2, 0, 1, 2, 0]
```

If Lean can't synthesize an instance of `NeZero n`, then there is no `OfNat (Fin n)` instance:

```
example : Fin 0 := 0
```

```
failed to synthesize
  OfNat (Fin 0) 0
numerals are polymorphic in Lean, but the numeral `0` cannot
be used in a context where the expected type is
  Fin 0
due to the absence of the instance above

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

```
example (k : Nat) : Fin k := 0
```

```
failed to synthesize
  OfNat (Fin k) 0
numerals are polymorphic in Lean, but the numeral `0` cannot
be used in a context where the expected type is
  Fin k
due to the absence of the instance above
```

```
Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

19.3.3. API Reference

19.3.3.1. Construction

def

```
Fin.last (n : Nat) : Fin (n + 1)
```

The greatest value of `Fin (n+1)`, namely `n`.

Examples:

- `Fin.last 4 = (4 : Fin 5)`
- `(Fin.last 0).val = (0 : Nat)`

def

```
Fin.succ {n : Nat} : Fin n → Fin (n + 1)
```

The successor, with an increased bound.

This differs from adding 1, which instead wraps around.

Examples:

- `(2 : Fin 3).succ = (3 : Fin 4)`
- `(2 : Fin 3) + 1 = (0 : Fin 3)`

def

```
Fin.pred {n : Nat} (i : Fin (n + 1)) (h : i ≠ 0) : Fin n
```

The predecessor of a non-zero element of `Fin (n+1)`, with the bound decreased.

Examples:

- `(4 : Fin 8).pred (by decide) = (3 : Fin 7)`
- `(1 : Fin 2).pred (by decide) = (0 : Fin 1)`

19.3.3.2. Arithmetic

Typically, arithmetic operations on `Fin` should be accessed using Lean's overloaded arithmetic notation, particularly via the instances `Add (Fin n)`, `Sub (Fin n)`, `Mul (Fin n)`, `Div (Fin n)`, and `Mod (Fin n)`. Heterogeneous operators such as `Fin.natAdd` do not have

corresponding heterogeneous instances (e.g. HAdd) to avoid confusing type inference behavior.

def

```
Fin.add {n : Nat} : Fin n → Fin n → Fin n
```

Addition modulo *n*, usually invoked via the + operator.

Examples:

- `(2 : Fin 8) + (2 : Fin 8) = (4 : Fin 8)`
- `(2 : Fin 3) + (2 : Fin 3) = (1 : Fin 3)`

def

```
Fin.natAdd {m : Nat} (n : Nat) (i : Fin m) : Fin (n + m)
```

Adds a natural number to a Fin, increasing the bound.

This is a generalization of Fin.succ.

Fin.addNat is a version of this function that takes its Nat parameter second.

Examples:

- `Fin.natAdd 3 (5 : Fin 8) = (8 : Fin 11)`
- `Fin.natAdd 1 (0 : Fin 8) = (1 : Fin 9)`
- `Fin.natAdd 1 (2 : Fin 8) = (3 : Fin 9)`

def

```
Fin.addNat {n : Nat} (i : Fin n) (m : Nat) : Fin (n + m)
```

Adds a natural number to a Fin, increasing the bound.

This is a generalization of Fin.succ.

Fin.natAdd is a version of this function that takes its Nat parameter first.

Examples:

- `Fin.addNat (5 : Fin 8) 3 = (8 : Fin 11)`
- `Fin.addNat (0 : Fin 8) 1 = (1 : Fin 9)`
- `Fin.addNat (1 : Fin 8) 2 = (3 : Fin 10)`

def

```
Fin.mul {n : Nat} : Fin n → Fin n → Fin n
```

Multiplication modulo *n*, usually invoked via the * operator.

Examples:

- `(2 : Fin 10) * (2 : Fin 10) = (4 : Fin 10)`
- `(2 : Fin 10) * (7 : Fin 10) = (4 : Fin 10)`
- `(3 : Fin 10) * (7 : Fin 10) = (1 : Fin 10)`

def

```
Fin.sub {n : Nat} : Fin n → Fin n → Fin n
```

Subtraction modulo *n*, usually invoked via the − operator.

Examples:

- `(5 : Fin 11) - (3 : Fin 11) = (2 : Fin 11)`
- `(3 : Fin 11) - (5 : Fin 11) = (9 : Fin 11)`

def

```
Fin.subNat {n : Nat} (m : Nat) (i : Fin (n + m)) (h : m ≤ ↑i) : Fin n
```

Subtraction of a natural number from a Fin, with the bound narrowed.

This is a generalization of Fin.pred. It is guaranteed to not underflow or wrap around.

Examples:

- `(5 : Fin 9).subNat 2 (by decide) = (3 : Fin 7)`
- `(5 : Fin 9).subNat 0 (by decide) = (5 : Fin 9)`
- `(3 : Fin 9).subNat 3 (by decide) = (0 : Fin 6)`

def

```
Fin.div {n : Nat} : Fin n → Fin n → Fin n
```

Division of bounded numbers, usually invoked via the / operator.

The resulting value is that computed by the / operator on Nat. In particular, the result of division by 0 is 0.

Examples:

- `(5 : Fin 10) / (2 : Fin 10) = (2 : Fin 10)`

- `(5 : Fin 10) / (0 : Fin 10) = (0 : Fin 10)`
- `(5 : Fin 10) / (7 : Fin 10) = (0 : Fin 10)`

def

`Fin.mod {n : Nat} : Fin n → Fin n → Fin n`

Modulus of bounded numbers, usually invoked via the `%` operator.

The resulting value is that computed by the `%` operator on `Nat`.

def

`Fin.modn {n : Nat} : Fin n → Nat → Fin n`

Modulus of bounded numbers with respect to a `Nat`.

The resulting value is that computed by the `%` operator on `Nat`.

def

`Fin.log2 {m : Nat} (n : Fin m) : Fin m`

Logarithm base 2 for bounded numbers.

The resulting value is the same as that computed by `Nat.log2`. In particular, the result for `0` is `0`.

Examples:

- `(8 : Fin 10).log2 = (3 : Fin 10)`
- `(7 : Fin 10).log2 = (2 : Fin 10)`
- `(4 : Fin 10).log2 = (2 : Fin 10)`
- `(3 : Fin 10).log2 = (1 : Fin 10)`
- `(1 : Fin 10).log2 = (0 : Fin 10)`
- `(0 : Fin 10).log2 = (0 : Fin 10)`

19.3.3.3. Bitwise Operations

Typically, bitwise operations on `Fin` should be accessed using Lean's overloaded bitwise operators, particularly via the instances `ShiftLeft (Fin n)`, `ShiftRight (Fin n)`, `AndOp (Fin n)`, `OrOp (Fin n)`, `Xor (Fin n)`

def

`Fin.shiftLeft {n : Nat} : Fin n → Fin n → Fin n`

Bitwise left shift of bounded numbers, with wraparound on overflow.

Examples:

- `(1 : Fin 10) <<< (1 : Fin 10) = (2 : Fin 10)`
- `(1 : Fin 10) <<< (3 : Fin 10) = (8 : Fin 10)`
- `(1 : Fin 10) <<< (4 : Fin 10) = (6 : Fin 10)`

def

```
Fin.shiftRight {n : Nat} : Fin n → Fin n → Fin n
```

Bitwise right shift of bounded numbers.

This operator corresponds to logical rather than arithmetic bit shifting. The new bits are always `0`.

Examples:

- `(15 : Fin 16) >>> (1 : Fin 16) = (7 : Fin 16)`
- `(15 : Fin 16) >>> (2 : Fin 16) = (3 : Fin 16)`
- `(15 : Fin 17) >>> (2 : Fin 17) = (3 : Fin 17)`

def

```
Fin.land {n : Nat} : Fin n → Fin n → Fin n
```

Bitwise and.

def

```
Fin.lor {n : Nat} : Fin n → Fin n → Fin n
```

Bitwise or.

def

```
Fin.xor {n : Nat} : Fin n → Fin n → Fin n
```

Bitwise xor (“exclusive or”).

19.3.3.4. Conversions

def

```
Fin.toNat {n : Nat} (i : Fin n) : Nat
```

Extracts the underlying `Nat` value.

This function is a synonym for `Fin.val`, which is the simp normal form. `Fin.val` is also a coercion, so values of type `Fin n` are automatically converted to `Nat`s as needed.

def

```
Fin.ofNat (n : Nat) [NeZero n] (a : Nat) : Fin n
```

Returns `a` modulo `n` as a `Fin n`.

The assumption `NeZero n` ensures that `Fin n` is nonempty.

def

```
Fin.cast {n m : Nat} (eq : n = m) (i : Fin n) : Fin m
```

Uses a proof that two bounds are equal to allow a value bounded by one to be used with the other.

In other words, when `eq : n = m`, `Fin.cast eq i` converts `i : Fin n` into a `Fin m`.

def

```
Fin.castLT {n m : Nat} (i : Fin m) (h : ↑i ≤ n) : Fin n
```

Replaces the bound with another that is suitable for the value.

The proof embedded in `i` can be used to cast to a larger bound even if the concrete value is not known.

Examples:

```
example : Fin 12 := (7 : Fin 10).castLT (by decide : 7 < 12)

example (i : Fin 10) : Fin 12 :=
  i.castLT <| by
    cases i; simp; omega
```

def

```
Fin.castLE {n m : Nat} (h : n ≤ m) (i : Fin n) : Fin m
```

Coarsens a bound to one at least as large.

See also `Fin.castAdd` for a version that represents the larger bound with addition rather than an explicit inequality proof.

def

```
Fin.castAdd {n : Nat} (m : Nat) : Fin n → Fin (n + m)
```

Coarsens a bound to one at least as large.

See also `Fin.natAdd` and `Fin.addNat` for addition functions that increase the bound, and `Fin.castLE` for a version that uses an explicit inequality proof.

def

```
Fin.castSucc {n : Nat} : Fin n → Fin (n + 1)
```

Coarsens a bound by one.

def

```
Fin.rev {n : Nat} (i : Fin n) : Fin n
```

Replaces a value with its difference from the largest value in the type.

Considering the values of `Fin n` as a sequence `0`, `1`, …, $n-2$, $n-1$, `Fin.rev` finds the corresponding element of the reversed sequence. In other words, it maps `0` to $n-1$, `1` to $n-2$, ..., and $n-1$ to `0`.

Examples:

- `(5 : Fin 6).rev = (0 : Fin 6)`
- `(0 : Fin 6).rev = (5 : Fin 6)`
- `(2 : Fin 5).rev = (2 : Fin 5)`

def

```
Fin.elim0.{u} {α : Sort u} : Fin 0 → α
```

The type `Fin 0` is uninhabited, so it can be used to derive any result whatsoever.

This is similar to `Empty.elim`. It can be thought of as a compiler-checked assertion that a code path is unreachable, or a logical contradiction from which `False` and thus anything else could be derived.

19.3.3.5. Iteration

def

```
Fin.foldr.{u_1} {α : Sort u_1} (n : Nat) (f : Fin n → α → α)
  (init : α) : α
```

Combine all the values that can be represented by `Fin n` with an initial value, starting at `n - 1` and nesting to the right.

Example:

- `Fin.foldr 3 (·.val + ·) (0 : Nat) = (0 : Fin 3).val + ((1 : Fin 3).val + ((2 : Fin 3).val + 0))`

def

```
Fin.foldrM.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type u_1}
[Monad m]
  (n : Nat) (f : Fin n → α → m α) (init : α) : m α
```

Folds a monadic function over `Fin n` from right to left, starting with `n-1`.

It is the sequence of steps:

```
Fin.foldrM n f xₙ = do
  let xₙ₋₁ ← f (n-1) xₙ
  let xₙ₋₂ ← f (n-2) xₙ₋₁
  ...
  let x₀ ← f 0 x₁
  pure x₀
```

def

```
Fin.foldl.{u_1} {α : Sort u_1} (n : Nat) (f : α → Fin n → α)
  (init : α) : α
```

Combine all the values that can be represented by `Fin n` with an initial value, starting at `0` and nesting to the left.

Example:

- `Fin.foldl 3 (· + ·.val) (0 : Nat) = ((0 + (0 : Fin 3).val) + (1 : Fin 3).val) + (2 : Fin 3).val`

def

```
Fin.foldlM.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type u_1}
[Monad m]
  (n : Nat) (f : α → Fin n → m α) (init : α) : m α
```

Folds a monadic function over all the values in `Fin n` from left to right, starting with `0`.

It is the sequence of steps:

```
Fin.foldlM n f x₀ = do
  let x₁ ← f x₀ 0
  let x₂ ← f x₁ 1
  ...
  let xₙ ← f xₙ₋₁ (n-1)
  pure xₙ
```

def

```
Fin.hIterate.{u_1} (P : Nat → Sort u_1) {n : Nat} (init : P 0)
  (f : (i : Fin n) → P ↑i → P (↑i + 1)) : P n
```

Applies an index-dependent function to all the values less than the given bound *n*, starting at `0` with an accumulator.

Concretely, `Fin.hIterate P init f` is equal to

```
init |> f 0 |> f 1 |> ... |> f (n-1)
```

Theorems about `Fin.hIterate` can be proven using the general theorem `Fin.hIterate_elim` or other more specialized theorems.

`Fin.hIterateFrom` is a variant that takes a custom starting value instead of `0`.

def

```
Fin.hIterateFrom.{u_1} (P : Nat → Sort u_1) {n : Nat}
  (f : (i : Fin n) → P ↑i → P (↑i + 1)) (i : Nat) (ubnd : i ≤ n)
  (a : P i) : P n
```

Applies an index-dependent function *f* to all of the values in `[i:n]`, starting at `i` with an initial accumulator *a*.

Concretely, `Fin.hIterateFrom P f i a` is equal to

```
a |> f i |> f (i + 1) |> ... |> f (n - 1)
```

Theorems about `Fin.hIterateFrom` can be proven using the general theorem `Fin.hIterateFrom_elim` or other more specialized theorems.

`Fin.hIterate` is a variant that always starts at `0`.

19.3.3.6. Reasoning

def

```
Fin.induction.{u_1} {n : Nat} {motive : Fin (n + 1) → Sort u_1}
  (zero : motive 0)
  (succ : (i : Fin n) → motive i.castSucc → motive i.succ)
  (i : Fin (n + 1)) : motive i
```

Proves a statement by induction on the underlying `Nat` value in a `Fin (n + 1)`.

For the induction:

- *zero* is the base case, demonstrating *motive* `0`.
- *succ* is the inductive step, assuming the motive for `i : Fin n` (lifted to `Fin (n + 1)` with `Fin.castSucc`) and demonstrating it for *i*`.succ`.

`Fin.inductionOn` is a version of this induction principle that takes the `Fin` as its first parameter, `Fin.cases` is the corresponding case analysis operator, and `Fin.reverseInduction` is a version that starts at the greatest value instead of `0`.

def

```
Fin.inductionOn.{u_1} {n : Nat} (i : Fin (n + 1))
  {motive : Fin (n + 1) → Sort u_1} (zero : motive 0)
  (succ : (i : Fin n) → motive i.castSucc → motive i.succ) : motive
i
```

Proves a statement by induction on the underlying `Nat` value in a `Fin (n + 1)`.

For the induction:

- *zero* is the base case, demonstrating *motive* `0`.
- *succ* is the inductive step, assuming the motive for `i : Fin n` (lifted to `Fin (n + 1)` with `Fin.castSucc`) and demonstrating it for *i*`.succ`.

`Fin.induction` is a version of this induction principle that takes the `Fin` as its last parameter.

def

```
Fin.reverseInduction.{u_1} {n : Nat} {motive : Fin (n + 1) → Sort
u_1}
  (last : motive (Fin.last n))
  (cast : (i : Fin n) → motive i.succ → motive i.castSucc)
  (i : Fin (n + 1)) : motive i
```

Proves a statement by reverse induction on the underlying `Nat` value in a `Fin (n + 1)`.

For the induction:

- *last* is the base case, demonstrating *motive* `(Fin.last n)`.
- `cast` is the inductive step, assuming the motive for `(`*j* `: Fin n).succ` and demonstrating it for the predecessor *j*`.castSucc`.

`Fin.induction` is the non-reverse induction principle.

def

```
Fin.cases.{u_1} {n : Nat} {motive : Fin (n + 1) → Sort u_1}
  (zero : motive 0) (succ : (i : Fin n) → motive i.succ)
  (i : Fin (n + 1)) : motive i
```

Proves a statement by cases on the underlying `Nat` value in a `Fin (n + 1)`.

The two cases are:

- *zero*, used when the value is of the form `(0 : Fin (n + 1))`
- *succ*, used when the value is of the form `(j : Fin n).succ`

The corresponding induction principle is `Fin.induction`.

def

```
Fin.lastCases.{u_1} {n : Nat} {motive : Fin (n + 1) → Sort u_1}
  (last : motive (Fin.last n)) (cast : (i : Fin n) → motive
i.castSucc)
  (i : Fin (n + 1)) : motive i
```

Proves a statement by cases on the underlying `Nat` value in a `Fin (n + 1)`, checking whether the value is the greatest representable or a predecessor of some other.

The two cases are:

- *last*, used when the value is `Fin.last n`
- `cast`, used when the value is of the form `(j : Fin n).succ`

The corresponding induction principle is `Fin.reverseInduction`.

def

```
Fin.addCases.{u} {m n : Nat} {motive : Fin (m + n) → Sort u}
  (left : (i : Fin m) → motive (Fin.castAdd n i))
  (right : (i : Fin n) → motive (Fin.natAdd m i)) (i : Fin (m + n))
:
  motive i
```

A case analysis operator for i : `Fin (m + n)` that separately handles the cases where i < m and where m ≤ i < m + n.

The first case, where i < m, is handled by *left*. In this case, i can be represented as `Fin.castAdd n (j : Fin m)`.

The second case, where m ≤ i < m + n, is handled by *right*. In this case, i can be represented as `Fin.natAdd m (j : Fin n)`.

def

```
Fin.succRec.{u_1} {motive : (n : Nat) → Fin n → Sort u_1}
  (zero : (n : Nat) → motive n.succ 0)
  (succ : (n : Nat) → (i : Fin n) → motive n i → motive n.succ
i.succ)
  {n : Nat} (i : Fin n) : motive n i
```

An induction principle for `Fin` that considers a given `i : Fin n` as given by a sequence of `i` applications of `Fin.succ`.

The cases in the induction are:

- `zero` demonstrates the motive for `(0 : Fin (n + 1))` for all bounds `n`
- `succ` demonstrates the motive for `Fin.succ` applied to an arbitrary `Fin` for an arbitrary bound `n`

Unlike `Fin.induction`, the motive quantifies over the bound, and the bound varies at each inductive step. `Fin.succRecOn` is a version of this induction principle that takes the `Fin` argument first.

def

```
Fin.succRecOn.{u_1} {n : Nat} (i : Fin n)
  {motive : (n : Nat) → Fin n → Sort u_1}
  (zero : (n : Nat) → motive (n + 1) 0)
  (succ : (n : Nat) → (i : Fin n) → motive n i → motive n.succ
i.succ) :
  motive n i
```

An induction principle for `Fin` that considers a given `i : Fin n` as given by a sequence of `i` applications of `Fin.succ`.

The cases in the induction are:

- `zero` demonstrates the motive for `(0 : Fin (n + 1))` for all bounds `n`
- `succ` demonstrates the motive for `Fin.succ` applied to an arbitrary `Fin` for an arbitrary bound `n`

Unlike `Fin.induction`, the motive quantifies over the bound, and the bound varies at each inductive step. `Fin.succRec` is a version of this induction principle that takes the `Fin` argument last.

19.4. Fixed-Precision Integers

Lean's standard library includes the usual assortment of fixed-width integer types. From the perspective of formalization and proofs, these types are wrappers around bitvectors of the appropriate size; the wrappers ensure that the correct implementations of e.g. arithmetic operations

are applied. In compiled code, they are represented efficiently: the compiler has special support for them, as it does for other fundamental types.

19.4.1. Logical Model

Fixed-width integers may be unsigned or signed. Furthermore, they are available in five sizes: 8, 16, 32, and 64 bits, along with the current architecture's word size. In their logical models, the unsigned integers are structures that wrap a `BitVec` of the appropriate width. Signed integers wrap the corresponding unsigned integers, and use a twos-complement representation.

19.4.1.1. Unsigned

structure

```
USize : Type
```

Unsigned integers that are the size of a word on the platform's architecture.

On a 32-bit architecture, `USize` is equivalent to `UInt32`. On a 64-bit machine, it is equivalent to `UInt64`.

Constructor

`USize.ofBitVec`

Creates a `USize` from a `BitVec System.Platform.numBits`. This function is overridden with a native implementation.

Fields

`toBitVec : BitVec System.Platform.numBits`

Unpacks a `USize` into a `BitVec System.Platform.numBits`. This function is overridden with a native implementation.

structure

```
UInt8 : Type
```

Unsigned 8-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 8-bit value rather than wrapping a `BitVec 8`.

Constructor

`UInt8.ofBitVec`

Creates a `UInt8` from a `BitVec 8`. This function is overridden with a native implementation.

Fields

`toBitVec : BitVec 8`

Unpacks a `UInt8` into a `BitVec 8`. This function is overridden with a native implementation.

structure

`UInt16 : Type`

Unsigned 16-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 16-bit value rather than wrapping a `BitVec 16`.

Constructor

`UInt16.ofBitVec`

Creates a `UInt16` from a `BitVec 16`. This function is overridden with a native implementation.

Fields

`toBitVec : BitVec 16`

Unpacks a `UInt16` into a `BitVec 16`. This function is overridden with a native implementation.

structure

`UInt32 : Type`

Unsigned 32-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 32-bit value rather than wrapping a `BitVec 32`.

Constructor

`UInt32.ofBitVec`

Creates a `UInt32` from a `BitVec 32`. This function is overridden with a native implementation.

Fields

`toBitVec : BitVec 32`

Unpacks a `UInt32` into a `BitVec 32`. This function is overridden with a native implementation.

structure

`UInt64 : Type`

Unsigned 64-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 64-bit value rather than wrapping a `BitVec 64`.

Constructor

`UInt64.ofBitVec`

Creates a `UInt64` from a `BitVec 64`. This function is overridden with a native implementation.

Fields

`toBitVec : BitVec 64`

Unpacks a `UInt64` into a `BitVec 64`. This function is overridden with a native implementation.

19.4.1.2. Signed

structure

`ISize : Type`

Signed integers that are the size of a word on the platform's architecture.

On a 32-bit architecture, `ISize` is equivalent to `Int32`. On a 64-bit machine, it is equivalent to `Int64`. This type has special support in the compiler so it can be represented by an unboxed value.

Fields

`toUSize : USize`

Converts a word-sized signed integer into the word-sized unsigned integer that is its two's complement encoding.

structure

```
Int8 : Type
```

Signed 8-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 8-bit value.

Fields

`toUInt8 : UInt8`

Converts an 8-bit signed integer into the 8-bit unsigned integer that is its two's complement encoding.

structure

```
Int16 : Type
```

Signed 16-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 16-bit value.

Fields

`toUInt16 : UInt16`

Converts an 16-bit signed integer into the 16-bit unsigned integer that is its two's complement encoding.

structure

```
Int32 : Type
```

Signed 32-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 32-bit value.

Fields

`toUInt32 : UInt32`

Converts an 32-bit signed integer into the 32-bit unsigned integer that is its two's complement encoding.

structure

`Int64 : Type`

Signed 64-bit integers.

This type has special support in the compiler so it can be represented by an unboxed 64-bit value.

Fields

`toUInt64 : UInt64`

Converts an 64-bit signed integer into the 64-bit unsigned integer that is its two's complement encoding.

19.4.2. Run-Time Representation

In compiled code in contexts that require boxed representations, fixed-width integer types that fit in one less bit than the platform's pointer size are always represented without additional allocations or indirections. This always includes `Int8`, `UInt8`, `Int16`, and `UInt16`. On 64-bit architectures, `Int32` and `UInt32` are also represented without pointers. On 32-bit architectures, `Int32` and `UInt32` require a pointer to an object on the heap. `ISize`, `USize`, `Int64` and `UInt64` may require pointers on all architectures.

Even though some fixed-with integer types require boxing in general, the compiler is able to represent them without boxing or pointer indirections in code paths that use only a specific fixed-width type rather than being polymorphic, potentially after a specialization pass. This applies in most practical situations where these types are used: their values are represented using the corresponding unsigned fixed-width C type when a constructor parameter, function parameter, function return value, or intermediate result is known to be a fixed-width integer type. The Lean run-time system includes primitives for storing fixed-width integers in constructors of inductive types, and the primitive operations are defined on the corresponding C types, so boxing tends to happen at the “edges” of integer calculations rather than for each intermediate result. In contexts where other types might occur, such as the contents of polymorphic containers like `Array`, these types are boxed, even if an array is statically known to contain only a single fixed-width integer type. (i) The monomorphic array type `ByteArray` avoids boxing for arrays of `UInt8`. Lean does not specialize the representation of inductive types or arrays. Inspecting a function's type in Lean is not sufficient to determine how fixed-width integer values will be represented, because boxed values are not eagerly unboxed—a function that projects an `Int64` from an array returns a boxed integer value.

19.4.3. Syntax

All the fixed-width integer types have `OfNat` instances, which allow numerals to be used as literals, both in expression and in pattern contexts. The signed types additionally have `Neg` instances, allowing negation to be applied.

▼ *Example: Fixed-Width Literals*

Lean allows both decimal and hexadecimal literals to be used for types with `OfNat` instances. In this example, literal notation is used to define masks.

```
structure Permissions where
  readable : Bool
  writable : Bool
  executable : Bool

def Permissions.encode (p : Permissions) : UInt8 :=
  let r := if p.readable then 0x01 else 0
  let w := if p.writable then 0x02 else 0
  let x := if p.executable then 0x04 else 0
  r ||| w ||| x

def Permissions.decode (i : UInt8) : Permissions :=
  ⟨i &&& 0x01 ≠ 0, i &&& 0x02 ≠ 0, i &&& 0x04 ≠ 0⟩
```

Literals that overflow their types' precision are interpreted modulus the precision. Signed types, are interpreted according to the underlying twos-complement representation.

▼ *Example: Overflowing Fixed-Width Literals*

The following statements are all true:

```
example : (255 : UInt8) = 255 := by rfl
example : (256 : UInt8) = 0   := by rfl
example : (257 : UInt8) = 1   := by rfl

example : (0x7f : Int8) = 127  := by rfl
example : (0x8f : Int8) = -113 := by rfl
example : (0xff : Int8) = -1   := by rfl
```

19.4.4. API Reference

19.4.4.1. Sizes

Each fixed-width integer has a *size*, which is the number of distinct values that can be represented by the type. This is not equivalent to C's `sizeof` operator, which instead determines how many bytes the type occupies.

def

```
USize.size : Nat
```

The number of distinct values representable by `USize`, that is, `2^System.Platform.numBits`.

def

```
ISize.size : Nat
```

The number of distinct values representable by `ISize`, that is, `2^System.Platform.numBits`.

def

```
UInt8.size : Nat
```

The number of distinct values representable by `UInt8`, that is, `2^8 = 256`.

def

```
Int8.size : Nat
```

The number of distinct values representable by `Int8`, that is, `2^8 = 256`.

def

```
UInt16.size : Nat
```

The number of distinct values representable by `UInt16`, that is, `2^16 = 65536`.

def

```
Int16.size : Nat
```

The number of distinct values representable by `Int16`, that is, `2^16 = 65536`.

def

```
UInt32.size : Nat
```

The number of distinct values representable by `UInt32`, that is, `2^32 = 4294967296`.

def

```
Int32.size : Nat
```

The number of distinct values representable by `Int32`, that is, `2^32 = 4294967296`.

def

```
UInt64.size : Nat
```

The number of distinct values representable by `UInt64`, that is, `2^64 = 18446744073709551616`.

def

```
Int64.size : Nat
```

The number of distinct values representable by `Int64`, that is, `2^64 = 18446744073709551616`.

19.4.4.2. Ranges

def

```
ISize.minValue : ISize
```

The smallest number that `ISize` can represent: `-2^(System.Platform.numBits - 1)`.

def

```
ISize.maxValue : ISize
```

The largest number that `ISize` can represent: `2^(System.Platform.numBits - 1) - 1`.

def

```
Int8.minValue : Int8
```

The smallest number that `Int8` can represent: `-2^7 = -128`.

def

```
Int8.maxValue : Int8
```

The largest number that `Int8` can represent: `2^7 - 1 = 127`.

def

```
Int16.minValue : Int16
```

The smallest number that `Int16` can represent: `-2^15 = -32768`.

def

```
Int16.maxValue : Int16
```

The largest number that `Int16` can represent: `2^15 - 1 = 32767`.

def

```
Int32.minValue : Int32
```

The smallest number that `Int32` can represent: `-2^31 = -2147483648`.

def

```
Int32.maxValue : Int32
```

The largest number that `Int32` can represent: `2^31 - 1 = 2147483647`.

def

```
Int64.minValue : Int64
```

The smallest number that `Int64` can represent: `-2^63 = -9223372036854775808`.

def

```
Int64.maxValue : Int64
```

The largest number that `Int64` can represent: `2^63 - 1 = 9223372036854775807`.

19.4.4.3. Conversions

19.4.4.3.1. To and From `Int`

def

```
ISize.toInt (i : ISize) : Int
```

Converts a word-sized signed integer to an arbitrary-precision integer that denotes the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.toInt (i : Int8) : Int
```

Converts an 8-bit signed integer to an arbitrary-precision integer that denotes the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.toInt (i : Int16) : Int
```

Converts a 16-bit signed integer to an arbitrary-precision integer that denotes the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.toInt (i : Int32) : Int
```

Converts a 32-bit signed integer to an arbitrary-precision integer that denotes the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.toInt (i : Int64) : Int
```

Converts a 64-bit signed integer to an arbitrary-precision integer that denotes the same number.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.ofInt (i : Int) : ISize
```

Converts an arbitrary-precision integer to a word-sized signed integer, wrapping around on over- or underflow.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.ofInt (i : Int) : Int8
```

Converts an arbitrary-precision integer to an 8-bit integer, wrapping on overflow or underflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int8.ofInt 48 = 48`

- `Int8.ofInt (-115) = -115`
- `Int8.ofInt (-129) = 127`
- `Int8.ofInt (128) = -128`

def

`Int16.ofInt (i : Int) : Int16`

Converts an arbitrary-precision integer to a 16-bit signed integer, wrapping on overflow or underflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int16.ofInt 48 = 48`
- `Int16.ofInt (-129) = -129`
- `Int16.ofInt (128) = 128`
- `Int16.ofInt 70000 = 4464`
- `Int16.ofInt (-40000) = 25536`

def

`Int32.ofInt (i : Int) : Int32`

Converts an arbitrary-precision integer to a 32-bit integer, wrapping on overflow or underflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int32.ofInt 48 = 48`
- `Int32.ofInt (-129) = -129`
- `Int32.ofInt 70000 = 70000`
- `Int32.ofInt (-40000) = -40000`
- `Int32.ofInt 2147483648 = -2147483648`
- `Int32.ofInt (-2147483649) = 2147483647`

def

`Int64.ofInt (i : Int) : Int64`

Converts an arbitrary-precision integer to a 64-bit integer, wrapping on overflow or underflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int64.ofInt 48 = 48`
- `Int64.ofInt (-40_000) = -40_000`
- `Int64.ofInt 2_147_483_648 = 2_147_483_648`
- `Int64.ofInt (-2_147_483_649) = -2_147_483_649`
- `Int64.ofInt 9_223_372_036_854_775_808 = -9_223_372_036_854_775_808`
- `Int64.ofInt (-9_223_372_036_854_775_809) = 9_223_372_036_854_775_807`

def

```
ISize.ofIntTruncate (i : Int) : ISize
```

Constructs an `ISize` from an `Int`, clamping if the value is too small or too large.

def

```
Int8.ofIntTruncate (i : Int) : Int8
```

Constructs an `Int8` from an `Int`, clamping if the value is too small or too large.

def

```
Int16.ofIntTruncate (i : Int) : Int16
```

Constructs an `Int16` from an `Int`, clamping if the value is too small or too large.

def

```
Int32.ofIntTruncate (i : Int) : Int32
```

Constructs an `Int32` from an `Int`, clamping if the value is too small or too large.

def

```
Int64.ofIntTruncate (i : Int) : Int64
```

Constructs an `Int64` from an `Int`, clamping if the value is too small or too large.

def

```
ISize.ofIntLE (i : Int) (_hl : ISize.minValue.toInt ≤ i)
  (_hr : i ≤ ISize.maxValue.toInt) : ISize
```

Constructs an `ISize` from an `Int` that is known to be in bounds.

def

```
Int8.ofIntLE (i : Int) (_hl : Int8.minValue.toInt ≤ i)
  (_hr : i ≤ Int8.maxValue.toInt) : Int8
```

Constructs an `Int8` from an `Int` that is known to be in bounds.

def

```
Int16.ofIntLE (i : Int) (_hl : Int16.minValue.toInt ≤ i)
  (_hr : i ≤ Int16.maxValue.toInt) : Int16
```

Constructs an `Int16` from an `Int` that is known to be in bounds.

def

```
Int32.ofIntLE (i : Int) (_hl : Int32.minValue.toInt ≤ i)
  (_hr : i ≤ Int32.maxValue.toInt) : Int32
```

Constructs an `Int32` from an `Int` that is known to be in bounds.

def

```
Int64.ofIntLE (i : Int) (_hl : Int64.minValue.toInt ≤ i)
  (_hr : i ≤ Int64.maxValue.toInt) : Int64
```

Constructs an `Int64` from an `Int` that is known to be in bounds.

19.4.4.3.2. To and From `Nat`

def

```
USize.ofNat (n : Nat) : USize
```

Converts an arbitrary-precision natural number to an unsigned word-sized integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.ofNat (n : Nat) : ISize
```

Converts an arbitrary-precision natural number to a word-sized signed integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

def

`UInt8.ofNat (n : Nat) : UInt8`

Converts a natural number to an 8-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt8.ofNat 5 = 5`
- `UInt8.ofNat 255 = 255`
- `UInt8.ofNat 256 = 0`
- `UInt8.ofNat 259 = 3`
- `UInt8.ofNat 32770 = 2`

def

`Int8.ofNat (n : Nat) : Int8`

Converts a natural number to an 8-bit signed integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int8.ofNat 53 = 53`
- `Int8.ofNat 127 = 127`
- `Int8.ofNat 128 = -128`
- `Int8.ofNat 255 = -1`

def

`UInt16.ofNat (n : Nat) : UInt16`

Converts a natural number to a 16-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt16.ofNat 5 = 5`
- `UInt16.ofNat 255 = 255`
- `UInt16.ofNat 32770 = 32770`
- `UInt16.ofNat 65537 = 1`

def

`Int16.ofNat (n : Nat) : Int16`

Converts a natural number to a 16-bit signed integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int16.ofNat 127 = 127`
- `Int16.ofNat 32767 = 32767`
- `Int16.ofNat 32768 = -32768`
- `Int16.ofNat 32770 = -32766`

def

`UInt32.ofNat (n : Nat) : UInt32`

Converts a natural number to a 32-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt32.ofNat 5 = 5`
- `UInt32.ofNat 65539 = 65539`
- `UInt32.ofNat 4_294_967_299 = 3`

def

`Int32.ofNat (n : Nat) : Int32`

Converts a natural number to a 32-bit signed integer, wrapping around on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int32.ofNat 127 = 127`
- `Int32.ofNat 32770 = 32770`

- `Int32.ofNat 2_147_483_647 = 2_147_483_647`
- `Int32.ofNat 2_147_483_648 = -2_147_483_648`

def

`UInt64.ofNat (n : Nat) : UInt64`

Converts a natural number to a 64-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt64.ofNat 5 = 5`
- `UInt64.ofNat 65539 = 65539`
- `UInt64.ofNat 4_294_967_299 = 4_294_967_299`
- `UInt64.ofNat 18_446_744_073_709_551_620 = 4`

def

`Int64.ofNat (n : Nat) : Int64`

Converts a natural number to a 64-bit signed integer, wrapping around to negative numbers on overflow.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int64.ofNat 127 = 127`
- `Int64.ofNat 2_147_483_648 = 2_147_483_648`
- `Int64.ofNat 9_223_372_036_854_775_807 = 9_223_372_036_854_775_807`
- `Int64.ofNat 9_223_372_036_854_775_808 = -9_223_372_036_854_775_808`
- `Int64.ofNat 18_446_744_073_709_551_618 = 0`

def

`USize.ofNat32 (n : Nat) (h : n ≤ 4294967296) : USize`

Converts a natural number to a `USize`. Overflow is impossible on any supported platform because `USize.size` is either `2^32` or `2^64`.

This function is overridden at runtime with an efficient implementation.

def

```
USize.ofNatLT (n : Nat) (h : n < USize.size) : USize
```

Converts a natural number to a `USize`. Requires a proof that the number is small enough to be representable without overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.ofNatLT (n : Nat) (h : n < UInt8.size) : UInt8
```

Converts a natural number to an 8-bit unsigned integer. Requires a proof that the number is small enough to be representable without overflow; it must be smaller than `2^8`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.ofNatLT (n : Nat) (h : n < UInt16.size) : UInt16
```

Converts a natural number to a 16-bit unsigned integer. Requires a proof that the number is small enough to be representable without overflow; it must be smaller than `2^16`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.ofNatLT (n : Nat) (h : n < UInt32.size) : UInt32
```

Converts a natural number to a 32-bit unsigned integer. Requires a proof that the number is small enough to be representable without overflow; it must be smaller than `2^32`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.ofNatLT (n : Nat) (h : n < UInt64.size) : UInt64
```

Converts a natural number to a 64-bit unsigned integer. Requires a proof that the number is small enough to be representable without overflow; it must be smaller than `2^64`.

This function is overridden at runtime with an efficient implementation.

def

```
USize.ofNatTruncate (n : Nat) : USize
```

Converts a natural number to `USize`, returning the largest representable value if the number is too large.

Returns `USize.size - 1`, which is `2^64 - 1` or `2^32 - 1` depending on the platform, for natural numbers greater than or equal to `USize.size`.

def

```
UInt8.ofNatTruncate (n : Nat) : UInt8
```

Converts a natural number to an 8-bit unsigned integer, returning the largest representable value if the number is too large.

Returns `2^8 - 1` for natural numbers greater than or equal to `2^8`.

def

```
UInt16.ofNatTruncate (n : Nat) : UInt16
```

Converts a natural number to a 16-bit unsigned integer, returning the largest representable value if the number is too large.

Returns `2^16 - 1` for natural numbers greater than or equal to `2^16`.

def

```
UInt32.ofNatTruncate (n : Nat) : UInt32
```

Converts a natural number to a 32-bit unsigned integer, returning the largest representable value if the number is too large.

Returns `2^32 - 1` for natural numbers greater than or equal to `2^32`.

def

```
UInt64.ofNatTruncate (n : Nat) : UInt64
```

Converts a natural number to a 64-bit unsigned integer, returning the largest representable value if the number is too large.

Returns `2^64 - 1` for natural numbers greater than or equal to `2^64`.

def

```
USize.toNat (n : USize) : Nat
```

Converts a word-sized unsigned integer to an arbitrary-precision natural number.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.toNatClampNeg (i : ISize) : Nat
```

Converts a word-sized signed integer to a natural number, mapping all negative numbers to `0`.

Use `ISize.toBitVec` to obtain the two's complement representation.

def

```
UInt8.toNat (n : UInt8) : Nat
```

Converts an 8-bit unsigned integer to an arbitrary-precision natural number.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.toNatClampNeg (i : Int8) : Nat
```

Converts an 8-bit signed integer to a natural number, mapping all negative numbers to `0`.

Use `Int8.toBitVec` to obtain the two's complement representation.

def

```
UInt16.toNat (n : UInt16) : Nat
```

Converts a 16-bit unsigned integer to an arbitrary-precision natural number.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.toNatClampNeg (i : Int16) : Nat
```

Converts a 16-bit signed integer to a natural number, mapping all negative numbers to `0`.

Use `Int16.toBitVec` to obtain the two's complement representation.

def

```
UInt32.toNat (n : UInt32) : Nat
```

Converts a 32-bit unsigned integer to an arbitrary-precision natural number.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.toNatClampNeg (i : Int32) : Nat
```

Converts a 32-bit signed integer to a natural number, mapping all negative numbers to `0`.

Use `Int32.toBitVec` to obtain the two's complement representation.

def

```
UInt64.toNat (n : UInt64) : Nat
```

Converts a 64-bit unsigned integer to an arbitrary-precision natural number.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.toNatClampNeg (i : Int64) : Nat
```

Converts a 64-bit signed integer to a natural number, mapping all negative numbers to `0`.

Use `Int64.toBitVec` to obtain the two's complement representation.

19.4.4.3.3. To Other Fixed-Width Integers

def

```
USize.toUInt8 (a : USize) : UInt8
```

Converts word-sized unsigned integers to 8-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
USize.toUInt16 (a : USize) : UInt16
```

Converts word-sized unsigned integers to 16-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
USize.toUInt32 (a : USize) : UInt32
```

Converts word-sized unsigned integers to 32-bit unsigned integers. Wraps around on overflow, which might occur on 64-bit architectures.

This function is overridden at runtime with an efficient implementation.

def

```
USize.toUInt64 (a : USize) : UInt64
```

Converts word-sized unsigned integers to 32-bit unsigned integers. This cannot overflow because `USize.size` is either `2^32` or `2^64`.

This function is overridden at runtime with an efficient implementation.

def

```
USize.toISize (i : USize) : ISize
```

Obtains the `ISize` that is 2's complement equivalent to the `USize`.

def

```
UInt8.toInt8 (i : UInt8) : Int8
```

Obtains the `Int8` that is 2's complement equivalent to the `UInt8`.

def

```
UInt8.toUInt16 (a : UInt8) : UInt16
```

Converts 8-bit unsigned integers to 16-bit unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.toUInt32 (a : UInt8) : UInt32
```

Converts 8-bit unsigned integers to 32-bit unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.toUInt64 (a : UInt8) : UInt64
```

Converts 8-bit unsigned integers to 64-bit unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

`UInt8.toUSize (a : UInt8) : USize`

Converts 8-bit unsigned integers to word-sized unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

`UInt16.toUInt8 (a : UInt16) : UInt8`

Converts 16-bit unsigned integers to 8-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

`UInt16.toInt16 (i : UInt16) : Int16`

Obtains the `Int16` that is 2's complement equivalent to the `UInt16`.

def

`UInt16.toUInt32 (a : UInt16) : UInt32`

Converts 16-bit unsigned integers to 32-bit unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

`UInt16.toUInt64 (a : UInt16) : UInt64`

Converts 16-bit unsigned integers to 64-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

`UInt16.toUSize (a : UInt16) : USize`

Converts 16-bit unsigned integers to word-sized unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.toUInt8 (a : UInt32) : UInt8
```

Converts a 32-bit unsigned integer to an 8-bit unsigned integer, wrapping on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.toUInt16 (a : UInt32) : UInt16
```

Converts 32-bit unsigned integers to 16-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.toInt32 (i : UInt32) : Int32
```

Obtains the `Int32` that is 2's complement equivalent to the `UInt32`.

def

```
UInt32.toUInt64 (a : UInt32) : UInt64
```

Converts 32-bit unsigned integers to 64-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.toUSize (a : UInt32) : USize
```

Converts 32-bit unsigned integers to word-sized unsigned integers.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.toUInt8 (a : UInt64) : UInt8
```

Converts 64-bit unsigned integers to 8-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.toUInt16 (a : UInt64) : UInt16
```

Converts 64-bit unsigned integers to 16-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.toUInt32 (a : UInt64) : UInt32
```

Converts 64-bit unsigned integers to 32-bit unsigned integers. Wraps around on overflow.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.toInt64 (i : UInt64) : Int64
```

Obtains the `Int64` that is 2's complement equivalent to the `UInt64`.

def

```
UInt64.toUSize (a : UInt64) : USize
```

Converts 64-bit unsigned integers to word-sized unsigned integers. On 32-bit machines, this may overflow, which results in the value wrapping around (that is, it is reduced modulo `USize.size`).

This function is overridden at runtime with an efficient implementation.

def

```
ISize.toInt8 (a : ISize) : Int8
```

Converts a word-sized signed integer to an 8-bit signed integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.toInt16 (a : ISize) : Int16
```

Converts a word-sized integer to a 16-bit integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.toInt32 (a : ISize) : Int32
```

Converts a word-sized signed integer to a 32-bit signed integer.

On 32-bit platforms, this conversion is lossless. On 64-bit platforms, the integer's bitvector representation is truncated to 32 bits. This function is overridden at runtime with an efficient implementation.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.toInt64 (a : ISize) : Int64
```

Converts word-sized signed integers to 64-bit signed integers that denote the same number. This conversion is lossless, because `ISize` is either `Int32` or `Int64`.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.toInt16 (a : Int8) : Int16
```

Converts 8-bit signed integers to 16-bit signed integers that denote the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.toInt32 (a : Int8) : Int32
```

Converts 8-bit signed integers to 32-bit signed integers that denote the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.toInt64 (a : Int8) : Int64
```

Converts 8-bit signed integers to 64-bit signed integers that denote the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.toISize (a : Int8) : ISize
```

Converts 8-bit signed integers to word-sized signed integers that denote the same number. This conversion is lossless, because `ISize` is either `Int32` or `Int64`.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.toInt8 (a : Int16) : Int8
```

Converts 16-bit signed integers to 8-bit signed integers by truncating their bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.toInt32 (a : Int16) : Int32
```

Converts 8-bit signed integers to 32-bit signed integers that denote the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.toInt64 (a : Int16) : Int64
```

Converts 16-bit signed integers to 64-bit signed integers that denote the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.toISize (a : Int16) : ISize
```

Converts 16-bit signed integers to word-sized signed integers that denote the same number. This conversion is lossless, because `ISize` is either `Int32` or `Int64`.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.toInt8 (a : Int32) : Int8
```

Converts a 32-bit signed integer to an 8-bit signed integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.toInt16 (a : Int32) : Int16
```

Converts a 32-bit signed integer to an 16-bit signed integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.toInt64 (a : Int32) : Int64
```

Converts 32-bit signed integers to 64-bit signed integers that denote the same number.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.toISize (a : Int32) : ISize
```

Converts 32-bit signed integers to word-sized signed integers that denote the same number. This conversion is lossless, because `ISize` is either `Int32` or `Int64`.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.toInt8 (a : Int64) : Int8
```

Converts a 64-bit signed integer to an 8-bit signed integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.toInt16 (a : Int64) : Int16
```

Converts a 64-bit signed integer to a 16-bit signed integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.toInt32 (a : Int64) : Int32
```

Converts a 64-bit signed integer to a 32-bit signed integer by truncating its bitvector representation.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.toISize (a : Int64) : ISize
```

Converts 64-bit signed integers to word-sized signed integers, truncating the bitvector representation on 32-bit platforms. This conversion is lossless on 64-bit platforms.

This function is overridden at runtime with an efficient implementation.

19.4.4.3.4. To Floating-Point Numbers

opaque

```
ISize.toFloat (n : ISize) : Float
```

Obtains a `Float` whose value is near the given `ISize`.

It will be exactly the value of the given `ISize` if such a `Float` exists. If no such `Float` exists, the returned value will either be the smallest `Float` that is larger than the given value, or the largest `Float` that is smaller than the given value.

This function does not reduce in the kernel.

opaque

```
ISize.toFloat32 (n : ISize) : Float32
```

Obtains a `Float32` whose value is near the given `ISize`.

It will be exactly the value of the given `ISize` if such a `Float32` exists. If no such `Float32` exists, the returned value will either be the smallest `Float32` that is larger than the given value, or the largest `Float32` that is smaller than the given value.

This function does not reduce in the kernel.

opaque

```
Int8.toFloat (n : Int8) : Float
```

Obtains the `Float` whose value is the same as the given `Int8`.

This function does not reduce in the kernel.

opaque

```
Int8.toFloat32 (n : Int8) : Float32
```

Obtains the `Float32` whose value is the same as the given `Int8`.

This function does not reduce in the kernel.

opaque

```
Int16.toFloat (n : Int16) : Float
```

Obtains the `Float` whose value is the same as the given `Int16`.

This function does not reduce in the kernel.

opaque

```
Int16.toFloat32 (n : Int16) : Float32
```

Obtains the `Float32` whose value is the same as the given `Int16`.

This function does not reduce in the kernel.

opaque

```
Int32.toFloat (n : Int32) : Float
```

Obtains the `Float` whose value is the same as the given `Int32`.

This function does not reduce in the kernel.

opaque

```
Int32.toFloat32 (n : Int32) : Float32
```

Obtains a `Float32` whose value is near the given `Int32`.

It will be exactly the value of the given `Int32` if such a `Float32` exists. If no such `Float32` exists, the returned value will either be the smallest `Float32` that is larger than the given value, or the largest `Float32` that is smaller than the given value.

This function does not reduce in the kernel.

opaque

```
Int64.toFloat (n : Int64) : Float
```

Obtains a `Float` whose value is near the given `Int64`.

It will be exactly the value of the given `Int64` if such a `Float` exists. If no such `Float` exists, the returned value will either be the smallest `Float` that is larger than the given value, or the largest `Float` that is smaller than the given value.

This function does not reduce in the kernel.

opaque

`Int64.toFloat32 (n : Int64) : Float32`

Obtains a `Float32` whose value is near the given `Int64`.

It will be exactly the value of the given `Int64` if such a `Float32` exists. If no such `Float32` exists, the returned value will either be the smallest `Float32` that is larger than the given value, or the largest `Float32` that is smaller than the given value.

This function does not reduce in the kernel.

opaque

`USize.toFloat (n : USize) : Float`

Obtains a `Float` whose value is near the given `USize`.

It will be exactly the value of the given `USize` if such a `Float` exists. If no such `Float` exists, the returned value will either be the smallest `Float` that is larger than the given value, or the largest `Float` that is smaller than the given value.

This function is opaque in the kernel, but is overridden at runtime with an efficient implementation.

opaque

`USize.toFloat32 (n : USize) : Float32`

Obtains a `Float32` whose value is near the given `USize`.

It will be exactly the value of the given `USize` if such a `Float32` exists. If no such `Float32` exists, the returned value will either be the smallest `Float32` that is larger than the given value, or the largest `Float32` that is smaller than the given value.

This function is opaque in the kernel, but is overridden at runtime with an efficient implementation.

opaque

`UInt8.toFloat (n : UInt8) : Float`

Obtains the Float whose value is the same as the given UInt8.

opaque

```
UInt8.toFloat32 (n : UInt8) : Float32
```

Obtains the Float32 whose value is the same as the given UInt8.

opaque

```
UInt16.toFloat (n : UInt16) : Float
```

Obtains the Float whose value is the same as the given UInt16.

opaque

```
UInt16.toFloat32 (n : UInt16) : Float32
```

Obtains the Float32 whose value is the same as the given UInt16.

opaque

```
UInt32.toFloat (n : UInt32) : Float
```

Obtains the Float whose value is the same as the given UInt32.

opaque

```
UInt32.toFloat32 (n : UInt32) : Float32
```

Obtains a Float32 whose value is near the given UInt32.

It will be exactly the value of the given UInt32 if such a Float32 exists. If no such Float32 exists, the returned value will either be the smallest Float32 that is larger than the given value, or the largest Float32 that is smaller than the given value.

This function is opaque in the kernel, but is overridden at runtime with an efficient implementation.

opaque

```
UInt64.toFloat (n : UInt64) : Float
```

Obtains a Float whose value is near the given UInt64.

It will be exactly the value of the given `UInt64` if such a `Float` exists. If no such `Float` exists, the returned value will either be the smallest `Float` that is larger than the given value, or the largest `Float` that is smaller than the given value.

This function is opaque in the kernel, but is overridden at runtime with an efficient implementation.

opaque

```
UInt64.toFloat32 (n : UInt64) : Float32
```

Obtains a `Float32` whose value is near the given `UInt64`.

It will be exactly the value of the given `UInt64` if such a `Float32` exists. If no such `Float32` exists, the returned value will either be the smallest `Float32` that is larger than the given value, or the largest `Float32` that is smaller than the given value.

This function is opaque in the kernel, but is overridden at runtime with an efficient implementation.

19.4.4.3.5. To and From Bitvectors

def

```
ISize.toBitVec (x : ISize) : BitVec System.Platform.numBits
```

Obtain the `BitVec` that contains the 2's complement representation of the `ISize`.

def

```
ISize.ofBitVec (b : BitVec System.Platform.numBits) : ISize
```

Obtains the `ISize` whose 2's complement representation is the given `BitVec`.

def

```
Int8.toBitVec (x : Int8) : BitVec 8
```

Obtain the `BitVec` that contains the 2's complement representation of the `Int8`.

def

```
Int8.ofBitVec (b : BitVec 8) : Int8
```

Obtains the `Int8` whose 2's complement representation is the given `BitVec 8`.

def

```
Int16.toBitVec (x : Int16) : BitVec 16
```

Obtain the BitVec that contains the 2's complement representation of the Int16.

def

```
Int16.ofBitVec (b : BitVec 16) : Int16
```

Obtains the Int16 whose 2's complement representation is the given BitVec 16.

def

```
Int32.toBitVec (x : Int32) : BitVec 32
```

Obtain the BitVec that contains the 2's complement representation of the Int32.

def

```
Int32.ofBitVec (b : BitVec 32) : Int32
```

Obtains the Int32 whose 2's complement representation is the given BitVec 32.

def

```
Int64.toBitVec (x : Int64) : BitVec 64
```

Obtain the BitVec that contains the 2's complement representation of the Int64.

def

```
Int64.ofBitVec (b : BitVec 64) : Int64
```

Obtains the Int64 whose 2's complement representation is the given BitVec 64.

19.4.4.3.6. To and From Finite Numbers

def

```
USize.toFin (x : USize) : Fin USize.size
```

Converts a USize into the corresponding Fin USize.size.

def

```
UInt8.toFin (x : UInt8) : Fin UInt8.size
```

Converts a `UInt8` into the corresponding `Fin UInt8.size`.

def

```
UInt16.toFin (x : UInt16) : Fin UInt16.size
```

Converts a `UInt16` into the corresponding `Fin UInt16.size`.

def

```
UInt32.toFin (x : UInt32) : Fin UInt32.size
```

Converts a `UInt32` into the corresponding `Fin UInt32.size`.

def

```
UInt64.toFin (x : UInt64) : Fin UInt64.size
```

Converts a `UInt64` into the corresponding `Fin UInt64.size`.

def

```
USize.ofFin (a : Fin USize.size) : USize
```

Converts a `Fin USize.size` into the corresponding `USize`.

def

```
UInt8.ofFin (a : Fin UInt8.size) : UInt8
```

Converts a `Fin UInt8.size` into the corresponding `UInt8`.

def

```
UInt16.ofFin (a : Fin UInt16.size) : UInt16
```

Converts a `Fin UInt16.size` into the corresponding `UInt16`.

def

```
UInt32.ofFin (a : Fin UInt32.size) : UInt32
```

Converts a `Fin UInt32.size` into the corresponding `UInt32`.

def

```
UInt64.ofFin (a : Fin UInt64.size) : UInt64
```

Converts a `Fin UInt64.size` into the corresponding `UInt64`.

def

```
USize.repr (n : USize) : String
```

Converts a word-sized unsigned integer into a decimal string.

This function is overridden at runtime with an efficient implementation.

Examples:

- `USize.repr 0 = "0"`
- `USize.repr 28 = "28"`
- `USize.repr 307 = "307"`

19.4.4.3.7. To Characters

The `Char` type is a wrapper around `UInt32` that requires a proof that the wrapped integer represents a Unicode code point. This predicate is part of the `UInt32` API.

def

```
UInt32.isValidChar (n : UInt32) : Prop
```

A `UInt32` denotes a valid Unicode code point if it is less than `0x110000` and it is also not a surrogate code point (the range `0xd800` to `0xdfff` inclusive).

19.4.4.4. Comparisons

The operators in this section are rarely invoked by name. Typically, comparisons operations on fixed-width integers should use the decidability of the corresponding relations, which consist of the equality type `Eq` and those implemented in instances of `LE` and `LT`.

def

```
USize.le (a b : USize) : Prop
```

Non-strict inequality of word-sized unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the ≤ operator.

def

```
ISize.le (a b : ISize) : Prop
```

Non-strict inequality of word-sized signed integers, defined as inequality of the corresponding integers. Usually accessed via the ≤ operator.

def

```
UInt8.le (a b : UInt8) : Prop
```

Non-strict inequality of 8-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the ≤ operator.

def

```
Int8.le (a b : Int8) : Prop
```

Non-strict inequality of 8-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the ≤ operator.

def

```
UInt16.le (a b : UInt16) : Prop
```

Non-strict inequality of 16-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the ≤ operator.

def

```
Int16.le (a b : Int16) : Prop
```

Non-strict inequality of 16-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the ≤ operator.

def

```
UInt32.le (a b : UInt32) : Prop
```

Non-strict inequality of 32-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the ≤ operator.

def

```
Int32.le (a b : Int32) : Prop
```

Non-strict inequality of 32-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the ≤ operator.

def

```
UInt64.le (a b : UInt64) : Prop
```

Non-strict inequality of 64-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the ≤ operator.

def

```
Int64.le (a b : Int64) : Prop
```

Non-strict inequality of 64-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the ≤ operator.

def

```
USize.lt (a b : USize) : Prop
```

Strict inequality of word-sized unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the < operator.

def

```
ISize.lt (a b : ISize) : Prop
```

Strict inequality of word-sized signed integers, defined as inequality of the corresponding integers. Usually accessed via the < operator.

def

```
UInt8.lt (a b : UInt8) : Prop
```

Strict inequality of 8-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the < operator.

def

```
Int8.lt (a b : Int8) : Prop
```

Strict inequality of 8-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the < operator.

def

```
UInt16.lt (a b : UInt16) : Prop
```

Strict inequality of 16-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the < operator.

def

```
Int16.lt (a b : Int16) : Prop
```

Strict inequality of 16-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the < operator.

def

```
UInt32.lt (a b : UInt32) : Prop
```

Strict inequality of 32-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the < operator.

def

```
Int32.lt (a b : Int32) : Prop
```

Strict inequality of 32-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the < operator.

def

```
UInt64.lt (a b : UInt64) : Prop
```

Strict inequality of 64-bit unsigned integers, defined as inequality of the corresponding natural numbers. Usually accessed via the < operator.

def

```
Int64.lt (a b : Int64) : Prop
```

Strict inequality of 64-bit signed integers, defined as inequality of the corresponding integers. Usually accessed via the < operator.

def

```
USize.decEq (a b : USize) : Decidable (a = b)
```

Decides whether two word-sized unsigned integers are equal. Usually accessed via the `DecidableEq USize` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `USize.decEq 123 123 = .isTrue rfl`
- `(if (6 : USize) = 7 then "yes" else "no") = "no"`
- `show (7 : USize) = 7 by decide`

def

`ISize.decEq (a b : ISize) : Decidable (a = b)`

Decides whether two word-sized signed integers are equal. Usually accessed via the `DecidableEq ISize` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `ISize.decEq 123 123 = .isTrue rfl`
- `(if ((-7) : ISize) = 7 then "yes" else "no") = "no"`
- `show (7 : ISize) = 7 by decide`

def

`UInt8.decEq (a b : UInt8) : Decidable (a = b)`

Decides whether two 8-bit unsigned integers are equal. Usually accessed via the `DecidableEq UInt8` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt8.decEq 123 123 = .isTrue rfl`
- `(if (6 : UInt8) = 7 then "yes" else "no") = "no"`
- `show (7 : UInt8) = 7 by decide`

def

`Int8.decEq (a b : Int8) : Decidable (a = b)`

Decides whether two 8-bit signed integers are equal. Usually accessed via the `DecidableEq Int8` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int8.decEq 123 123 = .isTrue rfl`
- `(if ((-7) : Int8) = 7 then "yes" else "no") = "no"`
- `show (7 : Int8) = 7 by decide`

def

`UInt16.decEq (a b : UInt16) : Decidable (a = b)`

Decides whether two 16-bit unsigned integers are equal. Usually accessed via the `DecidableEq UInt16` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt16.decEq 123 123 = .isTrue rfl`
- `(if (6 : UInt16) = 7 then "yes" else "no") = "no"`
- `show (7 : UInt16) = 7 by decide`

def

`Int16.decEq (a b : Int16) : Decidable (a = b)`

Decides whether two 16-bit signed integers are equal. Usually accessed via the `DecidableEq Int16` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int16.decEq 123 123 = .isTrue rfl`
- `(if ((-7) : Int16) = 7 then "yes" else "no") = "no"`
- `show (7 : Int16) = 7 by decide`

def

`UInt32.decEq (a b : UInt32) : Decidable (a = b)`

Decides whether two 32-bit unsigned integers are equal. Usually accessed via the `DecidableEq UInt32` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt32.decEq 123 123 = .isTrue rfl`

- `(if (6 : UInt32) = 7 then "yes" else "no") = "no"`
- `show (7 : UInt32) = 7 by decide`

def

`Int32.decEq (a b : Int32) : Decidable (a = b)`

Decides whether two 32-bit signed integers are equal. Usually accessed via the `DecidableEq Int32` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int32.decEq 123 123 = .isTrue rfl`
- `(if ((-7) : Int32) = 7 then "yes" else "no") = "no"`
- `show (7 : Int32) = 7 by decide`

def

`UInt64.decEq (a b : UInt64) : Decidable (a = b)`

Decides whether two 64-bit unsigned integers are equal. Usually accessed via the `DecidableEq UInt64` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt64.decEq 123 123 = .isTrue rfl`
- `(if (6 : UInt64) = 7 then "yes" else "no") = "no"`
- `show (7 : UInt64) = 7 by decide`

def

`Int64.decEq (a b : Int64) : Decidable (a = b)`

Decides whether two 64-bit signed integers are equal. Usually accessed via the `DecidableEq Int64` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int64.decEq 123 123 = .isTrue rfl`
- `(if ((-7) : Int64) = 7 then "yes" else "no") = "no"`

- `show (7 : Int64) = 7 by decide`

def

`USize.decLe (a b : USize) : Decidable (a ≤ b)`

Decides whether one word-sized unsigned integer is less than or equal to another. Usually accessed via the `DecidableLE USize` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (15 : USize) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : USize) ≤ 5 then "yes" else "no") = "no"`
- `(if (5 : USize) ≤ 15 then "yes" else "no") = "yes"`
- `show (7 : USize) ≤ 7 by decide`

def

`ISize.decLe (a b : ISize) : Decidable (a ≤ b)`

Decides whether one word-sized signed integer is less than or equal to another. Usually accessed via the `DecidableLE ISize` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : ISize) ≤ 7 then "yes" else "no") = "yes"`
- `(if (15 : ISize) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : ISize) ≤ 5 then "yes" else "no") = "no"`
- `show (7 : ISize) ≤ 7 by decide`

def

`UInt8.decLe (a b : UInt8) : Decidable (a ≤ b)`

Decides whether one 8-bit unsigned integer is less than or equal to another. Usually accessed via the `DecidableLE UInt8` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (15 : UInt8) ≤ 15 then "yes" else "no") = "yes"`

- `(if (15 : UInt8) ≤ 5 then "yes" else "no") = "no"`
- `(if (5 : UInt8) ≤ 15 then "yes" else "no") = "yes"`
- `show (7 : UInt8) ≤ 7 by decide`

def

`Int8.decLe (a b : Int8) : Decidable (a ≤ b)`

Decides whether one 8-bit signed integer is less than or equal to another. Usually accessed via the `DecidableLE Int8` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int8) ≤ 7 then "yes" else "no") = "yes"`
- `(if (15 : Int8) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : Int8) ≤ 5 then "yes" else "no") = "no"`
- `show (7 : Int8) ≤ 7 by decide`

def

`UInt16.decLe (a b : UInt16) : Decidable (a ≤ b)`

Decides whether one 16-bit unsigned integer is less than or equal to another. Usually accessed via the `DecidableLE UInt16` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (15 : UInt16) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : UInt16) ≤ 5 then "yes" else "no") = "no"`
- `(if (5 : UInt16) ≤ 15 then "yes" else "no") = "yes"`
- `show (7 : UInt16) ≤ 7 by decide`

def

`Int16.decLe (a b : Int16) : Decidable (a ≤ b)`

Decides whether one 16-bit signed integer is less than or equal to another. Usually accessed via the `DecidableLE Int16` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int16) ≤ 7 then "yes" else "no") = "yes"`
- `(if (15 : Int16) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : Int16) ≤ 5 then "yes" else "no") = "no"`
- `show (7 : Int16) ≤ 7 by decide`

def

`UInt32.decLe (a b : UInt32) : Decidable (a ≤ b)`

Decides whether one 32-bit signed integer is less than or equal to another. Usually accessed via the `DecidableLE UInt32` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (15 : UInt32) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : UInt32) ≤ 5 then "yes" else "no") = "no"`
- `(if (5 : UInt32) ≤ 15 then "yes" else "no") = "yes"`
- `show (7 : UInt32) ≤ 7 by decide`

def

`Int32.decLe (a b : Int32) : Decidable (a ≤ b)`

Decides whether one 32-bit signed integer is less than or equal to another. Usually accessed via the `DecidableLE Int32` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int32) ≤ 7 then "yes" else "no") = "yes"`
- `(if (15 : Int32) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : Int32) ≤ 5 then "yes" else "no") = "no"`
- `show (7 : Int32) ≤ 7 by decide`

def

`UInt64.decLe (a b : UInt64) : Decidable (a ≤ b)`

Decides whether one 64-bit unsigned integer is less than or equal to another. Usually accessed via the `DecidableLE UInt64` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (15 : UInt64) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : UInt64) ≤ 5 then "yes" else "no") = "no"`
- `(if (5 : UInt64) ≤ 15 then "yes" else "no") = "yes"`
- `show (7 : UInt64) ≤ 7 by decide`

def

`Int64.decLe (a b : Int64) : Decidable (a ≤ b)`

Decides whether one 8-bit signed integer is less than or equal to another. Usually accessed via the `DecidableLE Int64` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int64) ≤ 7 then "yes" else "no") = "yes"`
- `(if (15 : Int64) ≤ 15 then "yes" else "no") = "yes"`
- `(if (15 : Int64) ≤ 5 then "yes" else "no") = "no"`
- `show (7 : Int64) ≤ 7 by decide`

def

`USize.decLt (a b : USize) : Decidable (a < b)`

Decides whether one word-sized unsigned integer is strictly less than another. Usually accessed via the `DecidableLT USize` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (6 : USize) < 7 then "yes" else "no") = "yes"`
- `(if (5 : USize) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : USize) < 7) by decide`

def

`ISize.decLt (a b : ISize) : Decidable (a < b)`

Decides whether one word-sized signed integer is strictly less than another. Usually accessed via the `DecidableLT ISize` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : ISize) < 7 then "yes" else "no") = "yes"`
- `(if (5 : ISize) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : ISize) < 7) by decide`

def

`UInt8.decLt (a b : UInt8) : Decidable (a < b)`

Decides whether one 8-bit unsigned integer is strictly less than another. Usually accessed via the `DecidableLT UInt8` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (6 : UInt8) < 7 then "yes" else "no") = "yes"`
- `(if (5 : UInt8) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : UInt8) < 7) by decide`

def

`Int8.decLt (a b : Int8) : Decidable (a < b)`

Decides whether one 8-bit signed integer is strictly less than another. Usually accessed via the `DecidableLT Int8` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int8) < 7 then "yes" else "no") = "yes"`
- `(if (5 : Int8) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : Int8) < 7) by decide`

def

`UInt16.decLt (a b : UInt16) : Decidable (a < b)`

Decides whether one 16-bit unsigned integer is strictly less than another. Usually accessed via the `DecidableLT UInt16` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (6 : UInt16) < 7 then "yes" else "no") = "yes"`
- `(if (5 : UInt16) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : UInt16) < 7) by decide`

def

`Int16.decLt (a b : Int16) : Decidable (a ≤ b)`

Decides whether one 16-bit signed integer is strictly less than another. Usually accessed via the `DecidableLT Int16` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int16) < 7 then "yes" else "no") = "yes"`
- `(if (5 : Int16) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : Int16) < 7) by decide`

def

`UInt32.decLt (a b : UInt32) : Decidable (a ≤ b)`

Decides whether one 8-bit unsigned integer is strictly less than another. Usually accessed via the `DecidableLT UInt32` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (6 : UInt32) < 7 then "yes" else "no") = "yes"`
- `(if (5 : UInt32) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : UInt32) < 7) by decide`

def

`Int32.decLt (a b : Int32) : Decidable (a ≤ b)`

Decides whether one 32-bit signed integer is strictly less than another. Usually accessed via the `DecidableLT Int32` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int32) < 7 then "yes" else "no") = "yes"`

- `(if (5 : Int32) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : Int32) < 7) by decide`

def

`UInt64.decLt (a b : UInt64) : Decidable (a < b)`

Decides whether one 64-bit unsigned integer is strictly less than another. Usually accessed via the `DecidableLT UInt64` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if (6 : UInt64) < 7 then "yes" else "no") = "yes"`
- `(if (5 : UInt64) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : UInt64) < 7) by decide`

def

`Int64.decLt (a b : Int64) : Decidable (a < b)`

Decides whether one 8-bit signed integer is strictly less than another. Usually accessed via the `DecidableLT Int64` instance.

This function is overridden at runtime with an efficient implementation.

Examples:

- `(if ((-7) : Int64) < 7 then "yes" else "no") = "yes"`
- `(if (5 : Int64) < 5 then "yes" else "no") = "no"`
- `show ¬((7 : Int64) < 7) by decide`

19.4.4.5. Arithmetic

Typically, arithmetic operations on fixed-width integers should be accessed using Lean's overloaded arithmetic notation, particularly their instances of `Add`, `Sub`, `Mul`, `Div`, and `Mod`, as well as `Neg` for signed types.

def

`ISize.neg (i : ISize) : ISize`

Negates word-sized signed integers. Usually accessed via the − prefix operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.neg (i : Int8) : Int8
```

Negates 8-bit signed integers. Usually accessed via the − prefix operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.neg (i : Int16) : Int16
```

Negates 16-bit signed integers. Usually accessed via the − prefix operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.neg (i : Int32) : Int32
```

Negates 32-bit signed integers. Usually accessed via the − prefix operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.neg (i : Int64) : Int64
```

Negates 64-bit signed integers. Usually accessed via the − prefix operator.

This function is overridden at runtime with an efficient implementation.

def

```
USize.neg (a : USize) : USize
```

Negation of word-sized unsigned integers, computed modulo `USize.size`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.neg (a : UInt8) : UInt8
```

Negation of 8-bit unsigned integers, computed modulo `UInt8.size`.

`UInt8.neg a` is equivalent to `255 - a + 1`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.neg (a : UInt16) : UInt16
```

Negation of 16-bit unsigned integers, computed modulo `UInt16.size`.

`UInt16.neg` *a* is equivalent to `65_535 - a + 1`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.neg (a : UInt32) : UInt32
```

Negation of 32-bit unsigned integers, computed modulo `UInt32.size`.

`UInt32.neg` *a* is equivalent to `429_4967_295 - a + 1`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.neg (a : UInt64) : UInt64
```

Negation of 64-bit unsigned integers, computed modulo `UInt64.size`.

`UInt64.neg` *a* is equivalent to `18_446_744_073_709_551_615 - a + 1`.

This function is overridden at runtime with an efficient implementation.

def

```
USize.add (a b : USize) : USize
```

Adds two word-sized unsigned integers, wrapping around on overflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.add (a b : ISize) : ISize
```

Adds two word-sized signed integers, wrapping around on over- or underflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.add (a b : UInt8) : UInt8
```

Adds two 8-bit unsigned integers, wrapping around on overflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.add (a b : Int8) : Int8
```

Adds two 8-bit signed integers, wrapping around on over- or underflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.add (a b : UInt16) : UInt16
```

Adds two 16-bit unsigned integers, wrapping around on overflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.add (a b : Int16) : Int16
```

Adds two 16-bit signed integers, wrapping around on over- or underflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.add (a b : UInt32) : UInt32
```

Adds two 32-bit unsigned integers, wrapping around on overflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.add (a b : Int32) : Int32
```

Adds two 32-bit signed integers, wrapping around on over- or underflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.add (a b : UInt64) : UInt64
```

Adds two 64-bit unsigned integers, wrapping around on overflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.add (a b : Int64) : Int64
```

Adds two 64-bit signed integers, wrapping around on over- or underflow. Usually accessed via the + operator.

This function is overridden at runtime with an efficient implementation.

def

```
USize.sub (a b : USize) : USize
```

Subtracts one word-sized-bit unsigned integer from another, wrapping around on underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.sub (a b : ISize) : ISize
```

Subtracts one word-sized signed integer from another, wrapping around on over- or underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.sub (a b : UInt8) : UInt8
```

Subtracts one 8-bit unsigned integer from another, wrapping around on underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.sub (a b : Int8) : Int8
```

Subtracts one 8-bit signed integer from another, wrapping around on over- or underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.sub (a b : UInt16) : UInt16
```

Subtracts one 16-bit unsigned integer from another, wrapping around on underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.sub (a b : Int16) : Int16
```

Subtracts one 16-bit signed integer from another, wrapping around on over- or underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.sub (a b : UInt32) : UInt32
```

Subtracts one 32-bit unsigned integer from another, wrapping around on underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.sub (a b : Int32) : Int32
```

Subtracts one 32-bit signed integer from another, wrapping around on over- or underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.sub (a b : UInt64) : UInt64
```

Subtracts one 64-bit unsigned integer from another, wrapping around on underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.sub (a b : Int64) : Int64
```

Subtracts one 64-bit signed integer from another, wrapping around on over- or underflow. Usually accessed via the − operator.

This function is overridden at runtime with an efficient implementation.

def

```
USize.mul (a b : USize) : USize
```

Multiplies two word-sized unsigned integers, wrapping around on overflow. Usually accessed via the * operator.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.mul (a b : ISize) : ISize
```

Multiplies two word-sized signed integers, wrapping around on over- or underflow. Usually accessed via the * operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.mul (a b : UInt8) : UInt8
```

Multiplies two 8-bit unsigned integers, wrapping around on overflow. Usually accessed via the * operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.mul (a b : Int8) : Int8
```

Multiplies two 8-bit signed integers, wrapping around on over- or underflow. Usually accessed via the * operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.mul (a b : UInt16) : UInt16
```

Multiplies two 16-bit unsigned integers, wrapping around on overflow. Usually accessed via the `*` operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.mul (a b : Int16) : Int16
```

Multiplies two 16-bit signed integers, wrapping around on over- or underflow. Usually accessed via the `*` operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.mul (a b : UInt32) : UInt32
```

Multiplies two 32-bit unsigned integers, wrapping around on overflow. Usually accessed via the `*` operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.mul (a b : Int32) : Int32
```

Multiplies two 32-bit signed integers, wrapping around on over- or underflow. Usually accessed via the `*` operator.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.mul (a b : UInt64) : UInt64
```

Multiplies two 64-bit unsigned integers, wrapping around on overflow. Usually accessed via the `*` operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.mul (a b : Int64) : Int64
```

Multiplies two 64-bit signed integers, wrapping around on over- or underflow. Usually accessed via the `*` operator.

This function is overridden at runtime with an efficient implementation.

def

```
USize.div (a b : USize) : USize
```

Unsigned division for word-sized unsigned integers, discarding the remainder. Usually accessed via the `/` operator.

This operation is sometimes called “floor division.” Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.div (a b : ISize) : ISize
```

Truncating division for word-sized signed integers, rounding towards zero. Usually accessed via the `/` operator.

Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

Examples:

- `ISize.div 10 3 = 3`
- `ISize.div 10 (-3) = (-3)`
- `ISize.div (-10) (-3) = 3`
- `ISize.div (-10) 3 = (-3)`
- `ISize.div 10 0 = 0`

def

```
UInt8.div (a b : UInt8) : UInt8
```

Unsigned division for 8-bit unsigned integers, discarding the remainder. Usually accessed via the `/` operator.

This operation is sometimes called “floor division.” Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.div (a b : Int8) : Int8
```

Truncating division for 8-bit signed integers, rounding towards zero. Usually accessed via the / operator.

Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int8.div 10 3 = 3`
- `Int8.div 10 (-3) = (-3)`
- `Int8.div (-10) (-3) = 3`
- `Int8.div (-10) 3 = (-3)`
- `Int8.div 10 0 = 0`

def

```
UInt16.div (a b : UInt16) : UInt16
```

Unsigned division for 16-bit unsigned integers, discarding the remainder. Usually accessed via the / operator.

This operation is sometimes called “floor division.” Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.div (a b : Int16) : Int16
```

Truncating division for 16-bit signed integers, rounding towards zero. Usually accessed via the / operator.

Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int16.div 10 3 = 3`
- `Int16.div 10 (-3) = (-3)`

- `Int16.div (-10) (-3) = 3`
- `Int16.div (-10) 3 = (-3)`
- `Int16.div 10 0 = 0`

def

`UInt32.div (a b : UInt32) : UInt32`

Unsigned division for 32-bit unsigned integers, discarding the remainder. Usually accessed via the / operator.

This operation is sometimes called “floor division.” Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

def

`Int32.div (a b : Int32) : Int32`

Truncating division for 32-bit signed integers, rounding towards zero. Usually accessed via the / operator.

Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int32.div 10 3 = 3`
- `Int32.div 10 (-3) = (-3)`
- `Int32.div (-10) (-3) = 3`
- `Int32.div (-10) 3 = (-3)`
- `Int32.div 10 0 = 0`

def

`UInt64.div (a b : UInt64) : UInt64`

Unsigned division for 64-bit unsigned integers, discarding the remainder. Usually accessed via the / operator.

This operation is sometimes called “floor division.” Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.div (a b : Int64) : Int64
```

Truncating division for 64-bit signed integers, rounding towards zero. Usually accessed via the / operator.

Division by zero is defined to be zero.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int64.div 10 3 = 3`
- `Int64.div 10 (-3) = (-3)`
- `Int64.div (-10) (-3) = 3`
- `Int64.div (-10) 3 = (-3)`
- `Int64.div 10 0 = 0`

def

```
USize.mod (a b : USize) : USize
```

The modulo operator for word-sized unsigned integers, which computes the remainder when dividing one integer by another. Usually accessed via the % operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `USize.mod 5 2 = 1`
- `USize.mod 4 2 = 0`
- `USize.mod 4 0 = 4`

def

```
ISize.mod (a b : ISize) : ISize
```

The modulo operator for word-sized signed integers, which computes the remainder when dividing one integer by another with the T-rounding convention used by `ISize.div`. Usually accessed via the % operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `ISize.mod 5 2 = 1`
- `ISize.mod 5 (-2) = 1`
- `ISize.mod (-5) 2 = (-1)`
- `ISize.mod (-5) (-2) = (-1)`
- `ISize.mod 4 2 = 0`
- `ISize.mod 4 (-2) = 0`
- `ISize.mod 4 0 = 4`
- `ISize.mod (-4) 0 = (-4)`

def

`UInt8.mod (a b : UInt8) : UInt8`

The modulo operator for 8-bit unsigned integers, which computes the remainder when dividing one integer by another. Usually accessed via the % operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt8.mod 5 2 = 1`
- `UInt8.mod 4 2 = 0`
- `UInt8.mod 4 0 = 4`

def

`Int8.mod (a b : Int8) : Int8`

The modulo operator for 8-bit signed integers, which computes the remainder when dividing one integer by another with the T-rounding convention used by `Int8.div`. Usually accessed via the % operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int8.mod 5 2 = 1`
- `Int8.mod 5 (-2) = 1`

- `Int8.mod (-5) 2 = (-1)`
- `Int8.mod (-5) (-2) = (-1)`
- `Int8.mod 4 2 = 0`
- `Int8.mod 4 (-2) = 0`
- `Int8.mod 4 0 = 4`
- `Int8.mod (-4) 0 = (-4)`

def

`UInt16.mod (a b : UInt16) : UInt16`

The modulo operator for 16-bit unsigned integers, which computes the remainder when dividing one integer by another. Usually accessed via the % operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt16.mod 5 2 = 1`
- `UInt16.mod 4 2 = 0`
- `UInt16.mod 4 0 = 4`

def

`Int16.mod (a b : Int16) : Int16`

The modulo operator for 16-bit signed integers, which computes the remainder when dividing one integer by another with the T-rounding convention used by `Int16.div`. Usually accessed via the % operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int16.mod 5 2 = 1`
- `Int16.mod 5 (-2) = 1`
- `Int16.mod (-5) 2 = (-1)`
- `Int16.mod (-5) (-2) = (-1)`
- `Int16.mod 4 2 = 0`
- `Int16.mod 4 (-2) = 0`

- `Int16.mod 4 0 = 4`
- `Int16.mod (-4) 0 = (-4)`

def

`UInt32.mod (a b : UInt32) : UInt32`

The modulo operator for 32-bit unsigned integers, which computes the remainder when dividing one integer by another. Usually accessed via the `%` operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt32.mod 5 2 = 1`
- `UInt32.mod 4 2 = 0`
- `UInt32.mod 4 0 = 4`

def

`Int32.mod (a b : Int32) : Int32`

The modulo operator for 32-bit signed integers, which computes the remainder when dividing one integer by another with the T-rounding convention used by `Int32.div`. Usually accessed via the `%` operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int32.mod 5 2 = 1`
- `Int32.mod 5 (-2) = 1`
- `Int32.mod (-5) 2 = (-1)`
- `Int32.mod (-5) (-2) = (-1)`
- `Int32.mod 4 2 = 0`
- `Int32.mod 4 (-2) = 0`
- `Int32.mod 4 0 = 4`
- `Int32.mod (-4) 0 = (-4)`

def

`UInt64.mod (a b : UInt64) : UInt64`

The modulo operator for 64-bit unsigned integers, which computes the remainder when dividing one integer by another. Usually accessed via the `%` operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `UInt64.mod 5 2 = 1`
- `UInt64.mod 4 2 = 0`
- `UInt64.mod 4 0 = 4`

def

`Int64.mod (a b : Int64) : Int64`

The modulo operator for 64-bit signed integers, which computes the remainder when dividing one integer by another with the T-rounding convention used by `Int64.div`. Usually accessed via the `%` operator.

When the divisor is `0`, the result is the dividend rather than an error.

This function is overridden at runtime with an efficient implementation.

Examples:

- `Int64.mod 5 2 = 1`
- `Int64.mod 5 (-2) = 1`
- `Int64.mod (-5) 2 = (-1)`
- `Int64.mod (-5) (-2) = (-1)`
- `Int64.mod 4 2 = 0`
- `Int64.mod 4 (-2) = 0`
- `Int64.mod 4 0 = 4`
- `Int64.mod (-4) 0 = (-4)`

def

`USize.log2 (a : USize) : USize`

Base-two logarithm of word-sized unsigned integers. Returns $\lfloor \texttt{max 0 (log}_2\texttt{ a)} \rfloor$.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `USize.log2 0 = 0`
- `USize.log2 1 = 0`
- `USize.log2 2 = 1`
- `USize.log2 4 = 2`
- `USize.log2 7 = 2`
- `USize.log2 8 = 3`

def

`UInt8.log2 (a : UInt8) : UInt8`

Base-two logarithm of 8-bit unsigned integers. Returns $\lfloor \texttt{max 0 (log}_2\texttt{ a)} \rfloor$.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `UInt8.log2 0 = 0`
- `UInt8.log2 1 = 0`
- `UInt8.log2 2 = 1`
- `UInt8.log2 4 = 2`
- `UInt8.log2 7 = 2`
- `UInt8.log2 8 = 3`

def

`UInt16.log2 (a : UInt16) : UInt16`

Base-two logarithm of 16-bit unsigned integers. Returns $\lfloor \texttt{max 0 (log}_2\texttt{ a)} \rfloor$.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `UInt16.log2 0 = 0`
- `UInt16.log2 1 = 0`
- `UInt16.log2 2 = 1`

- `UInt16.log2 4 = 2`
- `UInt16.log2 7 = 2`
- `UInt16.log2 8 = 3`

def

```
UInt32.log2 (a : UInt32) : UInt32
```

Base-two logarithm of 32-bit unsigned integers. Returns $\lfloor \texttt{max 0 (log}_2\ \texttt{a)} \rfloor$.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `UInt32.log2 0 = 0`
- `UInt32.log2 1 = 0`
- `UInt32.log2 2 = 1`
- `UInt32.log2 4 = 2`
- `UInt32.log2 7 = 2`
- `UInt32.log2 8 = 3`

def

```
UInt64.log2 (a : UInt64) : UInt64
```

Base-two logarithm of 64-bit unsigned integers. Returns $\lfloor \texttt{max 0 (log}_2\ \texttt{a)} \rfloor$.

This function is overridden at runtime with an efficient implementation. This definition is the logical model.

Examples:

- `UInt64.log2 0 = 0`
- `UInt64.log2 1 = 0`
- `UInt64.log2 2 = 1`
- `UInt64.log2 4 = 2`
- `UInt64.log2 7 = 2`
- `UInt64.log2 8 = 3`

def

```
ISize.abs (a : ISize) : ISize
```

Computes the absolute value of a word-sized signed integer.

This function is equivalent to `if a < 0 then -a else a`, so in particular `ISize.minValue` will be mapped to `ISize.minValue`.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.abs (a : Int8) : Int8
```

Computes the absolute value of an 8-bit signed integer.

This function is equivalent to `if a < 0 then -a else a`, so in particular `Int8.minValue` will be mapped to `Int8.minValue`.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.abs (a : Int16) : Int16
```

Computes the absolute value of a 16-bit signed integer.

This function is equivalent to `if a < 0 then -a else a`, so in particular `Int16.minValue` will be mapped to `Int16.minValue`.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.abs (a : Int32) : Int32
```

Computes the absolute value of a 32-bit signed integer.

This function is equivalent to `if a < 0 then -a else a`, so in particular `Int32.minValue` will be mapped to `Int32.minValue`.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.abs (a : Int64) : Int64
```

Computes the absolute value of a 64-bit signed integer.

This function is equivalent to `if a < 0 then -a else a`, so in particular `Int64.minValue` will be mapped to `Int64.minValue`.

This function is overridden at runtime with an efficient implementation.

19.4.4.6. Bitwise Operations

Typically, bitwise operations on fixed-width integers should be accessed using Lean's overloaded operators, particularly their instances of `ShiftLeft`, `ShiftRight`, `AndOp`, `OrOp`, and `Xor`.

def

`USize.land (a b : USize) : USize`

Bitwise and for word-sized unsigned integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

`ISize.land (a b : ISize) : ISize`

Bitwise and for word-sized signed integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

`UInt8.land (a b : UInt8) : UInt8`

Bitwise and for 8-bit unsigned integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

`Int8.land (a b : Int8) : Int8`

Bitwise and for 8-bit signed integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.land (a b : UInt16) : UInt16
```

Bitwise and for 16-bit unsigned integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.land (a b : Int16) : Int16
```

Bitwise and for 16-bit signed integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.land (a b : UInt32) : UInt32
```

Bitwise and for 32-bit unsigned integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.land (a b : Int32) : Int32
```

Bitwise and for 32-bit signed integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.land (a b : UInt64) : UInt64
```

Bitwise and for 64-bit unsigned integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.land (a b : Int64) : Int64
```

Bitwise and for 64-bit signed integers. Usually accessed via the &&& operator.

Each bit of the resulting integer is set if the corresponding bits of both input integers are set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
USize.lor (a b : USize) : USize
```

Bitwise or for word-sized unsigned integers. Usually accessed via the ||| operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.lor (a b : ISize) : ISize
```

Bitwise or for word-sized signed integers. Usually accessed via the ||| operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.lor (a b : UInt8) : UInt8
```

Bitwise or for 8-bit unsigned integers. Usually accessed via the ||| operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.lor (a b : Int8) : Int8
```

Bitwise or for 8-bit signed integers. Usually accessed via the `|||` operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.lor (a b : UInt16) : UInt16
```

Bitwise or for 16-bit unsigned integers. Usually accessed via the `|||` operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.lor (a b : Int16) : Int16
```

Bitwise or for 16-bit signed integers. Usually accessed via the `|||` operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.lor (a b : UInt32) : UInt32
```

Bitwise or for 32-bit unsigned integers. Usually accessed via the `|||` operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.lor (a b : Int32) : Int32
```

Bitwise or for 32-bit signed integers. Usually accessed via the `|||` operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.lor (a b : UInt64) : UInt64
```

Bitwise or for 64-bit unsigned integers. Usually accessed via the ||| operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.lor (a b : Int64) : Int64
```

Bitwise or for 64-bit signed integers. Usually accessed via the ||| operator.

Each bit of the resulting integer is set if at least one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
USize.xor (a b : USize) : USize
```

Bitwise exclusive or for word-sized unsigned integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.xor (a b : ISize) : ISize
```

Bitwise exclusive or for word-sized signed integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.xor (a b : UInt8) : UInt8
```

Bitwise exclusive or for 8-bit unsigned integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.xor (a b : Int8) : Int8
```

Bitwise exclusive or for 8-bit signed integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.xor (a b : UInt16) : UInt16
```

Bitwise exclusive or for 16-bit unsigned integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.xor (a b : Int16) : Int16
```

Bitwise exclusive or for 16-bit signed integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.xor (a b : UInt32) : UInt32
```

Bitwise exclusive or for 32-bit unsigned integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.xor (a b : Int32) : Int32
```

Bitwise exclusive or for 32-bit signed integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.xor (a b : UInt64) : UInt64
```

Bitwise exclusive or for 64-bit unsigned integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of both input integers are set.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.xor (a b : Int64) : Int64
```

Bitwise exclusive or for 64-bit signed integers. Usually accessed via the ^^^ operator.

Each bit of the resulting integer is set if exactly one of the corresponding bits of the input integers is set, according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
USize.complement (a : USize) : USize
```

Bitwise complement, also known as bitwise negation, for word-sized unsigned integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.complement (a : ISize) : ISize
```

Bitwise complement, also known as bitwise negation, for word-sized signed integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer. Integers use the two's complement representation, so `ISize.complement a = -(a + 1)`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.complement (a : UInt8) : UInt8
```

Bitwise complement, also known as bitwise negation, for 8-bit unsigned integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.complement (a : Int8) : Int8
```

Bitwise complement, also known as bitwise negation, for 8-bit signed integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer. Integers use the two's complement representation, so `Int8.complement a = -(a + 1)`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.complement (a : UInt16) : UInt16
```

Bitwise complement, also known as bitwise negation, for 16-bit unsigned integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.complement (a : Int16) : Int16
```

Bitwise complement, also known as bitwise negation, for 16-bit signed integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer. Integers use the two's complement representation, so `Int16.complement a = -(a + 1)`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.complement (a : UInt32) : UInt32
```

Bitwise complement, also known as bitwise negation, for 32-bit unsigned integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.complement (a : Int32) : Int32
```

Bitwise complement, also known as bitwise negation, for 32-bit signed integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer. Integers use the two's complement representation, so `Int32.complement a = -(a + 1)`.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.complement (a : UInt64) : UInt64
```

Bitwise complement, also known as bitwise negation, for 64-bit unsigned integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.complement (a : Int64) : Int64
```

Bitwise complement, also known as bitwise negation, for 64-bit signed integers. Usually accessed via the ~~~ prefix operator.

Each bit of the resulting integer is the opposite of the corresponding bit of the input integer. Integers use the two's complement representation, so `Int64.complement a = -(a + 1)`.

This function is overridden at runtime with an efficient implementation.

def

```
USize.shiftLeft (a b : USize) : USize
```

Bitwise left shift for word-sized unsigned integers. Usually accessed via the <<< operator.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.shiftLeft (a b : ISize) : ISize
```

Bitwise left shift for word-sized signed integers. Usually accessed via the <<< operator.

Signed integers are interpreted as bitvectors according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.shiftLeft (a b : UInt8) : UInt8
```

Bitwise left shift for 8-bit unsigned integers. Usually accessed via the <<< operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.shiftLeft (a b : Int8) : Int8
```

Bitwise left shift for 8-bit signed integers. Usually accessed via the <<< operator.

Signed integers are interpreted as bitvectors according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.shiftLeft (a b : UInt16) : UInt16
```

Bitwise left shift for 16-bit unsigned integers. Usually accessed via the <<< operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.shiftLeft (a b : Int16) : Int16
```

Bitwise left shift for 16-bit signed integers. Usually accessed via the <<< operator.

Signed integers are interpreted as bitvectors according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.shiftLeft (a b : UInt32) : UInt32
```

Bitwise left shift for 32-bit unsigned integers. Usually accessed via the <<< operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.shiftLeft (a b : Int32) : Int32
```

Bitwise left shift for 32-bit signed integers. Usually accessed via the <<< operator.

Signed integers are interpreted as bitvectors according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.shiftLeft (a b : UInt64) : UInt64
```

Bitwise left shift for 64-bit unsigned integers. Usually accessed via the <<< operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.shiftLeft (a b : Int64) : Int64
```

Bitwise left shift for 64-bit signed integers. Usually accessed via the <<< operator.

Signed integers are interpreted as bitvectors according to the two's complement representation.

This function is overridden at runtime with an efficient implementation.

def

```
USize.shiftRight (a b : USize) : USize
```

Bitwise right shift for word-sized unsigned integers. Usually accessed via the >>> operator.

This function is overridden at runtime with an efficient implementation.

def

```
ISize.shiftRight (a b : ISize) : ISize
```

Arithmetic right shift for word-sized signed integers. Usually accessed via the <<< operator.

The high bits are filled with the value of the most significant bit.

This function is overridden at runtime with an efficient implementation.

def

```
UInt8.shiftRight (a b : UInt8) : UInt8
```

Bitwise right shift for 8-bit unsigned integers. Usually accessed via the >>> operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int8.shiftRight (a b : Int8) : Int8
```

Arithmetic right shift for 8-bit signed integers. Usually accessed via the <<< operator.

The high bits are filled with the value of the most significant bit.

This function is overridden at runtime with an efficient implementation.

def

```
UInt16.shiftRight (a b : UInt16) : UInt16
```

Bitwise right shift for 16-bit unsigned integers. Usually accessed via the >>> operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int16.shiftRight (a b : Int16) : Int16
```

Arithmetic right shift for 16-bit signed integers. Usually accessed via the <<< operator.

The high bits are filled with the value of the most significant bit.

This function is overridden at runtime with an efficient implementation.

def

```
UInt32.shiftRight (a b : UInt32) : UInt32
```

Bitwise right shift for 32-bit unsigned integers. Usually accessed via the >>> operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int32.shiftRight (a b : Int32) : Int32
```

Arithmetic right shift for 32-bit signed integers. Usually accessed via the <<< operator.

The high bits are filled with the value of the most significant bit.

This function is overridden at runtime with an efficient implementation.

def

```
UInt64.shiftRight (a b : UInt64) : UInt64
```

Bitwise right shift for 64-bit unsigned integers. Usually accessed via the >>> operator.

This function is overridden at runtime with an efficient implementation.

def

```
Int64.shiftRight (a b : Int64) : Int64
```

Arithmetic right shift for 64-bit signed integers. Usually accessed via the <<< operator.

The high bits are filled with the value of the most significant bit.

This function is overridden at runtime with an efficient implementation.

19.5. Bitvectors

Bitvectors are fixed-width sequences of binary digits. They are frequently used in software verification, because they closely model efficient data structures and operations that are similar to hardware. A bitvector can be understood from two perspectives: as a sequence of bits, or as a number encoded by a sequence of bits. When a bitvector represents a number, it can do so as either a signed or an unsigned number. Signed numbers are represented in two's complement form.

19.5.1. Logical Model

Bitvectors are represented as a wrapper around a `Fin` with a suitable bound. Because `Fin` itself is a wrapper around a `Nat`, bitvectors are able to use the kernel's special support for efficient computation with natural numbers.

structure

```
BitVec (w : Nat) : Type
```

A bitvector of the specified width.

This is represented as the underlying `Nat` number in both the runtime and the kernel, inheriting all the special support for `Nat`.

Constructor

`BitVec.ofFin`

Construct a `BitVec` *w* from a number less than 2^*w*. O(1), because we use `Fin` as the internal representation of a bitvector.

Fields

`toFin : Fin (2 ^ w)`

Interpret a bitvector as a number less than 2^*w*. O(1), because we use `Fin` as the internal representation of a bitvector.

19.5.2. Runtime Representation

Bitvectors are represented as a `Fin` with the corresponding range. Because `BitVec` is a trivial wrapper around `Fin` and `Fin` is a trivial wrapper around `Nat`, bitvectors use the same runtime representation as `Nat` in compiled code.

19.5.3. Syntax

There is an `OfNat (BitVec w) n` instance for all widths *w* and natural numbers *n*. Natural number literals, including those that use hexadecimal or binary notation, may be used to represent bitvectors in contexts where the expected type is known. When the expected type is not known, a dedicated syntax allows the width of the bitvector to be specified along with its value.

▼ *Example: Numeric Literals for Bitvectors*

The following literals are all equivalent:

```
example : BitVec 8 := 0xff
example : BitVec 8 := 255
example : BitVec 8 := 0b1111_1111
```

syntax

Fixed-Width Bitvector Literals

```
term ::= ...
    | num#term
```

This notation pairs a numeric literal with a term that denotes its width. Spaces are forbidden around the `#`. Literals that overflow the width of the bitvector are truncated.

▼ *Example: Fixed-Width Bitvector Literals*

Bitvectors may be represented by natural number literals, so `(5 : BitVec 8)` is a valid bitvector. Additionally, a width may be specified directly in the literal:

```
5#8
```

Spaces are not allowed on either side of the `#`:

```
1 5 #8
<example>:1:2-1:3: expected end of input
1 5# 8
<example>:1:3-1:4: expected no space before
```

A numeric literal is required to the left of the `#`:

```
1 (3 + 2)#8
<example>:1:7-1:8: expected end of input
```

However, a term is allowed to the right of the #:

```
5#(4 + 4)
```

If the literal is too large to fit in the specified number of bits, then it is truncated:

```
#eval 7#2
```

```
3#2
```

syntax

Bounded Bitvector Literals

```
term ::= ...
    | num#'term
```

This notation is available only when the `BitVec` namespace has been opened. Rather than an explicit width, it expects a proof that the literal value is representable by a bitvector of the corresponding width.

▼ *Example: Bounded Bitvector Literals*

The bounded bitvector literal notation ensures that literals do not overflow the specified number of bits. The notation is only available when the `BitVec` namespace has been opened.

```
open BitVec
```

Literals that are in bounds require a proof to that effect:

```
example : BitVec 8 := 1#'(by decide)
```

Literals that are not in bounds are not allowed:

```
example : BitVec 8 := 256#'(by decide)
```

```
tactic 'decide' proved that the proposition
  256 < 2 ^ 8
is false
```

19.5.4. Automation

In addition to the full suite of automation and tools provided by Lean for every type, the `by decide` tactic can solve many bitvector-related problems. This tactic invokes an external automated theorem prover (`cadical`) and reconstructs the proof that it provides in Lean's own logic. The resulting proofs rely only on the axiom `Lean.ofReduceBool`; the external prover is not part of the trusted code base.

▼ *Example: Popcount*

The function `popcount` returns the number of set bits in a bitvector. It can be implemented as a 32-iteration loop that tests each bit, incrementing a counter if the bit is set:

```
def popcount_spec (x : BitVec 32) : BitVec 32 :=
  (32 : Nat).fold (init := 0) fun i _ pop =>
    pop + ((x >>> i) &&& 1)
```

An alternative implementation of `popcount` is described in *Hacker's Delight, Second Edition*, by Henry S. Warren, Jr. in Figure 5-2 on p. 82. It uses low-level bitwise operations to compute the same value with far fewer operations:

```
def popcount (x : BitVec 32) : BitVec 32 :=
  let x := x - ((x >>> 1) &&& 0x55555555)
  let x := (x &&& 0x33333333) + ((x >>> 2) &&& 0x33333333)
  let x := (x + (x >>> 4)) &&& 0x0F0F0F0F
  let x := x + (x >>> 8)
  let x := x + (x >>> 16)
  let x := x &&& 0x0000003F
  x
```

These two implementations can be proven equivalent using **`by_decide`**:

```
theorem popcount_correct : popcount = popcount_spec := by
  funext x
  simp [popcount, popcount_spec]
  by_decide
```

19.5.5. API Reference

19.5.5.1. Bounds

def

```
BitVec.intMax (w : Nat) : BitVec w
```

The bitvector of width *w* that has the largest value when interpreted as an integer.

def

```
BitVec.intMin (w : Nat) : BitVec w
```

The bitvector of width *w* that has the smallest value when interpreted as an integer.

19.5.5.2. Construction

def

```
BitVec.fill (w : Nat) (b : Bool) : BitVec w
```

Fills a bitvector with *w* copies of the bit *b*.

def

```
BitVec.zero (n : Nat) : BitVec n
```

Returns a bitvector of size *n* where all bits are `0`.

def

```
BitVec.allOnes (n : Nat) : BitVec n
```

Returns a bitvector of size *n* where all bits are `1`.

def

```
BitVec.twoPow (w i : Nat) : BitVec w
```

`twoPow w i` is the bitvector `2^i` if `i < w`, and `0` otherwise. In other words, it is 2 to the power `i`.

From the bitwise point of view, it has the `i`th bit as `1` and all other bits as `0`.

19.5.5.3. Conversion

def

```
BitVec.toHex {n : Nat} (x : BitVec n) : String
```

Converts a bitvector into a fixed-width hexadecimal number with enough digits to represent it.

If *n* is `0`, then one digit is returned. Otherwise, `⌊(n + 3) / 4⌋` digits are returned.

def

```
BitVec.toInt {n : Nat} (x : BitVec n) : Int
```

Interprets the bitvector as an integer stored in two's complement form.

def

```
BitVec.toNat {w : Nat} (x : BitVec w) : Nat
```

Return the underlying `Nat` that represents a bitvector.

This is O(1) because `BitVec` is a (zero-cost) wrapper around a `Nat`.

def

```
BitVec.ofBool (b : Bool) : BitVec 1
```

Turns a `Bool` into a bitvector of length 1.

def

```
BitVec.ofBoolListBE (bs : List Bool) : BitVec bs.length
```

Converts a list of `Bool`s into a big-endian `BitVec`.

def

```
BitVec.ofBoolListLE (bs : List Bool) : BitVec bs.length
```

Converts a list of `Bool`s into a little-endian `BitVec`.

def

```
BitVec.ofInt (n : Nat) (i : Int) : BitVec n
```

Converts an integer to its two's complement representation as a bitvector of the given width n, over- and underflowing as needed.

The underlying `Nat` is `(2^n + (i mod 2^n)) mod 2^n`. Converting the bitvector back to an `Int` with `BitVec.toInt` results in the value `i.bmod (2^n)`.

def

```
BitVec.ofNat (n i : Nat) : BitVec n
```

The bitvector with value `i mod 2^n`.

Conventions for notations in identifiers:

- The recommended spelling of `0#n` in identifiers is `zero` (not `ofNat_zero`).
- The recommended spelling of `1#n` in identifiers is `one` (not `ofNat_one`).

def

```
BitVec.ofNatLT {w : Nat} (i : Nat) (p : i < 2 ^ w) : BitVec w
```

The BitVec with value i, given a proof that $i < 2^w$.

def

```
BitVec.cast {n m : Nat} (eq : n = m) (x : BitVec n) : BitVec m
```

If two natural numbers n and m are equal, then a bitvector of width n is also a bitvector of width m.

Using x.cast eq should be preferred over eq ▸ x because there are special-purpose simp lemmas that can more consistently simplify BitVec.cast away.

19.5.5.4. Comparisons

def

```
BitVec.ule {n : Nat} (x y : BitVec n) : Bool
```

Unsigned less-than-or-equal-to for bitvectors.

SMT-LIB name: bvule.

def

```
BitVec.sle {n : Nat} (x y : BitVec n) : Bool
```

Signed less-than-or-equal-to for bitvectors.

SMT-LIB name: bvsle.

def

```
BitVec.ult {n : Nat} (x y : BitVec n) : Bool
```

Unsigned less-than for bitvectors.

SMT-LIB name: bvult.

def

```
BitVec.slt {n : Nat} (x y : BitVec n) : Bool
```

Signed less-than for bitvectors.

SMT-LIB name: bvslt.

Examples:

- `BitVec.slt 6#4 7 = true`
- `BitVec.slt 7#4 8 = false`

def

```
BitVec.decEq {w : Nat} (x y : BitVec w) : Decidable (x = y)
```

Bitvectors have decidable equality.

This should be used via the instance `DecidableEq (BitVec w)`.

19.5.5.5. Hashing

def

```
BitVec.hash {n : Nat} (bv : BitVec n) : UInt64
```

Computes a hash of a bitvector, combining 64-bit words using `mixHash`.

19.5.5.6. Sequence Operations

These operations treat bitvectors as sequences of bits, rather than as encodings of numbers.

def

```
BitVec.nil : BitVec 0
```

The empty bitvector.

def

```
BitVec.cons {n : Nat} (msb : Bool) (lsbs : BitVec n) : BitVec (n + 1)
```

Prepends a single bit to the front of a bitvector, using big-endian order (see append).

The new bit is the most significant bit.

def

```
BitVec.concat {n : Nat} (msbs : BitVec n) (lsb : Bool) : BitVec (n + 1)
```

Append a single bit to the end of a bitvector, using big endian order (see append). That is, the new bit is the least significant bit.

def

```
BitVec.shiftConcat {n : Nat} (x : BitVec n) (b : Bool) : BitVec n
```

Shifts all bits of *x* to the left by 1 and sets the least significant bit to *b*.

This is a non-dependent version of BitVec.concat that does not change the total bitwidth.

def

```
BitVec.truncate {w : Nat} (v : Nat) (x : BitVec w) : BitVec v
```

Transforms a bitvector of length *w* into a bitvector of length *v*, padding with 0 as needed.

The specific behavior depends on the relationship between the starting width *w* and the final width *v*:

- If $v > w$, it is zero-extended; the high bits are padded with zeroes until the bitvector has *v* bits.
- If $v = w$, the bitvector is returned unchanged.
- If $v < w$, the high bits are truncated.

BitVec.setWidth, BitVec.zeroExtend, and BitVec.truncate are aliases for this operation.

SMT-LIB name: zero_extend.

def

```
BitVec.setWidth {w : Nat} (v : Nat) (x : BitVec w) : BitVec v
```

Transforms a bitvector of length *w* into a bitvector of length *v*, padding with 0 as needed.

The specific behavior depends on the relationship between the starting width *w* and the final width *v*:

- If $v > w$, it is zero-extended; the high bits are padded with zeroes until the bitvector has *v* bits.
- If $v = w$, the bitvector is returned unchanged.
- If $v < w$, the high bits are truncated.

BitVec.setWidth, BitVec.zeroExtend, and BitVec.truncate are aliases for this operation.

SMT-LIB name: zero_extend.

def

```
BitVec.setWidth' {n w : Nat} (le : n ≤ w) (x : BitVec n) : BitVec w
```

Increases the width of a bitvector to one that is at least as large by zero-extending it.

This is a constant-time operation because the underlying `Nat` is unmodified; because the new width is at least as large as the old one, no overflow is possible.

def

```
BitVec.append {n m : Nat} (msbs : BitVec n) (lsbs : BitVec m) :
  BitVec (n + m)
```

Concatenates two bitvectors using the “big-endian” convention that the more significant input is on the left. Usually accessed via the ++ operator.

SMT-LIB name: `concat`.

Example:

- `0xAB#8 ++ 0xCD#8 = 0xABCD#16`.

def

```
BitVec.replicate {w : Nat} (i : Nat) : BitVec w → BitVec (w * i)
```

Concatenates i copies of x into a new vector of length $w * i$.

def

```
BitVec.reverse {w : Nat} : BitVec w → BitVec w
```

Reverses the bits in a bitvector.

def

```
BitVec.rotateLeft {w : Nat} (x : BitVec w) (n : Nat) : BitVec w
```

Rotates the bits in a bitvector to the left.

All the bits of x are shifted to higher positions, with the top n bits wrapping around to fill the vacated low bits.

SMT-LIB name: **`rotate_left`**, except this operator uses a `Nat` shift amount.

Example:

- `(0b0011#4).rotateLeft 3 = 0b1001`

def

```
BitVec.rotateRight {w : Nat} (x : BitVec w) (n : Nat) : BitVec w
```

Rotates the bits in a bitvector to the right.

All the bits of *x* are shifted to lower positions, with the bottom *n* bits wrapping around to fill the vacated high bits.

SMT-LIB name: **`rotate_right`**, except this operator uses a `Nat` shift amount.

Example:

- `rotateRight 0b01001#5 1 = 0b10100`

19.5.5.6.1. Bit Extraction

def

```
BitVec.msb {n : Nat} (x : BitVec n) : Bool
```

Returns the most significant bit in a bitvector.

def

```
BitVec.getMsbD {w : Nat} (x : BitVec w) (i : Nat) : Bool
```

Returns the ith most significant bit, or `false` if $i \geq w$.

def

```
BitVec.getMsb {w : Nat} (x : BitVec w) (i : Fin w) : Bool
```

Returns the ith most significant bit.

def

```
BitVec.getMsb? {w : Nat} (x : BitVec w) (i : Nat) : Option Bool
```

Returns the ith most significant bit or `none` if $i \geq w$.

def

```
BitVec.getLsbD {w : Nat} (x : BitVec w) (i : Nat) : Bool
```

Returns the ith least significant bit or `false` if $i \geq w$.

def

```
BitVec.getLsb {w : Nat} (x : BitVec w) (i : Fin w) : Bool
```

Returns the ith least significant bit.

def

```
BitVec.getLsb? {w : Nat} (x : BitVec w) (i : Nat) : Option Bool
```

Returns the ith least significant bit, or none if $i \geq w$.

def

```
BitVec.extractLsb {n : Nat} (hi lo : Nat) (x : BitVec n) :
  BitVec (hi - lo + 1)
```

Extracts the bits from hi down to lo (both inclusive) from a bitvector, which is implicitly zero-extended if necessary.

The resulting bitvector has size $hi - lo + 1$.

SMT-LIB name: `extract`.

def

```
BitVec.extractLsb' {n : Nat} (start len : Nat) (x : BitVec n) :
  BitVec len
```

Extracts the bits $start$ to $start + len - 1$ from a bitvector of size n to yield a new bitvector of size len. If $start + len > n$, then the bitvector is zero-extended.

19.5.5.7. Bitwise Operators

These operators modify the individual bits of one or more bitvectors.

def

```
BitVec.and {n : Nat} (x y : BitVec n) : BitVec n
```

Bitwise and for bitvectors. Usually accessed via the &&& operator.

SMT-LIB name: `bvand`.

Example:

- `0b1010#4 &&& 0b0110#4 = 0b0010#4`

def

```
BitVec.or {n : Nat} (x y : BitVec n) : BitVec n
```

Bitwise or for bitvectors. Usually accessed via the `|||` operator.

SMT-LIB name: `bvor`.

Example:

- `0b1010#4 ||| 0b0110#4 = 0b1110#4`

def

`BitVec.not {n : Nat} (x : BitVec n) : BitVec n`

Bitwise complement for bitvectors. Usually accessed via the `~~~` prefix operator.

SMT-LIB name: `bvnot`.

Example:

- `~~~(0b0101#4) == 0b1010`

def

`BitVec.xor {n : Nat} (x y : BitVec n) : BitVec n`

Bitwise xor for bitvectors. Usually accessed via the `^^^` operator.

SMT-LIB name: `bvxor`.

Example:

- `0b1010#4 ^^^ 0b0110#4 = 0b1100#4`

def

`BitVec.zeroExtend {w : Nat} (v : Nat) (x : BitVec w) : BitVec v`

Transforms a bitvector of length *w* into a bitvector of length *v*, padding with `0` as needed.

The specific behavior depends on the relationship between the starting width *w* and the final width *v*:

- If *v* `>` *w*, it is zero-extended; the high bits are padded with zeroes until the bitvector has *v* bits.
- If *v* `=` *w*, the bitvector is returned unchanged.
- If *v* `<` *w*, the high bits are truncated.

`BitVec.setWidth`, `BitVec.zeroExtend`, and `BitVec.truncate` are aliases for this operation.

SMT-LIB name: `zero_extend`.

def

```
BitVec.signExtend {w : Nat} (v : Nat) (x : BitVec w) : BitVec v
```

Transforms a bitvector of length *w* into a bitvector of length *v*, padding as needed with the most significant bit's value.

If *x* is an empty bitvector, then the sign is treated as zero.

SMT-LIB name: `sign_extend`.

def

```
BitVec.ushiftRight {n : Nat} (x : BitVec n) (s : Nat) : BitVec n
```

Shifts a bitvector to the right. This is a logical right shift - the high bits are filled with zeros.

As a numeric operation, this is equivalent to `x / 2^s`, rounding down.

SMT-LIB name: `bvlshr` except this operator uses a `Nat` shift value.

def

```
BitVec.sshiftRight {n : Nat} (x : BitVec n) (s : Nat) : BitVec n
```

Shifts a bitvector to the right. This is an arithmetic right shift - the high bits are filled with most significant bit's value.

As a numeric operation, this is equivalent to `x.toInt >>> s`.

SMT-LIB name: `bvashr` except this operator uses a `Nat` shift value.

def

```
BitVec.sshiftRight' {n m : Nat} (a : BitVec n) (s : BitVec m) :
BitVec n
```

Shifts a bitvector to the right. This is an arithmetic right shift - the high bits are filled with most significant bit's value.

As a numeric operation, this is equivalent to `a.toInt >>> s.toNat`.

SMT-LIB name: `bvashr`.

def

```
BitVec.shiftLeft {n : Nat} (x : BitVec n) (s : Nat) : BitVec n
```

Shifts a bitvector to the left. The low bits are filled with zeros. As a numeric operation, this is equivalent to `x * 2^s`, modulo `2^n`.

SMT-LIB name: `bvshl` except this operator uses a `Nat` shift value.

def

```
BitVec.shiftLeftZeroExtend {w : Nat} (msbs : BitVec w) (m : Nat) :
  BitVec (w + m)
```

Returns `zeroExtend (w+n) x <<< n` without needing to compute `x % 2^(2+n)`.

19.5.5.8. Arithmetic

These operators treat bitvectors as numbers. Some operations are signed, while others are unsigned. Because bitvectors are understood as two's complement numbers, addition, subtraction and multiplication coincide for the signed and unsigned interpretations.

def

```
BitVec.add {n : Nat} (x y : BitVec n) : BitVec n
```

Adds two bitvectors. This can be interpreted as either signed or unsigned addition modulo `2^n`. Usually accessed via the + operator.

SMT-LIB name: `bvadd`.

def

```
BitVec.sub {n : Nat} (x y : BitVec n) : BitVec n
```

Subtracts one bitvector from another. This can be interpreted as either signed or unsigned subtraction modulo `2^n`. Usually accessed via the − operator.

def

```
BitVec.mul {n : Nat} (x y : BitVec n) : BitVec n
```

Multiplies two bitvectors. This can be interpreted as either signed or unsigned multiplication modulo `2^n`. Usually accessed via the * operator.

SMT-LIB name: `bvmul`.

19.5.5.8.1. Unsigned Operations

def

```
BitVec.udiv {n : Nat} (x y : BitVec n) : BitVec n
```

Unsigned division of bitvectors using the Lean convention where division by zero returns zero. Usually accessed via the `/` operator.

def

```
BitVec.smtUDiv {n : Nat} (x y : BitVec n) : BitVec n
```

Unsigned division of bitvectors using the SMT-LIB convention, where division by zero returns `BitVector.allOnes n`.

SMT-LIB name: `bvudiv`.

def

```
BitVec.umod {n : Nat} (x y : BitVec n) : BitVec n
```

Unsigned modulo for bitvectors. Usually accessed via the `%` operator.

SMT-LIB name: `bvurem`.

def

```
BitVec.uaddOverflow {w : Nat} (x y : BitVec w) : Bool
```

Checks whether addition of *x* and *y* results in *unsigned* overflow.

SMT-LIB name: `bvuaddo`.

def

```
BitVec.usubOverflow {w : Nat} (x y : BitVec w) : Bool
```

Checks whether subtraction of *x* and *y* results in *unsigned* overflow.

SMT-Lib name: `bvusubo`.

19.5.5.8.2. Signed Operations

def

```
BitVec.abs {n : Nat} (x : BitVec n) : BitVec n
```

Returns the absolute value of a signed bitvector.

def

```
BitVec.neg {n : Nat} (x : BitVec n) : BitVec n
```

Negation of bitvectors. This can be interpreted as either signed or unsigned negation modulo `2^n`. Usually accessed via the − prefix operator.

SMT-LIB name: `bvneg`.

def

```
BitVec.sdiv {n : Nat} (x y : BitVec n) : BitVec n
```

Signed T-division (using the truncating rounding convention) for bitvectors. This function obeys the Lean convention that division by zero returns zero.

Examples:

- `(7#4).sdiv 2 = 3#4`
- `(-9#4).sdiv 2 = -4#4`
- `(5#4).sdiv -2 = -2#4`
- `(-7#4).sdiv (-2) = 3#4`

def

```
BitVec.smtSDiv {n : Nat} (x y : BitVec n) : BitVec n
```

Signed division for bitvectors using the SMT-LIB using the SMT-LIB convention, where division by zero returns `BitVector.allOnes n`.

Specifically, `x.smtSDiv 0 = if x >= 0 then -1 else 1`

SMT-LIB name: `bvsdiv`.

def

```
BitVec.smod {m : Nat} (x y : BitVec m) : BitVec m
```

Remainder for signed division rounded to negative infinity.

SMT-LIB name: `bvsmod`.

def

```
BitVec.srem {n : Nat} (x y : BitVec n) : BitVec n
```

Remainder for signed division rounding to zero.

SMT-LIB name: `bvsrem`.

def

```
BitVec.saddOverflow {w : Nat} (x y : BitVec w) : Bool
```

Checks whether addition of *x* and *y* results in *signed* overflow, treating *x* and *y* as 2's complement signed bitvectors.

SMT-LIB name: `bvsaddo`.

def

```
BitVec.ssubOverflow {w : Nat} (x y : BitVec w) : Bool
```

Checks whether the subtraction of *x* and *y* results in *signed* overflow, treating *x* and *y* as 2's complement signed bitvectors.

SMT-Lib name: `bvssubo`.

19.5.5.9. Iteration

def

```
BitVec.iunfoldr.{u_1} {w : Nat} {α : Type u_1}
  (f : Fin w → α → α × Bool) (s : α) : α × BitVec w
```

Constructs a bitvector by iteratively computing a state for each bit using the function *f*, starting with the initial state *s*. At each step, the prior state and the current bit index are passed to *f*, and it produces a bit along with the next state value. These bits are assembled into the final bitvector.

It produces a sequence of state values `[s_0, s_1 .. s_w]` and a bitvector v where `f i s_i = (s_{i+1}, b_i)` and *b_i* is bit `i`th least-significant bit in v (e.g., `getLsb v i = b_i`).

The theorem `iunfoldr_replace` allows uses of `BitVec.iunfoldr` to be replaced with declarative specifications that are easier to reason about.

theorem

```
BitVec.iunfoldr_replace.{u_1} {w : Nat} {α : Type u_1}
  {f : Fin w → α → α × Bool} (state : Nat → α) (value : BitVec w)
  (a : α) (init : state 0 = a)
  (step : ∀ (i : Fin w), f i (state ↑i) = (state (↑i + 1), value[↑i])) :
  BitVec.iunfoldr f a = (state w, value)
```

Given a function *state* that provides the correct state for every potential iteration count and a function that computes these states from the correct initial state, the result of applying `BitVec.iunfoldr` *f* to the initial state is the state corresponding to the bitvector's width

paired with the bitvector that consists of each computed bit.

This theorem can be used to prove properties of functions that are defined using `BitVec.iunfoldr`.

19.5.5.10. Proof Automation

19.5.5.10.1. Bit Blasting

The standard library contains a number of helper implementations that are useful to implement bit blasting, which is the technique used by **`by_decide`** to encode propositions as Boolean satisfiability problems for external solvers.

def

```
BitVec.adc {w : Nat} (x y : BitVec w) : Bool → Bool × BitVec w
```

Bitwise addition implemented via a ripple carry adder.

def

```
BitVec.adcb (x y c : Bool) : Bool × Bool
```

Carry function for bitwise addition.

def

```
BitVec.carry {w : Nat} (i : Nat) (x y : BitVec w) (c : Bool) : Bool
```

carry i x y c returns true if the i carry bit is true when computing $x + y + c$.

def

```
BitVec.mulRec {w : Nat} (x y : BitVec w) (s : Nat) : BitVec w
```

A recurrence that describes multiplication as repeated addition.

This function is useful for bit blasting multiplication.

def

```
BitVec.divRec {w : Nat} (m : Nat) (args : BitVec.DivModArgs w)
  (qr : BitVec.DivModState w) : BitVec.DivModState w
```

A recursive definition of division for bit blasting, in terms of a shift-subtraction circuit.

def

```
BitVec.divSubtractShift {w : Nat} (args : BitVec.DivModArgs w)
  (qr : BitVec.DivModState w) : BitVec.DivModState w
```

One round of the division algorithm. It tries to perform a subtract shift.

This should only be called when `r.msb = false`, so it will not overflow.

def

```
BitVec.shiftLeftRec {w₁ w₂ : Nat} (x : BitVec w₁) (y : BitVec w₂)
  (n : Nat) : BitVec w₁
```

Shifts *x* to the left by the first *n* bits of *y*.

The theorem `BitVec.shiftLeft_eq_shiftLeftRec` proves the equivalence of `(x <<< y)` and `BitVec.shiftLeftRec x y`.

Together with equations `BitVec.shiftLeftRec_zero` and `BitVec.shiftLeftRec_succ`, this allows `BitVec.shiftLeft` to be unfolded into a circuit for bit blasting.

def

```
BitVec.sshiftRightRec {w₁ w₂ : Nat} (x : BitVec w₁) (y : BitVec w₂)
  (n : Nat) : BitVec w₁
```

Shifts *x* arithmetically (signed) to the right by the first *n* bits of *y*.

The theorem `BitVec.sshiftRight_eq_sshiftRightRec` proves the equivalence of `(x.sshiftRight y)` and `BitVec.sshiftRightRec x y`. Together with equations `BitVec.sshiftRightRec_zero`, and `BitVec.sshiftRightRec_succ`, this allows `BitVec.sshiftRight` to be unfolded into a circuit for bit blasting.

def

```
BitVec.ushiftRightRec {w₁ w₂ : Nat} (x : BitVec w₁) (y : BitVec w₂)
  (n : Nat) : BitVec w₁
```

Shifts *x* logically to the right by the first *n* bits of *y*.

The theorem `BitVec.shiftRight_eq_ushiftRightRec` proves the equivalence of `(x >>> y)` and `BitVec.ushiftRightRec`.

Together with equations `BitVec.ushiftRightRec_zero` and `BitVec.ushiftRightRec_succ`, this allows `BitVec.ushiftRight` to be unfolded into a circuit for bit blasting.

19.6. Floating-Point Numbers

Floating-point numbers are a an approximation of the real numbers that are efficiently implemented in computer hardware. Computations that use floating-point numbers are very efficient; however, the nature of the way that they approximate the real numbers is complex, with many corner cases. The IEEE 754 standard, which defines the floating-point format that is used on modern computers, allows hardware designers to make certain choices, and real systems differ in these small details. For example, there are many distinct bit representations of NaN, the indicator that a result is undefined, and some platforms differ with respect to *which* NaN is returned from adding two NaNs.

Lean exposes the underlying platform's floating-point values for use in programming, but they are not encoded in Lean's logic. They are represented by an opaque type. This means that the kernel is not capable of computing with or reasoning about floating-point values without additional axioms. A consequence of this is that equality of floating-point numbers is not decidable. Furthermore, comparisons between floating-point values are decidable, but the code that does so is opaque; in practice, the decision procedure can only be used in compiled code.

Lean provides two floating-point types: `Float` represents 64-bit floating point values, while `Float32` represents 32-bit floating point values. The precision of `Float` does not vary based on the platform that Lean is running on.

type

```
Float : Type
```

64-bit floating-point numbers.

`Float` corresponds to the IEEE 754 *binary64* format (`double` in C or `f64` in Rust). Floating-point numbers are a finite representation of a subset of the real numbers, extended with extra "sentinel" values that represent undefined and infinite results as well as separate positive and negative zeroes. Arithmetic on floating-point numbers approximates the corresponding operations on the real numbers by rounding the results to numbers that are representable, propagating error and infinite values.

Floating-point numbers include subnormal numbers. Their special values are:

- NaN, which denotes a class of "not a number" values that result from operations such as dividing zero by zero, and
- *`Inf`* and *`-Inf`*, which represent positive and infinities that result from dividing non-zero values by zero.

type

```
Float32 : Type
```

32-bit floating-point numbers.

`Float32` corresponds to the IEEE 754 *binary32* format (`float` in C or `f32` in Rust). Floating-point numbers are a finite representation of a subset of the real numbers, extended with extra "sentinel" values that represent undefined and infinite results as well as separate positive and negative zeroes. Arithmetic on floating-point numbers approximates the corresponding operations on the real numbers by rounding the results to numbers that are representable, propagating error and infinite values.

Floating-point numbers include subnormal numbers. Their special values are:

- NaN, which denotes a class of "not a number" values that result from operations such as dividing zero by zero, and
- *`Inf`* and *`-Inf`*, which represent positive and infinities that result from dividing non-zero values by zero.

▼ *Example: No Kernel Reasoning About Floating-Point Numbers*

The Lean kernel can compare expressions of type `Float` for syntactic equality, so `0.0` is definitionally equal to itself.

```
example : (0.0 : Float) = (0.0 : Float) := by rfl
```

Terms that require reduction to become syntactically equal cannot be checked by the kernel:

```
example : (0.0 : Float) = (0.0 + 0.0 : Float) := by rfl
```

```
tactic 'rfl' failed, the left-hand side
  0.0
is not definitionally equal to the right-hand side
  0.0 + 0.0
⊢ 0.0 = 0.0 + 0.0
```

Similarly, the kernel cannot evaluate `Bool`-valued comparisons of floating-point numbers while checking definitional equality:

```
theorem Float.zero_eq_zero_plus_zero :
    ((0.0 : Float) == (0.0 + 0.0 : Float)) = true :=
  by rfl
```

```
tactic 'rfl' failed, the left-hand side
  0.0 == 0.0 + 0.0
is not definitionally equal to the right-hand side
  true
⊢ (0.0 == 0.0 + 0.0) = true
```

However, the `native_decide` tactic can invoke the underlying platform's floating-point primitives that are used by Lean for run-time programs:

```
theorem Float.zero_eq_zero_plus_zero :
    ((0.0 : Float) == (0.0 + 0.0 : Float)) = true := by
```

`native_decide`

This tactic uses the axiom `Lean.ofReduceBool`, which states that the Lean compiler, interpreter and the low-level implementations of built-in operators are trusted in addition to the kernel.

```
#print axioms Float.zero_eq_zero_plus_zero
```

```
'Float.zero_eq_zero_plus_zero' depends on axioms: [propext,
Classical.choice, Lean.ofReduceBool]
```

▼ *Example: Floating-Point Equality Is Not Reflexive*

Floating-point operations may produce NaN values that indicate an undefined result. These values are not comparable with each other; in particular, all comparisons involving NaN will return `false`, including equality.

```
#eval ((0.0 : Float) / 0.0) == ((0.0 : Float) / 0.0)
```

▼ *Example: Floating-Point Equality Is Not a Congruence*

Applying a function to two equal floating-point numbers may not result in equal numbers. In particular, positive and negative zero are distinct values that are equated by floating-point equality, but division by positive or negative zero yields positive or negative infinite values.

```
def neg0 : Float := -0.0

def pos0 : Float := 0.0

#eval (neg0 == pos0, 1.0 / neg0 == 1.0 / pos0)
```

```
(true, false)
```

19.6.1. Syntax

Lean does not have dedicated floating-point literals. Instead, floating-point literals are resolved via the appropriate instances of the `OfScientific` and `Neg` type classes.

▼ *Example: Floating-Point Literals*

The term

```
(-2.523 : Float)
```

is syntactic sugar for

```
(Neg.neg (OfScientific.ofScientific 22523 true 4) : Float)
```

and the term

```
(413.52 : Float32)
```

is syntactic sugar for

```
(OfScientific.ofScientific 41352 true 2 : Float32)
```

19.6.2. API Reference

19.6.2.1. Properties

Floating-point numbers fall into one of three categories:

- Finite numbers are ordinary floating-point values.
- Infinities, which may be positive or negative, result from division by zero.
- NaNs, which are not numbers, result from other undefined operations, such as the square root of a negative number.

opaque

```
Float.isInf : Float → Bool
```

Checks whether a floating-point number is a positive or negative infinite number, but not a finite number or NaN.

This function does not reduce in the kernel. It is compiled to the C operator `isinf`.

opaque

```
Float32.isInf : Float32 → Bool
```

Checks whether a floating-point number is a positive or negative infinite number, but not a finite number or NaN.

This function does not reduce in the kernel. It is compiled to the C operator `isinf`.

opaque

```
Float.isNaN : Float → Bool
```

Checks whether a floating point number is NaN (“not a number”) value.

NaN values result from operations that might otherwise be errors, such as dividing zero by zero.

This function does not reduce in the kernel. It is compiled to the C operator `isnan`.

opaque

```
Float32.isNaN : Float32 → Bool
```

Checks whether a floating point number is `NaN` ("not a number") value.

`NaN` values result from operations that might otherwise be errors, such as dividing zero by zero.

This function does not reduce in the kernel. It is compiled to the C operator `isnan`.

opaque

```
Float.isFinite : Float → Bool
```

Checks whether a floating-point number is finite, that is, whether it is normal, subnormal, or zero, but not infinite or `NaN`.

This function does not reduce in the kernel. It is compiled to the C operator `isfinite`.

opaque

```
Float32.isFinite : Float32 → Bool
```

Checks whether a floating-point number is finite, that is, whether it is normal, subnormal, or zero, but not infinite or `NaN`.

This function does not reduce in the kernel. It is compiled to the C operator `isfinite`.

19.6.2.2. Syntax

These operations exist to support the `OfScientific Float` and `OfScientific Float32` instances and are normally invoked indirectly as a result of a literal value.

opaque

```
Float.ofScientific (m : Nat) (s : Bool) (e : Nat) : Float
```

Constructs a `Float` from the given mantissa, sign, and exponent values.

This function is part of the implementation of the `OfScientific Float` instance that is used to interpret floating-point literals.

opaque

```
Float32.ofScientific (m : Nat) (s : Bool) (e : Nat) : Float32
```

Constructs a `Float32` from the given mantissa, sign, and exponent values.

This function is part of the implementation of the `OfScientific Float32` instance that is used to interpret floating-point literals.

19.6.2.3. Conversions

opaque

`Float.toBits : Float → UInt64`

Bit-for-bit conversion to `UInt64`. Interprets a `Float` as a `UInt64`, ignoring the numeric value and treating the `Float`'s bit pattern as a `UInt64`.

`Float`s and `UInt64`s have the same endianness on all supported platforms. IEEE 754 very precisely specifies the bit layout of floats.

This function is distinct from `Float.toUInt64`, which attempts to preserve the numeric value rather than reinterpreting the bit pattern.

opaque

`Float32.toBits : Float32 → UInt32`

Bit-for-bit conversion to `UInt32`. Interprets a `Float32` as a `UInt32`, ignoring the numeric value and treating the `Float32`'s bit pattern as a `UInt32`.

`Float32`s and `UInt32`s have the same endianness on all supported platforms. IEEE 754 very precisely specifies the bit layout of floats.

This function is distinct from `Float.toUInt32`, which attempts to preserve the numeric value rather than reinterpreting the bit pattern.

This function does not reduce in the kernel.

opaque

`Float.ofBits : UInt64 → Float`

Bit-for-bit conversion from `UInt64`. Interprets a `UInt64` as a `Float`, ignoring the numeric value and treating the `UInt64`'s bit pattern as a `Float`.

`Float`s and `UInt64`s have the same endianness on all supported platforms. IEEE 754 very precisely specifies the bit layout of floats.

This function does not reduce in the kernel.

opaque

`Float32.ofBits : UInt32 → Float32`

Bit-for-bit conversion from `UInt32`. Interprets a `UInt32` as a `Float32`, ignoring the numeric value and treating the `UInt32`'s bit pattern as a `Float32`.

`Float32`s and `UInt32`s have the same endianness on all supported platforms. IEEE 754 very precisely specifies the bit layout of floats.

This function does not reduce in the kernel.

opaque

```
Float.toFloat32 : Float → Float32
```

Converts a 64-bit floating-point number to a 32-bit floating-point number. This may lose precision.

This function does not reduce in the kernel.

opaque

```
Float32.toFloat : Float32 → Float
```

Converts a 32-bit floating-point number to a 64-bit floating-point number.

This function does not reduce in the kernel.

opaque

```
Float.toString : Float → String
```

Converts a floating-point number to a string.

This function does not reduce in the kernel.

opaque

```
Float32.toString : Float32 → String
```

Converts a floating-point number to a string.

This function does not reduce in the kernel.

opaque

```
Float.toUInt8 : Float → UInt8
```

Converts a floating-point number to an 8-bit unsigned integer.

If the given `Float` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt8`. Returns `0` if the `Float` is negative or `NaN`, and returns the largest `UInt8` value (i.e. `UInt8.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float.toInt8 : Float → Int8
```

Truncates a floating-point number to the nearest 8-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int8` (including *Inf*), returns the maximum value of `Int8` (i.e. `Int8.maxValue`). If it is smaller than the minimum value for `Int8` (including −*Inf*), returns the minimum value of `Int8` (i.e. `Int8.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float32.toUInt8 : Float32 → UInt8
```

Converts a floating-point number to an 8-bit unsigned integer.

If the given `Float32` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt8`. Returns `0` if the `Float32` is negative or `NaN`, and returns the largest `UInt8` value (i.e. `UInt8.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float32.toInt8 : Float32 → Int8
```

Truncates a floating-point number to the nearest 8-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int8` (including *Inf*), returns the maximum value of `Int8` (i.e. `Int8.maxValue`). If it is smaller than the minimum value for `Int8` (including −*Inf*), returns the minimum value of `Int8` (i.e. `Int8.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float.toUInt16 : Float → UInt16
```

Converts a floating-point number to a 16-bit unsigned integer.

If the given `Float` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt16`. Returns `0` if the `Float` is negative or NaN, and returns the largest `UInt16` value (i.e. `UInt16.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

`Float.toInt16 : Float → Int16`

Truncates a floating-point number to the nearest 16-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int16` (including *Inf*), returns the maximum value of `Int16` (i.e. `Int16.maxValue`). If it is smaller than the minimum value for `Int16` (including −*Inf*), returns the minimum value of `Int16` (i.e. `Int16.minValue`). If it is NaN, returns `0`.

This function does not reduce in the kernel.

opaque

`Float32.toUInt16 : Float32 → UInt16`

Converts a floating-point number to a 16-bit unsigned integer.

If the given `Float32` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt16`. Returns `0` if the `Float32` is negative or NaN, and returns the largest `UInt16` value (i.e. `UInt16.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

`Float32.toInt16 : Float32 → Int16`

Truncates a floating-point number to the nearest 16-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int16` (including *Inf*), returns the maximum value of `Int16` (i.e. `Int16.maxValue`). If it is smaller than the minimum value for `Int16` (including −*Inf*), returns the minimum value of `Int16` (i.e. `Int16.minValue`). If it is NaN, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float.toUInt32 : Float → UInt32
```

Converts a floating-point number to a 32-bit unsigned integer.

If the given `Float` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt32`. Returns `0` if the `Float` is negative or `NaN`, and returns the largest `UInt32` value (i.e. `UInt32.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float32.toUInt32 : Float32 → UInt32
```

Converts a floating-point number to a 32-bit unsigned integer.

If the given `Float32` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt32`. Returns `0` if the `Float32` is negative or `NaN`, and returns the largest `UInt32` value (i.e. `UInt32.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float.toInt32 : Float → Int32
```

Truncates a floating-point number to the nearest 32-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int32` (including Inf), returns the maximum value of `Int32` (i.e. `Int32.maxValue`). If it is smaller than the minimum value for `Int32` (including $-Inf$), returns the minimum value of `Int32` (i.e. `Int32.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float32.toInt32 : Float32 → Int32
```

Truncates a floating-point number to the nearest 32-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int32` (including Inf), returns the maximum value of `Int32` (i.e. `Int32.maxValue`). If it is smaller than the minimum value for `Int32` (including $-Inf$), returns the minimum value of `Int32` (i.e. `Int32.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float.toUInt64 : Float → UInt64
```

Converts a floating-point number to a 64-bit unsigned integer.

If the given `Float` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt64`. Returns `0` if the `Float` is negative or `NaN`, and returns the largest `UInt64` value (i.e. `UInt64.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float.toInt64 : Float → Int64
```

Truncates a floating-point number to the nearest 64-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int64` (including *Inf*), returns the maximum value of `Int64` (i.e. `Int64.maxValue`). If it is smaller than the minimum value for `Int64` (including −*Inf*), returns the minimum value of `Int64` (i.e. `Int64.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float32.toUInt64 : Float32 → UInt64
```

Converts a floating-point number to a 64-bit unsigned integer.

If the given `Float32` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `UInt64`. Returns `0` if the `Float32` is negative or `NaN`, and returns the largest `UInt64` value (i.e. `UInt64.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float32.toInt64 : Float32 → Int64
```

Truncates a floating-point number to the nearest 64-bit signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `Int64` (including *Inf*), returns the maximum value of `Int64` (i.e. `Int64.maxValue`). If it is smaller than the minimum value for `Int64` (including −*Inf*), returns the minimum value of `Int64` (i.e. `Int64.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float.toUSize : Float → USize
```

Converts a floating-point number to a word-sized unsigned integer.

If the given `Float` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `USize`. Returns `0` if the `Float` is negative or `NaN`, and returns the largest `USize` value (i.e. `USize.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float32.toUSize : Float32 → USize
```

Converts a floating-point number to a word-sized unsigned integer.

If the given `Float32` is non-negative, truncates the value to a positive integer, rounding down and clamping to the range of `USize`. Returns `0` if the `Float32` is negative or `NaN`, and returns the largest `USize` value (i.e. `USize.size - 1`) if the float is larger than it.

This function does not reduce in the kernel.

opaque

```
Float.toISize : Float → ISize
```

Truncates a floating-point number to the nearest word-sized signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `ISize` (including *Inf*), returns the maximum value of `ISize` (i.e. `ISize.maxValue`). If it is smaller than the minimum value for `ISize` (including *−Inf*), returns the minimum value of `ISize` (i.e. `ISize.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

opaque

```
Float32.toISize : Float32 → ISize
```

Truncates a floating-point number to the nearest word-sized signed integer, rounding towards zero.

If the `Float` is larger than the maximum value for `ISize` (including *Inf*), returns the maximum value of `ISize` (i.e. `ISize.maxValue`). If it is smaller than the minimum value for `ISize` (including *−Inf*), returns the minimum value of `ISize` (i.e. `ISize.minValue`). If it is `NaN`, returns `0`.

This function does not reduce in the kernel.

def

```
Float.ofInt : Int → Float
```

Converts an integer into the closest-possible 64-bit floating-point number, or positive or negative infinite floating-point value if the range of `Float` is exceeded.

def

```
Float32.ofInt : Int → Float32
```

Converts an integer into the closest-possible 32-bit floating-point number, or positive or negative infinite floating-point value if the range of `Float32` is exceeded.

def

```
Float.ofNat (n : Nat) : Float
```

Converts a natural number into the closest-possible 64-bit floating-point number, or an infinite floating-point value if the range of `Float` is exceeded.

def

```
Float32.ofNat (n : Nat) : Float32
```

Converts a natural number into the closest-possible 32-bit floating-point number, or an infinite floating-point value if the range of `Float32` is exceeded.

def

```
Float.ofBinaryScientific (m : Nat) (e : Int) : Float
```

Computes `m * 2^e`.

def

```
Float32.ofBinaryScientific (m : Nat) (e : Int) : Float32
```

Computes `m * 2^e`.

opaque

```
Float.frExp : Float → Float × Int
```

Splits the given float *x* into a significand/exponent pair `(s, i)` such that `x = s * 2^i` where `s ∈ (-1;-0.5] ∪ [0.5; 1)`. Returns an undefined value if *x* is not finite.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `frexp`.

opaque

`Float32.frExp : Float32 → Float32 × Int`

Splits the given float *x* into a significand/exponent pair `(s, i)` such that `x = s * 2^i` where `s ∈ (-1;-0.5] ∪ [0.5; 1)`. Returns an undefined value if *x* is not finite.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `frexp`.

19.6.2.4. Comparisons

opaque

`Float.beq (a b : Float) : Bool`

Checks whether two floating-point numbers are equal according to IEEE 754.

Floating-point equality does not correspond with propositional equality. In particular, it is not reflexive since `NaN != NaN`, and it is not a congruence because `0.0 == -0.0`, but `1.0 / 0.0 != 1.0 / -0.0`.

This function does not reduce in the kernel. It is compiled to the C equality operator.

opaque

`Float32.beq (a b : Float32) : Bool`

Checks whether two floating-point numbers are equal according to IEEE 754.

Floating-point equality does not correspond with propositional equality. In particular, it is not reflexive since `NaN != NaN`, and it is not a congruence because `0.0 == -0.0`, but `1.0 / 0.0 != 1.0 / -0.0`.

This function does not reduce in the kernel. It is compiled to the C equality operator.

19.6.2.4.1. Inequalities

The decision procedures for inequalities are opaque constants in the logic. They can only be used via the `Lean.ofReduceBool` axiom, e.g. via the **`native_decide`** tactic.

def

```
Float.le : Float → Float → Prop
```

Non-strict inequality of floating-point numbers. Typically used via the ≤ operator.

def

```
Float32.le : Float32 → Float32 → Prop
```

Non-strict inequality of floating-point numbers. Typically used via the ≤ operator.

def

```
Float.lt : Float → Float → Prop
```

Strict inequality of floating-point numbers. Typically used via the < operator.

def

```
Float32.lt : Float32 → Float32 → Prop
```

Strict inequality of floating-point numbers. Typically used via the < operator.

opaque

```
Float.decLe (a b : Float) : Decidable (a ≤ b)
```

Compares two floating point numbers for non-strict inequality.

This function does not reduce in the kernel. It is compiled to the C inequality operator.

opaque

```
Float32.decLe (a b : Float32) : Decidable (a ≤ b)
```

Compares two floating point numbers for non-strict inequality.

This function does not reduce in the kernel. It is compiled to the C inequality operator.

opaque

```
Float.decLt (a b : Float) : Decidable (a ≤ b)
```

Compares two floating point numbers for strict inequality.

This function does not reduce in the kernel. It is compiled to the C inequality operator.

opaque

```
Float32.decLt (a b : Float32) : Decidable (a ≤ b)
```

Compares two floating point numbers for strict inequality.

This function does not reduce in the kernel. It is compiled to the C inequality operator.

19.6.2.5. Arithmetic

Arithmetic operations on floating-point values are typically invoked via the `Add Float`, `Sub Float`, `Mul Float`, `Div Float`, and `HomogeneousPow Float` instances, along with the corresponding `Float32` instances.

opaque

```
Float.add : Float → Float → Float
```

Adds two 64-bit floating-point numbers according to IEEE 754. Typically used via the + operator.

This function does not reduce in the kernel. It is compiled to the C addition operator.

opaque

```
Float32.add : Float32 → Float32 → Float32
```

Adds two 32-bit floating-point numbers according to IEEE 754. Typically used via the + operator.

This function does not reduce in the kernel. It is compiled to the C addition operator.

opaque

```
Float.sub : Float → Float → Float
```

Subtracts 64-bit floating-point numbers according to IEEE 754. Typically used via the − operator.

This function does not reduce in the kernel. It is compiled to the C subtraction operator.

opaque

```
Float32.sub : Float32 → Float32 → Float32
```

Subtracts 32-bit floating-point numbers according to IEEE 754. Typically used via the − operator.

This function does not reduce in the kernel. It is compiled to the C subtraction operator.

opaque

```
Float.mul : Float → Float → Float
```

Multiplies 64-bit floating-point numbers according to IEEE 754. Typically used via the `*` operator.

This function does not reduce in the kernel. It is compiled to the C multiplication operator.

opaque

```
Float32.mul : Float32 → Float32 → Float32
```

Multiplies 32-bit floating-point numbers according to IEEE 754. Typically used via the `*` operator.

This function does not reduce in the kernel. It is compiled to the C multiplication operator.

opaque

```
Float.div : Float → Float → Float
```

Divides 64-bit floating-point numbers according to IEEE 754. Typically used via the `/` operator.

In Lean, division by zero typically yields zero. For `Float`, it instead yields either *`Inf`*, *`-Inf`*, or `NaN`.

This function does not reduce in the kernel. It is compiled to the C division operator.

opaque

```
Float32.div : Float32 → Float32 → Float32
```

Divides 32-bit floating-point numbers according to IEEE 754. Typically used via the `/` operator.

In Lean, division by zero typically yields zero. For `Float32`, it instead yields either *`Inf`*, *`-Inf`*, or `NaN`.

This function does not reduce in the kernel. It is compiled to the C division operator.

opaque

```
Float.pow : Float → Float → Float
```

Raises one floating-point number to the power of another. Typically used via the `^` operator.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `pow`.

opaque

```
Float32.pow : Float32 → Float32 → Float32
```

Raises one floating-point number to the power of another. Typically used via the ^ operator.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `powf`.

opaque

```
Float.exp (x : Float) : Float
```

Computes the exponential e^x of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `exp`.

opaque

```
Float32.exp : Float32 → Float32
```

Computes the exponential e^x of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `expf`.

opaque

```
Float.exp2 (x : Float) : Float
```

Computes the base-2 exponential 2^x of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `exp2`.

opaque

```
Float32.exp2 : Float32 → Float32
```

Computes the base-2 exponential 2^x of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `exp2f`.

19.6.2.5.1. Roots

Computing the square root of a negative number yields `NaN`.

opaque

```
Float.sqrt : Float → Float
```

Computes the square root of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `sqrt`.

opaque

```
Float32.sqrt : Float32 → Float32
```

Computes the square root of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `sqrtf`.

opaque

```
Float.cbrt : Float → Float
```

Computes the cube root of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `cbrt`.

opaque

```
Float32.cbrt : Float32 → Float32
```

Computes the cube root of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `cbrtf`.

19.6.2.6. Logarithms

opaque

```
Float.log (x : Float) : Float
```

Computes the natural logarithm `ln x` of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `log`.

opaque

```
Float32.log : Float32 → Float32
```

Computes the natural logarithm `ln x` of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `logf`.

opaque

`Float.log10 : Float → Float`

Computes the base-10 logarithm of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `log10`.

opaque

`Float32.log10 : Float32 → Float32`

Computes the base-10 logarithm of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `log10f`.

opaque

`Float.log2 : Float → Float`

Computes the base-2 logarithm of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `log2`.

opaque

`Float32.log2 : Float32 → Float32`

Computes the base-2 logarithm of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `log2f`.

19.6.2.7. Scaling

opaque

`Float.scaleB (x : Float) (i : Int) : Float`

Efficiently computes `x * 2^i`.

This function does not reduce in the kernel.

opaque

```
Float32.scaleB (x : Float32) (i : Int) : Float32
```

Efficiently computes `x * 2^i`.

This function does not reduce in the kernel.

19.6.2.8. Rounding

opaque

```
Float.round : Float → Float
```

Rounds to the nearest integer, rounding away from zero at half-way points.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `round`.

opaque

```
Float32.round : Float32 → Float32
```

Rounds to the nearest integer, rounding away from zero at half-way points.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `roundf`.

opaque

```
Float.floor : Float → Float
```

Computes the floor of a floating-point number, which is the largest integer that's no larger than the given number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `floor`.

Examples:

- `Float.floor 1.5 = 1`
- `Float.floor (-1.5) = (-2)`

opaque

```
Float32.floor : Float32 → Float32
```

Computes the floor of a floating-point number, which is the largest integer that's no larger than the given number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `floorf`.

Examples:

- `Float32.floor 1.5 = 1`
- `Float32.floor (-1.5) = (-2)`

opaque

`Float.ceil : Float → Float`

Computes the ceiling of a floating-point number, which is the smallest integer that's no smaller than the given number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `ceil`.

Examples:

- `Float.ceil 1.5 = 2`
- `Float.ceil (-1.5) = (-1)`

opaque

`Float32.ceil : Float32 → Float32`

Computes the ceiling of a floating-point number, which is the smallest integer that's no smaller than the given number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `ceilf`.

Examples:

- `Float32.ceil 1.5 = 2`
- `Float32.ceil (-1.5) = (-1)`

19.6.2.9. Trigonometry

19.6.2.9.1. Sine

opaque

`Float.sin : Float → Float`

Computes the sine of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `sin`.

opaque

```
Float32.sin : Float32 → Float32
```

Computes the sine of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `sinf`.

opaque

```
Float.sinh : Float → Float
```

Computes the hyperbolic sine of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `sinh`.

opaque

```
Float32.sinh : Float32 → Float32
```

Computes the hyperbolic sine of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `sinhf`.

opaque

```
Float.asin : Float → Float
```

Computes the arc sine (inverse sine) of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `asin`.

opaque

```
Float32.asin : Float32 → Float32
```

Computes the arc sine (inverse sine) of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `asinf`.

opaque

```
Float.asinh : Float → Float
```

Computes the hyperbolic arc sine (inverse sine) of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `asinh`.

opaque

```
Float32.asinh : Float32 → Float32
```

Computes the hyperbolic arc sine (inverse sine) of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `asinhf`.

19.6.2.9.2. Cosine

opaque

```
Float.cos : Float → Float
```

Computes the cosine of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `cos`.

opaque

```
Float32.cos : Float32 → Float32
```

Computes the cosine of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `cosf`.

opaque

```
Float.cosh : Float → Float
```

Computes the hyperbolic cosine of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `cosh`.

opaque

```
Float32.cosh : Float32 → Float32
```

Computes the hyperbolic cosine of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `coshf`.

opaque

```
Float.acos : Float → Float
```

Computes the arc cosine (inverse cosine) of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `acos`.

opaque

```
Float32.acos : Float32 → Float32
```

Computes the arc cosine (inverse cosine) of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `acosf`.

opaque

```
Float.acosh : Float → Float
```

Computes the hyperbolic arc cosine (inverse cosine) of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `acosh`.

opaque

```
Float32.acosh : Float32 → Float32
```

Computes the hyperbolic arc cosine (inverse cosine) of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `acoshf`.

19.6.2.9.3. Tangent

opaque

```
Float.tan : Float → Float
```

Computes the tangent of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `tan`.

opaque

```
Float32.tan : Float32 → Float32
```

Computes the tangent of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `tanf`.

opaque

```
Float.tanh : Float → Float
```

Computes the hyperbolic tangent of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `tanh`.

opaque

```
Float32.tanh : Float32 → Float32
```

Computes the hyperbolic tangent of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `tanhf`.

opaque

```
Float.atan : Float → Float
```

Computes the arc tangent (inverse tangent) of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `atan`.

opaque

```
Float32.atan : Float32 → Float32
```

Computes the arc tangent (inverse tangent) of a floating-point number in radians.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `atanf`.

opaque

```
Float.atanh : Float → Float
```

Computes the hyperbolic arc tangent (inverse tangent) of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `atanh`.

opaque

```
Float32.atanh : Float32 → Float32
```

Computes the hyperbolic arc tangent (inverse tangent) of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `atanhf`.

opaque

```
Float.atan2 (y x : Float) : Float
```

Computes the arc tangent (inverse tangent) of y / x in radians, in the range $-\pi$–π. The signs of the arguments determine the quadrant of the result.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `atan2`.

opaque

```
Float32.atan2 : Float32 → Float32 → Float32
```

Computes the arc tangent (inverse tangent) of y / x in radians, in the range $-\pi$–π. The signs of the arguments determine the quadrant of the result.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `atan2f`.

19.6.2.10. Negation and Absolute Value

opaque

```
Float.abs : Float → Float
```

Computes the absolute value of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `fabs`.

opaque

```
Float32.abs : Float32 → Float32
```

Computes the absolute value of a floating-point number.

This function does not reduce in the kernel. It is implemented in compiled code by the C function `fabsf`.

opaque

```
Float.neg : Float → Float
```

Negates 64-bit floating-point numbers according to IEEE 754. Typically used via the − prefix operator.

This function does not reduce in the kernel. It is compiled to the C negation operator.

opaque

```
Float32.neg : Float32 → Float32
```

Negates 32-bit floating-point numbers according to IEEE 754. Typically used via the − prefix operator.

This function does not reduce in the kernel. It is compiled to the C negation operator.

19.7. Characters

Characters are represented by the type `Char`, which may be any Unicode scalar value. While strings are UTF-8-encoded arrays of bytes, characters are represented by full 32-bit values. Lean provides special syntax for character literals.

19.7.1. Logical Model

From the perspective of Lean's logic, characters consist of a 32-bit unsigned integer paired with a proof that it is a valid Unicode scalar value.

structure

```
Char : Type
```

Characters are Unicode scalar values.

Constructor

`Char.mk`

Fields

`val : UInt32`

The underlying Unicode scalar value as a `UInt32`.

`valid : self.val.isValidChar`

The value must be a legal scalar value.

19.7.2. Run-Time Representation

As a trivial wrapper, characters are represented identically to `UInt32`. In particular, characters are represented as 32-bit immediate values in monomorphic contexts. In other words, a field of a constructor or structure of type `Char` does not require indirection to access. In polymorphic contexts, characters are boxed.

19.7.3. Syntax

Character literals consist of a single character or an escape sequence enclosed in single quotes (`'`, Unicode `'APOSTROPHE' (U+0027)`). Between these single quotes, the character literal may contain character other that `'`, including newlines, which are included literally (with the caveat that all newlines in a Lean source file are interpreted as `'\n'`, regardless of file encoding and platform). Special characters may be escaped with a backslash, so `'\''` is a character literal that contains a single quote. The following forms of escape sequences are accepted:

`\r`, `\n`, `\t`, `\\`, `\"`, `\'`

These escape sequences have the usual meaning, mapping to CR, LF, tab, backslash, double quote, and single quote, respectively.

`\xNN`

When NN is a sequence of two hexadecimal digits, this escape denotes the character whose Unicode code point is indicated by the two-digit hexadecimal code.

`\uNNNN`

When NN is a sequence of two hexadecimal digits, this escape denotes the character whose Unicode code point is indicated by the four-digit hexadecimal code.

19.7.4. API Reference

19.7.4.1. Conversions

def

`Char.ofNat (n : Nat) : Char`

Converts a `Nat` into a `Char`. If the `Nat` does not encode a valid Unicode scalar value, `'\0'` is returned instead.

def

```
Char.toNat (c : Char) : Nat
```

The character's Unicode code point as a `Nat`.

def

```
Char.isValidCharNat (n : Nat) : Prop
```

True for natural numbers that are valid Unicode scalar values.

def

```
Char.ofUInt8 (n : UInt8) : Char
```

Converts an 8-bit unsigned integer into a character.

The integer's value is interpreted as a Unicode code point.

def

```
Char.toUInt8 (c : Char) : UInt8
```

Converts a character into a `UInt8` that contains its code point.

If the code point is larger than 255, it is truncated (reduced modulo 256).

There are two ways to convert a character to a string. `Char.toString` converts a character to a singleton string that consists of only that character, while `Char.quote` converts the character to a string representation of the corresponding character literal.

def

```
Char.toString (c : Char) : String
```

Constructs a singleton string that contains only the provided character.

Examples:

- `'L'.toString = "L"`
- `'"'.toString = "\""`

def

```
Char.quote (c : Char) : String
```

Quotes the character to its representation as a character literal, surrounded by single quotes and escaped as necessary.

Examples:

- `'L'.quote = "'L'"`
- `'"'.quote = "'\\\"'"`

▼ *Example: From Characters to Strings*

`Char.toString` produces a string that contains only the character in question:

```
#eval 'e'.toString
```
```
"e"
```
```
#eval '\x65'.toString
```
```
"e"
```
```
#eval '"'.toString
```
```
"\""
```

`Char.quote` produces a string that contains a character literal, suitably escaped:

```
#eval 'e'.quote
```
```
"'e'"
```
```
#eval '\x65'.quote
```
```
"'e'"
```
```
#eval '"'.quote
```
```
"'\\\"'"
```

19.7.4.2. Character Classes

def

```
Char.isAlpha (c : Char) : Bool
```

Returns `true` if the character is an ASCII letter.

The ASCII letters are the following: `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`.

def

```
Char.isAlphanum (c : Char) : Bool
```

Returns `true` if the character is an ASCII letter or digit.

The ASCII letters are the following: `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz`. The ASCII digits are the following: `0123456789`.

def

```
Char.isDigit (c : Char) : Bool
```

Returns `true` if the character is an ASCII digit.

The ASCII digits are the following: `0123456789`.

def

```
Char.isLower (c : Char) : Bool
```

Returns `true` if the character is a lowercase ASCII letter.

The lowercase ASCII letters are the following: `abcdefghijklmnopqrstuvwxyz`.

def

```
Char.isUpper (c : Char) : Bool
```

Returns `true` if the character is a uppercase ASCII letter.

The uppercase ASCII letters are the following: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`.

def

```
Char.isWhitespace (c : Char) : Bool
```

Returns `true` if the character is a space (`' '`, *U*+0020), a tab (`'\t'`, *U*+0009), a carriage return (`'\r'`, U+000D), or a newline (`'\n'`, U+000A).

19.7.4.3. Case Conversion

def

```
Char.toUpper (c : Char) : Char
```

Converts a lowercase ASCII letter to the corresponding uppercase letter. Letters outside the ASCII alphabet are returned unchanged.

The lowercase ASCII letters are the following: `abcdefghijklmnopqrstuvwxyz`.

def

```
Char.toLower (c : Char) : Char
```

Converts an uppercase ASCII letter to the corresponding lowercase letter. Letters outside the ASCII alphabet are returned unchanged.

The uppercase ASCII letters are the following: `ABCDEFGHIJKLMNOPQRSTUVWXYZ`.

19.7.4.4. Comparisons

def

```
Char.le (a b : Char) : Prop
```

One character is less than or equal to another if its code point is less than or equal to the other's.

def

```
Char.lt (a b : Char) : Prop
```

One character is less than another if its code point is strictly less than the other's.

19.7.4.5. Unicode

def

```
Char.utf8Size (c : Char) : Nat
```

Returns the number of bytes required to encode this `Char` in UTF-8.

def

```
Char.utf16Size (c : Char) : UInt32
```

Returns the number of bytes required to encode this `Char` in UTF-16.

19.8. Strings

Strings represent Unicode text. Strings are specially supported by Lean:

- They have a *logical model* that specifies their behavior in terms of lists of characters, which specifies the meaning of each operation on strings.
- They have an optimized run-time representation in compiled code, as packed arrays of bytes that encode the string as UTF-8, and the Lean runtime specially optimizes string operations.
- There is string literal syntax for writing strings.

The fact that strings are internally represented as UTF-8-encoded byte arrays is visible in the API:

- There is no operation to project a particular character out of the string, as this would be a performance trap. Use a `String.Iterator` in a loop instead of a `Nat`.
- Strings are indexed by `String.Pos`, which internally records *byte counts* rather than *character counts*, and thus takes constant time. Aside from `0`, these should not be constructed directly, but rather updated using `String.next` and `String.prev`.

19.8.1. Logical Model

structure

```
String : Type
```

A string is a sequence of Unicode code points.

At runtime, strings are represented by dynamic arrays of bytes using the UTF-8 encoding. Both the size in bytes (`String.utf8ByteSize`) and in characters (`String.length`) are cached and take constant time. Many operations on strings perform in-place modifications when the reference to the string is unique.

Constructor

`String.mk`

Pack a `List Char` into a `String`. This function is overridden by the compiler and is O(n) in the length of the list.

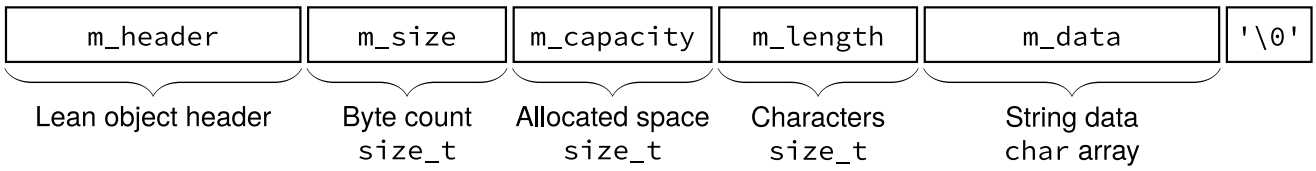
Fields

`data : List Char`

Unpack `String` into a `List Char`. This function is overridden by the compiler and is O(n) in the length of the list.

The logical model of strings in Lean is a structure that contains a single field, which is a list of characters. This is convenient when specifying and proving properties of string-processing functions at a low level.

19.8.2. Run-Time Representation



Memory layout of strings

Strings are represented as dynamic arrays of bytes, encoded in UTF-8. After the object header, a string contains:

byte count
: The number of bytes that currently contain valid string data

capacity
: The number of bytes presently allocated for the string

length
: The length of the encoded string, which may be shorter than the byte count due to UTF-8 multi-byte characters

data
: The actual character data in the string, null-terminated

Many string functions in the Lean runtime check whether they have exclusive access to their argument by consulting the reference count in the object header. If they do, and the string's capacity is sufficient, then the existing string can be mutated rather than allocating fresh memory. Otherwise, a new string must be allocated.

19.8.2.1. Performance Notes

Despite the fact that they appear to be an ordinary constructor and projection, `String.mk` and `String.data` take **time linear in the length of the string**. This is because they must implement the conversions between lists of characters and packed arrays of bytes, which must necessarily visit each character.

19.8.3. Syntax

Lean has three kinds of string literals: ordinary string literals, interpolated string literals, and raw string literals.

19.8.3.1. String Literals

String literals begin and end with a double-quote character `"`. Between these characters, they may contain any other character, including newlines, which are included literally (with the caveat that all newlines in a Lean source file are interpreted as `'\n'`, regardless of file encoding and platform).

Special characters that cannot otherwise be written in string literals may be escaped with a backslash, so `"\"Quotes\""` is a string literal that begins and ends with double quotes. The following forms of escape sequences are accepted:

`\r`, `\n`, `\t`, `\\`, `\"`, `\'`
: These escape sequences have the usual meaning, mapping to CR, LF, tab, backslash, double quote, and single quote, respectively.

`\xNN`
: When NN is a sequence of two hexadecimal digits, this escape denotes the character whose Unicode code point is indicated by the two-digit hexadecimal code.

`\uNNNN`
: When NN is a sequence of two hexadecimal digits, this escape denotes the character whose Unicode code point is indicated by the four-digit hexadecimal code.

String literals may contain *gaps*. A gap is indicated by an escaped newline, with no intervening characters between the escaping backslash and the newline. In this case, the string denoted by the literal is missing the newline and all leading whitespace from the next line. String gaps may not precede lines that contain only whitespace.

Here, `str1` and `str2` are the same string:

```
def str1 := "String with \
             a gap"
def str2 := "String with a gap"

example : str1 = str2 := rfl
```

If the line following the gap is empty, the string is rejected:

```
1 def str3 := "String with \
2
3             a gap"
```

The parser error is:

```
<example>:2:0-3:0: unexpected additional newline in string gap
```

19.8.3.2. Interpolated Strings

Preceding a string literal with `s!` causes it to be processed as an *interpolated string*, in which regions of the string surrounded by `{` and `}` characters are parsed and interpreted as Lean expressions. Interpolated strings are interpreted by appending the string that precedes the interpolation, the expression (with an added call to `toString` surrounding it), and the string that follows the interpolation.

For example:

```
example :
    s!"1 + 1 = {1 + 1}\n" =
    "1 + 1 = " ++ toString (1 + 1) ++ "\n" :=
  rfl
```

Preceding a literal with `m!` causes the interpolation to result in an instance of `MessageData`, the compiler's internal data structure for messages to be shown to users.

19.8.3.3. Raw String Literals

In raw string literals, there are no escape sequences or gaps, and each character denotes itself exactly. Raw string literals are preceded by `r`, followed by zero or more hash characters (`#`) and a double quote `"`. The string literal is completed at a double quote that is followed by *the same number* of hash characters. For example, they can be used to avoid the need to double-escape certain characters:

```
example : r"\t" = "\\t" := rfl
#eval r"Write backslash in a string using '\\\\'"
```

The `#eval` yields:

```
"Write backslash in a string using '\\\\\\\\'"
```

Including hash marks allows the strings to contain unescaped quotes:

```
example :
    r#"This is "literally" quoted"# =
    "This is \"literally\" quoted" :=
  rfl
```

Adding sufficiently many hash marks allows any raw literal to be written literally:

```
example :
    r##"This is r#"literally"# quoted"## =
    "This is r#\"literally\"# quoted" :=
  rfl
```

19.8.4. API Reference

19.8.4.1. Constructing

def

```
String.singleton (c : Char) : String
```

Returns a new string that contains only the character `c`.

Because strings are encoded in UTF-8, the resulting string may take multiple bytes.

Examples:

- `String.singleton 'L' = "L"`

- `String.singleton ' ' = " "`
- `String.singleton '"' = "\""`
- `String.singleton '𝒫' = "𝒫"`

def

`String.append : String → String → String`

Appends two strings. Usually accessed via the ++ operator.

The internal implementation will perform destructive updates if the string is not shared.

Examples:

- `"abc".append "def" = "abcdef"`
- `"abc" ++ "def" = "abcdef"`
- `"" ++ "" = ""`

def

`String.join (l : List String) : String`

Appends all the strings in a list of strings, in order.

Use `String.intercalate` to place a separator string between the strings in a list.

Examples:

- `String.join ["gr", "ee", "n"] = "green"`
- `String.join ["b", "", "l", "", "ue"] = "red"`
- `String.join [] = ""`

def

`String.intercalate (s : String) : List String → String`

Appends the strings in a list of strings, placing the separator `s` between each pair.

Examples:

- `", ".intercalate ["red", "green", "blue"] = "red, green, blue"`
- `" and ".intercalate ["tea", "coffee"] = "tea and coffee"`
- `" | ".intercalate ["M", "", "N"] = "M |  | N"`

19.8.4.2. Conversions

def

```
String.toList (s : String) : List Char
```

Converts a string to a list of characters.

Even though the logical model of strings is as a structure that wraps a list of characters, this operation takes time and space linear in the length of the string. At runtime, strings are represented as dynamic arrays of bytes.

Examples:

- `"abc".toList = ['a', 'b', 'c']`
- `"".toList = []`
- `"\n".toList = ['\n']`

def

```
String.isNat (s : String) : Bool
```

Checks whether the string can be interpreted as the decimal representation of a natural number.

A string can be interpreted as a decimal natural number if it is not empty and all the characters in it are digits.

Use `String.toNat?` or `String.toNat!` to convert such a string to a natural number.

Examples:

- `"".isNat = false`
- `"0".isNat = true`
- `"5".isNat = true`
- `"05".isNat = true`
- `"587".isNat = true`
- `"-587".isNat = false`
- `" 5".isNat = false`
- `"2+3".isNat = false`
- `"0xff".isNat = false`

def

```
String.toNat? (s : String) : Option Nat
```

Interprets a string as the decimal representation of a natural number, returning it. Returns `none` if the string does not contain a decimal natural number.

A string can be interpreted as a decimal natural number if it is not empty and all the characters in it are digits.

Use `String.isNat` to check whether `String.toNat?` would return `some`. `String.toNat!` is an alternative that panics instead of returning `none` when the string is not a natural number.

Examples:

- `"".toNat? = none`
- `"0".toNat? = some 0`
- `"5".toNat? = some 5`
- `"587".toNat? = some 587`
- `"-587".toNat? = none`
- `" 5".toNat? = none`
- `"2+3".toNat? = none`
- `"0xff".toNat? = none`

def

```
String.toNat! (s : String) : Nat
```

Interprets a string as the decimal representation of a natural number, returning it. Panics if the string does not contain a decimal natural number.

A string can be interpreted as a decimal natural number if it is not empty and all the characters in it are digits.

Use `String.isNat` to check whether `String.toNat!` would return a value. `String.toNat?` is a safer alternative that returns `none` instead of panicking when the string is not a natural number.

Examples:

- `"0".toNat! = 0`
- `"5".toNat! = 5`
- `"587".toNat! = 587`

def

```
String.isInt (s : String) : Bool
```

Checks whether the string can be interpreted as the decimal representation of an integer.

A string can be interpreted as a decimal integer if it only consists of at least one decimal digit and optionally − in front. Leading + characters are not allowed.

Use `String.toInt?` or `String.toInt!` to convert such a string to an integer.

Examples:

- `"".isInt = false`
- `"-".isInt = false`
- `"0".isInt = true`
- `"-0".isInt = true`
- `"5".isInt = true`
- `"587".isInt = true`
- `"-587".isInt = true`
- `"+587".isInt = false`
- `" 5".isInt = false`
- `"2-3".isInt = false`
- `"0xff".isInt = false`

def

```
String.toInt? (s : String) : Option Int
```

Interprets a string as the decimal representation of an integer, returning it. Returns `none` if the string does not contain a decimal integer.

A string can be interpreted as a decimal integer if it only consists of at least one decimal digit and optionally − in front. Leading + characters are not allowed.

Use `String.isInt` to check whether `String.toInt?` would return `some`. `String.toInt!` is an alternative that panics instead of returning `none` when the string is not an integer.

Examples:

- `"".toInt? = none`
- `"-".toInt? = none`
- `"0".toInt? = some 0`
- `"5".toInt? = some 5`
- `"-5".toInt? = some (-5)`

- `"587".toInt? = some 587`
- `"-587".toInt? = some (-587)`
- `" 5".toInt? = none`
- `"2-3".toInt? = none`
- `"0xff".toInt? = none`

def

`String.toInt! (s : String) : Int`

Interprets a string as the decimal representation of an integer, returning it. Panics if the string does not contain a decimal integer.

A string can be interpreted as a decimal integer if it only consists of at least one decimal digit and optionally − in front. Leading + characters are not allowed.

Use `String.isInt` to check whether `String.toInt!` would return a value. `String.toInt?` is a safer alternative that returns `none` instead of panicking when the string is not an integer.

Examples:

- `"0".toInt! = 0`
- `"5".toInt! = 5`
- `"587".toInt! = 587`
- `"-587".toInt! = -587`

def

`String.toFormat (s : String) : Std.Format`

Converts a string to a pretty-printer document, replacing newlines in the string with `Std.Format.line`.

19.8.4.3. Properties

def

`String.isEmpty (s : String) : Bool`

Checks whether a string is empty.

Empty strings are equal to `""` and have length and end position `0`.

Examples:

- `"".isEmpty = true`
- `"empty".isEmpty = false`
- `" ".isEmpty = false`

def

`String.length : String → Nat`

Returns the length of a string in Unicode code points.

Examples:

- `"".length = 0`
- `"abc".length = 3`
- `"LƎ∀N".length = 4`

19.8.4.4. Positions

structure

`String.Pos : Type`

A byte position in a `String`, according to its UTF-8 encoding.

Character positions (counting the Unicode code points rather than bytes) are represented by plain `Nat`s. Indexing a `String` by a `String.Pos` takes constant time, while character positions need to be translated internally to byte positions, which takes linear time.

A byte position `p` is *valid* for a string `s` if `0 ≤ p ≤ s.endPos` and `p` lies on a UTF-8 character boundary.

Constructor

`String.Pos.mk`

Fields

`byteIdx : Nat`

Get the underlying byte index of a `String.Pos`

def

`String.Pos.isValid (s : String) (p : String.Pos) : Bool`

Returns `true` if *p* is a valid UTF-8 position in the string *s*.

This means that $p \le s$.endPos and p lies on a UTF-8 character boundary. At runtime, this operation takes constant time.

Examples:

- String.Pos.isValid "abc" ⟨0⟩ = true
- String.Pos.isValid "abc" ⟨1⟩ = true
- String.Pos.isValid "abc" ⟨3⟩ = true
- String.Pos.isValid "abc" ⟨4⟩ = false
- String.Pos.isValid "𝒫(A)" ⟨0⟩ = true
- String.Pos.isValid "𝒫(A)" ⟨1⟩ = false
- String.Pos.isValid "𝒫(A)" ⟨2⟩ = false
- String.Pos.isValid "𝒫(A)" ⟨3⟩ = false
- String.Pos.isValid "𝒫(A)" ⟨4⟩ = true

def

```
String.atEnd : String → String.Pos → Bool
```

Returns true if a specified byte position is greater than or equal to the position which points to the end of a string. Otherwise, returns false.

Examples:

- (0 |> "abc".next |> "abc".next |> "abc".atEnd) = false
- (0 |> "abc".next |> "abc".next |> "abc".next |> "abc".next |> "abc".atEnd) = true
- (0 |> "LƎ∀N".next |> "LƎ∀N".next |> "LƎ∀N".next |> "LƎ∀N".atEnd) = false
- (0 |> "LƎ∀N".next |> "LƎ∀N".next |> "LƎ∀N".next |> "LƎ∀N".next |> "LƎ∀N".atEnd) = true
- "abc".atEnd ⟨4⟩ = true
- "LƎ∀N".atEnd ⟨7⟩ = false
- "LƎ∀N".atEnd ⟨8⟩ = true

def

```
String.endPos (s : String) : String.Pos
```

A UTF-8 byte position that points at the end of a string, just after the last character.

- "abc".endPos = ⟨3⟩

- `"LƎ∀N".endPos = ⟨8⟩`

def

`String.next (s : String) (p : String.Pos) : String.Pos`

Returns the next position in a string after position *p*. The result is unspecified if *p* is not a valid position or if *p* `= s.endPos`.

A run-time bounds check is performed to determine whether *p* is at the end of the string. If a bounds check has already been performed, use `String.next'` to avoid a repeated check.

Some examples where the result is unspecified:

- `"abc".next ⟨3⟩`, since `3 = "abc".endPos`
- `"LƎ∀N".next ⟨2⟩`, since 2 points into the middle of a multi-byte UTF-8 character

Examples:

- `"abc".get ("abc".next 0) = 'b'`
- `"LƎ∀N".get (0 |> "LƎ∀N".next |> "LƎ∀N".next) = '∀'`

def

`String.next' (s : String) (p : String.Pos) (h : ¬s.atEnd p = true) : String.Pos`

Returns the next position in a string after position *p*. The result is unspecified if *p* is not a valid position.

Requires evidence, *h*, that *p* is within bounds. No run-time bounds check is performed, as in `String.next`.

A typical pattern combines `String.next'` with a dependent **`if`**-expression to avoid the overhead of an additional bounds check. For example:

```
def next? (s: String) (p : String.Pos) : Option Char :=
  if h : s.atEnd p then none else s.get (s.next' p h)
```

Example:

- **`let`** *abc* `:= "abc"; abc.get (abc.next' 0 (`**`by decide`**`)) = 'b'`

def

`String.nextWhile (s : String) (p : Char → Bool) (i : String.Pos) : String.Pos`

Repeatedly increments a position in a string, as if by `String.next`, while the predicate *p* returns `true` for the character at the position. Stops incrementing at the end of the string or when *p* returns `false` for the current character.

Examples:

- `let s := "  a  "; s.get (s.nextWhile Char.isWhitespace 0) = 'a'`
- `let s := "a  "; s.get (s.nextWhile Char.isWhitespace 0) = 'a'`
- `let s := "ba  "; s.get (s.nextWhile Char.isWhitespace 0) = 'b'`

def

```
String.nextUntil (s : String) (p : Char → Bool) (i : String.Pos) :
  String.Pos
```

Repeatedly increments a position in a string, as if by `String.next`, while the predicate *p* returns `false` for the character at the position. Stops incrementing at the end of the string or when *p* returns `true` for the current character.

Examples:

- `let s := "  a  "; s.get (s.nextUntil Char.isWhitespace 0) = ' '`
- `let s := "  a  "; s.get (s.nextUntil Char.isLetter 0) = 'a'`
- `let s := "a  "; s.get (s.nextUntil Char.isWhitespace 0) = ' '`

def

```
String.prev : String → String.Pos → String.Pos
```

Returns the position in a string before a specified position, *p*. If *p* = `⟨0⟩`, returns `0`. If *p* is not a valid position, the result is unspecified.

For example, `"LƎ∀N".prev ⟨3⟩` is unspecified, since byte 3 occurs in the middle of the multi-byte character `'Ǝ'`.

Examples:

- `"abc".get ("abc".endPos |> "abc".prev) = 'c'`
- `"LƎ∀N".get ("LƎ∀N".endPos |> "LƎ∀N".prev |> "LƎ∀N".prev |> "LƎ∀N".prev) = 'Ǝ'`

def

```
String.Pos.min (p₁ p₂ : String.Pos) : String.Pos
```

Returns either p_1 or p_2, whichever has the least byte index.

19.8.4.5. Lookups and Modifications

def

```
String.get (s : String) (p : String.Pos) : Char
```

Returns the character at position *p* of a string. If *p* is not a valid position, returns the fallback value (`default : Char`), which is `'A'`, but does not panic.

This function is overridden with an efficient implementation in runtime code. See `String.utf8GetAux` for the reference implementation.

Examples:

- `"abc".get ⟨1⟩ = 'b'`
- `"abc".get ⟨3⟩ = (default : Char)` because byte 3 is at the end of the string.
- `"LƎ∀N".get ⟨2⟩ = (default : Char)` because byte 2 is in the middle of `'Ǝ'`.

def

```
String.get? : String → String.Pos → Option Char
```

Returns the character at position p of a string. If p is not a valid position, returns `none`.

This function is overridden with an efficient implementation in runtime code. See `String.utf8GetAux?` for the reference implementation.

Examples:

- `"abc".get? ⟨1⟩ = some 'b'`
- `"abc".get? ⟨3⟩ = none`
- `"LƎ∀N".get? ⟨1⟩ = some 'Ǝ'`
- `"LƎ∀N".get? ⟨2⟩ = none`

def

```
String.get! (s : String) (p : String.Pos) : Char
```

Returns the character at position *p* of a string. Panics if *p* is not a valid position.

See `String.get?` for a safer alternative.

This function is overridden with an efficient implementation in runtime code. See `String.utf8GetAux` for the reference implementation.

Examples

- `"abc".get! ⟨1⟩ = 'b'`

def

`String.get' (s : String) (p : String.Pos) (h : ¬s.atEnd p = true) : Char`

Returns the character at position *p* of a string. Returns `(default : Char)`, which is `'A'`, if *p* is not a valid position.

Requires evidence, *h*, that *p* is within bounds instead of performing a run-time bounds check as in `String.get`.

A typical pattern combines `get'` with a dependent **`if`**-expression to avoid the overhead of an additional bounds check. For example:

```
def getInBounds? (s : String) (p : String.Pos) : Option Char :=
  if h : s.atEnd p then none else some (s.get' p h)
```

Even with evidence of `¬ s.atEnd p`, *p* may be invalid if a byte index points into the middle of a multi-byte UTF-8 character. For example, `"LƎ∀N".get' ⟨2⟩ (by decide) = (default : Char)`.

Examples:

- `"abc".get' 0 (by decide) = 'a'`
- `let lean := "LƎ∀N"; lean.get' (0 |> lean.next |> lean.next) (by decide) = '∀'`

def

`String.extract : String → String.Pos → String.Pos → String`

Creates a new string that consists of the region of the input string delimited by the two positions.

The result is `""` if the start position is greater than or equal to the end position or if the start position is at the end of the string. If either position is invalid (that is, if either points at the middle of a multi-byte UTF-8 character) then the result is unspecified.

Examples:

- `"red green blue".extract ⟨0⟩ ⟨3⟩ = "red"`
- `"red green blue".extract ⟨3⟩ ⟨0⟩ = ""`
- `"red green blue".extract ⟨0⟩ ⟨100⟩ = "red green blue"`
- `"red green blue".extract ⟨4⟩ ⟨100⟩ = "green blue"`

- `"LƎ∀N".extract ⟨2⟩ ⟨100⟩ = "green blue"`

def

`String.take (s : String) (n : Nat) : String`

Creates a new string that contains the first *n* characters (Unicode code points) of *s*.

If *n* is greater than *s*`.length`, returns *s*.

Examples:

- `"red green blue".take 3 = "red"`
- `"red green blue".take 1 = "r"`
- `"red green blue".take 0 = ""`
- `"red green blue".take 100 = "red green blue"`

def

`String.takeWhile (s : String) (p : Char → Bool) : String`

Creates a new string that contains the longest prefix of *s* in which *p* returns `true` for all characters.

Examples:

- `"red green blue".takeWhile (·.isLetter) = "red"`
- `"red green blue".takeWhile (· == 'r') = "r"`
- `"red green blue".takeWhile (· != 'n') = "red gree"`
- `"red green blue".takeWhile (fun _ => true) = "red green blue"`

def

`String.takeRight (s : String) (n : Nat) : String`

Creates a new string that contains the last *n* characters (Unicode code points) of *s*.

If *n* is greater than *s*`.length`, returns *s*.

Examples:

- `"red green blue".takeRight 4 = "blue"`
- `"red green blue".takeRight 1 = "e"`
- `"red green blue".takeRight 0 = ""`

- `"red green blue".takeRight 100 = "red green blue"`

def

`String.takeRightWhile (s : String) (p : Char → Bool) : String`

Creates a new string that contains the longest suffix of *s* in which *p* returns `true` for all characters.

Examples:

- `"red green blue".takeRightWhile (·.isLetter) = "blue"`
- `"red green blue".takeRightWhile (· == 'e') = "e"`
- `"red green blue".takeRightWhile (· != 'n') = " blue"`
- `"red green blue".takeRightWhile (fun _ => true) = "red green blue"`

def

`String.drop (s : String) (n : Nat) : String`

Removes the specified number of characters (Unicode code points) from the start of the string.

If *n* is greater than *s*`.length`, returns `""`.

Examples:

- `"red green blue".drop 4 = "green blue"`
- `"red green blue".drop 10 = "blue"`
- `"red green blue".drop 50 = ""`

def

`String.dropWhile (s : String) (p : Char → Bool) : String`

Creates a new string by removing the longest prefix from *s* in which *p* returns `true` for all characters.

Examples:

- `"red green blue".dropWhile (·.isLetter) = " green blue"`
- `"red green blue".dropWhile (· == 'r') = "ed green blue"`
- `"red green blue".dropWhile (· != 'n') = "n blue"`
- `"red green blue".dropWhile (fun _ => true) = ""`

def

```
String.dropRight (s : String) (n : Nat) : String
```

Removes the specified number of characters (Unicode code points) from the end of the string.

If *n* is greater than *s*`.length`, returns `""`.

Examples:

- `"red green blue".dropRight 5 = "red green"`
- `"red green blue".dropRight 11 = "red"`
- `"red green blue".dropRight 50 = ""`

def

```
String.dropRightWhile (s : String) (p : Char → Bool) : String
```

Creates a new string by removing the longest suffix from *s* in which *p* returns `true` for all characters.

Examples:

- `"red green blue".dropRightWhile (·.isLetter) = "red green "`
- `"red green blue".dropRightWhile (· == 'e') = "red green blu"`
- `"red green blue".dropRightWhile (· != 'n') = "red green"`
- `"red green blue".dropRightWhile (fun _ => true) = ""`

def

```
String.dropPrefix? (s pre : String) : Option Substring
```

If `pre` is a prefix of *s*, returns the remainder. Returns `none` otherwise.

The string `pre` is a prefix of *s* if there exists a `t : String` such that `s = pre ++ t`. If so, the result is `some t`.

Use `String.stripPrefix` to return the string unchanged when `pre` is not a prefix.

Examples:

- `"red green blue".dropPrefix? "red " = some "green blue"`
- `"red green blue".dropPrefix? "reed " = none`
- `"red green blue".dropPrefix? "" = some "red green blue"`

def

```
String.stripPrefix (s pre : String) : String
```

If `pre` is a prefix of `s`, returns the remainder. Returns `s` unmodified otherwise.

The string `pre` is a prefix of `s` if there exists a `t : String` such that `s = pre ++ t`. If so, the result is `t`. Otherwise, it is `s`.

Use `String.dropPrefix?` to return `none` when `pre` is not a prefix.

Examples:

- `"red green blue".stripPrefix "red " = "green blue"`
- `"red green blue".stripPrefix "reed " = "red green blue"`
- `"red green blue".stripPrefix "" = "red green blue"`

def

```
String.dropSuffix? (s suff : String) : Option Substring
```

If `suff` is a suffix of `s`, returns the remainder. Returns `none` otherwise.

The string `suff` is a suffix of `s` if there exists a `t : String` such that `s = t ++ suff`. If so, the result is `some t`.

Use `String.stripSuffix` to return the string unchanged when `suff` is not a suffix.

Examples:

- `"red green blue".dropSuffix? " blue" = some "red green"`
- `"red green blue".dropSuffix? " blu " = none`
- `"red green blue".dropSuffix? "" = some "red green blue"`

def

```
String.stripSuffix (s suff : String) : String
```

If `suff` is a suffix of `s`, returns the remainder. Returns `s` unmodified otherwise.

The string `suff` is a suffix of `s` if there exists a `t : String` such that `s = t ++ suff`. If so, the result is `t`. Otherwise, it is `s`.

Use `String.dropSuffix?` to return `none` when `suff` is not a suffix.

Examples:

- `"red green blue".stripSuffix " blue" = "red green"`
- `"red green blue".stripSuffix " blu " = "red green blue"`
- `"red green blue".stripSuffix "" = "red green blue"`

def

`String.trim (s : String) : String`

Removes leading and trailing whitespace from a string.

“Whitespace” is defined as characters for which `Char.isWhitespace` returns `true`.

Examples:

- `"abc".trim = "abc"`
- `"   abc".trim = "abc"`
- `"abc \t  ".trim = "abc"`
- `"  abc   ".trim = "abc"`
- `"abc\ndef\n".trim = "abc\ndef"`

def

`String.trimLeft (s : String) : String`

Removes leading whitespace from a string.

“Whitespace” is defined as characters for which `Char.isWhitespace` returns `true`.

Examples:

- `"abc".trimLeft = "abc"`
- `"   abc".trimLeft = "   abc"`
- `"abc \t  ".trimLeft = "abc \t  "`
- `"  abc   ".trimLeft = "abc   "`
- `"abc\ndef\n".trimLeft = "abc\ndef\n"`

def

`String.trimRight (s : String) : String`

Removes trailing whitespace from a string.

“Whitespace” is defined as characters for which `Char.isWhitespace` returns `true`.

Examples:

- `"abc".trimRight = "abc"`
- `"   abc".trimRight = "   abc"`
- `"abc \t  ".trimRight = "abc"`
- `"  abc   ".trimRight = "  abc"`

- `"abc\ndef\n".trimRight = "abc\ndef"`

def

`String.removeLeadingSpaces (s : String) : String`

Consistently de-indents the lines in a string, removing the same amount of leading whitespace from each line such that the least-indented line has no leading whitespace.

The number of leading whitespace characters to remove from each line is determined by counting the number of leading space (`' '`) and tab (`'\t'`) characters on lines after the first line that also contain non-whitespace characters. No distinction is made between tab and space characters; both count equally.

The least number of leading whitespace characters found is then removed from the beginning of each line. The first line's leading whitespace is not counted when determining how far to de-indent the string, but leading whitespace is removed from it.

Examples:

- `"Here:\n  fun x =>\n    x + 1".removeLeadingSpaces = "Here:\nfun x =>\n  x + 1"`
- `"Here:\n\t\tfun x =>\n\t  \tx + 1".removeLeadingSpaces = "Here:\nfun x =>\n \tx + 1"`
- `"Here:\n\t\tfun x =>\n \n\t  \tx + 1".removeLeadingSpaces = "Here:\nfun x =>\n\n \tx + 1"`

def

`String.set : String → String.Pos → Char → String`

Replaces the character at a specified position in a string with a new character. If the position is invalid, the string is returned unchanged.

If both the replacement character and the replaced character are 7-bit ASCII characters and the string is not shared, then it is updated in-place and not copied.

Examples:

- `"abc".set ⟨1⟩ 'B' = "aBc"`
- `"abc".set ⟨3⟩ 'D' = "abc"`
- `"LƎ∀N".set ⟨4⟩ 'X' = "LƎXN"`
- `"LƎ∀N".set ⟨2⟩ 'X' = "LƎ∀N"` because `'Ǝ'` is a multi-byte character, so the byte index 2 is an invalid position.

def

```
String.modify (s : String) (i : String.Pos) (f : Char → Char) : String
```

Replaces the character at position p in the string `s` with the result of applying `f` to that character. If p is an invalid position, the string is returned unchanged.

If both the replacement character and the replaced character are 7-bit ASCII characters and the string is not shared, then it is updated in-place and not copied.

Examples:

- `abc.modify ⟨1⟩ Char.toUpper = "aBc"`
- `abc.modify ⟨3⟩ Char.toUpper = "abc"`

def

```
String.front (s : String) : Char
```

Returns the first character in `s`. If `s = ""`, returns `(default : Char)`.

Examples:

- `"abc".front = 'a'`
- `"".front = (default : Char)`

def

```
String.back (s : String) : Char
```

Returns the last character in `s`. If `s = ""`, returns `(default : Char)`.

Examples:

- `"abc".back = 'c'`
- `"".back = (default : Char)`

def

```
String.posOf (s : String) (c : Char) : String.Pos
```

Returns the position of the first occurrence of a character, `c`, in a string `s`. If `s` does not contain `c`, returns `s.endPos`.

Examples:

- `"abcba".posOf 'a' = ⟨0⟩`
- `"abcba".posOf 'z' = ⟨5⟩`

- `"LƎ∀N".posOf '∀' = ⟨4⟩`

def

```
String.revPosOf (s : String) (c : Char) : Option String.Pos
```

Returns the position of the last occurrence of a character, *c*, in a string *s*. If *s* does not contain *c*, returns none.

Examples:

- `"abcabc".refPosOf 'a' = some ⟨3⟩`
- `"abcabc".revPosOf 'z' = none`
- `"LƎ∀N".revPosOf '∀' = some ⟨4⟩`

def

```
String.contains (s : String) (c : Char) : Bool
```

Checks whether a string contains the specified character.

Examples:

- `"green".contains 'e' = true`
- `"green".contains 'x' = false`
- `"".contains 'x' = false`

def

```
String.offsetOfPos (s : String) (pos : String.Pos) : Nat
```

Returns the character index that corresponds to the provided position (i.e. UTF-8 byte index) in a string.

If the position is at the end of the string, then the string's length in characters is returned. If the position is invalid due to pointing at the middle of a UTF-8 byte sequence, then the character index of the next character after the position is returned.

Examples:

- `"LƎ∀N".offsetOfPos ⟨0⟩ = 0`
- `"LƎ∀N".offsetOfPos ⟨1⟩ = 1`
- `"LƎ∀N".offsetOfPos ⟨2⟩ = 2`
- `"LƎ∀N".offsetOfPos ⟨4⟩ = 2`
- `"LƎ∀N".offsetOfPos ⟨5⟩ = 3`

- `"LƎ∀N".offsetOfPos ⟨50⟩ = 4`

def

`String.replace (s pattern replacement : String) : String`

In the string `s`, replaces all occurrences of *`pattern`* with *`replacement`*.

Examples:

- `"red green blue".replace "e" "" = "rd grn blu"`
- `"red green blue".replace "ee" "E" = "red grEn blue"`
- `"red green blue".replace "e" "E" = "rEd grEEn bluE"`

def

`String.findLineStart (s : String) (pos : String.Pos) : String.Pos`

Returns the position of the beginning of the line that contains the position *`pos`*.

Lines are ended by `'\n'`, and the returned position is either `0 : String.Pos` or immediately after a `'\n'` character.

def

`String.find (s : String) (p : Char → Bool) : String.Pos`

Finds the position of the first character in a string for which the Boolean predicate *p* returns `true`. If there is no such character in the string, then the end position of the string is returned.

Examples:

- `"coffee tea water".find (·.isWhitespace) = ⟨6⟩`
- `"tea".find (· == 'X') = ⟨3⟩`
- `"".find (· == 'X') = ⟨0⟩`

def

`String.revFind (s : String) (p : Char → Bool) : Option String.Pos`

Finds the position of the last character in a string for which the Boolean predicate *p* returns `true`. If there is no such character in the string, then `none` is returned.

Examples:

- `"coffee tea water".revFind (·.isWhitespace) = some ⟨10⟩`
- `"tea".revFind (· == 'X') = none`

- `"".revFind (· == 'X') = none`

19.8.4.6. Folds and Aggregation

def

```
String.map (f : Char → Char) (s : String) : String
```

Applies the function *f* to every character in a string, returning a string that contains the resulting characters.

Examples:

- `"abc123".map Char.toUpper = "ABC123"`
- `"".map Char.toUpper = ""`

def

```
String.foldl.{u} {α : Type u} (f : α → Char → α) (init : α)
  (s : String) : α
```

Folds a function over a string from the left, accumulating a value starting with *init*. The accumulated value is combined with each character in order, using *f*.

Examples:

- `"coffee tea water".foldl (fun n c => if c.isWhitespace then n + 1 else n) 0 = 2`
- `"coffee tea and water".foldl (fun n c => if c.isWhitespace then n + 1 else n) 0 = 3`
- `"coffee tea water".foldl (·.push ·) "" = "coffee tea water"`

def

```
String.foldr.{u} {α : Type u} (f : Char → α → α) (init : α)
  (s : String) : α
```

Folds a function over a string from the right, accumulating a value starting with *init*. The accumulated value is combined with each character in reverse order, using *f*.

Examples:

- `"coffee tea water".foldr (fun c n => if c.isWhitespace then n + 1 else n) 0 = 2`
- `"coffee tea and water".foldr (fun c n => if c.isWhitespace then n + 1 else n) 0 = 3`

- `"coffee tea water".foldr (fun c s => c.push s) "" = "retaw dna aet eeffoc"`

def

`String.all (s : String) (p : Char → Bool) : Bool`

Checks whether the Boolean predicate *p* returns `true` for every character in a string.

Short-circuits at the first character for which *p* returns `false`.

Examples:

- `"brown".all (·.isLetter) = true`
- `"brown and orange".all (·.isLetter) = false`
- `"".all (fun _ => false) = true`

def

`String.any (s : String) (p : Char → Bool) : Bool`

Checks whether there is a character in a string for which the Boolean predicate *p* returns `true`.

Short-circuits at the first character for which *p* returns `true`.

Examples:

- `"brown".any (·.isLetter) = true`
- `"brown".any (·.isWhitespace) = false`
- `"brown and orange".any (·.isLetter) = true`
- `"".any (fun _ => false) = false`

19.8.4.7. Comparisons

The `LT String` instance is defined by the lexicographic ordering on strings based on the `LT Char` instance. Logically, this is modeled by the lexicographic ordering on the lists that model strings, so `List.Lex` defines the order. It is decidable, and the decision procedure is overridden at runtime with efficient code that takes advantage of the run-time representation of strings.

def

`String.le (a b : String) : Prop`

Non-strict inequality on strings, typically used via the ≤ operator.

a ≤ *b* is defined to mean ¬ *b* < *a*.

def

```
String.firstDiffPos (a b : String) : String.Pos
```

Returns the first position where the two strings differ.

If one string is a prefix of the other, then the returned position is the end position of the shorter string. If the strings are identical, then their end position is returned.

Examples:

- `"tea".firstDiffPos "ten" = ⟨2⟩`
- `"tea".firstDiffPos "tea" = ⟨3⟩`
- `"tea".firstDiffPos "teas" = ⟨3⟩`
- `"teas".firstDiffPos "tea" = ⟨3⟩`

def

```
String.substrEq (s1 : String) (pos1 : String.Pos) (s2 : String)
  (pos2 : String.Pos) (sz : Nat) : Bool
```

Checks whether substrings of two strings are equal. Substrings are indicated by their starting positions and a size in *UTF-8 bytes*. Returns `false` if the indicated substring does not exist in either string.

def

```
String.isPrefixOf (p s : String) : Bool
```

Checks whether the first string (`p`) is a prefix of the second (`s`).

`String.startsWith` is a version that takes the potential prefix after the string.

Examples:

- `"red".isPrefixOf "red green blue" = true`
- `"green".isPrefixOf "red green blue" = false`
- `"".isPrefixOf "red green blue" = true`

def

```
String.startsWith (s pre : String) : Bool
```

Checks whether the first string (`s`) begins with the second (`pre`).

`String.isPrefix` is a version that takes the potential prefix before the string.

Examples:

- `"red green blue".startsWith "red" = true`
- `"red green blue".startsWith "green" = false`
- `"red green blue".startsWith "" = true`
- `"red".startsWith "red" = true`

def

`String.endsWith (s post : String) : Bool`

Checks whether the first string (`s`) ends with the second (`post`).

Examples:

- `"red green blue".endsWith "blue" = true`
- `"red green blue".endsWith "green" = false`
- `"red green blue".endsWith "" = true`
- `"red".endsWith "red" = true`

def

`String.decEq (s₁ s₂ : String) : Decidable (s₁ = s₂)`

Decides whether two strings are equal. Normally used via the `DecidableEq String` instance and the = operator.

At runtime, this function is overridden with an efficient native implementation.

opaque

`String.hash (s : String) : UInt64`

Computes a hash for strings.

19.8.4.8. Manipulation

def

`String.split (s : String) (p : Char → Bool) : List String`

Splits a string at each character for which *p* returns `true`.

The characters that satisfy *p* are not included in any of the resulting strings. If multiple characters in a row satisfy *p*, then the resulting list will contain empty strings.

Examples:

- `"coffee tea water".split (·.isWhitespace) = ["coffee", "tea", "water"]`
- `"coffee  tea  water".split (·.isWhitespace) = ["coffee", "", "tea", "", "water"]`
- `"fun x =>\n  x + 1\n".split (· == '\n') = ["fun x =>", "  x + 1", ""]`

def

`String.splitOn (s : String) (sep : String := " ") : List String`

Splits a string `s` on occurrences of the separator string `sep`. The default separator is `" "`.

When `sep` is empty, the result is `[s]`. When `sep` occurs in overlapping patterns, the first match is taken. There will always be exactly `n+1` elements in the returned list if there were `n` non-overlapping matches of `sep` in the string. The separators are not included in the returned substrings.

Examples:

- `"here is some text ".splitOn = ["here", "is", "some", "text", ""]`
- `"here is some text ".splitOn "some" = ["here is ", " text "]`
- `"here is some text ".splitOn "" = ["here is some text "]`
- `"abababacabac".splitOn "aba" = ["", "bac", "c"]`

def

`String.push : String → Char → String`

Adds a character to the end of a string.

The internal implementation uses dynamic arrays and will perform destructive updates if the string is not shared.

Examples:

- `"abc".push 'd' = "abcd"`
- `"".push 'a' = "a"`

def

`String.pushn (s : String) (c : Char) (n : Nat) : String`

Adds multiple repetitions of a character to the end of a string.

Returns *s*, with *n* repetitions of *c* at the end. Internally, the implementation repeatedly calls `String.push`, so the string is modified in-place if there is a unique reference to it.

Examples:

- `"indeed".pushn '!' 2 = "indeed!!"`
- `"indeed".pushn '!' 0 = "indeed"`
- `"".pushn ' ' 4 = "    "`

def

```
String.capitalize (s : String) : String
```

Replaces the first character in *s* with the result of applying `Char.toUpper` to it. Returns the empty string if the string is empty.

`Char.toUpper` has no effect on characters outside of the range `'a'–'z'`.

Examples:

- `"orange".capitalize = "Orange"`
- `"ORANGE".capitalize = "ORANGE"`
- `"".capitalize = ""`

def

```
String.decapitalize (s : String) : String
```

Replaces the first character in *s* with the result of applying `Char.toLower` to it. Returns the empty string if the string is empty.

`Char.toLower` has no effect on characters outside of the range `'A'–'Z'`.

Examples:

- `"Orange".decapitalize = "orange"`
- `"ORANGE".decapitalize = "oRANGE"`
- `"".decapitalize = ""`

def

```
String.toUpper (s : String) : String
```

Replaces each character in *s* with the result of applying `Char.toUpper` to it.

`Char.toUpper` has no effect on characters outside of the range `'a'–'z'`.

Examples:

- `"orange".toUpper = "ORANGE"`
- `"abc123".toUpper = "ABC123"`

def

`String.toLower (s : String) : String`

Replaces each character in `s` with the result of applying `Char.toLower` to it.

`Char.toLower` has no effect on characters outside of the range `'A'–'Z'`.

Examples:

- `"ORANGE".toLower = "orange"`
- `"Orange".toLower = "orange"`
- `"ABc123".toLower = "abc123"`

19.8.4.9. Iterators

Fundamentally, a `String.Iterator` is a pair of a string and a valid position in the string. Iterators provide functions for getting the current character (`curr`), replacing the current character (`setCurr`), checking whether the iterator can move to the left or the right (`hasPrev` and `hasNext`, respectively), and moving the iterator (`prev` and `next`, respectively). Clients are responsible for checking whether they've reached the beginning or end of the string; otherwise, the iterator ensures that its position always points at a character.

structure

`String.Iterator : Type`

An iterator over the characters (Unicode code points) in a `String`. Typically created by `String.iter`.

String iterators pair a string with a valid byte index. This allows efficient character-by-character processing of strings while avoiding the need to manually ensure that byte indices are used with the correct strings.

An iterator is *valid* if the position `i` is *valid* for the string `s`, meaning `0 ≤ i ≤ s.endPos` and `i` lies on a UTF8 byte boundary. If `i = s.endPos`, the iterator is at the end of the string.

Most operations on iterators return unspecified values if the iterator is not valid. The functions in the `String.Iterator` API rule out the creation of invalid iterators, with two exceptions:

- `Iterator.next iter` is invalid if `iter` is already at the end of the string (`iter.atEnd` is `true`), and

- `Iterator.forward iter n`/`Iterator.nextn iter n` is invalid if `n` is strictly greater than the number of remaining characters.

Constructor

`String.Iterator.mk`

Fields

`s : String`

The string being iterated over.

`i : String.Pos`

The current UTF-8 byte position in the string `s`.

This position is not guaranteed to be valid for the string. If the position is not valid, then the current character is `(default : Char)`, similar to `String.get` on an invalid position.

def

`String.iter (s : String) : String.Iterator`

Creates an iterator at the beginning of the string.

def

`String.mkIterator (s : String) : String.Iterator`

Creates an iterator at the beginning of the string.

def

`String.Iterator.curr : String.Iterator → Char`

Gets the character at the iterator's current position.

A run-time bounds check is performed. Use `String.Iterator.curr'` to avoid redundant bounds checks.

If the position is invalid, returns `(default : Char)`.

def

`String.Iterator.hasNext : String.Iterator → Bool`

Checks whether the iterator is at or before the string's last character.

def

`String.Iterator.next : String.Iterator → String.Iterator`

Moves the iterator's position forward by one character, unconditionally.

It is only valid to call this function if the iterator is not at the end of the string (i.e. if `Iterator.atEnd` is `false`); otherwise, the resulting iterator will be invalid.

def

`String.Iterator.forward : String.Iterator → Nat → String.Iterator`

Moves the iterator's position forward by the specified number of characters.

The resulting iterator is only valid if the number of characters to skip is less than or equal to the number of characters left in the iterator.

def

`String.Iterator.nextn : String.Iterator → Nat → String.Iterator`

Moves the iterator's position forward by the specified number of characters.

The resulting iterator is only valid if the number of characters to skip is less than or equal to the number of characters left in the iterator.

def

`String.Iterator.hasPrev : String.Iterator → Bool`

Checks whether the iterator is after the beginning of the string.

def

`String.Iterator.prev : String.Iterator → String.Iterator`

Moves the iterator's position backward by one character, unconditionally.

The position is not changed if the iterator is at the beginning of the string.

def

`String.Iterator.prevn : String.Iterator → Nat → String.Iterator`

Moves the iterator's position back by the specified number of characters, stopping at the beginning of the string.

def

```
String.Iterator.atEnd : String.Iterator → Bool
```

Checks whether the iterator is past its string's last character.

def

```
String.Iterator.toEnd : String.Iterator → String.Iterator
```

Moves the iterator's position to the end of the string, just past the last character.

def

```
String.Iterator.setCurr : String.Iterator → Char → String.Iterator
```

Replaces the current character in the string.

Does nothing if the iterator is at the end of the string. If both the replacement character and the replaced character are 7-bit ASCII characters and the string is not shared, then it is updated in-place and not copied.

def

```
String.Iterator.extract : String.Iterator → String.Iterator → String
```

Extracts the substring between the positions of two iterators. The first iterator's position is the start of the substring, and the second iterator's position is the end.

Returns the empty string if the iterators are for different strings, or if the position of the first iterator is past the position of the second iterator.

def

```
String.Iterator.remainingToString : String.Iterator → String
```

The remaining characters in an iterator, as a string.

def

```
String.Iterator.remainingBytes : String.Iterator → Nat
```

The number of UTF-8 bytes remaining in the iterator.

def

```
String.Iterator.pos (self : String.Iterator) : String.Pos
```

The current UTF-8 byte position in the string `s`.

This position is not guaranteed to be valid for the string. If the position is not valid, then the current character is `(default : Char)`, similar to `String.get` on an invalid position.

19.8.4.10. Substrings

def

`String.toSubstring (s : String) : Substring`

Converts a `String` into a `Substring` that denotes the entire string.

def

`String.toSubstring' (s : String) : Substring`

Converts a `String` into a `Substring` that denotes the entire string.

This is a version of `String.toSubstring` that doesn't have an `@[inline]` annotation.

structure

`Substring : Type`

A region or slice of some underlying string.

A substring contains an string together with the start and end byte positions of a region of interest. Actually extracting a substring requires copying and memory allocation, while many substrings of the same underlying string may exist with very little overhead, and they are more convenient than tracking the bounds by hand.

Using its constructor explicitly, it is possible to construct a `Substring` in which one or both of the positions is invalid for the string. Many operations will return unexpected or confusing results if the start and stop positions are not valid. Instead, it's better to use API functions that ensure the validity of the positions in a substring to create and manipulate them.

Constructor

`Substring.mk`

Fields

`str : String`

The underlying string.

`startPos : String.Pos`

The byte position of the start of the string slice.

`stopPos : String.Pos`

The byte position of the end of the string slice.

19.8.4.10.1. Properties

def

```
Substring.isEmpty (ss : Substring) : Bool
```

Checks whether a substring is empty.

A substring is empty if its start and end positions are the same.

def

```
Substring.bsize : Substring → Nat
```

The number of bytes used by the string's UTF-8 encoding.

19.8.4.10.2. Positions

def

```
Substring.atEnd : Substring → String.Pos → Bool
```

Checks whether a position in a substring is precisely equal to its ending position.

The position is understood relative to the substring's starting position, rather than the underlying string's starting position.

def

```
Substring.posOf (s : Substring) (c : Char) : String.Pos
```

Returns the substring-relative position of the first occurrence of *c* in *s*, or *s* `.bsize` if *c* doesn't occur.

def

```
Substring.next : Substring → String.Pos → String.Pos
```

Returns the next position in a substring after the given position. If the position is at the end of the substring, it is returned unmodified.

Both the input position and the returned position are interpreted relative to the substring's start position, not the underlying string.

def

```
Substring.nextn : Substring → Nat → String.Pos → String.Pos
```

Returns the position that's the specified number of characters forward from the given position in a substring. If the end position of the substring is reached, it is returned.

Both the input position and the returned position are interpreted relative to the substring's start position, not the underlying string.

def

```
Substring.prev : Substring → String.Pos → String.Pos
```

Returns the previous position in a substring, just prior to the given position. If the position is at the beginning of the substring, it is returned unmodified.

Both the input position and the returned position are interpreted relative to the substring's start position, not the underlying string.

def

```
Substring.prevn : Substring → Nat → String.Pos → String.Pos
```

Returns the position that's the specified number of characters prior to the given position in a substring. If the start position of the substring is reached, it is returned.

Both the input position and the returned position are interpreted relative to the substring's start position, not the underlying string.

19.8.4.10.3. Folds and Aggregation

def

```
Substring.foldl.{u} {α : Type u} (f : α → Char → α) (init : α)
  (s : Substring) : α
```

Folds a function over a substring from the left, accumulating a value starting with `init`. The accumulated value is combined with each character in order, using `f`.

def

```
Substring.foldr.{u} {α : Type u} (f : Char → α → α) (init : α)
  (s : Substring) : α
```

Folds a function over a substring from the right, accumulating a value starting with `init`. The accumulated value is combined with each character in reverse order, using `f`.

def

```
Substring.all (s : Substring) (p : Char → Bool) : Bool
```

Checks whether the Boolean predicate *p* returns `true` for every character in a substring.

Short-circuits at the first character for which *p* returns `false`.

def

```
Substring.any (s : Substring) (p : Char → Bool) : Bool
```

Checks whether the Boolean predicate *p* returns `true` for any character in a substring.

Short-circuits at the first character for which *p* returns `true`.

19.8.4.10.4. Comparisons

def

```
Substring.beq (ss1 ss2 : Substring) : Bool
```

Checks whether two substrings represent equal strings. Usually accessed via the == operator.

Two substrings do not need to have the same underlying string or the same start and end positions; instead, they are equal if they contain the same sequence of characters.

def

```
Substring.sameAs (ss1 ss2 : Substring) : Bool
```

Checks whether two substrings have the same position and content.

The two substrings do not need to have the same underlying string for this check to succeed.

19.8.4.10.5. Prefix and Suffix

def

```
Substring.commonPrefix (s t : Substring) : Substring
```

Returns the longest common prefix of two substrings.

The returned substring uses the same underlying string as *s*.

def

```
Substring.commonSuffix (s t : Substring) : Substring
```

Returns the longest common suffix of two substrings.

The returned substring uses the same underlying string as *s*.

def

```
Substring.dropPrefix? (s pre : Substring) : Option Substring
```

If `pre` is a prefix of *s*, returns the remainder. Returns `none` otherwise.

The substring `pre` is a prefix of *s* if there exists a `t : Substring` such that *s*`.toString = `*pre*`.toString ++ `*t*`.toString`. If so, the result is the substring of *s* without the prefix.

def

```
Substring.dropSuffix? (s suff : Substring) : Option Substring
```

If *suff* is a suffix of *s*, returns the remainder. Returns `none` otherwise.

The substring *suff* is a suffix of *s* if there exists a `t : Substring` such that *s*`.toString = `*t*`.toString ++ `*suff*`.toString`. If so, the result the substring of *s* without the suffix.

19.8.4.10.6. Lookups

def

```
Substring.get : Substring → String.Pos → Char
```

Returns the character at the given position in the substring.

The position is relative to the substring, rather than the underlying string, and no bounds checking is performed with respect to the substring's end position. If the relative position is not a valid position in the underlying string, the fallback value `(default : Char)`, which is `'A'`, is returned. Does not panic.

def

```
Substring.contains (s : Substring) (c : Char) : Bool
```

Checks whether a substring contains the specified character.

def

```
Substring.front (s : Substring) : Char
```

Returns the first character in the substring.

If the substring is empty, but the substring's start position is a valid position in the underlying string, then the character at the start position is returned. If the substring's start position is not a valid position in the string, the fallback value `(default : Char)`, which is `'A'`, is returned.

Does not panic.

19.8.4.10.7. Modifications

def

```
Substring.drop : Substring → Nat → Substring
```

Removes the specified number of characters (Unicode code points) from the beginning of a substring by advancing its start position.

If the substring's end position is reached, the start position is not advanced past it.

def

```
Substring.dropWhile : Substring → (Char → Bool) → Substring
```

Removes the longest prefix of a substring in which a Boolean predicate returns `true` for all characters by moving the substring's start position. The start position is moved to the position of the first character for which the predicate returns `false`, or to the substring's end position if the predicate always returns `true`.

def

```
Substring.dropRight : Substring → Nat → Substring
```

Removes the specified number of characters (Unicode code points) from the end of a substring by moving its end position towards its start position.

If the substring's start position is reached, the end position is not retracted past it.

def

```
Substring.dropRightWhile : Substring → (Char → Bool) → Substring
```

Removes the longest suffix of a substring in which a Boolean predicate returns `true` for all characters by moving the substring's end position. The end position is moved just after the position of the last character for which the predicate returns `false`, or to the substring's start position if the predicate always returns `true`.

def

```
Substring.take : Substring → Nat → Substring
```

Retains only the specified number of characters (Unicode code points) at the beginning of a substring, by moving its end position towards its start position.

If the substring's start position is reached, the end position is not retracted past it.

def

```
Substring.takeWhile : Substring → (Char → Bool) → Substring
```

Retains only the longest prefix of a substring in which a Boolean predicate returns `true` for all characters by moving the substring's end position towards its start position.

def

```
Substring.takeRight : Substring → Nat → Substring
```

Retains only the specified number of characters (Unicode code points) at the end of a substring, by moving its start position towards its end position.

If the substring's end position is reached, the start position is not advanced past it.

def

```
Substring.takeRightWhile : Substring → (Char → Bool) → Substring
```

Retains only the longest suffix of a substring in which a Boolean predicate returns `true` for all characters by moving the substring's start position towards its end position.

def

```
Substring.extract : Substring → String.Pos → String.Pos → Substring
```

Returns the region of the substring delimited by the provided start and stop positions, as a substring. The positions are interpreted with respect to the substring's start position, rather than the underlying string.

If the resulting substring is empty, then the resulting substring is a substring of the empty string `""`. Otherwise, the underlying string is that of the input substring with the beginning and end positions adjusted.

def

```
Substring.trim : Substring → Substring
```

Removes leading and trailing whitespace from a substring by first moving its start position to the first non-whitespace character, and then moving its end position to the last non-whitespace character.

If the substring consists only of whitespace, then the resulting substring's start position is moved to its end position.

"Whitespace" is defined as characters for which `Char.isWhitespace` returns `true`.

Examples:

- `" red green blue ".toSubstring.trim.toString = "red green blue"`
- `" red green blue ".toSubstring.trim.startPos = ⟨1⟩`
- `" red green blue ".toSubstring.trim.stopPos = ⟨15⟩`
- `"     ".toSubstring.trim.startPos = ⟨5⟩`

def

```
Substring.trimLeft (s : Substring) : Substring
```

Removes leading whitespace from a substring by moving its start position to the first non-whitespace character, or to its end position if there is no non-whitespace character.

"Whitespace" is defined as characters for which `Char.isWhitespace` returns `true`.

def

```
Substring.trimRight (s : Substring) : Substring
```

Removes trailing whitespace from a substring by moving its end position to the last non-whitespace character, or to its start position if there is no non-whitespace character.

"Whitespace" is defined as characters for which `Char.isWhitespace` returns `true`.

def

```
Substring.splitOn (s : Substring) (sep : String := " ") : List Substring
```

Splits a substring `s` on occurrences of the separator string `sep`. The default separator is `" "`.

When `sep` is empty, the result is `[s]`. When `sep` occurs in overlapping patterns, the first match is taken. There will always be exactly $n+1$ elements in the returned list if there were n non-overlapping matches of `sep` in the string. The separators are not included in the returned substrings, which are all substrings of `s`'s string.

19.8.4.10.8. Conversions

def

```
Substring.toString : Substring → String
```

Copies the region of the underlying string pointed to by a substring into a fresh string.

def

```
Substring.isNat (s : Substring) : Bool
```

Checks whether the substring can be interpreted as the decimal representation of a natural number.

A substring can be interpreted as a decimal natural number if it is not empty and all the characters in it are digits.

Use `Substring.toNat?` to convert such a substring to a natural number.

def

```
Substring.toNat? (s : Substring) : Option Nat
```

Checks whether the substring can be interpreted as the decimal representation of a natural number, returning the number if it can.

A substring can be interpreted as a decimal natural number if it is not empty and all the characters in it are digits.

Use `Substring.isNat` to check whether the substring is such a substring.

def

```
Substring.toIterator : Substring → String.Iterator
```

Returns an iterator into the underlying string, at the substring's starting position. The ending position is discarded, so the iterator alone cannot be used to determine whether its current position is within the original substring.

def

```
Substring.toName (s : Substring) : Lean.Name
```

Converts a substring to the Lean compiler's representation of names. The resulting name is hierarchical, and the string is split at the dots (`'.'`).

`"a.b".toSubstring.toName` is the name `a.b`, not «`a.b`». For the latter, use `Name.mkSimple ∘ Substring.toString`.

19.8.4.11. Metaprogramming

def

```
String.toName (s : String) : Lean.Name
```

Converts a string to the Lean compiler's representation of names. The resulting name is hierarchical, and the string is split at the dots (`'.'`).

`"a.b".toName` is the name `a.b`, not «`a.b`». For the latter, use `Name.mkSimple`.

def

```
String.quote (s : String) : String
```

Converts a string to its corresponding Lean string literal syntax. Double quotes are added to each end, and internal characters are escaped as needed.

Examples:

- `"abc".quote = "\"abc\""`
- `"\"".quote = "\"\\\"\""`

19.8.4.12. Encodings

def

```
String.getUtf8Byte (s : String) (n : Nat) (h : n < s.utf8ByteSize)
:
  UInt8
```

Accesses the indicated byte in the UTF-8 encoding of a string.

At runtime, this function is implemented by efficient, constant-time code.

def

```
String.utf8ByteSize : String → Nat
```

The number of bytes used by the string's UTF-8 encoding.

At runtime, this function takes constant time because the byte length of strings is cached.

def

```
String.utf8EncodeChar (c : Char) : List UInt8
```

Returns the sequence of bytes in a character's UTF-8 encoding.

def

```
String.utf8DecodeChar? (a : ByteArray) (i : Nat) : Option Char
```

Decodes the UTF-8 character sequence that starts at a given index in a byte array, or none if index `i` is out of bounds or is not the start of a valid UTF-8 character.

def

```
String.fromUTF8 (a : ByteArray) (h : String.validateUTF8 a = true)
:
  String
```

Decodes an array of bytes that encode a string as UTF-8 into the corresponding string. Invalid UTF-8 characters in the byte array result in `(default : Char)`, or `'A'`, in the string.

def

```
String.fromUTF8? (a : ByteArray) : Option String
```

Decodes an array of bytes that encode a string as UTF-8 into the corresponding string, or returns none if the array is not a valid UTF-8 encoding of a string.

def

```
String.fromUTF8! (a : ByteArray) : String
```

Decodes an array of bytes that encode a string as UTF-8 into the corresponding string, or panics if the array is not a valid UTF-8 encoding of a string.

def

```
String.toUTF8 (a : String) : ByteArray
```

Encodes a string in UTF-8 as an array of bytes.

def

```
String.validateUTF8 (a : ByteArray) : Bool
```

Checks whether an array of bytes is a valid UTF-8 encoding of a string.

def

```
String.crlfToLf (text : String) : String
```

Replaces each `\r\n` with `\n` to normalize line endings, but does not validate that there are no isolated `\r` characters.

This is an optimized version of `String.replace text "\r\n" "\n"`.

19.8.5. FFI

FFI type

```
typedef struct {
    lean_object m_header;
    /* byte length including '\0' terminator */
    size_t      m_size;
    size_t      m_capacity;
    /* UTF8 length */
    size_t      m_length;
    char        m_data[0];
} lean_string_object;
```

The representation of strings in C. See the description of run-time `String`s for more details.

FFI function

```
bool lean_is_string(lean_object * o)
```

Returns `true` if `o` is a string, or `false` otherwise.

FFI function

```
lean_string_object * lean_to_string(lean_object * o)
```

Performs a runtime check that `o` is indeed a string. If `o` is not a string, an assertion fails.

19.9. The Unit Type

The unit type is the canonical type with exactly one element, named `Unit` and represented by the empty tuple `()`. It describes only a single value, which consists of said constructor applied to no arguments whatsoever.

`Unit` is analogous to `void` in languages derived from C: even though `void` has no elements that can be named, it represents the return of control flow from a function with no additional information. In functional programming, `Unit` is the return type of things that "return nothing". Mathematically, this is represented by a single completely uninformative value, as opposed to an empty type such as `Empty`, which represents unreachable code.

When programming with monads, `Unit` is especially useful. For any type α, $m\ \alpha$ represents an action that has side effects and returns a value of type α. The type $m\ \texttt{Unit}$ represents an action that has some side effects but does not return a value.

There are two variants of the unit type:

- `Unit` is a `Type` that exists in the smallest non-propositional universe.
- `PUnit` is universe polymorphic and can be used in any non-propositional universe.

Behind the scenes, `Unit` is actually defined as `PUnit.{1}`. `Unit` should be preferred over `PUnit` when possible to avoid unnecessary universe parameters. If in doubt, use `Unit` until universe errors occur.

def

```
Unit : Type
```

The canonical type with one element. This element is written `()`.

`Unit` has a number of uses:

- It can be used to model control flow that returns from a function call without providing other information.
- Monadic actions that return `Unit` have side effects without computing values.
- In polymorphic types, it can be used to indicate that no data is to be stored in a particular field.

def

```
Unit.unit : Unit
```

The only element of the unit type.

It can be written as an empty tuple: `()`.

inductive type

```
PUnit.{u} : Sort u
```

The canonical universe-polymorphic type with just one element.

It should be used in contexts that require a type to be universe polymorphic, thus disallowing `Unit`.

Constructors

```
| unit.{u} : PUnit
```

The only element of the universe-polymorphic unit type.

19.9.1. Definitional Equality

Unit-like types are inductive types that have a single constructor which takes no non-proof parameters. `PUnit` is one such type. All elements of unit-like types are definitionally equal to all other elements.

▼ *Example: Definitional Equality of `Unit`*

Every term with type `Unit` is definitionally equal to every other term with type `Unit`:

```
example (e1 e2 : Unit) : e1 = e2 := rfl
```

▼ *Example: Definitional Equality of Unit-Like Types*

Both `CustomUnit` and `AlsoUnit` are unit-like types, with a single constructor that takes no parameters. Every pair of terms with either type is definitionally equal.

```
inductive CustomUnit where
  | customUnit

example (e1 e2 : CustomUnit) : e1 = e2 := rfl

structure AlsoUnit where

example (e1 e2 : AlsoUnit) : e1 = e2 := rfl
```

Types with parameters, such as `WithParam`, are also unit-like if they have a single constructor that does not take parameters.

```
inductive WithParam (n : Nat) where
  | mk

example (x y : WithParam 3) : x = y := rfl
```

Constructors with non-proof parameters are not unit-like, even if the parameters are all unit-like types.

```
inductive NotUnitLike where
  | mk (u : Unit)

example (e1 e2 : NotUnitLike) : e1 = e2 := rfl
```

```
type mismatch
  rfl
has type
```

```
  ?m.13 = ?m.13 : Prop
but is expected to have type
  e1 = e2 : Prop
```

Constructors of unit-like types may take parameters that are proofs.

```
inductive ProofUnitLike where
  | mk : 2 = 2 → ProofUnitLike

example (e1 e2 : ProofUnitLike) : e1 = e2 := rfl
```

19.10. The Empty Type

The empty type Empty represents impossible values. It is an inductive type with no constructors whatsoever.

While the trivial type Unit, which has a single constructor that takes no parameters, can be used to model computations where a result is unwanted or uninteresting, Empty can be used in situations where no computation should be possible at all. Instantiating a polymorphic type with Empty can mark some of its constructors—those with a parameter of the corresponding type—as impossible; this can rule out certain code paths that are not desired.

The presence of a term with type Empty indicates that an impossible code path has been reached. There will never be a value with this type, due to the lack of constructors. On an impossible code path, there's no reason to write further code; the function Empty.elim can be used to escape an impossible path.

The universe-polymorphic equivalent of Empty is PEmpty.

inductive type

```
Empty : Type
```

The empty type. It has no constructors.

Use Empty.elim in contexts where a value of type Empty is in scope.

Constructors

inductive type

```
PEmpty.{u} : Sort u
```

The universe-polymorphic empty type, with no constructors.

PEmpty can be used in any universe, but this flexibility can lead to worse error messages and more challenges with universe level unification. Prefer the type Empty or the proposition False when possible.

Constructors

▼ *Example: Impossible Code Paths*

The type signature of the function `f` indicates that it might throw exceptions, but allows the exception type to be anything:

```
def f (n : Nat) : Except ε Nat := pure n
```

Instantiating `f`'s exception type with `Empty` exploits the fact that `f` never actually throws an exception to convert it to a function whose type indicates that no exceptions will be thrown. In particular, it allows `Empty.elim` to be used to avoid handing the impossible exception value.

```
def g (n : Nat) : Nat :=
  match f (ε := Empty) n with
  | .error e =>
    Empty.elim e
  | .ok v => v
```

19.10.1. API Reference

def

`Empty.elim.{u} {C : Sort u} : Empty → C`

`Empty.elim : Empty → C` says that a value of any type can be constructed from `Empty`. This can be thought of as a compiler-checked assertion that a code path is unreachable.

def

`PEmpty.elim.{u_1, u_2} {C : Sort u_1} : PEmpty → C`

`PEmpty.elim : Empty → C` says that a value of any type can be constructed from `PEmpty`. This can be thought of as a compiler-checked assertion that a code path is unreachable.

19.11. Booleans

inductive type

`Bool : Type`

The Boolean values, `true` and `false`.

Logically speaking, this is equivalent to `Prop` (the type of propositions). The distinction is public important for programming: both propositions and their proofs are erased in the code generator, while `Bool` corresponds to the Boolean type in most programming languages and carries precisely one bit of run-time information.

Constructors

| `false : Bool`

The Boolean value `false`, not to be confused with the proposition `False`.

| `true : Bool`

The Boolean value `true`, not to be confused with the proposition `True`.

The constructors `Bool.true` and `Bool.false` are exported from the `Bool` namespace, so they can be written `true` and `false`.

19.11.1. Run-Time Representation

Because `Bool` is an enum inductive type, it is represented by a single byte in compiled code.

19.11.2. Booleans and Propositions

Both `Bool` and `Prop` represent notions of truth. From a purely logical perspective, they are equivalent: propositional extensionality means that there are fundamentally only two propositions, namely `True` and `False`. However, there is an important pragmatic difference: `Bool` classifies *values* that can be computed by programs, while `Prop` classifies statements for which code generation doesn't make sense. In other words, `Bool` is the notion of truth and falsehood that's appropriate for programs, while `Prop` is the notion that's appropriate for mathematics. Because proofs are erased from compiled programs, keeping `Bool` and `Prop` distinct makes it clear which parts of a Lean file are intended for computation.

A `Bool` can be used wherever a `Prop` is expected. There is a coercion from every `Bool` b to the proposition `b = true`. By `propext`, `true = true` is equal to `True`, and `false = true` is equal to `False`.

Not every proposition can be used by programs to make run-time decisions. Otherwise, a program could branch on whether the Collatz conjecture is true or false! Many propositions, however, can be checked algorithmically. These propositions are called *decidable* propositions, and have instances of the `Decidable` type class. The function `Decidable.decide` converts a proof-carrying `Decidable` result into a `Bool`. This function is also a coercion from decidable propositions to `Bool`, so `(2 = 2 : Bool)` evaluates to `true`.

19.11.3. Syntax

syntax

Boolean Infix Operators

The infix operators &&, ||, and ^^ are notations for Bool.and, Bool.or, and Bool.xor, respectively.

```
term ::= ...
    | term && term

term ::= ...
    | term || term

term ::= ...
    | term ^^ term
```

syntax

Boolean Negation

The prefix operator ! is notation for Bool.not.

```
term ::= ...
    | !term
```

19.11.4. API Reference

19.11.4.1. Logical Operations

The functions cond, and, and or are short-circuiting. In other words, `false && BIG_EXPENSIVE_COMPUTATION` does not need to execute `BIG_EXPENSIVE_COMPUTATION` before returning `false`. These functions are defined using the **`macro_inline`** attribute, which causes the compiler to replace calls to them with their definitions while generating code, and the definitions use nested pattern matching to achieve the short-circuiting behavior.

def

```
cond.{u} {α : Sort u} (c : Bool) (x y : α) : α
```

The conditional function.

cond *c x y* is the same as **if** *c* **then** *x* **else** *y*, but optimized for a Boolean condition rather than a decidable proposition. It can also be written using the notation **bif** *c* **then** *x* **else** *y*.

Just like `ite`, cond is declared `@[macro_inline]`, which causes applications of cond to be unfolded. As a result, *x* and *y* are not evaluated at runtime until one of them is selected, and only the selected branch is evaluated.

def

```
Bool.dcond.{u} {α : Sort u} (c : Bool) (x : c = true → α)
  (y : c = false → α) : α
```

The dependent conditional function, in which each branch is provided with a local assumption about the condition's value. This allows the value to be used in proofs as well as for control flow.

`dcond c (fun h => x) (fun h => y)` is the same as **if** *h* : *c* **then** *x* **else** *y*, but optimized for a Boolean condition rather than a decidable proposition. Unlike the non-dependent version cond, there is no special notation for *dcond*.

Just like `ite`, `dite`, and cond, `dcond` is declared `@[macro_inline]`, which causes applications of `dcond` to be unfolded. As a result, *x* and *y* are not evaluated at runtime until one of them is selected, and only the selected branch is evaluated. `dcond` is intended for metaprogramming use, rather than for use in verified programs, so behavioral lemmas are not provided.

def

```
Bool.not : Bool → Bool
```

Boolean negation, also known as Boolean complement. not *x* can be written `!`*x*.

This is a function that maps the value `true` to `false` and the value `false` to `true`. The propositional connective is `Not : Prop → Prop`.

Conventions for notations in identifiers:

- The recommended spelling of `!` in identifiers is `not`.

def

```
Bool.and (x y : Bool) : Bool
```

Boolean “and”, also known as conjunction. and *x* *y* can be written *x* && *y*.

The corresponding propositional connective is `And : Prop → Prop → Prop`, written with the ∧ operator.

The Boolean and is a `@[macro_inline]` function in order to give it short-circuiting evaluation: if *x* is `false` then *y* is not evaluated at runtime.

Conventions for notations in identifiers:

- The recommended spelling of && in identifiers is and.
- The recommended spelling of || in identifiers is or.

def

```
Bool.or (x y : Bool) : Bool
```

Boolean “or”, also known as disjunction. `or x y` can be written `x || y`.

The corresponding propositional connective is `Or : Prop → Prop → Prop`, written with the ∨ operator.

The Boolean `or` is a `@[macro_inline]` function in order to give it short-circuiting evaluation: if *x* is `true` then *y* is not evaluated at runtime.

def

```
Bool.xor : Bool → Bool → Bool
```

Boolean “exclusive or”. `xor x y` can be written `x ^^ y`.

`x ^^ y` is `true` when precisely one of *x* or *y* is `true`. Unlike and and or, it does not have short-circuiting behavior, because one argument's value never determines the final value. Also unlike and and or, there is no commonly-used corresponding propositional connective.

Examples:

- `false ^^ false = false`
- `true ^^ false = true`
- `false ^^ true = true`
- `true ^^ true = false`

Conventions for notations in identifiers:

- The recommended spelling of ^^ in identifiers is xor.

19.11.4.2. Comparisons

Most comparisons on Booleans should be performed using the `DecidableEq Bool`, `LT Bool`, `LE Bool` instances.

def

```
Bool.decEq (a b : Bool) : Decidable (a = b)
```

Decides whether two Booleans are equal.

This function should normally be called via the `DecidableEq Bool` instance that it exists to support.

19.11.4.3. Conversions

def

```
Bool.toISize (b : Bool) : ISize
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toUInt8 (b : Bool) : UInt8
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toUInt16 (b : Bool) : UInt16
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toUInt32 (b : Bool) : UInt32
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toUInt64 (b : Bool) : UInt64
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toUSize (b : Bool) : USize
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toInt8 (b : Bool) : Int8
```

Converts `true` to 1 and `false` to 0.

def

```
Bool.toInt16 (b : Bool) : Int16
```

Converts `true` to 1 and `false` to `0`.

def

```
Bool.toInt32 (b : Bool) : Int32
```

Converts `true` to 1 and `false` to `0`.

def

```
Bool.toInt64 (b : Bool) : Int64
```

Converts `true` to 1 and `false` to `0`.

def

```
Bool.toNat (b : Bool) : Nat
```

Converts `true` to 1 and `false` to `0`.

def

```
Bool.toInt (b : Bool) : Int
```

Converts `true` to 1 and `false` to `0`.

19.12. Optional Values

`Option` α is the type of values which are either `some` v for some $v : \alpha$, or `none`. In functional programming, this type is used similarly to nullable types: `none` represents the absence of a value. Additionally, partial functions from α to β can be represented by the type $\alpha \to$ `Option` β, where `none` results when the function is undefined for some input. Computationally, these partial functions represent the possibility of failure or errors, and they correspond to a program that can terminate early but not throw an informative exception.

`Option` can also be thought of as being similar to a list that contains at most one element. From this perspective, iterating over `Option` consists of carrying out an operation only when a value is present. The `Option` API makes frequent use of this perspective.

▼ *Example: Options as Nullability*

The function `Std.HashMap.get?` looks up a specified key a : α inside a `HashMap` α β:

```
Std.HashMap.get?.{u, v} {α : Type u} {β : Type v}
  [BEq α] [Hashable α]
  (m : HashMap α β) (a : α) :
  Option β
```

Because there is no way to know in advance whether the key is actually in the map, the return type is `Option β`, where `none` means the key was not in the map, and `some b` means that the key was found and b is the value retrieved.

The `xs[i]` syntax, which is used to index into collections when there is an available proof that `i` is a valid index into `xs`, has a variant `xs[i]?` that returns an optional value depending on whether the given index is valid. If `m : HashMap α β` and `a : α`, then `m[a]?` is equivalent to `HashMap.get? m a`.

▼ *Example: Options as Safe Nullability*

In many programming languages, it is important to remember to check for the null value. When using `Option`, the type system requires these checks in the right places: `Option α` and `α` are not the same type, and converting from one to the other requires handling the case of `none`. This can be done via helpers such as `Option.getD`, or with pattern matching.

```
def postalCodes : Std.HashMap Nat String :=
  .empty |>.insert 12345 "Schenectady"

#eval postalCodes[12346]?.getD "not found"
```

```
"not found"
```

```
#eval
  match postalCodes[12346]? with
  | none => "not found"
  | some city => city
```

```
"not found"
```

```
#eval
  if let some city := postalCodes[12345]? then
    city
  else
    "not found"
```

```
"Schenectady"
```

inductive type

```
Option.{u} (α : Type u) : Type u
```

Optional values, which are either `some` around a value from the underlying type or `none`.

Option can represent nullable types or computations that might fail. In the codomain of a function type, it can also represent partiality.

Constructors

| none.{u} {α : Type u} : Option α

No value.

| some.{u} {α : Type u} (val : α) : Option α

Some value of type α.

19.12.1. Coercions

There is a coercion from α to Option α that wraps a value in some. This allows Option to be used in a style similar to nullable types in other languages, where values that are missing are indicated by none and values that are present are not specially marked.

▼ *Example: Coercions and* Option

In getAlpha, a line of input is read. If the line consists only of letters (after removing whitespace from the beginning and end of it), then it is returned; otherwise, the function returns none.

```
def getAlpha : IO (Option String) := do
  let line := (← (← IO.getStdin).getLine).trim
  if line.length > 0 && line.all Char.isAlpha then
    return line
  else
    return none
```

In the successful case, there is no explicit some wrapped around line. The some is automatically inserted by the coercion.

19.12.2. API Reference

19.12.2.1. Extracting Values

def

Option.get.{u} {α : Type u} (o : Option α) : o.isSome = true → α

Extracts the value from an option that can be proven to be some.

def

```
Option.get!.{u} {α : Type u} [Inhabited α] : Option α → α
```

Extracts the value from an Option, panicking on none.

def

```
Option.getD.{u_1} {α : Type u_1} (opt : Option α) (dflt : α) : α
```

Gets an optional value, returning a given default on none.

This function is `@[macro_inline]`, so `dflt` will not be evaluated unless `opt` turns out to be none.

Examples:

- `(some "hello").getD "goodbye" = "hello"`
- `none.getD "goodbye" = "hello"`

def

```
Option.getDM.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type u_1}
  [Pure m] (x : Option α) (y : m α) : m α
```

Gets the value in an option, monadically computing a default value on none.

This is the monadic analogue of Option.getD.

def

```
Option.getM.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type u_1}
  [Alternative m] : Option α → m α
```

Lifts an optional value to any Alternative, sending none to `failure`.

def

```
Option.elim.{u_1, u_2} {α : Type u_1} {β : Sort u_2} :
  Option α → β → (α → β) → β
```

A case analysis function for Option.

Given a value for none and a function to apply to the contents of some, Option.elim checks which constructor a given Option consists of, and uses the appropriate argument.

Option.elim is an elimination principle for Option. In particular, it is a non-dependent version of `Option.recOn`. It can also be seen as a combination of Option.map and Option.getD.

Examples:

- `(some "hello").elim 0 String.length = 5`
- `none.elim 0 String.length = 0`

def

```
Option.elimM.{u_1, u_2} {m : Type u_1 → Type u_2} {α β : Type u_1}
  [Monad m] (x : m (Option α)) (y : m β) (z : α → m β) : m β
```

A monadic case analysis function for `Option`.

Given a fallback computation for `none` and a monadic operation to apply to the contents of `some`, `Option.elimM` checks which constructor a given `Option` consists of, and uses the appropriate argument.

`Option.elimM` can also be seen as a combination of `Option.mapM` and `Option.getDM`. It is a monadic analogue of `Option.elim`.

def

```
Option.merge.{u_1} {α : Type u_1} (fn : α → α → α) :
  Option α → Option α → Option α
```

Applies a function to a two optional values if both are present. Otherwise, if one value is present, it is returned and the function is not used.

The value is `some (fn a b)` if the inputs are `some a` and `some b`. Otherwise, the behavior is equivalent to `Option.orElse`: if only one input is `some x`, then the value is `some x`, and if both are `none`, then the value is `none`.

Examples:

- `Option.merge (· + ·) none (some 3) = some 3`
- `Option.merge (· + ·) (some 2) (some 3) = some 5`
- `Option.merge (· + ·) (some 2) none = some 2`
- `Option.merge (· + ·) none none = none`

19.12.2.2. Properties and Comparisons

def

```
Option.isNone.{u_1} {α : Type u_1} : Option α → Bool
```

Returns `true` on `none` and `false` on `some x`.

This function is more flexible than `(· == none)` because it does not require a `BEq α` instance.

Examples:

- `(none : Option Nat).isNone = true`
- `(some Nat.add).isNone = false`

def

`Option.isSome.{u_1} {α : Type u_1} : Option α → Bool`

Returns `true` on `some x` and `false` on `none`.

def

`Option.isEqSome.{u_1} {α : Type u_1} [BEq α] : Option α → α → Bool`

Checks whether an optional value is both present and equal to some other value.

Given `x? : Option α` and `y : α`, `x?.isEqSome y` is equivalent to `x? == some y`. It is more efficient because it avoids an allocation.

Ordering of optional values typically uses the `DecidableEq (Option α)`, `LT (Option α)`, `Min (Option α)`, and `Max (Option α)` instances.

def

`Option.min.{u_1} {α : Type u_1} [Min α] : Option α → Option α → Option α`

The minimum of two optional values, with `none` treated as the least element. This function is usually accessed through the `Min (Option α)` instance, rather than directly.

Prior to *nightly-2025-02-27*, `none` was treated as the greatest element, so `min none (some x) = min (some x) none = some x`.

Examples:

- `Option.min (some 2) (some 5) = some 2`
- `Option.min (some 5) (some 2) = some 2`
- `Option.min (some 2) none = none`
- `Option.min none (some 5) = none`
- `Option.min none none = none`

def

```
Option.max.{u_1} {α : Type u_1} [Max α] : Option α → Option α →
Option α
```

The maximum of two optional values.

This function is usually accessed through the `Max (Option α)` instance, rather than directly.

Examples:

- `Option.max (some 2) (some 5) = some 5`
- `Option.max (some 5) (some 2) = some 5`
- `Option.max (some 2) none = some 2`
- `Option.max none (some 5) = some 5`
- `Option.max none none = none`

def

```
Option.lt.{u_1, u_2} {α : Type u_1} {β : Type u_2} (r : α → β →
Prop) :
  Option α → Option β → Prop
```

Lifts an ordering relation to `Option`, such that `none` is the least element.

It can be understood as adding a distinguished least element, represented by `none`, to both α and β.

This definition is part of the implementation of the `LT (Option α)` instance. However, because it can be used with heterogeneous relations, it is sometimes useful on its own.

Examples:

- `Option.lt (fun n k : Nat => n < k) none none = False`
- `Option.lt (fun n k : Nat => n < k) none (some 3) = True`
- `Option.lt (fun n k : Nat => n < k) (some 3) none = False`
- `Option.lt (fun n k : Nat => n < k) (some 4) (some 5) = True`
- `Option.lt (fun n k : Nat => n < k) (some 4) (some 4) = False`

def

```
Option.decidableEqNone.{u_1} {α : Type u_1} {o : Option α} :
  Decidable (o = none)
```

Equality with `none` is decidable even if the wrapped type does not have decidable equality.

This is not an instance because it is not definitionally equal to the standard instance of `DecidableEq (Option α)`, which can cause problems. It can be locally bound if needed.

Try to use the Boolean comparisons `Option.isNone` or `Option.isSome` instead.

19.12.2.3. Conversion

def

```
Option.toArray.{u_1} {α : Type u_1} : Option α → Array α
```

Converts an optional value to an array with zero or one element.

Examples:

- `(some "value").toArray = #["value"]`
- `none.toArray = #[]`

def

```
Option.toList.{u_1} {α : Type u_1} : Option α → List α
```

Converts an optional value to a list with zero or one element.

Examples:

- `(some "value").toList = ["value"]`
- `none.toList = []`

def

```
Option.repr.{u_1} {α : Type u_1} [Repr α] : Option α → Nat →
Std.Format
```

Returns a representation of an optional value that should be able to be parsed as an equivalent optional value.

This function is typically accessed through the `Repr (Option α)` instance.

def

```
Option.format.{u} {α : Type u} [Std.ToFormat α] : Option α →
Std.Format
```

Formats an optional value, with no expectation that the Lean parser should be able to parse the result.

This function is usually accessed through the `ToFormat (Option α)` instance.

19.12.2.4. Control

`Option` can be thought of as describing a computation that may fail to return a value. The `Monad Option` instance, along with `Alternative Option`, is based on this understanding. Returning `none` can also be thought of as throwing an exception that contains no interesting information, which is captured in the `MonadExcept Unit Option` instance.

def

```
Option.guard.{u_1} {α : Type u_1} (p : α → Bool) (a : α) : Option α
```

Returns `none` if a value doesn't satisfy a Boolean predicate, or the value itself otherwise.

From the perspective of `Option` as computations that might fail, this function is a run-time assertion operator in the `Option` monad.

Examples:

- `Option.guard (· > 2) 1 = none`
- `Option.guard (· > 2) 5 = some 5`

def

```
Option.bind.{u_1, u_2} {α : Type u_1} {β : Type u_2} :
  Option α → (α → Option β) → Option β
```

Sequencing of `Option` computations.

From the perspective of `Option` as computations that might fail, this function sequences potentially-failing computations, failing if either fails. From the perspective of `Option` as a collection with at most one element, the function is applied to the element if present, and the final result is empty if either the initial or the resulting collections are empty.

This function is often accessed via the >>= operator from the `Bind (Option α)` instance, or implicitly via **do**-notation, but it is also idiomatic to call it using generalized field notation.

Examples:

- `none.bind (fun x => some x) = none`
- `(some 4).bind (fun x => some x) = some 4`
- `none.bind (Option.guard (· > 2)) = none`
- `(some 2).bind (Option.guard (· > 2)) = none`
- `(some 4).bind (Option.guard (· > 2)) = some 4`

def

```
Option.bindM.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type
u_3}
  {β : Type u_1} [Pure m] (f : α → m (Option β)) :
  Option α → m (Option β)
```

Runs the monadic action f on o's value, if any, and returns the result, or none if there is no value.

From the perspective of Option as a collection with at most one element, the monadic the function is applied to the element if present, and the final result is empty if either the initial or the resulting collections are empty.

def

```
Option.join.{u_1} {α : Type u_1} (x : Option (Option α)) : Option α
```

Flattens nested optional values, preserving any value found.

This is analogous to List.flatten.

Examples:

- none.join = none
- (some none).join = none
- (some (some v)).join = some v

def

```
Option.sequence.{u, u_1} {m : Type u → Type u_1} [Applicative m]
  {α : Type u} : Option (m α) → m (Option α)
```

Converts an optional monadic computation into a monadic computation of an optional value.

This function only requires m to be an applicative functor.

Example:

```
#eval show IO (Option String) from
  Option.sequence <| some do
    IO.println "hello"
    return "world"
```

```
hello
```

```
some "world"
```

def

```
Option.tryCatch.{u_1} {α : Type u_1} (x : Option α)
  (handle : Unit → Option α) : Option α
```

Recover from failing `Option` computations with a handler function.

This function is usually accessed through the `MonadExceptOf Unit Option` instance.

Examples:

- `Option.tryCatch none (fun () => some "handled") = some "handled"`
- `Option.tryCatch (some "succeeded") (fun () => some "handled") = some "succeeded"`

def

```
Option.or.{u_1} {α : Type u_1} : Option α → Option α → Option α
```

Returns the first of its arguments that is `some`, or `none` if neither is `some`.

This is similar to the <|> operator, also known as `OrElse.orElse`, but both arguments are always evaluated without short-circuiting.

def

```
Option.orElse.{u_1} {α : Type u_1} :
  Option α → (Unit → Option α) → Option α
```

Implementation of `OrElse`'s <|> syntax for `Option`. If the first argument is `some a`, returns `some a`, otherwise evaluates and returns the second argument.

See also `or` for a version that is strict in the second argument.

19.12.2.5. Iteration

`Option` can be thought of as a collection that contains at most one value. From this perspective, iteration operators can be understood as performing some operation on the contained value, if present, or doing nothing if not.

def

```
Option.all.{u_1} {α : Type u_1} (p : α → Bool) : Option α → Bool
```

Checks whether an optional value either satisfies a Boolean predicate or is `none`.

Examples:

- `(some 33).all (· % 2 == 0) = false
- `(some 22).all (· % 2 == 0) = true
- `none.all (fun x : Nat => x % 2 == 0) = true

def

```
Option.any.{u_1} {α : Type u_1} (p : α → Bool) : Option α → Bool
```

Checks whether an optional value is not `none` and satisfies a Boolean predicate.

Examples:

- `(some 33).any (· % 2 == 0) = false
- `(some 22).any (· % 2 == 0) = true
- `none.any (fun x : Nat => true) = false

def

```
Option.filter.{u_1} {α : Type u_1} (p : α → Bool) : Option α →
Option α
```

Keeps an optional value only if it satisfies a Boolean predicate.

If `Option` is thought of as a collection that contains at most one element, then `Option.filter` is analogous to `List.filter` or `Array.filter`.

Examples:

- `(some 5).filter (· % 2 == 0) = none`
- `(some 4).filter (· % 2 == 0) = some 4`
- `none.filter (fun x : Nat => x % 2 == 0) = none`
- `none.filter (fun x : Nat => true) = none`

def

```
Option.forM.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type
u_3}
  [Pure m] : Option α → (α → m PUnit) → m PUnit
```

Executes a monadic action on an optional value if it is present, or does nothing if there is no value.

Examples:

```
#eval ((some 5).forM set : StateM Nat Unit).run 0
```

```
((), 5)

#eval (none.forM (fun x : Nat => set x) : StateM Nat Unit).run 0

((), 0)
```

def

```
Option.map.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α → β) :
  Option α → Option β
```

Apply a function to an optional value, if present.

From the perspective of `Option` as a container with at most one value, this is analogous to `List.map`. It can also be accessed via the `Functor Option` instance.

Examples:

- `(none : Option Nat).map (· + 1) = none`
- `(some 3).map (· + 1) = some 4`

def

```
Option.mapA.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type
u_3}
  {β : Type u_1} [Applicative m] (f : α → m β) : Option α → m
(Option β)
```

Applies a function in some applicative functor to an optional value, returning `none` with no effects if the value is missing.

This is an alias for `Option.mapM`, which already works for applicative functors.

def

```
Option.mapM.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type
u_3}
  {β : Type u_1} [Applicative m] (f : α → m β) : Option α → m
(Option β)
```

Applies a function in some applicative functor to an optional value, returning `none` with no effects if the value is missing.

Runs a monadic function f on an optional value, returning the result. If the optional value is `none`, the function is not called and the result is also `none`.

From the perspective of `Option` as a container with at most one element, this is analogous to `List.mapM`, returning the result of running the monadic function on all elements of the container.

This function only requires *m* to be an applicative functor. An alias Option.mapA is provided.

19.12.2.6. Recursion Helpers

def

```
Option.attach.{u_1} {α : Type u_1} (xs : Option α) :
  Option { x // xs = some x }
```

“Attaches” a proof that an optional value, if present, is indeed this value, returning a subtype that expresses this fact.

This function is primarily used to allow definitions by well-founded recursion that use iteration operators (such as Option.map) to prove that an optional value drawn from a parameter is smaller than the parameter. This allows the well-founded recursion mechanism to prove that the function terminates.

def

```
Option.attachWith.{u_1} {α : Type u_1} (xs : Option α) (P : α →
Prop)
  (H : ∀ (x : α), xs = some x → P x) : Option { x // P x }
```

“Attaches” a proof that some predicate holds for an optional value, if present, returning a subtype that expresses this fact.

This function is primarily used to implement Option.attach, which allows definitions by well-founded recursion that use iteration operators (such as Option.map) to prove that an optional value drawn from a parameter is smaller than the parameter. This allows the well-founded recursion mechanism to prove that the function terminates.

def

```
Option.unattach.{u_1} {α : Type u_1} {p : α → Prop}
  (o : Option { x // p x }) : Option α
```

Remove an attached proof that the value in an Option is indeed that value.

This function is usually inserted automatically by Lean, rather than explicitly in code. It is introduced as an intermediate step during the elaboration of definitions by well-founded recursion.

If this function is encountered in a proof state, the right approach is usually the tactic **simp** `[Option.unattach, -Option.map_subtype]`.

It is a synonym for Option.map Subtype.val.

19.12.2.7. Reasoning

def

```
Option.choice.{u_1} (α : Type u_1) : Option α
```

An optional arbitrary element of a given type.

If `α` is non-empty, then there exists some `v : α` and this arbitrary element is `some v`. Otherwise, it is `none`.

def

```
Option.pbind.{u_1, u_2} {α : Type u_1} {β : Type u_2} (o : Option α)
  (f : (a : α) → o = some a → Option β) : Option β
```

Given an optional value and a function that can be applied when the value is `some`, returns the result of applying the function if this is possible.

The function `f` is *partial* because it is only defined for the values `a : α` such that `o = some a`. This restriction allows the function to use the fact that it can only be called when `o` is not `none`: it can relate its argument to the optional value `o`. Its runtime behavior is equivalent to that of `Option.bind`.

Examples:

```
def attach (v : Option α) : Option { y : α // v = some y } :=
  v.pbind fun x h => some ⟨x, h⟩
```

```
#reduce attach (some 3)
```

```
some ⟨3, ⋯⟩
```

```
#reduce attach none
```

```
none
```

def

```
Option.pelim.{u_1, u_2} {α : Type u_1} {β : Sort u_2} (o : Option α)
  (b : β) (f : (a : α) → o = some a → β) : β
```

Given an optional value and a function that can be applied when the value is `some`, returns the result of applying the function if this is possible, or a fallback value otherwise.

The function `f` is *partial* because it is only defined for the values `a : α` such that `o = some a`. This restriction allows the function to use the fact that it can only be called when `o` is not `none`: it can relate its argument to the optional value `o`. Its runtime behavior is equivalent to

that of Option.elim.

Examples:

```
def attach (v : Option α) : Option { y : α // v = some y } :=
  v.pelim none fun x h => some ⟨x, h⟩

#reduce attach (some 3)
```

```
some ⟨3, ⋯⟩
```

```
#reduce attach none
```

```
none
```

def

```
Option.pmap.{u_1, u_2} {α : Type u_1} {β : Type u_2} {p : α → Prop}
  (f : (a : α) → p a → β) (o : Option α) :
  (∀ (a : α), o = some a → p a) → Option β
```

Given a function from the elements of α that satisfy p to β and a proof that an optional value satisfies p if it's present, applies the function to the value.

Examples:

```
def attach (v : Option α) : Option { y : α // v = some y } :=
  v.pmap (fun a (h : a ∈ v) => ⟨_, h⟩) (fun _ h => h)

#reduce attach (some 3)
```

```
some ⟨3, ⋯⟩
```

```
#reduce attach none
```

```
none
```

19.13. Tuples

The Lean standard library includes a variety of tuple-like types. In practice, they differ in four ways:

- whether the first projection is a type or a proposition
- whether the second projection is a type or a proposition
- whether the second projection's type depends on the first projection's value
- whether the type as a whole is a proposition or type

Type	First Projection	Second Projection	Dependent?	Universe
Prod	Type u	Type v	❌	Type (max u v)
And	Prop	Prop	❌	Prop
Sigma	Type u	Type v	✓	Type (max u v)
Subtype	Type u	Prop	✓	Type u
Exists	Type u	Prop	✓	Prop

Some potential rows in this table do not exist in the library:

- There is no dependent pair where the first projection is a proposition, because proof irrelevance renders this meaningless.
- There is no non-dependent pair that combines a type with a proposition because the situation is rare in practice: grouping data with *unrelated* proofs is uncommon.

These differences lead to very different use cases. `Prod` and its variants `PProd` and `MProd` simply group data together—they are products. Because its second projection is dependent, `Sigma` has the character of a sum: for each element of the first projection's type, there may be a different type in the second projection. `Subtype` selects the values of a type that satisfy a predicate. Even though it syntactically resembles a pair, in practice it is treated as an actual subset. `And` is a logical connective, and `Exists` is a quantifier. This chapter documents the tuple-like pairs, namely `Prod` and `Sigma`.

19.13.1. Ordered Pairs

The type $\alpha \times \beta$, which is a notation for `Prod α β`, contains ordered pairs in which the first item is an α and the second is a β. These pairs are written in parentheses, separated by commas. Larger tuples are represented as nested tuples, so $\alpha \times \beta \times \gamma$ is equivalent to $\alpha \times (\beta \times \gamma)$ and (x, y, z) is equivalent to $(x, (y, z))$.

syntax

Product Types

```
term ::= ...
    | term × term
```

The product `Prod α β` is written $\alpha \times \beta$.

syntax

Pairs

```
term ::= ...
    | (term, term)
```

structure

`Prod.{u, v} (α : Type u) (β : Type v) : Type (max u v)`

The product type, usually written `α × β`. Product types are also called pair or tuple types. Elements of this type are pairs in which the first element is an `α` and the second element is a `β`.

Products nest to the right, so `(x, y, z) : α × β × γ` is equivalent to `(x, (y, z)) : α × (β × γ)`.

Conventions for notations in identifiers:

- The recommended spelling of × in identifiers is Prod.

Constructor

Prod.mk.{u, v}

Constructs a pair. This is usually written `(x, y)` instead of Prod.mk `x y`.

Conventions for notations in identifiers:

- The recommended spelling of `(a, b)` in identifiers is mk.

Fields

`fst : α`

The first element of a pair.

`snd : β`

The second element of a pair.

There are also the variants `α ×' β` (which is notation for PProd `α β`) and MProd, which differ with respect to universe levels: like PSum, PProd allows either `α` or `β` to be a proposition, while MProd requires that both be types at the *same* universe level. Generally speaking, PProd is primarily used in the implementation of proof automation and the elaborator, as it tends to give rise to universe level unification problems that can't be solved. MProd, on the other hand, can simplify universe level issues in certain advanced use cases.

syntax

Products of Arbitrary Sorts

```
term ::= ...
    | term ×' term
```

The product PProd α β, in which both types could be propositions, is written α × β.

structure

```
PProd.{u, v} (α : Sort u) (β : Sort v) : Sort (max (max 1 u) v)
```

A product type in which the types may be propositions, usually written α ×' β.

This type is primarily used internally and as an implementation detail of proof automation. It is rarely useful in hand-written code.

Conventions for notations in identifiers:

- The recommended spelling of ×' in identifiers is PProd.

Constructor

PProd.mk.{u, v}

Fields

fst : α

The first element of a pair.

snd : β

The second element of a pair.

structure

```
MProd.{u} (α β : Type u) : Type u
```

A product type in which both α and β are in the same universe.

It is called MProd is because it is the *universe-monomorphic* product type.

Constructor

MProd.mk.{u}

Fields

fst : α

The first element of a pair.

`snd : β`

The second element of a pair.

19.13.1.1. API Reference

As a mere pair, the primary API for `Prod` is provided by pattern matching and by the first and second projections `Prod.fst` and `Prod.snd`.

19.13.1.1.1. Transformation

def

```
Prod.map.{u₁, u₂, v₁, v₂} {α₁ : Type u₁} {α₂ : Type u₂} {β₁ : Type v₁}
  {β₂ : Type v₂} (f : α₁ → α₂) (g : β₁ → β₂) : α₁ × β₁ → α₂ × β₂
```

Transforms a pair by applying functions to both elements.

Examples:

- `(1, 2).map (· + 1) (· * 3) = (2, 6)`
- `(1, 2).map toString (· * 3) = ("1", 6)`

def

```
Prod.swap.{u_1, u_2} {α : Type u_1} {β : Type u_2} : α × β → β × α
```

Swaps the elements in a pair.

Examples:

- `(1, 2).swap = (2, 1)`
- `("orange", -87).swap = (-87, "orange")`

19.13.1.1.2. Natural Number Ranges

def

```
Prod.allI (i : Nat × Nat)
  (f : (j : Nat) → i.fst ≤ j → j ≤ i.snd → Bool) : Bool
```

Checks whether a predicate holds for all natural numbers in a range.

In particular, `(start, stop).allI f` returns true if `f` is true for all natural numbers from `start` (inclusive) to `stop` (exclusive).

Examples:

- `(5, 8).allI (fun j _ _ => j < 10) = (5 < 10) && (6 < 10) && (7 < 10)`
- `(5, 8).allI (fun j _ _ => j % 2 = 0) = false`
- `(6, 7).allI (fun j _ _ => j % 2 = 0) = true`

def

```
Prod.anyI (i : Nat × Nat)
  (f : (j : Nat) → i.fst ≤ j → j ≤ i.snd → Bool) : Bool
```

Checks whether a predicate holds for any natural number in a range.

In particular, `(start, stop).allI f` returns true if `f` is true for any natural number from `start` (inclusive) to `stop` (exclusive).

Examples:

- `(5, 8).anyI (fun j _ _ => j == 6) = (5 == 6) || (6 == 6) || (7 == 6)`
- `(5, 8).anyI (fun j _ _ => j % 2 = 0) = true`
- `(6, 6).anyI (fun j _ _ => j % 2 = 0) = false`

def

```
Prod.foldI.{u} {α : Type u} (i : Nat × Nat)
  (f : (j : Nat) → i.fst ≤ j → j ≤ i.snd → α → α) (init : α) : α
```

Combines an initial value with each natural number from a range, in increasing order.

In particular, `(start, stop).foldI f init` applies `f` on all the numbers from `start` (inclusive) to `stop` (exclusive) in increasing order:

Examples:

- `(5, 8).foldI (fun j _ _ xs => xs.push j) #[] = (#[] |>.push 5 |>.push 6 |>.push 7)`
- `(5, 8).foldI (fun j _ _ xs => xs.push j) #[] = #[5, 6, 7]`
- `(5, 8).foldI (fun j _ _ xs => toString j :: xs) [] = ["7", "6", "5"]`

19.13.1.1.3. Ordering

def

```
Prod.lexLt.{u_1, u_2} {α : Type u_1} {β : Type u_2} [LT α] [LT β]
  (s t : α × β) : Prop
```

Lexicographical order for products.

Two pairs are lexicographically ordered if their first elements are ordered or if their first elements are equal and their second elements are ordered.

19.13.2. Dependent Pairs

Dependent pairs, also known as *dependent sums* or *Σ-types*, are pairs in which the second term's type may depend on the *value* of the first term. They are closely related to the existential quantifier and Subtype. Unlike existentially quantified statements, dependent pairs are in the Type universe and are computationally relevant data. Unlike subtypes, the second term is also computationally relevant data. Like ordinary pairs, dependent pairs may be nested; this nesting is right-associative.

syntax

Dependent Pair Types

```
term ::= ...
    | (ident : term) × term

term ::= ...
    | Σ ident ident* (: term)?, term

term ::= ...
    | Σ (ident ident* : term), term
```

Dependent pair types bind one or more variables, which are then in scope in the final term. If there is one variable, then its type is a that of the first element in the pair and the final term is the type of the second element in the pair. If there is more than one variable, the types are nested right-associatively. The identifiers may also be _. With parentheses, multiple bound variables may have different types, while the unparenthesized variant requires that all have the same type.

▼ *Example: Nested Dependent Pair Types*

The type

```
Σ n k : Nat, Fin (n * k)
```

is equivalent to

```
Σ n : Nat, Σ k : Nat, Fin (n * k)
```

and

```
(n : Nat) × (k : Nat) × Fin (n * k)
```

The type

```
Σ (n k : Nat) (i : Fin (n * k)) , Fin i.val
```

is equivalent to

```
Σ (n : Nat), Σ (k : Nat), Σ (i : Fin (n * k)) , Fin i.val
```

and

```
(n : Nat) × (k : Nat) × (i : Fin (n * k)) × Fin i.val
```

The two styles of annotation cannot be mixed in a single Σ-type:

```
1 Σ n k (i : Fin (n * k)) , Fin i.val
```

```
<example>:1:5-1:7: unexpected token '('; expected ','
```

Dependent pairs are typically used in one of two ways:

1. They can be used to “package” a concrete type index together with a value of the indexed family, used when the index value is not known ahead of time. The type `Σ n, Fin n` is a pair of a natural number and some other number that's strictly smaller. This is the most common way to use dependent pairs.

2. The first element can be thought of as a “tag” that's used to select from among different types for the second term. This is similar to the way that selecting a constructor of a sum type determines the types of the constructor's arguments. For example, the type

   ```
   Σ (b : Bool), if b then Unit else α
   ```

   is equivalent to `Option α`, where `none` is `⟨true, ()⟩` and `some x` is `⟨false, x⟩`. Using dependent pairs this way is uncommon, because it's typically much easier to define a special-purpose inductive type directly.

structure

```
Sigma.{u, v} {α : Type u} (β : α → Type v) : Type (max u v)
```

Dependent pairs, in which the second element's type depends on the value of the first element. The type `Sigma β` is typically written `Σ a : α, β a` or `(a : α) × β a`.

Although its values are pairs, `Sigma` is sometimes known as the *dependent sum type*, since it is the type level version of an indexed summation.

Constructor

`Sigma.mk.{u, v}`

Constructs a dependent pair.

Using this constructor in a context in which the type is not known usually requires a type ascription to determine `β`. This is because the desired relationship between the two values can't generally be determined automatically.

Fields

`fst : α`

The first component of a dependent pair.

`snd : β self.fst`

The second component of a dependent pair. Its type depends on the first component.

▼ *Example: Dependent Pairs with Data*

The type `Vector`, which associates a known length with an array, can be placed in a dependent pair with the length itself. While this is logically equivalent to just using `Array`, this construction is sometimes necessary to bridge gaps in an API.

```
def getNLinesRev : (n : Nat) → IO (Vector String n)
  | 0 => pure #v[]
  | n + 1 => do
    let xs ← getNLinesRev n
    return xs.push (← (← IO.getStdin).getLine)

def getNLines (n : Nat) : IO (Vector String n) := do
  return (← getNLinesRev n).reverse

partial def getValues : IO (Σ n, Vector String n) := do
  let stdin ← IO.getStdin

  IO.println "How many lines to read?"
  let howMany ← stdin.getLine

  if let some howMany := howMany.trim.toNat? then
    return ⟨howMany, (← getNLines howMany)⟩
  else
    IO.eprintln "Please enter a number."
    getValues

def main : IO Unit := do
  let values ← getValues
  IO.println s!"Got {values.fst} values. They are:"
  for x in values.snd do
    IO.println x.trim
```

When calling the program with this standard input:

```
stdin
  1 4
  2 Apples
  3 Quince
  4 Plums
```

```
   5 Raspberries
```

the output is:

```
stdout
   1 How many lines to read?
   2 Got 4 values. They are:
   3 Raspberries
   4 Plums
   5 Quince
   6 Apples
```

▼ *Example: Dependent Pairs as Sums*

Sigma can be used to implement sum types. The Bool in the first projection of Sum' indicates which type the second projection is drawn from.

```
def Sum' (α : Type) (β : Type) : Type :=
  Σ (b : Bool),
    match b with
    | true => α
    | false => β
```

The injections pair a tag (a Bool) with a value of the indicated type. Annotating them with **match_pattern** allows them to be used in patterns as well as in ordinary terms.

```
variable {α β : Type}

@[match_pattern]
def Sum'.inl (x : α) : Sum' α β := ⟨true, x⟩

@[match_pattern]
def Sum'.inr (x : β) : Sum' α β := ⟨false, x⟩

def Sum'.swap : Sum' α β → Sum' β α
  | .inl x => .inr x
  | .inr y => .inl y
```

Just as Prod has a variant PProd that accepts propositions as well as types, PSigma allows its projections to be propositions. This has the same drawbacks as PProd: it is much more likely to lead to failures of universe level unification. However, PSigma can be necessary when implementing custom proof automation or in some rare, advanced use cases.

syntax

Fully-Polymorphic Dependent Pair Types

```
term ::= ...
    | Σ' ident ident* (: term)? , term

term ::= ...
    | Σ' (ident ident* : term), term
```

The rules for nesting `Σ'`, as well as those that govern its binding structure, are the same as those for `Σ`.

structure

`PSigma.{u, v} {α : Sort u} (β : α → Sort v) : Sort (max (max 1 u) v)`

Fully universe-polymorphic dependent pairs, in which the second element's type depends on the value of the first element and both types are allowed to be propositions. The type `PSigma β` is typically written `Σ' a : α, β a` or `(a : α) ×' β a`.

In practice, this generality leads to universe level constraints that are difficult to solve, so `PSigma` is rarely used in manually-written code. It is usually only used in automation that constructs pairs of arbitrary types.

To pair a value with a proof that a predicate holds for it, use `Subtype`. To demonstrate that a value exists that satisfies a predicate, use `Exists`. A dependent pair with a proposition as its first component is not typically useful due to proof irrelevance: there's no point in depending on a specific proof because all proofs are equal anyway.

Constructor

`PSigma.mk.{u, v}`

Constructs a fully universe-polymorphic dependent pair.

Fields

`fst : α`

The first component of a dependent pair.

`snd : β self.fst`

The second component of a dependent pair. Its type depends on the first component.

19.14. Sum Types

Sum types represent a choice between two types: an element of the sum is an element of one of the other types, paired with an indication of which type it came from. Sums are also known as disjoint unions, discriminated unions, or tagged unions. The constructors of a sum are also called *injections*; mathematically, they can be considered as injective functions from each summand to the sum.

There are two varieties of the sum type:

- Sum is polymorphic over all Type universes, and is never a proposition.
- PSum is allows the summands to be propositions or types. Unlike Or, the PSum of two propositions is still a type, and non-propositional code can check which injection was used to construct a given value.

Manually-written Lean code almost always uses only Sum, while PSum is used as part of the implementation of proof automation. This is because it imposes problematic constraints that universe level unification cannot solve. In particular, this type is in the universe `Sort (max 1 u v)`, which can cause problems for universe level unification because the equation `max 1 u v = ? u + 1` has no solution in level arithmetic. PSum is usually only used in automation that constructs sums of arbitrary types.

inductive type

```
Sum.{u, v} (α : Type u) (β : Type v) : Type (max u v)
```

The disjoint union of types α and β, ordinarily written $\alpha \oplus \beta$.

An element of $\alpha \oplus \beta$ is either an $a : \alpha$ wrapped in Sum.inl or a $b : \beta$ wrapped in Sum.inr. $\alpha \oplus \beta$ is not equivalent to the set-theoretic union of α and β because its values include an indication of which of the two types was chosen. The union of a singleton set with itself contains one element, while Unit ⊕ Unit contains distinct values `inl ()` and `inr ()`.

Constructors

▎`inl.{u, v} {α : Type u} {β : Type v} (val : α) : α ⊕ β`

Left injection into the sum type $\alpha \oplus \beta$.

▎`inr.{u, v} {α : Type u} {β : Type v} (val : β) : α ⊕ β`

Right injection into the sum type $\alpha \oplus \beta$.

inductive type

```
PSum.{u, v} (α : Sort u) (β : Sort v) : Sort (max (max 1 u) v)
```

The disjoint union of arbitrary sorts α β, or α ⊕' β.

It differs from $\alpha \oplus \beta$ in that it allows α and β to have arbitrary sorts `Sort u` and `Sort v`, instead of restricting them to `Type u` and `Type v`. This means that it can be used in situations where one side is a proposition, like `True ⊕' Nat`. However, the resulting universe level constraints are often more difficult to solve than those that result from Sum.

Constructors

```
| inl.{u, v} {α : Sort u} {β : Sort v} (val : α) : α ⊕' β
```

Left injection into the sum type $\alpha \oplus' \beta$.

```
| inr.{u, v} {α : Sort u} {β : Sort v} (val : β) : α ⊕' β
```

Right injection into the sum type $\alpha \oplus' \beta$.

19.14.1. Syntax

The names Sum and PSum are rarely written explicitly. Most code uses the corresponding infix operators.

syntax

Sum Types

```
term ::= ...
    | term ⊕ term
```

$\alpha \oplus \beta$ is notation for `Sum α β`.

syntax

Potentially-Propositional Sum Types

```
term ::= ...
    | term ⊕' term
```

$\alpha \oplus' \beta$ is notation for `PSum α β`.

19.14.2. API Reference

Sum types are primarily used with pattern matching rather than explicit function calls from an API. As such, their primary API is the constructors `inl` and `inr`.

19.14.2.1. Case Distinction

def

```
Sum.isLeft.{u_1, u_2} {α : Type u_1} {β : Type u_2} : α ⊕ β → Bool
```

Checks whether a sum is the left injection `inl`.

def

```
Sum.isRight.{u_1, u_2} {α : Type u_1} {β : Type u_2} : α ⊕ β →
Bool
```

Checks whether a sum is the right injection `inr`.

19.14.2.2. Extracting Values

def

```
Sum.elim.{u_1, u_2, u_3} {α : Type u_1} {β : Type u_2} {γ : Sort
u_3}
  (f : α → γ) (g : β → γ) : α ⊕ β → γ
```

Case analysis for sums that applies the appropriate function f or g after checking which constructor is present.

def

```
Sum.getLeft.{u_1, u_2} {α : Type u_1} {β : Type u_2} (ab : α ⊕ β)
:
  ab.isLeft = true → α
```

Retrieves the contents from a sum known to be `inl`.

def

```
Sum.getLeft?.{u_1, u_2} {α : Type u_1} {β : Type u_2} : α ⊕ β →
Option α
```

Checks whether a sum is the left injection `inl` and, if so, retrieves its contents.

def

```
Sum.getRight.{u_1, u_2} {α : Type u_1} {β : Type u_2} (ab : α ⊕ β)
:
  ab.isRight = true → β
```

Retrieves the contents from a sum known to be `inr`.

def

```
Sum.getRight?.{u_1, u_2} {α : Type u_1} {β : Type u_2} :
  α ⊕ β → Option β
```

Checks whether a sum is the right injection `inr` and, if so, retrieves its contents.

19.14.2.3. Transformations

def

```
Sum.map.{u_1, u_2, u_3, u_4} {α : Type u_1} {α' : Type u_2}
  {β : Type u_3} {β' : Type u_4} (f : α → α') (g : β → β') :
  α ⊕ β → α' ⊕ β'
```

Transforms a sum according to functions on each type.

This function maps $\alpha \oplus \beta$ to $\alpha' \oplus \beta'$, sending α to α' and β to β'.

def

```
Sum.swap.{u_1, u_2} {α : Type u_1} {β : Type u_2} : α ⊕ β → β ⊕ α
```

Swaps the factors of a sum type.

The constructor `Sum.inl` is replaced with `Sum.inr`, and vice versa.

19.14.2.4. Inhabited

The `Inhabited` definitions for `Sum` and `PSum` are not registered as instances. This is because there are two separate ways to construct a default value (via `inl` or `inr`), and instance synthesis might result in either choice. The result could be situations where two identically-written terms elaborate differently and are not definitionally equal.

Both types have `Nonempty` instances, for which proof irrelevance makes the choice of `inl` or `inr` not matter. This is enough to enable `partial` functions. For situations that require an `Inhabited` instance, such as programs that use `panic!`, the instance can be explicitly used by adding it to the local context with `have` or `let`.

▼ *Example: Inhabited Sum Types*

In Lean's logic, `panic!` is equivalent to the default value specified in its type's `Inhabited` instance. This means that the type must have such an instance—a `Nonempty` instance combined with the axiom of choice would render the program non-computable.

Products have the right instance:

```
example : Nat × String := panic! "Cant' find it"
```

Sums do not, by default:

```
example : Nat ⊕ String := panic! "Cant' find it"
```

```
failed to synthesize
  Inhabited (Nat ⊕ String)

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

The desired instance can be made available to instance synthesis using `have`:

```
example : Nat ⊕ String :=
  have : Inhabited (Nat ⊕ String) := Sum.inhabitedLeft
  panic! "Cant' find it"
```

def

```
Sum.inhabitedLeft.{u, v} {α : Type u} {β : Type v} [Inhabited α] :
  Inhabited (α ⊕ β)
```

If the left type in a sum is inhabited then the sum is inhabited.

This is not an instance to avoid non-canonical instances when both the left and right types are inhabited.

def

```
Sum.inhabitedRight.{u, v} {α : Type u} {β : Type v} [Inhabited β] :
  Inhabited (α ⊕ β)
```

If the right type in a sum is inhabited then the sum is inhabited.

This is not an instance to avoid non-canonical instances when both the left and right types are inhabited.

def

```
PSum.inhabitedLeft.{u_1, u_2} {α : Sort u_1} {β : Sort u_2}
  [Inhabited α] : Inhabited (α ⊕' β)
```

If the left type in a sum is inhabited then the sum is inhabited.

This is not an instance to avoid non-canonical instances when both the left and right types are inhabited.

def

```
PSum.inhabitedRight.{u_1, u_2} {α : Sort u_1} {β : Sort u_2}
  [Inhabited β] : Inhabited (α ⊕' β)
```

If the right type in a sum is inhabited then the sum is inhabited.

This is not an instance to avoid non-canonical instances when both the left and right types are inhabited.

19.15. Linked Lists

Linked lists, implemented as the inductive type `List`, contain an ordered sequence of elements. Unlike arrays, Lean compiles lists according to the ordinary rules for inductive types; however, some operations on lists are replaced by tail-recursive equivalents in compiled code using the **`csimp`** mechanism. Lean provides syntax for both literal lists and the constructor `List.cons`.

inductive type

```
List.{u} (α : Type u) : Type u
```

Linked lists: ordered lists, in which each element has a reference to the next element.

Most operations on linked lists take time proportional to the length of the list, because each element must be traversed to find the next element.

`List α` is isomorphic to `Array α`, but they are useful for different things:

- `List α` is easier for reasoning, and `Array α` is modeled as a wrapper around `List α`.
- `List α` works well as a persistent data structure, when many copies of the tail are shared. When the value is not shared, `Array α` will have better performance because it can do destructive updates.

Constructors

```
| nil.{u} {α : Type u} : List α
```

The empty list, usually written `[]`.

Conventions for notations in identifiers:

- The recommended spelling of `[]` in identifiers is `nil`.

```
cons.{u} {α : Type u} (head : α) (tail : List α) : List α
```

The list whose first element is `head`, where `tail` is the rest of the list. Usually written *head* `::` *tail*.

Conventions for notations in identifiers:

- The recommended spelling of `::` in identifiers is `cons`.
- The recommended spelling of `[a]` in identifiers is `singleton`.

19.15.1. Syntax

List literals are written in square brackets, with the elements of the list separated by commas. The constructor `List.cons` that adds an element to the front of a list is represented by the infix operator `::`. The syntax for lists can be used both in ordinary terms and in patterns.

syntax

List Literals

```
term ::= ...
    | [term,*]
```

The syntax `[a, b, c]` is shorthand for `a :: b :: c :: []`, or `List.cons a (List.cons b (List.cons c List.nil))`. It allows conveniently constructing list literals.

For lists of length at least 64, an alternative desugaring strategy is used which uses let bindings as intermediates as in `let left := [d, e, f]; a :: b :: c :: left` to avoid creating very deep expressions. Note that this changes the order of evaluation, although it should not be observable unless you use side effecting operations like **`dbg_trace`**.

Conventions for notations in identifiers:

- The recommended spelling of `[]` in identifiers is `nil`.
- The recommended spelling of `[a]` in identifiers is `singleton`.

syntax

List Construction

```
term ::= ...
    | term :: term
```

The list whose first element is *head*, where `tail` is the rest of the list. Usually written *head* `:: tail`.

Conventions for notations in identifiers:

- The recommended spelling of `::` in identifiers is `cons`.

▼ *Example: Constructing Lists*

All of these examples are equivalent:

```
example : List Nat := [1, 2, 3]
example : List Nat := 1 :: [2, 3]
example : List Nat := 1 :: 2 :: [3]
example : List Nat := 1 :: 2 :: 3 :: []
example : List Nat := 1 :: 2 :: 3 :: .nil
example : List Nat := 1 :: 2 :: .cons 3 .nil
example : List Nat := .cons 1 (.cons 2 (.cons 3 .nil))
```

▼ *Example: Pattern Matching and Lists*

All of these functions are equivalent:

```
def split : List α → List α × List α
  | [] => ([], [])
  | [x] => ([x], [])
  | x :: x' :: xs =>
    let (ys, zs) := split xs
    (x :: ys, x' :: zs)

def split' : List α → List α × List α
  | .nil => (.nil, .nil)
  | x :: [] => (.singleton x, .nil)
  | x :: x' :: xs =>
    let (ys, zs) := split xs
    (x :: ys, x' :: zs)

def split'' : List α → List α × List α
  | .nil => (.nil, .nil)
  | .cons x .nil=> (.singleton x, .nil)
  | .cons x (.cons x' xs) =>
    let (ys, zs) := split xs
    (.cons x ys, .cons x' zs)
```

19.15.2. Performance Notes

The representation of lists is not overridden or modified by the compiler: they are linked lists, with a pointer indirection for each element. Calculating the length of a list requires a full traversal, and modifying an element in a list requires a traversal and reallocation of the prefix of the list that is prior to the element being modified. Due to Lean's reference-counting-based memory management, operations such as `List.map` that traverse a list, allocating a new `List.cons` constructor for each in the prior list, can re-use the original list's memory when there are no other references to it.

Because of the important role played by lists in specifications, most list functions are written as straightforwardly as possible using structural recursion. This makes it easier to write proofs by induction, but it also means that these operations consume stack space proportional to the length of the list. There are tail-recursive versions of many list functions that are equivalent to the non-tail-recursive versions, but more are difficult to use when reasoning. In compiled code, the tail-recursive versions are automatically used instead of the non-tail-recursive versions.

19.15.3. API Reference

19.15.3.1. Predicates and Relations

def

```
List.IsPrefix.{u} {α : Type u} (l₁ l₂ : List α) : Prop
```

The first list is a prefix of the second.

`IsPrefix l₁ l₂`, written `l₁ <+: l₂`, means that there exists some `t : List α` such that `l₂` has the form `l₁ ++ t`.

The function `List.isPrefixOf` is a Boolean equivalent.

Conventions for notations in identifiers:

- The recommended spelling of `<+:` in identifiers is **`prefix`** (not `isPrefix`).

syntax

List Prefix

```
term ::= ...
    | term <+: term
```

The first list is a prefix of the second.

`IsPrefix l₁ l₂`, written `l₁ <+: l₂`, means that there exists some `t : List α` such that `l₂` has the form `l₁ ++ t`.

The function `List.isPrefixOf` is a Boolean equivalent.

Conventions for notations in identifiers:

- The recommended spelling of `<+:` in identifiers is **`prefix`** (not `isPrefix`).

def

`List.IsSuffix.{u} {α : Type u} (l₁ l₂ : List α) : Prop`

The first list is a suffix of the second.

`IsSuffix l₁ l₂`, written `l₁ <:+ l₂`, means that there exists some `t : List α` such that `l₂` has the form `t ++ l₁`.

The function `List.isSuffixOf` is a Boolean equivalent.

Conventions for notations in identifiers:

- The recommended spelling of `<:+` in identifiers is `suffix` (not `isSuffix`).

syntax

List Suffix

```
term ::= ...
    | term <:+ term
```

The first list is a suffix of the second.

`IsSuffix l₁ l₂`, written `l₁ <:+ l₂`, means that there exists some `t : List α` such that `l₂` has the form `t ++ l₁`.

The function `List.isSuffixOf` is a Boolean equivalent.

Conventions for notations in identifiers:

- The recommended spelling of `<:+` in identifiers is `suffix` (not `isSuffix`).

def

`List.IsInfix.{u} {α : Type u} (l₁ l₂ : List α) : Prop`

The first list is a contiguous sub-list of the second list. Typically written with the `<:+:` operator.

In other words, `l₁ <:+: l₂` means that there exist lists `s : List α` and `t : List α` such that `l₂` has the form `s ++ l₁ ++ t`.

Conventions for notations in identifiers:

- The recommended spelling of `<:+:` in identifiers is **`infix`** (not `isInfix`).

syntax

List Infix

```
term ::= ...
    | term <:+: term
```

The first list is a contiguous sub-list of the second list. Typically written with the `<:+:` operator.

In other words, l_1 `<:+:` l_2 means that there exist lists s `: List` α and t `: List` α such that l_2 has the form `s ++` l_1 `++ t`.

Conventions for notations in identifiers:

- The recommended spelling of `<:+:` in identifiers is **`infix`** (not `isInfix`).

inductive predicate

`List.Sublist.{u_1} {α : Type u_1} : List α → List α → Prop`

The first list is a non-contiguous sub-list of the second list. Typically written with the `<+` operator.

In other words, l_1 `<+` l_2 means that l_1 can be transformed into l_2 by repeatedly inserting new elements.

Constructors

| `slnil.{u_1} {α : Type u_1} : [].Sublist []`

The base case: `[]` is a sublist of `[]`

| `cons.{u_1} {α : Type u_1} {l₁ l₂ : List α} (a : α) : l₁.Sublist l₂ → l₁.Sublist (a :: l₂)`

If l_1 is a subsequence of l_2, then it is also a subsequence of a `::` l_2.

| `cons₂.{u_1} {α : Type u_1} {l₁ l₂ : List α} (a : α) : l₁.Sublist l₂ → (a :: l₁).Sublist (a :: l₂)`

If l_1 is a subsequence of l_2, then a `::` l_1 is a subsequence of a `::` l_2.

syntax

Sublists

```
term ::= ...
    | term <+ term
```

The first list is a non-contiguous sub-list of the second list. Typically written with the <+ operator.

In other words, `l₁ <+ l₂` means that `l₁` can be transformed into `l₂` by repeatedly inserting new elements.

This syntax is only available when the `List` namespace is opened.

inductive predicate

`List.Perm.{u} {α : Type u} : List α → List α → Prop`

Two lists are permutations of each other if they contain the same elements, each occurring the same number of times but not necessarily in the same order.

One list can be proven to be a permutation of another by showing how to transform one into the other by repeatedly swapping adjacent elements.

`List.isPerm` is a Boolean equivalent of this relation.

Constructors

| `nil.{u} {α : Type u} : [].Perm []`

  The empty list is a permutation of the empty list: `[] ~ []`.

| `cons.{u} {α : Type u} (x : α) {l₁ l₂ : List α} : l₁.Perm l₂ → (x :: l₁).Perm (x :: l₂)`

  If one list is a permutation of the other, adding the same element as the head of each yields lists that are permutations of each other: `l₁ ~ l₂ → x::l₁ ~ x::l₂`.

| `swap.{u} {α : Type u} (x y : α) (l : List α) : (y :: x :: l).Perm (x :: y :: l)`

  If two lists are identical except for having their first two elements swapped, then they are permutations of each other: `x::y::l ~ y::x::l`.

| `trans.{u} {α : Type u} {l₁ l₂ l₃ : List α} : l₁.Perm l₂ → l₂.Perm l₃ → l₁.Perm l₃`

  Permutation is transitive: `l₁ ~ l₂ → l₂ ~ l₃ → l₁ ~ l₃`.

syntax

List Permutation

term `::= ...`

```
  | term ~ term
```

Two lists are permutations of each other if they contain the same elements, each occurring the same number of times but not necessarily in the same order.

One list can be proven to be a permutation of another by showing how to transform one into the other by repeatedly swapping adjacent elements.

`List.isPerm` is a Boolean equivalent of this relation.

This syntax is only available when the `List` namespace is opened.

inductive predicate

```
List.Pairwise.{u} {α : Type u} (R : α → α → Prop) : List α → Prop
```

Each element of a list is related to all later elements of the list by *R*.

`Pairwise R l` means that all the elements of `l` with earlier indexes are *R*-related to all the elements with later indexes.

For example, `Pairwise (· ≠ ·) l` asserts that `l` has no duplicates, and if `Pairwise (· < ·) l` asserts that `l` is (strictly) sorted.

Examples:

- `Pairwise (· < ·) [1, 2, 3] ↔ (1 < 2 ∧ 1 < 3) ∧ 2 < 3`
- `Pairwise (· = ·) [1, 2, 3] = False`
- `Pairwise (· ≠ ·) [1, 2, 3] = True`

Constructors

```
| nil.{u} {α : Type u} {R : α → α → Prop} : List.Pairwise R []
```

All elements of the empty list are vacuously pairwise related.

```
| cons.{u} {α : Type u} {R : α → α → Prop} {a : α}
    {l : List α} :
    (∀ (a' : α), a' ∈ l → R a a') →
      List.Pairwise R l → List.Pairwise R (a :: l)
```

A nonempty list is pairwise related with *R* if the head is related to every element of the tail and the tail is itself pairwise related.

That is, `a :: l` is `Pairwise R` if:

- *R* relates *a* to every element of `l`

- l is `Pairwise` R.

def

`List.Nodup.{u} {α : Type u} : List α → Prop`

The list has no duplicates: it contains every element at most once.

It is defined as `Pairwise (· ≠ ·)`: each element is unequal to all other elements.

inductive predicate

`List.Lex.{u} {α : Type u} (r : α → α → Prop) (as bs : List α) : Prop`

Lexicographic ordering for lists with respect to an ordering of elements.

as is lexicographically smaller than bs if

- as is empty and bs is non-empty, or
- both as and bs are non-empty, and the head of as is less than the head of bs according to r, or
- both as and bs are non-empty, their heads are equal, and the tail of as is less than the tail of bs.

Constructors

| `nil.{u} {α : Type u} {r : α → α → Prop} {a : α} {l : List α} : List.Lex r [] (a :: l)`

`[]` is the smallest element in the lexicographic order.

| `rel.{u} {α : Type u} {r : α → α → Prop} {a₁ : α} {l₁ : List α} {a₂ : α} {l₂ : List α} (h : r a₁ a₂) : List.Lex r (a₁ :: l₁) (a₂ :: l₂)`

If the head of the first list is smaller than the head of the second, then the first list is lexicographically smaller than the second list.

| `cons.{u} {α : Type u} {r : α → α → Prop} {a : α} {l₁ l₂ : List α} (h : List.Lex r l₁ l₂) : List.Lex r (a :: l₁) (a :: l₂)`

If two lists have the same head, then their tails determine their lexicographic order. If the tail of the first list is lexicographically smaller than the tail of the second list, then the entire first list is lexicographically smaller than the entire second list.

inductive predicate

```
List.Mem.{u} {α : Type u} (a : α) : List α → Prop
```

List membership, typically accessed via the ∈ operator.

`a ∈ l` means that `a` is an element of the list `l`. Elements are compared according to Lean's logical equality.

The related function `List.elem` is a Boolean membership test that uses a `BEq α` instance.

Examples:

- `a ∈ [x, y, z] ↔ a = x ∨ a = y ∨ a = z`

Constructors

```
| head.{u} {α : Type u} {a : α} (as : List α) :
    List.Mem a (a :: as)
```

The head of a list is a member: `a ∈ a :: as`.

```
| tail.{u} {α : Type u} {a : α} (b : α) {as : List α} :
    List.Mem a as → List.Mem a (b :: as)
```

A member of the tail of a list is a member of the list: `a ∈ l → a ∈ b :: l`.

19.15.3.2. Constructing Lists

def

```
List.singleton.{u} {α : Type u} (a : α) : List α
```

Constructs a single-element list.

Examples:

- `List.singleton 5 = [5]`.
- `List.singleton "green" = ["green"]`.
- `List.singleton [1, 2, 3] = [[1, 2, 3]]`

def

```
List.concat.{u} {α : Type u} : List α → α → List α
```

Adds an element to the *end* of a list.

The added element is the last element of the resulting list.

Examples:

- `List.concat ["red", "yellow"] "green" = ["red", "yellow", "green"]`
- `List.concat [1, 2, 3] 4 = [1, 2, 3, 4]`
- `List.concat [] () = [()]`

def

```
List.replicate.{u} {α : Type u} (n : Nat) (a : α) : List α
```

Creates a list that contains *n* copies of *a*.

- `List.replicate 5 "five" = ["five", "five", "five", "five", "five"]`
- `List.replicate 0 "zero" = []`
- `List.replicate 2 ' ' = [' ', ' ']`

def

```
List.replicateTR.{u} {α : Type u} (n : Nat) (a : α) : List α
```

Creates a list that contains *n* copies of *a*.

This is a tail-recursive version of `List.replicate`.

- `List.replicateTR 5 "five" = ["five", "five", "five", "five", "five"]`
- `List.replicateTR 0 "zero" = []`
- `List.replicateTR 2 ' ' = [' ', ' ']`

def

```
List.ofFn.{u_1} {α : Type u_1} {n : Nat} (f : Fin n → α) : List α
```

Creates a list by applying *f* to each potential index in order, starting at `0`.

Examples:

- `List.ofFn (n := 3) toString = ["0", "1", "2"]`
- `List.ofFn (fun i => #["red", "green", "blue"].get i.val i.isLt) = ["red", "green", "blue"]`

def

```
List.append.{u} {α : Type u} (xs ys : List α) : List α
```

Appends two lists. Normally used via the ++ operator.

Appending lists takes time proportional to the length of the first list: `O(|xs|)`.

Examples:

- `[1, 2, 3] ++ [4, 5] = [1, 2, 3, 4, 5]`.
- `[] ++ [4, 5] = [4, 5]`.
- `[1, 2, 3] ++ [] = [1, 2, 3]`.

def

```
List.appendTR.{u} {α : Type u} (as bs : List α) : List α
```

Appends two lists. Normally used via the ++ operator.

Appending lists takes time proportional to the length of the first list: `O(|xs|)`.

This is a tail-recursive version of `List.append`.

Examples:

- `[1, 2, 3] ++ [4, 5] = [1, 2, 3, 4, 5]`.
- `[] ++ [4, 5] = [4, 5]`.
- `[1, 2, 3] ++ [] = [1, 2, 3]`.

def

```
List.range (n : Nat) : List Nat
```

Returns a list of the numbers from `0` to *n* exclusive, in increasing order.

`O(n)`.

Examples:

- `range 5 = [0, 1, 2, 3, 4]`
- `range 0 = []`
- `range 2 = [0, 1]`

def

```
List.range' (start len : Nat) (step : Nat := 1) : List Nat
```

Returns a list of the numbers with the given length *len*, starting at *start* and increasing by *step* at each element.

In other words, `List.range' start len step` is `[start, start+step, ..., start+(len-1)*step]`.

Examples:

- `List.range' 0 3 (step := 1) = [0, 1, 2]`
- `List.range' 0 3 (step := 2) = [0, 2, 4]`
- `List.range' 0 4 (step := 2) = [0, 2, 4, 6]`
- `List.range' 3 4 (step := 2) = [3, 5, 7, 9]`

def

```
List.range'TR (s n : Nat) (step : Nat := 1) : List Nat
```

Returns a list of the numbers with the given length `len`, starting at `start` and increasing by `step` at each element.

In other words, `List.range'TR start len step` is `[start, start+step, ..., start+(len-1)*step]`.

This is a tail-recursive version of `List.range'`.

Examples:

- `List.range'TR 0 3 (step := 1) = [0, 1, 2]`
- `List.range'TR 0 3 (step := 2) = [0, 2, 4]`
- `List.range'TR 0 4 (step := 2) = [0, 2, 4, 6]`
- `List.range'TR 3 4 (step := 2) = [3, 5, 7, 9]`

def

```
List.finRange (n : Nat) : List (Fin n)
```

Lists all elements of `Fin n` in order, starting at `0`.

Examples:

- `List.finRange 0 = ([] : List (Fin 0))`
- `List.finRange 2 = ([0, 1] : List (Fin 2))`

19.15.3.3. Length

def

```
List.length.{u_1} {α : Type u_1} : List α → Nat
```

The length of a list.

This function is overridden in the compiler to `lengthTR`, which uses constant stack space.

Examples:

- `([] : List String).length = 0`
- `["green", "brown"].length = 2`

def

`List.lengthTR.{u_1} {α : Type u_1} (as : List α) : Nat`

The length of a list.

This is a tail-recursive version of `List.length`, used to implement `List.length` without running out of stack space.

Examples:

- `([] : List String).lengthTR = 0`
- `["green", "brown"].lengthTR = 2`

def

`List.isEmpty.{u} {α : Type u} : List α → Bool`

Checks whether a list is empty.

`O(1)`.

Examples:

- `[].isEmpty = true`
- `["grape"].isEmpty = false`
- `["apple", "banana"].isEmpty = false`

19.15.3.4. Head and Tail

def

`List.head.{u} {α : Type u} (as : List α) : as ≠ [] → α`

Returns the first element of a non-empty list.

def

```
List.head?.{u} {α : Type u} : List α → Option α
```

Returns the first element in the list, if there is one. Returns `none` if the list is empty.

Use `List.headD` to provide a fallback value for empty lists, or `List.head!` to panic on empty lists.

Examples:

- `([] : List Nat).head? = none`
- `[3, 2, 1].head? = some 3`

def

```
List.headD.{u} {α : Type u} (as : List α) (fallback : α) : α
```

Returns the first element in the list if there is one, or *`fallback`* if the list is empty.

Use `List.head?` to return an `Option`, and `List.head!` to panic on empty lists.

Examples:

- `[].headD "empty" = "empty"`
- `[].headD 2 = 2`
- `["head", "shoulders", "knees"].headD "toes" = "head"`

def

```
List.head!.{u_1} {α : Type u_1} [Inhabited α] : List α → α
```

Returns the first element in the list. If the list is empty, panics and returns `default`.

Safer alternatives include:

- `List.head`, which requires a proof that the list is non-empty,
- `List.head?`, which returns an `Option`, and
- `List.headD`, which returns an explicitly-provided fallback value on empty lists.

def

```
List.tail.{u} {α : Type u} : List α → List α
```

Drops the first element of a nonempty list, returning the tail. Returns `[]` when the argument is empty.

Examples:

- `["apple", "banana", "grape"].tail = ["banana", "grape"]`
- `["apple"].tail = []`
- `([] : List String).tail = []`

def

`List.tail!.{u_1} {α : Type u_1} : List α → List α`

Drops the first element of a nonempty list, returning the tail. If the list is empty, this function panics when executed and returns the empty list.

Safer alternatives include

- `tail`, which returns the empty list without panicking,
- `tail?`, which returns an `Option`, and
- `tailD`, which returns a fallback value when passed the empty list.

Examples:

- `["apple", "banana", "grape"].tail! = ["banana", "grape"]`
- `["banana", "grape"].tail! = ["grape"]`

def

`List.tail?.{u} {α : Type u} : List α → Option (List α)`

Drops the first element of a nonempty list, returning the tail. Returns `none` when the argument is empty.

Alternatives include `List.tail`, which returns the empty list on failure, `List.tailD`, which returns an explicit fallback value, and `List.tail!`, which panics on the empty list.

Examples:

- `["apple", "banana", "grape"].tail? = some ["banana", "grape"]`
- `["apple"].tail? = some []`
- `([] : List String).tail = none`

def

`List.tailD.{u} {α : Type u} (l fallback : List α) : List α`

Drops the first element of a nonempty list, returning the tail. Returns `none` when the argument is empty.

Alternatives include `List.tail`, which returns the empty list on failure, `List.tail?`, which returns an `Option`, and `List.tail!`, which panics on the empty list.

Examples:

- `["apple", "banana", "grape"].tailD ["orange"] = ["banana", "grape"]`
- `["apple"].tailD ["orange"] = []`
- `[].tailD ["orange"] = ["orange"]`

19.15.3.5. Lookups

def

```
List.get.{u} {α : Type u} (as : List α) : Fin as.length → α
```

Returns the element at the provided index, counting from `0`.

In other words, for `i : Fin as.length`, `as.get i` returns the `i`'th element of the list `as`. Because the index is a `Fin` bounded by the list's length, the index will never be out of bounds.

Examples:

- `["spring", "summer", "fall", "winter"].get (2 : Fin 4) = "fall"`
- `["spring", "summer", "fall", "winter"].get (0 : Fin 4) = "spring"`

def

```
List.getD.{u_1} {α : Type u_1} (as : List α) (i : Nat) (fallback : α) :
  α
```

Returns the element at the provided index, counting from `0`. Returns `fallback` if the index is out of bounds.

To return an `Option` depending on whether the index is in bounds, use `as[i]?`. To panic if the index is out of bounds, use `as[i]!`.

Examples:

- `["spring", "summer", "fall", "winter"].getD 2 "never" = "fall"`
- `["spring", "summer", "fall", "winter"].getD 0 "never" = "spring"`

- `["spring", "summer", "fall", "winter"].getD 4 "never" = "never"`

def

`List.getLast.{u} {α : Type u} (as : List α) : as ≠ [] → α`

Returns the last element of a non-empty list.

Examples:

- `["circle", "rectangle"].getLast (by decide) = "rectangle"`
- `["circle"].getLast (by decide) = "circle"`

def

`List.getLast?.{u} {α : Type u} : List α → Option α`

Returns the last element in the list, or `none` if the list is empty.

Alternatives include `List.getLastD`, which takes a fallback value for empty lists, and `List.getLast!`, which panics on empty lists.

Examples:

- `["circle", "rectangle"].getLast? = some "rectangle"`
- `["circle"].getLast? = some "circle"`
- `([] : List String).getLast? = none`

def

`List.getLastD.{u} {α : Type u} (as : List α) (fallback : α) : α`

Returns the last element in the list, or *fallback* if the list is empty.

Alternatives include `List.getLast?`, which returns an `Option`, and `List.getLast!`, which panics on empty lists.

Examples:

- `["circle", "rectangle"].getLastD "oval" = "rectangle"`
- `["circle"].getLastD "oval" = "circle"`
- `([] : List String).getLastD "oval" = "oval"`

def

`List.getLast!.{u_1} {α : Type u_1} [Inhabited α] : List α → α`

Returns the last element in the list. Panics and returns `default` if the list is empty.

Safer alternatives include:

- `getLast?`, which returns an `Option`,
- `getLastD`, which takes a fallback value for empty lists, and
- `getLast`, which requires a proof that the list is non-empty.

Examples:

- `["circle", "rectangle"].getLast! = "rectangle"`
- `["circle"].getLast! = "circle"`

def

```
List.lookup.{u, v} {α : Type u} {β : Type v} [BEq α] :
  α → List (α × β) → Option β
```

Treats the list as an association list that maps keys to values, returning the first value whose key is equal to the specified key.

O(|l|).

Examples:

- `[(1, "one"), (3, "three"), (3, "other")].lookup 3 = some "three"`
- `[(1, "one"), (3, "three"), (3, "other")].lookup 2 = none`

def

```
List.max?.{u} {α : Type u} [Max α] : List α → Option α
```

Returns the largest element of the list if it is not empty, or `none` if it is empty.

Examples:

- `[].max? = none`
- `[4].max? = some 4`
- `[1, 4, 2, 10, 6].max? = some 10`

def

```
List.min?.{u} {α : Type u} [Min α] : List α → Option α
```

Returns the smallest element of the list if it is not empty, or `none` if it is empty.

Examples:

- `[].min? = none`
- `[4].min? = some 4`
- `[1, 4, 2, 10, 6].min? = some 1`

19.15.3.6. Queries

def

`List.count.{u} {α : Type u} [BEq α] (a : α) : List α → Nat`

Counts the number of times an element occurs in a list.

Examples:

- `[1, 1, 2, 3, 5].count 1 = 2`
- `[1, 1, 2, 3, 5].count 5 = 1`
- `[1, 1, 2, 3, 5].count 4 = 0`

def

`List.countP.{u} {α : Type u} (p : α → Bool) (l : List α) : Nat`

Counts the number of elements in the list l that satisfy the Boolean predicate p.

Examples:

- `[1, 2, 3, 4, 5].countP (· % 2 == 0) = 2`
- `[1, 2, 3, 4, 5].countP (· < 5) = 4`
- `[1, 2, 3, 4, 5].countP (· > 5) = 0`

def

`List.idxOf.{u} {α : Type u} [BEq α] (a : α) : List α → Nat`

Returns the index of the first element equal to a, or the length of the list if no element is equal to a.

Examples:

- `["carrot", "potato", "broccoli"].idxOf "carrot" = 0`
- `["carrot", "potato", "broccoli"].idxOf "broccoli" = 2`
- `["carrot", "potato", "broccoli"].idxOf "tomato" = 3`

- `["carrot", "potato", "broccoli"].idxOf "anything else" = 3`

def

```
List.idxOf?.{u} {α : Type u} [BEq α] (a : α) : List α → Option Nat
```

Returns the index of the first element equal to *a*, or `none` if no element is equal to *a*.

Examples:

- `["carrot", "potato", "broccoli"].idxOf? "carrot" = some 0`
- `["carrot", "potato", "broccoli"].idxOf? "broccoli" = some 2`
- `["carrot", "potato", "broccoli"].idxOf? "tomato" = none`
- `["carrot", "potato", "broccoli"].idxOf? "anything else" = none`

def

```
List.finIdxOf?.{u} {α : Type u} [BEq α] (a : α) (l : List α) :
  Option (Fin l.length)
```

Returns the index of the first element equal to *a*, or the length of the list if no element is equal to *a*. The index is returned as a `Fin`, which guarantees that it is in bounds.

Examples:

- `["carrot", "potato", "broccoli"].finIdxOf? "carrot" = some 0`
- `["carrot", "potato", "broccoli"].finIdxOf? "broccoli" = some 2`
- `["carrot", "potato", "broccoli"].finIdxOf? "tomato" = none`
- `["carrot", "potato", "broccoli"].finIdxOf? "anything else" = none`

def

```
List.find?.{u} {α : Type u} (p : α → Bool) : List α → Option α
```

Returns the first element of the list for which the predicate *p* returns `true`, or `none` if no such element is found.

O(|l|).

Examples:

- `[7, 6, 5, 8, 1, 2, 6].find? (· < 5) = some 1`
- `[7, 6, 5, 8, 1, 2, 6].find? (· < 1) = none`

def

```
List.findFinIdx?.{u} {α : Type u} (p : α → Bool) (l : List α) :
  Option (Fin l.length)
```

Returns the index of the first element for which *p* returns `true`, or `none` if there is no such element. The index is returned as a `Fin`, which guarantees that it is in bounds.

Examples:

- `[7, 6, 5, 8, 1, 2, 6].findFinIdx? (· < 5) = some (4 : Fin 7)`
- `[7, 6, 5, 8, 1, 2, 6].findFinIdx? (· < 1) = none`

def

```
List.findIdx.{u} {α : Type u} (p : α → Bool) (l : List α) : Nat
```

Returns the index of the first element for which *p* returns `true`, or the length of the list if there is no such element.

Examples:

- `[7, 6, 5, 8, 1, 2, 6].findIdx (· < 5) = 4`
- `[7, 6, 5, 8, 1, 2, 6].findIdx (· < 1) = 7`

def

```
List.findIdx?.{u} {α : Type u} (p : α → Bool) (l : List α) : Option
Nat
```

Returns the index of the first element for which *p* returns `true`, or `none` if there is no such element.

Examples:

- `[7, 6, 5, 8, 1, 2, 6].findIdx (· < 5) = some 4`
- `[7, 6, 5, 8, 1, 2, 6].findIdx (· < 1) = none`

def

```
List.findM?.{u} {m : Type → Type u} [Monad m] {α : Type}
  (p : α → m Bool) : List α → m (Option α)
```

Returns the first element of the list for which the monadic predicate *p* returns `true`, or `none` if no such element is found. Elements of the list are checked in order.

`O(|l|)`.

Example:

```
#eval [7, 6, 5, 8, 1, 2, 6].findM? fun i => do
  if i < 5 then
    return true
  if i ≤ 6 then
    IO.println s!"Almost! {i}"
  return false

Almost! 6
Almost! 5

some 1
```

def

```
List.findSome?.{u, v} {α : Type u} {β : Type v} (f : α → Option β)
:
  List α → Option β
```

Returns the first non-`none` result of applying `f` to each element of the list in order. Returns `none` if `f` returns `none` for all elements of the list.

O(|l|).

Examples:

- `[7, 6, 5, 8, 1, 2, 6].findSome? (fun x => if x < 5 then some (10 * x) else none) = some 10`
- `[7, 6, 5, 8, 1, 2, 6].findSome? (fun x => if x < 1 then some (10 * x) else none) = none`

def

```
List.findSomeM?.{u, v, w} {m : Type u → Type v} [Monad m] {α : Type w}
  {β : Type u} (f : α → m (Option β)) : List α → m (Option β)
```

Returns the first non-`none` result of applying the monadic function `f` to each element of the list, in order. Returns `none` if `f` returns `none` for all elements.

O(|l|).

Example:

```
#eval [7, 6, 5, 8, 1, 2, 6].findSomeM? fun i => do
  if i < 5 then
    return some (i * 10)
  if i ≤ 6 then
    IO.println s!"Almost! {i}"
```

```
  return none

Almost! 6
Almost! 5

some 10
```

19.15.3.7. Conversions

def

```
List.asString (s : List Char) : String
```

Creates a string that contains the characters in a list, in order.

Examples:

- `['L', 'Ǝ', '∀', 'N'].asString = "LƎ∀N"`
- `[].asString = ""`
- `['a', 'a', 'a'].asString = "aaa"`

def

```
List.toArray.{u_1} {α : Type u_1} (xs : List α) : Array α
```

Converts a `List α` into an `Array α`.

$O(|xs|)$. At runtime, this operation is implemented by `List.toArrayImpl` and takes time linear in the length of the list. `List.toArray` should be used instead of `Array.mk`.

Examples:

- `[1, 2, 3].toArray = #[1, 2, 3]`
- `["monday", "wednesday", friday"].toArray = #["monday", "wednesday", friday"].`

def

```
List.toArrayImpl.{u_1} {α : Type u_1} (xs : List α) : Array α
```

Converts a `List α` into an `Array α` by repeatedly pushing elements from the list onto an empty array. $O(|xs|)$.

Use `List.toArray` instead of calling this function directly. At runtime, this operation implements both `List.toArray` and `Array.mk`.

def

```
List.toByteArray (bs : List UInt8) : ByteArray
```

Converts a list of bytes into a `ByteArray`.

def

```
List.toFloatArray (ds : List Float) : FloatArray
```

Converts a list of floats into a `FloatArray`.

def

```
List.toString.{u_1} {α : Type u_1} [ToString α] : List α → String
```

Converts a list into a string, using `ToString.toString` to convert its elements.

The resulting string resembles list literal syntax, with the elements separated by `", "` and enclosed in square brackets.

The resulting string may not be valid Lean syntax, because there's no such expectation for `ToString` instances.

Examples:

- `[1, 2, 3].toString = "[1, 2, 3]"`
- `["cat", "dog"].toString = "[cat, dog]"`
- `["cat", "dog", ""].toString = "[cat, dog, ]"`

19.15.3.8. Modification

def

```
List.set.{u_1} {α : Type u_1} (l : List α) (n : Nat) (a : α) : List α
```

Replaces the value at (zero-based) index *n* in `l` with *a*. If the index is out of bounds, then the list is returned unmodified.

Examples:

- `["water", "coffee", "soda", "juice"].set 1 "tea" = ["water", "tea", "soda", "juice"]`
- `["water", "coffee", "soda", "juice"].set 4 "tea" = ["water", "coffee", "soda", "juice"]`

def

```
List.setTR.{u_1} {α : Type u_1} (l : List α) (n : Nat) (a : α) :
List α
```

Replaces the value at (zero-based) index *n* in `l` with *a*. If the index is out of bounds, then the list is returned unmodified.

This is a tail-recursive version of `List.set` that's used at runtime.

Examples:

- `["water", "coffee", "soda", "juice"].set 1 "tea" = ["water", "tea", "soda", "juice"]`
- `["water", "coffee", "soda", "juice"].set 4 "tea" = ["water", "coffee", "soda", "juice"]`

def

```
List.modify.{u} {α : Type u} (l : List α) (i : Nat) (f : α → α) :
List α
```

Replaces the element at the given index, if it exists, with the result of applying *f* to it. If the index is invalid, the list is returned unmodified.

Examples:

- `[1, 2, 3].modify 0 (· * 10) = [10, 2, 3]`
- `[1, 2, 3].modify 2 (· * 10) = [1, 2, 30]`
- `[1, 2, 3].modify 3 (· * 10) = [1, 2, 3]`

def

```
List.modifyTR.{u_1} {α : Type u_1} (l : List α) (i : Nat) (f : α →
α) :
  List α
```

Replaces the element at the given index, if it exists, with the result of applying *f* to it.

This is a tail-recursive version of `List.modify`.

Examples:

- `[1, 2, 3].modifyTR 0 (· * 10) = [10, 2, 3]`
- `[1, 2, 3].modifyTR 2 (· * 10) = [1, 2, 30]`
- `[1, 2, 3].modifyTR 3 (· * 10) = [1, 2, 3]`

def

```
List.modifyHead.{u} {α : Type u} (f : α → α) : List α → List α
```

Replace the head of the list with the result of applying f to it. Returns the empty list if the list is empty.

Examples:

- `[1, 2, 3].modifyHead (· * 10) = [10, 2, 3]`
- `[].modifyHead (· * 10) = []`

def

```
List.modifyTailIdx.{u} {α : Type u} (l : List α) (i : Nat)
  (f : List α → List α) : List α
```

Replaces the nth tail of l with the result of applying f to it. Returns the input without using f if the index is larger than the length of the List.

Examples:

```
["circle", "square", "triangle"].modifyTailIdx 1 List.reverse
```

```
["circle", "triangle", "square"]
```

```
["circle", "square", "triangle"].modifyTailIdx 1 (fun xs => xs ++ xs)
```

```
["circle", "square", "triangle", "square", "triangle"]
```

```
["circle", "square", "triangle"].modifyTailIdx 2 (fun xs => xs ++ xs)
```

```
["circle", "square", "triangle", "triangle"]
```

```
["circle", "square", "triangle"].modifyTailIdx 5 (fun xs => xs ++ xs)
```

```
["circle", "square", "triangle"]
```

def

```
List.erase.{u_1} {α : Type u_1} [BEq α] : List α → α → List α
```

Removes the first occurrence of a from `l`. If a does not occur in `l`, the list is returned unmodified.

`O(|l|)`.

Examples:

- `[1, 5, 3, 2, 5].erase 5 = [1, 3, 2, 5]`
- `[1, 5, 3, 2, 5].erase 6 = [1, 5, 3, 2, 5]`

def

`List.eraseTR.{u_1} {α : Type u_1} [BEq α] (l : List α) (a : α) : List α`

Removes the first occurrence of a from l. If a does not occur in l, the list is returned unmodified.

`O(|l|)`.

This is a tail-recursive version of `List.erase`, used in runtime code.

Examples:

- `[1, 5, 3, 2, 5].eraseTR 5 = [1, 3, 2, 5]`
- `[1, 5, 3, 2, 5].eraseTR 6 = [1, 5, 3, 2, 5]`

def

`List.eraseDups.{u_1} {α : Type u_1} [BEq α] (as : List α) : List α`

Erases duplicated elements in the list, keeping the first occurrence of duplicated elements.

`O(|l|^2)`.

Examples:

- `[1, 3, 2, 2, 3, 5].eraseDups = [1, 3, 2, 5]`
- `["red", "green", "green", "blue"].eraseDups = ["red", "green", "blue"]`

def

`List.eraseIdx.{u} {α : Type u} (l : List α) (i : Nat) : List α`

Removes the element at the specified index. If the index is out of bounds, the list is returned unmodified.

`O(i)`.

Examples:

- `[0, 1, 2, 3, 4].eraseIdx 0 = [1, 2, 3, 4]`
- `[0, 1, 2, 3, 4].eraseIdx 1 = [0, 2, 3, 4]`

- `[0, 1, 2, 3, 4].eraseIdx 5 = [0, 1, 2, 3, 4]`

def

`List.eraseIdxTR.{u_1} {α : Type u_1} (l : List α) (n : Nat) : List α`

Removes the element at the specified index. If the index is out of bounds, the list is returned unmodified.

`O(i)`.

This is a tail-recursive version of `List.eraseIdx`, used at runtime.

Examples:

- `[0, 1, 2, 3, 4].eraseIdxTR 0 = [1, 2, 3, 4]`
- `[0, 1, 2, 3, 4].eraseIdxTR 1 = [0, 2, 3, 4]`
- `[0, 1, 2, 3, 4].eraseIdxTR 5 = [0, 1, 2, 3, 4]`

def

`List.eraseP.{u} {α : Type u} (p : α → Bool) : List α → List α`

Removes the first element of a list for which *p* returns `true`. If no element satisfies *p*, then the list is returned unchanged.

Examples:

- `[2, 1, 2, 1, 3, 4].eraseP (· < 2) = [2, 2, 1, 3, 4]`
- `[2, 1, 2, 1, 3, 4].eraseP (· > 2) = [2, 1, 2, 1, 4]`
- `[2, 1, 2, 1, 3, 4].eraseP (· > 8) = [2, 1, 2, 1, 3, 4]`

def

`List.erasePTR.{u_1} {α : Type u_1} (p : α → Bool) (l : List α) : List α`

Removes the first element of a list for which *p* returns `true`. If no element satisfies *p*, then the list is returned unchanged.

This is a tail-recursive version of `eraseP`, used at runtime.

Examples:

- `[2, 1, 2, 1, 3, 4].erasePTR (· < 2) = [2, 2, 1, 3, 4]`
- `[2, 1, 2, 1, 3, 4].erasePTR (· > 2) = [2, 1, 2, 1, 4]`

- `[2, 1, 2, 1, 3, 4].erasePTR (· > 8) = [2, 1, 2, 1, 3, 4]`

def

```
List.eraseReps.{u_1} {α : Type u_1} [BEq α] (as : List α) : List α
```

Erases repeated elements, keeping the first element of each run.

`O(|l|)`.

Example:

- `[1, 3, 2, 2, 2, 3, 3, 5].eraseReps = [1, 3, 2, 3, 5]`

def

```
List.extract.{u} {α : Type u} (l : List α) (start : Nat := 0)
  (stop : Nat := l.length) : List α
```

Returns the slice of `l` from indices `start` (inclusive) to `stop` (exclusive).

Examples:

- [0, 1, 2, 3, 4, 5].extract 1 2 = [1]
- [0, 1, 2, 3, 4, 5].extract 2 2 = []
- [0, 1, 2, 3, 4, 5].extract 2 4 = [2, 3]
- [0, 1, 2, 3, 4, 5].extract 2 = [2, 3, 4, 5]
- [0, 1, 2, 3, 4, 5].extract (stop := 2) = [0, 1]

def

```
List.removeAll.{u} {α : Type u} [BEq α] (xs ys : List α) : List α
```

Removes all elements of `xs` that are present in `ys`.

`O(|xs| * |ys|)`.

Examples:

- `[1, 1, 5, 1, 2, 4, 5].removeAll [1, 2, 2] = [5, 4, 5]`
- `[1, 2, 3, 2].removeAll [] = [1, 2, 3, 2]`
- `[1, 2, 3, 2].removeAll [3] = [1, 2, 2]`

def

```
List.replace.{u} {α : Type u} [BEq α] (l : List α) (a b : α) : List
α
```

Replaces the first element of the list `l` that is equal to *a* with *b*. If no element is equal to *a*, then the list is returned unchanged.

`O(|l|)`.

Examples:

- `[1, 4, 2, 3, 3, 7].replace 3 6 = [1, 4, 2, 6, 3, 7]`
- `[1, 4, 2, 3, 3, 7].replace 5 6 = [1, 4, 2, 3, 3, 7]`

def

```
List.replaceTR.{u_1} {α : Type u_1} [BEq α] (l : List α) (b c : α)
:
  List α
```

Replaces the first element of the list `l` that is equal to a with *b*. If no element is equal to a, then the list is returned unchanged.

`O(|l|)`. This is a tail-recursive version of `List.replace` that's used in runtime code.

Examples:

- `[1, 4, 2, 3, 3, 7].replaceTR 3 6 = [1, 4, 2, 6, 3, 7]`
- `[1, 4, 2, 3, 3, 7].replaceTR 5 6 = [1, 4, 2, 3, 3, 7]`

def

```
List.reverse.{u} {α : Type u} (as : List α) : List α
```

Reverses a list.

`O(|as|)`.

Because of the “functional but in place” optimization implemented by Lean's compiler, this function does not allocate a new list when its reference to the input list is unshared: it simply walks the linked list and reverses all the node pointers.

Examples:

- `[1, 2, 3, 4].reverse = [4, 3, 2, 1]`
- `[].reverse = []`

def

```
List.flatten.{u} {α : Type u} : List (List α) → List α
```

Concatenates a list of lists into a single list, preserving the order of the elements.

`O(|flatten L|)`.

Examples:

- `[["a"], ["b", "c"]].flatten = ["a", "b", "c"]`
- `[["a"], [], ["b", "c"], ["d", "e", "f"]].flatten = ["a", "b", "c", "d", "e", "f"]`

def

```
List.flattenTR.{u_1} {α : Type u_1} (l : List (List α)) : List α
```

Concatenates a list of lists into a single list, preserving the order of the elements.

`O(|flatten L|)`. This is a tail-recursive version of `List.flatten`, used in runtime code.

Examples:

- `[["a"], ["b", "c"]].flattenTR = ["a", "b", "c"]`
- `[["a"], [], ["b", "c"], ["d", "e", "f"]].flattenTR = ["a", "b", "c", "d", "e", "f"]`

def

```
List.rotateLeft.{u} {α : Type u} (xs : List α) (i : Nat := 1) :
List α
```

Rotates the elements of `xs` to the left, moving `i % xs.length` elements from the start of the list to the end.

`O(|xs|)`.

Examples:

- `[1, 2, 3, 4, 5].rotateLeft 3 = [4, 5, 1, 2, 3]`
- `[1, 2, 3, 4, 5].rotateLeft 5 = [1, 2, 3, 4, 5]`
- `[1, 2, 3, 4, 5].rotateLeft 1 = [2, 3, 4, 5, 1]`

def

```
List.rotateRight.{u} {α : Type u} (xs : List α) (i : Nat := 1) :
List α
```

Rotates the elements of `xs` to the right, moving `i % xs.length` elements from the end of the list to the start.

After rotation, the element at `xs[n]` is at index `(i + n) % l.length`. O(|xs|).

Examples:

- `[1, 2, 3, 4, 5].rotateRight 3 = [3, 4, 5, 1, 2]`
- `[1, 2, 3, 4, 5].rotateRight 5 = [1, 2, 3, 4, 5]`
- `[1, 2, 3, 4, 5].rotateRight 1 = [5, 1, 2, 3, 4]`

def

```
List.leftpad.{u} {α : Type u} (n : Nat) (a : α) (l : List α) : List α
```

Pads `l : List α` on the left with repeated occurrences of `a : α` until it is of length `n`. If `l` already has at least `n` elements, it is returned unmodified.

Examples:

- `[1, 2, 3].leftpad 5 0 = [0, 0, 1, 2, 3]`
- `["red", "green", "blue"].leftpad 4 "blank" = ["blank", "red", "green", "blue"]`
- `["red", "green", "blue"].leftpad 3 "blank" = ["red", "green", "blue"]`
- `["red", "green", "blue"].leftpad 1 "blank" = ["red", "green", "blue"]`

def

```
List.leftpadTR.{u} {α : Type u} (n : Nat) (a : α) (l : List α) : List α
```

Pads `l : List α` on the left with repeated occurrences of `a : α` until it is of length `n`. If `l` already has at least `n` elements, it is returned unmodified.

This is a tail-recursive version of `List.leftpad`, used at runtime.

Examples:

- `[1, 2, 3].leftPadTR 5 0 = [0, 0, 1, 2, 3]`
- `["red", "green", "blue"].leftPadTR 4 "blank" = ["blank", "red", "green", "blue"]`

- `["red", "green", "blue"].leftPadTR 3 "blank" = ["red", "green", "blue"]`
- `["red", "green", "blue"].leftPadTR 1 "blank" = ["red", "green", "blue"]`

def

```
List.rightpad.{u} {α : Type u} (n : Nat) (a : α) (l : List α) : List α
```

Pads `l : List α` on the right with repeated occurrences of `a : α` until it is of length `n`. If `l` already has at least `n` elements, it is returned unmodified.

Examples:

- `[1, 2, 3].rightpad 5 0 = [1, 2, 3, 0, 0]`
- `["red", "green", "blue"].rightpad 4 "blank" = ["red", "green", "blue", "blank"]`
- `["red", "green", "blue"].rightpad 3 "blank" = ["red", "green", "blue"]`
- `["red", "green", "blue"].rightpad 1 "blank" = ["red", "green", "blue"]`

19.15.3.8.1. Insertion

def

```
List.insert.{u} {α : Type u} [BEq α] (a : α) (l : List α) : List α
```

Inserts an element into a list without duplication.

If the element is present in the list, the list is returned unmodified. Otherwise, the new element is inserted at the head of the list.

Examples:

- `[1, 2, 3].insert 0 = [0, 1, 2, 3]`
- `[1, 2, 3].insert 4 = [4, 1, 2, 3]`
- `[1, 2, 3].insert 2 = [1, 2, 3]`

def

```
List.insertIdx.{u} {α : Type u} (xs : List α) (i : Nat) (a : α) : List α
```

Inserts an element into a list at the specified index. If the index is greater than the length of the list, then the list is returned unmodified.

In other words, the new element is inserted into the list `l` after the first i elements of `l`.

Examples:

- `["tues", "thur", "sat"].insertIdx 1 "wed" = ["tues", "wed", "thur", "sat"]`
- `["tues", "thur", "sat"].insertIdx 2 "wed" = ["tues", "thur", "wed", "sat"]`
- `["tues", "thur", "sat"].insertIdx 3 "wed" = ["tues", "thur", "sat", "wed"]`
- `["tues", "thur", "sat"].insertIdx 4 "wed" = ["tues", "thur", "sat"]`

def

```
List.insertIdxTR.{u_1} {α : Type u_1} (l : List α) (n : Nat) (a : α) :
  List α
```

Inserts an element into a list at the specified index. If the index is greater than the length of the list, then the list is returned unmodified.

In other words, the new element is inserted into the list l after the first `i` elements of l.

This is a tail-recursive version of `List.insertIdx`, used at runtime.

Examples:

- `["tues", "thur", "sat"].insertIdxTR 1 "wed" = ["tues", "wed", "thur", "sat"]`
- `["tues", "thur", "sat"].insertIdxTR 2 "wed" = ["tues", "thur", "wed", "sat"]`
- `["tues", "thur", "sat"].insertIdxTR 3 "wed" = ["tues", "thur", "sat", "wed"]`
- `["tues", "thur", "sat"].insertIdxTR 4 "wed" = ["tues", "thur", "sat"]`

def

```
List.intersperse.{u} {α : Type u} (sep : α) (l : List α) : List α
```

Alternates the elements of l with sep.

O(|l|).

List.intercalate is a similar function that alternates a separator list with elements of a list of lists.

Examples:

- List.intersperse "then" [] = []
- List.intersperse "then" ["walk"] = ["walk"]
- List.intersperse "then" ["walk", "run"] = ["walk", "then", "run"]
- List.intersperse "then" ["walk", "run", "rest"] = ["walk", "then", "run", "then", "rest"]

def

```
List.intersperseTR.{u} {α : Type u} (sep : α) (l : List α) : List α
```

Alternates the elements of l with sep.

O(|l|).

This is a tail-recursive version of List.intersperse, used at runtime.

Examples:

- List.intersperseTR "then" [] = []
- List.intersperseTR "then" ["walk"] = ["walk"]
- List.intersperseTR "then" ["walk", "run"] = ["walk", "then", "run"]
- List.intersperseTR "then" ["walk", "run", "rest"] = ["walk", "then", "run", "then", "rest"]

def

```
List.intercalate.{u} {α : Type u} (sep : List α) (xs : List (List α)) :
  List α
```

Alternates the lists in xs with the separator sep, appending them. The resulting list is flattened.

O(|xs|).

List.intersperse is a similar function that alternates a separator element with the elements of a list.

Examples:

- `List.intercalate sep [] = []`
- `List.intercalate sep [a] = a`
- `List.intercalate sep [a, b] = a ++ sep ++ b`
- `List.intercalate sep [a, b, c] = a ++ sep ++ b ++ sep ++ c`

def

```
List.intercalateTR.{u_1} {α : Type u_1} (sep : List α)
  (xs : List (List α)) : List α
```

Alternates the lists in `xs` with the separator `sep`.

This is a tail-recursive version of `List.intercalate` used at runtime.

Examples:

- `List.intercalateTR sep [] = []`
- `List.intercalateTR sep [a] = a`
- `List.intercalateTR sep [a, b] = a ++ sep ++ b`
- `List.intercalateTR sep [a, b, c] = a ++ sep ++ b ++ sep ++ c`

19.15.3.9. Sorting

def

```
List.mergeSort.{u_1} {α : Type u_1} (xs : List α)
  (le : α → α → Bool := by exact fun a b => a ≤ b) : List α
```

A stable merge sort.

This function is a simplified implementation that's designed to be easy to reason about, rather than for efficiency. In particular, it uses the non-tail-recursive `List.merge` function and traverses lists unnecessarily.

It is replaced at runtime by an efficient implementation that has been proven to be equivalent.

def

```
List.merge.{u_1} {α : Type u_1} (xs ys : List α)
  (le : α → α → Bool := by exact fun a b => a ≤ b) : List α
```

Merges two lists, using `le` to select the first element of the resulting list if both are non-empty.

If both input lists are sorted according to `le`, then the resulting list is also sorted according to `le.O(|xs| + |ys|)`.

This implementation is not tail-recursive, but it is replaced at runtime by a proven-equivalent tail-recursive merge.

19.15.3.10. Iteration

def

```
List.forA.{u, v, w} {m : Type u → Type v} [Applicative m] {α : Type w}
  (as : List α) (f : α → m PUnit) : m PUnit
```

Applies the applicative action `f` to every element in the list, in order.

If `m` is also a `Monad`, then using `List.forM` can be more efficient.

`List.mapA` is a variant that collects results.

def

```
List.forM.{u, v, w} {m : Type u → Type v} [Monad m] {α : Type w}
  (as : List α) (f : α → m PUnit) : m PUnit
```

Applies the monadic action `f` to every element in the list, in order.

`List.mapM` is a variant that collects results. `List.forA` is a variant that works on any `Applicative`.

def

```
List.firstM.{u, v, w} {m : Type u → Type v} [Alternative m] {α : Type w}
  {β : Type u} (f : α → m β) : List α → m β
```

Maps `f` over the list and collects the results with <|>. The result for the end of the list is `failure`.

Examples:

- `[[], [1, 2], [], [2]].firstM List.head? = some 1`
- `[[], [], []].firstM List.head? = none`
- `[].firstM List.head? = none`

def

```
List.sum.{u_1} {α : Type u_1} [Add α] [Zero α] : List α → α
```

Computes the sum of the elements of a list.

Examples:

- `[a, b, c].sum = a + (b + (c + 0))`
- `[1, 2, 5].sum = 8`

19.15.3.10.1. Folds

Folds are operators that combine the elements of a list using a function. They come in two varieties, named after the nesting of the function calls:

Left folds

Left folds combine the elements from the head of the list towards the end. The head of the list is combined with the initial value, and the result of this operation is then combined with the next value, and so forth.

Right folds

Right folds combine the elements from the end of the list towards the start, as if each `cons` constructor were replaced by a call to the combining function and `nil` were replaced by the initial value.

Monadic folds, indicated with an `-M` suffix, allow the combining function to use effects in a monad, which may include stopping the fold early.

def

```
List.foldl.{u, v} {α : Type u} {β : Type v} (f : α → β → α) (init : α) :
  List β → α
```

Folds a function over a list from the left, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in order, using `f`.

Examples:

- `[a, b, c].foldl f z  = f (f (f z a) b) c`
- `[1, 2, 3].foldl (· ++ toString ·) "" = "123"`
- `[1, 2, 3].foldl (s!"({·} {·})") "" = "((( 1) 2) 3)"`

def

```
List.foldlM.{u, v, w} {m : Type u → Type v} [Monad m] {s : Type u}
  {α : Type w} (f : s → α → m s) (init : s) : List α → m s
```

Folds a monadic function over a list from the left, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in order, using `f`.

Example:

```
example [Monad m] (f : α → β → m α) :
    List.foldlM (m := m) f x₀ [a, b, c] = (do
      let x₁ ← f x₀ a
      let x₂ ← f x₁ b
      let x₃ ← f x₂ c
      pure x₃)
  := by rfl
```

def

```
List.foldlRecOn.{u_1, u_2, u_3} {β : Type u_1} {α : Type u_2}
  {motive : β → Sort u_3} (l : List α) (op : β → α → β) {b : β} :
  motive b →
    ((b : β) → motive b → (a : α) → a ∈ l → motive (op b a)) →
      motive (List.foldl op b l)
```

A reasoning principle for proving propositions about the result of `List.foldl` by establishing an invariant that is true for the initial data and preserved by the operation being folded.

Because the motive can return a type in any sort, this function may be used to construct data as well as to prove propositions.

Example:

```
example {xs : List Nat} : xs.foldl (· + ·) 1 > 0 := by
  apply List.foldlRecOn
  . show 0 < 1; trivial
  . show ∀ (b : Nat), 0 < b → ∀ (a : Nat), a ∈ xs → 0 < b + a
    intros; omega
```

def

```
List.foldr.{u, v} {α : Type u} {β : Type v} (f : α → β → β) (init :
β)
  (l : List α) : β
```

Folds a function over a list from the right, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in reverse order, using `f`.

O(|l|). Replaced at runtime with `List.foldrTR`.

Examples:

- `[a, b, c].foldr f init  = f a (f b (f c init))`
- `[1, 2, 3].foldr (toString · ++ ·) "" = "123"`
- `[1, 2, 3].foldr (s!"({·} {·})") "!" = "(1 (2 (3 !)))"`

def

```
List.foldrM.{u, v, w} {m : Type u → Type v} [Monad m] {s : Type u}
  {α : Type w} (f : α → s → m s) (init : s) (l : List α) : m s
```

Folds a monadic function over a list from the right, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in reverse order, using `f`.

Example:

```
example [Monad m] (f : α → β → m β) :
  List.foldrM (m := m) f x₀ [a, b, c] = (do
    let x₁ ← f c x₀
    let x₂ ← f b x₁
    let x₃ ← f a x₂
    pure x₃)
  := by rfl
```

def

```
List.foldrRecOn.{u_1, u_2, u_3} {β : Type u_1} {α : Type u_2}
  {motive : β → Sort u_3} (l : List α) (op : α → β → β) {b : β} :
  motive b →
    ((b : β) → motive b → (a : α) → a ∈ l → motive (op a b)) →
      motive (List.foldr op b l)
```

A reasoning principle for proving propositions about the result of `List.foldr` by establishing an invariant that is true for the initial data and preserved by the operation being folded.

Because the motive can return a type in any sort, this function may be used to construct data as well as to prove propositions.

Example:

```
example {xs : List Nat} : xs.foldr (· + ·) 1 > 0 := by
  apply List.foldrRecOn
  . show 0 < 1; trivial
  . show ∀ (b : Nat), 0 < b → ∀ (a : Nat), a ∈ xs → 0 < a + b
    intros; omega
```

def

```
List.foldrTR.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α → β →
β)
  (init : β) (l : List α) : β
```

Folds a function over a list from the right, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in reverse order, using `f`.

`O(|l|)`. This is the tail-recursive replacement for `List.foldr` in runtime code.

Examples:

- `[a, b, c].foldrTR f init = f a (f b (f c init))`
- `[1, 2, 3].foldrTR (toString · ++ ·) "" = "123"`
- `[1, 2, 3].foldrTR (s!"({·} {·})") "!" = "(1 (2 (3 !)))"`

19.15.3.11. Transformation

def

```
List.map.{u, v} {α : Type u} {β : Type v} (f : α → β) (l : List α)
:
  List β
```

Applies a function to each element of the list, returning the resulting list of values.

`O(|l|)`.

Examples:

- `[a, b, c].map f = [f a, f b, f c]`
- `[].map Nat.succ = []`
- `["one", "two", "three"].map (·.length) = [3, 3, 5]`
- `["one", "two", "three"].map (·.reverse) = ["eno", "owt", "eerht"]`

def

```
List.mapTR.{u, v} {α : Type u} {β : Type v} (f : α → β) (as : List
α) :
  List β
```

Applies a function to each element of the list, returning the resulting list of values.

`O(|l|)`. This is the tail-recursive variant of `List.map`, used in runtime code.

Examples:

- `[a, b, c].mapTR f = [f a, f b, f c]`
- `[].mapTR Nat.succ = []`
- `["one", "two", "three"].mapTR (·.length) = [3, 3, 5]`
- `["one", "two", "three"].mapTR (·.reverse) = ["eno", "owt", "eerht"]`

def

```
List.mapM.{u, v, w} {m : Type u → Type v} [Monad m] {α : Type w}
  {β : Type u} (f : α → m β) (as : List α) : m (List β)
```

Applies the monadic action `f` to every element in the list, left-to-right, and returns the list of results.

This implementation is tail recursive. `List.mapM'` is a a non-tail-recursive variant that may be more convenient to reason about. `List.forM` is the variant that discards the results and `List.mapA` is the variant that works with `Applicative`.

def

```
List.mapM'.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type u_3}
  {β : Type u_1} [Monad m] (f : α → m β) : List α → m (List β)
```

Applies the monadic action `f` on every element in the list, left-to-right, and returns the list of results.

This is a non-tail-recursive variant of `List.mapM` that's easier to reason about. It cannot be used as the main definition and replaced by the tail-recursive version because they can only be proved equal when `m` is a `LawfulMonad`.

def

```
List.mapA.{u, v, w} {m : Type u → Type v} [Applicative m] {α : Type
w}
  {β : Type u} (f : α → m β) : List α → m (List β)
```

Applies the applicative action `f` on every element in the list, left-to-right, and returns the list of results.

If `m` is also a `Monad`, then using `mapM` can be more efficient.

See `List.forA` for the variant that discards the results. See `List.mapM` for the variant that works with `Monad`.

This function is not tail-recursive, so it may fail with a stack overflow on long lists.

def

```
List.mapFinIdx.{u_1, u_2} {α : Type u_1} {β : Type u_2} (as : List
α)
  (f : (i : Nat) → α → i ≤ as.length → β) : List β
```

Applies a function to each element of the list along with the index at which that element is found, returning the list of results. In addition to the index, the function is also provided with a proof that the index is valid.

List.mapIdx is a variant that does not provide the function with evidence that the index is valid.

def

```
List.mapFinIdxM.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type u_3}
  {β : Type u_1} [Monad m] (as : List α)
  (f : (i : Nat) → α → i < as.length → m β) : m (List β)
```

Applies a monadic function to each element of the list along with the index at which that element is found, returning the list of results. In addition to the index, the function is also provided with a proof that the index is valid.

List.mapIdxM is a variant that does not provide the function with evidence that the index is valid.

def

```
List.mapIdx.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : Nat → α → β)
  (as : List α) : List β
```

Applies a function to each element of the list along with the index at which that element is found, returning the list of results.

List.mapFinIdx is a variant that additionally provides the function with a proof that the index is valid.

def

```
List.mapIdxM.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type u_3}
  {β : Type u_1} [Monad m] (f : Nat → α → m β) (as : List α) :
  m (List β)
```

Applies a monadic function to each element of the list along with the index at which that element is found, returning the list of results.

List.mapFinIdxM is a variant that additionally provides the function with a proof that the index is valid.

def

```
List.mapMono.{u_1} {α : Type u_1} (as : List α) (f : α → α) : List α
```

Applies a function to each element of a list, returning the list of results. The function is monomorphic: it is required to return a value of the same type. The internal implementation uses pointer equality, and does not allocate a new list if the result of each function call is pointer-equal to its argument.

For verification purposes, `List.mapMono = List.map`.

def

```
List.mapMonoM.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type u_1}
  [Monad m] (as : List α) (f : α → m α) : m (List α)
```

Applies a monadic function to each element of a list, returning the list of results. The function is monomorphic: it is required to return a value of the same type. The internal implementation uses pointer equality, and does not allocate a new list if the result of each function call is pointer-equal to its argument.

def

```
List.flatMap.{u, v} {α : Type u} {β : Type v} (b : α → List β)
  (as : List α) : List β
```

Applies a function that returns a list to each element of a list, and concatenates the resulting lists.

Examples:

- `[2, 3, 2].flatMap List.range = [0, 1, 0, 1, 2, 0, 1]`
- `["red", "blue"].flatMap String.toList = ['r', 'e', 'd', 'b', 'l', 'u', 'e']`

def

```
List.flatMapTR.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α →
List β)
  (as : List α) : List β
```

Applies a function that returns a list to each element of a list, and concatenates the resulting lists.

This is the tail-recursive version of `List.flatMap` that's used at runtime.

Examples:

- `[2, 3, 2].flatMapTR List.range = [0, 1, 0, 1, 2, 0, 1]`

- `["red", "blue"].flatMapTR String.toList = ['r', 'e', 'd', 'b', 'l', 'u', 'e']`

def

```
List.flatMapM.{u, v, w} {m : Type u → Type v} [Monad m] {α : Type w}
  {β : Type u} (f : α → m (List β)) (as : List α) : m (List β)
```

Applies a monadic function that returns a list to each element of a list, from left to right, and concatenates the resulting lists.

def

```
List.zip.{u, v} {α : Type u} {β : Type v} :
  List α → List β → List (α × β)
```

Combines two lists into a list of pairs in which the first and second components are the corresponding elements of each list. The resulting list is the length of the shorter of the input lists.

`O(min |xs| |ys|)`.

Examples:

- `["Mon", "Tue", "Wed"].zip [1, 2, 3] = [("Mon", 1), ("Tue", 2), ("Wed", 3)]`
- `["Mon", "Tue", "Wed"].zip [1, 2] = [("Mon", 1), ("Tue", 2)]`
- $[x_1, x_2, x_3]$`.zip` $[y_1, y_2, y_3, y_4] = [(x_1, y_1), (x_2, y_2), (x_3, y_3)]$

def

```
List.zipIdx.{u} {α : Type u} (l : List α) (n : Nat := 0) :
  List (α × Nat)
```

Pairs each element of a list with its index, optionally starting from an index other than `0`.

`O(|l|)`.

Examples:

- $[a, b, c]$`.zipIdx = [(`a`, 0), (`b`, 1), (`c`, 2)]`
- $[a, b, c]$`.zipIdx 5 = [(`a`, 5), (`b`, 6), (`c`, 7)]`

def

```
List.zipIdxTR.{u_1} {α : Type u_1} (l : List α) (n : Nat := 0) :
  List (α × Nat)
```

Pairs each element of a list with its index, optionally starting from an index other than `0`.

`O(|l|)`. This is a tail-recursive version of `List.zipIdx` that's used at runtime.

Examples:

- `[a, b, c].zipIdxTR = [(a, 0), (b, 1), (c, 2)]`
- `[a, b, c].zipIdxTR 5 = [(a, 5), (b, 6), (c, 7)]`

def

```
List.zipWith.{u, v, w} {α : Type u} {β : Type v} {γ : Type w}
  (f : α → β → γ) (xs : List α) (ys : List β) : List γ
```

Applies a function to the corresponding elements of two lists, stopping at the end of the shorter list.

`O(min |xs| |ys|)`.

Examples:

- `[1, 2].zipWith (· + ·) [5, 6] = [6, 8]`
- `[1, 2, 3].zipWith (· + ·) [5, 6, 10] = [6, 8, 13]`
- `[].zipWith (· + ·) [5, 6] = []`
- `[x₁, x₂, x₃].zipWith f [y₁, y₂, y₃, y₄] = [f x₁ y₁, f x₂ y₂, f x₃ y₃]`

def

```
List.zipWithTR.{u_1, u_2, u_3} {α : Type u_1} {β : Type u_2}
  {γ : Type u_3} (f : α → β → γ) (as : List α) (bs : List β) : List
γ
```

Applies a function to the corresponding elements of two lists, stopping at the end of the shorter list.

`O(min |xs| |ys|)`. This is a tail-recursive version of `List.zipWith` that's used at runtime.

Examples:

- `[1, 2].zipWithTR (· + ·) [5, 6] = [6, 8]`
- `[1, 2, 3].zipWithTR (· + ·) [5, 6, 10] = [6, 8, 13]`

- `[].zipWithTR (· + ·) [5, 6] = []`
- `[x₁, x₂, x₃].zipWithTR f [y₁, y₂, y₃, y₄] = [f x₁ y₁, f x₂ y₂, f x₃ y₃]`

def

```
List.zipWithAll.{u, v, w} {α : Type u} {β : Type v} {γ : Type w}
  (f : Option α → Option β → γ) : List α → List β → List γ
```

Applies a function to the corresponding elements of both lists, stopping when there are no more elements in either list. If one list is shorter than the other, the function is passed none for the missing elements.

Examples:

- `[1, 6].zipWithAll min [5, 2] = [some 1, some 2]`
- `[1, 2, 3].zipWithAll Prod.mk [5, 6] = [(some 1, some 5), (some 2, some 6), (some 3, none)]`
- `[x₁, x₂].zipWithAll f [y] = [f (some x₁) (some y), f (some x₂) none]`

def

```
List.unzip.{u, v} {α : Type u} {β : Type v} (l : List (α × β)) :
  List α × List β
```

Separates a list of pairs into two lists that contain the respective first and second components.

`O(|l|)`.

Examples:

- `[("Monday", 1), ("Tuesday", 2)].unzip = (["Monday", "Tuesday"], [1, 2])`
- `[(x₁, y₁), (x₂, y₂), (x₃, y₃)].unzip = ([x₁, x₂, x₃], [y₁, y₂, y₃])`
- `([] : List (Nat × String)).unzip = (([], []) : List Nat × List String)`

def

```
List.unzipTR.{u, v} {α : Type u} {β : Type v} (l : List (α × β)) :
  List α × List β
```

Separates a list of pairs into two lists that contain the respective first and second components.

O(|l|). This is a tail-recursive version of List.unzip that's used at runtime.

Examples:

- [("Monday", 1), ("Tuesday", 2)].unzipTR = (["Monday", "Tuesday"], [1, 2])
- [(x_1, y_1), (x_2, y_2), (x_3, y_3)].unzipTR = ([x_1, x_2, x_3], [y_1, y_2, y_3])
- ([] : List (Nat × String)).unzipTR = (([], []) : List Nat × List String)

19.15.3.12. Filtering

def

List.filter.{u} {α : Type u} (p : α → Bool) (l : List α) : List α

Returns the list of elements in l for which p returns true.

O(|l|).

Examples:

- [1, 2, 5, 2, 7, 7].filter (· > 2) = [5, 7, 7]
- [1, 2, 5, 2, 7, 7].filter (fun _ => false) = []
- [1, 2, 5, 2, 7, 7].filter (fun _ => true) = [1, 2, 5, 2, 7, 7]

def

List.filterTR.{u} {α : Type u} (p : α → Bool) (as : List α) : List α

Returns the list of elements in l for which p returns true.

O(|l|). This is a tail-recursive version of List.filter, used at runtime.

Examples:

- [1, 2, 5, 2, 7, 7].filterTR (· > 2) = [5, 7, 7]
- [1, 2, 5, 2, 7, 7].filterTR (fun _ => false) = []
- [1, 2, 5, 2, 7, 7].filterTR (fun _ => true) = * [1, 2, 5, 2, 7, 7]

def

```
List.filterM.{v} {m : Type → Type v} [Monad m] {α : Type}
  (p : α → m Bool) (as : List α) : m (List α)
```

Applies the monadic predicate *p* to every element in the list, in order from left to right, and returns the list of elements for which *p* returns `true`.

O(|l|).

Example:

```
#eval [1, 2, 5, 2, 7, 7].filterM fun x => do
  IO.println s!"Checking {x}"
  return x < 3
```

```
Checking 1
Checking 2
Checking 5
Checking 2
Checking 7
Checking 7
```

```
[1, 2, 2]
```

def

```
List.filterRevM.{v} {m : Type → Type v} [Monad m] {α : Type}
  (p : α → m Bool) (as : List α) : m (List α)
```

Applies the monadic predicate *p* on every element in the list in reverse order, from right to left, and returns those elements for which *p* returns `true`. The elements of the returned list are in the same order as in the input list.

Example:

```
#eval [1, 2, 5, 2, 7, 7].filterRevM fun x => do
  IO.println s!"Checking {x}"
  return x < 3
```

```
Checking 7
Checking 7
Checking 2
Checking 5
Checking 2
Checking 1
```

```
[1, 2, 2]
```

def

```
List.filterMap.{u, v} {α : Type u} {β : Type v} (f : α → Option β)
:
  List α → List β
```

Applies a function that returns an `Option` to each element of a list, collecting the non-`none` values.

`O(|l|)`.

Example:

```
#eval [1, 2, 5, 2, 7, 7].filterMap fun x =>
  if x > 2 then some (2 * x) else none
```

```
[10, 14, 14]
```

def

```
List.filterMapTR.{u_1, u_2} {α : Type u_1} {β : Type u_2}
  (f : α → Option β) (l : List α) : List β
```

Applies a function that returns an `Option` to each element of a list, collecting the non-`none` values.

`O(|l|)`. This is a tail-recursive version of `List.filterMap`, used at runtime.

Example:

```
#eval [1, 2, 5, 2, 7, 7].filterMapTR fun x =>
  if x > 2 then some (2 * x) else none
```

```
[10, 14, 14]
```

def

```
List.filterMapM.{u, v, w} {m : Type u → Type v} [Monad m] {α : Type
w}
  {β : Type u} (f : α → m (Option β)) (as : List α) : m (List β)
```

Applies a monadic function that returns an `Option` to each element of a list, collecting the non-`none` values.

`O(|l|)`.

Example:

```
#eval [1, 2, 5, 2, 7, 7].filterMapM fun x => do
  IO.println s!"Examining {x}"
  if x > 2 then return some (2 * x)
  else return none
```

```
Examining 1
Examining 2
Examining 5
Examining 2
Examining 7
Examining 7

[10, 14, 14]
```

19.15.3.12.1. Partitioning

def

```
List.take.{u} {α : Type u} (n : Nat) (xs : List α) : List α
```

Extracts the first *n* elements of `xs`, or the whole list if *n* is greater than `xs.length`.

`O(min n |xs|)`.

Examples:

- `[a, b, c, d, e].take 0 = []`
- `[a, b, c, d, e].take 3 = [a, b, c]`
- `[a, b, c, d, e].take 6 = [a, b, c, d, e]`

def

```
List.takeTR.{u_1} {α : Type u_1} (n : Nat) (l : List α) : List α
```

Extracts the first *n* elements of `xs`, or the whole list if *n* is greater than `xs.length`.

`O(min n |xs|)`. This is a tail-recursive version of `List.take`, used at runtime.

Examples:

- `[a, b, c, d, e].takeTR 0 = []`
- `[a, b, c, d, e].takeTR 3 = [a, b, c]`
- `[a, b, c, d, e].takeTR 6 = [a, b, c, d, e]`

def

```
List.takeWhile.{u} {α : Type u} (p : α → Bool) (xs : List α) : List
α
```

Returns the longest initial segment of `xs` for which *p* returns true.

`O(|xs|)`.

Examples:

- `[7, 6, 4, 8].takeWhile (· > 5) = [7, 6]`
- `[7, 6, 6, 5].takeWhile (· > 5) = [7, 6, 6]`
- `[7, 6, 6, 8].takeWhile (· > 5) = [7, 6, 6, 8]`

def

```
List.takeWhileTR.{u_1} {α : Type u_1} (p : α → Bool) (l : List α) :
  List α
```

Returns the longest initial segment of `xs` for which *p* returns true.

`O(|xs|)`. This is a tail-recursive version of `List.take`, used at runtime.

Examples:

- `[7, 6, 4, 8].takeWhileTR (· > 5) = [7, 6]`
- `[7, 6, 6, 5].takeWhileTR (· > 5) = [7, 6, 6]`
- `[7, 6, 6, 8].takeWhileTR (· > 5) = [7, 6, 6, 8]`

def

```
List.drop.{u} {α : Type u} (n : Nat) (xs : List α) : List α
```

Removes the first *n* elements of the list *xs*. Returns the empty list if *n* is greater than the length of the list.

`O(min n |xs|)`.

Examples:

- `[0, 1, 2, 3, 4].drop 0 = [0, 1, 2, 3, 4]`
- `[0, 1, 2, 3, 4].drop 3 = [3, 4]`
- `[0, 1, 2, 3, 4].drop 6 = []`

def

```
List.dropWhile.{u} {α : Type u} (p : α → Bool) : List α → List α
```

Removes the longest prefix of a list for which *p* returns `true`.

Elements are removed from the list until one is encountered for which *p* returns `false`. This element and the remainder of the list are returned.

`O(|l|)`.

Examples:

- `[1, 3, 2, 4, 2, 7, 4].dropWhile (· < 4) = [4, 2, 7, 4]`
- `[8, 3, 2, 4, 2, 7, 4].dropWhile (· < 4) = [8, 3, 2, 4, 2, 7, 4]`
- `[8, 3, 2, 4, 2, 7, 4].dropWhile (· < 100) = []`

def

```
List.dropLast.{u_1} {α : Type u_1} : List α → List α
```

Removes the last element of the list, if one exists.

Examples:

- `[].dropLast = []`
- `["tea"].dropLast = []`
- `["tea", "coffee", "juice"].dropLast = ["tea", "coffee"]`

def

```
List.dropLastTR.{u_1} {α : Type u_1} (l : List α) : List α
```

Removes the last element of the list, if one exists.

This is a tail-recursive version of `List.dropLast`, used at runtime.

Examples:

- `[].dropLastTR = []`
- `["tea"].dropLastTR = []`
- `["tea", "coffee", "juice"].dropLastTR = ["tea", "coffee"]`

def

```
List.splitAt.{u} {α : Type u} (n : Nat) (l : List α) : List α × List α
```

Splits a list at an index, resulting in the first n elements of l paired with the remaining elements.

If n is greater than the length of l, then the resulting pair consists of l and the empty list. `List.splitAt` is equivalent to a combination of `List.take` and `List.drop`, but it is more efficient.

Examples:

- `["red", "green", "blue"].splitAt 2 = (["red", "green"], ["blue"])`
- `["red", "green", "blue"].splitAt 3 = (["red", "green", "blue"], [])`
- `["red", "green", "blue"].splitAt 4 = (["red", "green", "blue"], [])`

def

```
List.span.{u} {α : Type u} (p : α → Bool) (as : List α) :
  List α × List α
```

Splits a list into the the longest initial segment for which *p* returns `true`, paired with the remainder of the list.

`O(|l|)`.

Examples:

- `[6, 8, 9, 5, 2, 9].span (· > 5) = ([6, 8, 9], [5, 2, 9])`
- `[6, 8, 9, 5, 2, 9].span (· > 10) = ([], [6, 8, 9, 5, 2, 9])`
- `[6, 8, 9, 5, 2, 9].span (· > 0) = ([6, 8, 9, 5, 2, 9], [])`

def

```
List.splitBy.{u} {α : Type u} (R : α → α → Bool) :
  List α → List (List α)
```

Splits a list into the longest segments in which each pair of adjacent elements are related by *R*.

`O(|l|)`.

Examples:

- `[1, 1, 2, 2, 2, 3, 2].splitBy (· == ·) = [[1, 1], [2, 2, 2], [3], [2]]`
- `[1, 2, 5, 4, 5, 1, 4].splitBy (· < ·) = [[1, 2, 5], [4, 5], [1, 4]]`
- `[1, 2, 5, 4, 5, 1, 4].splitBy (fun _ _ => true) = [[1, 2, 5, 4, 5, 1, 4]]`
- `[1, 2, 5, 4, 5, 1, 4].splitBy (fun _ _ => false) = [[1], [2], [5], [4], [5], [1], [4]]`

def

```
List.partition.{u} {α : Type u} (p : α → Bool) (as : List α) :
  List α × List α
```

Returns a pair of lists that together contain all the elements of `as`. The first list contains those elements for which `p` returns `true`, and the second contains those for which `p` returns `false`.

`O(|l|).as.partition p` is equivalent to `(as.filter p, as.filter (not ∘ p))`, but it is slightly more efficient since it only has to do one pass over the list.

Examples:

- `[1, 2, 5, 2, 7, 7].partition (· > 2) = ([5, 7, 7], [1, 2, 2])`
- `[1, 2, 5, 2, 7, 7].partition (fun _ => false) = ([], [1, 2, 5, 2, 7, 7])`
- `[1, 2, 5, 2, 7, 7].partition (fun _ => true) = ([1, 2, 5, 2, 7, 7], [])`

def

```
List.partitionM.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  (p : α → m Bool) (l : List α) : m (List α × List α)
```

Returns a pair of lists that together contain all the elements of `as`. The first list contains those elements for which the monadic predicate `p` returns `true`, and the second contains those for which `p` returns `false`. The list's elements are examined in order, from left to right.

This is a monadic version of `List.partition`.

Example:

```
def posOrNeg (x : Int) : Except String Bool :=
  if x > 0 then pure true
  else if x < 0 then pure false
  else throw "Zero is not positive or negative"

#eval [-1, 2, 3].partitionM posOrNeg

Except.ok ([2, 3], [-1])

#eval [0, 2, 3].partitionM posOrNeg

Except.error "Zero is not positive or negative"
```

def

```
List.partitionMap.{u_1, u_2, u_3} {α : Type u_1} {β : Type u_2}
  {γ : Type u_3} (f : α → β ⊕ γ) (l : List α) : List β × List γ
```

Applies a function that returns a disjoint union to each element of a list, collecting the `Sum.inl` and `Sum.inr` results into separate lists.

Examples:

- `[0, 1, 2, 3].partitionMap (fun x => if x % 2 = 0 then .inl x else .inr x) = ([0, 2], [1, 3])`
- `[0, 1, 2, 3].partitionMap (fun x => if x = 0 then .inl x else .inr x) = ([0], [1, 2, 3])`

def

```
List.groupByKey.{u, v} {α : Type u} {β : Type v} [BEq α] [Hashable α]
  (key : β → α) (xs : List β) : Std.HashMap α (List β)
```

Groups the elements of a list `xs` according to the function `key`, returning a hash map in which each group is associated with its key. Groups preserve the relative order of elements in `xs`.

Example:

```
#eval [0, 1, 2, 3, 4, 5, 6].groupByKey (· % 2)
```

```
Std.HashMap.ofList [(0, [0, 2, 4, 6]), (1, [1, 3, 5])]
```

19.15.3.13. Element Predicates

def

```
List.contains.{u} {α : Type u} [BEq α] (as : List α) (a : α) : Bool
```

Checks whether `a` is an element of `as`, using == to compare elements.

`O(|as|)`. `List.elem` is a synonym that takes the element before the list.

The preferred simp normal form is `l.contains a`, and when `LawfulBEq α` is available, `l.contains a = true ↔ a ∈ l` and `l.contains a = false ↔ a ∉ l`.

Examples:

- `[1, 4, 2, 3, 3, 7].contains 3 = true`
- `List.contains [1, 4, 2, 3, 3, 7] 5 = false`

def

```
List.elem.{u} {α : Type u} [BEq α] (a : α) (l : List α) : Bool
```

Checks whether *a* is an element of *l*, using == to compare elements.

O(|l|). `List.contains` is a synonym that takes the list before the element.

The preferred simp normal form is *l*`.contains` *a*. When `LawfulBEq` *α* is available, *l*`.contains` *a* `= true` ↔ *a* ∈ *l* and *l*`.contains` *a* `= false` ↔ *a* ∉ *l*.

Example:

- `List.elem 3 [1, 4, 2, 3, 3, 7] = true`
- `List.elem 5 [1, 4, 2, 3, 3, 7] = false`

def

```
List.all.{u} {α : Type u} : List α → (α → Bool) → Bool
```

Returns `true` if p returns `true` for every element of `l`.

O(|l|). Short-circuits upon encountering the first `false`.

Examples:

- `[a, b, c].all p = (p a && (p b && p c))`
- `[2, 4, 6].all (· % 2 = 0) = true`
- `[2, 4, 5, 6].all (· % 2 = 0) = false`

def

```
List.allM.{u, v} {m : Type → Type u} [Monad m] {α : Type v}
  (p : α → m Bool) (l : List α) : m Bool
```

Returns true if the monadic predicate *p* returns `true` for every element of *l*.

O(|l|). Short-circuits upon encountering the first `false`. The elements in *l* are examined in order from left to right.

def

```
List.any.{u} {α : Type u} (l : List α) (p : α → Bool) : Bool
```

Returns `true` if *p* returns `true` for any element of *l*.

O(|l|). Short-circuits upon encountering the first `true`.

Examples:

- `[2, 4, 6].any (· % 2 = 0) = true`

- `[2, 4, 6].any (· % 2 = 1) = false`
- `[2, 4, 5, 6].any (· % 2 = 0) = true`
- `[2, 4, 5, 6].any (· % 2 = 1) = true`

def

```
List.anyM.{u, v} {m : Type → Type u} [Monad m] {α : Type v}
  (p : α → m Bool) (l : List α) : m Bool
```

Returns true if the monadic predicate p returns `true` for any element of l.

O(|l|). Short-circuits upon encountering the first `true`. The elements in l are examined in order from left to right.

def

```
List.and (bs : List Bool) : Bool
```

Returns `true` if every element of bs is the value `true`.

O(|bs|). Short-circuits at the first `false` value.

- `[true, true, true].and = true`
- `[true, false, true].and = false`
- `[true, false, false].and = false`
- `[].and = true`

def

```
List.or (bs : List Bool) : Bool
```

Returns `true` if `true` is an element of the list bs.

O(|bs|). Short-circuits at the first `true` value.

- `[true, true, true].or = true`
- `[true, false, true].or = true`
- `[false, false, false].or = false`
- `[false, false, true].or = true`
- `[].or = false`

19.15.3.14. Comparisons

def

```
List.beq.{u} {α : Type u} [BEq α] : List α → List α → Bool
```

Checks whether two lists have the same length and their elements are pairwise BEq. Normally used via the == operator.

def

```
List.isEqv.{u} {α : Type u} (as bs : List α) (eqv : α → α → Bool) : Bool
```

Returns true if *as* and *bs* have the same length and they are pairwise related by *eqv*.

O(min |as| |bs|). Short-circuits at the first non-related pair of elements.

Examples:

- [1, 2, 3].isEqv [2, 3, 4] (· < ·) = true
- [1, 2, 3].isEqv [2, 2, 4] (· < ·) = false
- [1, 2, 3].isEqv [2, 3] (· < ·) = false

def

```
List.isPerm.{u} {α : Type u} [BEq α] : List α → List α → Bool
```

Returns true if l_1 and l_2 are permutations of each other. $O(|l_1| * |l_2|)$.

The relation List.Perm is a logical characterization of permutations. When the BEq α instance corresponds to DecidableEq α, isPerm l_1 l_2 ↔ l_1 ~ l_2 (use the theorem isPerm_iff).

def

```
List.isPrefixOf.{u} {α : Type u} [BEq α] : List α → List α → Bool
```

Checks whether the first list is a prefix of the second.

The relation List.IsPrefixOf expresses this property with respect to logical equality.

Examples:

- [1, 2].isPrefixOf [1, 2, 3] = true
- [1, 2].isPrefixOf [1, 2] = true
- [1, 2].isPrefixOf [1] = false

- `[1, 2].isPrefixOf [1, 1, 2, 3] = false`

def

```
List.isPrefixOf?.{u} {α : Type u} [BEq α] (l₁ l₂ : List α) :
  Option (List α)
```

If the first list is a prefix of the second, returns the result of dropping the prefix.

In other words, `isPrefixOf? l₁ l₂` returns `some t` when `l₂ == l₁ ++ t`.

Examples:

- `[1, 2].isPrefixOf? [1, 2, 3] = some [3]`
- `[1, 2].isPrefixOf? [1, 2] = some []`
- `[1, 2].isPrefixOf? [1] = none`
- `[1, 2].isPrefixOf? [1, 1, 2, 3] = none`

def

```
List.isSublist.{u} {α : Type u} [BEq α] : List α → List α → Bool
```

True if the first list is a potentially non-contiguous sub-sequence of the second list, comparing elements with the == operator.

The relation `List.Sublist` is a logical characterization of this property.

Examples:

- `[1, 3].isSublist [0, 1, 2, 3, 4] = true`
- `[1, 3].isSublist [0, 1, 2, 4] = false`

def

```
List.isSuffixOf.{u} {α : Type u} [BEq α] (l₁ l₂ : List α) : Bool
```

Checks whether the first list is a suffix of the second.

The relation `List.IsSuffixOf` expresses this property with respect to logical equality.

Examples:

- `[2, 3].isSuffixOf [1, 2, 3] = true`
- `[2, 3].isSuffixOf [1, 2, 3, 4] = false`
- `[2, 3].isSuffixOf [1, 2] = false`

- `[2, 3].isSuffixOf [1, 1, 2, 3] = true`

def

```
List.isSuffixOf?.{u} {α : Type u} [BEq α] (l₁ l₂ : List α) :
  Option (List α)
```

If the first list is a suffix of the second, returns the result of dropping the suffix from the second.

In other words, `isSuffixOf? l₁ l₂` returns `some t` when `l₂ == t ++ l₁`.

Examples:

- `[2, 3].isSuffixOf? [1, 2, 3] = some [1]`
- `[2, 3].isSuffixOf? [1, 2, 3, 4] = none`
- `[2, 3].isSuffixOf? [1, 2] = none`
- `[2, 3].isSuffixOf? [1, 1, 2, 3] = some [1, 1]`

def

```
List.le.{u} {α : Type u} [LT α] (as bs : List α) : Prop
```

Non-strict ordering of lists with respect to a strict ordering of their elements.

`as ≤ bs` if `¬ bs < as`.

This relation can be treated as a lexicographic order if the underlying `LT α` instance is well-behaved. In particular, it should be irreflexive, asymmetric, and antisymmetric. These requirements are precisely formulated in `List.cons_le_cons_iff`. If these hold, then `as ≤ bs` if and only if:

- `as` is empty, or
- both `as` and `bs` are non-empty, and the head of `as` is less than the head of `bs`, or
- both `as` and `bs` are non-empty, their heads are equal, and the tail of `as` is less than or equal to the tail of `bs`.

def

```
List.lt.{u} {α : Type u} [LT α] : List α → List α → Prop
```

Lexicographic ordering of lists with respect to an ordering on their elements.

`as < bs` if

- `as` is empty and `bs` is non-empty, or
- both `as` and `bs` are non-empty, and the head of `as` is less than the head of `bs`, or

- both `as` and `bs` are non-empty, their heads are equal, and the tail of `as` is less than the tail of `bs`.

def

```
List.lex.{u} {α : Type u} [BEq α] (l₁ l₂ : List α)
  (lt : α → α → Bool := by exact (· < ·)) : Bool
```

Compares lists lexicographically with respect to a comparison on their elements.

The lexicographic order with respect to `lt` is:

- `[].lex (b :: bs)` is `true`
- `as.lex [] = false` is `false`
- `(a :: as).lex (b :: bs)` is true if `lt a b` or `a == b` and `lex lt as bs` is true.

19.15.3.15. Termination Helpers

def

```
List.attach.{u_1} {α : Type u_1} (l : List α) : List { x // x ∈ l }
```

“Attaches” the proof that the elements of `l` are in fact elements of `l`, producing a new list with the same elements but in the subtype `{ x // x ∈ l }`.

`O(1)`.

This function is primarily used to allow definitions by well-founded recursion that use higher-order functions (such as `List.map`) to prove that an value taken from a list is smaller than the list. This allows the well-founded recursion mechanism to prove that the function terminates.

def

```
List.attachWith.{u_1} {α : Type u_1} (l : List α) (P : α → Prop)
  (H : ∀ (x : α), x ∈ l → P x) : List { x // P x }
```

“Attaches” individual proofs to a list of values that satisfy a predicate *P*, returning a list of elements in the corresponding subtype `{ x // P x }`.

`O(1)`.

def

```
List.unattach.{u_1} {α : Type u_1} {p : α → Prop}
  (l : List { x // p x }) : List α
```

Maps a list of terms in a subtype to the corresponding terms in the type by forgetting that they satisfy the predicate.

This is the inverse of `List.attachWith` and a synonym for `l.map (·.val)`.

Mostly this should not be needed by users. It is introduced as an intermediate step by lemmas such as `map_subtype`, and is ideally subsequently simplified away by `unattach_attach`.

This function is usually inserted automatically by Lean as an intermediate step while proving termination. It is rarely used explicitly in code. It is introduced as an intermediate step during the elaboration of definitions by well-founded recursion. If this function is encountered in a proof state, the right approach is usually the tactic `simp [List.unattach, -List.map_subtype]`.

def

```
List.pmap.{u_1, u_2} {α : Type u_1} {β : Type u_2} {P : α → Prop}
  (f : (a : α) → P a → β) (l : List α) (H : ∀ (a : α), a ∈ l → P a)
:
  List β
```

Maps a partially defined function (defined on those terms of α that satisfy a predicate P) over a list `l : List α`, given a proof that every element of `l` in fact satisfies P.

`O(|l|)`. `List.pmap`, named for “partial map,” is the equivalent of `List.map` for such partial functions.

19.16. Arrays

The `Array` type represents sequences of elements, addressable by their position in the sequence. Arrays are specially supported by Lean:

- They have a *logical model* that specifies their behavior in terms of lists of elements, which specifies the meaning of each operation on arrays.
- They have an optimized run-time representation in compiled code as dynamic arrays, and the Lean runtime specially optimizes array operations.
- There is array literal syntax for writing arrays.

Arrays can be vastly more efficient than lists or other sequences in compiled code. In part, this is because they offer good locality: because all the elements of the sequence are next to each other in memory, the processor's caches can be used efficiently. Even more importantly, if there is only a single reference to an array, operations that might otherwise copy or allocate a data structure can be implemented via mutation. Lean code that uses an array in such a way that there's only ever one

unique reference (that is, uses it *linearly*) avoids the performance overhead of persistent data structures while still being as convenient to write, read, and prove things about as ordinary pure functional programs.

19.16.1. Logical Model

structure

```
Array.{u} (α : Type u) : Type u
```

`Array α` is the type of dynamic arrays with elements from `α`. This type has special support in the runtime.

Arrays perform best when unshared. As long as there is never more than one reference to an array, all updates will be performed *destructively*. This results in performance comparable to mutable arrays in imperative programming languages.

An array has a size and a capacity. The size is the number of elements present in the array, while the capacity is the amount of memory currently allocated for elements. The size is accessible via `Array.size`, but the capacity is not observable from Lean code. `Array.emptyWithCapacity n` creates an array which is equal to `#[]`, but internally allocates an array of capacity `n`. When the size exceeds the capacity, allocation is required to grow the array.

From the point of view of proofs, `Array α` is just a wrapper around `List α`.

Constructor

`Array.mk.{u}`

Converts a `List α` into an `Array α`.

The function `List.toArray` is preferred.

At runtime, this constructor is overridden by `List.toArrayImpl` and is `O(n)` in the length of the list.

Fields

`toList : List α`

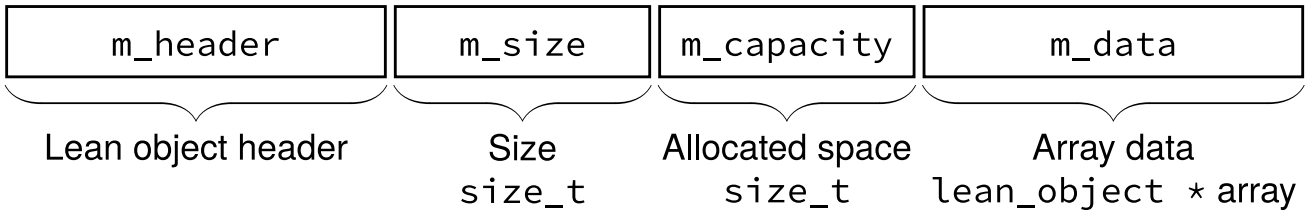
Converts an `Array α` into a `List α` that contains the same elements in the same order.

At runtime, this is implemented by `Array.toListImpl` and is `O(n)` in the length of the array.

The logical model of arrays is a structure that contains a single field, which is a list of elements. This is convenient when specifying and proving properties of array-processing functions at a low level.

19.16.2. Run-Time Representation

Lean's arrays are *dynamic arrays*, which are blocks of continuous memory with a defined capacity, not all of which is typically in use. As long as the number of elements in the array is less than the capacity, new items can be added to the end without reallocating or moving the data. Adding items to an array that has no extra space results in a reallocation that doubles the capacity. The amortized overhead scales linearly with the size of the array. The values in the array are represented as described in the section on the foreign function interface.



Memory layout of arrays

After the object header, an array contains:

size
: The number of objects currently stored in the array

capacity
: The number of objects that fit in the memory allocated for the array

data
: The values in the array

Many array functions in the Lean runtime check whether they have exclusive access to their argument by consulting the reference count in the object header. If they do, and the array's capacity is sufficient, then the existing array can be mutated rather than allocating fresh memory. Otherwise, a new array must be allocated.

19.16.2.1. Performance Notes

Despite the fact that they appear to be an ordinary constructor and projection, `Array.mk` and `Array.toList` take **time linear in the size of the array** in compiled code. This is because converting between linked lists and packed arrays must necessarily visit each element.

Mutable arrays can be used to write very efficient code. However, they are a poor persistent data structure. Updating a shared array rules out mutation, and requires time linear in the size of the array. When using arrays in performance-critical code, it's important to ensure that they are used linearly.

19.16.3. Syntax

Array literals allow arrays to be written directly in code. They may be used in expression or pattern contexts.

syntax

Array Literals

Array literals begin with `#[` and contain a comma-separated sequence of terms, terminating with `]`.

```
term ::= ...
    | #[term,*]
```

▼ *Example: Array Literals*

Array literals may be used as expressions or as patterns.

```
def oneTwoThree : Array Nat := #[1, 2, 3]

#eval
  match oneTwoThree with
  | #[x, y, z] => some ((x + z) / y)
  | _ => none
```

Additionally, sub-arrays may be extracted using the following syntax:

syntax

Sub-Arrays

A start index followed by a colon constructs a sub-array that contains the values from the start index onwards (inclusive):

```
term ::= ...
    | term[term :]
```

Providing start and end indices constructs a sub-array that contains the values from the start index (inclusive) to the end index (exclusive):

```
term ::= ...
    | term[term : term]
```

▼ *Example: Sub-Array Syntax*

The array `ten` contains the first ten natural numbers.

```
def ten : Array Nat :=
  ..range 10
```

A sub-array that represents the second half of `ten` can be constructed using the sub-array syntax:

```
#eval ten[5:]
```

```
#[5, 6, 7, 8, 9].toSubarray
```

Similarly, sub-array that contains two through five can be constructed by providing a stopping point:

```
#eval ten[2:6]
```

```
#[2, 3, 4, 5].toSubarray
```

Because sub-arrays merely store the start and end indices of interest in the underlying array, the array itself can be recovered:

```
#eval ten[2:6].array == ten
```

```
true
```

19.16.4. API Reference

19.16.4.1. Constructing Arrays

def

```
Array.empty.{u} {α : Type u} : Array α
```

Constructs a new empty array with initial capacity `0`.

Use `Array.emptyWithCapacity` to create an array with a greater initial capacity.

def

```
Array.emptyWithCapacity.{u} {α : Type u} (c : Nat) : Array α
```

Constructs a new empty array with initial capacity *c*.

def

```
Array.singleton.{u} {α : Type u} (v : α) : Array α
```

Constructs a single-element array that contains *v*.

Examples:

- `Array.singleton 5 = #[5]`

- `Array.singleton "one" = #["one"]`

def

```
Array.range (n : Nat) : Array Nat
```

Constructs an array that contains all the numbers from `0` to *n*, exclusive.

Examples:

- `Array.range 5 := #[0, 1, 2, 3, 4]`
- `Array.range 0 := #[]`
- `Array.range 1 := #[0]`

def

```
Array.range' (start size : Nat) (step : Nat := 1) : Array Nat
```

Constructs an array of numbers of size *size*, starting at *start* and increasing by *step* at each element.

In other words, `Array.range' start size step` is `#[start, start+step, ..., start+(len-1)*step]`.

Examples:

- `Array.range' 0 3 (step := 1) = #[0, 1, 2]`
- `Array.range' 0 3 (step := 2) = #[0, 2, 4]`
- `Array.range' 0 4 (step := 2) = #[0, 2, 4, 6]`
- `Array.range' 3 4 (step := 2) = #[3, 5, 7, 9]`

def

```
Array.finRange (n : Nat) : Array (Fin n)
```

Returns an array of all elements of `Fin n` in order, starting at `0`.

Examples:

- `Array.finRange 0 = (#[] : Array (Fin 0))`
- `Array.finRange 2 = (#[0, 1] : Array (Fin 2))`

def

```
Array.ofFn.{u} {α : Type u} {n : Nat} (f : Fin n → α) : Array α
```

Creates an array by applying *f* to each potential index in order, starting at `0`.

Examples:

- `Array.ofFn (n := 3) toString = #["0", "1", "2"]`
- `Array.ofFn (fun i => #["red", "green", "blue"].get i.val i.isLt) = #["red", "green", "blue"]`

def

`Array.replicate.{u} {α : Type u} (n : Nat) (v : α) : Array α`

Creates an array that contains *n* repetitions of *v*.

The corresponding `List` function is `List.replicate`.

Examples:

- `Array.replicate 2 true = #[true, true]`
- `Array.replicate 3 () = #[(), (), ()]`
- `Array.replicate 0 "anything" = #[]`

def

`Array.append.{u} {α : Type u} (as bs : Array α) : Array α`

Appends two arrays. Normally used via the ++ operator.

Appending arrays takes time proportional to the length of the second array.

Examples:

- `#[1, 2, 3] ++ #[4, 5] = #[1, 2, 3, 4, 5]`.
- `#[] ++ #[4, 5] = #[4, 5]`.
- `#[1, 2, 3] ++ #[] = #[1, 2, 3]`.

def

`Array.appendList.{u} {α : Type u} (as : Array α) (bs : List α) : Array α`

Appends an array and a list.

Takes time proportional to the length of the list..

Examples:

- `#[1, 2, 3].appendList [4, 5] = #[1, 2, 3, 4, 5]`.
- `#[].appendList [4, 5] = #[4, 5]`.
- `#[1, 2, 3].appendList [] = #[1, 2, 3]`.

def

```
Array.leftpad.{u} {α : Type u} (n : Nat) (a : α) (xs : Array α) :
  Array α
```

Pads `xs : Array α` on the left with repeated occurrences of `a : α` until it is of size `n`. If `xs` already has at least `n` elements, it is returned unmodified.

Examples:

- `#[1, 2, 3].leftpad 5 0 = #[0, 0, 1, 2, 3]`
- `#["red", "green", "blue"].leftpad 4 "blank" = #["blank", "red", "green", "blue"]`
- `#["red", "green", "blue"].leftpad 3 "blank" = #["red", "green", "blue"]`
- `#["red", "green", "blue"].leftpad 1 "blank" = #["red", "green", "blue"]`

def

```
Array.rightpad.{u} {α : Type u} (n : Nat) (a : α) (xs : Array α) :
  Array α
```

Pads `xs : Array α` on the right with repeated occurrences of `a : α` until it is of length `n`. If `l` already has at least `n` elements, it is returned unmodified.

Examples:

- `#[1, 2, 3].rightpad 5 0 = #[1, 2, 3, 0, 0]`
- `#["red", "green", "blue"].rightpad 4 "blank" = #["red", "green", "blue", "blank"]`
- `#["red", "green", "blue"].rightpad 3 "blank" = #["red", "green", "blue"]`
- `#["red", "green", "blue"].rightpad 1 "blank" = #["red", "green", "blue"]`

19.16.4.2. Size

def

```
Array.size.{u} {α : Type u} (a : Array α) : Nat
```

Gets the number of elements stored in an array.

This is a cached value, so it is `O(1)` to access. The space allocated for an array, referred to as its *capacity*, is at least as large as its size, but may be larger. The capacity of an array is an internal detail that's not observable by Lean code.

def

```
Array.usize.{u} {α : Type u} (a : Array α) : USize
```

Returns the size of the array as a platform-native unsigned integer.

This is a low-level version of `Array.size` that directly queries the runtime system's representation of arrays. While this is not provable, `Array.usize` always returns the exact size of the array since the implementation only supports arrays of size less than `USize.size`.

def

```
Array.isEmpty.{u} {α : Type u} (xs : Array α) : Bool
```

Checks whether an array is empty.

An array is empty if its size is `0`.

Examples:

- `(#[] : Array String).isEmpty = true`
- `#[1, 2].isEmpty = false`
- `#[()].isEmpty = false`

19.16.4.3. Lookups

def

```
Array.extract.{u_1} {α : Type u_1} (as : Array α) (start : Nat :=
0)
  (stop : Nat := as.size) : Array α
```

Returns the slice of *as* from indices *start* to *stop* (exclusive). The resulting array has size *(min stop as.size) - start*.

If *start* is greater or equal to *stop*, the result is empty. If *stop* is greater than the size of *as*, the size is used instead.

Examples:

- `#[0, 1, 2, 3, 4].extract 1 3 = #[1, 2]`

- `#[0, 1, 2, 3, 4].extract 1 30 = #[1, 2, 3, 4]`
- `#[0, 1, 2, 3, 4].extract 0 0 = #[]`
- `#[0, 1, 2, 3, 4].extract 2 1 = #[]`
- `#[0, 1, 2, 3, 4].extract 2 2 = #[]`
- `#[0, 1, 2, 3, 4].extract 2 3 = #[2]`
- `#[0, 1, 2, 3, 4].extract 2 4 = #[2, 3]`

def

```
Array.getD.{u_1} {α : Type u_1} (a : Array α) (i : Nat) (v₀ : α) : α
```

Returns the element at the provided index, counting from `0`. Returns the fallback value v_0 if the index is out of bounds.

To return an `Option` depending on whether the index is in bounds, use `a[i]?`. To panic if the index is out of bounds, use `a[i]!`.

Examples:

- `#["spring", "summer", "fall", "winter"].getD 2 "never" = "fall"`
- `#["spring", "summer", "fall", "winter"].getD 0 "never" = "spring"`
- `#["spring", "summer", "fall", "winter"].getD 4 "never" = "never"`

def

```
Array.uget.{u} {α : Type u} (a : Array α) (i : USize)
  (h : i.toNat < a.size) : α
```

Low-level indexing operator which is as fast as a C array read.

This avoids overhead due to unboxing a `Nat` used as an index.

def

```
Array.back.{u} {α : Type u} (xs : Array α)
  (h : 0 < xs.size := by get_elem_tactic) : α
```

Returns the last element of an array, given a proof that the array is not empty.

See Array.back! for the version that panics if the array is empty, or Array.back? for the version that returns an option.

def

```
Array.back?.{u} {α : Type u} (xs : Array α) : Option α
```

Returns the last element of an array, or none if the array is empty.

See Array.back! for the version that panics if the array is empty, or Array.back for the version that requires a proof the array is non-empty.

def

```
Array.back!.{u} {α : Type u} [Inhabited α] (xs : Array α) : α
```

Returns the last element of an array, or panics if the array is empty.

Safer alternatives include Array.back, which requires a proof the array is non-empty, and Array.back?, which returns an Option.

def

```
Array.getMax?.{u} {α : Type u} (as : Array α) (lt : α → α → Bool) :
  Option α
```

Returns the largest element of the array, as determined by the comparison lt, or none if the array is empty.

Examples:

- (#[] : Array Nat).getMax? (· < ·) = none
- #["red", "green", "blue"].getMax? (·.length < ·.length) = some "green"
- #["red", "green", "blue"].getMax? (· < ·) = some "red"

19.16.4.4. Queries

def

```
Array.count.{u} {α : Type u} [BEq α] (a : α) (as : Array α) : Nat
```

Counts the number of times an element occurs in an array.

Examples:

- #[1, 1, 2, 3, 5].count 1 = 2

- `#[1, 1, 2, 3, 5].count 5 = 1`
- `#[1, 1, 2, 3, 5].count 4 = 0`

def

```
Array.countP.{u} {α : Type u} (p : α → Bool) (as : Array α) : Nat
```

Counts the number of elements in the array *as* that satisfy the Boolean predicate *p*.

Examples:

- `#[1, 2, 3, 4, 5].countP (· % 2 == 0) = 2`
- `#[1, 2, 3, 4, 5].countP (· < 5) = 4`
- `#[1, 2, 3, 4, 5].countP (· > 5) = 0`

def

```
Array.idxOf.{u} {α : Type u} [BEq α] (a : α) : Array α → Nat
```

Returns the index of the first element equal to *a*, or the size of the array if no element is equal to *a*.

Examples:

- `#["carrot", "potato", "broccoli"].idxOf "carrot" = 0`
- `#["carrot", "potato", "broccoli"].idxOf "broccoli" = 2`
- `#["carrot", "potato", "broccoli"].idxOf "tomato" = 3`
- `#["carrot", "potato", "broccoli"].idxOf "anything else" = 3`

def

```
Array.idxOf?.{u} {α : Type u} [BEq α] (xs : Array α) (v : α) :
  Option Nat
```

Returns the index of the first element equal to a, or `none` if no element is equal to a.

Examples:

- `#["carrot", "potato", "broccoli"].idxOf? "carrot" = some 0`
- `#["carrot", "potato", "broccoli"].idxOf? "broccoli" = some 2`
- `#["carrot", "potato", "broccoli"].idxOf? "tomato" = none`
- `#["carrot", "potato", "broccoli"].idxOf? "anything else" = none`

def

```
Array.finIdxOf?.{u} {α : Type u} [BEq α] (xs : Array α) (v : α) :
  Option (Fin xs.size)
```

Returns the index of the first element equal to a, or the size of the array if no element is equal to a. The index is returned as a `Fin`, which guarantees that it is in bounds.

Examples:

- `#["carrot", "potato", "broccoli"].finIdxOf? "carrot" = some 0`
- `#["carrot", "potato", "broccoli"].finIdxOf? "broccoli" = some 2`
- `#["carrot", "potato", "broccoli"].finIdxOf? "tomato" = none`
- `#["carrot", "potato", "broccoli"].finIdxOf? "anything else" = none`

19.16.4.5. Conversions

def

```
Array.toList.{u} {α : Type u} (self : Array α) : List α
```

Converts an `Array α` into a `List α` that contains the same elements in the same order.

At runtime, this is implemented by `Array.toListImpl` and is `O(n)` in the length of the array.

def

```
Array.toListRev.{u_1} {α : Type u_1} (xs : Array α) : List α
```

Converts an array to a list that contains the same elements in the opposite order.

This is equivalent to, but more efficient than, `Array.toList ∘ List.reverse`.

Examples:

- `#[1, 2, 3].toListRev = [3, 2, 1]`
- `#["blue", "yellow"].toListRev = ["yellow", "blue"]`

def

```
Array.toListAppend.{u} {α : Type u} (as : Array α) (l : List α) :
List α
```

Prepends an array to a list. The elements of the array are at the beginning of the resulting list.

Equivalent to *as*`.toList ++ `*l*.

Examples:

- `#[1, 2].toListAppend [3, 4] = [1, 2, 3, 4]`
- `#[1, 2].toListAppend [] = [1, 2]`
- `#[].toListAppend [3, 4, 5] = [3, 4, 5]`

def

```
Array.toVector.{u_1} {α : Type u_1} (xs : Array α) : Vector α
  xs.size
```

Converts an array to a vector. The resulting vector's size is the array's size.

def

```
Array.toSubarray.{u} {α : Type u} (as : Array α) (start : Nat := 0)
  (stop : Nat := as.size) : Subarray α
```

Returns a subarray of an array, with the given bounds.

If *start* or *stop* are not valid bounds for a subarray, then they are clamped to array's size. Additionally, the starting index is clamped to the ending index.

def

```
Array.ofSubarray.{u} {α : Type u} (s : Subarray α) : Array α
```

Allocates a new array that contains the contents of the subarray.

19.16.4.6. Modification

def

```
Array.push.{u} {α : Type u} (a : Array α) (v : α) : Array α
```

Adds an element to the end of an array. The resulting array's size is one greater than the input array. If there are no other references to the array, then it is modified in-place.

This takes amortized `O(1)` time because `Array α` is represented by a dynamic array.

Examples:

- `#[].push "apple" = #["apple"]`
- `#["apple"].push "orange" = #["apple", "orange"]`

def

```
Array.pop.{u} {α : Type u} (xs : Array α) : Array α
```

Removes the last element of an array. If the array is empty, then it is returned unmodified. The modification is performed in-place when the reference to the array is unique.

Examples:

- `#[1, 2, 3].pop = #[1, 2]`
- `#["orange", "yellow"].pop = #["orange"]`
- `(#[] : Array String).pop = #[]`

def

```
Array.popWhile.{u} {α : Type u} (p : α → Bool) (as : Array α) :
Array α
```

Removes all the elements that satisfy a predicate from the end of an array.

The longest contiguous sequence of elements that all satisfy the predicate is removed.

Examples:

- `#[0, 1, 2, 3, 4].popWhile (· > 2) = #[0, 1, 2]`
- `#[3, 2, 3, 4].popWhile (· > 2) = #[3, 2]`
- `(#[] : Array Nat).popWhile (· > 2) = #[]`

def

```
Array.erase.{u} {α : Type u} [BEq α] (as : Array α) (a : α) : Array
α
```

Removes the first occurrence of a specified element from an array, or does nothing if it is not present.

This function takes worst-case $O(n)$ time because it back-shifts all later elements.

Examples:

- `#[1, 2, 3].erase 2 = #[1, 3]`
- `#[1, 2, 3].erase 5 = #[1, 2, 3]`
- `#[1, 2, 3, 2, 1].erase 2 = #[1, 3, 2, 1]`
- `(#[] : List Nat).erase 2 = #[]`

def

```
Array.eraseP.{u} {α : Type u} (as : Array α) (p : α → Bool) : Array α
```

Removes the first element that satisfies the predicate *p*. If no element satisfies *p*, the array is returned unmodified.

This function takes worst-case `O(n)` time because it back-shifts all later elements.

Examples:

- `#["red", "green", "", "blue"].eraseP (·.isEmpty) = #["red", "green", "blue"]`
- `#["red", "green", "", "blue", ""].eraseP (·.isEmpty) = #["red", "green", "blue", ""]`
- `#["red", "green", "blue"].eraseP (·.length % 2 == 0) = #["red", "green"]`
- `#["red", "green", "blue"].eraseP (fun _ => true) = #["green", "blue"]`
- `(#[] : Array String).eraseP (fun _ => true) = #[]`

def

```
Array.eraseIdx.{u} {α : Type u} (xs : Array α) (i : Nat)
  (h : i < xs.size := by get_elem_tactic) : Array α
```

Removes the element at a given index from an array without a run-time bounds check.

This function takes worst-case `O(n)` time because it back-shifts all elements at positions greater than i.

Examples:

- `#["apple", "pear", "orange"].eraseIdx 0 = #["pear", "orange"]`
- `#["apple", "pear", "orange"].eraseIdx 1 = #["apple", "orange"]`
- `#["apple", "pear", "orange"].eraseIdx 2 = #["apple", "pear"]`

def

```
Array.eraseIdx!.{u} {α : Type u} (xs : Array α) (i : Nat) : Array α
```

Removes the element at a given index from an array. Panics if the index is out of bounds.

This function takes worst-case `O(n)` time because it back-shifts all elements at positions greater than i.

def

```
Array.eraseIdxIfInBounds.{u} {α : Type u} (xs : Array α) (i : Nat)
:
  Array α
```

Removes the element at a given index from an array. Does nothing if the index is out of bounds.

This function takes worst-case `O(n)` time because it back-shifts all elements at positions greater than `i`.

Examples:

- `#["apple", "pear", "orange"].eraseIdxIfInBounds 0 = #["pear", "orange"]`
- `#["apple", "pear", "orange"].eraseIdxIfInBounds 1 = #["apple", "orange"]`
- `#["apple", "pear", "orange"].eraseIdxIfInBounds 2 = #["apple", "pear"]`
- `#["apple", "pear", "orange"].eraseIdxIfInBounds 3 = #["apple", "pear", "orange"]`
- `#["apple", "pear", "orange"].eraseIdxIfInBounds 5 = #["apple", "pear", "orange"]`

def

```
Array.eraseReps.{u_1} {α : Type u_1} [BEq α] (as : Array α) : Array
α
```

Erases repeated elements, keeping the first element of each run.

`O(|as|)`.

Example:

- `#[1, 3, 2, 2, 2, 3, 3, 5].eraseReps = #[1, 3, 2, 3, 5]`

def

```
Array.swap.{u} {α : Type u} (xs : Array α) (i j : Nat)
  (hi : i ≤ xs.size := by get_elem_tactic)
  (hj : j ≤ xs.size := by get_elem_tactic) : Array α
```

Swaps two elements of an array. The modification is performed in-place when the reference to the array is unique.

Examples:

- `#["red", "green", "blue", "brown"].swap 0 3 = #["brown", "green", "blue", "red"]`
- `#["red", "green", "blue", "brown"].swap 0 2 = #["blue", "green", "red", "brown"]`
- `#["red", "green", "blue", "brown"].swap 1 2 = #["red", "blue", "green", "brown"]`
- `#["red", "green", "blue", "brown"].swap 3 0 = #["brown", "green", "blue", "red"]`

def

```
Array.swapIfInBounds.{u} {α : Type u} (xs : Array α) (i j : Nat) :
  Array α
```

Swaps two elements of an array, returning the array unchanged if either index is out of bounds. The modification is performed in-place when the reference to the array is unique.

Examples:

- `#["red", "green", "blue", "brown"].swapIfInBounds 0 3 = #["brown", "green", "blue", "red"]`
- `#["red", "green", "blue", "brown"].swapIfInBounds 0 2 = #["blue", "green", "red", "brown"]`
- `#["red", "green", "blue", "brown"].swapIfInBounds 1 2 = #["red", "blue", "green", "brown"]`
- `#["red", "green", "blue", "brown"].swapIfInBounds 0 4 = #["red", "green", "blue", "brown"]`
- `#["red", "green", "blue", "brown"].swapIfInBounds 9 2 = #["red", "green", "blue", "brown"]`

def

```
Array.swapAt.{u} {α : Type u} (xs : Array α) (i : Nat) (v : α)
  (hi : i ≤ xs.size := by get_elem_tactic) : α × Array α
```

Swaps a new element with the element at the given index.

Returns the value formerly found at i, paired with an array in which the value at i has been replaced with v.

Examples:

- `#["spinach", "broccoli", "carrot"].swapAt 1 "pepper" = ("broccoli", #["spinach", "pepper", "carrot"])`

- `#["spinach", "broccoli", "carrot"].swapAt 2 "pepper" = ("carrot", #["spinach", "broccoli", "pepper"])`

def

```
Array.swapAt!.{u} {α : Type u} (xs : Array α) (i : Nat) (v : α) :
  α × Array α
```

Swaps a new element with the element at the given index. Panics if the index is out of bounds.

Returns the value formerly found at i, paired with an array in which the value at i has been replaced with v.

Examples:

- `#["spinach", "broccoli", "carrot"].swapAt! 1 "pepper" = (#["spinach", "pepper", "carrot"], "broccoli")`
- `#["spinach", "broccoli", "carrot"].swapAt! 2 "pepper" = (#["spinach", "broccoli", "pepper"], "carrot")`

def

```
Array.replace.{u} {α : Type u} [BEq α] (xs : Array α) (a b : α) :
  Array α
```

Replaces the first occurrence of a with b in an array. The modification is performed in-place when the reference to the array is unique. Returns the array unmodified when a is not present.

Examples:

- `#[1, 2, 3, 2, 1].replace 2 5 = #[1, 5, 3, 2, 1]`
- `#[1, 2, 3, 2, 1].replace 0 5 = #[1, 2, 3, 2, 1]`
- `#[].replace 2 5 = #[]`

def

```
Array.set.{u_1} {α : Type u_1} (xs : Array α) (i : Nat) (v : α)
  (h : i ≤ xs.size := by get_elem_tactic) : Array α
```

Replaces the element at a given index in an array.

No bounds check is performed, but the function requires a proof that the index is in bounds. This proof can usually be omitted, and will be synthesized automatically.

The array is modified in-place if there are no other references to it.

Examples:

- `#[0, 1, 2].set 1 5 = #[0, 5, 2]`
- `#["orange", "apple"].set 1 "grape" = #["orange", "grape"]`

def

```
Array.set!.{u_1} {α : Type u_1} (xs : Array α) (i : Nat) (v : α) :
  Array α
```

Set an element in an array, or panic if the index is out of bounds.

This will perform the update destructively provided that a has a reference count of 1 when called.

def

```
Array.setIfInBounds.{u_1} {α : Type u_1} (xs : Array α) (i : Nat)
  (v : α) : Array α
```

Replaces the element at the provided index in an array. The array is returned unmodified if the index is out of bounds.

The array is modified in-place if there are no other references to it.

Examples:

- `#[0, 1, 2].setIfInBounds 1 5 = #[0, 5, 2]`
- `#["orange", "apple"].setIfInBounds 1 "grape" = #["orange", "grape"]`
- `#["orange", "apple"].setIfInBounds 5 "grape" = #["orange", "apple"]`

def

```
Array.uset.{u} {α : Type u} (xs : Array α) (i : USize) (v : α)
  (h : i.toNat ≤ xs.size) : Array α
```

Low-level modification operator which is as fast as a C array write. The modification is performed in-place when the reference to the array is unique.

This avoids overhead due to unboxing a `Nat` used as an index.

def

```
Array.modify.{u} {α : Type u} (xs : Array α) (i : Nat) (f : α → α)
:
  Array α
```

Replaces the element at the given index, if it exists, with the result of applying *f* to it. If the index is invalid, the array is returned unmodified.

Examples:

- `#[1, 2, 3].modify 0 (· * 10) = #[10, 2, 3]`
- `#[1, 2, 3].modify 2 (· * 10) = #[1, 2, 30]`
- `#[1, 2, 3].modify 3 (· * 10) = #[1, 2, 3]`

def

```
Array.modifyM.{u, u_1} {α : Type u} {m : Type u → Type u_1} [Monad m]
  (xs : Array α) (i : Nat) (f : α → m α) : m (Array α)
```

Replaces the element at the given index, if it exists, with the result of applying the monadic function *f* to it. If the index is invalid, the array is returned unmodified and *f* is not called.

Examples:

```
#eval #[1, 2, 3, 4].modifyM 2 fun x => do
  IO.println s!"It was {x}"
  return x * 10
```

```
It was 3
```

```
#[1, 2, 30, 4]
```

```
#eval #[1, 2, 3, 4].modifyM 6 fun x => do
  IO.println s!"It was {x}"
  return x * 10
```

```
#[1, 2, 3, 4]
```

def

```
Array.modifyOp.{u} {α : Type u} (xs : Array α) (idx : Nat) (f : α → α) :
  Array α
```

Replaces the element at the given index, if it exists, with the result of applying *f* to it. If the index is invalid, the array is returned unmodified.

Examples:

- `#[1, 2, 3].modifyOp 0 (· * 10) = #[10, 2, 3]`
- `#[1, 2, 3].modifyOp 2 (· * 10) = #[1, 2, 30]`

- `#[1, 2, 3].modifyOp 3 (· * 10) = #[1, 2, 3]`

def

```
Array.insertIdx.{u} {α : Type u} (as : Array α) (i : Nat) (a : α) :
  autoParam (i ≤ as.size) _auto✝ → Array α
```

Inserts an element into an array at the specified index. If the index is greater than the size of the array, then the array is returned unmodified.

In other words, the new element is inserted into the array as after the first i elements of as.

This function takes worst case $O(n)$ time because it has to swap the inserted element into place.

Examples:

- `#["tues", "thur", "sat"].insertIdx 1 "wed" = #["tues", "wed", "thur", "sat"]`
- `#["tues", "thur", "sat"].insertIdx 2 "wed" = #["tues", "thur", "wed", "sat"]`
- `#["tues", "thur", "sat"].insertIdx 3 "wed" = #["tues", "thur", "sat", "wed"]`

def

```
Array.insertIdx!.{u} {α : Type u} (as : Array α) (i : Nat) (a : α)
:
  Array α
```

Inserts an element into an array at the specified index. Panics if the index is greater than the size of the array.

In other words, the new element is inserted into the array as after the first i elements of as.

This function takes worst case $O(n)$ time because it has to swap the inserted element into place. `Array.insertIdx` and `Array.insertIdxIfInBounds` are safer alternatives.

Examples:

- `#["tues", "thur", "sat"].insertIdx! 1 "wed" = #["tues", "wed", "thur", "sat"]`
- `#["tues", "thur", "sat"].insertIdx! 2 "wed" = #["tues", "thur", "wed", "sat"]`
- `#["tues", "thur", "sat"].insertIdx! 3 "wed" = #["tues", "thur", "sat", "wed"]`

def

```
Array.insertIdxIfInBounds.{u} {α : Type u} (as : Array α) (i : Nat)
  (a : α) : Array α
```

Inserts an element into an array at the specified index. The array is returned unmodified if the index is greater than the size of the array.

In other words, the new element is inserted into the array `as` after the first `i` elements of `as`.

This function takes worst case `O(n)` time because it has to swap the inserted element into place.

Examples:

- `#["tues", "thur", "sat"].insertIdxIfInBounds 1 "wed" = #["tues", "wed", "thur", "sat"]`
- `#["tues", "thur", "sat"].insertIdxIfInBounds 2 "wed" = #["tues", "thur", "wed", "sat"]`
- `#["tues", "thur", "sat"].insertIdxIfInBounds 3 "wed" = #["tues", "thur", "sat", "wed"]`
- `#["tues", "thur", "sat"].insertIdxIfInBounds 4 "wed" = #["tues", "thur", "sat"]`

def

```
Array.reverse.{u} {α : Type u} (as : Array α) : Array α
```

Reverses an array by repeatedly swapping elements.

The original array is modified in place if there are no other references to it.

Examples:

- `(#[] : Array Nat).reverse = #[]`
- `#[0, 1].reverse = #[1, 0]`
- `#[0, 1, 2].reverse = #[2, 1, 0]`

def

```
Array.take.{u} {α : Type u} (xs : Array α) (i : Nat) : Array α
```

Returns a new array that contains the first `i` elements of `xs`. If `xs` has fewer than `i` elements, the new array contains all the elements of `xs`.

The returned array is always a new array, even if it contains the same elements as the input array.

Examples:

- `#["red", "green", "blue"].take 1 = #["red"]`
- `#["red", "green", "blue"].take 2 = #["red", "green"]`
- `#["red", "green", "blue"].take 5 = #["red", "green", "blue"]`

def

```
Array.takeWhile.{u} {α : Type u} (p : α → Bool) (as : Array α) :
Array α
```

Returns a new array that contains the longest prefix of elements that satisfy the predicate *p* from an array.

Examples:

- `#[0, 1, 2, 3, 2, 1].takeWhile (· < 2) = #[0, 1]`
- `#[0, 1, 2, 3, 2, 1].takeWhile (· < 20) = #[0, 1, 2, 3, 2, 1]`
- `#[0, 1, 2, 3, 2, 1].takeWhile (· < 0) = #[]`

def

```
Array.drop.{u} {α : Type u} (xs : Array α) (i : Nat) : Array α
```

Removes the first i elements of xs. If xs has fewer than i elements, the new array is empty.

The returned array is always a new array, even if it contains the same elements as the input array.

Examples:

- `#["red", "green", "blue"].drop 1 = #["green", "blue"]`
- `#["red", "green", "blue"].drop 2 = #["blue"]`
- `#["red", "green", "blue"].drop 5 = #[]`

def

```
Array.shrink.{u} {α : Type u} (xs : Array α) (n : Nat) : Array α
```

Returns the first *n* elements of an array. The resulting array is produced by repeatedly calling `Array.pop`. If *n* is greater than the size of the array, it is returned unmodified.

If the reference to the array is unique, then this function uses in-place modification.

Examples:

- `#[0, 1, 2, 3, 4].shrink 2 = #[0, 1]`
- `#[0, 1, 2, 3, 4].shrink 0 = #[]`
- `#[0, 1, 2, 3, 4].shrink 10 = #[0, 1, 2, 3, 4]`

def

```
Array.flatten.{u} {α : Type u} (xss : Array (Array α)) : Array α
```

Appends the contents of array of arrays into a single array. The resulting array contains the same elements as the nested arrays in the same order.

Examples:

- `#[#[5], #[4], #[3, 2]].flatten = #[5, 4, 3, 2]`
- `#[#[0, 1], #[], #[2], #[1, 0, 1]].flatten = #[0, 1, 2, 1, 0, 1]`
- `(#[] : Array Nat).flatten = #[]`

def

```
Array.getEvenElems.{u} {α : Type u} (as : Array α) : Array α
```

Returns a new array that contains the elements at even indices in `as`, starting with the element at index `0`.

Examples:

- `#[0, 1, 2, 3, 4].getEvenElems = #[0, 2, 4]`
- `#[1, 2, 3, 4].getEvenElems = #[1, 3]`
- `#["red", "green", "blue"].getEvenElems = #["red", "blue"]`
- `(#[] : Array String).getEvenElems = #[]`

19.16.4.7. Sorted Arrays

def

```
Array.qsort.{u_1} {α : Type u_1} (as : Array α)
  (lt : α → α → Bool := by exact (· < ·)) (lo : Nat := 0)
  (hi : Nat := as.size - 1) : Array α
```

In-place quicksort.

`qsort as lt lo hi` sorts the subarray `as[lo...=hi]` in-place using `lt` to compare elements.

def

```
Array.qsortOrd.{u_1} {α : Type u_1} [ord : Ord α] (xs : Array α) :
  Array α
```

Sort an array using `compare` to compare elements.

def

```
Array.insertionSort.{u_1} {α : Type u_1} (xs : Array α)
  (lt : α → α → Bool := by exact (· < ·)) : Array α
```

Sorts an array using insertion sort.

The optional parameter `lt` specifies an ordering predicate. It defaults to `LT.lt`, which must be decidable to be used for sorting.

def

```
Array.binInsert.{u} {α : Type u} (lt : α → α → Bool) (as : Array α)
  (k : α) : Array α
```

Inserts an element into a sorted array such that the resulting array is sorted. If the element is already present in the array, it is not inserted.

The ordering predicate `lt` should be a total order on elements, and the array `as` should be sorted with respect to `lt`.

`Array.binInsertM` is a more general operator that provides greater control over the handling of duplicate elements in addition to running in a monad.

Examples:

- `#[0, 1, 3, 5].binInsert (· < ·) 2 = #[0, 1, 2, 3, 5]`
- `#[0, 1, 3, 5].binInsert (· < ·) 1 = #[0, 1, 3, 5]`
- `#[].binInsert (· < ·) 1 = #[1]`

def

```
Array.binInsertM.{u, v} {α : Type u} {m : Type u → Type v} [Monad
m]
  (lt : α → α → Bool) (merge : α → m α) (add : Unit → m α)
  (as : Array α) (k : α) : m (Array α)
```

Inserts an element *k* into a sorted array `as` such that the resulting array is sorted.

The ordering predicate `lt` should be a total order on elements, and the array `as` should be sorted with respect to `lt`.

If an element that `lt` equates to *k* is already present in `as`, then `merge` is applied to the existing element to determine the value of that position in the resulting array. If no element equal to *k* is present, then *add* is used to determine the value to be inserted.

def

```
Array.binSearch {α : Type} (as : Array α) (k : α) (lt : α → α →
Bool)
  (lo : Nat := 0) (hi : Nat := as.size - 1) : Option α
```

Binary search for an element equivalent to *k* in the sorted array `as`. Returns the element from the array, if it is found, or none otherwise.

The array `as` must be sorted according to the comparison operator `lt`, which should be a total order.

The optional parameters `lo` and `hi` determine the region of the array indices to be searched. Both are inclusive, and default to searching the entire array.

def

```
Array.binSearchContains {α : Type} (as : Array α) (k : α)
  (lt : α → α → Bool) (lo : Nat := 0) (hi : Nat := as.size - 1) :
Bool
```

Binary search for an element equivalent to *k* in the sorted array `as`. Returns `true` if the element is found, or `false` otherwise.

The array `as` must be sorted according to the comparison operator `lt`, which should be a total order.

The optional parameters `lo` and `hi` determine the region of the array indices to be searched. Both are inclusive, and default to searching the entire array.

19.16.4.8. Iteration

def

```
Array.foldr.{u, v} {α : Type u} {β : Type v} (f : α → β → β) (init
: β)
  (as : Array α) (start : Nat := as.size) (stop : Nat := 0) : β
```

Folds a function over an array from the right, accumulating a value starting with `init`. The accumulated value is combined with the each element of the array in reverse order, using `f`.

The optional parameters `start` and `stop` control the region of the array to be folded. Folding proceeds from `start` (exclusive) to `stop` (inclusive), so no folding occurs unless `start > stop`. By default, the entire array is used.

Examples:

- `#[a, b, c].foldr f init = f a (f b (f c init))`
- `#[1, 2, 3].foldr (toString · ++ ·) "" = "123"`
- `#[1, 2, 3].foldr (s!"({·} {·})") "!" = "(1 (2 (3 !)))"`

def

```
Array.foldrM.{u, v, w} {α : Type u} {β : Type v} {m : Type v → Type w}
  [Monad m] (f : α → β → m β) (init : β) (as : Array α)
  (start : Nat := as.size) (stop : Nat := 0) : m β
```

Folds a monadic function over an array from the right, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in reverse order, using `f`.

The optional parameters `start` and `stop` control the region of the array to be folded. Folding proceeds from `start` (exclusive) to `stop` (inclusive), so no folding occurs unless `start > stop`. By default, the entire array is folded.

Examples:

```
example [Monad m] (f : α → β → m β) :
  Array.foldrM (m := m) f x₀ #[a, b, c] = (do
    let x₁ ← f c x₀
    let x₂ ← f b x₁
    let x₃ ← f a x₂
    pure x₃)
  := by rfl

example [Monad m] (f : α → β → m β) :
  Array.foldrM (m := m) f x₀ #[a, b, c] (start := 2) = (do
    let x₁ ← f b x₀
    let x₂ ← f a x₁
    pure x₂)
  := by rfl
```

def

```
Array.foldl.{u, v} {α : Type u} {β : Type v} (f : β → α → β) (init : β)
  (as : Array α) (start : Nat := 0) (stop : Nat := as.size) : β
```

Folds a function over an array from the left, accumulating a value starting with `init`. The accumulated value is combined with the each element of the array in order, using `f`.

The optional parameters `start` and `stop` control the region of the array to be folded. Folding proceeds from `start` (inclusive) to `stop` (exclusive), so no folding occurs unless `start < stop`. By default, the entire array is used.

Examples:

- `#[a, b, c].foldl f z = f (f (f z a) b) c`
- `#[1, 2, 3].foldl (· ++ toString ·) "" = "123"`
- `#[1, 2, 3].foldl (s!"({·} {·})") "" = "((( 1) 2) 3)"`

def

```
Array.foldlM.{u, v, w} {α : Type u} {β : Type v} {m : Type v → Type w}
  [Monad m] (f : β → α → m β) (init : β) (as : Array α)
  (start : Nat := 0) (stop : Nat := as.size) : m β
```

Folds a monadic function over a list from the left, accumulating a value starting with `init`. The accumulated value is combined with the each element of the list in order, using `f`.

The optional parameters `start` and `stop` control the region of the array to be folded. Folding proceeds from `start` (inclusive) to `stop` (exclusive), so no folding occurs unless `start < stop`. By default, the entire array is folded.

Examples:

```
example [Monad m] (f : α → β → m α) :
    Array.foldlM (m := m) f x₀ #[a, b, c] = (do
      let x₁ ← f x₀ a
      let x₂ ← f x₁ b
      let x₃ ← f x₂ c
      pure x₃)
  := by rfl

example [Monad m] (f : α → β → m α) :
    Array.foldlM (m := m) f x₀ #[a, b, c] (start := 1) = (do
      let x₁ ← f x₀ b
      let x₂ ← f x₁ c
      pure x₂)
  := by rfl
```

def

```
Array.forM.{u, v, w} {α : Type u} {m : Type v → Type w} [Monad m]
  (f : α → m PUnit) (as : Array α) (start : Nat := 0)
  (stop : Nat := as.size) : m PUnit
```

Applies the monadic action `f` to each element of an array, in order.

The optional parameters *start* and *stop* control the region of the array to which *f* should be applied. Iteration proceeds from *start* (inclusive) to *stop* (exclusive), so *f* is not invoked unless *start* < *stop*. By default, the entire array is used.

def

```
Array.forRevM.{u, v, w} {α : Type u} {m : Type v → Type w} [Monad
m]
  (f : α → m PUnit) (as : Array α) (start : Nat := as.size)
  (stop : Nat := 0) : m PUnit
```

Applies the monadic action *f* to each element of an array from right to left, in reverse order.

The optional parameters *start* and *stop* control the region of the array to which *f* should be applied. Iteration proceeds from *start* (exclusive) to *stop* (inclusive), so no *f* is not invoked unless *start* > *stop*. By default, the entire array is used.

def

```
Array.firstM.{u, v, w} {β : Type v} {α : Type u} {m : Type v → Type
w}
  [Alternative m] (f : α → m β) (as : Array α) : m β
```

Maps *f* over the array and collects the results with <|>. The result for the end of the array is `failure`.

Examples:

- `#[[], [1, 2], [], [2]].firstM List.head? = some 1`
- `#[[], [], []].firstM List.head? = none`
- `#[].firstM List.head? = none`

def

```
Array.sum.{u_1} {α : Type u_1} [Add α] [Zero α] : Array α → α
```

Computes the sum of the elements of an array.

Examples:

- `#[a, b, c].sum = a + (b + (c + 0))`
- `#[1, 2, 5].sum = 8`

19.16.4.9. Transformation

def

```
Array.map.{u, v} {α : Type u} {β : Type v} (f : α → β) (as : Array
α) :
  Array β
```

Applies a function to each element of the array, returning the resulting array of values.

Examples:

- `#[a, b, c].map f = #[f a, f b, f c]`
- `#[].map Nat.succ = #[]`
- `#["one", "two", "three"].map (·.length) = #[3, 3, 5]`
- `#["one", "two", "three"].map (·.reverse) = #["eno", "owt", "eerht"]`

def

```
Array.mapMono.{u_1} {α : Type u_1} (as : Array α) (f : α → α) :
Array α
```

Applies a function to each element of an array, returning the array of results. The function is monomorphic: it is required to return a value of the same type. The internal implementation uses pointer equality, and does not allocate a new array if the result of each function call is pointer-equal to its argument.

def

```
Array.mapM.{u, v, w} {α : Type u} {β : Type v} {m : Type v → Type
w}
  [Monad m] (f : α → m β) (as : Array α) : m (Array β)
```

Applies the monadic action `f` to every element in the array, left-to-right, and returns the array of results.

def

```
Array.mapM'.{u_1, u_2, u_3} {m : Type u_1 → Type u_2} {α : Type
u_3}
  {β : Type u_1} [Monad m] (f : α → m β) (as : Array α) :
  m { bs // bs.size = as.size }
```

Applies the monadic action f to every element in the array, left-to-right, and returns the array of results. Furthermore, the resulting array's type guarantees that it contains the same number of elements as the input array.

def

```
Array.mapMonoM.{u_1, u_2} {m : Type u_1 → Type u_2} {α : Type u_1}
  [Monad m] (as : Array α) (f : α → m α) : m (Array α)
```

Applies a monadic function to each element of an array, returning the array of results. The function is monomorphic: it is required to return a value of the same type. The internal implementation uses pointer equality, and does not allocate a new array if the result of each function call is pointer-equal to its argument.

def

```
Array.mapIdx.{u, v} {α : Type u} {β : Type v} (f : Nat → α → β)
  (as : Array α) : Array β
```

Applies a function to each element of the array along with the index at which that element is found, returning the array of results.

Array.mapFinIdx is a variant that additionally provides the function with a proof that the index is valid.

def

```
Array.mapIdxM.{u, v, w} {α : Type u} {β : Type v} {m : Type v →
Type w}
  [Monad m] (f : Nat → α → m β) (as : Array α) : m (Array β)
```

Applies the monadic action f to every element in the array, along with the element's index, from left to right. Returns the array of results.

def

```
Array.mapFinIdx.{u, v} {α : Type u} {β : Type v} (as : Array α)
  (f : (i : Nat) → α → i ≤ as.size → β) : Array β
```

Applies a function to each element of the array along with the index at which that element is found, returning the array of results. In addition to the index, the function is also provided with a proof that the index is valid.

Array.mapIdx is a variant that does not provide the function with evidence that the index is valid.

def

```
Array.mapFinIdxM.{u, v, w} {α : Type u} {β : Type v}
  {m : Type v → Type w} [Monad m] (as : Array α)
  (f : (i : Nat) → α → i < as.size → m β) : m (Array β)
```

Applies the monadic action `f` to every element in the array, along with the element's index and a proof that the index is in bounds, from left to right. Returns the array of results.

def

```
Array.flatMap.{u, u_1} {α : Type u} {β : Type u_1} (f : α → Array
β)
  (as : Array α) : Array β
```

Applies a function that returns an array to each element of an array. The resulting arrays are appended.

Examples:

- `#[2, 3, 2].flatMap Array.range = #[0, 1, 0, 1, 2, 0, 1]`
- `#[['a', 'b'], ['c', 'd', 'e']].flatMap List.toArray = #['a', 'b', 'c', 'd', 'e']`

def

```
Array.flatMapM.{u, u_1, u_2} {α : Type u} {m : Type u_1 → Type u_2}
  {β : Type u_1} [Monad m] (f : α → m (Array β)) (as : Array α) :
  m (Array β)
```

Applies a monadic function that returns an array to each element of an array, from left to right. The resulting arrays are appended.

def

```
Array.zip.{u, u_1} {α : Type u} {β : Type u_1} (as : Array α)
  (bs : Array β) : Array (α × β)
```

Combines two arrays into an array of pairs in which the first and second components are the corresponding elements of each input array. The resulting array is the length of the shorter of the input arrays.

Examples:

- `#["Mon", "Tue", "Wed"].zip #[1, 2, 3] = #[("Mon", 1), ("Tue", 2), ("Wed", 3)]`
- `#["Mon", "Tue", "Wed"].zip #[1, 2] = #[("Mon", 1), ("Tue", 2)]`

- `#[x₁, x₂, x₃].zip #[y₁, y₂, y₃, y₄] = #[(x₁, y₁), (x₂, y₂), (x₃, y₃)]`

def

```
Array.zipWith.{u, u_1, u_2} {α : Type u} {β : Type u_1} {γ : Type u_2}
  (f : α → β → γ) (as : Array α) (bs : Array β) : Array γ
```

Applies a function to the corresponding elements of two arrays, stopping at the end of the shorter array.

Examples:

- `#[1, 2].zipWith (· + ·) #[5, 6] = #[6, 8]`
- `#[1, 2, 3].zipWith (· + ·) #[5, 6, 10] = #[6, 8, 13]`
- `#[].zipWith (· + ·) #[5, 6] = #[]`
- `#[x₁, x₂, x₃].zipWith f #[y₁, y₂, y₃, y₄] = #[f x₁ y₁, f x₂ y₂, f x₃ y₃]`

def

```
Array.zipWithAll.{u, u_1, u_2} {α : Type u} {β : Type u_1}
  {γ : Type u_2} (f : Option α → Option β → γ) (as : Array α)
  (bs : Array β) : Array γ
```

Applies a function to the corresponding elements of both arrays, stopping when there are no more elements in either array. If one array is shorter than the other, the function is passed `none` for the missing elements.

Examples:

- `#[1, 6].zipWithAll min #[5, 2] = #[some 1, some 2]`
- `#[1, 2, 3].zipWithAll Prod.mk #[5, 6] = #[(some 1, some 5), (some 2, some 6), (some 3, none)]`
- `#[x₁, x₂].zipWithAll f #[y] = #[f (some x₁) (some y), f (some x₂) none]`

def

```
Array.zipIdx.{u} {α : Type u} (xs : Array α) (start : Nat := 0) :
  Array (α × Nat)
```

Pairs each element of an array with its index, optionally starting from an index other than `0`.

Examples:

- `#[a, b, c].zipIdx = #[(a, 0), (b, 1), (c, 2)]`
- `#[a, b, c].zipIdx 5 = #[(a, 5), (b, 6), (c, 7)]`

def

```
Array.unzip.{u, u_1} {α : Type u} {β : Type u_1} (as : Array (α × β)) :
  Array α × Array β
```

Separates an array of pairs into two arrays that contain the respective first and second components.

Examples:

- `#[("Monday", 1), ("Tuesday", 2)].unzip = (#["Monday", "Tuesday"], #[1, 2])`
- `#[(x₁, y₁), (x₂, y₂), (x₃, y₃)].unzip = (#[x₁, x₂, x₃], #[y₁, y₂, y₃])`
- `(#[] : Array (Nat × String)).unzip = ((#[], #[]) : List Nat × List String)`

19.16.4.10. Filtering

def

```
Array.filter.{u} {α : Type u} (p : α → Bool) (as : Array α)
  (start : Nat := 0) (stop : Nat := as.size) : Array α
```

Returns the array of elements in `as` for which `p` returns `true`.

Only elements from `start` (inclusive) to `stop` (exclusive) are considered. Elements outside that range are discarded. By default, the entire array is considered.

Examples:

- `#[1, 2, 5, 2, 7, 7].filter (· > 2) = #[5, 7, 7]`
- `#[1, 2, 5, 2, 7, 7].filter (fun _ => false) = #[]`
- `#[1, 2, 5, 2, 7, 7].filter (fun _ => true) = #[1, 2, 5, 2, 7, 7]`
- `#[1, 2, 5, 2, 7, 7].filter (· > 2) (start := 3) = #[7, 7]`
- `#[1, 2, 5, 2, 7, 7].filter (fun _ => true) (start := 3) = #[2, 7, 7]`
- `#[1, 2, 5, 2, 7, 7].filter (fun _ => true) (stop := 3) = #[1, 2, 5]`

def

```
Array.filterM.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  (p : α → m Bool) (as : Array α) (start : Nat := 0)
  (stop : Nat := as.size) : m (Array α)
```

Applies the monadic predicate *p* to every element in the array, in order from left to right, and returns the array of elements for which *p* returns `true`.

Only elements from `start` (inclusive) to `stop` (exclusive) are considered. Elements outside that range are discarded. By default, the entire array is checked.

Example:

```
#eval #[1, 2, 5, 2, 7, 7].filterM fun x => do
  IO.println s!"Checking {x}"
  return x < 3
```

```
Checking 1
Checking 2
Checking 5
Checking 2
Checking 7
Checking 7

#[1, 2, 2]
```

def

```
Array.filterRevM.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  (p : α → m Bool) (as : Array α) (start : Nat := as.size)
  (stop : Nat := 0) : m (Array α)
```

Applies the monadic predicate *p* on every element in the array in reverse order, from right to left, and returns those elements for which *p* returns `true`. The elements of the returned list are in the same order as in the input list.

Only elements from `start` (exclusive) to `stop` (inclusive) are considered. Elements outside that range are discarded. Because the array is examined in reverse order, elements are only examined when `start > stop`. By default, the entire array is considered.

Example:

```
#eval #[1, 2, 5, 2, 7, 7].filterRevM fun x => do
  IO.println s!"Checking {x}"
  return x < 3
```

```
Checking 7
Checking 7
Checking 2
Checking 5
```

```
Checking 2
Checking 1

#[1, 2, 2]
```

def

```
Array.filterMap.{u, u_1} {α : Type u} {β : Type u_1} (f : α → Option β)
  (as : Array α) (start : Nat := 0) (stop : Nat := as.size) : Array β
```

Applies a function that returns an `Option` to each element of an array, collecting the non-`none` values.

Example:

```
#eval #[1, 2, 5, 2, 7, 7].filterMap fun x =>
  if x > 2 then some (2 * x) else none

#[10, 14, 14]
```

def

```
Array.filterMapM.{u, u_1, u_2} {α : Type u} {m : Type u_1 → Type u_2}
  {β : Type u_1} [Monad m] (f : α → m (Option β)) (as : Array α)
  (start : Nat := 0) (stop : Nat := as.size) : m (Array β)
```

Applies a monadic function that returns an `Option` to each element of an array, collecting the non-`none` values.

Only elements from `start` (inclusive) to `stop` (exclusive) are considered. Elements outside that range are discarded. By default, the entire array is considered.

Example:

```
#eval #[1, 2, 5, 2, 7, 7].filterMapM fun x => do
  IO.println s!"Examining {x}"
  if x > 2 then return some (2 * x)
  else return none

Examining 1
Examining 2
Examining 5
Examining 2
Examining 7
Examining 7

#[10, 14, 14]
```

def

```
Array.filterSepElems (a : Array Lean.Syntax) (p : Lean.Syntax →
Bool) :
  Array Lean.Syntax
```

Filters an array of syntax, treating every other element as a separator rather than an element to test with the predicate *p*. The resulting array contains the tested elements for which *p* returns `true`, separated by the corresponding separator elements.

def

```
Array.filterSepElemsM {m : Type → Type} [Monad m]
  (a : Array Lean.Syntax) (p : Lean.Syntax → m Bool) :
  m (Array Lean.Syntax)
```

Filters an array of syntax, treating every other element as a separator rather than an element to test with the monadic predicate *p*. The resulting array contains the tested elements for which *p* returns `true`, separated by the corresponding separator elements.

19.16.4.11. Partitioning

def

```
Array.partition.{u} {α : Type u} (p : α → Bool) (as : Array α) :
  Array α × Array α
```

Returns a pair of arrays that together contain all the elements of *as*. The first array contains those elements for which *p* returns `true`, and the second contains those for which *p* returns `false`.

as`.partition` *p* is equivalent to (*as*`.filter` *p*, *as*`.filter (not ∘` *p*`)`), but it is more efficient since it only has to do one pass over the array.

Examples:

- `#[1, 2, 5, 2, 7, 7].partition (· > 2) = (#[5, 7, 7], #[1, 2, 2])`
- `#[1, 2, 5, 2, 7, 7].partition (fun _ => false) = (#[], #[1, 2, 5, 2, 7, 7])`
- `#[1, 2, 5, 2, 7, 7].partition (fun _ => true) = (#[1, 2, 5, 2, 7, 7], #[])`

def

```
Array.groupByKey.{u, v} {α : Type u} {β : Type v} [BEq α] [Hashable
α]
  (key : β → α) (xs : Array β) : Std.HashMap α (Array β)
```

Groups the elements of an array *xs* according to the function *key*, returning a hash map in which each group is associated with its key. Groups preserve the relative order of elements in *xs*.

Example:

```
#eval #[0, 1, 2, 3, 4, 5, 6].groupByKey (· % 2)
```

```
Std.HashMap.ofList [(0, #[0, 2, 4, 6]), (1, #[1, 3, 5])]
```

19.16.4.12. Element Predicates

def

```
Array.contains.{u} {α : Type u} [BEq α] (as : Array α) (a : α) : Bool
```

Checks whether *a* is an element of *as*, using == to compare elements.

`Array.elem` is a synonym that takes the element before the array.

Examples:

- `#[1, 4, 2, 3, 3, 7].contains 3 = true`
- `Array.contains #[1, 4, 2, 3, 3, 7] 5 = false`

def

```
Array.elem.{u} {α : Type u} [BEq α] (a : α) (as : Array α) : Bool
```

Checks whether *a* is an element of *as*, using == to compare elements.

`Array.contains` is a synonym that takes the array before the element.

For verification purposes, `Array.elem` is simplified to `Array.contains`.

Example:

- `Array.elem 3 #[1, 4, 2, 3, 3, 7] = true`
- `Array.elem 5 #[1, 4, 2, 3, 3, 7] = false`

def

```
Array.find?.{u} {α : Type u} (p : α → Bool) (as : Array α) : Option α
```

Returns the first element of the array for which the predicate *p* returns `true`, or `none` if no such element is found.

Examples:

- `#[7, 6, 5, 8, 1, 2, 6].find? (· < 5) = some 1`
- `#[7, 6, 5, 8, 1, 2, 6].find? (· < 1) = none`

def

```
Array.findRev? {α : Type} (p : α → Bool) (as : Array α) : Option α
```

Returns the last element of the array for which the predicate *p* returns `true`, or `none` if no such element is found.

Examples:

- `#[7, 6, 5, 8, 1, 2, 6].findRev? (· < 5) = some 2`
- `#[7, 6, 5, 8, 1, 2, 6].findRev? (· < 1) = none`

def

```
Array.findIdx.{u} {α : Type u} (p : α → Bool) (as : Array α) : Nat
```

Returns the index of the first element for which *p* returns `true`, or the size of the array if there is no such element.

Examples:

- `#[7, 6, 5, 8, 1, 2, 6].findIdx (· < 5) = 4`
- `#[7, 6, 5, 8, 1, 2, 6].findIdx (· < 1) = 7`

def

```
Array.findIdx?.{u} {α : Type u} (p : α → Bool) (as : Array α) :
  Option Nat
```

Returns the index of the first element for which *p* returns `true`, or `none` if there is no such element.

Examples:

- `#[7, 6, 5, 8, 1, 2, 6].findIdx (· < 5) = some 4`
- `#[7, 6, 5, 8, 1, 2, 6].findIdx (· < 1) = none`

def

```
Array.findIdxM?.{u, u_1} {α : Type u} {m : Type → Type u_1} [Monad
m]
  (p : α → m Bool) (as : Array α) : m (Option Nat)
```

Finds the index of the first element of an array for which the monadic predicate *p* returns `true`. Elements are examined in order from left to right, and the search is terminated when an element that satisfies *p* is found. If no such element exists in the array, then `none` is returned.

def

```
Array.findFinIdx?.{u} {α : Type u} (p : α → Bool) (as : Array α) :
  Option (Fin as.size)
```

Returns the index of the first element for which *p* returns `true`, or `none` if there is no such element. The index is returned as a `Fin`, which guarantees that it is in bounds.

Examples:

- `#[7, 6, 5, 8, 1, 2, 6].findFinIdx? (· < 5) = some (4 : Fin 7)`
- `#[7, 6, 5, 8, 1, 2, 6].findFinIdx? (· < 1) = none`

def

```
Array.findM?.{u_1} {m : Type → Type u_1} {α : Type} [Monad m]
  (p : α → m Bool) (as : Array α) : m (Option α)
```

Returns the first element of the array for which the monadic predicate *p* returns `true`, or `none` if no such element is found. Elements of the array are checked in order.

The monad *m* is restricted to `Type → Type` to avoid needing to use `ULift Bool` in *p*'s type.

Example:

```
#eval #[7, 6, 5, 8, 1, 2, 6].findM? fun i => do
  if i < 5 then
    return true
  if i ≤ 6 then
    IO.println s!"Almost! {i}"
  return false
```

```
Almost! 6
Almost! 5
```

```
some 1
```

def

```
Array.findRevM?.{w} {α : Type} {m : Type → Type w} [Monad m]
  (p : α → m Bool) (as : Array α) : m (Option α)
```

Returns the last element of the array for which the monadic predicate *p* returns `true`, or `none` if no such element is found. Elements of the array are checked in reverse, from right to left..

The monad *m* is restricted to `Type → Type` to avoid needing to use `ULift Bool` in *p*'s type.

Example:

```
#eval #[7, 5, 8, 1, 2, 6, 5, 8].findRevM? fun i => do
  if i < 5 then
    return true
  if i ≤ 6 then
    IO.println s!"Almost! {i}"
  return false

Almost! 5
Almost! 6

some 2
```

def

```
Array.findSome?.{u, v} {α : Type u} {β : Type v} (f : α → Option β)
  (as : Array α) : Option β
```

Returns the first non-`none` result of applying the function *f* to each element of the array, in order. Returns `none` if *f* returns `none` for all elements.

Example:

```
#eval #[7, 6, 5, 8, 1, 2, 6].findSome? fun i =>
  if i < 5 then
    some (i * 10)
  else
    none

some 10
```

def

```
Array.findSome!.{u, v} {α : Type u} {β : Type v} [Inhabited β]
  (f : α → Option β) (xs : Array α) : β
```

Returns the first non-`none` result of applying the function *f* to each element of the array, in order. Panics if *f* returns `none` for all elements.

Example:

```
#eval #[7, 6, 5, 8, 1, 2, 6].findSome? fun i =>
  if i < 5 then
    some (i * 10)
  else
    none

some 10
```

def

```
Array.findSomeM?.{u, v, w} {α : Type u} {β : Type v}
  {m : Type v → Type w} [Monad m] (f : α → m (Option β))
  (as : Array α) : m (Option β)
```

Returns the first non-none result of applying the monadic function f to each element of the array, in order. Returns none if f returns none for all elements.

Example:

```
#eval #[7, 6, 5, 8, 1, 2, 6].findSomeM? fun i => do
  if i < 5 then
    return some (i * 10)
  if i ≤ 6 then
    IO.println s!"Almost! {i}"
  return none
```

```
Almost! 6
Almost! 5
```

```
some 10
```

def

```
Array.findSomeRev?.{u, v} {α : Type u} {β : Type v} (f : α → Option β)
  (as : Array α) : Option β
```

Returns the first non-none result of applying f to each element of the array in reverse order, from right to left. Returns none if f returns none for all elements of the array.

Examples:

- `#[7, 6, 5, 8, 1, 2, 6].findSome? (fun x => if x < 5 then some (10 * x) else none) = some 10`
- `#[7, 6, 5, 8, 1, 2, 6].findSome? (fun x => if x < 1 then some (10 * x) else none) = none`

def

```
Array.findSomeRevM?.{u, v, w} {α : Type u} {β : Type v}
  {m : Type v → Type w} [Monad m] (f : α → m (Option β))
  (as : Array α) : m (Option β)
```

Returns the first non-none result of applying the monadic function f to each element of the array in reverse order, from right to left. Once a non-none result is found, no further elements are checked. Returns none if f returns none for all elements of the array.

Examples:

```
#eval #[1, 2, 0, -4, 1].findSomeRevM? (m := Except String) fun x
=> do
  if x = 0 then throw "Zero!"
  else if x < 0 then return (some x)
  else return none

Except.ok (some (-4))

#eval #[1, 2, 0, 4, 1].findSomeRevM? (m := Except String) fun x
=> do
  if x = 0 then throw "Zero!"
  else if x < 0 then return (some x)
  else return none

Except.error "Zero!"
```

def

```
Array.all.{u} {α : Type u} (as : Array α) (p : α → Bool)
  (start : Nat := 0) (stop : Nat := as.size) : Bool
```

Returns `true` if *p* returns `true` for every element of *as*.

Short-circuits upon encountering the first `false`.

The optional parameters *start* and *stop* control the region of the array to be checked. Only the elements with indices from *start* (inclusive) to *stop* (exclusive) are checked. By default, the entire array is checked.

Examples:

- `#[a, b, c].all p = (p a && (p b && p c))`
- `#[2, 4, 6].all (· % 2 = 0) = true`
- `#[2, 4, 5, 6].all (· % 2 = 0) = false`

def

```
Array.allM.{u, w} {α : Type u} {m : Type → Type w} [Monad m]
  (p : α → m Bool) (as : Array α) (start : Nat := 0)
  (stop : Nat := as.size) : m Bool
```

Returns `true` if the monadic predicate *p* returns `true` for every element of *as*.

Short-circuits upon encountering the first `false`. The elements in *as* are examined in order from left to right.

The optional parameters *start* and *stop* control the region of the array to be checked. Only the elements with indices from *start* (inclusive) to *stop* (exclusive) are checked. By default, the entire array is checked.

def

```
Array.any.{u} {α : Type u} (as : Array α) (p : α → Bool)
  (start : Nat := 0) (stop : Nat := as.size) : Bool
```

Returns `true` if *p* returns `true` for any element of *as*.

Short-circuits upon encountering the first `true`.

The optional parameters *start* and *stop* control the region of the array to be checked. Only the elements with indices from *start* (inclusive) to *stop* (exclusive) are checked. By default, the entire array is checked.

Examples:

- `#[2, 4, 6].any (· % 2 = 0) = true`
- `#[2, 4, 6].any (· % 2 = 1) = false`
- `#[2, 4, 5, 6].any (· % 2 = 0) = true`
- `#[2, 4, 5, 6].any (· % 2 = 1) = true`

def

```
Array.anyM.{u, w} {α : Type u} {m : Type → Type w} [Monad m]
  (p : α → m Bool) (as : Array α) (start : Nat := 0)
  (stop : Nat := as.size) : m Bool
```

Returns `true` if the monadic predicate *p* returns `true` for any element of *as*.

Short-circuits upon encountering the first `true`. The elements in *as* are examined in order from left to right.

The optional parameters *start* and *stop* control the region of the array to be checked. Only the elements with indices from *start* (inclusive) to *stop* (exclusive) are checked. By default, the entire array is checked.

def

```
Array.allDiff.{u} {α : Type u} [BEq α] (as : Array α) : Bool
```

Returns `true` if no two elements of *as* are equal according to the == operator.

Examples:

- `#["red", "green", "blue"].allDiff = true`
- `#["red", "green", "red"].allDiff = false`
- `(#[] : Array Nat).allDiff = true`

def

```
Array.isEqv.{u} {α : Type u} (xs ys : Array α) (p : α → α → Bool) :
Bool
```

Returns `true` if `as` and `bs` have the same length and they are pairwise related by `eqv`.

Short-circuits at the first non-related pair of elements.

Examples:

- `#[1, 2, 3].isEqv #[2, 3, 4] (· < ·) = true`
- `#[1, 2, 3].isEqv #[2, 2, 4] (· < ·) = false`
- `#[1, 2, 3].isEqv #[2, 3] (· < ·) = false`

19.16.4.13. Comparisons

def

```
Array.isPrefixOf.{u} {α : Type u} [BEq α] (as bs : Array α) : Bool
```

Return `true` if `as` is a prefix of `bs`, or `false` otherwise.

Examples:

- `#[0, 1, 2].isPrefixOf #[0, 1, 2, 3] = true`
- `#[0, 1, 2].isPrefixOf #[0, 1, 2] = true`
- `#[0, 1, 2].isPrefixOf #[0, 1] = false`
- `#[].isPrefixOf #[0, 1] = true`

def

```
Array.lex.{u_1} {α : Type u_1} [BEq α] (as bs : Array α)
  (lt : α → α → Bool := by exact (· < ·)) : Bool
```

Compares arrays lexicographically with respect to a comparison `lt` on their elements.

Specifically, `Array.lex as bs lt` is true if

- `bs` is larger than `as` and `as` is pairwise equivalent via == to the initial segment of `bs`, or
- there is an index `i` such that `lt as[i] bs[i]`, and for all `j < i`, `as[j] == bs[j]`.

19.16.4.14. Termination Helpers

def

```
Array.attach.{u_1} {α : Type u_1} (xs : Array α) : Array { x // x ∈
xs }
```

"Attaches" the proof that the elements of `xs` are in fact elements of `xs`, producing a new array with the same elements but in the subtype `{ x // x ∈ xs }`.

`O(1)`.

This function is primarily used to allow definitions by well-founded recursion that use higher-order functions (such as `Array.map`) to prove that an value taken from a list is smaller than the list. This allows the well-founded recursion mechanism to prove that the function terminates.

def

```
Array.attachWith.{u_1} {α : Type u_1} (xs : Array α) (P : α → Prop)
  (H : ∀ (x : α), x ∈ xs → P x) : Array { x // P x }
```

"Attaches" individual proofs to an array of values that satisfy a predicate `P`, returning an array of elements in the corresponding subtype `{ x // P x }`.

`O(1)`.

def

```
Array.unattach.{u_1} {α : Type u_1} {p : α → Prop}
  (xs : Array { x // p x }) : Array α
```

Maps an array of terms in a subtype to the corresponding terms in the type by forgetting that they satisfy the predicate.

This is the inverse of `Array.attachWith` and a synonym for `xs.map (·.val)`.

Mostly this should not be needed by users. It is introduced as an intermediate step by lemmas such as `map_subtype`, and is ideally subsequently simplified away by `unattach_attach`.

This function is usually inserted automatically by Lean as an intermediate step while proving termination. It is rarely used explicitly in code. It is introduced as an intermediate step during the elaboration of definitions by well-founded recursion. If this function is encountered in a proof state, the right approach is usually the tactic **`simp`** `[Array.unattach, -Array.map_subtype]`.

def

```
Array.pmap.{u_1, u_2} {α : Type u_1} {β : Type u_2} {P : α → Prop}
  (f : (a : α) → P a → β) (xs : Array α) (H : ∀ (a : α), a ∈ xs → P
a) :
  Array β
```

Maps a partially defined function (defined on those terms of α that satisfy a predicate P) over an array $xs : \texttt{Array}\ \alpha$, given a proof that every element of xs in fact satisfies P.

`Array.pmap`, named for "partial map," is the equivalent of `Array.map` for such partial functions.

19.16.5. Sub-Arrays

def

```
Subarray.{u} (α : Type u) : Type u
```

A region of some underlying array.

A subarray contains an array together with the start and end indices of a region of interest. Subarrays can be used to avoid copying or allocating space, while being more convenient than tracking the bounds by hand. The region of interest consists of every index that is both greater than or equal to `start` and strictly less than **`stop`**.

def

```
Subarray.toArray.{u_1} {α : Type u_1} (s : Subarray α) : Array α
```

Allocates a new array that contains the contents of the subarray.

def

```
Subarray.empty.{u_1} {α : Type u_1} : Subarray α
```

The empty subarray.

This empty subarray is backed by an empty array.

19.16.5.1. Size

def

```
Subarray.size.{u_1} {α : Type u_1} (s : Subarray α) : Nat
```

Computes the size of the subarray.

19.16.5.2. Resizing

def

```
Subarray.drop.{u_1} {α : Type u_1} (arr : Subarray α) (i : Nat) :
  Subarray α
```

Removes the first i elements of the subarray. If there are i or fewer elements, the resulting subarray is empty.

def

```
Subarray.take.{u_1} {α : Type u_1} (arr : Subarray α) (i : Nat) :
  Subarray α
```

Keeps only the first i elements of the subarray. If there are i or fewer elements, the resulting subarray is empty.

def

```
Subarray.popFront.{u_1} {α : Type u_1} (s : Subarray α) : Subarray
α
```

Shrinks the subarray by incrementing its starting index if possible, returning it unchanged if not.

Examples:

- `#[1,2,3].toSubarray.popFront.toArray = #[2, 3]`
- `#[1,2,3].toSubarray.popFront.popFront.toArray = #[3]`
- `#[1,2,3].toSubarray.popFront.popFront.popFront.toArray = #[]`
- `#[1,2,3].toSubarray.popFront.popFront.popFront.popFront.toArray = #[]`

def

```
Subarray.split.{u_1} {α : Type u_1} (s : Subarray α)
  (i : Fin s.size.succ) : Subarray α × Subarray α
```

Splits a subarray into two parts, the first of which contains the first i elements and the second of which contains the remainder.

19.16.5.3. Lookups

def

```
Subarray.get.{u_1} {α : Type u_1} (s : Subarray α) (i : Fin s.size)
: α
```

Extracts an element from the subarray.

The index is relative to the start of the subarray, rather than the underlying array.

def

```
Subarray.get!.{u_1} {α : Type u_1} [Inhabited α] (s : Subarray α)
  (i : Nat) : α
```

Extracts an element from the subarray, or returns a default value when the index is out of bounds.

The index is relative to the start and end of the subarray, rather than the underlying array. The default value is that provided by the `Inhabited α` instance.

def

```
Subarray.getD.{u_1} {α : Type u_1} (s : Subarray α) (i : Nat) (v₀ :
α) :
  α
```

Extracts an element from the subarray, or returns a default value v_0 when the index is out of bounds.

The index is relative to the start and end of the subarray, rather than the underlying array.

19.16.5.4. Iteration

def

```
Subarray.foldl.{u, v} {α : Type u} {β : Type v} (f : β → α → β)
  (init : β) (as : Subarray α) : β
```

Folds an operation from left to right over the elements in a subarray.

An accumulator of type β is constructed by starting with `init` and combining each element of the subarray with the current accumulator value in turn.

Examples:

- `#["red", "green", "blue"].toSubarray.foldl (· + ·.length) 0 = 12`
- `#["red", "green", "blue"].toSubarray.popFront.foldl (· + ·.length) 0 = 9`

def

```
Subarray.foldlM.{u, v, w} {α : Type u} {β : Type v}
  {m : Type v → Type w} [Monad m] (f : β → α → m β) (init : β)
  (as : Subarray α) : m β
```

Folds a monadic operation from left to right over the elements in a subarray.

An accumulator of type β is constructed by starting with $init$ and monadically combining each element of the subarray with the current accumulator value in turn. The monad in question may permit early termination or repetition.

Examples:

```
#eval #["red", "green", "blue"].toSubarray.foldlM (init := "")
fun acc x => do
  let l ← Option.guard (· ≠ 0) x.length
  return s!"{acc}({l}){x} "

some "(3)red (5)green (4)blue "
```

```
#eval #["red", "green", "blue"].toSubarray.foldlM (init := 0) fun
acc x => do
  let l ← Option.guard (· ≠ 5) x.length
  return s!"{acc}({l}){x} "

none
```

def

```
Subarray.foldr.{u, v} {α : Type u} {β : Type v} (f : α → β → β)
  (init : β) (as : Subarray α) : β
```

Folds an operation from right to left over the elements in a subarray.

An accumulator of type β is constructed by starting with $init$ and combining each element of the subarray with the current accumulator value in turn, moving from the end to the start.

Examples:

- `#eval #["red", "green", "blue"].toSubarray.foldr (·.length + ·) 0 = 12`
- `#["red", "green", "blue"].toSubarray.popFront.foldlr (·.length + ·) 0 = 9`

def

```
Subarray.foldrM.{u, v, w} {α : Type u} {β : Type v}
  {m : Type v → Type w} [Monad m] (f : α → β → m β) (init : β)
  (as : Subarray α) : m β
```

Folds a monadic operation from right to left over the elements in a subarray.

An accumulator of type β is constructed by starting with $init$ and monadically combining each element of the subarray with the current accumulator value in turn, moving from the end to the start. The monad in question may permit early termination or repetition.

Examples:

```
#eval #["red", "green", "blue"].toSubarray.foldrM (init := "")
fun x acc => do
  let l ← Option.guard (· ≠ 0) x.length
  return s!"{acc}({l}){x} "

some "(4)blue (5)green (3)red "
```

```
#eval #["red", "green", "blue"].toSubarray.foldrM (init := 0) fun
x acc => do
  let l ← Option.guard (· ≠ 5) x.length
  return s!"{acc}({l}){x} "

none
```

def

```
Subarray.forM.{u, v, w} {α : Type u} {m : Type v → Type w} [Monad
m]
  (f : α → m PUnit) (as : Subarray α) : m PUnit
```

Runs a monadic action on each element of a subarray.

The elements are processed starting at the lowest index and moving up.

def

```
Subarray.forRevM.{u, v, w} {α : Type u} {m : Type v → Type w}
[Monad m]
  (f : α → m PUnit) (as : Subarray α) : m PUnit
```

Runs a monadic action on each element of a subarray, in reverse order.

The elements are processed starting at the highest index and moving down.

opaque

```
Subarray.forIn.{u, v, w} {α : Type u} {β : Type v} {m : Type v →
Type w}
  [Monad m] (s : Subarray α) (b : β) (f : α → β → m (ForInStep β))
: m β
```

The implementation of ForIn.forIn for Subarray, which allows it to be used with **for** loops in **do**-notation.

19.16.5.5. Element Predicates

def

```
Subarray.findRev? {α : Type} (as : Subarray α) (p : α → Bool) :
Option α
```

Tests each element in a subarray with a Boolean predicate in reverse order, stopping at the first element that satisfies the predicate. The element that satisfies the predicate is returned, or none if no element satisfies the predicate.

Examples:

- `#["red", "green", "blue"].toSubarray.findRev? (·.length ≠ 4) = some "green"`
- `#["red", "green", "blue"].toSubarray.findRev? (fun _ => true) = some "blue"`
- `#["red", "green", "blue"].toSubarray 0 0 |>.findRev? (fun _ => true) = none`

def

```
Subarray.findRevM?.{w} {α : Type} {m : Type → Type w} [Monad m]
  (as : Subarray α) (p : α → m Bool) : m (Option α)
```

Applies a monadic Boolean predicate to each element in a subarray in reverse order, stopping at the first element that satisfies the predicate. The element that satisfies the predicate is returned, or none if no element satisfies it.

Example:

```
#eval #["red", "green", "blue"].toSubarray.findRevM? fun x => do
  IO.println x
  return (x.length = 5)
```

```
blue
green
```

```
some 5
```

def

```
Subarray.findSomeRevM?.{u, v, w} {α : Type u} {β : Type v}
  {m : Type v → Type w} [Monad m] (as : Subarray α)
  (f : α → m (Option β)) : m (Option β)
```

Applies a monadic function to each element in a subarray in reverse order, stopping at the first element for which the function succeeds by returning a value other than none. The succeeding value is returned, or none if there is no success.

Example:

```
#eval #["red", "green", "blue"].toSubarray.findSomeRevM? fun x =>
do
  IO.println x
```

```
  return Option.guard (· = 5) x.length

blue
green

some 5
```

def

```
Subarray.all.{u} {α : Type u} (p : α → Bool) (as : Subarray α) :
Bool
```

Checks whether all of the elements in a subarray satisfy a Boolean predicate.

The elements are tested starting at the lowest index and moving up. The search terminates as soon as an element that does not satisfy the predicate is found.

def

```
Subarray.allM.{u, w} {α : Type u} {m : Type → Type w} [Monad m]
  (p : α → m Bool) (as : Subarray α) : m Bool
```

Checks whether all of the elements in a subarray satisfy a monadic Boolean predicate.

The elements are tested starting at the lowest index and moving up. The search terminates as soon as an element that does not satisfy the predicate is found.

Example:

```
#eval #["red", "green", "blue",
"orange"].toSubarray.popFront.allM fun x => do
  IO.println x
  pure (x.length == 5)

green
blue

false
```

def

```
Subarray.any.{u} {α : Type u} (p : α → Bool) (as : Subarray α) :
Bool
```

Checks whether any of the elements in a subarray satisfy a Boolean predicate.

The elements are tested starting at the lowest index and moving up. The search terminates as soon as an element that satisfies the predicate is found.

def

```
Subarray.anyM.{u, w} {α : Type u} {m : Type → Type w} [Monad m]
  (p : α → m Bool) (as : Subarray α) : m Bool
```

Checks whether any of the elements in a subarray satisfy a monadic Boolean predicate.

The elements are tested starting at the lowest index and moving up. The search terminates as soon as an element that satisfies the predicate is found.

Example:

```
#eval #["red", "green", "blue",
"orange"].toSubarray.popFront.anyM fun x => do
  IO.println x
  pure (x == "blue")

green
blue

true
```

19.16.6. FFI

FFI type

```
typedef struct {
    lean_object   m_header;
    size_t        m_size;
    size_t        m_capacity;
    lean_object * m_data[];
} lean_array_object;
```

The representation of arrays in C. See the description of run-time Arrays for more details.

FFI function

```
bool lean_is_array(lean_object * o)
```

Returns `true` if `o` is an array, or `false` otherwise.

FFI function

```
lean_array_object * lean_to_array(lean_object * o)
```

Performs a runtime check that o is indeed an array. If o is not an array, an assertion fails.

19.17. Maps and Sets

A *map* is a data structure that associates keys with values. They are also referred to as *dictionaries*, *associative arrays*, or simply as hash tables.

In Lean, maps may have the following properties:

Representation

The in-memory representation of a map may be either a tree or a hash table. Tree-based representations are better when the reference to the data structure is shared, because hash tables are based on arrays. Arrays are copied in full on modification when the reference is not unique, while only the path from the root of the tree to the modified nodes must be copied on modification of a tree. Hash tables, on the other hand, can be more efficient when references are not shared, because non-shared arrays can be modified in constant time. Furthermore, tree-based maps store data in order and thus support ordered traversals of the data.

Extensionality

Maps can be viewed as partial functions from keys to values. *Extensional maps* are maps for which propositional equality matches this interpretation. This can be convenient for reasoning, but it also rules out some useful operations that would be able to distinguish between them. In general, extensional maps should be used only when needed for verification.

Dependent or Not

A *dependent map* is one in which the type of each value is determined by its corresponding key, rather than being constant. Dependent maps have more expressive power, but are also more difficult to use. They impose more requirements on their users. For example, many operations on `DHashMap` require `LawfulBEq` instances rather than `BEq`.

Map	Representation	Extensional?	Dependent?
`TreeMap`	Tree	No	No
`DTreeMap`	Tree	No	Yes
`HashMap`	Hash Table	No	No
`DHashMap`	Hash Table	No	Yes
`ExtHashMap`	Hash Table	Yes	No
`ExtDHashMap`	Hash Table	Yes	Yes

A map can always be used as a set by setting its type of values to `Unit`. The following set types are provided:

- `Std.HashSet` is a set based on hash tables. Its performance characteristics are like those of `Std.HashMap`: it is based on arrays and can be efficiently updated, but only when not shared.
- `Std.TreeSet` is a set based on balanced trees. Its performance characteristics are like those of `Std.TreeMap`.
- `Std.ExtHashSet` is an extensional hash set type that matches the mathematical notion of finite sets: two sets are equal if they contain the same elements.

19.17.1. Library Design

All the basic operations on maps and sets are fully verified. They are proven correct with respect to simpler models implemented with lists. At the same time, maps and sets have predictable performance.

Some types include additional operations that are not yet fully verified. These operations are useful, and not all programs need full verification. Examples include `HashMap.partition` and `TreeMap.filterMap`.

19.17.1.1. Fused Operations

It is common to modify a table based on its pre-existing contents. To avoid having to traverse a data structure twice, many query/modification pairs are provided in “fused” variants that perform a query while modifying a map or set. In some cases, the result of the query affects the modification.

For example, `Std.HashMap` provides `containsThenInsert`, which inserts a key-value pair into a map while signalling whether it was previously found, and `containsThenInsertIfNew`, which inserts the new mapping only if it was not previously present. The `alter` function modifies the value for a given key without having to search for the key multiple times; the alternation is performed by a function in which missing values are represented by `none`.

19.17.1.2. Raw Data and Invariants

Both hash-based and tree-based maps rely on certain internal well-formedness invariants, such as that trees are balanced and ordered. In Lean's standard library, these data structures are represented as a pair of the underlying data with a proof that it is well formed. This fact is mostly an internal implementation detail; however, it is relevant to users in one situation: this representation prevents them from being used in nested inductive types.

To enable their use in nested inductive types, the standard library provides “raw” variants of each container along with separate “unbundled” versions of their invariants. These use the following naming convention:

- `T.Raw` is the version of type `T` without its invariants. For example, `Std.HashMap.Raw` is a version of `Std.HashMap` without the embedded proofs.

- `T.Raw.WF` is the corresponding well-formedness predicate. For example, `Std.HashMap.Raw.WF` asserts that a `Std.HashMap.Raw` is well-formed.
- Each operation on `T`, called `T.f`, has a corresponding operation on `T.Raw` called `T.Raw.f`. For example, `Std.HashMap.Raw.insert` is the version of `Std.HashMap.insert` to be used with raw hash maps.
- Each operation `T.Raw.f` has an associated well-formedness lemma `T.Raw.WF.f`. For example, `Std.HashMap.Raw.WF.insert` asserts that inserting a new key-value pair into a well-formed raw hash map results in a well-formed raw hash map.

Because the vast majority of use cases do not require them, not all lemmas about raw types are imported by default with the data structures. It is usually necessary to import `Std.Data.T.RawLemmas` (where `T` is the data structure in question).

A nested inductive type that occurs inside a map or set should be defined in three stages:

1. First, define a raw version of the nested inductive type that uses the raw version of the map or set type. Define any necessary operations.
2. Next, define an inductive predicate that asserts that all maps or sets in the raw nested type are well formed. Show that the operations on the raw type preserve well-formedness.
3. Construct an appropriate interface to the nested inductive type by defining an API that proves well-formedness properties as needed, hiding them from users.

▼ *Example: Nested Inductive Types with `Std.HashMap`*

This example requires that `Std.Data.HashMap.RawLemmas` is imported. To keep the code shorter, the `Std` namespace is opened:

```
open Std
```

The map of an adventure game may consist of a series of rooms, connected by passages. Each room has a description, and each passage faces in a particular direction. This can be represented as a recursive structure.

```
structure Maze where
  description : String
  passages : HashMap String Maze
```

This definition is rejected:

```
(kernel) application type mismatch
  DHashMap.Raw.WF inner
argument has type
  _nested.Std.DHashMap.Raw_3
but function has type
  (DHashMap.Raw String fun x => Maze) → Prop
```

Making this work requires separating the well-formedness predicates from the structure. The first step is to redefine the type without embedded hash map invariants:

```
structure RawMaze where
  description : String
  passages : Std.HashMap.Raw String RawMaze
```

The most basic raw maze has no passages:

```
def RawMaze.base (description : String) : RawMaze where
  description := description
  passages := ∅
```

A passage to a further maze can be added to a raw maze using `RawMaze.insert`:

```
def RawMaze.insert (maze : RawMaze)
    (direction : String) (next : RawMaze) : RawMaze :=
  { maze with
    passages := maze.passages.insert direction next
  }
```

The second step is to define a well-formedness predicate for `RawMaze` that ensures that each included hash map is well-formed. If the `passages` field itself is well-formed, and all raw mazes included in it are well-formed, then a raw maze is well-formed.

```
inductive RawMaze.WF : RawMaze → Prop
  | mk {description passages} :
    (∀ (dir : String) v, passages[dir]? = some v → WF v) →
    passages.WF →
    WF { description, passages := passages }
```

Base mazes are well-formed, and inserting a passage to a well-formed maze into some other well-formed maze produces a well-formed maze:

```
theorem RawMaze.base_wf (description : String) :
    RawMaze.WF (.base description) := by
  constructor
  . intro v h h'
    simp [Std.HashMap.Raw.getElem?_empty] at *
  . exact HashMap.Raw.WF.empty

def RawMaze.insert_wf (maze : RawMaze) :
    WF maze → WF next → WF (maze.insert dir next) := by
  let ⟨desc, passages⟩ := maze
  intro ⟨wfMore, wfPassages⟩ wfNext
  constructor
  . intro dir' v
    rw [HashMap.Raw.getElem?_insert wfPassages]
    split <;> intros <;> simp_all [wfMore dir']
  . simp_all [HashMap.Raw.WF.insert]
```

Finally, a more friendly interface can be defined that frees users from worrying about well-formedness. A `Maze` bundles up a `RawMaze` with a proof that it is well-formed:

```
structure Maze where
  raw : RawMaze
  wf : raw.WF
```

The `base` and `insert` operators take care of the well-formedness proof obligations:

```
def Maze.base (description : String) : Maze where
  raw := .base description
  wf := by apply RawMaze.base_wf

def Maze.insert (maze : Maze)
    (dir : String) (next : Maze) : Maze where
  raw := maze.raw.insert dir next.raw
  wf := RawMaze.insert_wf maze.raw maze.wf next.wf
```

Users of the `Maze` API may either check the description of the current maze or attempt to go in a direction to a new maze:

```
def Maze.description (maze : Maze) : String :=
  maze.raw.description

def Maze.go? (maze : Maze) (dir : String) : Option Maze :=
  match h : maze.raw.passages[dir]? with
  | none => none
  | some m' =>
    Maze.mk m' <| by
      let ⟨r, wf⟩ := maze
      let ⟨wfAll, _⟩ := wf
      apply wfAll dir
      apply h
```

19.17.1.3. Suitable Operators for Uniqueness

Care should be taken when working with data structures to ensure that as many references are unique as possible, which enables Lean to use destructive mutation behind the scenes while maintaining a pure functional interface. The map and set library provides operators that can be used to maintain uniqueness of references. In particular, when possible, operations such as `alter` or `modify` should be preferred over explicitly retrieving a value, modifying it, and reinserting it. These operations avoid creating a second reference to the value during modification.

▼ *Example: Modifying Values in Maps*

```
open Std
```

The function addAlias is used to track aliases of a string in some data set. One way to add an alias is to first look up the existing aliases, defaulting to the empty array, then insert the new alias, and finally save the resulting array in the map:

```
def addAlias (aliases : HashMap String (Array String))
    (key value : String) :
    HashMap String (Array String) :=
  let prior := aliases.getD key #[]
  aliases.insert key (prior.push value)
```

This implementation has poor performance characteristics. Because the map retains a reference to the prior values, the array must be copied rather than mutated. A better implementation explicitly erases the prior value from the map before modifying it:

```
def addAlias' (aliases : HashMap String (Array String))
    (key value : String) :
    HashMap String (Array String) :=
  let prior := aliases.getD key #[]
  let aliases := aliases.erase key
  aliases.insert key (prior.push value)
```

Using HashMap.alter is even better. It removes the need to explicitly delete and re-insert the value:

```
def addAlias'' (aliases : HashMap String (Array String))
    (key value : String) :
    HashMap String (Array String) :=
  aliases.alter key fun prior? =>
    some ((prior?.getD #[]).push value)
```

19.17.2. Hash Maps

The declarations in this section should be imported using `import Std.HashMap`.

structure

```
Std.HashMap.{u, v} (α : Type u) (β : Type v) [BEq α] [Hashable α] :
  Type (max u v)
```

Hash maps.

This is a simple separate-chaining hash table. The data of the hash map consists of a cached size and an array of buckets, where each bucket is a linked list of key-value pais. The number of buckets is always a power of two. The hash map doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an Array. Users should make sure that the hash map is used linearly to avoid expensive copies.

The hash map uses == (provided by the BEq typeclass) to compare keys and hash (provided by the Hashable typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and `a == b` should imply `hash a = hash b` (see also the EquivBEq and LawfulHashable typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

These hash maps contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.Data.HashMap.Raw` and `Std.Data.HashMap.Raw.WF` unbundle the invariant from the hash map. When in doubt, prefer `HashMap` over `HashMap.Raw`.

Dependent hash maps, in which keys may occur in their values' types, are available as `Std.Data.DHashMap`.

19.17.2.1. Creation

def

```
Std.HashMap.emptyWithCapacity.{u, v} {α : Type u} {β : Type v} [BEq α]
  [Hashable α] (capacity : Nat := 8) : Std.HashMap α β
```

Creates a new empty hash map. The optional parameter *capacity* can be supplied to presize the map so that it can hold the given number of mappings without reallocating. It is also possible to use the empty collection notations ∅ and {} to create an empty hash map with the default capacity.

19.17.2.2. Properties

def

```
Std.HashMap.size.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : Nat
```

The number of mappings present in the hash map

def

```
Std.HashMap.isEmpty.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : Bool
```

Returns true if the hash map contains no mappings.

Note that if your BEq instance is not reflexive or your Hashable instance is not lawful, then it is possible that this function returns false even though is not possible to get anything out of the hash map.

structure

```
Std.HashMap.Equiv.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m₁ m₂ : Std.HashMap α β) : Prop
```

Two hash maps are equivalent in the sense of `Equiv` iff all the keys and values are equal.

Constructor

`Std.HashMap.Equiv.mk.{u, v}`

Fields

`inner : m₁.inner.Equiv m₂.inner`

Internal implementation detail of the hash map

syntax

Equivalence

The relation `HashMap.Equiv` can also be written with an infix operator, which is scoped to its namespace:

```
term ::= ...
    | term ~m term
```

19.17.2.3. Queries

def

```
Std.HashMap.contains.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) : Bool
```

Returns `true` if there is a mapping for the given key. There is also a `Prop`-valued version of this: `a ∈ m` is equivalent to `m.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` uses `==` for comparisons, while for hash maps, both use `==`.

def

```
Std.HashMap.get.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) (h : a ∈ m) : β
```

The notation `m[a]` or `m[a]'h` is preferred over calling this function directly.

Retrieves the mapping for the given key. Ensures that such a mapping exists by requiring a proof of $a \in m$.

def

```
Std.HashMap.get!.{u, v} {α : Type u} {β : Type v} {✝ : BEq α}
  {✝¹ : Hashable α} [Inhabited β] (m : Std.HashMap α β) (a : α) :
β
```

The notation `m[a]!` is preferred over calling this function directly.

Tries to retrieve the mapping for the given key, panicking if no such mapping is present.

def

```
Std.HashMap.get?.{u, v} {α : Type u} {β : Type v} {✝ : BEq α}
  {✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) : Option β
```

The notation `m[a]?` is preferred over calling this function directly.

Tries to retrieve the mapping for the given key, returning none if no such mapping is present.

def

```
Std.HashMap.getD.{u, v} {α : Type u} {β : Type v} {✝ : BEq α}
  {✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) (fallback : β) :
β
```

Tries to retrieve the mapping for the given key, returning `fallback` if no such mapping is present.

def

```
Std.HashMap.getKey.{u, v} {α : Type u} {β : Type v} {✝ : BEq α}
  {✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) (h : a ∈ m) : α
```

Retrieves the key from the mapping that matches a. Ensures that such a mapping exists by requiring a proof of $a \in m$. The result is guaranteed to be pointer equal to the key in the map.

def

```
Std.HashMap.getKey!.{u, v} {α : Type u} {β : Type v} {✝ : BEq α}
  {✝¹ : Hashable α} [Inhabited α] (m : Std.HashMap α β) (a : α) :
α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.HashMap.getKey?.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) : Option α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise none. The result in the some case is guaranteed to be pointer equal to the key in the map.

def

```
Std.HashMap.getKeyD.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a fallback : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise *fallback*. If a mapping exists the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.HashMap.keys.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : List α
```

Returns a list of all keys present in the hash map in some order.

def

```
Std.HashMap.keysArray.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : Array α
```

Returns an array of all keys present in the hash map in some order.

def

```
Std.HashMap.values.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : List β
```

Returns a list of all values present in the hash map in some order.

def

```
Std.HashMap.valuesArray.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : Array β
```

Returns an array of all values present in the hash map in some order.

19.17.2.4. Modification

def

```
Std.HashMap.alter.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α)
  (f : Option β → Option β) : Std.HashMap α β
```

Modifies in place the value associated with a given key, allowing creating new values and deleting values via an `Option` valued replacement function.

This function ensures that the value is used linearly.

def

```
Std.HashMap.modify.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) (f : β → β) :
  Std.HashMap α β
```

Modifies in place the value associated with a given key.

This function ensures that the value is used linearly.

def

```
Std.HashMap.containsThenInsert.{u, v} {α : Type u} {β : Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α)
  (b : β) : Bool × Std.HashMap α β
```

Checks whether a key is present in a map, and unconditionally inserts a value for the key.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.HashMap.containsThenInsertIfNew.{u, v} {α : Type u} {β : Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α)
  (b : β) : Bool × Std.HashMap α β
```

Checks whether a key is present in a map and inserts a value for the key if it was not found.

If the returned `Bool` is `true`, then the returned map is unaltered. If the `Bool` is `false`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `contains` followed by `insertIfNew`.

def

```
Std.HashMap.erase.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) : Std.HashMap α
β
```

Removes the mapping for the given key if it exists.

def

```
Std.HashMap.filter.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (f : α → β → Bool) (m : Std.HashMap α β) :
  Std.HashMap α β
```

Removes all mappings of the hash map for which the given function returns `false`.

def

```
Std.HashMap.filterMap.{u, v, w} {α : Type u} {β : Type v} {γ : Type
w}
  [BEq α] [Hashable α] (f : α → β → Option γ) (m : Std.HashMap α β)
:
  Std.HashMap α γ
```

Updates the values of the hash map by applying the given function to all mappings, keeping only those mappings where the function returns `some` value.

def

```
Std.HashMap.insert.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) (b : β) :
  Std.HashMap α β
```

Inserts the given mapping into the map. If there is already a mapping for the given key, then both key and value will be replaced.

Note: this replacement behavior is true for HashMap, DHashMap, HashMap.Raw and DHashMap.Raw. The `insert` function on HashSet and HashSet.Raw behaves differently: it will return the set unchanged if a matching key is already present.

def

```
Std.HashMap.insertIfNew.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α) (b : β) :
  Std.HashMap α β
```

If there is no mapping for the given key, inserts the given mapping into the map. Otherwise, returns the map unaltered.

def

```
Std.HashMap.getThenInsertIfNew?.{u, v} {α : Type u} {β : Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} (m : Std.HashMap α β) (a : α)
  (b : β) : Option β × Std.HashMap α β
```

Checks whether a key is present in a map, returning the associate value, and inserts a value for the key if it was not found.

If the returned value is `some v`, then the returned map is unaltered. If it is `none`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `get?` followed by `insertIfNew`.

def

```
Std.HashMap.insertMany.{u, v, w} {α : Type u} {β : Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} {ρ : Type w} [ForIn Id ρ (α × β)]
  (m : Std.HashMap α β) (l : ρ) : Std.HashMap α β
```

Inserts multiple mappings into the hash map by iterating over the given collection and calling `insert`. If the same key appears multiple times, the last occurrence takes precedence.

Note: this precedence behavior is true for `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw`. The `insertMany` function on `HashSet` and `HashSet.Raw` behaves differently: it will prefer the first appearance.

def

```
Std.HashMap.insertManyIfNewUnit.{u, w} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} {ρ : Type w} [ForIn Id ρ α]
  (m : Std.HashMap α Unit) (l : ρ) : Std.HashMap α Unit
```

Inserts multiple keys with the value `()` into the hash map by iterating over the given collection and calling `insertIfNew`. If the same key appears multiple times, the first occurrence takes precedence.

This is mainly useful to implement `HashSet.insertMany`, so if you are considering using this, `HashSet` or `HashSet.Raw` might be a better fit for you.

def

```
Std.HashMap.partition.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (f : α → β → Bool) (m : Std.HashMap α β) :
  Std.HashMap α β × Std.HashMap α β
```

Partition a hash map into two hash map based on a predicate.

def

```
Std.HashMap.union.{u, v} {α : Type u} {β : Type v} [BEq α]
[Hashable α]
  (m₁ m₂ : Std.HashMap α β) : Std.HashMap α β
```

Computes the union of the given hash maps, by traversing m_2 and inserting its elements into m_1.

19.17.2.5. Iteration

def

```
Std.HashMap.map.{u, v, w} {α : Type u} {β : Type v} {γ : Type w}
[BEq α]
  [Hashable α] (f : α → β → γ) (m : Std.HashMap α β) : Std.HashMap
α γ
```

Updates the values of the hash map by applying the given function to all mappings.

def

```
Std.HashMap.fold.{u, v, w} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} {γ : Type w} (f : γ → α → β → γ) (init : γ)
  (b : Std.HashMap α β) : γ
```

Folds the given function over the mappings in the hash map in some order.

def

```
Std.HashMap.foldM.{u, v, w} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} {m : Type w → Type w} [Monad m] {γ : Type w}
  (f : γ → α → β → m γ) (init : γ) (b : Std.HashMap α β) : m γ
```

Monadically computes a value by folding the given function over the mappings in the hash map in some order.

def

```
Std.HashMap.forIn.{u, v, w} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} {m : Type w → Type w} [Monad m] {γ : Type w}
  (f : α → β → γ → m (ForInStep γ)) (init : γ) (b : Std.HashMap α
β) :
  m γ
```

Support for the **`for`** loop construct in **`do`** blocks.

def

```
Std.HashMap.forM.{u, v, w} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} {m : Type w → Type w} [Monad m]
  (f : α → β → m PUnit) (b : Std.HashMap α β) : m PUnit
```

Carries out a monadic action on each mapping in the hash map in some order.

19.17.2.6. Conversion

def

```
Std.HashMap.ofList.{u, v} {α : Type u} {β : Type v} [BEq α]
[Hashable α]
  (l : List (α × β)) : Std.HashMap α β
```

Creates a hash map from a list of mappings. If the same key appears multiple times, the last occurrence takes precedence.

def

```
Std.HashMap.toArray.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : Array (α × β)
```

Transforms the hash map into an array of mappings in some order.

def

```
Std.HashMap.toList.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashMap α β) : List (α × β)
```

Transforms the hash map into a list of mappings in some order.

def

```
Std.HashMap.unitOfArray.{u} {α : Type u} [BEq α] [Hashable α]
  (l : Array α) : Std.HashMap α Unit
```

Creates a hash map from an array of keys, associating the value `()` with each key.

This is mainly useful to implement `HashSet.ofArray`, so if you are considering using this, `HashSet` or `HashSet.Raw` might be a better fit for you.

def

```
Std.HashMap.unitOfList.{u} {α : Type u} [BEq α] [Hashable α]
  (l : List α) : Std.HashMap α Unit
```

Creates a hash map from a list of keys, associating the value `()` with each key.

This is mainly useful to implement `HashSet.ofList`, so if you are considering using this, `HashSet` or `HashSet.Raw` might be a better fit for you.

19.17.2.7. Unbundled Variants

Unbundled maps separate well-formedness proofs from data. This is primarily useful when defining nested inductive types. To use these variants, import the modules `Std.HashMap.Raw` and `Std.HashMap.RawLemmas`.

structure

```
Std.HashMap.Raw.{u, v} (α : Type u) (β : Type v) : Type (max u v)
```

Hash maps without a bundled well-formedness invariant, suitable for use in nested inductive types. The well-formedness invariant is called `Raw.WF`. When in doubt, prefer `HashMap` over `HashMap.Raw`. Lemmas about the operations on `Std.Data.HashMap.Raw` are available in the module `Std.Data.HashMap.RawLemmas`.

This is a simple separate-chaining hash table. The data of the hash map consists of a cached size and an array of buckets, where each bucket is a linked list of key-value pais. The number of buckets is always a power of two. The hash map doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an `Array`. Users should make sure that the hash map is used linearly to avoid expensive copies.

The hash map uses == (provided by the `BEq` typeclass) to compare keys and `hash` (provided by the `Hashable` typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and `a == b` should imply `hash a = hash b` (see also the `EquivBEq` and `LawfulHashable` typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

Dependent hash maps, in which keys may occur in their values' types, are available as `Std.Data.Raw.DHashMap`.

Constructor

Std.HashMap.Raw.mk.{u, v}

Fields

inner : Std.DHashMap.Raw α fun x => β

Internal implementation detail of the hash map

structure

Std.HashMap.Raw.WF.{u, v} {α : Type u} {β : Type v} [BEq α] [Hashable α]
(m : Std.HashMap.Raw α β) : Prop

Well-formedness predicate for hash maps. Users of `HashMap` will not need to interact with this. Users of `HashMap.Raw` will need to provide proofs of `WF` to lemmas and should use lemmas `WF.empty` and `WF.insert` (which are always named exactly like the operations they are about) to show that map operations preserve well-formedness.

Constructor

Std.HashMap.Raw.WF.mk.{u, v}

Fields

out : m.inner.WF

Internal implementation detail of the hash map

19.17.3. Dependent Hash Maps

The declarations in this section should be imported using `import Std.DHashMap`.

structure

Std.DHashMap.{u, v} (α : Type u) (β : α → Type v) [BEq α] [Hashable α] :
Type (max u v)

Dependent hash maps.

This is a simple separate-chaining hash table. The data of the hash map consists of a cached size and an array of buckets, where each bucket is a linked list of key-value pais. The number of buckets is always a power of two. The hash map doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an Array. Users should make sure that the hash map is used linearly to avoid expensive copies.

The hash map uses == (provided by the BEq typeclass) to compare keys and hash (provided by the Hashable typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and a == b should imply hash a = hash b (see also the EquivBEq and LawfulHashable typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if a == b implies $a = b$.

These hash maps contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, Std.DHashMap.Raw and Std.DHashMap.Raw.WF unbundle the invariant from the hash map. When in doubt, prefer DHashMap over DHashMap.Raw.

For a variant that is more convenient for use in proofs because of extensionalities, see Std.ExtDHashMap which is defined in the module Std.Data.ExtDHashMap.

19.17.3.1. Creation

def

```
Std.DHashMap.emptyWithCapacity.{u, v} {α : Type u} {β : α → Type v}
  [BEq α] [Hashable α] (capacity : Nat := 8) : Std.DHashMap α β
```

Creates a new empty hash map. The optional parameter *capacity* can be supplied to presize the map so that it can hold the given number of mappings without reallocating. It is also possible to use the empty collection notations ∅ and {} to create an empty hash map with the default capacity.

19.17.3.2. Properties

def

```
Std.DHashMap.size.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) : Nat
```

The number of mappings present in the hash map

def

```
Std.DHashMap.isEmpty.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) : Bool
```

Returns true if the hash map contains no mappings.

Note that if your BEq instance is not reflexive or your Hashable instance is not lawful, then it is possible that this function returns false even though is not possible to get anything out of the hash map.

structure

```
Std.DHashMap.Equiv.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m₁ m₂ : Std.DHashMap α β) : Prop
```

Two hash maps are equivalent in the sense of Equiv iff all the keys and values are equal.

Constructor

Std.DHashMap.Equiv.mk.{u, v}

Fields

inner : m_1.inner.Equiv m_2.inner

Internal implementation detail of the hash map

syntax

Equivalence

The relation DHashMap.Equiv can also be written with an infix operator, which is scoped to its namespace:

```
term ::= ...
    | term ~m term
```

19.17.3.3. Queries

def

```
Std.DHashMap.contains.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α) : Bool
```

Returns true if there is a mapping for the given key. There is also a Prop-valued version of this: $a \in m$ is equivalent to m.contains a = true.

Observe that this is different behavior than for lists: for lists, ∈ uses = and contains uses == for comparisons, while for hash maps, both use ==.

def

```
Std.DHashMap.get.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α β) (a : α)
  (h : a ∈ m) : β a
```

Retrieves the mapping for the given key. Ensures that such a mapping exists by requiring a proof of `a ∈ m`.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.DHashMap.get!.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α β) (a : α)
  [Inhabited (β a)] : β a
```

Tries to retrieve the mapping for the given key, panicking if no such mapping is present.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.DHashMap.get?.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α β) (a : α) :
  Option (β a)
```

Tries to retrieve the mapping for the given key, returning `none` if no such mapping is present.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.DHashMap.getD.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α β) (a : α)
  (fallback : β a) : β a
```

Tries to retrieve the mapping for the given key, returning `fallback` if no such mapping is present.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.DHashMap.getKey.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α) (h : a ∈ m) : α
```

Retrieves the key from the mapping that matches a. Ensures that such a mapping exists by requiring a proof of $a \in m$. The result is guaranteed to be pointer equal to the key in the map.

def

```
Std.DHashMap.getKey!.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [Inhabited α] (m : Std.DHashMap α β) (a : α) :
α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.DHashMap.getKey?.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α) : Option α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise none. The result in the some case is guaranteed to be pointer equal to the key in the map.

def

```
Std.DHashMap.getKeyD.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a fallback : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise $fallback$. If a mapping exists the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.DHashMap.keys.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) : List α
```

Returns a list of all keys present in the hash map in some order.

def

```
Std.DHashMap.keysArray.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) : Array α
```

Returns an array of all keys present in the hash map in some order.

def

```
Std.DHashMap.values.{u, v} {α : Type u} {x✝ : BEq α} {x✝¹ :
Hashable α}
  {β : Type v} (m : Std.DHashMap α fun x => β) : List β
```

Returns a list of all values present in the hash map in some order.

def

```
Std.DHashMap.valuesArray.{u, v} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} {β : Type v} (m : Std.DHashMap α fun x => β) :
  Array β
```

Returns an array of all values present in the hash map in some order.

19.17.3.4. Modification

def

```
Std.DHashMap.alter.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α β) (a : α)
  (f : Option (β a) → Option (β a)) : Std.DHashMap α β
```

Modifies in place the value associated with a given key, allowing creating new values and deleting values via an `Option` valued replacement function.

This function ensures that the value is used linearly.

def

```
Std.DHashMap.modify.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α β) (a : α)
  (f : β a → β a) : Std.DHashMap α β
```

Modifies in place the value associated with a given key.

This function ensures that the value is used linearly.

def

```
Std.DHashMap.containsThenInsert.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α)
  (b : β a) : Bool × Std.DHashMap α β
```

Checks whether a key is present in a map, and unconditionally inserts a value for the key.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.DHashMap.containsThenInsertIfNew.{u, v} {α : Type u}
  {β : α → Type v} {x✝ : BEq α} {x✝¹ : Hashable α}
  (m : Std.DHashMap α β) (a : α) (b : β a) : Bool × Std.DHashMap α
β
```

Checks whether a key is present in a map and inserts a value for the key if it was not found.

If the returned `Bool` is `true`, then the returned map is unaltered. If the `Bool` is `false`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `contains` followed by `insertIfNew`.

def

```
Std.DHashMap.erase.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α) : Std.DHashMap
α β
```

Removes the mapping for the given key if it exists.

def

```
Std.DHashMap.filter.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (f : (a : α) → β a → Bool) (m : Std.DHashMap α
β) :
  Std.DHashMap α β
```

Removes all mappings of the hash map for which the given function returns `false`.

def

```
Std.DHashMap.filterMap.{u, v, w} {α : Type u} {β : α → Type v}
  {δ : α → Type w} [BEq α] [Hashable α]
  (f : (a : α) → β a → Option (δ a)) (m : Std.DHashMap α β) :
  Std.DHashMap α δ
```

Updates the values of the hash map by applying the given function to all mappings, keeping only those mappings where the function returns `some` value.

def

```
Std.DHashMap.insert.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α) (b : β a) :
  Std.DHashMap α β
```

Inserts the given mapping into the map. If there is already a mapping for the given key, then both key and value will be replaced.

Note: this replacement behavior is true for `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw`. The `insert` function on `HashSet` and `HashSet.Raw` behaves differently: it will return the set unchanged if a matching key is already present.

def

```
Std.DHashMap.insertIfNew.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} (m : Std.DHashMap α β) (a : α)
  (b : β a) : Std.DHashMap α β
```

If there is no mapping for the given key, inserts the given mapping into the map. Otherwise, returns the map unaltered.

def

```
Std.DHashMap.getThenInsertIfNew?.{u, v} {α : Type u} {β : α → Type
v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.DHashMap α
β)
  (a : α) (b : β a) : Option (β a) × Std.DHashMap α β
```

Checks whether a key is present in a map, returning the associated value, and inserts a value for the key if it was not found.

If the returned value is `some v`, then the returned map is unaltered. If it is `none`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `get?` followed by `insertIfNew`.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.DHashMap.insertMany.{u, v, w} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} {ρ : Type w}
  [ForIn Id ρ ((a : α) × β a)] (m : Std.DHashMap α β) (l : ρ) :
  Std.DHashMap α β
```

Inserts multiple mappings into the hash map by iterating over the given collection and calling `insert`. If the same key appears multiple times, the last occurrence takes precedence.

Note: this precedence behavior is true for `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw`. The `insertMany` function on `HashSet` and `HashSet.Raw` behaves differently: it will prefer the first appearance.

def

```
Std.DHashMap.partition.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} (f : (a : α) → β a → Bool) (m : Std.DHashMap α β) :
  Std.DHashMap α β × Std.DHashMap α β
```

Partition a hash map into two hash map based on a predicate.

def

```
Std.DHashMap.union.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] (m₁ m₂ : Std.DHashMap α β) : Std.DHashMap α β
```

Computes the union of the given hash maps, by traversing m_2 and inserting its elements into m_1.

19.17.3.5. Iteration

def

```
Std.DHashMap.map.{u, v, w} {α : Type u} {β : α → Type v}
  {δ : α → Type w} [BEq α] [Hashable α] (f : (a : α) → β a → δ a)
  (m : Std.DHashMap α β) : Std.DHashMap α δ
```

Updates the values of the hash map by applying the given function to all mappings.

def

```
Std.DHashMap.fold.{u, v, w} {α : Type u} {β : α → Type v} {δ : Type w}
  {x✝ : BEq α} {x✝¹ : Hashable α} (f : δ → (a : α) → β a → δ) (init : δ)
  (b : Std.DHashMap α β) : δ
```

Folds the given function over the mappings in the hash map in some order.

def

```
Std.DHashMap.foldM.{u, v, w} {α : Type u} {β : α → Type v} {δ :
Type w}
  {m : Type w → Type w} [Monad m] {x✝ : BEq α} {x✝¹ : Hashable α}
  (f : δ → (a : α) → β a → m δ) (init : δ) (b : Std.DHashMap α β) :
m δ
```

Monadically computes a value by folding the given function over the mappings in the hash map in some order.

def

```
Std.DHashMap.forIn.{u, v, w} {α : Type u} {β : α → Type v} {δ :
Type w}
  {m : Type w → Type w} [Monad m] {x✝ : BEq α} {x✝¹ : Hashable α}
  (f : (a : α) → β a → δ → m (ForInStep δ)) (init : δ)
  (b : Std.DHashMap α β) : m δ
```

Support for the **`for`** loop construct in **`do`** blocks.

def

```
Std.DHashMap.forM.{u, v, w} {α : Type u} {β : α → Type v}
  {m : Type w → Type w} [Monad m] {x✝ : BEq α} {x✝¹ : Hashable α}
  (f : (a : α) → β a → m PUnit) (b : Std.DHashMap α β) : m PUnit
```

Carries out a monadic action on each mapping in the hash map in some order.

19.17.3.6. Conversion

def

```
Std.DHashMap.ofList.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] (l : List ((a : α) × β a)) : Std.DHashMap α β
```

Creates a hash map from a list of mappings. If the same key appears multiple times, the last occurrence takes precedence.

def

```
Std.DHashMap.toArray.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} (m : Std.DHashMap α β) : Array ((a : α) × β a)
```

Transforms the hash map into an array of mappings in some order.

def

```
Std.DHashMap.toList.{u, v} {α : Type u} {β : α → Type v} {✝ : BEq
α}
  {✝¹ : Hashable α} (m : Std.DHashMap α β) : List ((a : α) × β a)
```

Transforms the hash map into a list of mappings in some order.

19.17.3.7. Unbundled Variants

Unbundled maps separate well-formedness proofs from data. This is primarily useful when defining nested inductive types. To use these variants, import the modules `Std.DHashMap.Raw` and `Std.DHashMap.RawLemmas`.

structure

```
Std.DHashMap.Raw.{u, v} (α : Type u) (β : α → Type v) : Type (max u
v)
```

Dependent hash maps without a bundled well-formedness invariant, suitable for use in nested inductive types. The well-formedness invariant is called `Raw.WF`. When in doubt, prefer `DHashMap` over `DHashMap.Raw`. Lemmas about the operations on `Std.Data.DHashMap.Raw` are available in the module `Std.Data.DHashMap.RawLemmas`.

The hash table is backed by an `Array`. Users should make sure that the hash map is used linearly to avoid expensive copies.

This is a simple separate-chaining hash table. The data of the hash map consists of a cached size and an array of buckets, where each bucket is a linked list of key-value pais. The number of buckets is always a power of two. The hash map doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash map uses == (provided by the `BEq` typeclass) to compare keys and `hash` (provided by the `Hashable` typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and `a == b` should imply `hash a = hash b` (see also the `EquivBEq` and `LawfulHashable` typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

Constructor

`Std.DHashMap.Raw.mk.{u, v}`

Fields

`size : Nat`

The number of mappings present in the hash map

```
buckets : Array (Std.DHashMap.Internal.AssocList α β)
```

Internal implementation detail of the hash map

inductive predicate

```
Std.DHashMap.Raw.WF.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] : Std.DHashMap.Raw α β → Prop
```

Well-formedness predicate for hash maps. Users of `DHashMap` will not need to interact with this. Users of `DHashMap.Raw` will need to provide proofs of `WF` to lemmas and should use lemmas like `WF.empty` and `WF.insert` (which are always named exactly like the operations they are about) to show that map operations preserve well-formedness. The constructors of this type are internal implementation details and should not be accessed by users.

Constructors

```
| wf.{u, v} {α : Type u} {β : α → Type v} [BEq α] [Hashable α]
    {m : Std.DHashMap.Raw α β} :
    0 ≤ m.buckets.size →
      (∀ [EquivBEq α] [LawfulHashable α],
          Std.DHashMap.Internal.Raw.WFImp m) →
        m.WF
```

Internal implementation detail of the hash map

```
| emptyWithCapacity₀.{u, v} {α : Type u} {β : α → Type v}
    [BEq α] [Hashable α] {c : Nat} :
    (Std.DHashMap.Internal.Raw₀.emptyWithCapacity c).val.WF
```

Internal implementation detail of the hash map

```
| insert₀.{u, v} {α : Type u} {β : α → Type v} [BEq α]
    [Hashable α] {m : Std.DHashMap.Raw α β}
    {h : 0 ≤ m.buckets.size} {a : α} {b : β a} :
    m.WF →
      (Std.DHashMap.Internal.Raw₀.insert ⟨m, h⟩ a b).val.WF
```

Internal implementation detail of the hash map

```
| containsThenInsert₀.{u, v} {α : Type u} {β : α → Type v}
    [BEq α] [Hashable α] {m : Std.DHashMap.Raw α β}
    {h : 0 ≤ m.buckets.size} {a : α} {b : β a} :
    m.WF →
      (Std.DHashMap.Internal.Raw₀.containsThenInsert ⟨m, h⟩ a
              b).snd.val.WF
```

Internal implementation detail of the hash map

```
containsThenInsertIfNew₀.{u, v} {α : Type u}
  {β : α → Type v} [BEq α] [Hashable α]
  {m : Std.DHashMap.Raw α β} {h : 0 ≤ m.buckets.size}
  {a : α} {b : β a} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.containsThenInsertIfNew
          ⟨m, h⟩ a b).snd.val.WF
```

Internal implementation detail of the hash map

```
erase₀.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] {m : Std.DHashMap.Raw α β}
  {h : 0 ≤ m.buckets.size} {a : α} :
  m.WF → (Std.DHashMap.Internal.Raw₀.erase ⟨m, h⟩ a).val.WF
```

Internal implementation detail of the hash map

```
insertIfNew₀.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] {m : Std.DHashMap.Raw α β}
  {h : 0 ≤ m.buckets.size} {a : α} {b : β a} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.insertIfNew ⟨m, h⟩ a
      b).val.WF
```

Internal implementation detail of the hash map

```
getThenInsertIfNew?₀.{u, v} {α : Type u} {β : α → Type v}
  [BEq α] [Hashable α] [LawfulBEq α]
  {m : Std.DHashMap.Raw α β} {h : 0 ≤ m.buckets.size}
  {a : α} {b : β a} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.getThenInsertIfNew? ⟨m, h⟩ a
          b).snd.val.WF
```

Internal implementation detail of the hash map

```
filter₀.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] {m : Std.DHashMap.Raw α β}
  {h : 0 ≤ m.buckets.size} {f : (a : α) → β a → Bool} :
  m.WF → (Std.DHashMap.Internal.Raw₀.filter f ⟨m, h⟩).val.WF
```

Internal implementation detail of the hash map

```
constGetThenInsertIfNew?₀.{u, v} {α : Type u} {β : Type v}
  [BEq α] [Hashable α] {m : Std.DHashMap.Raw α fun x => β}
  {h : 0 ≤ m.buckets.size} {a : α} {b : β} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.Const.getThenInsertIfNew?
          ⟨m, h⟩ a b).snd.val.WF
```

Internal implementation detail of the hash map

```
modify₀.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] [LawfulBEq α] {m : Std.DHashMap.Raw α β}
  {h : 0 ≤ m.buckets.size} {a : α} {f : β a → β a} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.modify ⟨m, h⟩ a f).val.WF
```

Internal implementation detail of the hash map

```
constModify₀.{u, v} {α : Type u} {β : Type v} [BEq α]
  [Hashable α] {m : Std.DHashMap.Raw α fun x => β}
  {h : 0 ≤ m.buckets.size} {a : α} {f : β → β} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.Const.modify ⟨m, h⟩ a
          f).val.WF
```

Internal implementation detail of the hash map

```
alter₀.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] [LawfulBEq α] {m : Std.DHashMap.Raw α β}
  {h : 0 ≤ m.buckets.size} {a : α}
  {f : Option (β a) → Option (β a)} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.alter ⟨m, h⟩ a f).val.WF
```

Internal implementation detail of the hash map

```
constAlter₀.{u, v} {α : Type u} {β : Type v} [BEq α]
  [Hashable α] {m : Std.DHashMap.Raw α fun x => β}
  {h : 0 ≤ m.buckets.size} {a : α}
  {f : Option β → Option β} :
  m.WF →
    (Std.DHashMap.Internal.Raw₀.Const.alter ⟨m, h⟩ a
          f).val.WF
```

Internal implementation detail of the hash map

19.17.4. Extensional Hash Maps

The declarations in this section should be imported using `import Std.ExtHashMap`.

structure

```
Std.ExtHashMap.{u, v} (α : Type u) (β : Type v) [BEq α] [Hashable α] :
  Type (max u v)
```

Hash maps.

This is a simple separate-chaining hash table. The data of the hash map consists of a cached size and an array of buckets, where each bucket is a linked list of key-value pais. The number of buckets is always a power of two. The hash map doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an `Array`. Users should make sure that the hash map is used linearly to avoid expensive copies.

The hash map uses == (provided by the `BEq` typeclass) to compare keys and `hash` (provided by the `Hashable` typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and `a == b` should imply `hash a = hash b` (see also the `EquivBEq` and `LawfulHashable` typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

In contrast to regular hash maps, `Std.ExtHashMap` offers several extensionality lemmas and therefore has more lemmas about equality of hash maps. This however also makes it lose the ability to iterate freely over hash maps.

These hash maps contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.HashMap.Raw` and `Std.HashMap.Raw.WF` unbundle the invariant from the hash map. When in doubt, prefer `HashMap` or `ExtHashMap` over `HashMap.Raw`.

Dependent hash maps, in which keys may occur in their values' types, are available as `Std.ExtDHashMap` in the module `Std.Data.ExtDHashMap`.

19.17.4.1. Creation

def

```
Std.ExtHashMap.emptyWithCapacity.{u, v} {α : Type u} {β : Type v}
  [BEq α] [Hashable α] (capacity : Nat := 8) : Std.ExtHashMap α β
```

Creates a new empty hash map. The optional parameter *capacity* can be supplied to presize the map so that it can hold the given number of mappings without reallocating. It is also possible to use the empty collection notations ∅ and `{}` to create an empty hash map with the default capacity.

19.17.4.2. Properties

def

```
Std.ExtHashMap.size.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) : Nat
```

The number of mappings present in the hash map

def

```
Std.ExtHashMap.isEmpty.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) : Bool
```

Returns `true` if the hash map contains no mappings.

Note that if your `BEq` instance is not reflexive or your `Hashable` instance is not lawful, then it is possible that this function returns `false` even though is not possible to get anything out of the hash map.

19.17.4.3. Queries

def

```
Std.ExtHashMap.contains.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) : Bool
```

Returns `true` if there is a mapping for the given key. There is also a `Prop`-valued version of this: `a ∈ m` is equivalent to `m.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` uses `==` for comparisons, while for hash maps, both use `==`.

def

```
Std.ExtHashMap.get.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (h : a ∈ m) : β
```

The notation `m[a]` or `m[a]'h` is preferred over calling this function directly.

Retrieves the mapping for the given key. Ensures that such a mapping exists by requiring a proof of `a ∈ m`.

def

```
Std.ExtHashMap.get!.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α] [Inhabited β]
  (m : Std.ExtHashMap α β) (a : α) : β
```

The notation `m[a]!` is preferred over calling this function directly.

Tries to retrieve the mapping for the given key, panicking if no such mapping is present.

def

```
Std.ExtHashMap.get?.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) : Option β
```

The notation `m[a]?` is preferred over calling this function directly.

Tries to retrieve the mapping for the given key, returning none if no such mapping is present.

def

```
Std.ExtHashMap.getD.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (fallback : β) : β
```

Tries to retrieve the mapping for the given key, returning *fallback* if no such mapping is present.

def

```
Std.ExtHashMap.getKey.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (h : a ∈ m) : α
```

Retrieves the key from the mapping that matches a. Ensures that such a mapping exists by requiring a proof of $a \in m$. The result is guaranteed to be pointer equal to the key in the map.

def

```
Std.ExtHashMap.getKey!.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α] [Inhabited α]
  (m : Std.ExtHashMap α β) (a : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.ExtHashMap.getKey?.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) : Option α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise none. The result in the some case is guaranteed to be pointer equal to the key in the map.

def

```
Std.ExtHashMap.getKeyD.{u, v} {α : Type u} {β : Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a fallback : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise *fallback*. If a mapping exists the result is guaranteed to be pointer equal to the key in the map.

19.17.4.4. Modification

def

```
Std.ExtHashMap.alter.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (f : Option β → Option β) :
  Std.ExtHashMap α β
```

Modifies in place the value associated with a given key, allowing creating new values and deleting values via an Option valued replacement function.

This function ensures that the value is used linearly.

def

```
Std.ExtHashMap.modify.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (f : β → β) : Std.ExtHashMap α β
```

Modifies in place the value associated with a given key.

This function ensures that the value is used linearly.

def

```
Std.ExtHashMap.containsThenInsert.{u, v} {α : Type u} {β : Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
```

```
  (m : Std.ExtHashMap α β) (a : α) (b : β) : Bool × Std.ExtHashMap
α β
```

Checks whether a key is present in a map, and unconditionally inserts a value for the key.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.ExtHashMap.containsThenInsertIfNew.{u, v} {α : Type u} {β : Type
v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (b : β) : Bool × Std.ExtHashMap
α β
```

Checks whether a key is present in a map and inserts a value for the key if it was not found.

If the returned `Bool` is `true`, then the returned map is unaltered. If the `Bool` is `false`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `contains` followed by `insertIfNew`.

def

```
Std.ExtHashMap.erase.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) : Std.ExtHashMap α β
```

Removes the mapping for the given key if it exists.

def

```
Std.ExtHashMap.filter.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α] (f : α → β →
Bool)
  (m : Std.ExtHashMap α β) : Std.ExtHashMap α β
```

Removes all mappings of the hash map for which the given function returns `false`.

def

```
Std.ExtHashMap.filterMap.{u, v, w} {α : Type u} {β : Type v}
  {γ : Type w} {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α]
  [LawfulHashable α] (f : α → β → Option γ) (m : Std.ExtHashMap α
β) :
  Std.ExtHashMap α γ
```

Updates the values of the hash map by applying the given function to all mappings, keeping only those mappings where the function returns some value.

def

```
Std.ExtHashMap.insert.{u, v} {α : Type u} {β : Type v} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (b : β) : Std.ExtHashMap α β
```

Inserts the given mapping into the map. If there is already a mapping for the given key, then both key and value will be replaced.

Note: this replacement behavior is true for `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw`. The `insert` function on `HashSet` and `HashSet.Raw` behaves differently: it will return the set unchanged if a matching key is already present.

def

```
Std.ExtHashMap.insertIfNew.{u, v} {α : Type u} {β : Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (b : β) : Std.ExtHashMap α β
```

If there is no mapping for the given key, inserts the given mapping into the map. Otherwise, returns the map unaltered.

def

```
Std.ExtHashMap.getThenInsertIfNew?.{u, v} {α : Type u} {β : Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashMap α β) (a : α) (b : β) :
  Option β × Std.ExtHashMap α β
```

Checks whether a key is present in a map, returning the associate value, and inserts a value for the key if it was not found.

If the returned value is some v, then the returned map is unaltered. If it is none, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `get?` followed by `insertIfNew`.

def

```
Std.ExtHashMap.insertMany.{u, v, w} {α : Type u} {β : Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  {ρ : Type w} [ForIn Id ρ (α × β)] (m : Std.ExtHashMap α β) (l :
```

```
ρ) :
  Std.ExtHashMap α β
```

Inserts multiple mappings into the hash map by iterating over the given collection and calling `insert`. If the same key appears multiple times, the last occurrence takes precedence.

Note: this precedence behavior is true for `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw`. The `insertMany` function on `HashSet` and `HashSet.Raw` behaves differently: it will prefer the first appearance.

def

```
Std.ExtHashMap.insertManyIfNewUnit.{u, w} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α] {ρ : Type w}
  [ForIn Id ρ α] (m : Std.ExtHashMap α Unit) (l : ρ) :
  Std.ExtHashMap α Unit
```

Inserts multiple keys with the value `()` into the hash map by iterating over the given collection and calling `insertIfNew`. If the same key appears multiple times, the first occurrence takes precedence.

This is mainly useful to implement `HashSet.insertMany`, so if you are considering using this, `HashSet` or `HashSet.Raw` might be a better fit for you.

19.17.4.5. Iteration

def

```
Std.ExtHashMap.map.{u, v, w} {α : Type u} {β : Type v} {γ : Type w}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (f : α → β → γ) (m : Std.ExtHashMap α β) : Std.ExtHashMap α γ
```

Updates the values of the hash map by applying the given function to all mappings.

19.17.4.6. Conversion

def

```
Std.ExtHashMap.ofList.{u, v} {α : Type u} {β : Type v} [BEq α]
  [Hashable α] (l : List (α × β)) : Std.ExtHashMap α β
```

Creates a hash map from a list of mappings. If the same key appears multiple times, the last occurrence takes precedence.

def

```
Std.ExtHashMap.unitOfArray.{u} {α : Type u} [BEq α] [Hashable α]
  (l : Array α) : Std.ExtHashMap α Unit
```

Creates a hash map from an array of keys, associating the value `()` with each key.

This is mainly useful to implement `HashSet.ofArray`, so if you are considering using this, `HashSet` or `HashSet.Raw` might be a better fit for you.

def

```
Std.ExtHashMap.unitOfList.{u} {α : Type u} [BEq α] [Hashable α]
  (l : List α) : Std.ExtHashMap α Unit
```

Creates a hash map from a list of keys, associating the value `()` with each key.

This is mainly useful to implement `HashSet.ofList`, so if you are considering using this, `HashSet` or `HashSet.Raw` might be a better fit for you.

19.17.5. Extensional Dependent Hash Maps

The declarations in this section should be imported using `import Std.ExtDHashMap`.

structure

```
Std.ExtDHashMap.{u, v} (α : Type u) (β : α → Type v) [BEq α]
  [Hashable α] : Type (max u v)
```

Extensional dependent hash maps.

This is a simple separate-chaining hash table. The data of the hash map consists of a cached size and an array of buckets, where each bucket is a linked list of key-value pais. The number of buckets is always a power of two. The hash map doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an `Array`. Users should make sure that the hash map is used linearly to avoid expensive copies.

The hash map uses == (provided by the `BEq` typeclass) to compare keys and `hash` (provided by the `Hashable` typeclass) to hash them. To ensure that the operations behave as expected, == must be an equivalence relation and `a == b` must imply `hash a = hash b` (see also the `EquivBEq` and `LawfulHashable` typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

In contrast to regular dependent hash maps, `Std.ExtDHashMap` offers several extensionality lemmas and therefore has more lemmas about equality of hash maps. This however also makes it lose the ability to iterate freely over the hash map.

These hash maps contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.DHashMap.Raw` and `Std.DHashMap.Raw.WF` unbundle the invariant from the hash map. When in doubt, prefer `DHashMap` over `DHashMap.Raw`.

19.17.5.1. Creation

def

```
Std.ExtDHashMap.emptyWithCapacity.{u, v} {α : Type u} {β : α → Type
v}
  [BEq α] [Hashable α] (capacity : Nat := 8) : Std.ExtDHashMap α β
```

Creates a new empty hash map. The optional parameter *capacity* can be supplied to presize the map so that it can hold the given number of mappings without reallocating. It is also possible to use the empty collection notations ∅ and `{}` to create an empty hash map with the default capacity.

19.17.5.2. Properties

def

```
Std.ExtDHashMap.size.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) : Nat
```

The number of mappings present in the hash map

def

```
Std.ExtDHashMap.isEmpty.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) : Bool
```

Returns `true` if the hash map contains no mappings.

Note that if your `BEq` instance is not reflexive or your `Hashable` instance is not lawful, then it is possible that this function returns `false` even though is not possible to get anything out of the hash map.

19.17.5.3. Queries

def

```
Std.ExtDHashMap.contains.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
```

```
  (m : Std.ExtDHashMap α β) (a : α) : Bool
```

Returns `true` if there is a mapping for the given key. There is also a `Prop`-valued version of this: `a ∈ m` is equivalent to `m.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` uses `==` for comparisons, while for hash maps, both use `==`.

def

```
Std.ExtDHashMap.get.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.ExtDHashMap α β) (a :
α)
  (h : a ∈ m) : β a
```

Retrieves the mapping for the given key. Ensures that such a mapping exists by requiring a proof of `a ∈ m`.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.ExtDHashMap.get!.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.ExtDHashMap α β) (a :
α)
  [Inhabited (β a)] : β a
```

Tries to retrieve the mapping for the given key, panicking if no such mapping is present.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.ExtDHashMap.get?.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.ExtDHashMap α β) (a :
α) :
  Option (β a)
```

Tries to retrieve the mapping for the given key, returning `none` if no such mapping is present.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.ExtDHashMap.getD.{u, v} {α : Type u} {β : α → Type v} {x✝ : BEq
α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.ExtDHashMap α β) (a :
α)
  (fallback : β a) : β a
```

Tries to retrieve the mapping for the given key, returning *fallback* if no such mapping is present.

Uses the LawfulBEq instance to cast the retrieved value to the correct type.

def

```
Std.ExtDHashMap.getKey.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a : α) (h : a ∈ m) : α
```

Retrieves the key from the mapping that matches *a*. Ensures that such a mapping exists by requiring a proof of *a ∈ m*. The result is guaranteed to be pointer equal to the key in the map.

def

```
Std.ExtDHashMap.getKey!.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  [Inhabited α] (m : Std.ExtDHashMap α β) (a : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.ExtDHashMap.getKey?.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a : α) : Option α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise none. The result in the some case is guaranteed to be pointer equal to the key in the map.

def

```
Std.ExtDHashMap.getKeyD.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a fallback : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise *fallback*. If a mapping exists the result is guaranteed to be pointer equal to the key in the map.

19.17.5.4. Modification

def

```
Std.ExtDHashMap.alter.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.ExtDHashMap α β) (a :
α)
  (f : Option (β a) → Option (β a)) : Std.ExtDHashMap α β
```

Modifies in place the value associated with a given key, allowing creating new values and deleting values via an `Option` valued replacement function.

This function ensures that the value is used linearly.

def

```
Std.ExtDHashMap.modify.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [LawfulBEq α] (m : Std.ExtDHashMap α β) (a :
α)
  (f : β a → β a) : Std.ExtDHashMap α β
```

Modifies in place the value associated with a given key.

This function ensures that the value is used linearly.

def

```
Std.ExtDHashMap.containsThenInsert.{u, v} {α : Type u} {β : α → Type
v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a : α) (b : β a) :
  Bool × Std.ExtDHashMap α β
```

Checks whether a key is present in a map, and unconditionally inserts a value for the key.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.ExtDHashMap.containsThenInsertIfNew.{u, v} {α : Type u}
  {β : α → Type v} {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α]
```

```
  [LawfulHashable α] (m : Std.ExtDHashMap α β) (a : α) (b : β a) :
  Bool × Std.ExtDHashMap α β
```

Checks whether a key is present in a map and inserts a value for the key if it was not found.

If the returned `Bool` is `true`, then the returned map is unaltered. If the `Bool` is `false`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `contains` followed by `insertIfNew`.

def

```
Std.ExtDHashMap.erase.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a : α) : Std.ExtDHashMap α β
```

Removes the mapping for the given key if it exists.

def

```
Std.ExtDHashMap.filter.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (f : (a : α) → β a → Bool) (m : Std.ExtDHashMap α β) :
  Std.ExtDHashMap α β
```

Removes all mappings of the hash map for which the given function returns `false`.

def

```
Std.ExtDHashMap.filterMap.{u, v, w} {α : Type u} {β : α → Type v}
  {γ : α → Type w} {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α]
  [LawfulHashable α] (f : (a : α) → β a → Option (γ a))
  (m : Std.ExtDHashMap α β) : Std.ExtDHashMap α γ
```

Updates the values of the hash map by applying the given function to all mappings, keeping only those mappings where the function returns `some` value.

def

```
Std.ExtDHashMap.insert.{u, v} {α : Type u} {β : α → Type v} {x✝ :
BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a : α) (b : β a) : Std.ExtDHashMap α β
```

Inserts the given mapping into the map. If there is already a mapping for the given key, then both key and value will be replaced.

Note: this replacement behavior is true for `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw`. The `insert` function on `HashSet` and `HashSet.Raw` behaves differently: it will return the set unchanged if a matching key is already present.

def

```
Std.ExtDHashMap.insertIfNew.{u, v} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtDHashMap α β) (a : α) (b : β a) : Std.ExtDHashMap α β
```

If there is no mapping for the given key, inserts the given mapping into the map. Otherwise, returns the map unaltered.

def

```
Std.ExtDHashMap.getThenInsertIfNew?.{u, v} {α : Type u} {β : α →
Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [LawfulBEq α]
  (m : Std.ExtDHashMap α β) (a : α) (b : β a) :
  Option (β a) × Std.ExtDHashMap α β
```

Checks whether a key is present in a map, returning the associated value, and inserts a value for the key if it was not found.

If the returned value is `some v`, then the returned map is unaltered. If it is `none`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `get?` followed by `insertIfNew`.

Uses the `LawfulBEq` instance to cast the retrieved value to the correct type.

def

```
Std.ExtDHashMap.insertMany.{u, v, w} {α : Type u} {β : α → Type v}
  {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  {ρ : Type w} [ForIn Id ρ ((a : α) × β a)] (m : Std.ExtDHashMap α
β)
  (l : ρ) : Std.ExtDHashMap α β
```

Inserts multiple mappings into the hash map by iterating over the given collection and calling `insert`. If the same key appears multiple times, the last occurrence takes precedence.

Note: this precedence behavior is true for HashMap, DHashMap, HashMap.Raw and DHashMap.Raw. The insertMany function on HashSet and HashSet.Raw behaves differently: it will prefer the first appearance.

19.17.5.5. Iteration

def

```
Std.ExtDHashMap.map.{u, v, w} {α : Type u} {β : α → Type v}
  {γ : α → Type w} {x✝ : BEq α} {x✝¹ : Hashable α} [EquivBEq α]
  [LawfulHashable α] (f : (a : α) → β a → γ a)
  (m : Std.ExtDHashMap α β) : Std.ExtDHashMap α γ
```

Updates the values of the hash map by applying the given function to all mappings.

19.17.5.6. Conversion

def

```
Std.ExtDHashMap.ofList.{u, v} {α : Type u} {β : α → Type v} [BEq α]
  [Hashable α] (l : List ((a : α) × β a)) : Std.ExtDHashMap α β
```

Creates a hash map from a list of mappings. If the same key appears multiple times, the last occurrence takes precedence.

19.17.6. Hash Sets

structure

```
Std.HashSet.{u} (α : Type u) [BEq α] [Hashable α] : Type u
```

Hash sets.

This is a simple separate-chaining hash table. The data of the hash set consists of a cached size and an array of buckets, where each bucket is a linked list of keys. The number of buckets is always a power of two. The hash set doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an Array. Users should make sure that the hash set is used linearly to avoid expensive copies.

The hash set uses == (provided by the BEq typeclass) to compare elements and hash (provided by the Hashable typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and a == b should imply hash a = hash b (see also the EquivBEq and LawfulHashable typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if a == b implies $a = b$.

These hash sets contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.Data.HashSet.Raw` and `Std.Data.HashSet.Raw.WF` unbundle the invariant from the hash set. When in doubt, prefer `HashSet` over `HashSet.Raw`.

Constructor

`Std.HashSet.mk.{u}`

Fields

`inner : Std.HashMap α Unit`

Internal implementation detail of the hash set.

19.17.6.1. Creation

def

```
Std.HashSet.emptyWithCapacity.{u} {α : Type u} [BEq α] [Hashable α]
  (capacity : Nat := 8) : Std.HashSet α
```

Creates a new empty hash set. The optional parameter *capacity* can be supplied to presize the set so that it can hold the given number of elements without reallocating. It is also possible to use the empty collection notations ∅ and `{}` to create an empty hash set with the default capacity.

19.17.6.2. Properties

def

```
Std.HashSet.isEmpty.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  (m : Std.HashSet α) : Bool
```

Returns `true` if the hash set contains no elements.

Note that if your `BEq` instance is not reflexive or your `Hashable` instance is not lawful, then it is possible that this function returns `false` even though `m.contains a = false` for all *a*.

def

```
Std.HashSet.size.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  (m : Std.HashSet α) : Nat
```

The number of elements present in the set

structure

```
Std.HashSet.Equiv.{u} {α : Type u} {✝ : BEq α} {✝¹ : Hashable α}
  (m₁ m₂ : Std.HashSet α) : Prop
```

Two hash sets are equivalent in the sense of `Equiv` iff all their values are equal.

Constructor

`Std.HashSet.Equiv.mk.{u}`

Fields

`inner : m₁.inner.Equiv m₂.inner`

Internal implementation detail of the hash map

syntax

Equivalence

The relation `HashSet.Equiv` can also be written with an infix operator, which is scoped to its namespace:

```
term ::= ...
    | term ~m term
```

19.17.6.3. Queries

def

```
Std.HashSet.contains.{u} {α : Type u} {✝ : BEq α} {✝¹ : Hashable
α}
  (m : Std.HashSet α) (a : α) : Bool
```

Returns `true` if the given key is present in the set. There is also a `Prop`-valued version of this: `a ∈ m` is equivalent to `m.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` use `==` for comparisons, while for hash sets, both use `==`.

def

```
Std.HashSet.get.{u} {α : Type u} [BEq α] [Hashable α]
  (m : Std.HashSet α) (a : α) (h : a ∈ m) : α
```

Retrieves the key from the set that matches `a`. Ensures that such a key exists by requiring a proof of `a ∈ m`. The result is guaranteed to be pointer equal to the key in the set.

def

```
Std.HashSet.get!.{u} {α : Type u} [BEq α] [Hashable α] [Inhabited
α]
  (m : Std.HashSet α) (a : α) : α
```

Checks if given key is contained and returns the key if it is, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the set.

def

```
Std.HashSet.get?.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  (m : Std.HashSet α) (a : α) : Option α
```

Checks if given key is contained and returns the key if it is, otherwise none. The result in the some case is guaranteed to be pointer equal to the key in the set.

def

```
Std.HashSet.getD.{u} {α : Type u} [BEq α] [Hashable α]
  (m : Std.HashSet α) (a fallback : α) : α
```

Checks if given key is contained and returns the key if it is, otherwise *fallback*. If they key is contained the result is guaranteed to be pointer equal to the key in the set.

19.17.6.4. Modification

def

```
Std.HashSet.insert.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  (m : Std.HashSet α) (a : α) : Std.HashSet α
```

Inserts the given element into the set. If the hash set already contains an element that is equal (with regard to ==) to the given element, then the hash set is returned unchanged.

Note: this non-replacement behavior is true for `HashSet` and `HashSet.Raw`. The `insert` function on `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw` behaves differently: it will overwrite an existing mapping.

def

```
Std.HashSet.insertMany.{u, v} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} {ρ : Type v} [ForIn Id ρ α] (m : Std.HashSet
α)
  (l : ρ) : Std.HashSet α
```

Inserts multiple mappings into the hash set by iterating over the given collection and calling `insert`. If the same key appears multiple times, the first occurrence takes precedence.

Note: this precedence behavior is true for `HashSet` and `HashSet.Raw`. The `insertMany` function on `HashMap`, `DHashMap`, `HashMap.Raw` and `DHashMap.Raw` behaves differently: it will prefer the last appearance.

def

```
Std.HashSet.erase.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  (m : Std.HashSet α) (a : α) : Std.HashSet α
```

Removes the element if it exists.

def

```
Std.HashSet.filter.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  (f : α → Bool) (m : Std.HashSet α) : Std.HashSet α
```

Removes all elements from the hash set for which the given function returns `false`.

def

```
Std.HashSet.containsThenInsert.{u} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} (m : Std.HashSet α) (a : α) : Bool ×
Std.HashSet α
```

Checks whether an element is present in a set and inserts the element if it was not found. If the hash set already contains an element that is equal (with regard to ==) to the given element, then the hash set is returned unchanged.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.HashSet.partition.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  (f : α → Bool) (m : Std.HashSet α) : Std.HashSet α × Std.HashSet
α
```

Partition a hashset into two hashsets based on a predicate.

def

```
Std.HashSet.union.{u} {α : Type u} [BEq α] [Hashable α]
  (m₁ m₂ : Std.HashSet α) : Std.HashSet α
```

Computes the union of the given hash sets, by traversing m_2 and inserting its elements into m_1.

19.17.6.5. Iteration

def

```
Std.HashSet.all.{u} {α : Type u} {✝ : BEq α} {✝¹ : Hashable α}
  (m : Std.HashSet α) (p : α → Bool) : Bool
```

Check if all elements satisfy the predicate, short-circuiting if a predicate fails.

def

```
Std.HashSet.any.{u} {α : Type u} {✝ : BEq α} {✝¹ : Hashable α}
  (m : Std.HashSet α) (p : α → Bool) : Bool
```

Check if any element satisfies the predicate, short-circuiting if a predicate succeeds.

def

```
Std.HashSet.fold.{u, v} {α : Type u} {✝ : BEq α} {✝¹ : Hashable
α}
  {β : Type v} (f : β → α → β) (init : β) (m : Std.HashSet α) : β
```

Folds the given function over the elements of the hash set in some order.

def

```
Std.HashSet.foldM.{u, v} {α : Type u} {✝ : BEq α} {✝¹ : Hashable
α}
  {m : Type v → Type v} [Monad m] {β : Type v} (f : β → α → m β)
  (init : β) (b : Std.HashSet α) : m β
```

Monadically computes a value by folding the given function over the elements in the hash set in some order.

def

```
Std.HashSet.forIn.{u, v} {α : Type u} {✝ : BEq α} {✝¹ : Hashable
α}
  {m : Type v → Type v} [Monad m] {β : Type v}
  (f : α → β → m (ForInStep β)) (init : β) (b : Std.HashSet α) : m
β
```

Support for the **`for`** loop construct in **`do`** blocks.

def

```
Std.HashSet.forM.{u, v} {α : Type u} {✝ : BEq α} {✝¹ : Hashable
α}
```

```
  {m : Type v → Type v} [Monad m] (f : α → m PUnit)
  (b : Std.HashSet α) : m PUnit
```

Carries out a monadic action on each element in the hash set in some order.

19.17.6.6. Conversion

def

```
Std.HashSet.ofList.{u} {α : Type u} [BEq α] [Hashable α] (l : List
α) :
  Std.HashSet α
```

Creates a hash set from a list of elements. Note that unlike repeatedly calling `insert`, if the collection contains multiple elements that are equal (with regard to ==), then the last element in the collection will be present in the returned hash set.

def

```
Std.HashSet.toList.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  (m : Std.HashSet α) : List α
```

Transforms the hash set into a list of elements in some order.

def

```
Std.HashSet.ofArray.{u} {α : Type u} [BEq α] [Hashable α]
  (l : Array α) : Std.HashSet α
```

Creates a hash set from an array of elements. Note that unlike repeatedly calling `insert`, if the collection contains multiple elements that are equal (with regard to ==), then the last element in the collection will be present in the returned hash set.

def

```
Std.HashSet.toArray.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  (m : Std.HashSet α) : Array α
```

Transforms the hash set into an array of elements in some order.

19.17.6.7. Unbundled Variants

Unbundled maps separate well-formedness proofs from data. This is primarily useful when defining nested inductive types. To use these variants, import the modules `Std.HashSet.Raw` and `Std.HashSet.RawLemmas`.

structure

```
Std.HashSet.Raw.{u} (α : Type u) : Type u
```

Hash sets without a bundled well-formedness invariant, suitable for use in nested inductive types. The well-formedness invariant is called `Raw.WF`. When in doubt, prefer `HashSet` over `HashSet.Raw`. Lemmas about the operations on `Std.Data.HashSet.Raw` are available in the module `Std.Data.HashSet.RawLemmas`.

This is a simple separate-chaining hash table. The data of the hash set consists of a cached size and an array of buckets, where each bucket is a linked list of keys. The number of buckets is always a power of two. The hash set doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an `Array`. Users should make sure that the hash set is used linearly to avoid expensive copies.

The hash set uses == (provided by the `BEq` typeclass) to compare elements and `hash` (provided by the `Hashable` typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and `a == b` should imply `hash a = hash b` (see also the `EquivBEq` and `LawfulHashable` typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

Constructor

`Std.HashSet.Raw.mk.{u}`

Fields

`inner : Std.HashMap.Raw α Unit`

Internal implementation detail of the hash set.

structure

```
Std.HashSet.Raw.WF.{u} {α : Type u} [BEq α] [Hashable α]
  (m : Std.HashSet.Raw α) : Prop
```

Well-formedness predicate for hash sets. Users of `HashSet` will not need to interact with this. Users of `HashSet.Raw` will need to provide proofs of `WF` to lemmas and should use lemmas like `WF.empty` and `WF.insert` (which are always named exactly like the operations they are about) to show that set operations preserve well-formedness.

Constructor

`Std.HashSet.Raw.WF.mk.{u}`

Fields

`out : m.inner.WF`

Internal implementation detail of the hash set

19.17.7. Extensional Hash Sets

structure

`Std.ExtHashSet.{u} (α : Type u) [BEq α] [Hashable α] : Type u`

Hash sets.

This is a simple separate-chaining hash table. The data of the hash set consists of a cached size and an array of buckets, where each bucket is a linked list of keys. The number of buckets is always a power of two. The hash set doubles its size upon inserting an element such that the number of elements is more than 75% of the number of buckets.

The hash table is backed by an `Array`. Users should make sure that the hash set is used linearly to avoid expensive copies.

The hash set uses == (provided by the `BEq` typeclass) to compare elements and `hash` (provided by the `Hashable` typeclass) to hash them. To ensure that the operations behave as expected, == should be an equivalence relation and `a == b` should imply `hash a = hash b` (see also the `EquivBEq` and `LawfulHashable` typeclasses). Both of these conditions are automatic if the BEq instance is lawful, i.e., if `a == b` implies $a = b$.

In contrast to regular hash sets, `Std.ExtHashSet` offers several extensionality lemmas and therefore has more lemmas about equality of hash maps. This however also makes it lose the ability to iterate freely over hash sets.

These hash sets contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.HashSet.Raw` and `Std.HashSet.Raw.WF` unbundle the invariant from the hash set. When in doubt, prefer `HashSet` or `ExtHashSet` over `HashSet.Raw`.

Constructor

`Std.ExtHashSet.mk.{u}`

Fields

`inner : Std.ExtHashMap α Unit`

Internal implementation detail of the hash set.

19.17.7.1. Creation

def

```
Std.ExtHashSet.emptyWithCapacity.{u} {α : Type u} [BEq α] [Hashable
α]
  (capacity : Nat := 8) : Std.ExtHashSet α
```

Creates a new empty hash set. The optional parameter *capacity* can be supplied to presize the set so that it can hold the given number of elements without reallocating. It is also possible to use the empty collection notations ∅ and {} to create an empty hash set with the default capacity.

19.17.7.2. Properties

def

```
Std.ExtHashSet.isEmpty.{u} {α : Type u} {x✝ : BEq α} {x✝¹ :
Hashable α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) : Bool
```

Returns `true` if the hash set contains no elements.

Note that if your `BEq` instance is not reflexive or your `Hashable` instance is not lawful, then it is possible that this function returns `false` even though `m.contains a = false` for all *a*.

def

```
Std.ExtHashSet.size.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) : Nat
```

The number of elements present in the set

19.17.7.3. Queries

def

```
Std.ExtHashSet.contains.{u} {α : Type u} {x✝ : BEq α} {x✝¹ :
Hashable α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) (a : α) :
Bool
```

Returns `true` if the given key is present in the set. There is also a `Prop`-valued version of this: *a* ∈ *m* is equivalent to `m.contains a = true`.

Observe that this is different behavior than for lists: for lists, ∈ uses = and `contains` use == for comparisons, while for hash sets, both use ==.

def

```
Std.ExtHashSet.get.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) (a : α)
  (h : a ∈ m) : α
```

Retrieves the key from the set that matches *a*. Ensures that such a key exists by requiring a proof of *a ∈ m*. The result is guaranteed to be pointer equal to the key in the set.

def

```
Std.ExtHashSet.get!.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] [Inhabited α] (m : Std.ExtHashSet
α)
  (a : α) : α
```

Checks if given key is contained and returns the key if it is, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the set.

def

```
Std.ExtHashSet.get?.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) (a : α) :
  Option α
```

Checks if given key is contained and returns the key if it is, otherwise `none`. The result in the `some` case is guaranteed to be pointer equal to the key in the set.

def

```
Std.ExtHashSet.getD.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α)
  (a fallback : α) : α
```

Checks if given key is contained and returns the key if it is, otherwise *fallback*. If they key is contained the result is guaranteed to be pointer equal to the key in the set.

19.17.7.4. Modification

def

```
Std.ExtHashSet.insert.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) (a : α) :
  Std.ExtHashSet α
```

Inserts the given element into the set. If the hash set already contains an element that is equal (with regard to ==) to the given element, then the hash set is returned unchanged.

Note: this non-replacement behavior is true for `ExtHashSet` and `ExtHashSet.Raw`. The `insert` function on `ExtHashMap`, `DExtHashMap`, `ExtHashMap.Raw` and `DExtHashMap.Raw` behaves differently: it will overwrite an existing mapping.

def

```
Std.ExtHashSet.insertMany.{u, v} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α] {ρ : Type v}
  [ForIn Id ρ α] (m : Std.ExtHashSet α) (l : ρ) : Std.ExtHashSet α
```

Inserts multiple mappings into the hash set by iterating over the given collection and calling `insert`. If the same key appears multiple times, the first occurrence takes precedence.

Note: this precedence behavior is true for `ExtHashSet` and `ExtHashSet.Raw`. The `insertMany` function on `ExtHashMap`, `DExtHashMap`, `ExtHashMap.Raw` and `DExtHashMap.Raw` behaves differently: it will prefer the last appearance.

def

```
Std.ExtHashSet.erase.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] (m : Std.ExtHashSet α) (a : α) :
  Std.ExtHashSet α
```

Removes the element if it exists.

def

```
Std.ExtHashSet.filter.{u} {α : Type u} {x✝ : BEq α} {x✝¹ : Hashable
α}
  [EquivBEq α] [LawfulHashable α] (f : α → Bool)
  (m : Std.ExtHashSet α) : Std.ExtHashSet α
```

Removes all elements from the hash set for which the given function returns `false`.

def

```
Std.ExtHashSet.containsThenInsert.{u} {α : Type u} {x✝ : BEq α}
  {x✝¹ : Hashable α} [EquivBEq α] [LawfulHashable α]
  (m : Std.ExtHashSet α) (a : α) : Bool × Std.ExtHashSet α
```

Checks whether an element is present in a set and inserts the element if it was not found. If the hash set already contains an element that is equal (with regard to ==) to the given element, then the hash set is returned unchanged.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

19.17.7.5. Conversion

def

```
Std.ExtHashSet.ofList.{u} {α : Type u} [BEq α] [Hashable α]
  (l : List α) : Std.ExtHashSet α
```

Creates a hash set from a list of elements. Note that unlike repeatedly calling `insert`, if the collection contains multiple elements that are equal (with regard to ==), then the last element in the collection will be present in the returned hash set.

def

```
Std.ExtHashSet.ofArray.{u} {α : Type u} [BEq α] [Hashable α]
  (l : Array α) : Std.ExtHashSet α
```

Creates a hash set from an array of elements. Note that unlike repeatedly calling `insert`, if the collection contains multiple elements that are equal (with regard to ==), then the last element in the collection will be present in the returned hash set.

19.17.8. Tree-Based Maps

The declarations in this section should be imported using `import Std.TreeMap`.

structure

```
Std.TreeMap.{u, v} (α : Type u) (β : Type v)
  (cmp : α → α → Ordering := by exact compare) : Type (max u v)
```

Tree maps.

A tree map stores an assignment of keys to values. It depends on a comparator function that defines an ordering on the keys and provides efficient order-dependent queries, such as retrieval of the minimum or maximum.

To ensure that the operations behave as expected, the comparator function *cmp* should satisfy certain laws that ensure a consistent ordering:

- If a is less than (or equal) to b, then b is greater than (or equal) to a and vice versa (see the `OrientedCmp` typeclass).
- If a is less than or equal to b and b is, in turn, less than or equal to c, then a is less than or equal to c (see the `TransCmp` typeclass).

Keys for which *cmp a b* = `Ordering.eq` are considered the same, i.e., there can be only one entry with key either *a* or *b* in a tree map. Looking up either *a* or *b* always yields the same entry, if any is present.

To avoid expensive copies, users should make sure that the tree map is used linearly.

Internally, the tree maps are represented as size-bounded trees, a type of self-balancing binary search tree with efficient order statistic lookups.

For use in proofs, the type `Std.ExtTreeMap` of extensional tree maps should be preferred. This type comes with several extensionality lemmas and provides the same functions but requires a `TransCmp` instance to work with.

These tree maps contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.TreeMap.Raw` and `Std.TreeMap.Raw.WF` unbundle the invariant from the tree map. When in doubt, prefer `TreeMap` over `TreeMap.Raw`.

19.17.8.1. Creation

def

```
Std.TreeMap.empty.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} : Std.TreeMap α β cmp
```

Creates a new empty tree map. It is also possible and recommended to use the empty collection notations ∅ and `{}` to create an empty tree map. **`simp`** replaces `empty` with ∅.

19.17.8.2. Properties

def

```
Std.TreeMap.size.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Nat
```

Returns the number of mappings present in the map.

def

```
Std.TreeMap.isEmpty.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Bool
```

Returns `true` if the tree map contains no mappings.

19.17.8.3. Queries

def

```
Std.TreeMap.contains.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (l : Std.TreeMap α β cmp) (a : α) : Bool
```

Returns `true` if there is a mapping for the given key *a* or a key that is equal to *a* according to the comparator *cmp*. There is also a `Prop`-valued version of this: `a ∈ t` is equivalent to `t.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` uses `==` for equality checks, while for tree maps, both use the given comparator *cmp*.

def

```
Std.TreeMap.get.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α)
  (h : a ∈ t) : β
```

Given a proof that a mapping for the given key is present, retrieves the mapping for the given key.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.TreeMap.get!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α)
  [Inhabited β] : β
```

Tries to retrieve the mapping for the given key, panicking if no such mapping is present.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.TreeMap.get?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) :
Option β
```

Tries to retrieve the mapping for the given key, returning none if no such mapping is present.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.TreeMap.getD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α)
  (fallback : β) : β
```

Tries to retrieve the mapping for the given key, returning *fallback* if no such mapping is present.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.TreeMap.getKey.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α)
  (h : a ∈ t) : α
```

Retrieves the key from the mapping that matches *a*. Ensures that such a mapping exists by requiring a proof of `a ∈ m`. The result is guaranteed to be pointer equal to the key in the map.

def

```
Std.TreeMap.getKey!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
  (a : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.TreeMap.getKey?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) :
Option α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise none. The result in the some case is guaranteed to be pointer equal to the key in the map.

def

```
Std.TreeMap.getKeyD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a fallback :
```

```
α) :
  α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise `fallback`. If a mapping exists the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.TreeMap.keys.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : List α
```

Returns a list of all keys present in the tree map in ascending order.

def

```
Std.TreeMap.keysArray.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Array α
```

Returns an array of all keys present in the tree map in ascending order.

def

```
Std.TreeMap.values.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : List β
```

Returns a list of all values present in the tree map in ascending order.

def

```
Std.TreeMap.valuesArray.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Array β
```

Returns an array of all values present in the tree map in ascending order.

19.17.8.3.1. Ordering-Based Queries

def

```
Std.TreeMap.entryAtIdx.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (n : Nat)
  (h : n ≤ t.size) : α × β
```

Returns the key-value pair with the *n*-th smallest key.

def

```
Std.TreeMap.entryAtIdx!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)] (t : Std.TreeMap α β
cmp)
  (n : Nat) : α × β
```

Returns the key-value pair with the *n*-th smallest key, or panics if *n* is at least `t.size`.

def

```
Std.TreeMap.entryAtIdx?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (n : Nat) :
  Option (α × β)
```

Returns the key-value pair with the *n*-th smallest key, or `none` if *n* is at least `t.size`.

def

```
Std.TreeMap.entryAtIdxD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (n : Nat)
  (fallback : α × β) : α × β
```

Returns the key-value pair with the *n*-th smallest key, or `fallback` if *n* is at least `t.size`.

def

```
Std.TreeMap.getEntryGE.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ (cmp a k).isGE = true) : α × β
```

Given a proof that such a mapping exists, retrieves the key-value pair with the smallest key that is greater than or equal to the given key.

def

```
Std.TreeMap.getEntryGE!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)] (t : Std.TreeMap α β
cmp)
  (k : α) : α × β
```

Tries to retrieve the key-value pair with the smallest key that is greater than or equal to the given key, panicking if no such pair exists.

def

```
Std.TreeMap.getEntryGE?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
  Option (α × β)
```

Tries to retrieve the key-value pair with the smallest key that is greater than or equal to the given key, returning none if no such pair exists.

def

```
Std.TreeMap.getEntryGED.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α)
  (fallback : α × β) : α × β
```

Tries to retrieve the key-value pair with the smallest key that is greater than or equal to the given key, returning *fallback* if no such pair exists.

def

```
Std.TreeMap.getEntryGT.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ cmp a k = Ordering.gt) : α × β
```

Given a proof that such a mapping exists, retrieves the key-value pair with the smallest key that is greater than the given key.

def

```
Std.TreeMap.getEntryGT!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)] (t : Std.TreeMap α β
cmp)
  (k : α) : α × β
```

Tries to retrieve the key-value pair with the smallest key that is greater than the given key, panicking if no such pair exists.

def

```
Std.TreeMap.getEntryGT?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
  Option (α × β)
```

Tries to retrieve the key-value pair with the smallest key that is greater than the given key, returning none if no such pair exists.

def

```
Std.TreeMap.getEntryGTD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α)
  (fallback : α × β) : α × β
```

Tries to retrieve the key-value pair with the smallest key that is greater than the given key, returning *fallback* if no such pair exists.

def

```
Std.TreeMap.getEntryLE.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ (cmp a k).isLE = true) : α × β
```

Given a proof that such a mapping exists, retrieves the key-value pair with the largest key that is less than or equal to the given key.

def

```
Std.TreeMap.getEntryLE!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)] (t : Std.TreeMap α β
cmp)
  (k : α) : α × β
```

Tries to retrieve the key-value pair with the largest key that is less than or equal to the given key, panicking if no such pair exists.

def

```
Std.TreeMap.getEntryLE?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
  Option (α × β)
```

Tries to retrieve the key-value pair with the largest key that is less than or equal to the given key, returning none if no such pair exists.

def

```
Std.TreeMap.getEntryLED.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α)
  (fallback : α × β) : α × β
```

Tries to retrieve the key-value pair with the largest key that is less than or equal to the given key, returning *fallback* if no such pair exists.

def

```
Std.TreeMap.getEntryLT.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ cmp a k = Ordering.lt) : α × β
```

Given a proof that such a mapping exists, retrieves the key-value pair with the largest key that is less than the given key.

def

```
Std.TreeMap.getEntryLT!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)] (t : Std.TreeMap α β
cmp)
  (k : α) : α × β
```

Tries to retrieve the key-value pair with the largest key that is less than the given key, panicking if no such pair exists.

def

```
Std.TreeMap.getEntryLT?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
  Option (α × β)
```

Tries to retrieve the key-value pair with the largest key that is less than the given key, returning none if no such pair exists.

def

```
Std.TreeMap.getEntryLTD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α)
  (fallback : α × β) : α × β
```

Tries to retrieve the key-value pair with the largest key that is less than the given key, returning *fallback* if no such pair exists.

def

```
Std.TreeMap.getKeyGE.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ (cmp a k).isGE = true) : α
```

Given a proof that such a mapping exists, retrieves the smallest key that is greater than or equal to the given key.

def

```
Std.TreeMap.getKeyGE!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
  (k : α) : α
```

Tries to retrieve the smallest key that is greater than or equal to the given key, panicking if no such key exists.

def

```
Std.TreeMap.getKeyGE?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
Option α
```

Tries to retrieve the smallest key that is greater than or equal to the given key, returning none if no such key exists.

def

```
Std.TreeMap.getKeyGED.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k fallback :
α) :
  α
```

Tries to retrieve the smallest key that is greater than or equal to the given key, returning *fallback* if no such key exists.

def

```
Std.TreeMap.getKeyGT.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ cmp a k = Ordering.gt) : α
```

Given a proof that such a mapping exists, retrieves the smallest key that is greater than the given key.

def

```
Std.TreeMap.getKeyGT!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
  (k : α) : α
```

Tries to retrieve the smallest key that is greater than the given key, panicking if no such key exists.

def

```
Std.TreeMap.getKeyGT?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
Option α
```

Tries to retrieve the smallest key that is greater than the given key, returning none if no such key exists.

def

```
Std.TreeMap.getKeyGTD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k fallback :
α) :
  α
```

Tries to retrieve the smallest key that is greater than the given key, returning *fallback* if no such key exists.

def

```
Std.TreeMap.getKeyLE.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ (cmp a k).isLE = true) : α
```

Given a proof that such a mapping exists, retrieves the largest key that is less than or equal to the given key.

def

```
Std.TreeMap.getKeyLE!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
  (k : α) : α
```

Tries to retrieve the largest key that is less than or equal to the given key, panicking if no such key exists.

def

```
Std.TreeMap.getKeyLE?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
Option α
```

Tries to retrieve the largest key that is less than or equal to the given key, returning none if no such key exists.

def

```
Std.TreeMap.getKeyLED.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k fallback :
α) :
  α
```

Tries to retrieve the largest key that is less than or equal to the given key, returning *fallback* if no such key exists.

def

```
Std.TreeMap.getKeyLT.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Std.TransCmp cmp] (t : Std.TreeMap α β
cmp)
  (k : α) (h : ∃ a, a ∈ t ∧ cmp a k = Ordering.lt) : α
```

Given a proof that such a mapping exists, retrieves the largest key that is less than the given key.

def

```
Std.TreeMap.getKeyLT!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
  (k : α) : α
```

Tries to retrieve the largest key that is less than the given key, panicking if no such key exists.

def

```
Std.TreeMap.getKeyLT?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k : α) :
Option α
```

Tries to retrieve the largest key that is less than the given key, returning none if no such key exists.

def

```
Std.TreeMap.getKeyLTD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (k fallback :
α) :
  α
```

Tries to retrieve the largest key that is less than the given key, returning *fallback* if no such key exists.

def

```
Std.TreeMap.keyAtIdx.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (n : Nat)
  (h : n < t.size) : α
```

Returns the *n*-th smallest key.

def

```
Std.TreeMap.keyAtIdx!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
  (n : Nat) : α
```

Returns the *n*-th smallest key, or panics if *n* is at least `t.size`.

def

```
Std.TreeMap.keyAtIdx?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (n : Nat) :
  Option α
```

Returns the *n*-th smallest key, or `none` if *n* is at least `t.size`.

def

```
Std.TreeMap.keyAtIdxD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (n : Nat)
  (fallback : α) : α
```

Returns the *n*-th smallest key, or `fallback` if *n* is at least `t.size`.

def

```
Std.TreeMap.minEntry.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (h : t.isEmpty = false) : α × β
```

Given a proof that the tree map is not empty, retrieves the key-value pair with the smallest key.

def

```
Std.TreeMap.minEntry!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)]
  (t : Std.TreeMap α β cmp) : α × β
```

Tries to retrieve the key-value pair with the smallest key in the tree map, panicking if the map is empty.

def

```
Std.TreeMap.minEntry?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Option (α × β)
```

Tries to retrieve the key-value pair with the smallest key in the tree map, returning none if the map is empty.

def

```
Std.TreeMap.minEntryD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (fallback : α × β) : α × β
```

Tries to retrieve the key-value pair with the smallest key in the tree map, returning *fallback* if the tree map is empty.

def

```
Std.TreeMap.minKey.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (h : t.isEmpty = false) : α
```

Given a proof that the tree map is not empty, retrieves the smallest key.

def

```
Std.TreeMap.minKey!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
: α
```

Tries to retrieve the smallest key in the tree map, panicking if the map is empty.

def

```
Std.TreeMap.minKey?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Option α
```

Tries to retrieve the smallest key in the tree map, returning none if the map is empty.

def

```
Std.TreeMap.minKeyD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (fallback : α)
: α
```

Tries to retrieve the smallest key in the tree map, returning `fallback` if the tree map is empty.

def

```
Std.TreeMap.maxEntry.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (h : t.isEmpty = false) : α × β
```

Given a proof that the tree map is not empty, retrieves the key-value pair with the largest key.

def

```
Std.TreeMap.maxEntry!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited (α × β)]
  (t : Std.TreeMap α β cmp) : α × β
```

Tries to retrieve the key-value pair with the largest key in the tree map, panicking if the map is empty.

def

```
Std.TreeMap.maxEntry?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Option (α × β)
```

Tries to retrieve the key-value pair with the largest key in the tree map, returning none if the map is empty.

def

```
Std.TreeMap.maxEntryD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (fallback : α × β) : α × β
```

Tries to retrieve the key-value pair with the largest key in the tree map, returning `fallback` if the tree map is empty.

def

```
Std.TreeMap.maxKey.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (h : t.isEmpty = false) : α
```

Given a proof that the tree map is not empty, retrieves the largest key.

def

```
Std.TreeMap.maxKey!.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.TreeMap α β cmp)
: α
```

Tries to retrieve the largest key in the tree map, panicking if the map is empty.

def

```
Std.TreeMap.maxKey?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Option α
```

Tries to retrieve the largest key in the tree map, returning none if the map is empty.

def

```
Std.TreeMap.maxKeyD.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (fallback : α)
: α
```

Tries to retrieve the largest key in the tree map, returning *fallback* if the tree map is empty.

19.17.8.4. Modification

def

```
Std.TreeMap.alter.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α)
  (f : Option β → Option β) : Std.TreeMap α β cmp
```

Modifies in place the value associated with a given key, allowing creating new values and deleting values via an Option valued replacement function.

This function ensures that the value is used linearly.

def

```
Std.TreeMap.modify.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α)
  (f : β → β) : Std.TreeMap α β cmp
```

Modifies in place the value associated with a given key.

This function ensures that the value is used linearly.

def

```
Std.TreeMap.containsThenInsert.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) (b :
β) :
  Bool × Std.TreeMap α β cmp
```

Checks whether a key is present in a map and unconditionally inserts a value for the key.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.TreeMap.containsThenInsertIfNew.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) (b :
β) :
  Bool × Std.TreeMap α β cmp
```

Checks whether a key is present in a map and inserts a value for the key if it was not found. If the returned `Bool` is `true`, then the returned map is unaltered. If the `Bool` is `false`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `contains` followed by `insertIfNew`.

def

```
Std.TreeMap.erase.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) :
  Std.TreeMap α β cmp
```

Removes the mapping for the given key if it exists.

def

```
Std.TreeMap.eraseMany.{u, v, u_1} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {ρ : Type u_1} [ForIn Id ρ α]
  (t : Std.TreeMap α β cmp) (l : ρ) : Std.TreeMap α β cmp
```

Erases multiple mappings from the tree map by iterating over the given collection and calling `erase`.

def

```
Std.TreeMap.filter.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (f : α → β → Bool)
  (m : Std.TreeMap α β cmp) : Std.TreeMap α β cmp
```

Removes all mappings of the map for which the given function returns `false`.

def

```
Std.TreeMap.filterMap.{u, v, w} {α : Type u} {β : Type v} {γ : Type w}
  {cmp : α → α → Ordering} (f : α → β → Option γ)
  (m : Std.TreeMap α β cmp) : Std.TreeMap α γ cmp
```

Updates the values of the map by applying the given function to all mappings, keeping only those mappings where the function returns `some` value.

def

```
Std.TreeMap.insert.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (l : Std.TreeMap α β cmp) (a : α) (b : β) :
  Std.TreeMap α β cmp
```

Inserts the given mapping into the map. If there is already a mapping for the given key, then both key and value will be replaced.

def

```
Std.TreeMap.insertIfNew.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) (b : β) :
  Std.TreeMap α β cmp
```

If there is no mapping for the given key, inserts the given mapping into the map. Otherwise, returns the map unaltered.

def

```
Std.TreeMap.getThenInsertIfNew?.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) (a : α) (b : β) :
  Option β × Std.TreeMap α β cmp
```

Checks whether a key is present in a map, returning the associated value, and inserts a value for the key if it was not found.

If the returned value is `some v`, then the returned map is unaltered. If it is `none`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `get?` followed by `insertIfNew`.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.TreeMap.insertMany.{u, v, u_1} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {ρ : Type u_1} [ForIn Id ρ (α × β)]
  (t : Std.TreeMap α β cmp) (l : ρ) : Std.TreeMap α β cmp
```

Inserts multiple mappings into the tree map by iterating over the given collection and calling `insert`. If the same key appears multiple times, the last occurrence takes precedence.

Note: this precedence behavior is true for `TreeMap`, `DTreeMap`, `TreeMap.Raw` and `DTreeMap.Raw`. The `insertMany` function on `TreeSet` and `TreeSet.Raw` behaves differently: it will prefer the first appearance.

def

```
Std.TreeMap.insertManyIfNewUnit.{u, u_1} {α : Type u}
  {cmp : α → α → Ordering} {ρ : Type u_1} [ForIn Id ρ α]
  (t : Std.TreeMap α Unit cmp) (l : ρ) : Std.TreeMap α Unit cmp
```

Inserts multiple elements into the tree map by iterating over the given collection and calling `insertIfNew`. If the same key appears multiple times, the first occurrence takes precedence.

def

```
Std.TreeMap.mergeWith.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (mergeFn : α → β → β → β)
  (t₁ t₂ : Std.TreeMap α β cmp) : Std.TreeMap α β cmp
```

Returns a map that contains all mappings of t_1 and t_2. In case that both maps contain the same key k with respect to *cmp*, the provided function is used to determine the new value from the respective values in t_1 and t_2.

This function ensures that t_1 is used linearly. It also uses the individual values in t_1 linearly if the merge function uses the second argument (i.e. the first of type β a) linearly. Hence, as long as t_1 is unshared, the performance characteristics follow the following imperative description: Iterate over all mappings in t_2, inserting them into t_1 if t_1 does not contain a conflicting mapping yet. If t_1 does contain a conflicting mapping, use the given merge function to merge the mapping in t_2 into the mapping in t_1. Then return t_1.

Hence, the runtime of this method scales logarithmically in the size of t_1 and linearly in the size of t_2 as long as t_1 is unshared.

def

```
Std.TreeMap.partition.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (f : α → β → Bool)
  (t : Std.TreeMap α β cmp) : Std.TreeMap α β cmp × Std.TreeMap α β
cmp
```

Partitions a tree map into two tree maps based on a predicate.

19.17.8.5. Iteration

def

```
Std.TreeMap.map.{u, v, w} {α : Type u} {β : Type v} {γ : Type w}
  {cmp : α → α → Ordering} (f : α → β → γ) (t : Std.TreeMap α β
cmp) :
  Std.TreeMap α γ cmp
```

Updates the values of the map by applying the given function to all mappings.

def

```
Std.TreeMap.all.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (p : α → β → Bool) : Bool
```

Check if all elements satisfy the predicate, short-circuiting if a predicate fails.

def

```
Std.TreeMap.any.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp)
  (p : α → β → Bool) : Bool
```

Check if any element satisfes the predicate, short-circuiting if a predicate fails.

def

```
Std.TreeMap.foldl.{u, v, w} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {δ : Type w} (f : δ → α → β → δ) (init :
δ)
  (t : Std.TreeMap α β cmp) : δ
```

Folds the given function over the mappings in the map in ascending order.

def

```
Std.TreeMap.foldlM.{u, v, w, w₂} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {δ : Type w} {m : Type w → Type w₂}
```

```
[Monad m]
  (f : δ → α → β → m δ) (init : δ) (t : Std.TreeMap α β cmp) : m δ
```

Folds the given monadic function over the mappings in the map in ascending order.

def

```
Std.TreeMap.foldr.{u, v, w} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {δ : Type w} (f : α → β → δ → δ) (init :
δ)
  (t : Std.TreeMap α β cmp) : δ
```

Folds the given function over the mappings in the map in descending order.

def

```
Std.TreeMap.foldrM.{u, v, w, w₂} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {δ : Type w} {m : Type w → Type w₂}
[Monad m]
  (f : α → β → δ → m δ) (init : δ) (t : Std.TreeMap α β cmp) : m δ
```

Folds the given monadic function over the mappings in the map in descending order.

def

```
Std.TreeMap.forIn.{u, v, w, w₂} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {δ : Type w} {m : Type w → Type w₂}
[Monad m]
  (f : α → β → δ → m (ForInStep δ)) (init : δ)
  (t : Std.TreeMap α β cmp) : m δ
```

Support for the **`for`** loop construct in **`do`** blocks. Iteration happens in ascending order.

def

```
Std.TreeMap.forM.{u, v, w, w₂} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} {m : Type w → Type w₂} [Monad m]
  (f : α → β → m PUnit) (t : Std.TreeMap α β cmp) : m PUnit
```

Carries out a monadic action on each mapping in the tree map in ascending order.

19.17.8.6. Conversion

def

```
Std.TreeMap.ofList.{u, v} {α : Type u} {β : Type v} (l : List (α ×
β).)
```

```
  (cmp : α → α → Ordering := by exact compare) : Std.TreeMap α β
cmp
```

Transforms a list of mappings into a tree map.

def

```
Std.TreeMap.toList.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : List (α × β)
```

Transforms the tree map into a list of mappings in ascending order.

def

```
Std.TreeMap.ofArray.{u, v} {α : Type u} {β : Type v} (a : Array (α
× β))
  (cmp : α → α → Ordering := by exact compare) : Std.TreeMap α β
cmp
```

Transforms a list of mappings into a tree map.

def

```
Std.TreeMap.toArray.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap α β cmp) : Array (α ×
β)
```

Transforms the tree map into a list of mappings in ascending order.

def

```
Std.TreeMap.unitOfArray.{u} {α : Type u} (a : Array α)
  (cmp : α → α → Ordering := by exact compare) : Std.TreeMap α Unit
cmp
```

Transforms an array of keys into a tree map.

def

```
Std.TreeMap.unitOfList.{u} {α : Type u} (l : List α)
  (cmp : α → α → Ordering := by exact compare) : Std.TreeMap α Unit
cmp
```

Transforms a list of keys into a tree map.

19.17.8.6.1. Unbundled Variants

Unbundled maps separate well-formedness proofs from data. This is primarily useful when defining nested inductive types. To use these variants, import the module `Std.TreeMap.Raw`.

structure

```
Std.TreeMap.Raw.{u, v} (α : Type u) (β : Type v)
  (cmp : α → α → Ordering := by exact compare) : Type (max u v)
```

Tree maps without a bundled well-formedness invariant, suitable for use in nested inductive types. The well-formedness invariant is called `Raw.WF`. When in doubt, prefer `TreeMap` over `TreeMap.Raw`. Lemmas about the operations on `Std.TreeMap.Raw` are available in the module `Std.Data.TreeMap.Raw.Lemmas`.

A tree map stores an assignment of keys to values. It depends on a comparator function that defines an ordering on the keys and provides efficient order-dependent queries, such as retrieval of the minimum or maximum.

To ensure that the operations behave as expected, the comparator function *cmp* should satisfy certain laws that ensure a consistent ordering:

- If a is less than (or equal) to b, then b is greater than (or equal) to a and vice versa (see the `OrientedCmp` typeclass).
- If a is less than or equal to b and b is, in turn, less than or equal to c, then a is less than or equal to c (see the `TransCmp` typeclass).

Keys for which *cmp a b* = `Ordering.eq` are considered the same, i.e., there can be only one entry with key either *a* or *b* in a tree map. Looking up either *a* or *b* always yields the same entry, if any is present.

To avoid expensive copies, users should make sure that the tree map is used linearly.

Internally, the tree maps are represented as size-bounded trees, a type of self-balancing binary search tree with efficient order statistic lookups.

Constructor

`Std.TreeMap.Raw.mk.{u, v}`

Fields

`inner : Std.DTreeMap.Raw α (fun x => β) cmp`

Internal implementation detail of the tree map.

structure

```
Std.TreeMap.Raw.WF.{u, v} {α : Type u} {β : Type v}
  {cmp : α → α → Ordering} (t : Std.TreeMap.Raw α β cmp) : Prop
```

Well-formedness predicate for tree maps. Users of `TreeMap` will not need to interact with this. Users of `TreeMap.Raw` will need to provide proofs of `WF` to lemmas and should use lemmas like `WF.empty` and `WF.insert` (which are always named exactly like the operations they are about) to show that map operations preserve well-formedness. The constructors of this type are internal implementation details and should not be accessed by users.

Constructor

`Std.TreeMap.Raw.WF.mk.{u, v}`

Fields

`out : t.inner.WF`

Internal implementation detail of the tree map.

19.17.9. Dependent Tree-Based Maps

The declarations in this section should be imported using `import Std.DTreeMap`.

structure

```
Std.DTreeMap.{u, v} (α : Type u) (β : α → Type v)
  (cmp : α → α → Ordering := by exact compare) : Type (max u v)
```

Dependent tree maps.

A tree map stores an assignment of keys to values. It depends on a comparator function that defines an ordering on the keys and provides efficient order-dependent queries, such as retrieval of the minimum or maximum.

To ensure that the operations behave as expected, the comparator function *cmp* should satisfy certain laws that ensure a consistent ordering:

- If a is less than (or equal) to b, then b is greater than (or equal) to a and vice versa (see the `OrientedCmp` typeclass).
- If a is less than or equal to b and b is, in turn, less than or equal to c, then a is less than or equal to c (see the `TransCmp` typeclass).

Keys for which *cmp a b* = `Ordering.eq` are considered the same, i.e., there can be only one entry with key either *a* or *b* in a tree map. Looking up either *a* or *b* always yields the same entry, if any is present. The `get` operations of the *dependent* tree map additionally require a

`LawfulEqCmp` instance to ensure that *cmp a b = .eq* always implies *a = b*, so that their respective value types are equal.

To avoid expensive copies, users should make sure that the tree map is used linearly.

Internally, the tree maps are represented as size-bounded trees, a type of self-balancing binary search tree with efficient order statistic lookups.

For use in proofs, the type `Std.ExtDTreeMap` of extensional dependent tree maps should be preferred. This type comes with several extensionality lemmas and provides the same functions but requires a `TransCmp` instance to work with.

These tree maps contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, `Std.DTreeMap.Raw` and `Std.DTreeMap.Raw.WF` unbundle the invariant from the tree map. When in doubt, prefer `DTreeMap` over `DTreeMap.Raw`.

19.17.9.1. Creation

def

```
Std.DTreeMap.empty.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} : Std.DTreeMap α β cmp
```

Creates a new empty tree map. It is also possible and recommended to use the empty collection notations ∅ and `{}` to create an empty tree map. **`simp`** replaces `empty` with ∅.

19.17.9.2. Properties

def

```
Std.DTreeMap.size.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) : Nat
```

Returns the number of mappings present in the map.

def

```
Std.DTreeMap.isEmpty.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) : Bool
```

Returns `true` if the tree map contains no mappings.

19.17.9.3. Queries

def

```
Std.DTreeMap.contains.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α) :
Bool
```

Returns `true` if there is a mapping for the given key *a* or a key that is equal to *a* according to the comparator *cmp*. There is also a `Prop`-valued version of this: `a ∈ t` is equivalent to `t.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` uses `==` for equality checks, while for tree maps, both use the given comparator *cmp*.

def

```
Std.DTreeMap.get.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
  (t : Std.DTreeMap α β cmp) (a : α) (h : a ∈ t) : β a
```

Given a proof that a mapping for the given key is present, retrieves the mapping for the given key.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.DTreeMap.get!.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
  (t : Std.DTreeMap α β cmp) (a : α) [Inhabited (β a)] : β a
```

Tries to retrieve the mapping for the given key, panicking if no such mapping is present.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.DTreeMap.get?.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
  (t : Std.DTreeMap α β cmp) (a : α) : Option (β a)
```

Tries to retrieve the mapping for the given key, returning `none` if no such mapping is present.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.DTreeMap.getD.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
  (t : Std.DTreeMap α β cmp) (a : α) (fallback : β a) : β a
```

Tries to retrieve the mapping for the given key, returning *fallback* if no such mapping is present.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.DTreeMap.getKey.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α)
  (h : a ∈ t) : α
```

Retrieves the key from the mapping that matches *a*. Ensures that such a mapping exists by requiring a proof of `a ∈ m`. The result is guaranteed to be pointer equal to the key in the map.

def

```
Std.DTreeMap.getKey!.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Inhabited α] (t : Std.DTreeMap α β cmp)
  (a : α) : α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.DTreeMap.getKey?.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α) :
Option α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise `none`. The result in the `some` case is guaranteed to be pointer equal to the key in the map.

def

```
Std.DTreeMap.getKeyD.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a fallback :
α) :
  α
```

Checks if a mapping for the given key exists and returns the key if it does, otherwise *fallback*. If a mapping exists the result is guaranteed to be pointer equal to the key in the map.

def

```
Std.DTreeMap.keys.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) : List α
```

Returns a list of all keys present in the tree map in ascending order.

def

```
Std.DTreeMap.keysArray.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) : Array α
```

Returns an array of all keys present in the tree map in ascending order.

def

```
Std.DTreeMap.values.{u, v} {α : Type u} {cmp : α → α → Ordering}
  {β : Type v} (t : Std.DTreeMap α (fun x => β) cmp) : List β
```

Returns a list of all values present in the tree map in ascending order.

def

```
Std.DTreeMap.valuesArray.{u, v} {α : Type u} {cmp : α → α →
Ordering}
  {β : Type v} (t : Std.DTreeMap α (fun x => β) cmp) : Array β
```

Returns an array of all values present in the tree map in ascending order.

19.17.9.4. Modification

def

```
Std.DTreeMap.alter.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
  (t : Std.DTreeMap α β cmp) (a : α) (f : Option (β a) → Option (β
a)) :
  Std.DTreeMap α β cmp
```

Modifies in place the value associated with a given key, allowing creating new values and deleting values via an `Option` valued replacement function.

This function ensures that the value is used linearly.

def

```
Std.DTreeMap.modify.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
```

```
  (t : Std.DTreeMap α β cmp) (a : α) (f : β a → β a) :
  Std.DTreeMap α β cmp
```

Modifies in place the value associated with a given key.

This function ensures that the value is used linearly.

def

```
Std.DTreeMap.containsThenInsert.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α)
  (b : β a) : Bool × Std.DTreeMap α β cmp
```

Checks whether a key is present in a map and unconditionally inserts a value for the key.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.DTreeMap.containsThenInsertIfNew.{u, v} {α : Type u}
  {β : α → Type v} {cmp : α → α → Ordering} (t : Std.DTreeMap α β
cmp)
  (a : α) (b : β a) : Bool × Std.DTreeMap α β cmp
```

Checks whether a key is present in a map and inserts a value for the key if it was not found. If the returned `Bool` is `true`, then the returned map is unaltered. If the `Bool` is `false`, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `contains` followed by `insertIfNew`.

def

```
Std.DTreeMap.erase.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α) :
  Std.DTreeMap α β cmp
```

Removes the mapping for the given key if it exists.

def

```
Std.DTreeMap.filter.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (f : (a : α) → β a → Bool)
  (t : Std.DTreeMap α β cmp) : Std.DTreeMap α β cmp
```

Removes all mappings of the map for which the given function returns `false`.

def

```
Std.DTreeMap.filterMap.{u, v, w} {α : Type u} {β : α → Type v}
  {γ : α → Type w} {cmp : α → α → Ordering}
  (f : (a : α) → β a → Option (γ a)) (t : Std.DTreeMap α β cmp) :
  Std.DTreeMap α γ cmp
```

Updates the values of the map by applying the given function to all mappings, keeping only those mappings where the function returns some value.

def

```
Std.DTreeMap.insert.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α)
  (b : β a) : Std.DTreeMap α β cmp
```

Inserts the given mapping into the map. If there is already a mapping for the given key, then both key and value will be replaced.

def

```
Std.DTreeMap.insertIfNew.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) (a : α)
  (b : β a) : Std.DTreeMap α β cmp
```

If there is no mapping for the given key, inserts the given mapping into the map. Otherwise, returns the map unaltered.

def

```
Std.DTreeMap.getThenInsertIfNew?.{u, v} {α : Type u} {β : α → Type
v}
  {cmp : α → α → Ordering} [Std.LawfulEqCmp cmp]
  (t : Std.DTreeMap α β cmp) (a : α) (b : β a) :
  Option (β a) × Std.DTreeMap α β cmp
```

Checks whether a key is present in a map, returning the associated value, and inserts a value for the key if it was not found.

If the returned value is some v, then the returned map is unaltered. If it is none, then the returned map has a new value inserted.

Equivalent to (but potentially faster than) calling `get?` followed by `insertIfNew`.

Uses the `LawfulEqCmp` instance to cast the retrieved value to the correct type.

def

```
Std.DTreeMap.insertMany.{u, v, u_1} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} {ρ : Type u_1} [ForIn Id ρ ((a : α) × β
a)]
  (t : Std.DTreeMap α β cmp) (l : ρ) : Std.DTreeMap α β cmp
```

Inserts multiple mappings into the tree map by iterating over the given collection and calling `insert`. If the same key appears multiple times, the last occurrence takes precedence.

Note: this precedence behavior is true for `TreeMap`, `DTreeMap`, `TreeMap.Raw` and `DTreeMap.Raw`. The `insertMany` function on `TreeSet` and `TreeSet.Raw` behaves differently: it will prefer the first appearance.

def

```
Std.DTreeMap.partition.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (f : (a : α) → β a → Bool)
  (t : Std.DTreeMap α β cmp) :
  Std.DTreeMap α β cmp × Std.DTreeMap α β cmp
```

Partitions a tree map into two tree maps based on a predicate.

19.17.9.5. Iteration

def

```
Std.DTreeMap.map.{u, v, w} {α : Type u} {β : α → Type v}
  {γ : α → Type w} {cmp : α → α → Ordering} (f : (a : α) → β a → γ
a)
  (t : Std.DTreeMap α β cmp) : Std.DTreeMap α γ cmp
```

Updates the values of the map by applying the given function to all mappings.

def

```
Std.DTreeMap.foldl.{u, v, w} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} {δ : Type w} (f : δ → (a : α) → β a → δ)
  (init : δ) (t : Std.DTreeMap α β cmp) : δ
```

Folds the given function over the mappings in the map in ascending order.

def

```
Std.DTreeMap.foldlM.{u, v, w, w₂} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} {δ : Type w} {m : Type w → Type w₂}
[Monad m]
  (f : δ → (a : α) → β a → m δ) (init : δ) (t : Std.DTreeMap α β
```

```
cmp) :
  m δ
```

Folds the given monadic function over the mappings in the map in ascending order.

def

```
Std.DTreeMap.forIn.{u, v, w, w₂} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} {δ : Type w} {m : Type w → Type w₂}
[Monad m]
  (f : (a : α) → β a → δ → m (ForInStep δ)) (init : δ)
  (t : Std.DTreeMap α β cmp) : m δ
```

Support for the **`for`** loop construct in **`do`** blocks. Iteration happens in ascending order.

def

```
Std.DTreeMap.forM.{u, v, w, w₂} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} {m : Type w → Type w₂} [Monad m]
  (f : (a : α) → β a → m PUnit) (t : Std.DTreeMap α β cmp) : m
PUnit
```

Carries out a monadic action on each mapping in the tree map in ascending order.

19.17.9.6. Conversion

def

```
Std.DTreeMap.ofList.{u, v} {α : Type u} {β : α → Type v}
  (l : List ((a : α) × β a))
  (cmp : α → α → Ordering := by exact compare) : Std.DTreeMap α β
cmp
```

Transforms a list of mappings into a tree map.

def

```
Std.DTreeMap.toArray.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) :
  Array ((a : α) × β a)
```

Transforms the tree map into a list of mappings in ascending order.

def

```
Std.DTreeMap.toList.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap α β cmp) :
```

```
List ((α : α) × β α)
```

Transforms the tree map into a list of mappings in ascending order.

19.17.9.7. Unbundled Variants

Unbundled maps separate well-formedness proofs from data. This is primarily useful when defining nested inductive types. To use these variants, import the module `Std.DTreeMap.Raw`.

structure

```
Std.DTreeMap.Raw.{u, v} (α : Type u) (β : α → Type v)
  (_cmp : α → α → Ordering := by exact compare) : Type (max u v)
```

Dependent tree maps without a bundled well-formedness invariant, suitable for use in nested inductive types. The well-formedness invariant is called `Raw.WF`. When in doubt, prefer `DTreeMap` over `DTreeMap.Raw`. Lemmas about the operations on `Std.DTreeMap.Raw` are available in the module `Std.Data.DTreeMap.Raw.Lemmas`.

A tree map stores an assignment of keys to values. It depends on a comparator function that defines an ordering on the keys and provides efficient order-dependent queries, such as retrieval of the minimum or maximum.

To ensure that the operations behave as expected, the comparator function `cmp` should satisfy certain laws that ensure a consistent ordering:

- If a is less than (or equal) to b, then b is greater than (or equal) to a and vice versa (see the `OrientedCmp` typeclass).
- If a is less than or equal to b and b is, in turn, less than or equal to c, then a is less than or equal to c (see the `TransCmp` typeclass).

Keys for which `cmp a b = Ordering.eq` are considered the same, i.e., there can be only one entry with key either *a* or *b* in a tree map. Looking up either *a* or *b* always yields the same entry, if any is present. The `get` operations of the *dependent* tree map additionally require a `LawfulEqCmp` instance to ensure that `cmp a b = .eq` always implies $a = b$, so that their respective value types are equal.

To avoid expensive copies, users should make sure that the tree map is used linearly.

Internally, the tree maps are represented as size-bounded trees, a type of self-balancing binary search tree with efficient order statistic lookups.

Constructor

`Std.DTreeMap.Raw.mk.{u, v}`

Fields

`inner : Std.DTreeMap.Internal.Impl α β`

Internal implementation detail of the tree map.

structure

```
Std.DTreeMap.Raw.WF.{u, v} {α : Type u} {β : α → Type v}
  {cmp : α → α → Ordering} (t : Std.DTreeMap.Raw α β cmp) : Prop
```

Well-formedness predicate for tree maps. Users of `DTreeMap` will not need to interact with this. Users of `DTreeMap.Raw` will need to provide proofs of `WF` to lemmas and should use lemmas like `WF.empty` and `WF.insert` (which are always named exactly like the operations they are about) to show that map operations preserve well-formedness. The constructors of this type are internal implementation details and should not be accessed by users.

Constructor

`Std.DTreeMap.Raw.WF.mk.{u, v}`

Fields

`out : t.inner.WF`

Internal implementation detail of the tree map.

19.17.10. Tree-Based Sets

structure

```
Std.TreeSet.{u} (α : Type u)
  (cmp : α → α → Ordering := by exact compare) : Type u
```

Tree sets.

A tree set stores elements of a certain type in a certain order. It depends on a comparator function that defines an ordering on the keys and provides efficient order-dependent queries, such as retrieval of the minimum or maximum.

To ensure that the operations behave as expected, the comparator function *cmp* should satisfy certain laws that ensure a consistent ordering:

- If a is less than (or equal) to b, then b is greater than (or equal) to a and vice versa (see the `OrientedCmp` typeclass).
- If a is less than or equal to b and b is, in turn, less than or equal to c, then a is less than or equal to c (see the `TransCmp` typeclass).

Keys for which *cmp a b* = Ordering.eq are considered the same, i.e., there can be only one of them be contained in a single tree set at the same time.

To avoid expensive copies, users should make sure that the tree set is used linearly.

Internally, the tree sets are represented as size-bounded trees, a type of self-balancing binary search tree with efficient order statistic lookups.

For use in proofs, the type `Std.ExtTreeSet` of extensional tree sets should be preferred. This type comes with several extensionality lemmas and provides the same functions but requires a `TransCmp` instance to work with.

These tree sets contain a bundled well-formedness invariant, which means that they cannot be used in nested inductive types. For these use cases, Std.TreeSet.Raw and Std.TreeSet.Raw.WF unbundle the invariant from the tree set. When in doubt, prefer `TreeSet` over `TreeSet.Raw`.

19.17.10.1. Creation

def

```
Std.TreeSet.empty.{u} {α : Type u} {cmp : α → α → Ordering} :
  Std.TreeSet α cmp
```

Creates a new empty tree set. It is also possible and recommended to use the empty collection notations ∅ and {} to create an empty tree set. **simp** replaces `empty` with ∅.

19.17.10.2. Properties

def

```
Std.TreeSet.isEmpty.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) : Bool
```

Returns true if the tree set contains no mappings.

def

```
Std.TreeSet.size.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) : Nat
```

Returns the number of mappings present in the map.

19.17.10.3. Queries

def

```
Std.TreeSet.contains.{u} {α : Type u} {cmp : α → α → Ordering}
  (l : Std.TreeSet α cmp) (a : α) : Bool
```

Returns `true` if *a*, or an element equal to *a* according to the comparator *cmp*, is contained in the set. There is also a `Prop`-valued version of this: `a ∈ t` is equivalent to `t.contains a = true`.

Observe that this is different behavior than for lists: for lists, `∈` uses `=` and `contains` uses `==` for equality checks, while for tree sets, both use the given comparator *cmp*.

def

```
Std.TreeSet.get.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (a : α) (h : a ∈ t) : α
```

Retrieves the key from the set that matches *a*. Ensures that such a key exists by requiring a proof of `a ∈ m`. The result is guaranteed to be pointer equal to the key in the set.

def

```
Std.TreeSet.get!.{u} {α : Type u} {cmp : α → α → Ordering}
[Inhabited α]
  (t : Std.TreeSet α cmp) (a : α) : α
```

Checks if given key is contained and returns the key if it is, otherwise panics. If no panic occurs the result is guaranteed to be pointer equal to the key in the set.

def

```
Std.TreeSet.get?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (a : α) : Option α
```

Checks if given key is contained and returns the key if it is, otherwise `none`. The result in the `some` case is guaranteed to be pointer equal to the key in the map.

def

```
Std.TreeSet.getD.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (a fallback : α) : α
```

Checks if given key is contained and returns the key if it is, otherwise *fallback*. If they key is contained the result is guaranteed to be pointer equal to the key in the set.

19.17.10.3.1. Ordering-Based Queries

def

```
Std.TreeSet.atIdx.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (n : Nat) (h : n ≤ t.size) : α
```

Returns the *n*-th smallest element.

def

```
Std.TreeSet.atIdx!.{u} {α : Type u} {cmp : α → α → Ordering}
  [Inhabited α] (t : Std.TreeSet α cmp) (n : Nat) : α
```

Returns the *n*-th smallest element, or panics if *n* is at least `t.size`.

def

```
Std.TreeSet.atIdx?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (n : Nat) : Option α
```

Returns the *n*-th smallest element, or `none` if *n* is at least `t.size`.

def

```
Std.TreeSet.atIdxD.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (n : Nat) (fallback : α) : α
```

Returns the *n*-th smallest element, or `fallback` if *n* is at least `t.size`.

def

```
Std.TreeSet.getGE.{u} {α : Type u} {cmp : α → α → Ordering}
  [Std.TransCmp cmp] (t : Std.TreeSet α cmp) (k : α)
  (h : ∃ a, a ∈ t ∧ (cmp a k).isGE = true) : α
```

Given a proof that such an element exists, retrieves the smallest element that is greater than or equal to the given element.

def

```
Std.TreeSet.getGE!.{u} {α : Type u} {cmp : α → α → Ordering}
  [Inhabited α] (t : Std.TreeSet α cmp) (k : α) : α
```

Tries to retrieve the smallest element that is greater than or equal to the given element, panicking if no such element exists.

def

```
Std.TreeSet.getGE?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k : α) : Option α
```

Tries to retrieve the smallest element that is greater than or equal to the given element, returning none if no such element exists.

def

```
Std.TreeSet.getGED.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k fallback : α) : α
```

Tries to retrieve the smallest element that is greater than or equal to the given element, returning *fallback* if no such element exists.

def

```
Std.TreeSet.getGT.{u} {α : Type u} {cmp : α → α → Ordering}
  [Std.TransCmp cmp] (t : Std.TreeSet α cmp) (k : α)
  (h : ∃ a, a ∈ t ∧ cmp a k = Ordering.gt) : α
```

Given a proof that such an element exists, retrieves the smallest element that is greater than the given element.

def

```
Std.TreeSet.getGT!.{u} {α : Type u} {cmp : α → α → Ordering}
  [Inhabited α] (t : Std.TreeSet α cmp) (k : α) : α
```

Tries to retrieve the smallest element that is greater than the given element, panicking if no such element exists.

def

```
Std.TreeSet.getGT?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k : α) : Option α
```

Tries to retrieve the smallest element that is greater than the given element, returning none if no such element exists.

def

```
Std.TreeSet.getGTD.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k fallback : α) : α
```

Tries to retrieve the smallest element that is greater than the given element, returning *fallback* if no such element exists.

def

```
Std.TreeSet.getLE.{u} {α : Type u} {cmp : α → α → Ordering}
  [Std.TransCmp cmp] (t : Std.TreeSet α cmp) (k : α)
  (h : ∃ a, a ∈ t ∧ (cmp a k).isLE = true) : α
```

Given a proof that such an element exists, retrieves the largest element that is less than or equal to the given element.

def

```
Std.TreeSet.getLE!.{u} {α : Type u} {cmp : α → α → Ordering}
  [Inhabited α] (t : Std.TreeSet α cmp) (k : α) : α
```

Tries to retrieve the largest element that is less than or equal to the given element, panicking if no such element exists.

def

```
Std.TreeSet.getLE?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k : α) : Option α
```

Tries to retrieve the largest element that is less than or equal to the given element, returning none if no such element exists.

def

```
Std.TreeSet.getLED.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k fallback : α) : α
```

Tries to retrieve the largest element that is less than or equal to the given element, returning *fallback* if no such element exists.

def

```
Std.TreeSet.getLT.{u} {α : Type u} {cmp : α → α → Ordering}
  [Std.TransCmp cmp] (t : Std.TreeSet α cmp) (k : α)
  (h : ∃ a, a ∈ t ∧ cmp a k = Ordering.lt) : α
```

Given a proof that such an element exists, retrieves the smallest element that is less than the given element.

def

```
Std.TreeSet.getLT!.{u} {α : Type u} {cmp : α → α → Ordering}
  [Inhabited α] (t : Std.TreeSet α cmp) (k : α) : α
```

Tries to retrieve the smallest element that is less than the given element, panicking if no such element exists.

def

```
Std.TreeSet.getLT?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k : α) : Option α
```

Tries to retrieve the smallest element that is less than the given element, returning none if no such element exists.

def

```
Std.TreeSet.getLTD.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (k fallback : α) : α
```

Tries to retrieve the smallest element that is less than the given element, returning *fallback* if no such element exists.

def

```
Std.TreeSet.min.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (h : t.isEmpty = false) : α
```

Given a proof that the tree set is not empty, retrieves the smallest element.

def

```
Std.TreeSet.min!.{u} {α : Type u} {cmp : α → α → Ordering}
[Inhabited α]
  (t : Std.TreeSet α cmp) : α
```

Tries to retrieve the smallest element of the tree set, panicking if the set is empty.

def

```
Std.TreeSet.min?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) : Option α
```

Tries to retrieve the smallest element of the tree set, returning none if the set is empty.

def

```
Std.TreeSet.minD.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (fallback : α) : α
```

Tries to retrieve the smallest element of the tree set, returning *fallback* if the tree set is empty.

def

```
Std.TreeSet.max.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (h : t.isEmpty = false) : α
```

Given a proof that the tree set is not empty, retrieves the largest element.

def

```
Std.TreeSet.max!.{u} {α : Type u} {cmp : α → α → Ordering}
[Inhabited α]
  (t : Std.TreeSet α cmp) : α
```

Tries to retrieve the largest element of the tree set, panicking if the set is empty.

def

```
Std.TreeSet.max?.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) : Option α
```

Tries to retrieve the largest element of the tree set, returning none if the set is empty.

def

```
Std.TreeSet.maxD.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (fallback : α) : α
```

Tries to retrieve the largest element of the tree set, returning *fallback* if the tree set is empty.

19.17.10.4. Modification

def

```
Std.TreeSet.insert.{u} {α : Type u} {cmp : α → α → Ordering}
  (l : Std.TreeSet α cmp) (a : α) : Std.TreeSet α cmp
```

Inserts the given element into the set. If the tree set already contains an element that is equal (with regard to *cmp*) to the given element, then the tree set is returned unchanged.

Note: this non-replacement behavior is true for `TreeSet` and `TreeSet.Raw`. The `insert` function on `TreeMap`, `DTreeMap`, `TreeMap.Raw` and `DTreeMap.Raw` behaves differently: it will overwrite an existing mapping.

def

```
Std.TreeSet.insertMany.{u, u_1} {α : Type u} {cmp : α → α →
Ordering}
  {ρ : Type u_1} [ForIn Id ρ α] (t : Std.TreeSet α cmp) (l : ρ) :
  Std.TreeSet α cmp
```

Inserts multiple elements into the tree set by iterating over the given collection and calling `insert`. If the same element (with respect to *cmp*) appears multiple times, the first occurrence takes precedence.

Note: this precedence behavior is true for `TreeSet` and `TreeSet.Raw`. The `insertMany` function on `TreeMap`, `DTreeMap`, `TreeMap.Raw` and `DTreeMap.Raw` behaves differently: it will prefer the last appearance.

def

```
Std.TreeSet.containsThenInsert.{u} {α : Type u} {cmp : α → α →
Ordering}
  (t : Std.TreeSet α cmp) (a : α) : Bool × Std.TreeSet α cmp
```

Checks whether an element is present in a set and inserts the element if it was not found. If the tree set already contains an element that is equal (with regard to *cmp* to the given element, then the tree set is returned unchanged.

Equivalent to (but potentially faster than) calling `contains` followed by `insert`.

def

```
Std.TreeSet.erase.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (a : α) : Std.TreeSet α cmp
```

Removes the given key if it exists.

def

```
Std.TreeSet.eraseMany.{u, u_1} {α : Type u} {cmp : α → α →
Ordering}
  {ρ : Type u_1} [ForIn Id ρ α] (t : Std.TreeSet α cmp) (l : ρ) :
  Std.TreeSet α cmp
```

Erases multiple items from the tree set by iterating over the given collection and calling erase.

def

```
Std.TreeSet.filter.{u} {α : Type u} {cmp : α → α → Ordering}
  (f : α → Bool) (m : Std.TreeSet α cmp) : Std.TreeSet α cmp
```

Removes all elements from the tree set for which the given function returns `false`.

def

```
Std.TreeSet.merge.{u} {α : Type u} {cmp : α → α → Ordering}
  (t₁ t₂ : Std.TreeSet α cmp) : Std.TreeSet α cmp
```

Returns a set that contains all mappings of t_1 and `t₂.

This function ensures that t_1 is used linearly. Hence, as long as t_1 is unshared, the performance characteristics follow the following imperative description: Iterate over all mappings in t_2, inserting them into t_1.

Hence, the runtime of this method scales logarithmically in the size of t_1 and linearly in the size of t_2 as long as t_1 is unshared.

def

```
Std.TreeSet.partition.{u} {α : Type u} {cmp : α → α → Ordering}
  (f : α → Bool) (t : Std.TreeSet α cmp) :
  Std.TreeSet α cmp × Std.TreeSet α cmp
```

Partitions a tree set into two tree sets based on a predicate.

19.17.10.5. Iteration

def

```
Std.TreeSet.all.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (p : α → Bool) : Bool
```

Check if any element satisfies the predicate, short-circuiting if a predicate succeeds.

def

```
Std.TreeSet.any.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) (p : α → Bool) : Bool
```

Check if all elements satisfy the predicate, short-circuiting if a predicate fails.

def

```
Std.TreeSet.foldl.{u, w} {α : Type u} {cmp : α → α → Ordering}
  {δ : Type w} (f : δ → α → δ) (init : δ) (t : Std.TreeSet α cmp) :
```

```
δ
```

Folds the given function over the elements of the tree set in ascending order.

def

```
Std.TreeSet.foldlM.{u, u_1, u_2} {α : Type u} {cmp : α → α →
Ordering}
  {m : Type u_1 → Type u_2} {δ : Type u_1} [Monad m] (f : δ → α → m
δ)
  (init : δ) (t : Std.TreeSet α cmp) : m δ
```

Monadically computes a value by folding the given function over the elements in the tree set in ascending order.

def

```
Std.TreeSet.foldr.{u, w} {α : Type u} {cmp : α → α → Ordering}
  {δ : Type w} (f : α → δ → δ) (init : δ) (t : Std.TreeSet α cmp) :
δ
```

Folds the given function over the elements of the tree set in descending order.

def

```
Std.TreeSet.foldrM.{u, u_1, u_2} {α : Type u} {cmp : α → α →
Ordering}
  {m : Type u_1 → Type u_2} {δ : Type u_1} [Monad m] (f : α → δ → m
δ)
  (init : δ) (t : Std.TreeSet α cmp) : m δ
```

Monadically computes a value by folding the given function over the elements in the tree set in descending order.

def

```
Std.TreeSet.forIn.{u, w, w₂} {α : Type u} {cmp : α → α → Ordering}
  {δ : Type w} {m : Type w → Type w₂} [Monad m]
  (f : α → δ → m (ForInStep δ)) (init : δ) (t : Std.TreeSet α cmp)
: m δ
```

Support for the **`for`** loop construct in **`do`** blocks. The iteration happens in ascending order.

def

```
Std.TreeSet.forM.{u, w, w₂} {α : Type u} {cmp : α → α → Ordering}
  {m : Type w → Type w₂} [Monad m] (f : α → m PUnit)
```

```
  (t : Std.TreeSet α cmp) : m PUnit
```

Carries out a monadic action on each element in the tree set in ascending order.

19.17.10.6. Conversion

def

```
Std.TreeSet.toList.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) : List α
```

Transforms the tree set into a list of elements in ascending order.

def

```
Std.TreeSet.ofList.{u} {α : Type u} (l : List α)
  (cmp : α → α → Ordering := by exact compare) : Std.TreeSet α cmp
```

Transforms a list into a tree set.

def

```
Std.TreeSet.toArray.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet α cmp) : Array α
```

Transforms the tree set into an array of elements in ascending order.

def

```
Std.TreeSet.ofArray.{u} {α : Type u} (a : Array α)
  (cmp : α → α → Ordering := by exact compare) : Std.TreeSet α cmp
```

Transforms an array into a tree set.

19.17.10.6.1. Unbundled Variants

Unbundled sets separate well-formedness proofs from data. This is primarily useful when defining nested inductive types. To use these variants, import the module `Std.TreeSet.Raw`.

structure

```
Std.TreeSet.Raw.{u} (α : Type u)
  (cmp : α → α → Ordering := by exact compare) : Type u
```

Tree sets without a bundled well-formedness invariant, suitable for use in nested inductive types. The well-formedness invariant is called `Raw.WF`. When in doubt, prefer `TreeSet` over `TreeSet.Raw`. Lemmas about the operations on `Std.TreeSet.Raw` are available in the

module `Std.Data.TreeSet.Raw.Lemmas`.

A tree set stores elements of a certain type in a certain order. It depends on a comparator function that defines an ordering on the keys and provides efficient order-dependent queries, such as retrieval of the minimum or maximum.

To ensure that the operations behave as expected, the comparator function *cmp* should satisfy certain laws that ensure a consistent ordering:

- If a is less than (or equal) to b, then b is greater than (or equal) to a and vice versa (see the `OrientedCmp` typeclass).
- If a is less than or equal to b and b is, in turn, less than or equal to c, then a is less than or equal to c (see the `TransCmp` typeclass).

Keys for which *cmp a b* = `Ordering.eq` are considered the same, i.e only one of them can be contained in a single tree set at the same time.

To avoid expensive copies, users should make sure that the tree set is used linearly.

Internally, the tree sets are represented as size-bounded trees, a type of self-balancing binary search tree with efficient order statistic lookups.

Constructor

`Std.TreeSet.Raw.mk.{u}`

Fields

`inner : Std.TreeMap.Raw α Unit cmp`

Internal implementation detail of the tree set.

structure

```
Std.TreeSet.Raw.WF.{u} {α : Type u} {cmp : α → α → Ordering}
  (t : Std.TreeSet.Raw α cmp) : Prop
```

Well-formedness predicate for tree sets. Users of `TreeSet` will not need to interact with this. Users of `TreeSet.Raw` will need to provide proofs of `WF` to lemmas and should use lemmas like `WF.empty` and `WF.insert` (which are always named exactly like the operations they are about) to show that set operations preserve well-formedness. The constructors of this type are internal implementation details and should not be accessed by users.

Constructor

`Std.TreeSet.Raw.WF.mk.{u}`

Fields

`out : t.inner.WF`

Internal implementation detail of the tree map.

19.18. Subtypes

The structure `Subtype` represents the elements of a type that satisfy some predicate. They are used pervasively both in mathematics and in programming; in mathematics, they are used similarly to subsets, while in programming, they allow information that is known about a value to be represented in a way that is visible to Lean's logic.

Syntactically, an element of a `Subtype` resembles a tuple of the base type's element and the proof that it satisfies the proposition. They differ from dependent pair types (`Sigma`) in that the second element is a proof of a proposition rather than data, and from existential quantification in that the entire `Subtype` is a type rather than a proposition. Even though they are pairs syntactically, `Subtype` should really be thought of as elements of the base type with associated proof obligations.

Subtypes are trivial wrappers. They are thus represented identically to the base type in compiled code.

structure

```
Subtype.{u} {α : Sort u} (p : α → Prop) : Sort (max 1 u)
```

All the elements of a type that satisfy a predicate.

`Subtype p`, usually written `{ x : α // p x }` or `{ x // p x }`, contains all elements `x : α` for which `p x` is true. Its constructor is a pair of the value and the proof that it satisfies the predicate. In run-time code, `{ x : α // p x }` is represented identically to `α`.

There is a coercion from `{ x : α // p x }` to `α`, so elements of a subtype may be used where the underlying type is expected.

Examples:

- `{ n : Nat // n % 2 = 0 }` is the type of even numbers.
- `{ xs : Array String // xs.size = 5 }` is the type of arrays with five `String`s.
- Given `xs : List α`, `List { x : α // x ∈ xs }` is the type of lists in which all elements are contained in `xs`.

Conventions for notations in identifiers:

- The recommended spelling of `{ x // p x }` in identifiers is `subtype`.

Constructor

`Subtype.mk.{u}`

Fields

`val : α`

The value in the underlying type that satisfies the predicate.

`property : p self.val`

The proof that `val` satisfies the predicate `p`.

syntax

Subtypes

```
term ::= ...
    | { ident : term // term }
```

`{ x : α // p }` is a notation for `Subtype fun (x : α) => p`.

The type ascription may be omitted:

```
term ::= ...
    | { ident // term }
```

`{ x // p }` is a notation for `Subtype fun (x : _) => p`.

Due to proof irrelevance and η-equality, two elements of a subtype are definitionally equal when the elements of the base type are definitionally equal. In a proof, the **`ext`** tactic can be used to transform a goal of equality of elements of a subtype into equality of their values.

▼ *Example: Definitional Equality of Subtypes*

The non-empty strings `s1` and `s2` are definitionally equal despite the fact that their embedded proof terms are different. No case splitting is needed in order to prove that they are equal.

```
def NonEmptyString := { x : String // x ≠ "" }

def s1 : NonEmptyString :=
  ⟨"equal", ne_of_beq_false rfl⟩

def s2 : NonEmptyString where
  val := "equal"
  property :=
    fun h =>
```

```
      List.cons_ne_nil _ _ (String.data_eq_of_eq h)

theorem s1_eq_s2 : s1 = s2 := by rfl
```

▼ *Example: Extensional Equality of Subtypes*

The non-empty strings `s1` and `s2` are definitionally equal. Ignoring that fact, the equality of the embedded strings can be used to prove that they are equal. The **`ext`** tactic transforms a goal that consists of equality of non-empty strings into a goal that consists of equality of the strings.

```
abbrev NonEmptyString := { x : String // x ≠ "" }

def s1 : NonEmptyString :=
  ⟨"equal", ne_of_beq_false rfl⟩

def s2 : NonEmptyString where
  val := "equal"
  property :=
    fun h =>
      List.cons_ne_nil _ _ (String.data_eq_of_eq h)

theorem s1_eq_s2 : s1 = s2 := by
  ext
  dsimp only [s1, s2]
```

There is a coercion from a subtype to its base type. This allows subtypes to be used in positions where the base type is expected, essentially erasing the proof that the value satisfies the predicate.

▼ *Example: Subtype Coercions*

Elements of subtypes can be coerced to their base type. Here, `nine` is coerced from a subtype of `Nat` that contains multiples of 3 to `Nat`.

```
abbrev DivBy3 := { x : Nat // x % 3 = 0 }

def nine : DivBy3 := ⟨9, by rfl⟩

set_option eval.type true in
#eval Nat.succ nine
```

```
10 : Nat
```

19.19. Lazy Computations

A *thunk* delays the computation of a value. In particular, the `Thunk` type is used to delay the computation of a value in compiled code until it is explicitly requested—this request is called *forcing* the thunk. The computed value is saved, so subsequent requests do not result in recomputation. Computing values at most once, when explicitly requested, is called *lazy evaluation*. This caching is invisible to Lean's logic, in which `Thunk` is equivalent to a function from `Unit`.

19.19.1. Logical Model

Thunks are modeled as a single-field structure that contains a function from `Unit`. The structure's field is private, so the function itself cannot be directly accessed. Instead, `Thunk.get` should be used. From the perspective of the logic, they are equivalent; `Thunk.get` exists to be overridden in the compiler by the platform primitive that implements lazy evaluation.

structure

```
Thunk.{u} (α : Type u) : Type u
```

Delays evaluation. The delayed code is evaluated at most once.

A thunk is code that constructs a value when it is requested via `Thunk.get`, `Thunk.map`, or `Thunk.bind`. The resulting value is cached, so the code is executed at most once. This is also known as lazy or call-by-need evaluation.

The Lean runtime has special support for the `Thunk` type in order to implement the caching behavior.

Constructor

`Thunk.mk.{u}`

Constructs a new thunk from a function `Unit → α` that will be called when the thunk is first forced.

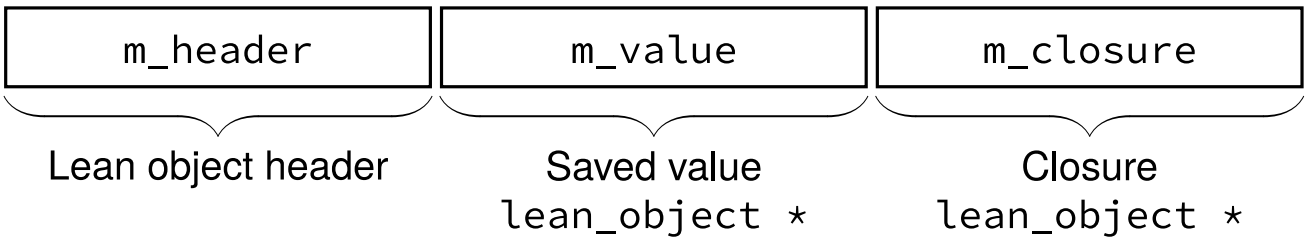
The result is cached. It is re-used when the thunk is forced again.

Fields

`fn : Unit → α`

Extract the getter function out of a thunk. Use `Thunk.get` instead.

19.19.2. Runtime Representation



Memory layout of thunks

Thunks are one of the primitive object types supported by the Lean runtime. The object header contains a specific tag that indicates that an object is a thunk.

Thunks have two fields:

- `m_value` is a pointer to a saved value, which is a null pointer if the value has not yet been computed.
- `m_closure` is a closure which is to be called when the value should be computed.

The runtime system maintains the invariant that either the closure or the saved value is a null pointer. If both are null pointers, then the thunk is being forced on another thread.

When a thunk is forced, the runtime system first checks whether the saved value has already been computed, returning it if so. Otherwise, it attempts to acquire a lock on the closure by atomically swapping it with a null pointer. If the lock is acquired, it is invoked to compute the value; the computed value is stored in the saved value field and the reference to the closure is dropped. If not, then another thread is already computing the value; the system waits until it is computed.

19.19.3. Coercions

There is a coercion from any type α to `Thunk α` that converts a term e into `Thunk.mk fun () => e`. Because the elaborator unfolds coercions, evaluation of the original term e is delayed; the coercion is not equivalent to `Thunk.pure`.

▼ *Example: Lazy Lists*

Lazy lists are lists that may contain thunks. The `delayed` constructor causes part of the list to be computed on demand.

```
inductive LazyList (α : Type u) where
  | nil
  | cons : α → LazyList α → LazyList α
  | delayed : Thunk (LazyList α) → LazyList α
deriving Inhabited
```

Lazy lists can be converted to ordinary lists by forcing all the embedded thunks.

```
def LazyList.toList : LazyList α → List α
  | .nil => []
  | .cons x xs => x :: xs.toList
  | .delayed xs => xs.get.toList
```

Many operations on lazy lists can be implemented without forcing the embedded thunks, instead building up further thunks. The body of `delayed` does not need to be an explicit call to `Thunk.mk` because of the coercion.

```
def LazyList.take : Nat → LazyList α → LazyList α
  | 0, _ => .nil
  | _, .nil => .nil
  | n + 1, .cons x xs => .cons x <| .delayed <| take n xs
  | n + 1, .delayed xs => .delayed <| take (n + 1) xs.get

def LazyList.ofFn (f : Fin n → α) : LazyList α :=
  Fin.foldr n (init := .nil) fun i xs =>
    .delayed <| LazyList.cons (f i) xs

def LazyList.append (xs ys : LazyList α) : LazyList α :=
  .delayed <|
    match xs with
    | .nil => ys
    | .cons x xs' => LazyList.cons x (append xs' ys)
    | .delayed xs' => append xs'.get ys
```

Laziness is ordinarily invisible to Lean programs: there is no way to check whether a thunk has been forced. However, `dbg_trace` can be used to gain insight into thunk evaluation.

```
def observe (tag : String) (i : Fin n) : Nat :=
  dbg_trace "{tag}: {i.val}"
  i.val
```

The lazy lists `xs` and `ys` emit traces when evaluated.

```
def xs := LazyList.ofFn (n := 3) (observe "xs")
def ys := LazyList.ofFn (n := 3) (observe "ys")
```

Converting `xs` to an ordinary list forces all of the embedded thunks:

```
#eval xs.toList
```

```
xs: 0
xs: 1
xs: 2
```

```
[0, 1, 2]
```

Likewise, converting `xs.append ys` to an ordinary list forces the embedded thunks:

```
#eval xs.append ys |>.toList
```

```
xs: 0
xs: 1
xs: 2
ys: 0
ys: 1
ys: 2
```

```
[0, 1, 2, 0, 1, 2]
```

Appending `xs` to itself before forcing the thunks results in a single set of traces, because each thunk's code is evaluated just once:

```
#eval xs.append xs |>.toList
```

```
xs: 0
xs: 1
xs: 2
```

```
[0, 1, 2, 0, 1, 2]
```

Finally, taking a prefix of `xs.append ys` results in only some of the thunks in `ys` being evaluated:

```
#eval xs.append ys |>.take 4 |>.toList
```

```
xs: 0
xs: 1
xs: 2
ys: 0
```

```
[0, 1, 2, 0]
```

19.19.4. API Reference

def

```
Thunk.get.{u_1} {α : Type u_1} (x : Thunk α) : α
```

Gets the thunk's value. If the value is cached, it is returned in constant time; if not, it is computed.

Computed values are cached, so the value is not recomputed.

def

```
Thunk.map.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α → β)
  (x : Thunk α) : Thunk β
```

Constructs a new thunk that forces x and then applies x to the result. Upon forcing, the result of f is cached and the reference to the thunk x is dropped.

def

```
Thunk.pure.{u_1} {α : Type u_1} (a : α) : Thunk α
```

Stores an already-computed value in a thunk.

Because the value has already been computed, there is no laziness.

def

```
Thunk.bind.{u_1, u_2} {α : Type u_1} {β : Type u_2} (x : Thunk α)
  (f : α → Thunk β) : Thunk β
```

Constructs a new thunk that applies f to the result of x when forced.

20. Notations and Macros

Different mathematical fields have their own notational conventions, and many notations are re-used with differing meanings in different fields. It is important that formal developments are able to use established notations: formalizing mathematics is already difficult, and the mental overhead of translating between syntaxes can be substantial. At the same time, it's important to be able to control the scope of notational extensions. Many fields use related notations with very different meanings, and it should be possible to combine developments from these separate fields in a way where both readers and the system know which convention is in force in any given region of a file.

Lean addresses the problem of notational extensibility with a variety of mechanisms, each of which solves a different aspect of the problem. They can be combined flexibly to achieve the necessary results:

- The *extensible parser* allows a great variety of notational conventions to be implemented declaratively, and combined flexibly.
- Macros allow new syntax to be easily mapped to existing syntax, which is a simple way to provide meaning to new constructs. Due to hygiene and automatic propagation of source positions, this process doesn't interfere with Lean's interactive features.
- Elaborators provide new syntax with the same tools available to Lean's own syntax in cases where a macro is insufficiently expressive.
- Notations allow the simultaneous definition of a parser extension, a macro, and a pretty printer. When defining infix, prefix, or postfix operators, custom operators automatically take care of precedence and associativity.
- Low-level parser extensions allow the parser to be extended in ways that modify its rules for tokens and whitespace, or that even completely replace Lean's syntax. This is an advanced topic that requires familiarity with Lean internals; nevertheless, the possibility of doing this

without modifying the compiler is important. This reference manual is written using a language extension that replaces Lean's concrete syntax with a Markdown-like language for writing documents, but the source files are still Lean files.

20.1. Custom Operators

Lean supports custom infix, prefix, and postfix operators. New operators can be added by any Lean library, and the new operators have equal status to those that are part of the language. Each new operator is assigned an interpretation as a function, after which uses of the operator are translated into uses of the function. The operator's translation into a function call is referred to as its *expansion*. If this function is a type class method, then the resulting operator can be overloaded by defining instances of the class.

All operators have a *precedence*. Operator precedence determines the order of operations for unparenthesized expressions: because multiplication has a higher precedence than addition, `2 + 3 * 4` is equivalent to `2 + (3 * 4)`, and `2 * 3 + 4` is equivalent to `(2 * 3) + 4`. Infix operators additionally have an *associativity* that determines the meaning of a chain of operators that have the same precedence:

Left-associative

These operators nest to the left. Addition is left- associative, so `2 + 3 + 4 + 5` is equivalent to `((2 + 3) + 4) + 5`.

Right-associative

These operators nest to the right. The product type is right-associative, so `Nat × String × Unit × Option Int` is equivalent to `Nat × (String × (Unit × Option Int))`.

Non-associative

Chaining these operators is a syntax error. Explicit parenthesization is required. Equality is non-associative, so the following is an error:

```
1 1 + 2 = 3 = 2 + 1
```

The parser error is:

```
<example>:1:10-1:11: expected end of input
```

▼ *Example: Precedence for Prefix and Infix Operators*

The proposition `¬A ∧ B` is equivalent to `(¬A) ∧ B`, because `¬` has a higher precedence than `∧`. Because `∧` has higher precedence than `=` and is right-associative, `¬A ∧ B = (¬A) ∧ B` is equivalent to `¬A ∧ ((B = ¬A) ∧ B)`.

Lean provides commands for defining new operators:

syntax

Operator Declarations

Non-associative infix operators are defined using `infix`:

```
command ::= ...
    | docComment?
      attributes?
      attrKind infix:prec ((name := ident))? ((priority := prio))?
str => term
```

Left-associative infix operators are defined using `infixl`:

```
command ::= ...
    | docComment?
      attributes?
      attrKind infixl:prec ((name := ident))? ((priority :=
prio))? str => term
```

Right-associative infix operators are defined using `infixr`:

```
command ::= ...
    | docComment?
      attributes?
      attrKind infixr:prec ((name := ident))? ((priority :=
prio))? str => term
```

Prefix operators are defined using `prefix`:

```
command ::= ...
    | docComment?
      attributes?
      attrKind prefix:prec ((name := ident))? ((priority :=
prio))? str => term
```

Postfix operators are defined using `postfix`:

```
command ::= ...
    | docComment?
      attributes?
      attrKind postfix:prec ((name := ident))? ((priority :=
prio))? str => term
```

Each of these commands may be preceded by documentation comments and attributes. The documentation comment is shown when the user hovers their mouse over the operator, and attributes may invoke arbitrary metaprograms, just as for any other declaration. The attribute `inherit_doc` causes the documentation of the function that implements the operator to be re-used for the operator itself.

Operators interact with section scopes in the same manner as attributes. By default, operators are available in any module that transitively imports the one in which they are established, but they may be declared `scoped` or `local` to restrict their availability either to contexts in which the current namespace has been opened or to the current section scope, respectively.

Custom operators require a precedence specifier, following a colon. There is no default precedence to fall back to for custom operators.

Operators may be explicitly named. This name denotes the extension to Lean's syntax, and is primarily used for metaprogramming. If no name is explicitly provided, then Lean generates one based on the operator. The specifics of the assignment of this name should not be relied upon, both because the internal name assignment algorithm may change and because the introduction of similar operators in upstream dependencies may lead to a clash, in which case Lean will modify the assigned name until it is unique.

▼ *Example: Assigned Operator Names*

Given this infix operator:

```
infix:90 " ⤴ " => Option.getD
```

the internal name «`term_⤴_`» is assigned to the resulting parser extension.

▼ *Example: Provided Operator Names*

Given this infix operator:

```
infix:90 (name := getDOp) " ⤴ " => Option.getD
```

the resulting parser extension is named `getDOp`.

▼ *Example: Inheriting Documentation*

Given this infix operator:

```
@[inherit_doc]
infix:90 " ⤴ " => Option.getD
```

the resulting parser extension has the same documentation as `Option.getD`.

When multiple operators are defined that share the same syntax, Lean's parser attempts all of them. If more than one succeed, the one that used the most input is selected—this is called the *local longest-match rule*. In some cases, parsing multiple operators may succeed, all of them covering the same range of the input. In these cases, the operator's priority is used to select the appropriate

result. Finally, if multiple operators with the same priority tie for the longest match, the parser saves all of the results, and the elaborator attempts each in turn, failing if elaboration does not succeed on exactly one of them.

▼ *Example: Ambiguous Operators and Priorities*

Defining an alternative implementation of + as Or requires only an infix operator declaration.

```
infix:65  " + " => Or
```

With this declaration, Lean attempts to elaborate addition both using the built-in syntax for HAdd.hAdd and the new syntax for Or:

```
#check True + False
```

```
True + False : Prop
```

```
#check 2 + 2
```

```
2 + 2 : Nat
```

However, because the new operator is not associative, the local longest-match rule means that only HAdd.hAdd applies to an unparenthesized three-argument version:

```
#check True + False + True
```

```
failed to synthesize
  HAdd Prop Prop ?m.140

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

If the infix operator is declared with high priority, then Lean does not try the built-in HAdd.hAdd operator in ambiguous cases:

```
infix:65 (priority := high)  " + " => Or
```

```
#check True + False
```

```
True + False : Prop
```

```
#check 2 + 2
```

```
failed to synthesize
  OfNat Prop 2
numerals are polymorphic in Lean, but the numeral `2` cannot
be used in a context where the expected type is
  Prop
due to the absence of the instance above

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

The new operator is not associative, so the local longest-match rule means that only HAdd.hAdd applies to the three-argument version:

```
#check True + False + True
```

```
failed to synthesize
  HAdd Prop Prop ?m.80

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

The actual operator is provided as a string literal. The new operator must satisfy the following requirements:

- It must contain at least one character.
- The first character may not be a single or double quote (' or "), unless the operator is '' .
- It may not begin with a backtick (`) followed by a character that would be a valid prefix of a quoted name.
- It may not begin with a digit.
- It may not include internal whitespace.

The operator string literal may begin or end with a space. These are not part of the operator's syntax, and their presence does not require spaces around uses of the operator. However, the presence of spaces cause Lean to insert spaces when showing the operator to the user. Omitting them causes the operator's arguments to be displayed immediately next to the operator itself.

Finally, the operator's meaning is provided, separated from the operator by =>. This may be any Lean term. Uses of the operator are desugared into function applications, with the provided term in the function position. Prefix and postfix operators apply the term to their single argument as an explicit argument. Infix operators apply the term to the left and right arguments, in that order. Other than its ability to accept arguments at each call site, there are no specific requirements imposed on the term. Operators may construct functions, so the term may expect more parameters than the operator. Implicit and instance-implicit parameters are resolved at each application site, which allows the operator to be defined by a type class method.

If the term consists either of a name from the global environment or of an application of such a name to one or more arguments, then Lean automatically generates an unexpander for the operator. This means that the operator will be displayed in proof states, error messages, and other output from Lean when the function term otherwise would have been displayed. Lean does not track whether the operator was used in the original term; it is inserted at every opportunity.

▼ *Example: Custom Operators in Lean's Output*

The function perhapsFactorial computes a factorial for a number if it's not too large.

```
def fact : Nat → Nat
  | 0 => 1
  | n+1 => (n + 1) * fact n

def perhapsFactorial (n : Nat) : Option Nat :=
  if n < 8 then some (fact n) else none
```

The postfix interrobang operator can be used to represent it.

```
postfix:90 "?" => perhapsFactorial
```

When attempting to prove that `∀ n, n ≥ 8 → (perhapsFactorial n).isNone`, the initial proof state uses the new operator, even though the theorem as written does not:

```
⊢ ∀ (n : Nat), n ≥ 8 → n?.isNone = true
```

20.2. Precedence

Infix operators, notations, and other syntactic extensions to Lean make use of explicit precedence annotations. While precedences in Lean can technically be any natural number, by convention they range from 10 to 1024, respectively denoted `min` and `max`. Function application has the highest precedence.

syntax

Parser Precedences

Most operator precedences consist of explicit numbers. The named precedence levels denote the outer edges of the range, close to the minimum or maximum, and are typically used by more involved syntax extensions.

```
prec ::=
    num
```

Precedences may also be denoted as sums or differences of precedences; these are typically used to assign precedences that are relative to one of the named precedences.

```
prec ::= ...
    | prec + prec

prec ::= ...
    | prec - prec

prec ::= ...
    | (prec)
```

The maximum precedence is used to parse terms that occur in a function position. Operators should typically not use use this level, because it can interfere with users' expectation that function application binds more tightly than any other operator, but it is useful in more involved syntax extensions to indicate how other constructs interact with function application.

```
prec ::= ...
    | max
```

Argument precedence is one less than the maximum precedence. This level is useful for defining syntax that should be treated as an argument to a function, such as `fun` or `do`.

```
prec ::= ...
    | arg
```

Lead precedence is less that argument precedence, and should be used for custom syntax that should not occur as a function argument, such as `let`.

```
prec ::= ...
    | lead
```

The minimum precedence can be used to ensure that an operator binds less tightly than all other operators.

```
prec ::= ...
    | min
```

20.3. Notations

The term *notation* is used in two ways in Lean: it can refer to the general concept of concise ways of writing down ideas, and it is the name of a language feature that allows notations to be conveniently implemented with little code. Like custom operators, Lean notations allow the grammar of terms to be extended with new forms. However, notations are more general: the new syntax may freely intermix required keywords or operators with subterms, and they provide more precise control over precedence levels. Notations may also rearrange their parameters in the resulting subterms, while infix operators provide them to the function term in a fixed order. Because notations may define operators that use a mix of prefix, infix, and postfix components, they can be called *mixfix* operators.

syntax

Notation Declarations

Notations are defined using the `notation` command.

```
command ::= ...
    | docComment?
      attributes?
```

```
      attrKind notation(:prec)? ((name := ident))? ((priority :=
prio))? notationItem* => term
```

syntax

Notation Items

The body of a notation definition consists of a sequence of *notation items*, which may be either string literals or identifiers with optional precedences.

```
notationItem ::=
    str

notationItem ::= ...
    | ident(:prec)?
```

As with operator declarations, the contents of the documentation comments are shown to users while they interact with the new syntax. Adding the **`inherit_doc`** attribute causes the documentation comment of the function at the head of the term into which the notation expands to be copied to the new syntax. Other attributes may be added to invoke other compile-time metaprograms on the resulting definition.

Notations interact with section scopes in the same manner as attributes and operators. By default, notations are available in any module that transitively imports the one in which they are established, but they may be declared `scoped` or `local` to restrict their availability either to contexts in which the current namespace has been opened or to the current section scope, respectively.

Like operators, the local longest-match rule is used while parsing notations. If more than one notation ties for the longest match, the declared priorities are used to determine which parse result applies. If this still does not resolve the ambiguity, then all are saved, and the elaborator is expected to attempt all of them, succeeding when exactly one can be elaborated.

Rather than a single operator with its fixity and token, the body of a notation declaration consists of a sequence of *notation items*, which may be either new atoms (including both keywords such as `if`, `#eval`, or `where` and symbols such as `=>`, `+`, `↗`, `⟦`, or `⋉`) or positions for terms. Just as they do in operators, string literals identify the placement of atoms. Leading and trailing spaces in the strings do not affect parsing, but they cause Lean to insert spaces in the corresponding position when displaying the syntax in proof states and error messages. Identifiers indicate positions where terms are expected, and name the corresponding term so it can be inserted in the notation's expansion.

While custom operators have a single notion of precedence, there are many involved in a notation. The notation itself has a precedence, as does each term to be parsed. The notation's precedence determines which contexts it may be parsed in: the parser only attempts to parse productions whose precedence is at least as high as the current context. For example, because multiplication has higher

precedence than addition, the parser will attempt to parse an infix multiplication term while parsing the arguments to addition, but not vice versa. The precedence of each term to be parsed determines which other productions may occur in them.

If no precedence is supplied for the notation itself, the default value depends on the form of the notation. If the notation both begins and ends with an atom (represented by string literals), then the default precedence is `max`. This applies both to notations that consist only of a single atom and to notations with multiple items, in which the first and last items are both atoms. Otherwise, the default precedence of the whole notation is `lead`. If no precedence is provided for notation items that are terms, then they default to precedence `min`.

After the required double arrow (`=>`), the notation is provided with an expansion. While operators are always expanded by applying their function to the operator's arguments in order, notations may place their term items in any position in the expansion. The terms are referred to by name. Term items may occur any number of times in the expansion. Because notation expansion is a purely syntactic process that occurs prior to elaboration or code generation, duplicating terms in the expansion may lead to duplicated computation when the resulting term is evaluated, or even duplicated side effects when working in a monad.

▼ *Example: Ignored Terms in Notation Expansion*

This notation ignores its first parameter:

```
notation (name := ignore) "ignore " _ign:arg e:arg => e
```

The term in the ignored position is discarded, and Lean never attempts to elaborate it, so terms that would otherwise result in errors can be used here:

```
#eval ignore (2 + "whatever") 5
```

```
5
```

However, the ignored term must still be syntactically valid:

```
1 #eval ignore (2 +) 5
```

```
<example>:1:17-1:18: unexpected token ')'; expected term
```

▼ *Example: Duplicated Terms in Notation Expansion*

The `dup!` notation duplicates its sub-term.

```
notation (name := dup) "dup!" t:arg => (t, t)
```

Because the term is duplicated, it can be elaborated separately with different types:

```
def e : Nat × Int := dup! (2 + 2)
```

Printing the resulting definition demonstrates that the work of addition will be performed twice:

```
#print e
```

```
def e : Nat × Int :=
(2 + 2, 2 + 2)
```

When the expansion consists of the application of a function defined in the global environment and each term in the notation occurs exactly once, an unexpander is generated. The new notation will be displayed in proof states, error messages, and other output from Lean when matching function application terms otherwise would have been displayed. As with custom operators, Lean does not track whether the notation was used in the original term; it is used at every opportunity in Lean's output.

20.3.1. Operators and Notations

Internally, operator declarations are translated into notation declarations. Term notation items are inserted where the operator would expect arguments, and in the corresponding positions in the expansion. For prefix and postfix operators, the notation's precedence as well as the precedences of its term items is the operator's declared precedence. For non-associative infix operators, the notation's precedence is the declared precedence, but both arguments are parsed at a precedence level that is one higher, which prevents successive uses of the notation without parentheses. Associative infix operators use the operator's precedence for the notation and for one argument, while a precedence that is one level higher is used for the other argument; this prevents successive applications in one direction only. Left-associative operators use the higher precedence for their right argument, while right-associative operators use the higher precedence for their left argument.

20.4. Defining New Syntax

Lean's uniform representation of syntax is very general and flexible. This means that extensions to Lean's parser do not require extensions to the representation of parsed syntax.

20.4.1. Syntax Model

Lean's parser produces a concrete syntax tree, of type `Lean.Syntax`. `Lean.Syntax` is an inductive type that represents all of Lean's syntax, including commands, terms, tactics, and any custom extensions. All of these are represented by a few basic building blocks:

Atoms

Atoms are the fundamental terminals of the grammar, including literals (such as those for characters and numbers), parentheses, operators, and keywords.

Identifiers

Identifiers represent names, such as *x*, `Nat`, or `Nat.add`. Identifier syntax includes a list of pre-resolved names that the identifier might refer to.

Nodes

Nodes represent the parsing of nonterminals. Nodes contain a *syntax kind*, which identifies the syntax rule that the node results from, along with an array of child `Syntax` values.

Missing Syntax

When the parser encounters an error, it returns a partial result, so Lean can provide some feedback about partially-written programs or programs that contain mistakes. Partial results contain one or more instances of missing syntax.

Atoms and identifiers are collectively referred to as *tokens*.

inductive type

`Lean.Syntax : Type`

Lean syntax trees.

Syntax trees are used pervasively throughout Lean: they are produced by the parser, transformed by the macro expander, and elaborated. They are also produced by the delaborator and presented to users.

Constructors

| `missing : Lean.Syntax`

A portion of the syntax tree that is missing because of a parse error.

The indexing operator on `Syntax` also returns `Syntax.missing` when the index is out of bounds.

| `node (info : Lean.SourceInfo) (kind : Lean.SyntaxNodeKind) (args : Array Lean.Syntax) : Lean.Syntax`

A node in the syntax tree that may have further syntax as child nodes. The node's *kind* determines its interpretation.

For nodes produced by the parser, the *info* field is typically `Lean.SourceInfo.none`, and source information is stored in the corresponding fields of identifiers and atoms. This field is used in two ways:

1. The delaborator uses it to associate nodes with metadata that are used to implement interactive features.
2. Nodes created by quotations use the field to mark the syntax as synthetic (storing the result of `Lean.SourceInfo.fromRef`) even when its leading or trailing tokens are not.

| `atom (info : Lean.SourceInfo) (val : String) : Lean.Syntax`

A non-identifier atomic component of syntax.

All of the following are atoms:

- keywords, such as **`def`**, **`fun`**, and **`inductive`**
- literals, such as numeric or string literals
- punctuation and delimiters, such as `(`, `)`, and `=>`.

Identifiers are represented by the `Lean.Syntax.ident` constructor. Atoms also correspond to quoted strings inside `syntax` declarations.

```
ident (info : Lean.SourceInfo) (rawVal : Substring)
  (val : Lean.Name)
  (preresolved : List Lean.Syntax.Preresolved) : Lean.Syntax
```

An identifier.

In addition to source information, identifiers have the following fields:

- *`rawVal`* is the literal substring from the input file
- *`val`* is the parsed Lean name, potentially including macro scopes.
- *`preresolved`* is the list of possible declarations this could refer to, populated by quotations.

inductive type

`Lean.Syntax.Preresolved : Type`

A possible binding of an identifier in the context in which it was quoted.

Identifiers in quotations may refer to either global declarations or to namespaces that are in scope at the site of the quotation. These are saved in the `Syntax.ident` constructor and are part of the implementation of hygienic macros.

Constructors

```
«namespace» (ns : Lean.Name) : Lean.Syntax.Preresolved
```

A potential namespace reference

```
decl (n : Lean.Name) (fields : List String) :
  Lean.Syntax.Preresolved
```

A potential global constant or section variable reference, with additional field accesses

20.4.2. Syntax Node Kinds

Syntax node kinds typically identify the parser that produced the node. This is one place where the names given to operators or notations (or their automatically-generated internal names) occur. While only nodes contain a field that identifies their kind, identifiers have the kind `identKind` by convention, while atoms have their internal string as their kind by convention. Lean's parser wraps each keyword atom `KW` in a singleton node whose kind is `` `token.KW ``. The kind of a syntax value can be extracted using `Syntax.getKind`.

def

```
Lean.SyntaxNodeKind : Type
```

Specifies the interpretation of a `Syntax.node` value. An abbreviation for `Name`.

Node kinds may be any name, and do not need to refer to declarations in the environment. Conventionally, however, a node's kind corresponds to the `Parser` or `ParserDesc` declaration that produces it. There are also a number of built-in node kinds that are used by the parsing infrastructure, such as `nullKind` and `choiceKind`; these do not correspond to parser declarations.

def

```
Lean.Syntax.isOfKind (stx : Lean.Syntax) (k : Lean.SyntaxNodeKind) :
  Bool
```

Checks whether syntax has the given kind or pseudo-kind.

“Pseudo-kinds” are kinds that are assigned by convention to non-`Syntax.node` values: `identKind` for `Syntax.ident`, `` `missing `` for `Syntax.missing`, and the atom's string literal for atoms.

def

```
Lean.Syntax.getKind (stx : Lean.Syntax) : Lean.SyntaxNodeKind
```

Gets the kind of a `Syntax.node` value, or the pseudo-kind of any other `Syntax` value.

“Pseudo-kinds” are kinds that are assigned by convention to non-`Syntax.node` values: `identKind` for `Syntax.ident`, `` `missing `` for `Syntax.missing`, and the atom's string literal for atoms.

def

```
Lean.Syntax.setKind (stx : Lean.Syntax) (k : Lean.SyntaxNodeKind) :
  Lean.Syntax
```

Changes the kind at the root of a `Syntax.node` to *k*.

Returns all other `Syntax` values unchanged.

20.4.3. Token and Literal Kinds

A number of named kinds are associated with the basic tokens produced by the parser. Typically, single-token syntax productions consist of a node that contains a single atom; the kind saved in the node allows the value to be recognized. Atoms for literals are not interpreted by the parser: string atoms include their leading and trailing double-quote characters along with any escape sequences contained within, and hexadecimal numerals are saved as a string that begins with `"0x"`. Helpers such as `Lean.TSyntax.getString` are provided to perform this decoding on demand.

def

`Lean.identKind : Lean.SyntaxNodeKind`

The pseudo-kind assigned to identifiers: `` `ident ``.

The name `` `ident `` is not actually used as a kind for `Syntax.node` values. It is used by convention as the kind of `Syntax.ident` values.

def

`Lean.strLitKind : Lean.SyntaxNodeKind`

`` `str `` is the node kind of string literals like `"foo"`.

def

`Lean.interpolatedStrKind : Lean.SyntaxNodeKind`

`` `interpolatedStrKind `` is the node kind of an interpolated string literal like `"value = {x}"` in `s!"value = {x}"`.

def

`Lean.interpolatedStrLitKind : Lean.SyntaxNodeKind`

`` `interpolatedStrLitKind `` is the node kind of interpolated string literal fragments like `"value = {` and `}"` in `s!"value = {x}"`.

def

`Lean.charLitKind : Lean.SyntaxNodeKind`

`` `char `` is the node kind of character literals like `'A'`.

def

`Lean.numLitKind : Lean.SyntaxNodeKind`

`num is the node kind of number literals like `42.

def

`Lean.scientificLitKind : Lean.SyntaxNodeKind`

`scientific is the node kind of floating point literals like `1.23e-3.

def

`Lean.nameLitKind : Lean.SyntaxNodeKind`

`name is the node kind of name literals like `` `foo.

def

`Lean.fieldIdxKind : Lean.SyntaxNodeKind`

`` fieldIdx is the node kind of projection indices like the 2 in `x.2`.

20.4.4. Internal Kinds

def

`Lean.groupKind : Lean.SyntaxNodeKind`

The `group kind is used for nodes that result from `Lean.Parser.group`. This avoids confusion with the null kind when used inside `optional`.

def

`Lean.nullKind : Lean.SyntaxNodeKind`

`null is the “fallback” kind, used when no other kind applies. Null nodes result from repetition operators, and empty null nodes represent the failure of an optional parse.

The null kind is used for raw list parsers like `many`.

def

`Lean.choiceKind : Lean.SyntaxNodeKind`

The `choice kind is used to represent ambiguous parse results.

The parser prioritizes longer matches over shorter ones, but there is not always a unique longest match. All the parse results are saved, and the determination of which to use is deferred until typing information is available.

def

`Lean.hygieneInfoKind : Lean.SyntaxNodeKind`

`` `hygieneInfo `` is the node kind of the `Lean.Parser.hygieneInfo` parser, which produces an "invisible token" that captures the hygiene information at the current point without parsing anything.

They can be used to generate identifiers (with `Lean.HygieneInfo.mkIdent`) as if they were introduced in a macro's input, rather than by its implementation.

20.4.5. Source Positions

Atoms, identifiers, and nodes optionally contain source information that tracks their correspondence with the original file. The parser saves source information for all tokens, but not for nodes; position information for parsed nodes is reconstructed from their first and last tokens. Not all `Syntax` data results from the parser: it may be the result of macro expansion, in which case it typically contains a mix of generated and parsed syntax, or it may be the result of delaborating an internal term to display it to a user. In these use cases, nodes may themselves contain source information.

Source information comes in two varieties:

Original

Original source information comes from the parser. In addition to the original source location, it also contains leading and trailing whitespace that was skipped by the parser, which allows the original string to be reconstructed. This whitespace is saved as offsets into the string representation of the original source code (that is, as `Substring`) to avoid having to allocate copies of substrings.

Synthetic

Synthetic source information comes from metaprograms (including macros) or from Lean's internals. Because there is no original string to be reconstructed, it does not save leading and trailing whitespace. Synthetic source positions are used to provide accurate feedback even when terms have been automatically transformed, as well as to track the correspondence between elaborated expressions and their presentation in Lean's output. A synthetic position may be marked *canonical*, in which case some operations that would ordinarily ignore synthetic positions will treat it as if it were not.

inductive type

`Lean.SourceInfo : Type`

Source information that relates syntax to the context that it came from.

The primary purpose of `SourceInfo` is to relate the output of the parser and the macro expander to the original source file. When produced by the parser, `Syntax.node` does not carry source info; the parser associates it only with atoms and identifiers. If a `Syntax.node` is introduced by a quotation, then it has synthetic source info that both associates it with an original reference position and indicates that the original atoms in it may not originate from the Lean file under elaboration.

Source info is also used to relate Lean's output to the internal data that it represents; this is the basis for many interactive features. When used this way, it can occur on `Syntax.node` as well.

Constructors

```
original (leading : Substring) (pos : String.Pos)
  (trailing : Substring) (endPos : String.Pos) :
  Lean.SourceInfo
```

A token produced by the parser from original input that includes both leading and trailing whitespace as well as position information.

The *leading* whitespace is inferred after parsing by `Syntax.updateLeading`. This is because the “preceding token” is not well-defined during parsing, especially in the presence of backtracking.

```
synthetic (pos endPos : String.Pos)
  (canonical : Bool := false) : Lean.SourceInfo
```

Synthetic syntax is syntax that was produced by a metaprogram or by Lean itself (e.g. by a quotation). Synthetic syntax is annotated with a source span from the original syntax, which relates it to the source file.

The delaborator uses this constructor to store an encoded indicator of which core language expression gave rise to the syntax.

The *canonical* flag on synthetic syntax is enabled for syntax that is not literally part of the original input syntax but should be treated “as if” the user really wrote it for the purpose of hovers and error messages. This is usually used on identifiers in order to connect the binding site to the user's original syntax even if the name of the identifier changes during expansion, as well as on tokens that should receive targeted messages.

Generally speaking, a macro expansion should only use a given piece of input syntax in a single canonical token. An exception to this rule is when the same identifier is used to declare two binders, as in the macro expansion for dependent if:

```
`(if $h : $cond then $t else $e) ~>
`(dite $cond (fun $h => $t) (fun $h => $t))
```

In these cases, if the user hovers over h they will see information about both binding sites.

```
none : Lean.SourceInfo
```

A synthesized token without position information.

20.4.6. Inspecting Syntax

There are three primary ways to inspect `Syntax` values:

The `Repr` Instance

The `Repr Syntax` instance produces a very detailed representation of syntax in terms of the constructors of the `Syntax` type.

The `ToString` Instance

The `ToString Syntax` instance produces a compact view, representing certain syntax kinds with particular conventions that can make it easier to read at a glance. This instance suppresses source position information.

The Pretty Printer

Lean's pretty printer attempts to render the syntax as it would look in a source file, but fails if the nesting structure of the syntax doesn't match the expected shape.

▼ *Example: Representing Syntax as Constructors*

The `Repr` instance's representation of syntax can be inspected by quoting it in the context of `#eval`, which can run actions in the command elaboration monad `CommandElabM`. To reduce the size of the example output, the helper `removeSourceInfo` is used to remove source information prior to display.

```
partial def removeSourceInfo : Syntax → Syntax
  | .atom _ str => .atom .none str
  | .ident _ str x pre => .ident .none str x pre
  | .node _ k children => .node .none k (children.map
removeSourceInfo)
  | .missing => .missing

#eval do
  let stx ← `(2 + $(⟨.missing⟩))
  logInfo (repr (removeSourceInfo stx.raw))
```

```
Lean.Syntax.node
  (Lean.SourceInfo.none)
  `«term_+_»
  #[Lean.Syntax.node (Lean.SourceInfo.none) `num #
[Lean.Syntax.atom (Lean.SourceInfo.none) "2"],
    Lean.Syntax.atom (Lean.SourceInfo.none) "+",
Lean.Syntax.missing]
```

In the second example, macro scopes inserted by quotation are visible on the call to `List.length`.

```
#eval do
  let stx ← `(List.length ["Rose", "Daffodil", "Lily"])
  logInfo (repr (removeSourceInfo stx.raw))
```

The contents of the pre-resolved identifier `List.length` are visible here:

```
Lean.Syntax.node
  (Lean.SourceInfo.none)
  `Lean.Parser.Term.app
  #[Lean.Syntax.ident
      (Lean.SourceInfo.none)
      "List.length".toSubstring
      (Lean.Name.mkNum
`List.length._@.Manual.NotationsMacros.SyntaxDef._hyg 2)
      [Lean.Syntax.Preresolved.decl `List.length [],
Lean.Syntax.Preresolved.namespace `List.length],
    Lean.Syntax.node
      (Lean.SourceInfo.none)
      `null
      #[Lean.Syntax.node
          (Lean.SourceInfo.none)
          `«term[_]»
          #[Lean.Syntax.atom (Lean.SourceInfo.none) "[",
            Lean.Syntax.node
              (Lean.SourceInfo.none)
              `null
              #[Lean.Syntax.node (Lean.SourceInfo.none) `str
#[Lean.Syntax.atom (Lean.SourceInfo.none) "\"Rose\""],
                Lean.Syntax.atom (Lean.SourceInfo.none) ",",
                Lean.Syntax.node (Lean.SourceInfo.none) `str
#[Lean.Syntax.atom (Lean.SourceInfo.none) "\"Daffodil\""],
                Lean.Syntax.atom (Lean.SourceInfo.none) ",",
                Lean.Syntax.node (Lean.SourceInfo.none) `str
#[Lean.Syntax.atom (Lean.SourceInfo.none) "\"Lily\""]],
            Lean.Syntax.atom (Lean.SourceInfo.none) "]"]]]
```

The `ToString` instance represents the constructors of `Syntax` as follows:

- The `ident` constructor is represented as the underlying name. Source information and pre-resolved names are not shown.
- The `atom` constructor is represented as a string.
- The `missing` constructor is represented by `<missing>`.
- The representation of the `node` constructor depends on the kind. If the kind is `` `null ``, then the node is represented by its child nodes order in square brackets. Otherwise, the node is represented by its kind followed by its child nodes, both surrounded by parentheses.

▼ *Example: Syntax as Strings*

The string representation of syntax can be inspected by quoting it in the context of `#eval`, which can run actions in the command elaboration monad `CommandElabM`.

```
#eval do
  let stx ← `(2 + $(⟨.missing⟩))
  logInfo (toString stx)
```

```
(«term_+_» (num "2") "+" <missing>)
```

In the second example, macro scopes inserted by quotation are visible on the call to `List.length`.

```
#eval do
  let stx ← `(List.length ["Rose", "Daffodil", "Lily"])
  logInfo (toString stx)
```

```
(Term.app
 `List.length._@.Manual.NotationsMacros.SyntaxDef._hyg.2
 [(«term[_]» "[" [(str "\"Rose\"") "," (str "\"Daffodil\"")
"," (str "\"Lily\"")] "]")])
```

Pretty printing syntax is typically most useful when including it in a message to a user. Normally, Lean automatically invokes the pretty printer when necessary. However, `ppTerm` can be explicitly invoked if needed.

▼ *Example: Pretty-Printed Syntax*

The string representation of syntax can be inspected by quoting it in the context of `#eval`, which can run actions in the command elaboration monad `CommandElabM`. Because new syntax declarations also equip the pretty printer with instructions for displaying them, the pretty printer requires a configuration object. This context can be constructed with a helper:

```
def getPPContext : CommandElabM PPContext := do
  return {
    env := (← getEnv),
    opts := (← getOptions),
    currNamespace := (← getCurrNamespace),
    openDecls := (← getOpenDecls)
  }

#eval show CommandElabM Unit from do
  let stx ← `(2 + 5)
  let fmt ← ppTerm (← getPPContext) stx
  logInfo fmt
```

```
2 + 5
```

In the second example, the macro scopes inserted on `List.length` by quotation cause it to be displayed with a dagger (`✝`).

```
#eval do
  let stx ← `(List.length ["Rose", "Daffodil", "Lily"])
  let fmt ← ppTerm (← getPPContext) stx
  logInfo fmt
```

```
List.length✝ ["Rose", "Daffodil", "Lily"]
```

Pretty printing wraps lines and inserts indentation automatically. A coercion typically converts the pretty printer's output to the type expected by `logInfo`, using a default layout width. The width can be controlled by explicitly calling `pretty` with a named argument.

```
#eval do
  let flowers := #["Rose", "Daffodil", "Lily"]
  let manyFlowers := flowers ++ flowers ++ flowers
  let stx ← `(List.length [$(manyFlowers.map (quote (k :=
`term))),*])
  let fmt ← ppTerm (← getPPContext) stx
  logInfo (fmt.pretty (width := 40))
```

```
List.length✝
  ["Rose", "Daffodil", "Lily", "Rose",
    "Daffodil", "Lily", "Rose",
    "Daffodil", "Lily"]
```

20.4.7. Typed Syntax

Syntax may additionally be annotated with a type that specifies which syntax category it belongs to. The `TSyntax` structure contains a type-level list of syntax categories along with a syntax tree. The list of syntax categories typically contains precisely one element, in which case the list structure itself is not shown.

structure

`Lean.TSyntax (ks : Lean.SyntaxNodeKinds) : Type`

Typed syntax, which tracks the potential kinds of the `Syntax` it contains.

While syntax quotations produce or expect `TSyntax` values of the correct kinds, this is not otherwise enforced; it can easily be circumvented by direct use of the constructor.

Constructor

`Lean.TSyntax.mk`

Fields

`raw : Lean.Syntax`

The underlying `Syntax` value.

def

```
Lean.SyntaxNodeKinds : Type
```

`SyntaxNodeKinds` is a set of `SyntaxNodeKind`, implemented as a list.

Singleton `SyntaxNodeKinds` are extremely common. They are written as name literals, rather than as lists; list syntax is required only for empty or non-singleton sets of kinds.

Quasiquotations prevent the substitution of typed syntax that does not come from the correct syntactic category. For many of Lean's built-in syntactic categories, there is a set of coercions that appropriately wrap one kind of syntax for another category, such as a coercion from the syntax of string literals to the syntax of terms. Additionally, many helper functions that are only valid on some syntactic categories are defined for the appropriate typed syntax only.

The constructor of `TSyntax` is public, and nothing prevents users from constructing values that break internal invariants. The use of `TSyntax` should be seen as a way to reduce common mistakes, rather than rule them out entirely.

In addition to `TSyntax`, there are types that represent arrays of syntax, with or without separators. These correspond to repeated elements in syntax declarations or antiquotations.

def

```
Lean.TSyntaxArray (ks : Lean.SyntaxNodeKinds) : Type
```

An array of syntaxes of kind *ks*.

structure

```
Lean.Syntax.TSepArray (ks : Lean.SyntaxNodeKinds) (sep : String) :
Type
```

An array of syntax elements that alternate with the given separator. Each syntax element has a kind drawn from *ks*.

Separator arrays result from repetition operators such as `,*`. Coercions to and from `Array (TSyntax ks)` insert or remove separators as required. The untyped equivalent is `Lean.Syntax.SepArray`.

Constructor

`Lean.Syntax.TSepArray.mk`

Fields

`elemsAndSeps : Array Lean.Syntax`

The array of elements and separators, ordered like `#[el1, sep1, el2, sep2, el3]`.

20.4.8. Aliases

A number of aliases are provided for commonly-used typed syntax varieties. These aliases allow code to be written at a higher level of abstraction.

def

`Lean.Syntax.Term : Type`

Syntax that represents a Lean term.

def

`Lean.Syntax.Command : Type`

Syntax that represents a command.

def

`Lean.Syntax.Level : Type`

Syntax that represents a universe level.

def

`Lean.Syntax.Tactic : Type`

Syntax that represents a tactic.

def

`Lean.Syntax.Prec : Type`

Syntax that represents a precedence (e.g. for an operator).

def

`Lean.Syntax.Prio : Type`

Syntax that represents a priority (e.g. for an instance declaration).

def

```
Lean.Syntax.Ident : Type
```

Syntax that represents an identifier.

def

```
Lean.Syntax.StrLit : Type
```

Syntax that represents a string literal.

def

```
Lean.Syntax.CharLit : Type
```

Syntax that represents a character literal.

def

```
Lean.Syntax.NameLit : Type
```

Syntax that represents a quoted name literal that begins with a back-tick.

def

```
Lean.Syntax.NumLit : Type
```

Syntax that represents a numeric literal.

def

```
Lean.Syntax.ScientificLit : Type
```

Syntax that represents a scientific numeric literal that may have decimal and exponential parts.

def

```
Lean.Syntax.HygieneInfo : Type
```

Syntax that represents macro hygiene info.

20.4.9. Helpers for Typed Syntax

For literals, Lean's parser produces a singleton node that contains an `atom`. The inner atom contains a string with source information, while the node's kind specifies how the atom is to be interpreted. This may involve decoding string escape sequences or interpreting base-16 numeric literals. The helpers in this section perform the correct interpretation.

def

def

```
Lean.TSyntax.getId (s : Lean.Ident) : Lean.Name
```

Extracts the parsed name from the syntax of an identifier.

Returns `Name.anonymous` if the syntax is malformed.

def

```
Lean.TSyntax.getName (s : Lean.NameLit) : Lean.Name
```

Decodes a quoted name literal, returning the name.

Returns `Lean.Name.anonymous` if the syntax is malformed.

def

```
Lean.TSyntax.getNat (s : Lean.NumLit) : Nat
```

Interprets a numeric literal as a natural number.

Returns `0` if the syntax is malformed.

def

```
Lean.TSyntax.getScientific (s : Lean.ScientificLit) : Nat × Bool × Nat
```

Extracts the components of a scientific numeric literal.

Returns a triple `(n, sign, e) : Nat × Bool × Nat`; the number's value is given by:

```
if sign then n * 10 ^ (-e) else n * 10 ^ e
```

Returns `(0, false, 0)` if the syntax is malformed.

def

```
Lean.TSyntax.getString (s : Lean.StrLit) : String
```

Decodes a string literal, removing quotation marks and unescaping escaped characters.

Returns `""` if the syntax is malformed.

def

```
Lean.TSyntax.getChar (s : Lean.CharLit) : Char
```

Decodes a character literal.

Returns `(default : Char)` if the syntax is malformed.

def

```
Lean.TSyntax.getHygieneInfo (s : Lean.HygieneInfo) : Lean.Name
```

Decodes macro hygiene information.

20.4.10. Syntax Categories

Lean's parser contains a table of *syntax categories*, which correspond to nonterminals in a context-free grammar. Some of the most important categories are terms, commands, universe levels, priorities, precedences, and the categories that represent tokens such as literals. Typically, each syntax kind corresponds to a category. New categories can be declared using `declare_syntax_cat`.

syntax

Declaring Syntactic Categories

Declares a new syntactic category.

```
command ::= ...
    | docComment?
      declare_syntax_cat ident ((behavior := (catBehaviorBoth |
catBehaviorSymbol)))?
```

The leading identifier behavior is an advanced feature that usually does not need to be modified. It controls the behavior of the parser when it encounters an identifier, and can sometimes cause the identifier to be treated as a non-reserved keyword instead. This is used to avoid turning the name of every tactic into a reserved keyword.

inductive type

```
Lean.Parser.LeadingIdentBehavior : Type
```

Specifies how the parsing table lookup function behaves for identifiers.

The function `Lean.Parser.prattParser` uses two tables: one each for leading and trailing parsers. These tables map tokens to parsers. Because keyword tokens are distinct from identifier tokens, keywords and identifiers cannot be confused, even when they are syntactically identical. Specifying an alternative leading identifier behavior allows greater flexibility and makes it possible to avoid reserved keywords in some situations.

When the leading token is syntactically an identifier, the current syntax category's `LeadingIdentBehavior` specifies how the parsing table lookup function behaves, and allows controlled “punning” between identifiers and keywords. This feature is used to avoid

creating a reserved symbol for each built-in tactic (e.g., `apply` or `assumption`). As a result, tactic names can be used as identifiers.

Constructors

| `default : Lean.Parser.LeadingIdentBehavior`

If the leading token is an identifier, then the parser just executes the parsers associated with the auxiliary token “ident”, which parses identifiers.

| `symbol : Lean.Parser.LeadingIdentBehavior`

If the leading token is an identifier `<foo>`, and there are parsers P associated with the token `<foo>`, then the parser executes P. Otherwise, it executes only the parsers associated with the auxiliary token “ident”, which parses identifiers.

| `both : Lean.Parser.LeadingIdentBehavior`

If the leading token is an identifier `<foo>`, then it executes the parsers associated with token `<foo>` and parsers associated with the auxiliary token “ident”, which parses identifiers.

20.4.11. Syntax Rules

Each syntax category is associated with a set of *syntax rules*, which correspond to productions in a context-free grammar. Syntax rules can be defined using the `syntax` command.

syntax

Syntax Rules

```
command ::= ...
    | docComment?
      attributes?
      attrKind
      syntax(:prec)? ((name := ident))? ((priority := prio))? stx*
: ident
```

As with operator and notation declarations, the contents of the documentation comments are shown to users while they interact with the new syntax. Attributes may be added to invoke compile-time metaprograms on the resulting definition.

Syntax rules interact with section scopes in the same manner as attributes, operators, and notations. By default, syntax rules are available to the parser in any module that transitively imports the one in which they are established, but they may be declared `scoped` or `local` to restrict their availability either to contexts in which the current namespace has been opened or to the current section scope, respectively.

When multiple syntax rules for a category can match the current input, the local longest-match rule is used to select one of them. Like notations and operators, if there is a tie for the longest match then the declared priorities are used to determine which parse result applies. If this still does not resolve the ambiguity, then all the results that tied are saved. The elaborator is expected to attempt all of them, succeeding when exactly one can be elaborated.

The syntax rule's precedence, written immediately after the `syntax` keyword, restricts the parser to use this new syntax only when the precedence context is at least the provided value. Just as with operators and notations, syntax rules may be manually provided with a name; if they are not, an otherwise-unused name is generated. Whether provided or generated, this name is used as the syntax kind in the resulting node.

The body of a syntax declaration is even more flexible than that of a notation. String literals specify atoms to match. Subterms may be drawn from any syntax category, rather than just terms, and they may be optional or repeated, with or without interleaved comma separators. Identifiers in syntax rules indicate syntax categories, rather than naming subterms as they do in notations.

Finally, the syntax rule specifies which syntax category it extends. It is an error to declare a syntax rule in a nonexistent category.

syntax

Syntax Specifiers

The syntactic category `stx` is the grammar of specifiers that may occur in the body of a `syntax` command.

String literals are parsed as atoms (including both keywords such as `if`, `#eval`, or `where`):

```
stx ::=
    str
```

Leading and trailing spaces in the strings do not affect parsing, but they cause Lean to insert spaces in the corresponding position when displaying the syntax in proof states and error messages. Ordinarily, valid identifiers occurring as atoms in syntax rules become reserved keywords. Preceding a string literal with an ampersand (&) suppresses this behavior:

```
stx ::= ...
    | &str
```

Identifiers specify the syntactic category expected in a given position, and may optionally provide a precedence:

```
stx ::= ...
    | ident(:prec)?
```

The `*` modifier is the Kleene star, matching zero or more repetitions of the preceding syntax. It can also be written using `many`.

```
stx ::= ...
    | stx *
```

The + modifier matches one or more repetitions of the preceding syntax. It can also be written using `many1`.

```
stx ::= ...
    | stx +
```

The ? modifier makes a subterm optional, and matches zero or one, but not more, repetitions of the preceding syntax. It can also be written as `optional`.

```
stx ::= ...
    | stx ?
```

```
stx ::= ...
    | optional(stx)
```

The `,*` modifier matches zero or more repetitions of the preceding syntax with interleaved commas. It can also be written using `sepBy`.

```
stx ::= ...
    | stx ,*
```

The `,+` modifier matches one or more repetitions of the preceding syntax with interleaved commas. It can also be written using `sepBy1`.

```
stx ::= ...
    | stx ,+
```

The `,*,?` modifier matches zero or more repetitions of the preceding syntax with interleaved commas, allowing an optional trailing comma after the final repetition. It can also be written using `sepBy` with the `allowTrailingSep` modifier.

```
stx ::= ...
    | stx ,*,?
```

The `,+,?` modifier matches one or more repetitions of the preceding syntax with interleaved commas, allowing an optional trailing comma after the final repetition. It can also be written using `sepBy1` with the `allowTrailingSep` modifier.

```
stx ::= ...
    | stx ,+,?
```

The `<|>` operator, which can be written `orelse`, matches either syntax. However, if the first branch consumes any tokens, then it is committed to, and failures will not be backtracked:

```
stx ::= ...
    | stx <|> stx
```

```
stx ::= ...
    | orelse(stx, stx)
```

The `!` operator matches the complement of its argument. If its argument fails, then it succeeds, resetting the parsing state.

```
stx ::= ...
    | ! stx
```

Syntax specifiers may be grouped using parentheses.

```
stx ::= ...
    | (stx)
```

Repetitions may be defined using `many` and `many1`. The latter requires at least one instance of the repeated syntax.

```
stx ::= ...
    | many(stx)

stx ::= ...
    | many1(stx)
```

Repetitions with separators may be defined using `sepBy` and `sepBy1`, which respectively match zero or more occurrences and one or more occurrences, separated by some other syntax. They come in three varieties:

- The two-parameter version uses the atom provided in the string literal to parse the separators, and does not allow trailing separators.
- The three-parameter version uses the third parameter to parse the separators, using the atom for pretty-printing.
- The four-parameter version optionally allows the separator to occur an extra time at the end of the sequence. The fourth argument must always literally be the keyword `allowTrailingSep`.

```
stx ::= ...
    | sepBy(stx, str)

stx ::= ...
    | sepBy(stx, str, stx)

stx ::= ...
    | sepBy(stx, str, stx, allowTrailingSep)

stx ::= ...
    | sepBy1(stx, str)
```

```
stx ::= ...
    | sepBy1(stx, str, stx)

stx ::= ...
    | sepBy1(stx, str, stx, allowTrailingSep)
```

▼ *Example: Parsing Matched Parentheses and Brackets*

A language that consists of matched parentheses and brackets can be defined using syntax rules. The first step is to declare a new syntax category:

```
declare_syntax_cat balanced
```

Next, rules can be added for parentheses and square brackets. To rule out empty strings, the base cases consist of empty pairs.

```
syntax "(" ")" : balanced
syntax "[" "]" : balanced
syntax "(" balanced ")" : balanced
syntax "[" balanced "]" : balanced
syntax balanced balanced : balanced
```

In order to invoke Lean's parser on these rules, there must also be an embedding from the new syntax category into one that may already be parsed:

```
syntax (name := termBalanced) "balanced " balanced : term
```

These terms cannot be elaborated, but reaching an elaboration error indicates that parsing succeeded:

```
/--
error: elaboration function for 'termBalanced' has not been
implemented
  balanced ()
-/
#guard_msgs in
example := balanced ()

/--
error: elaboration function for 'termBalanced' has not been
implemented
  balanced []
-/
#guard_msgs in
example := balanced []

/--
error: elaboration function for 'termBalanced' has not been
implemented
  balanced [[]()([])]
```

```
-/
#guard_msgs in
example := balanced [[] () ([])]
```

Similarly, parsing fails when they are mismatched:

```
1 example := balanced [() (]]
```

```
<example>:1:25-1:26: unexpected token ']'; expected ')' or
balanced
```

▼ *Example: Parsing Comma-Separated Repetitions*

A variant of list literals that requires double square brackets and allows a trailing comma can be added with the following syntax:

```
syntax "[[" term,*,? "]]" : term
```

Adding a macro that describes how to translate it into an ordinary list literal allows it to be used in tests.

```
macro_rules
  | `(term|[[$e:term,*]]) => `([$e,*])

#eval [["Dandelion", "Thistle",]]
```

```
["Dandelion", "Thistle"]
```

20.4.12. Indentation

Internally, the parser maintains a saved source position. Syntax rules may include instructions that interact with these saved positions, causing parsing to fail when a condition is not met. Indentation-sensitive constructs, such as do, save a source position, parse their constituent parts while taking this saved position into account, and then restore the original position.

In particular, Indentation-sensitvity is specified by combining `withPosition` or `withPositionAfterLinebreak`, which save the source position at the start of parsing some other syntax, with `colGt`, `colGe`, and `colEq`, which compare the current column with the column from the most recently-saved position. `lineEq` can also be used to ensure that two positions are on the same line in the source file.

parser alias

withPosition(p)

Arity is sum of arguments' arities

`withPosition(p)` runs p while setting the "saved position" to the current position. This has no effect on its own, but various other parsers access this position to achieve some composite effect:

- `colGt`, `colGe`, `colEq` compare the column of the saved position to the current position, used to implement Python-style indentation sensitive blocks
- `lineEq` ensures that the current position is still on the same line as the saved position, used to implement composite tokens

The saved position is only available in the read-only state, which is why this is a scoping parser: after the `withPosition(..)` block the saved position will be restored to its original value.

This parser has the same arity as p - it just forwards the results of p.

parser alias

`withoutPosition``(p)`

Arity is sum of arguments' arities

`withoutPosition(p)` runs p without the saved position, meaning that position-checking parsers like `colGt` will have no effect. This is usually used by bracketing constructs like `(...)` so that the user can locally override whitespace sensitivity.

This parser has the same arity as p - it just forwards the results of p.

parser alias

`withPositionAfterLinebreak`

- Arity: 1
- Automatically wraps arguments in a `null` node unless there's exactly one

parser alias

`colGt`

- Arity: 0
- Automatically wraps arguments in a `null` node unless there's exactly one

The `colGt` parser requires that the next token starts a strictly greater column than the saved position (see `withPosition`). This can be used for whitespace sensitive syntax for the arguments to a tactic, to ensure that the following tactic is not interpreted as an argument.

```
example (x : False) : False := by
  revert x
  exact id
```

Here, the `revert` tactic is followed by a list of `colGt ident`, because otherwise it would interpret `exact` as an identifier and try to revert a variable named `exact`.

This parser has arity 0 - it does not capture anything.

parser alias

`colGe`

- Arity: 0
- Automatically wraps arguments in a `null` node unless there's exactly one

The `colGe` parser requires that the next token starts from at least the column of the saved position (see `withPosition`), but allows it to be more indented. This can be used for whitespace sensitive syntax to ensure that a block does not go outside a certain indentation scope. For example it is used in the lean grammar for `else if`, to ensure that the `else` is not less indented than the `if` it matches with.

This parser has arity 0 - it does not capture anything.

parser alias

`colEq`

- Arity: 0
- Automatically wraps arguments in a `null` node unless there's exactly one

The `colEq` parser ensures that the next token starts at exactly the column of the saved position (see `withPosition`). This can be used to do whitespace sensitive syntax like a `by` block or `do` block, where all the lines have to line up.

This parser has arity 0 - it does not capture anything.

parser alias

`lineEq`

- Arity: 0
- Automatically wraps arguments in a `null` node unless there's exactly one

The `lineEq` parser requires that the current token is on the same line as the saved position (see `withPosition`). This can be used to ensure that composite tokens are not "broken up" across different lines. For example, `else if` is parsed using `lineEq` to ensure that the two tokens are on the same line.

This parser has arity 0 - it does not capture anything.

▼ *Example: Aligned Columns*

This syntax for saving notes takes a bulleted list of items, each of which must be aligned at the same column.

```
syntax "note " ppLine withPosition((colEq "◦ " str ppLine)+) :
term
```

There is no elaborator or macro associated with this syntax, but the following example is accepted by the parser:

```
#check
  note
    ◦ "One"
    ◦ "Two"
```

```
elaboration function for '«termNote__◦__»' has not been
implemented
  note
    ◦ "One"
    ◦ "Two"
```

The syntax does not require that the list is indented with respect to the opening token, which would require an extra `withPosition` and a `colGt`.

```
#check
  note
◦ "One"
◦ "Two"
```

```
elaboration function for '«termNote__◦__»' has not been
implemented
  note
    ◦ "One"
    ◦ "Two"
```

The following examples are not syntactically valid because the columns of the bullet points do not match.

```
1 #check
2   note
3     ◦ "One"
4    ◦ "Two"
```

```
<example>:4:3-4:4: expected end of input
```

```
1 #check
2   note
3    ◦ "One"
4      ◦ "Two"
```

```
<example>:4:5-4:6: expected end of input
```

20.5. Macros

Macros are transformations from Syntax to Syntax that occur during elaboration and during tactic execution. Replacing syntax with the result of transforming it with a macro is called *macro expansion*. Multiple macros may be associated with a single syntax kind, and they are attempted in order of definition. Macros are run in a monad that has access to some compile-time metadata and has the ability to either emit an error message or to delegate to subsequent macros, but the macro monad is much less powerful than the elaboration monads.

Macros are associated with syntax kinds. An internal table maps syntax kinds to macros of type Syntax → MacroM Syntax. Macros delegate to the next entry in the table by throwing the unsupportedSyntax exception. A given Syntax value *is a macro* when there is a macro associated with its syntax kind that does not throw unsupportedSyntax. If a macro throws any other exception, an error is reported to the user. Syntax categories are irrelevant to macro expansion; however, because each syntax kind is typically associated with a single syntax category, they do not interfere in practice.

▼ *Example: Macro Error Reporting*

The following macro reports an error when its parameter is the literal numeral five. It expands to its argument in all other cases.

```
syntax &"notFive" term:arg : term
open Lean in
macro_rules
  | `(term|notFive 5) =>
    Macro.throwError "'5' is not allowed here"
  | `(term|notFive $e) =>
    pure e
```

When applied to terms that are not syntactically the numeral five, elaboration succeeds:

```
#eval notFive (2 + 3)
```

```
5
```

When the error case is triggered, the user receives an error message:

```
#eval notFive 5
```

```
'5' is not allowed here
```

Before elaborating a piece of syntax, the elaborator checks whether its syntax kind has macros associated with it. These are attempted in order. If a macro succeeds, potentially returning syntax with a different kind, the check is repeated and macros are expanded again until the outermost layer of syntax is no longer a macro. Elaboration or tactic execution can then proceed. Only the outermost layer of syntax (typically a node) is expanded, and the output of macro expansion may contain nested syntax that is a macro. These nested macros are expanded in turn when the elaborator reaches them.

In particular, macro expansion occurs in three situations in Lean:

1. During term elaboration, macros in the outermost layer of the syntax to be elaborated are expanded prior to invoking the syntax's term elaborator.
2. During command elaboration, macros in the outermost layer of the syntax to be elaborated are expanded prior to invoking the syntax's command elaborator.
3. During tactic execution, macros in the outermost layer of the syntax to be elaborated are expanded prior to executing the syntax as a tactic.

20.5.1. Hygiene

A macro is *hygienic* if its expansion cannot result in identifier capture. Identifier capture is when an identifier ends up referring to a binding site other than that which is in scope where the identifier occurs in the source code. There are two types of identifier capture:

- If a macro's expansion introduces binders, then identifiers that are parameters to the macro may end up referring to the introduced binders if their names happen to match.
- If a macro's expansion is intended to refer to a name, but the macro is used in a context that either locally binds this name or in which a new global name has been introduced, it may end up referring to the wrong name.

The first kind of variable capture can be avoided by ensuring that every binding introduced by a macro uses a freshly generated, globally-unique name, while the second can be avoided by always using fully-qualified names to refer to constants. The fresh names must be generated again at each invocation of the macro to avoid variable capture in recursive macros. These techniques are error-prone. Variable capture issues are difficult to test for because they rely on coincidences of name choices, and consistently applying these techniques results in noisy code.

Lean features automatic hygiene: in almost all cases, macros are automatically hygienic. Capture by introduced bindings is avoided by annotating identifiers introduced by a macro with *macro scopes*, which uniquely identify each invocation of macro expansion. If the binding and the use of the identifier have the same macro scopes, then they were introduced by the same step of macro expansion and should refer to one another. Similarly, uses of global names in code generated by a macro are not captured by local bindings in the context in which they are expanded because these use sites have macro scopes that are not present in the binding occurrence. Capture by newly-introduced global names is prevented by annotating potential global name references with the set of global names that match at quotation time in code produced in the macro's body. Identifiers

annotated with potential referents are called *pre-resolved identifiers*, and the `Syntax.Preresolved` field on the `Syntax.ident` constructor is used to store the potential referents. During elaboration, if an identifier has pre-resolved global names associated with it, then other global names are not considered as valid reference targets.

The introduction of macro scopes and pre-resolved identifiers to generated syntax occurs during quotation. Macros that construct syntax by other means than quotation should also ensure hygiene by some other means. For more details on Lean's hygiene algorithm, please consult Ullrich and de Moura (2020) (i Sebastian Ullrich and Leonardo de Moura, 2020. “Beyond notations: Hygienic macro expansion for theorem proving languages”. In *Proceedings of the International Joint Conference on Automated Reasoning.*) and Ullrich (2023) (i Sebastian Ullrich, 2023. *An Extensible Theorem Proving Frontend*. Dr. Ing. dissertation, Karlsruhe Institute of Technology).

20.5.2. The Macro Monad

The macro monad `MacroM` is sufficiently powerful to implement hygiene and report errors. Macro expansion does not have the ability to modify the environment directly, to carry out unification, to examine the current local context, or to do anything else that only makes sense in one particular context. This allows the same macro mechanism to be used throughout Lean, and it makes macros much easier to write than elaborators.

def

```
Lean.MacroM (α : Type) : Type
```

The `MacroM` monad is the main monad for macro expansion. It has the information needed to handle hygienic name generation, and is the monad that **`macro`** definitions live in.

Notably, this is a (relatively) pure monad: there is no `IO` and no access to the `Environment`. That means that things like declaration lookup are impossible here, as well as `IO.Ref` or other side-effecting operations. For more capabilities, macros can instead be written as **`elab`** using `adaptExpander`.

def

```
Lean.Macro.expandMacro? (stx : Lean.Syntax) :
  Lean.MacroM (Option Lean.Syntax)
```

`expandMacro? stx` returns `some stxNew` if `stx` is a macro, and `stxNew` is its expansion.

def

```
Lean.Macro.trace (clsName : Lean.Name) (msg : String) : Lean.MacroM
Unit
```

Add a new trace message, with the given trace class and message.

20.5.2.1. Exceptions and Errors

The `unsupportedSyntax` exception is used for control flow during macro expansion. It indicates that the current macro is incapable of expanding the received syntax, but that an error has not occurred. The exceptions thrown by `throwError` and `throwErrorAt` terminate macro expansion, reporting the error to the user.

def

```
Lean.Macro.throwUnsupported {α : Type} : Lean.MacroM α
```

Throw an `unsupportedSyntax` exception.

constructor of Lean.Macro.Exception

```
Lean.Macro.Exception.unsupportedSyntax : Lean.Macro.Exception
```

An unsupported syntax exception. We keep this separate because it is used for control flow: if one macro does not support a syntax then we try the next one.

def

```
Lean.Macro.throwError {α : Type} (msg : String) : Lean.MacroM α
```

Throw an error with the given message, using the `ref` for the location information.

def

```
Lean.Macro.throwErrorAt {α : Type} (ref : Lean.Syntax) (msg :
String) :
  Lean.MacroM α
```

Throw an error with the given message and location information.

20.5.2.2. Hygiene-Related Operations

Hygiene is implemented by adding macro scopes to the identifiers that occur in syntax. Ordinarily, the process of quotation adds all necessary scopes, but macros that construct syntax directly must add macro scopes to the identifiers that they introduce.

def

```
Lean.Macro.withFreshMacroScope {α : Type} (x : Lean.MacroM α) :
  Lean.MacroM α
```

Increments the macro scope counter so that inside the body of *x* the macro scope is fresh.

def

```
Lean.Macro.addMacroScope (n : Lean.Name) : Lean.MacroM Lean.Name
```

Add a new macro scope to the name *n*.

20.5.2.3. Querying the Environment

Macros have only limited support for querying the environment. They can check whether a constant exists and resolve names, but further introspection is unavailable.

def

```
Lean.Macro.hasDecl (declName : Lean.Name) : Lean.MacroM Bool
```

Returns `true` if the environment contains a declaration with name *declName*

def

```
Lean.Macro.getCurrNamespace : Lean.MacroM Lean.Name
```

Gets the current namespace given the position in the file.

def

```
Lean.Macro.resolveNamespace (n : Lean.Name) :
  Lean.MacroM (List Lean.Name)
```

Resolves the given name to an overload list of namespaces.

def

```
Lean.Macro.resolveGlobalName (n : Lean.Name) :
  Lean.MacroM (List (Lean.Name × List String))
```

Resolves the given name to an overload list of global definitions. The `List String` in each alternative is the deduced list of projections (which are ambiguous with name components).

Remark: it will not trigger actions associated with reserved names. Recall that Lean has reserved names. For example, a definition `foo` has a reserved name `foo.def` for theorem containing stating that `foo` is equal to its definition. The action associated with `foo.def` automatically proves the theorem. At the macro level, the name is resolved, but the action is not executed. The actions are executed by the elaborator when converting `Syntax` into `Expr`.

20.5.3. Quotation

Quotation marks code for representation as data of type `Syntax`. Quoted code is parsed, but not elaborated—while it must be syntactically correct, it need not make sense. Quotation makes it much easier to programmatically generate code: rather than reverse-engineering the specific nesting of `node` values that Lean's parser would produce, the parser can be directly invoked to create them. This is also more robust in the face of refactoring of the grammar that may change the internals of the parse tree without affecting the user-visible concrete syntax. Quotation in Lean is surrounded by `` `( `` and `)`.

The syntactic category or parser being quoted may be indicated by placing its name after the opening backtick and parenthesis, followed by a vertical bar (`|`). As a special case, the name `tactic` may be used to parse either tactics or sequences of tactics. If no syntactic category or parser is provided, Lean attempts to parse the quotation both as a term and as a non-empty sequence of commands. Term quotations have higher priority than command quotations, so in cases of ambiguity, the interpretation as a term is chosen; this can be overridden by explicitly indicating that the quotation is of a command sequence.

▼ *Example: Term vs Command Quotation Syntax*

In the following example, the contents of the quotation could either be a function application or a sequence of commands. Both match the same region of the file, so the `local longest-match rule` is not relevant. Term quotation has a higher priority than command quotation, so the quotation is interpreted as a term. Terms expect their `antiquotations` to have type `TSyntax` `` `term `` rather than `TSyntax` `` `command ``.

```
example (cmd1 cmd2 : TSyntax `command) : MacroM (TSyntax
`command) := `($cmd1 $cmd2)
```

The result is two type errors like the following:

```
Application type mismatch: In the application
  cmd1.raw
the argument
  cmd1
has type
  TSyntax `command : Type
but is expected to have type
  TSyntax `term : Type
```

The type of the quotation (`MacroM (TSyntax `command)`) is not used to select a result because syntax priorities are applied prior to elaboration. In this case, specifying that the antiquotations are commands resolves the ambiguity because function application would require terms in these positions:

```
example (cmd1 cmd2 : TSyntax `command) : MacroM (TSyntax
`command) := `($cmd1:command $cmd2:command)
```

Similarly, inserting a command into the quotation eliminates the possibility that it could be a term:

```
example (cmd1 cmd2 : TSyntax `command) : MacroM (TSyntax
`command) := `($cmd1 $cmd2 #eval "hello!")
```

syntax

Quotations

Lean's syntax includes quotations for terms, commands, tactics, and sequences of tactics, as well as a general quotation syntax that allows any input that Lean can parse to be quoted. Term quotations have the highest priority, followed by tactic quotations, general quotations, and finally command quotations.

```
term ::=
      `(term)
    | `(command+)
    | `(tactic|tactic)
    | `(tactic|tactic;*)
    | `(p:ident|Parse a p here)
```

Rather than having type Syntax, quotations are monadic actions with type `m Syntax`. Quotation is monadic because it implements hygiene by adding macro scopes and pre-resolving identifiers, as described in the section on hygiene. The specific monad to be used is an implicit parameter to the quotation, and any monad for which there is an instance of the `MonadQuotation` type class is suitable. `MonadQuotation` extends `MonadRef`, which gives the quotation access to the source location of the syntax that the macro expander or elaborator is currently processing. `MonadQuotation` additionally includes the ability to add macro scopes to identifiers and use a fresh macro scope for a sub-task. Monads that support quotation include `MacroM`, `TermElabM`, `CommandElabM`, and `TacticM`.

20.5.3.1. Quasiquotation

Quasiquotation is a form of quotation that may contain *antiquotations*, which are regions of the quotation that are not quoted, but instead are expressions that are evaluated to yield syntax. A quasiquotation is essentially a template; the outer quoted region provides a fixed framework that always yields the same outer syntax, while the antiquotations yield the parts of the final syntax that vary. All quotations in Lean are quasiquotations, so no special syntax is needed to distinguish quasiquotations from other quotations. The quotation process does not add macro scopes to identifiers that are inserted via antiquotations, because these identifiers either come from another quotation (in which case they already have macro scopes) or from the macro's input (in which case they should not have macro scopes, because they are not introduced by the macro).

Basic antiquotations consist of a dollar sign ($) immediately followed by an identifier. This means that the value of the corresponding variable, which should be a syntax tree, is to be substituted into this position of the quoted syntax. Entire expressions may be used as antiquotations by wrapping them in parentheses.

Lean's parser assigns every antiquotation a syntax category based on what the parser expects at the given position. If the parser expects syntax category *c*, then the antiquotation's type is `TSyntax c`.

Some syntax categories can be matched by elements of other categories. For example, numeric and string literals are valid terms in addition to being their own syntax categories. Antiquotations may be annotated with the expected category by suffixing them with a colon and the category name, which causes the parser to validate that the annotated category is acceptable in the given position and construct any intermediate layers that are required in the parse tree.

syntax

Antiquotations

```
antiquot ::=
      $ident(:ident)?
    | $(term)(:ident)?
```

Whitespace is not permitted between the dollar sign ('$') that initiates an antiquotation and the identifier or parenthesized term that follows. Similarly, no whitespace is permitted around the colon that annotates the syntax category of the antiquotation.

▼ *Example: Quasiquotation*

Both forms of antiquotation are used in this example. Because natural numbers are not syntax, `quote` is used to transform a number into syntax that represents it.

```
open Lean in
example [Monad m] [MonadQuotation m] (x : Term) (n : Nat) : m
Syntax :=
  `($x + $(quote (n + 2)))
```

▼ *Example: Antiquotation Annotations*

This example requires that *m* is a monad that can perform quotation.

```
variable {m : Type → Type} [Monad m] [MonadQuotation m]
```

By default, the antiquotation `$e` is expected to be a term, because that's the syntactic category that's immediately expected as the second argument to addition.

```
def ex1 (e) := show m _ from `(2 + $e)
#check ex1
```

```
ex1 {m : Type → Type} [Monad m] [MonadQuotation m] (e :
TSyntax `term) : m (TSyntax `term)
```

Annotating $e as a numeric literal succeeds, because numeric literals are also valid terms. The expected type of the parameter e changes to TSyntax `num.

```
def ex2 (e) := show m _ from `(2 + $e:num)
#check ex2
```

```
ex2 {m : Type → Type} [Monad m] [MonadQuotation m] (e :
TSyntax `num) : m (TSyntax `term)
```

Spaces are not allowed between the dollar sign and the identifier.

```
1 def ex2 (e) := show m _ from `(2 + $ e:num)
```

```
<example>:1:34-1:36: unexpected token '$'; expected
'`(tactic|' or no space before spliced term
```

Spaces are also not allowed before the colon:

```
1 def ex2 (e) := show m _ from `(2 + $e :num)
```

```
<example>:1:37-1:39: unexpected token ':'; expected ')'
```

▼ *Example: Expanding Quasiquotation*

Printing the definition of f demonstrates the expansion of a quasiquotation.

```
open Lean in
def f [Monad m] [MonadQuotation m]
    (x : Term) (n : Nat) : m Syntax :=
  `(fun k => $x + $(quote (n + 2)) + k)
#print f
```

```
def f : {m : Type → Type} → [Monad m] → [Lean.MonadQuotation
m] → Lean.Term → Nat → m Syntax :=
fun {m} [Monad m] [Lean.MonadQuotation m] x n => do
  let info ← Lean.MonadRef.mkInfoFromRefPos
  let scp ← Lean.getCurrMacroScope
  let mainModule ← Lean.getMainModule
  pure
      {
          raw :=
            Syntax.node2 info `Lean.Parser.Term.fun
(Syntax.atom info "fun")
              (Syntax.node4 info `Lean.Parser.Term.basicFun
                (Syntax.node1 info `null (Syntax.ident info
```

```
"k".toSubstring' (Lean.addMacroScope mainModule `k scp) []))
                (Syntax.node info `null #[]) (Syntax.atom
info "=>")
                (Syntax.node3 info `«term_+_»
                  (Syntax.node3 info `«term_+_» x.raw
(Syntax.atom info "+") (Lean.quote `term (n + 2)).raw)
                  (Syntax.atom info "+")
                  (Syntax.ident info "k".toSubstring'
(Lean.addMacroScope mainModule `k scp) [])))  }.raw
```

In this output, the quotation is a do block. It begins by constructing the source information for the resulting syntax, obtained by querying the compiler about the current user syntax being processed. It then obtains the current macro scope and the name of the module being processed, because macro scopes are added with respect to a module to enable independent compilation and avoid the need for a global counter. It then constructs a node using helpers such as `Syntax.node1` and `Syntax.node2`, which create a `Syntax.node` with the indicated number of children. The macro scope is added to each identifier, and `TSyntax.raw` is used to extract the contents of typed syntax wrappers. The antiquotations of *x* and `quote (n + 2)` occur directly in the expansion, as parameters to `Syntax.node3`.

20.5.3.2. Splices

In addition to including other syntax via antiquotations, quasiquotations can include *splices*. Splices indicate that the elements of an array are to be inserted in order. The repeated elements may include separators, such as the commas between list or array elements. Splices may consist of an ordinary antiquotation with a *splice suffix*, or they may be *extended splices* that provide additional repeated structure.

Splice suffixes consist of either an asterisk or a valid atom followed by an asterisk (`*`). Suffixes may follow any identifier or term antiquotation. An antiquotation with the splice suffix `*` corresponds to a use of `many` or `many1`; both the `*` and `+` suffixes in syntax rules correspond to the `*` splice suffix. An antiquotation with a splice suffix that includes an atom prior to the asterisk corresponds to a use of `sepBy` or `sepBy1`. The splice suffix `?` corresponds to a use of `optional` or the `?` suffix in a syntax rule. Because `?` is a valid identifier character, identifiers must be parenthesized to use it as a suffix.

While there is overlap between repetition specifiers for syntax and antiquotation suffixes, they have distinct syntaxes. When defining syntax, the suffixes `*`, `+`, `,*`, `,+`, `,*,?`, and `,+,?` are built in to Lean. There is no shorter way to specify separators other than `,`. Antiquotation suffixes are either just `*` or whatever atom was provided to `sepBy` or `sepBy1` followed by `*`. The syntax repetitions `+` and `*` correspond to the splice suffix `*`; the repetitions `,*`, `,+`, `,*,?`, and `,+,?` correspond to `,*`. The optional suffix `?` in syntax and splices correspond with each other.

Syntax Repetition	Splice Suffix
`+` `*`	`*`
`,*` `,+` `,*,?` `,+,?`	`,*`

Syntax Repetition		Splice Suffix
`sepBy(_, "S")`	`sepBy1(_, "S")`	`S*`
`?`		`?`

▼ *Example: Suffixed Splices*

This example requires that *m* is a monad that can perform quotation.

```
variable {m : Type → Type} [Monad m] [MonadQuotation m]
```

By default, the antiquotation `$e` is expected to be an array of terms separated by commas, as is expected in the body of a list:

```
def ex1 (xs) := show m _ from `(#[$xs,*])
#check ex1
```

```
ex1 {m : Type → Type} [Monad m] [MonadQuotation m] (xs :
Syntax.TSepArray `term ",") : m (TSyntax `term)
```

However, Lean includes a collection of coercions between various representations of arrays that will automatically insert or remove separators, so an ordinary array of terms is also acceptable:

```
def ex2 (xs : Array (TSyntax `term)) :=
  show m _ from `(#[$xs,*])
#check ex2
```

```
ex2 {m : Type → Type} [Monad m] [MonadQuotation m] (xs :
Array (TSyntax `term)) : m (TSyntax `term)
```

Repetition annotations may also be used with term antiquotations and syntax category annotations. This example is in `CommandElabM` so the result can be conveniently logged.

```
def ex3 (size : Nat) := show CommandElabM _ from do
  let mut nums : Array Nat := #[]
  for i in [0:size] do
    nums := nums.push i
  let stx ← `(#[$(nums.map (Syntax.mkNumLit ∘ toString)):num,*])
  -- Using logInfo here causes the syntax to be rendered via
  -- the pretty printer.
  logInfo stx

#eval ex3 4
```

```
#[0, 1, 2, 3]
```

▼ *Example: Non-Comma Separators*

The following unconventional syntax for lists separates numeric elements by either em dashes or double asterisks, rather than by commas.

```
syntax "⟦" sepBy1(num, " — ") "⟧": term
syntax "⟦" sepBy1(num, " ** ") "⟧": term
```

This means that `—*` and `***` are valid splice suffixes between the ⟦ and ⟧ atoms. In the case of `***`, the first two asterisks are the atom in the syntax rule, while the third is the repetition suffix.

```
macro_rules
  | `(⟦$n:num—*⟧) => `(⟦$n***⟧)
  | `(⟦$n:num***⟧) => `([$n,*])

#eval ⟦1 — 2 — 3⟧
```

```
[1, 2, 3]
```

▼ *Example: Optional Splices*

The following syntax declaration optionally matches a term between two tokens. The parentheses around the nested `term` are needed because `term?` is a valid identifier.

```
syntax "⟨| " (term)? " |⟩": term
```

The `?` splice suffix for a term expects an `Option Term`:

```
def mkStx [Monad m] [MonadQuotation m]
    (e : Option Term) : m Term :=
  `(⟨| $(e)? |⟩)

#check mkStx
```

```
mkStx {m : Type → Type} [Monad m] [MonadQuotation m] (e :
Option Term) : m Term
```

Supplying `some` results in the optional term being present.

```
#eval do logInfo (← mkStx (some (quote 5)))
```

```
⟨| 5 |⟩
```

Supplying `none` results in the optional term being absent.

```
#eval do logInfo (← mkStx none)
```

```
⟨| |⟩
```

20.5.3.3. Token Antiquotations

In addition to antiquotations of complete syntax, Lean features *token antiquotations* which allow the source information of an atom to be replaced with the source information from some other syntax. The resulting synthetic source information is marked canonical so that it will be used for error messages, proof states, and other feedback. This is primarily useful to control the placement of error messages or other information that Lean reports to users. A token antiquotation does not allow an arbitrary atom to be inserted via evaluation. A token antiquotation consists of an atom (that is, a keyword)

syntax

Token Antiquotations

Token antiquotations replace the source information (of type `SourceInfo`) on a token with the source information from some other syntax.

```
antiquot ::= ...
    | atom%$ident
```

20.5.4. Matching Syntax

Quasiquotations can be used in pattern matching to recognize syntax that matches a template. Just as antiquotations in a quotation that's used as a term are regions that are treated as ordinary non-quoted expressions, antiquotations in patterns are regions that are treated as ordinary Lean patterns. Quote patterns are compiled differently from other patterns, so they can't be intermixed with non-quote patterns in a single `match` expression. Like ordinary quotations, quote patterns are first processed by Lean's parser. The parser's output is then compiled into code that determines whether there is a match. Syntax matching assumes that the syntax being matched was produced by Lean's parser, either via quotation or directly in user code, and uses this to omit some checks. For example, if nothing but a particular keyword can be present in a given position, the check may be omitted.

Syntax matches a quote pattern in the following cases:

Atoms

Keyword atoms (such as `if` or `match`) result in singleton nodes whose kind is `token.` followed by the atom. In many cases, it is not necessary to check for specific atom values because the grammar allows only a single keyword, and no checking will be performed. If the syntax of the term being matched requires the check, then the node kind is compared.

Literals, such as string or numeric literals, are compared via their underlying string representation. The pattern `` `(0x15) `` and the quotation `` `(21) `` do not match.

Nodes

If both the pattern and the value being matched represent `Syntax.node`, there is a match when both have the same syntax kind, the same number of children, and each child pattern matches the corresponding child value.

Identifiers

If both the pattern and the value being matched are identifiers, then their literal `Name` values are compared for equality modulo macro scopes. Identifiers that "look" the same match, and it does not matter if they refer to the same binding. This design choice allows quote pattern matching to be used in contexts that don't have access to a compile-time environment in which names can be compared by reference.

Because quotation pattern matching is based on the node kinds emitted by the parser, quotations that look identical may not match if they come from different syntax categories. If in doubt, including the syntax category in the quotation can help.

20.5.5. Defining Macros

There are two primary ways to define macros: the `macro_rules` command and the `macro` command. The `macro_rules` command associates a macro with existing syntax, while the `macro` command simultaneously defines new syntax and a macro that translates it to existing syntax. The `macro` command can be seen as a generalization of `notation` that allows the expansion to be generated programmatically, rather than simply by substitution.

20.5.5.1. The `macro_rules` Command

syntax

Rule-Based Macros With `macro_rules`

The `macro_rules` command takes a sequence of rewrite rules, specified as syntax pattern matches, and adds each as a macro. The rules are attempted in order, before previously-defined macros, and later macro definitions may add further macro rules.

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
      attrKind macro_rules ((kind := ident))?
        (| `((p:ident|)?Suitable syntax for p) => term)*
```

The patterns in the macros must be quotation patterns. They may match syntax from any syntax category, but a given pattern can only ever match a single syntax kind. If no category or parser is specified for the quotation, then it may match terms or (sequences of) commands, but never both. In case of ambiguity, the term parser is chosen.

Internally, macros are tracked in a table that maps each syntax kind to its macros. The `macro_rules` command may be explicitly annotated with a syntax kind.

If a syntax kind is explicitly provided, the macro definition checks that each quotation pattern has that kind. If the parse result for the quotation was a choice node (that is, if the parse was ambiguous), then the pattern is duplicated once for each alternative with the specified kind. It is an error if none of the alternatives have the specified kind.

If no kind is provided explicitly, then the kind determined by the parser is used for each pattern. The patterns are not required to all have the same syntax kind; macros are defined for each syntax kind used by at least one of the patterns. It is an error if the parse result for a quotation pattern was a choice node (that is, if the parse was ambiguous).

The documentation comment associated with `macro_rules` is displayed to users if the syntax itself has no documentation comment. Otherwise, the documentation comment for the syntax itself is shown.

As with notations and operators, macro rules may be declared `scoped` or `local`. Scoped macros are only active when the current namespace is open, and local macro rules are only active in the current section scope.

▼ *Example: Idiom Brackets*

Idiom brackets are an alternative syntax for working with applicative functors. If the idiom brackets contain a function application, then the function is wrapped in `pure` and applied to each argument using `<*>`. Lean does not support idiom brackets by default, but they can be defined using a macro.

```
syntax (name := idiom) "⟦" (term:arg)+ "⟧" : term

macro_rules
  | `(⟦$f $args*⟧) => do
    let mut out ← `(pure $f)
    for arg in args do
      out ← `($out <*> $arg)
    return out
```

This new syntax can be used immediately.

```
def addFirstThird [Add α] (xs : List α) : Option α :=
  ⟦Add.add xs[0]? xs[2]?⟧

#eval addFirstThird (α := Nat) []
```

```
none
```

```
#eval addFirstThird [1]
```

```
none
```

```
#eval addFirstThird [1,2,3,4]
```

```
some 4
```

▼ *Example: Scoped Macros*

Scoped macro rules are active only in their namespace. When the namespace `ConfusingNumbers` is open, numeric literals will be assigned an incorrect meaning.

```
namespace ConfusingNumbers
```

The following macro recognizes terms that are odd numeric literals, and replaces them with double their value. If it unconditionally replaced them with double their value, then macro expansion would become an infinite loop because the same rule would always match the output.

```
scoped macro_rules
  | `($n:num) => do
    if n.getNat % 2 = 0 then Lean.Macro.throwUnsupported
    let n' := (n.getNat * 2)
    `($(Syntax.mkNumLit (info := n.raw.getHeadInfo) (toString
n')))
```

Once the namespace ends, the macro is no longer used.

```
end ConfusingNumbers
```

Without opening the namespace, numeric literals function in the usual way.

```
#eval (3, 4)
```

```
(3, 4)
```

When the namespace is open, the macro replaces 3 with 6.

```
open ConfusingNumbers

#eval (3, 4)
```

```
(6, 4)
```

It is not typically useful to change the interpretation of numeric or other literals in macros. However, scoped macros can be very useful when adding new rules to extensible tactics such as `trivial` that work well with the contents of the namespaces but should not always be used.

Behind the scenes, a `macro_rules` command generates one macro function for each syntax kind that is matched in its quote patterns. This function has a default case that throws the `unsupportedSyntax` exception, so further macros may be attempted.

A single `macro_rules` command with two rules is not always equivalent to two separate single-match commands. First, the rules in a `macro_rules` are tried from top to bottom, but recently-declared macros are attempted first, so the order would need to be reversed. Additionally, if an earlier rule in the macro throws the `unsupportedSyntax` exception, then the later rules are not tried; if they were instead in separate `macro_rules` commands, then they would be attempted.

▼ *Example: One vs. Two Sets of Macro Rules*

The `arbitrary!` macro is intended to expand to some arbitrarily-determined value of a given type.

```
syntax (name := arbitrary!) "arbitrary! " term:arg : term

macro_rules
  | `(arbitrary! ()) => `(())
  | `(arbitrary! Nat) => `(42)
  | `(arbitrary! ($t1 × $t2)) => `((arbitrary! $t1, arbitrary!
$t2))
  | `(arbitrary! Nat) => `(0)
```

Users may extend it by defining further sets of macro rules, such as this rule for `Empty` that fails:

```
macro_rules
  | `(arbitrary! Empty) => throwUnsupported

#eval arbitrary! (Nat × Nat)
```

```
(42, 42)
```

If all of the macro rules had been defined as individual cases, then the result would have instead used the later case for `Nat`. This is because the rules in a single `macro_rules` command are checked from top to bottom, but more recently-defined `macro_rules` commands take precedence over earlier ones.

```
macro_rules
  | `(arbitrary! ()) =>
    `(())
macro_rules
  | `(arbitrary! Nat) =>
    `(42)
macro_rules
  | `(arbitrary! ($t1 × $t2)) =>
    `((arbitrary! $t1, arbitrary! $t2))
macro_rules
  | `(arbitrary! Nat) =>
    `(0)
macro_rules
  | `(arbitrary! Empty) =>
    throwUnsupported

#eval arbitrary! (Nat × Nat)
```

```
(0, 0)
```

Additionally, if any rule throws the `unsupportedSyntax` exception, no further rules in that command are checked.

```
macro_rules
  | `(arbitrary! (List Nat)) => throwUnsupported
  | `(arbitrary! (List $_)) => `([])

macro_rules
  | `(arbitrary! (Array Nat)) => `(#[42])
macro_rules
  | `(arbitrary! (Array $_)) => throwUnsupported
```

The case for `List Nat` fails to elaborate, because macro expansion did not translate the `arbitrary!` syntax into something supported by the elaborator.

```
#eval arbitrary! (List Nat)
```

```
elaboration function for 'arbitrary!' has not been
implemented
  arbitrary! (List Nat)
```

The case for `Array Nat` succeeds, because the first set of macro rules are attempted after the second throws the exception.

```
#eval arbitrary! (Array Nat)
```

```
#[42]
```

20.5.5.2. The `macro` Command

The `macro` command simultaneously defines a new syntax rule and associates it with a macro. Unlike `notation`, which can define only new term syntax and in which the expansion is a term into which the parameters are to be substituted, the `macro` command may define syntax in any syntax category and it may use arbitrary code in the `MacroM` monad to generate the expansion. Because macros are so much more flexible than notations, Lean cannot automatically generate an unexpander; this means that new syntax implemented via the `macro` command is available for use in *input* to Lean, but Lean's output does not use it without further work.

syntax

Macro Declarations

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
      attrKind macro(:prec)? ((name := ident))? ((priority :=
prio))? macroArg* : ident =>
        macroRhs
```

syntax

Macro Arguments

A macro's arguments are either syntax items (as used in the `syntax` command) or syntax items with attached names.

```
macroArg ::=
    stx

macroArg ::= ...
    | ident:stx
```

In the expansion, the names that are attached to syntax items are bound; they have type `TSyntax` for the appropriate syntax kinds. If the syntax matched by the parser does not have a defined kind (e.g. because the name is applied to a complex specification), then the type is `TSyntax Name.anonymous`.

The documentation comment is associated with the new syntax, and the attribute kind (none, `local`, or `scoped`) governs the visibility of the macro just as it does for notations: `scoped` macros are available in the namespace in which they are defined or in any section scope that opens that namespace, while `local` macros are available only in the local section scope.

Behind the scenes, the `macro` command is itself implemented by a macro that expands it to a `syntax` command and a `macro_rules` command. Any attributes applied to the macro command are applied to the syntax definition, but not to the `macro_rules` command.

20.5.5.3. The Macro Attribute

Macros can be manually added to a syntax kind using the `macro` attribute. This low-level means of specifying macros is typically not useful, except as a result of code generation by macros that themselves generate macro definitions.

attribute

The `macro` Attribute

The `macro` attribute specifies that a function is to be considered a macro for the specified syntax kind.

```
attr ::= ...
    | macro ident
```

▼ *Example: The Macro Attribute*

```
/-- Generate a list based on N syntactic copies of a term -/
syntax (name := rep) "[" num " !!! " term "]" : term

@[macro rep]
```

```
def expandRep : Macro
  | `([ $n:num !!! $e:term]) =>
    let e' := Array.mkArray n.getNat e
    `([$e',*])
  | _ =>
    throwUnsupported
```

Evaluating this new expression demonstrates that the macro is present.

```
#eval [3 !!! "hello"]
```

```
["hello", "hello", "hello"]
```

20.6. Elaborators

While macros allow Lean to be extended by translating new syntax into existing syntax, *elaborators* allow the new syntax to be processed directly. Elaborators have access to everything that Lean itself uses to implement each feature of the language. Defining a new elaborator allows a language extension to be just as powerful as any built-in feature of Lean.

Elaborators come in two varieties:

- *Command elaborators* are used to add new commands to Lean. Commands are implemented as side effects: they may add new constants to the global environment, extend compile-time tables such as the one that tracks instances, they can provide feedback in the form of information, warnings, or errors, and they have full access to the `IO` monad. Command elaborators are associated with the syntax kinds that they can handle.
- *Term elaborators* are used to implement new terms by translating the syntax into Lean's core type theory. They can do everything that command elaborators can do, and they additionally have access to the local context in which the term is being elaborated. Term elaborators can look up bound variables, bind new variables, unify two terms, and much more. A term elaborator must return a value of type `Lean.Expr`, which is the AST of the core type theory.

This section provides an overview and a few examples of elaborators. Because Lean's own elaborator uses the same tools, the source code of the elaborator is a good source of further examples. Just like macros, multiple elaborators may be associated with a syntax kind; they are tried in order, and elaborators may delegate to the next elaborator in the table by throwing the `unsupportedSyntax` exception.

syntax

Elaboration Rules

The `elab_rules` command takes a sequence of elaboration rules, specified as syntax pattern matches, and adds each as an elaborator. The rules are attempted in order, before previously-defined elaborators, and later elaborators may add further options.

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
      attrKind elab_rules ((kind := ident))? (: ident)? (<=
ident)?
        (| `((p:ident|)?Suitable syntax for p ) => term)*
```

Commands, terms, and tactics each maintain a table that maps syntax kinds to elaborators. The syntax category for which the elaborator should be used is specified after the colon, and must be `term`, `command`, or `tactic`. The `<=` binds the provided identifier to the current expected type in the context in which a term is being elaborated; it may only be used for term elaborators, and if present, then `term` is implied as the syntax category.

attribute

Elaborator Attribute

Elaborators can be directly associated with syntax kinds by applying the appropriate attributes. Each takes the name of a syntax kind and associates the definition with the kind.

```
attr ::= ...
    | term_elab (prio
       | ident)

attr ::= ...
    | command_elab (prio
       | ident)

attr ::= ...
    | tactic (prio
       | ident)
```

20.6.1. Command Elaborators

A command elaborator has type `CommandElab`, which is an abbreviation for `Syntax → CommandElabM Unit`. Command elaborators may be implicitly defined using **`elab_rules`**, or explicitly by defining a function and applying the **`command_elab`** attribute.

▼ *Example: Querying the Environment*

A command elaborator can be used to query the environment to discover how many constants have a given name. This example uses `getEnv` from the `MonadEnv` class to get the current environment. `Environment.constants` yields a mapping from names to information about them (e.g. their type and whether they are a definition, inductive type declaration, etc).

`logInfoAt` allows informational output to be associated with syntax from the original program, and a token antiquotation is used to implement the Lean convention that output from interactive commands is associated with their keyword.

```
syntax "#count_constants " ident : command

elab_rules : command
  | `(#count_constants%$tok $x) => do
    let pattern := x.getId
    let env ← getEnv
    let mut count := 0
    for (y, _) in env.constants do
      if pattern.isSuffixOf y then
        count := count + 1
    logInfoAt tok m!"Found {count} instances of '{pattern}'"

def interestingName := 55
def NS.interestingName := "Another one"

#count_constants interestingName
```

```
Found 2 instances of 'interestingName'
```

20.6.2. Term Elaborators

A term elaborator has type `TermElab`, which is an abbreviation for `Syntax → Option Expr → TermElabM Expr`. The optional `Expr` parameter is the type expected for the term being elaborated, which is none if no type is yet known. Like command elaborators, term elaborators may be implicitly defined using `elab_rules`, or explicitly by defining a function and applying the **`term_elab`** attribute.

▼ *Example: Avoiding a Type*

This examples demonstrates an elaborator for syntax that is the opposite of a type ascription. The provided term may have any type *other* than the one indicated, and metavariables are solved pessimistically. In this example, `elabType` invokes the term elaborator and then ensures that the resulting term is a type. `Meta.inferType` infers a type for a term, and `Meta.isDefEq` attempts to make two terms definitionally equal by unification, returning `true` if it succeeds.

```
syntax (name := notType) "(" term  " !: " term ")" : term

@[term_elab notType]
def elabNotType : TermElab := fun stx _ => do
  let `(($tm:term !: $ty:term)) := stx
    | throwUnsupportedSyntax
  let unexpected ← elabType ty
  let e ← elabTerm tm none
```

```
  let eTy ← Meta.inferType e
  if (← Meta.isDefEq eTy unexpected) then
    throwErrorAt tm m!"Got unwanted type {eTy}"
  else pure e
```

If the type position does not contain a type, then `elabType` throws an error:

```
#eval ([1, 2, 3] !: "not a type")
```

```
type expected, got
  ("not a type" : String)
```

If the term's type is definitely not equal to the provided type, then elaboration succeeds:

```
#eval ([1, 2, 3] !: String)
```

```
[1, 2, 3]
```

If the types match, an error is thrown:

```
#eval (5 !: Nat)
```

```
Got unwanted type Nat
```

The type equality check may fill in missing information, so **`sorry`** (which may have any type) is also rejected:

```
#eval (sorry !: String)
```

```
Got unwanted type String
```

▼ *Example: Using Any Local Variable*

Term elaborators have access to the expected type and to the local context. This can be used to create a term analogue of the **`assumption`** tactic.

The first step is to access the local context using `getLocalHyps`. It returns the context with the outermost bindings on the left, so it is traversed in reverse order. For each local assumption, a type is inferred with `Meta.inferType`. If it can be equal to the expected type, then the assumption is returned; if no assumption is suitable, then an error is produced.

```
syntax "anything!" : term

elab_rules <= expected
  | `(anything!) => do
    let hyps ← getLocalHyps
    for h in hyps.reverse do
      let t ← Meta.inferType h
      if (← Meta.isDefEq t expected) then return h

    throwError m!"No assumption in {hyps} has type {expected}"
```

The new syntax finds the function's bound variable:

```
#eval (fun (n : Nat) => 2 + anything!) 5
```

```
7
```

It chooses the most recent suitable variable, as desired:

```
#eval
  let x := "x"
  let y := "y"
  "It was " ++ y
```

```
"It was y"
```

When no assumption is suitable, it returns an error that describes the attempt:

```
#eval
  let x := Nat.zero
  let y := "hello"
  fun (f : Nat → Nat) =>
    (anything! : Int → Int)
```

```
No assumption in [x, y, f] has type Int → Int
```

Because it uses unification, the natural number literal is chosen here, because numeric literals may have any type with an `OfNat` instance. Unfortunately, there is no `OfNat` instance for functions, so instance synthesis later fails.

```
#eval
  let x := 5
  let y := "hello"
  (anything! : Int → Int)
```

```
failed to synthesize
  OfNat (Int → Int) 5
numerals are polymorphic in Lean, but the numeral `5` cannot
be used in a context where the expected type is
  Int → Int
due to the absence of the instance above

Hint: Additional diagnostic information may be available
using the `set_option diagnostics true` command.
```

20.6.3. Custom Tactics

Custom tactics are described in the section on tactics.

20.7. Extending Lean's Output

Extending Lean with new syntax and implementing the new syntax with macros and elaborators allows users to express ideas to Lean more conveniently. However, Lean is an *interactive* theorem prover: it is also important that the feedback it provides can be readily understood. Syntax extensions should be used in *output* as well as in *input*.

There are two primary mechanisms for instructing Lean to use a syntax extension in its output:

Unexpanders
: Unexpanders are the inverse of macros. Macros implement new syntax in terms of the old syntax by translation, *expanding* the new feature into encodings in pre-existing features. Like macros, *unexpanders* translate `Syntax` into `Syntax`; unlike macros, they transform the encodings into the new extensions.

Delaborators
: Delaborators are the inverse of elaborators. While elaborators translate `Syntax` into the core type theory's `Expr`, *delaborators* translate `Expr`s into `Syntax`.

Before an `Expr` is displayed, it is first delaborated and then unexpanded. The delaborator tracks the position in the original `Expr` that its output originated from; this position is encoded in the resulting syntax's `SourceInfo`. Just as macro expansion automatically annotates the resulting syntax with synthetic source information that correspond to the original syntax's position, the unexpansion mechanism preserves the resulting syntax's association with the underlying `Expr`. This association enables Lean's interactive features that provide information about the resulting syntax when it is shown in proof states and diagnostics.

20.7.1. Unexpanders

Just as macros are registered in a table that maps syntax kinds to macro implementations, unexpanders are registered in a table that maps the names of constants to unexpander implementations. Before Lean displays syntax to users, it attempts to rewrite each application of a constant in the syntax according to this table. Occurrences of the context that are not applications are treated as applications with zero arguments.

Unexpansion proceeds from the inside out. The unexpander is passed the syntax of the application, with implicit arguments hidden, after the arguments have been unexpanded. If the option `pp.explicit` is `true` or `pp.notation` is `false`, then unexpanders are not used.

An unexpander has type `Lean.PrettyPrinter.Unexpander`, which is an abbreviation for `Syntax → Lean.PrettyPrinter.UnexpandM Syntax`. In the remainder of this section, the names `Unexpander` and `UnexpandM` are used unqualified. `UnexpandM` is a monad that supports quotation and failure via its `MonadQuotation` and `MonadExcept Unit` instances.

An unexpander should either return unexpanded syntax or fail using `throw ()`. If the unexpander succeeds, then the resulting syntax is unexpanded again; if it fails, then the next unexpander is tried. When no unexpander succeeds for the syntax, its child nodes are unexpanded until all opportunities

for unexpansion are exhausted.

def

```
Lean.PrettyPrinter.Unexpander : Type
```

Function that tries to reverse macro expansions as a post-processing step of delaboration. While less general than an arbitrary delaborator, it can be declared without importing `Lean`. Used by the `[app_unexpander]` attribute.

def

```
Lean.PrettyPrinter.UnexpandM (α : Type) : Type
```

The unexpander monad, essentially `Syntax → Option α`. The `Syntax` is the `ref`, and it has the possibility of failure without an error message.

An unexpander for a constant is registered by applying the **`app_unexpander`** attribute. Custom operators and notations automatically create unexpanders for the syntax that they introduce.

attribute

Unexpander Registration

```
attr ::= ...
    | app_unexpander ident
```

Registers an unexpander of type `Unexpander` for applications of a constant.

▼ *Example: Custom Unit Type*

A type equivalent to `Unit`, but with its own notation, can be defined as a zero-field structure and a macro:

```
structure Solo where
  mk ::

syntax "‹" "›" : term

macro_rules
  | `(term|‹›) => ``(Solo.mk)
```

While the new notation can be used to write theorem statements, it does not appear in proof states. For example, when proving that all values of type `Solo` are equal to `‹›`, the initial proof state is:

```
v : Solo
```

```
⊢ v = { }
```

This proof state shows the constructor using structure instance syntax. An unexpander can be used to override this choice. Because `Solo.mk` cannot be applied to any arguments, the unexpander is free to ignore the syntax, which will always be `` `(Solo.mk) ``.

```
@[app_unexpander Solo.mk]
def unexpandSolo : Lean.PrettyPrinter.Unexpander
  | _ => `(‹›)
```

With this unexpander, the initial state of the proof now renders with the correct syntax:

```
v : Solo
⊢ v = ‹›
```

▼ *Example: Unexpansion and Arguments*

A `ListCursor` represents a position in a `List`. `ListCursor.before` contains the reversed list of elements prior to the position, and `ListCursor.after` contains the elements after the position.

```
structure ListCursor (α) where
  before : List α
  after : List α
deriving Repr
```

List cursors can be moved to the left or to the right:

```
def ListCursor.left : ListCursor α → Option (ListCursor α)
  | ⟨[], _⟩ => none
  | ⟨l :: ls, rs⟩ => some ⟨ls, l :: rs⟩

def ListCursor.right : ListCursor α → Option (ListCursor α)
  | ⟨_, []⟩ => none
  | ⟨ls, r :: rs⟩ => some ⟨r :: ls, rs⟩
```

They can also be moved all the way to the left or all the way to the right:

```
def ListCursor.rewind : ListCursor α → ListCursor α
  | xs@⟨[], _⟩ => xs
  | ⟨l :: ls, rs⟩ => rewind ⟨ls, l :: rs⟩
termination_by xs => xs.before

def ListCursor.fastForward : ListCursor α → ListCursor α
  | xs@⟨_, []⟩ => xs
  | ⟨ls, r :: rs⟩ => fastForward ⟨r :: ls, rs⟩
termination_by xs => xs.after
```

However, the need to reverse the list of previous elements can make list cursors difficult to understand. A cursor can be given a notation in which a flag (🚩) marks the cursor's location in a list:

```
syntax "[" term,* " 🚩 " term,* "]": term
macro_rules
  | `([$ls,* 🚩 $rs,*]) =>
    ``(ListCursor.mk [$[$((ls : Array Lean.Term).reverse)],*]
[$rs,*])
```

In the macro, the sequences of elements have type `Syntax.TSepArray `term ","`. The type annotation as `Array Lean.Term` causes a coercion to fire so that `Array.reverse` can be applied, and a similar coercion reinserts the separating commas. These coercions are described in the section on typed syntax.

While the syntax works, it is not used in Lean's output:

```
#check [1, 2, 3 🚩 4, 5]
```

```
{ before := [3, 2, 1], after := [4, 5] } : ListCursor Nat
```

An unexpander solves this problem. The unexpander relies on the built-in unexpanders for list literals already having rewritten the two lists:

```
@[app_unexpander ListCursor.mk]
def unexpandListCursor : Lean.PrettyPrinter.Unexpander
  | `($_ [$ls,*] [$rs,*]) =>
    `([$((ls : Array Lean.Term).reverse),* 🚩 $(rs),*])
  | _ => throw ()

#check [1, 2, 3 🚩 4, 5]
```

```
[1, 2, 3 🚩 4, 5] : ListCursor Nat
```

```
#reduce [1, 2, 3 🚩 4, 5].right
```

```
some [1, 2, 3, 4 🚩 5]
```

```
#reduce [1, 2, 3 🚩 4, 5].left >>= (·.left)
```

```
some [1 🚩 2, 3, 4, 5]
```

20.7.2. Delaborators

A delaborator is function of type `Lean.PrettyPrinter.Delaborator.Delab`, which is an abbreviation for `Lean.PrettyPrinter.Delaborator.DelabM Term`. Unlike unexpanders, delaborators are not implemented as functions. This is to make it easier to implement them correctly: the monad `DelabM` tracks the current position in the expression that's being delaborated so the delaboration mechanism can annotate the resulting syntax.

Delaborators are registered with the **delab** attribute. An internal table maps the names of the constructors of `Expr` (without namespaces) to delaborators. Additionally, the name `app.`c is consulted to find delaborators for applications of the constant c, and the name `mdata.`k is consulted to find delaborators for `Expr.mdata` constructors with a single key k in their metadata.

attribute

Delaborator Registration

The **delab** attribute registers a delaborator for the indicated constructor or metadata key of `Expr`.

```
attr ::= ...
    | delab ident
```

The **app_delab** attribute registers a delaborator for applications of the indicated constant after resolving it in the current scope.

```
attr ::= ...
    | app_delab ident
```

The monad `DelabM` is a reader monad that includes access to the current position in the `Expr`. Recursive delaboration is performed by adjusting the reader monad's tracked position, rather than by explicitly passing a subexpression to another function. The most important functions for working with subexpressions in delaborators are in the namespace `Lean.PrettyPrinter.Delaborator.SubExp`:

- `getExpr` retrieves the current expression for analysis.
- `withAppFn` adjusts the current position to be that of the function in an application.
- `withAppArg` adjusts the current position to be that of the argument in an application
- `withAppFnArgs` decomposes the current expression into a non-application function and its arguments, focusing on each.
- `withBindingBody` descends into the body of a function or function type.

Further functions to descend into the remaining constructors of `Expr` are available.

21. Run-Time Code

Compiled Lean code uses services provided by the Lean runtime. The runtime contains efficient, low-level primitives that bridge the gap between the Lean language and the supported platforms. These services include:

Memory management

Lean does not require programmers to manually manage memory. Space is allocated when needed to store a value, and values that can no longer be reached (and are thus irrelevant) are deallocated. In particular, Lean uses reference counting, where each allocated object maintains a count of incoming references. The compiler emits calls to memory management routines that allocate memory and modify reference counts, and these routines are provided by the runtime, along with the data structures that represent Lean values in compiled code.

Multiple Threads

The `Task` API provides the ability to write parallel and concurrent code. The runtime is responsible for scheduling Lean tasks across operating-system threads.

Primitive operators

Many built-in types, including `Nat`, `Array`, `String`, and fixed-width integers, have special representations for reasons of efficiency. The runtime provides implementations of these types' primitive operators that take advantage of these optimized representations.

There are many primitive operators. They are described in their respective sections under Basic Types.

21.1. Boxing

Lean values may be represented at runtime in two ways:

- *Boxed* values may be pointers to heap values or require shifting and masking.
- *Unboxed* values are immediately available.

Boxed values are either a pointer to an object, in which case the lowest-order bit is 0, or an immediate value, in which case the lowest-order bit is 1 and the value is found by shifting the representation to the right by one bit.

Types with an unboxed representation, such as `UInt8` and enum inductive types, are represented as the corresponding C types in contexts where the compiler can be sure that the value has said type. In some contexts, such as generic container types like `Array`, otherwise-unboxed values must be boxed prior to storage. In other words, `Bool.not` is called with and returns unboxed `uint8_t` values because the enum inductive type `Bool` has an unboxed representation, but the individual `Bool` values in an `Array Bool` are boxed. A field of type `Bool` in an inductive type's constructor is represented unboxed, while `Bool`s stored in polymorphic fields that are instantiated as `Bool` are boxed.

21.2. Reference Counting

Lean uses *reference counting* for memory management. Each allocated object maintains a count of how many other objects refer to it. When a new reference is added, the count is incremented, and when a reference is dropped, the count is decremented. When a reference count reaches zero, the

object is no longer reachable and can play no part in the further execution of the program. It is deallocated and all of its references to other objects are dropped, which may trigger further deallocations.

Reference counting provides a number of benefits:

Re-Use of Memory

If an object's reference count drops to zero just as another of the same size is to be allocated, then the original object's memory can be safely re-used for the new object. As a result, many common data-structure traversals (such as `List.map`) do not need to allocate memory when there is exactly one reference to the data structure to be traversed.

Opportunistic In-Place Updates

Primitive types, such as strings and arrays, may provide operations that copy shared data but modify unshared data in-place. As long as they hold the only reference to the value being modified, many operations on these primitive types will modify it rather than copy it. This can lead to substantial performance benefits. Carefully-written `Array` code avoids the performance overhead of immutable data structures while maintaining the ease of reasoning provided by pure functions.

Predictability

Reference counts are decremented at predictable times. As a result, reference-counted objects can be used to manage other resources, such as file handles. In Lean, a `Handle` does not need to be explicitly closed because it is closed immediately when it is no longer accessible.

Simpler FFI

Objects managed with reference counting don't need to be relocated as part of reclaiming unused memory. This greatly simplifies interaction with code written in other languages, such as C.

The traditional drawbacks of reference counting include the performance overhead due to updating reference counts along with the inability to recognize and deallocate cyclic data. The former drawback is minimized by an analysis based on *borrowing* that allows many reference count updates to be elided. Nevertheless, multi-threaded code requires that reference count updates are synchronized between threads, which also imposes a substantial overhead. To reduce this overhead, Lean values are partitioned into those which are reachable from multiple threads and those which are not. Single-threaded reference counts can be updated much faster than multi-threaded reference counts, and many values are accessed only on a single thread. Together, these techniques greatly reduce the performance overhead of reference counting. Because Lean cannot create cyclic data, no technique is needed to detect it. Ullrich and de Moura (2019) (i) Sebastian Ullrich and Leonardo de Moura, 2019. "Counting Immutable Beans: Reference Counting Optimized for Purely Functional Programming". In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL 2019).* provide more details on the implementation of reference counting in Lean.

21.2.1. Observing Uniqueness

Ensuring that arrays and strings are uniquely referenced is key to writing fast code in Lean. The primitive `dbgTraceIfShared` can be used to check whether a data structure is aliased. When called, it returns its argument unchanged, printing the provided trace message if the argument's reference count is greater than one.

def

```
dbgTraceIfShared.{u} {α : Type u} (s : String) (a : α) : α
```

Display the given message if *a* is shared, that is, RC(a) > 1

Due to the specifics of how `#eval` is implemented, using `dbgTraceIfShared` with `#eval` can be misleading. Instead, it should be used in code that's explicitly compiled and run.

▼ *Example: Observing Uniqueness*

This program reads a line of input from the user, printing it after replacing its first character with a space. Replacing characters in a string uses an in-place update if the string is not shared and the characters are both contained in the 7-bit ASCII subset of Unicode. The `dbgTraceIfShared` call does nothing, indicating that the string will indeed be updated in place rather than copied.

```
def process (str : String) : IO Unit := do
  IO.println ((dbgTraceIfShared "String update" str).set 0 ' ')

def main : IO Unit := do
  let line := (← (← IO.getStdin).getLine).trim
  process line
```

When run with this input:

```
stdin
  1 Here is input.
```

the program emits:

```
stdout
  1  ere is input.
```

with an empty standard error output:

```
stderr
    <empty>
```

This version of the program retains a reference to the original string, which necessitates copying the string in the call to `String.set`. This fact is visible in its standard error output.

```
def process (str : String) : IO Unit := do
  IO.println ((dbgTraceIfShared "String update" str).set 0 ' ')

def main : IO Unit := do
  let line := (← (← IO.getStdin).getLine).trim
  process line
  IO.println "Original input:"
  IO.println line
```

When run with this input:

```
stdin
  1 Here is input.
```

the program emits:

```
stdout
  1  ere is input.
  2 Original input:
  3 Here is input.
```

In its standard error, the message passed to `dbgTraceIfShared` is visible.

```
stderr
  1 shared RC String update
```

21.2.2. Compiler IR

The compiler option `trace.compiler.ir.result` can be used to inspect the compiler's intermediate representation (IR) for a function. In this intermediate representation, reference counting, allocation, and reuse are explicit:

- The `isShared` operator checks whether a reference count is `1`.
- `ctor_`n allocates the nth constructor of a type.
- `proj_`n retrieves the nth field from a constructor value.
- `set` x`[`n`]` mutates the nth field of the constructor in x.
- `ret` x returns the value in x.

The specifics of reference count manipulations can depend on the results of optimization passes such as inlining. While the vast majority of Lean code doesn't require this kind of attention to achieve good performance, knowing how to diagnose unique reference issues can be very important when

writing performance-critical code.

option

`trace.compiler.ir.result`

Default value: `false`

(trace) enable/disable tracing for the given module and submodules

▼ *Example: Reference Counts in IR*

Compiler IR can be used to observe when reference counts are incremented, which can help diagnose situations when a value is expected to have a unique incoming reference, but is in fact shared. Here, `process` and `process'` each take a string as a parameter and modify it with `String.set`, returning a pair of strings. While `process` returns a constant string as the second element of the pair, `process'` returns the original string.

```
set_option trace.compiler.ir.result true

def process (str : String) : String × String :=
  (str.set 0 ' ', "")

def process' (str : String) : String × String:=
  (str.set 0 ' ', str)
```

The IR for `process` includes no `inc` or `dec` instructions. If the incoming string `x_1` is a unique reference, then it is still a unique reference when passed to `String.set`, which can then use in-place modification:

```
[Compiler.IR] [result]
    def process (x_1 : obj) : obj :=
      let x_2 : obj := 0;
      let x_3 : u32 := 32;
      let x_4 : obj := String.set x_1 x_2 x_3;
      let x_5 : obj :=
block._closed_160._@.Manual.Runtime._hyg.3359;
      let x_6 : obj := ctor_0[Prod.mk] x_4 x_5;
      ret x_6
```

The IR for `process'`, on the other hand, increments the reference count of the string just before calling `String.set`. Thus, the modified string `x_4` is a copy, regardless of whether the original reference to `x_1` is unique:

```
[Compiler.IR] [result]
    def process' (x_1 : obj) : obj :=
      let x_2 : obj := 0;
      let x_3 : u32 := 32;
      inc x_1;
      let x_4 : obj := String.set x_1 x_2 x_3;
```

```
      let x_5 : obj := ctor_0[Prod.mk] x_4 x_1;
      ret x_5
```

▼ *Example: Memory Re-Use in IR*

The function `discardElems` is a simplified version of `List.map` that replaces every element in a list with `()`. Inspecting its intermediate representation demonstrates that it will re-use the list's memory when its reference is unique.

```
set_option trace.compiler.ir.result true

def discardElems : List α → List Unit
  | [] => []
  | x :: xs => () :: discardElems xs
```

This emits the following IR:

```
[Compiler.IR] [result]
    def discardElems._redArg (x_1 : obj) : obj :=
      case x_1 : obj of
      List.nil →
        let x_2 : obj := ctor_0[List.nil];
        ret x_2
      List.cons →
        let x_3 : u8 := isShared x_1;
        case x_3 : u8 of
        Bool.false →
          let x_4 : obj := proj[1] x_1;
          let x_5 : obj := proj[0] x_1;
          dec x_5;
          let x_6 : obj := ctor_0[PUnit.unit];
          let x_7 : obj := discardElems._redArg x_4;
          set x_1[1] := x_7;
          set x_1[0] := x_6;
          ret x_1
        Bool.true →
          let x_8 : obj := proj[1] x_1;
          inc x_8;
          dec x_1;
          let x_9 : obj := ctor_0[PUnit.unit];
          let x_10 : obj := discardElems._redArg x_8;
          let x_11 : obj := ctor_1[List.cons] x_9 x_10;
          ret x_11
    def discardElems (x_1 : ▪) (x_2 : obj) : obj :=
      let x_3 : obj := discardElems._redArg x_2;
      ret x_3
```

In the IR, the `List.cons` case explicitly checks whether the argument value is shared (i.e. whether it's reference count is greater than one). If the reference is unique, the reference count of the discarded list element `x_5` is decremented and the constructor value is reused. If it is

shared, a new `List.cons` is allocated in `x_11` for the result.

21.3. Multi-Threaded Execution

Lean includes primitives for parallel and concurrent programs, described using tasks. The Lean runtime system includes a task manager that assigns hardware resources to tasks. Along with the API for defining tasks, this is described in detail in the section on multi-threaded programs.

21.4. Foreign Function Interface

The current interface was designed for internal use in Lean and should be considered unstable. It will be refined and extended in the future.

Lean offers efficient interoperability with any language that supports the C ABI. This support is, however, currently limited to transferring Lean data types; in particular, it is not yet possible to pass or return compound data structures such as C **`struct`**s by value from or to Lean.

There are two primary attributes for interoperating with other languages:

- `@[export sym] def leanSym : ...`

attribute

External Symbols

```
attr ::= ...
    | extern str
```

Binds a Lean declaration to the specified external symbol.

attribute

Exported Symbols

```
attr ::= ...
    | export ident
```

Exports a Lean constant with the unmangled symbol name `sym`.

For simple examples of how to call foreign code from Lean and vice versa, see the FFI and reverse FFI examples in the Lean source repository.

21.4.1. The Lean ABI

The Lean *Application Binary Interface* (ABI) describes how the signature of a Lean declaration is encoded in the platform-native calling convention. It is based on the standard C ABI and calling convention of the target platform. Lean declarations can be marked for interaction with foreign functions using either the attribute **`extern "sym"`**, which causes compiled code to use the C declaration `sym` as the implementation, or the attribute **`export sym`**, which makes the declaration available as `sym` to C.

In both cases, the C declaration's type is derived from the Lean type of the declaration with the attribute. Let $\alpha_1 \to \ldots \to \alpha_n \to \beta$ be the declaration's normalized type. If n is 0, the corresponding C declaration is

```
extern s sym;
```

where `s` is the C translation of β as specified in the next section. In the case of a definition marked **`extern`**, the symbol's value is only guaranteed to be initialized after calling the Lean module's initializer or that of an importing module. The section on initialization describes initializers in greater detail.

If n is greater than 0, the corresponding C declaration is

```
s sym(t_1, ..., t_n);
```

where the parameter types t_i are the C translations of the types α_i. In the case of **`extern`**, all irrelevant types are removed first.

21.4.1.1. Translating Types from Lean to C

In the ABI, Lean types are translated to C types as follows:

- The integer types `UInt8`, …, `UInt64`, `USize` are represented by the C types **`uint8_t`**, ..., **`uint64_t`**, **`size_t`**, respectively. If their run-time representation requires boxing, then they are unboxed at the FFI boundary.
- `Char` is represented by **`uint32_t`**.
- `Float` is represented by **`double`**.
- `Nat` and `Int` are represented by **`lean_object *`**. Their runtime values is either a pointer to an opaque bignum object or, if the lowest bit of the "pointer" is 1 (`lean_is_scalar`), an encoded natural number or integer (`lean_box`/`lean_unbox`).
- A universe `Sort u`, type constructor `... → Sort u`, or proposition `p :Prop` is irrelevant and is either statically erased (see above) or represented as a **`lean_object *`** with the runtime value `lean_box(0)`
- The ABI for other inductive types that don't have special compiler support depends on the specifics of the type. It is the same as the run-time representation of these types. Its runtime value is either a pointer to an object of a subtype of **`lean_object`** (see the "Inductive types"

section below) or it is the value `lean_box(cidx)` for the `cidx`th constructor of an inductive type if this constructor does not have any relevant parameters.

▼ *Example: `Unit` in the ABI*

The runtime value of *u* : `Unit` is always `lean_box(0)`.

21.4.1.2. Borrowing

By default, all **`lean_object *`** parameters of an **`extern`** function are considered *owned*. The external code is passed a "virtual RC token" and is responsible for passing this token along to another consuming function (exactly once) or freeing it via `lean_dec`. To reduce reference counting overhead, parameters can be marked as *borrowed* by prefixing their type with `@&`. Borrowed objects must only be passed to other non-consuming functions (arbitrarily often) or converted to owned values using `lean_inc`. In `lean.h`, the **`lean_object *`** aliases **`lean_obj_arg`** and **`b_lean_obj_arg`** are used to mark this difference on the C side. Return values and `@[export]` parameters are always owned at the moment.

syntax

Borrowed Parameters

```
term ::= ...
    | @& term
```

Parameters may be marked as borrowed by prefixing their types with `@&`.

21.4.2. Initialization

When including Lean code in a larger program, modules must be *initialized* before accessing any of their declarations. Module initialization entails:

- initialization of all "constant definitions" (nullary functions), including closed terms lifted out of other functions,
- execution of all code marked with the **`init`** attribute, and
- execution of all code marked with the **`builtin_init`** attribute, if the `builtin` parameter of the module initializer has been set.

The module initializer is automatically run with the `builtin` flag for executables compiled from Lean code and for "plugins" loaded with `lean --plugin`. For all other modules imported by `lean`, the initializer is run without `builtin`. In other words, **`init`** functions are run if and only if their module is imported, regardless of whether they have native code available, while **`builtin_init`** functions are only run for native executable or plugins, regardless of whether their

module is imported. The Lean compiler uses built-in initializers for purposes such as registering basic parsers that should be available even without importing their module, which is necessary for bootstrapping.

The initializer for module `A.B` is called `initialize_A_B` and will automatically initialize any imported modules. Module initializers are idempotent when run with the same `builtin` flag, but not thread-safe. Together with initialization of the Lean runtime, code like the following should be run exactly once before accessing any Lean declarations:

```
void lean_initialize_runtime_module();
void lean_initialize();
lean_object * initialize_A_B(uint8_t builtin, lean_object *);
lean_object * initialize_C(uint8_t builtin, lean_object *);
...

lean_initialize_runtime_module();
// Necessary (and replaces `lean_initialize_runtime_module`) for code
that
// (indirectly) accesses the `Lean` package:
//lean_initialize();

lean_object * res;
// use same default as for Lean executables
uint8_t builtin = 1;
res = initialize_A_B(builtin, lean_io_mk_world());
if (lean_io_result_is_ok(res)) {
    lean_dec_ref(res);
} else {
    lean_io_result_show_error(res);
    lean_dec(res);
    // do not access Lean declarations if initialization failed
    return ...;
}
res = initialize_C(builtin, lean_io_mk_world());
if (lean_io_result_is_ok(res)) {
...

// Necessary for code that (indirectly) uses `Task`
//lean_init_task_manager();
lean_io_mark_end_initialization();
```

In addition, any other thread not spawned by the Lean runtime itself must be initialized for Lean use by calling

```
void lean_initialize_thread();
```

and should be finalized in order to free all thread-local resources by calling

```
void lean_finalize_thread();
```

21.4.3. `@[extern]` in the Interpreter

The Lean interpreter can run Lean declarations for which symbols are available in loaded shared libraries, which includes declarations that are marked **`extern`**. To run this code (e.g. with `#eval`), the following steps are necessary:

1. The module containing the declaration and its dependencies must be compiled into a shared library
2. This shared library to should be provided to `lean --load-dynlib=` to run code that imports the module.

It is not sufficient to load the foreign library containing the external symbol because the interpreter depends on code that is emitted for each **`extern`** declaration. Thus it is not possible to interpret an **`extern`** declaration in the same file. The Lean source repository contains an example of this usage in `tests/compiler/foreign`.

22. Build Tools and Distribution

The Lean *toolchain* is the collection of command-line tools that are used to check proofs and compile programs in collections of Lean files. Toolchains are managed by `elan`, which installs toolchains as needed. Lean toolchains are designed to be self-contained, and most command-line users will never need to explicitly invoke any other than `lake` and `elan`. The contain the following tools:

`lean`
: The Lean compiler, used to elaborate and compile a Lean source file.

`lake`
: The Lean build tool, used to invoke incrementally invoke `lean` and other tools while tracking dependencies.

`leanc`
: The C compiler that ships with Lean, which is a version of Clang.

`leanmake`
: An implementation of the `make` build tool, used for compiling C dependencies.

`leanchecker`
: A tool that replays elaboration results from `.olean` files through the Lean kernel, providing additional assurance that all terms were properly checked.

In addition to these build tools, toolchains contain files that are needed to build Lean code. This includes source code, `.olean` files, compiled libraries, C header files, and the compiled Lean run-time system. They also include external proof automation tools that are used by tactics included with Lean, such as `cadical` for **`bv_decide`**.

22.1. Lake

Lake is the standard Lean build tool. It is responsible for:

- Configuring builds and building Lean code
- Fetching and building external dependencies
- Integrating with Reservoir, the Lean package server
- Running tests, linters, and other development workflows

Lake is extensible. It provides a rich API that can be used to define incremental build tasks for software artifacts that are not written in Lean, to automate administrative tasks, and to integrate with external workflows. For build configurations that do not need these features, Lake provides a declarative configuration language that can be written either in TOML or as a Lean file.

This section describes Lake's command-line interface, configuration files, and internal API. All three share a set of concepts and terminology.

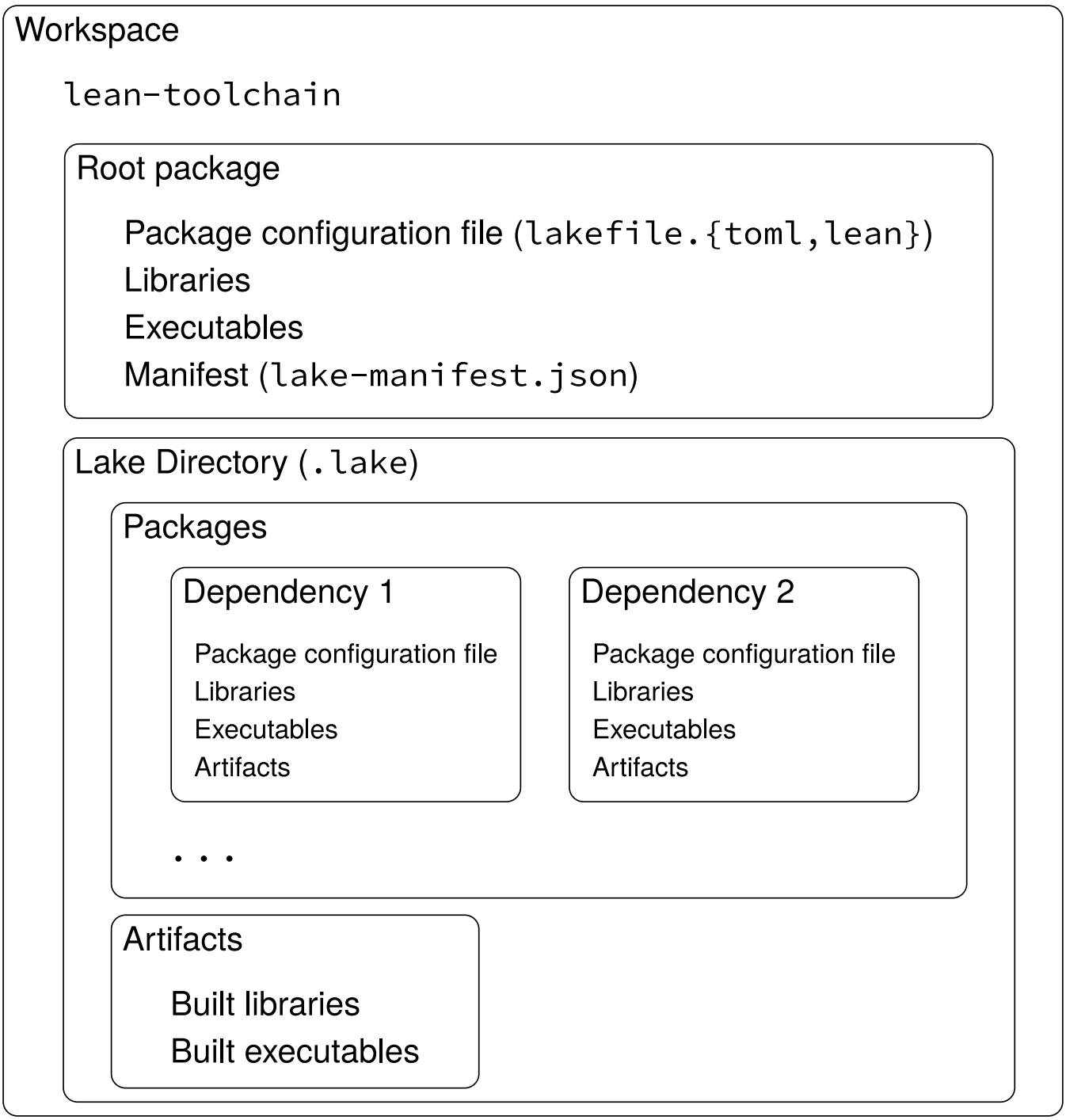
22.1.1. Concepts and Terminology

A *package* is the basic unit of Lean code distribution. A single package may contain multiple libraries or executable programs. A package consist of a directory that contains a package configuration file together with source code. Packages may *require* other packages, in which case those packages' code (more specifically, their targets) are made available. The *direct dependencies* of a package are those that it requires, and the *transitive dependencies* are the direct dependencies of a package together with their transitive dependencies. Packages may either be obtained from Reservoir, the Lean package repository, or from a manually-specified location. *Git dependencies* are specified by a Git repository URL along with a revision (branch, tag, or hash) and must be cloned locally prior to build, while local *path dependencies* are specified by a path relative to the package's directory.

A *workspace* is a directory on disk that contains a working copy of a package's source code and the source code of all transitive dependencies that are not specified as local paths. The package for which the workspace was created is the *root package*. The workspace also contains any built artifacts for the package, enabling incremental builds. Dependencies and artifacts do not need to be present for a directory to be considered a workspace; commands such as `lake update` and `lake build` produce them if they are missing. Lake is typically used in a workspace. (i) `lake init` and `lake new`, which create workspaces, are exceptions. Workspaces typically have the following layout:

- `lean-toolchain` - The toolchain file.
- `lakefile.toml` or `lakefile.lean` - The package configuration file for the root package.
- `lake-manifest.json` - The root package's manifest.
- `.lake/` - Intermediate state managed by Lake, such as built artifacts and dependency source code.
    - `.lake/lakefile.olean` - The root package's configuration, cached.

- `.lake/packages/` - The workspace's *package directory*, which contains copies of all non-local transitive dependencies of the root package, with their built artifacts in their own `.lake` directories.
- `.lake/build/` - The *build directory*, which contains built artifacts for the root package:
  - `.lake/build/bin` - The package's *binary directory*, which contains built executables.
  - `.lake/build/lib` - The package's *library directory*, which contains built libraries and `.olean` files.
  - `.lake/build/ir` - The package's intermediate result directory, which contains generated intermediate artifacts, primarily C code.



Workspace Layout

A *package configuration* file specifies the dependencies, settings, and targets of a package. Packages can specify configuration options that apply to all their contained targets. They can be written in two formats:

- The TOML format (`lakefile.toml`) is used for fully declarative package configurations.
- The Lean format (`lakefile.lean`) additionally supports the use of Lean code to configure the package in ways not supported by the declarative options.

A *manifest* tracks the specific versions of other packages that are used in a package. Together, a manifest and a package configuration file specify a unique set of transitive dependencies for the package. Before building, Lake synchronizes the local copy of each dependency with the version specified in the manifest. If no manifest is available, Lake fetches the latest matching versions of each dependency and creates a manifest. It is an error if the package names listed in the manifest do not match those used by the package; the manifest must be updated using `lake update` prior to building. Manifests should be considered part of the package's code and should normally be checked into source control.

A *target* represents an output that can be requested by a user. A persistent build output, such as object code, an executable binary, or an `.olean` file, is called an *artifact*. In the process of producing an artifact, Lake may need to produce further artifacts; for example, compiling a Lean program into an executable requires that it and its dependencies be compiled to object files, which are themselves produced from C source files, which result from elaborating Lean sourcefiles and producing `.olean` files. Each link in this chain is a target, and Lake arranges for each to be built in turn. At the start of the chain are the *initial targets*:

- *Packages* are units of Lean code that are distributed as a unit.
- *Libraries* are collections of Lean modules, organized hierarchically under one or more *module roots*.
- *Executables* consist of a *single* module that defines `main`.
- *External libraries* are non-Lean **static** libraries that will be linked to the binaries of the package and its dependents, including both their shared libraries and executables.
- *Custom targets* contain arbitrary code to run a build, written using Lake's internal API.

In addition to their Lean code, packages, libraries, and executables contain configuration settings that affect subsequent build steps. Packages may specify a set of *default targets*. Default targets are the initial targets in the package that are to be built in contexts where a package is specified but specific targets are not.

The *log* contains information produced during a build. Logs are saved so they can be replayed during incremental builds. Messages in the log have four levels, ordered by severity:

1. *Trace messages* contain internal build details that are often specific to the machine on which the build is running, including the specific invocations of Lean and other tools that are passed to the shell.
2. *Informational messages* contain general informational output that is not expected to indicate a problem with the code, such as the results of a `#eval` command.

3. *Warnings* indicate potential problems, such as unused variable bindings.

4. *Errors* explain why parsing and elaboration could not complete.

By default, trace messages are hidden and the others are shown. The threshold can be adjusted using the `--log-level` option, the `--verbose` flag, or the `--quiet` flag.

22.1.1.1. Builds

Producing a desired artifact, such as a `.olean` file or an executable binary, is called a *build*. Builds are triggered by the `lake build` command or by other commands that require an artifact to be present, such as `lake exe`. A build consists of the following steps:

Configuring the package

If package configuration file is newer than the cached configuration file `lakefile.olean`, then the package configuration is re-elaborated. This also occurs when the cached file is missing or when the `--reconfigure` or `-R` flag is provided. Changes to options using `-K` do not trigger re-elaboration of the configuration file; `-R` is necessary in these cases.

Computing dependencies

The set of artifacts that are required to produce the desired output are determined, along with the targets and facets that produce them. This process is recursive, and the result is a *graph* of dependencies. The dependencies in this graph are distinct from those declared for a package: packages depend on other packages, while build targets depend on other build targets, which may be in the same package or in a different one. One facet of a given target may depend on other facets of the same target. Lake automatically analyzes the imports of Lean modules to discover their dependencies, and the `extraDepTargets` field can be used to add additional dependencies to a target.

Replaying traces

Rather than rebuilding everything in the dependency graph from scratch, Lake uses saved *trace files* to determine which artifacts require building. During a build, Lake records which source files or other artifacts were used to produce each artifact, saving a hash of each input; these *traces* are saved in the build directory. (i) More specifically, each artifact's trace file contains a Merkle tree hash mixture of its inputs' hashes. If the inputs are all unmodified, then the corresponding artifact is not rebuilt. Trace files additionally record the log from each build task; these outputs are replayed as if the artifact had been built anew. Re-using prior build products when possible is called an *incremental build*.

Building artifacts

When all unmodified dependencies in the dependency graph have been replayed from their trace files, Lake proceeds to build each artifact. This involves running the appropriate build tool on the input files and saving the artifact and its trace file, as specified in the corresponding facet.

Lake uses two separate hash algorithms. Text files are hashed after normalizing newlines, so that files that differ only by platform-specific newline conventions are hashed identically. Other files are hashed without any normalization.

Along with the trace files, Lean caches input hashes. Whenever an artifact is built, its hash is saved in a separate file that can be re-read instead of computing the hash from scratch. This is a performance optimization. This feature can be disabled, causing all hashes to be recomputed from their inputs, using the `--rehash` command-line option.

During a build, the following directories are provided to the underlying build tools:

- The *source directory* contains Lean source code that is available for import.
- The *library directories* contain `.olean` files along with the shared and static libraries that are available for linking; it normally consists of the root package's library directory (found in `.lake/build/lib`), the library directories for the other packages in the workspace, the library directory for the current Lean toolchain, and the system library directory.
- The *Lake home* is the directory in which Lake is installed, including binaries, source code, and libraries. The libraries in the Lake home are needed to elaborate Lake configuration files, which have access to the full power of Lean.

22.1.1.2. Facets

A *facet* describes the production of a target from another. Conceptually, any target may have facets. However, executables, external libraries, and custom targets provide only a single implicit facet. Packages, libraries, and modules have multiple facets that can be requested by name when invoking `lake build` to select the corresponding target.

When no facet is explicitly requested, but an initial target is designated, `lake build` produces the initial target's *default facet*. Each type of initial target has a corresponding default facet (e.g. producing an executable binary from an executable target or building a package's default targets); other facets may be explicitly requested in the package configuration or via Lake's command-line interface. Lake's internal API may be used to write custom facets.

The facets available for packages are:

`extraDep`
: The default facets of the package's extra dependency targets, specified in the `extraDepTargets` field.

`deps`
: The package's direct dependencies.

`transDeps`
: The package's transitive dependencies, topologically sorted.

`optCache`
: A package's optional cached build archive (e.g., from Reservoir or GitHub). Will **not** cause the whole build to fail if the archive cannot be fetched.

`cache`
: A package's cached build archive (e.g., from Reservoir or GitHub). Will cause the whole build to fail if the archive cannot be fetched.

`optBarrel`
: A package's optional cached build archive (e.g., from Reservoir or GitHub). Will **not** cause the whole build to fail if the archive cannot be fetched.

`barrel`
: A package's cached build archive (e.g., from Reservoir or GitHub). Will cause the whole build to fail if the archive cannot be fetched.

`optRelease`
: A package's optional build archive from a GitHub release. Will **not** cause the whole build to fail if the release cannot be fetched.

`release`
: A package's build archive from a GitHub release. Will cause the whole build to fail if the archive cannot be fetched.

The facets available for libraries are:

`leanArts`
: The artifacts that the Lean compiler produces for the library or executable (`*.olean`, `*.ilean`, and `*.c` files).

`static`
: The static library produced by the C compiler from the `leanArts` (that is, a `*.a` file).

`static.export`
: The static library produced by the C compiler from the `leanArts` (that is, a `*.a` file), with exported symbols.

`shared`
: The shared library produced by the C compiler from the `leanArts` (that is, a `*.so`, `*.dll`, or `*.dylib` file, depending on the platform).

`extraDep`
: A Lean library's `extraDepTargets` and those of its package.

Executables have a single `exe` facet that consists of the executable binary.

The facets available for modules are:

`lean`
: The module's Lean source file.

`leanArts` (default)
: The module's Lean artifacts (`*.olean`, `*.ilean`, `*.c` files).

`deps`
: The module's dependencies (e.g., imports or shared libraries).

`olean`

The module's `.olean` file.

`ilean`

The module's `.ilean` file, which is metadata used by the Lean language server.

`header`

The parsed module header of the module's source file.

`input`

The module's processed Lean source file. Combines tracing the file with parsing its header.

`imports`

The immediate imports of the Lean module, but not the full set of transitive imports.

`precompileImports`

The transitive imports of the Lean module, compiled to object code.

`transImports`

The transitive imports of the Lean module, as `.olean` files.

`allImports`

Both the immediate and transitive imports of the Lean module.

`setup`

All of a module's dependencies: transitive local imports and shared libraries to be loaded with `--load-dynlib`. Returns the list of shared libraries to load along with their search path.

`c`

The C file produced by the Lean compiler.

`bc`

LLVM bitcode file, produced by the Lean compiler.

`c.o`

The compiled object file, produced from the C file. On Windows, this is equivalent to `.c.o.noexport`, while it is equivalent to `.c.o.export` on other platforms.

`c.o.export`

The compiled object file, produced from the C file, with Lean symbols exported.

`c.o.noexport`

The compiled object file, produced from the C file, with Lean symbols exported.

`bc.o`

The compiled object file, produced from the LLVM bitcode file.

`o`

The compiled object file for the configured backend.

`dynlib`

A shared library (e.g., for the Lean option `--load-dynlib`).

22.1.1.3. Scripts

Lake package configuration files may include *Lake scripts*, which are embedded programs that can be executed from the command line. Scripts are intended to be used for project-specific tasks that are not already well-served by Lake's other features. While ordinary executable programs are run in the `IO` monad, scripts are run in `ScriptM`, which extends `IO` with information about the workspace. Because they are Lean definitions, Lake scripts can only be defined in the Lean configuration format.

22.1.1.4. Test and Lint Drivers

A *test driver* is responsible for running the tests for a package. Test drivers may be executable targets or Lake scripts, in which case the `lake test` command runs them, or they may be libraries, in which case `lake test` causes them to be elaborated, with the expectation that test failures are registered as elaboration failures.

Similarly, a *lint driver* is responsible for checking the code for stylistic issues. Lint drivers may be executables or scripts, which are run by `lake lint`.

A test or lint driver can be configured by either setting the `testDriver` or `lintDriver` package configuration options or by tagging a script, executable, or library with the `test_driver` or `lint_driver` attribute in a Lean-format configuration file. A definition in a dependency can be used as a test or lint driver by using the `<pkg>/<name>` syntax for the appropriate configuration option.

22.1.1.5. GitHub Release Builds

Lake supports uploading and downloading build artifacts (i.e., the archived build directory) to/from the GitHub releases of packages. This enables end users to fetch pre-built artifacts from the cloud without needed to rebuild the package from source themselves. The `LAKE_NO_CACHE` environment variable can be used to disable this feature.

22.1.1.5.1. Downloading

To download artifacts, one should configure the package options `releaseRepo` and `buildArchive` to point to the GitHub repository hosting the release and the correct artifact name within it (if the defaults are not sufficient). Then, set `preferReleaseBuild := true` to tell Lake to fetch and unpack it as an extra package dependency.

Lake will only fetch release builds as part of its standard build process if the package wanting it is a dependency (as the root package is expected to modified and thus not often compatible with this scheme). However, should one wish to fetch a release for a root package (e.g., after cloning the release's source but before editing), one can manually do so via `lake build :release`.

Lake internally uses `curl` to download the release and `tar` to unpack it, so the end user must have both tools installed in order to use this feature. If Lake fails to fetch a release for any reason, it will move on to building from the source. This mechanism is not technically limited to GitHub: any Git host that uses the same URL scheme works as well.

22.1.1.5.2. Uploading

To upload a built package as an artifact to a GitHub release, Lake provides the `lake upload` command as a convenient shorthand. This command uses `tar` to pack the package's build directory into an archive and uses `gh release upload` to attach it to a pre-existing GitHub release for the specified tag. Thus, in order to use it, the package uploader (but not the downloader) needs to have `gh`, the GitHub CLI, installed and in `PATH`.

22.1.2. Command-Line Interface

Lake's command-line interface is structured into a series of subcommands. All of the subcommands share the ability to be configured by certain environment variables and global command-line options. Each subcommand should be understood as a utility in its own right, with its own required argument syntax and documentation.

Some of Lake's commands delegate to other command-line utilities that are not included in a Lean distribution. These utilities must be available on the `PATH` in order to use the corresponding features:

- `git` is required in order to access Git dependencies.
- `tar` is required to create or extract cloud build archives, and `curl` is required to fetch them.
- `gh` is required to upload build artifacts to GitHub releases.

Lean distributions include a C compiler toolchain.

22.1.2.1. Environment Variables

When invoking the Lean compiler or other tools, Lake sets or modifies a number of environment variables. These values are system-dependent. Invoking `lake env` without any arguments displays the environment variables and their values. Otherwise, the provided command is invoked in Lake's environment.

The following variables are set, overriding previous values:

`LAKE`	The detected Lake executable
`LAKE_HOME`	The detected Lake home
`LEAN_SYSROOT`	The detected Lean toolchain directory
`LEAN_AR`	The detected Lean `ar` binary
`LEAN_CC`	The detected C compiler (if not using the bundled one)

The following variables are augmented with additional information:

`LEAN_PATH`	Lake's and the workspace's Lean library directories are added.
`LEAN_SRC_PATH`	Lake's and the workspace's source directories are added.
`PATH`	Lean's, Lake's, and the workspace's binary directories are added. On Windows, Lean's and the workspace's library directories are also added.
`DYLD_LIBRARY_PATH`	On macOS, Lean's and the workspace's library directories are added.
`LD_LIBRARY_PATH`	On platforms other than Windows and macOS, Lean's and the workspace's library directories are added.

Lake itself can be configured with the following environment variables:

`ELAN_HOME`	The location of the Elan installation, which is used for automatic toolchain updates.
`ELAN`	The location of the `elan` binary, which is used for automatic toolchain updates. If it is not set, an occurrence of `elan` must exist on the `PATH`.
`LAKE_HOME`	The location of the Lake installation. This environment variable is only consulted when Lake is unable to determine its installation path from the location of the `lake` executable that's currently running.
`LEAN_SYSROOT`	The location of the Lean installation, used to find the Lean compiler, the standard library, and other bundled tools. Lake first checks whether its binary is colocated with a Lean install, using that installation if so. If not, or if `LAKE_OVERRIDE_LEAN` is true, then Lake consults `LEAN_SYSROOT`. If this is not set, Lake consults the `LEAN` environment variable to find the Lean compiler, and attempts to find the Lean installation relative to the compiler. If `LEAN` is set but empty, Lake considers Lean to be disabled. If `LEAN_SYSROOT` and `LEAN` are unset, the first occurrence of `lean` on the `PATH` is used to find the installation.
`LEAN_CC` and `LEAN_AR`	If `LEAN_CC` and/or `LEAN_AR` is set, its value is used as the C compiler or `ar` command when building libraries. If not, Lake will fall back to the bundled tool in the Lean installation. If the bundled tool is not found, the value of `CC` or `AR`, followed by a `cc` or `ar` on the `PATH`, are used.
`LAKE_NO_CACHE`	If true, Lake does not use cached builds from Reservoir or GitHub. This environment variable can be overridden using the `--try-cache` command-line option.

Lake considers an environment variable to be true when its value is `y`, `yes`, `t`, `true`, `on`, or `1`, compared case-insensitively. It considers a variable to be false when its value is `n`, `no`, `f`, `false`, `off`, or `0`, compared case-insensitively. If the variable is unset, or its value is neither true nor false, a default value is used.

22.1.2.2. Options

Lake's command-line interface provides a number of global options as well as subcommands that perform important tasks. Single-character flags cannot be combined; `-HR` is not equivalent to `-H -R`.

`--version`

Lake outputs its version and exits without doing anything else.

`--help` or `-h`

Lake outputs its version along with usage information and exits without doing anything else. Subcommands may be used with `--help`, in which case usage information for the subcommand is output.

`--dir=DIR` or `-d=DIR`

Use the provided directory as location of the package instead of the current working directory. This is not always equivalent to changing to the directory first, because the version of `lake` indicated by the current directory's toolchain file will be used, rather than that of `DIR`.

`--file=FILE` or `-f=FILE`

Use the specified package configuration file instead of the default.

`--old`

Only rebuild modified modules, ignoring transitive dependencies. Modules that import the modified module will not be rebuilt. In order to accomplish this, file modification times are used instead of hashes to determine whether a module has changed.

`--rehash` or `-H`

Ignored cached file hashes, recomputing them. Lake uses hashes of dependencies to determine whether to rebuild an artifact. These hashes are cached on disk whenever a module is built. To save time during builds, these cached hashes are used instead of recomputing each hash unless `--rehash` is specified.

`--update`

Update dependencies after the package configuration is loaded but prior to performing other tasks, such as a build. This is equivalent to running `lake update` before the selected command, but it may be faster due to not having to load the configuration twice.

`--packages=FILE`

Use the contents of `FILE` to specify the versions of some or all dependencies instead of the manifest. `FILE` should be a syntactically valid manifest, but it does not need to be complete.

`--reconfigure` or `-R`

Normally, the package configuration file is elaborated when a package is first configured, with the result cached to a `.olean` file that is used for future invocations until the package configuration Providing this flag causes the configuration file to be re-elaborated.

`--keep-toolchain`

By default, Lake attempts to update the local workspace's toolchain file. Providing this flag disables automatic toolchain updates.

`--no-build`

Lake exits immediately if a build target is not up-to-date, returning a non-zero exit code.

`--no-cache`

Instead of using available cloud build caches, build all packages locally. Build caches are not downloaded.

`--try-cache`

attempt to download build caches for supported packages

22.1.2.3. Controlling Output

These options provide allow control over the log that is produced while building. In addition to showing or hiding messages, a build can be made to fail when warnings or even information is emitted; this can be used to enforce a style guide that disallows output during builds.

`--quiet`, `-q`

Hides informational logs and the progress indicator.

`--verbose`, `-v`

Shows trace logs (typically command invocations) and built targets.

`--ansi`, `--no-ansi`

Enables or disables the use of ANSI escape codes that add colors and animations to Lake's output.

`--log-level=LV`

Sets the minimum level of logs to be shown when builds succeed. `LV` may be `trace`, `info`, `warning`, or `error`, compared case-insensitively. When a build fails, all levels are shown. The default log level is `info`.

`--fail-level=LV`

Sets the threshold at which a message in the log causes a build to be considered a failure. If a message is emitted to the log with a level that is greater than or equal to the threshold, the build fails. `LV` may be `trace`, `info`, `warning`, or `error`, compared case-insensitively; it is `error` by default.

`--iofail`

Causes builds to fail if any I/O or other info is logged. This is equivalent to `--fail-level=info`.

`--wfail`

Causes builds to fail if any warnings are logged. This is equivalent to `--fail-level=warning`.

22.1.2.4. Automatic Toolchain Updates

The `lake update` command checks for changes to dependencies, fetching their sources and updating the manifest accordingly. By default, `lake update` also attempts to update the root package's toolchain file when a new version of a dependency specifies an updated toolchain. This behavior can be disabled with the `--keep-toolchain` flag.

If multiple dependencies specify newer toolchains, Lake selects the newest compatible toolchain, if it exists. To determine the newest compatible toolchain, Lake parses the toolchain listed in the packages' `lean-toolchain` files into four categories:

- Releases, which are compared by version number (e.g., `v4.4.0` < `v4.8.0` and `v4.6.0-rc1` < `v4.6.0`)
- Nightly builds, which are compared by date (e.g., `nightly-2024-01-10` < `nightly-2024-10-01`)
- Builds from pull requests to the Lean compiler, which are incomparable
- Other versions, which are also incomparable

Toolchain versions from multiple categories are incomparable. If there is not a single newest toolchain, Lake will print a warning and continue updating without changing the toolchain.

If Lake does find a new toolchain, then it updates the workspace's `lean-toolchain` file accordingly and restarts the `lake update` using the new toolchain's Lake. If Elan is detected, it will spawn the new Lake process via `elan run` with the same arguments Lake was initially run with. If Elan is missing, it will prompt the user to restart Lake manually and exit with a special error code (namely, 4). The Elan executable used by Lake can be configured using the `ELAN` environment variable.

22.1.2.5. Creating Packages

Lake command

`lake new` *`name`* `[`*`template`*`][.`*`language`*`]`

Running `lake new` creates an initial Lean package in a new directory. This command is equivalent to creating a directory named *`name`* and then running `lake init`

Lake command

`lake init` *`name`* `[`*`template`*`][.`*`language`*`]`

Running `lake init` creates an initial Lean package in the current directory. The package's contents are based on a template, with the names of the package, its targets, and their module roots derived from the name of the current directory.

The *template* may be:

std (default)
: Creates a package that contains a library and an executable.

exe
: Creates a package that contains only an executable.

lib
: Creates a package that contains only a library.

math
: Creates a package that contains a library that depends on Mathlib.

The *language* selects the file format used for the package configuration file and may be `lean` (the default) or `toml`.

22.1.2.6. Building and Running

Lake command

`lake build` [*targets...*]

Builds the specified facts of the specified targets.

Each of the *targets* is specified by a string of the form:

[[@]*package*[/]][*target*|[+]*module*][:*facet*]

The optional @ and + markers can be used to disambiguate packages and modules from file paths as well as executables, and libraries, which are specified by name as *target*. If not provided, *package* defaults to the workspace's root package. If the same target name exists in multiple packages in the workspace, then the first occurrence of the target name found in a topological sort of the package dependency graph is selected. Module targets may also be specified by their filename, with an optional facet after a colon.

The available facets depend on whether a package, library, executable, or module is to be built. They are listed in the section on facets.

▼ *Example: Target and Facet Specifications*

a	The default facet(s) of target a
@a	The default targets of package a
+A	The Lean artifacts of module A (because the default facet of modules is `leanArts`)

`@a/b`	The default facet of target b of package a
`@a/+A:c`	The C file compiled from module A of package a
`:foo`	The root package's facet `foo`
`A/B/C.lean:o`	The compiled object code for the module in the file `A/B/C.lean`

Lake command

`lake check-build`

Exits with code 0 if the workspace's root package has any default targets configured. Errors (with exit code 1) otherwise.

`lake check-build` does **not** verify that the configured default targets are valid. It merely verifies that at least one is specified.

Lake command

`lake query` [*`targets...`*]

Builds a set of targets, reporting progress on standard error and outputting the results on standard out. Target results are output in the same order they are listed and end with a newline. If `--json` is set, results are formatted as JSON. Otherwise, they are printed as raw strings.

Targets which do not have output configured will be printed as an empty string or `null`. For executable targets, the output is the path to the built executable.

Targets are specified using the same syntax as in `lake build`.

Lake command

`lake exe` *`exe-target`* [*`args...`*]

Alias: `lake exec`

Looks for the executable target *`exe-target`* in the workspace, builds it if it is out of date, and then runs it with the given *`args`* in Lake's environment.

See `lake build` for the syntax of target specifications and `lake env` for a description of how the environment is set up.

Lake command

`lake clean` [*`packages...`*]

If no package is specified, deletes the build directories of every package in the workspace. Otherwise, it just deletes those of the specified *packages*.

Lake command

lake env [*cmd* [*args...*]]

When *cmd* is provided, it is executed in the Lake environment with arguments *args*.

If *cmd* is not provided, Lake prints the environment in which it runs tools. This environment is system-specific.

Lake command

lake lean *file* [-- *args...*]

Builds the imports of the given *file* and then runs `lean` on it using the workspace's root package's additional Lean arguments and the given *args*, in that order. The `lean` process is executed in Lake's environment.

22.1.2.7. Development Tools

Lake includes support for specifying standard development tools and workflows. On the command line, these tools can be invoked using the appropriate `lake` subcommands.

22.1.2.7.1. Tests and Linters

Lake command

lake test [-- *args...*]

Test the workspace's root package using its configured test driver.

A test driver that is an executable will be built and then run with the package configuration's `testDriverArgs` plus the CLI *args*. A test driver that is a Lake script is run with the same arguments as an executable test driver. A library test driver will just be built; it is expected that tests are implemented such that failures cause the build to fail via elaboration-time errors.

Lake command

lake lint [-- *args...*]

Lint the workspace's root package using its configured lint driver

A script lint driver will be run with the package configuration's `lintDriverArgs` plus the CLI `args`. An executable lint driver will be built and then run like a script.

Lake command

`lake check-test`

Check if there is a properly configured test driver

Exits with code 0 if the workspace's root package has a properly configured lint driver. Errors (with code 1) otherwise.

Does NOT verify that the configured test driver actually exists in the package or its dependencies. It merely verifies that one is specified.

This is useful for distinguishing between failing tests and incorrectly configured packages.

Lake command

`lake check-lint`

Check if there is a properly configured lint driver

Exits with code 0 if the workspace's root package has a properly configured lint driver. Errors (with code 1) otherwise.

Does NOT verify that the configured lint driver actually exists in the package or its dependencies. It merely verifies that one is specified.

This is useful for distinguishing between failing lints and incorrectly configured packages.

22.1.2.7.2. Scripts

Lake command

`lake script list`

Alias: `lake scripts`

Lists the available scripts in the workspace.

Lake command

`lake script run` `[[`*`package`*`/]`*`script`* `[`*`args...`*`]]`

Alias: `lake run`

This command runs the *`script`* of the workspace (or the specified *`package`*), passing *`args`* to it.

A bare `lake run` command will run the default script(s) of the root package(with no arguments).

Lake command

`lake script doc` *`script`*

Prints the documentation comment for *`script`*.

22.1.2.7.3. Language Server

Lake command

`lake serve` [`--` *`args...`*]

Runs the Lean language server in the workspace's root project with the package configuration's `moreServerArgs` field and *`args`*.

This command is typically invoked by editors or other tooling, rather than manually.

22.1.2.8. Dependency Management

Lake command

`lake update` [*`packages...`*]

Updates the Lake package manifest (i.e., `lake-manifest.json`), downloading and upgrading packages as needed. For each new (transitive) Git dependency, the appropriate commit is cloned into a subdirectory of the workspace's package directory. No copy is made of local dependencies.

If a set of packages *`packages`* is specified, then these dependencies are upgraded to the latest version compatible with the package's configuration (or removed if removed from the configuration). If there are dependencies on multiple versions of the same package, an arbitrary version is selected.

A bare `lake update` will upgrade all dependencies.

22.1.2.9. Packaging and Distribution

Lake command

`lake upload` *`tag`*

Packs the root package's `buildDir` into a `tar.gz` archive using `tar` and then uploads the asset to the pre-existing GitHub release *`tag`* using gh. Other hosts are not yet supported.

22.1.2.9.1. Cached Cloud Builds

These commands are still experimental. They are likely change in future versions of Lake based on user feedback. Packages that use Reservoir cloud build archives should enable the `platformIndependent` setting.

Lake command

`lake pack` [*`archive.tar.gz`*]

Packs the root package's build directory into a gzipped tar archive using `tar`. If a path for the archive is not specified, the archive in the package's Lake directory (`.lake`) and named according to its `buildArchive` setting. This command does not build any artifacts: it only archives what is present. Users should ensure that the desired artifacts are present before running this command.

Lake command

`lake unpack` [*`archive.tar.gz`*]

Unpacks the contents of the gzipped tar archive *`archive.tgz`* into the root package's build directory. If *`archive.tgz`* is not specified, the package's `buildArchive` setting is used to determine a filename, and the file is expected in package's Lake directory (`.lake`).

22.1.2.10. Configuration Files

Lake command

`lake translate-config` *`lang`* [*`out-file`*]

Translates the loaded package's configuration into another of Lake's supported configuration languages (i.e., either `lean` or `toml`). The produced file is written to `out-file` or, if not provided, the path of the configuration file with the new language's extension. If the output file already exists, Lake will error.

Translation is lossy. It does not preserve comments or formatting and non-declarative configuration is discarded.

22.1.3. Configuration File Format

Lake offers two formats for package configuration files:

TOML
: The TOML configuration format is fully declarative. Projects that don't include custom targets, facets, or scripts can use the TOML format. Because TOML parsers are available for a wide variety of languages, using this format facilitates integration with tools that are not written in Lean.

Lean

The Lean configuration format is more flexible and allows for custom targets, facets, and scripts. It features an embedded domain-specific language for describing the declarative subset of configuration options that is available from the TOML format. Additionally, the Lake API can be used to express build configurations that are outside of the possibilities of the declarative options.

The command `lake translate-config` can be used to automatically convert between the two formats.

Both formats are processed similarly by Lake, which extracts the package configuration from the configuration file in the form of internal structure types. When the package is configured, the resulting data structures are written to `lakefile.olean` in the build directory.

22.1.3.1. Declarative TOML Format

TOML *Tom's Obvious Minimal Language* is a standardized format for configuration files. configuration files describe the most-used, declarative subset of Lake package configuration files. TOML files denote *tables*, which map keys to values. Values may consist of strings, numbers, arrays of values, or further tables. Because TOML allows considerable flexibility in file structure, this reference documents the values that are expected rather than the specific syntax used to produce them.

The contents of `lakefile.toml` should denote a TOML table that describes a Lean package. This configuration consists of both scalar fields that describe the entire package, as well as the following fields that contain arrays of further tables:

- `require`
- `lean_lib`
- `lean_exe`

Fields that are not part of the configuration tables described here are presently ignored. To reduce the risk of typos, this is likely to change in the future. Field names not used by Lake should not be used to store metadata to be processed by other tools.

22.1.3.1.1. Package Configuration

The top-level contents of `lakefile.toml` specify the options that apply to the package itself, including metadata such as the name and version, the locations of the files in the workspace, compiler flags to be used for all targets, and The only mandatory field is `name`, which declares the package's name.

TOML table

`Package Configuration`

A `Package`'s declarative configuration.

Metadata:

These options describe the package. They are used by Reservoir to index and display packages. If a field is left out, Reservoir may use information from the package's GitHub repository to fill in details.

`version`

Contains: Version string

The package version. Versions have the form:

`v!"<major>.<minor>.<patch>[-<specialDescr>]"`

A version with a `-` suffix is considered a "prerelease".

Lake suggest the following guidelines for incrementing versions:

- **Major version increment** *(e.g., v1.3.0 → v2.0.0)* Indicates significant breaking changes in the package. Package users are not expected to update to the new version without manual intervention.
- **Minor version increment** *(e.g., v1.3.0 → v1.4.0)* Denotes notable changes that are expected to be generally backwards compatible. Package users are expected to update to this version automatically and should be able to fix any breakages and/or warnings easily.
- **Patch version increment** *(e.g., v1.3.0 → v1.3.1)* Reserved for bug fixes and small touchups. Package users are expected to update automatically and should not expect significant breakage, except in the edge case of users relying on the behavior of patched bugs.

Note that backwards-incompatible changes may occur at any version increment. The is because the current nature of Lean (e.g., transitive imports, rich metaprogramming, reducibility in proofs), makes it infeasible to define a completely stable interface for a package. Instead, the different version levels indicate a change's intended significance and how difficult migration is expected to be.

Versions of form the `0.x.x` are considered development versions prior to first official release. Like prerelease, they are not expected to closely follow the above guidelines.

Packages without a defined version default to `0.0.0`.

`versionTags`

Contains: string pattern

Git tags of this package's repository that should be treated as versions. Package indices (e.g., Reservoir) can make use of this information to determine the Git revisions corresponding to released versions.

Defaults to tags that are "version-like". That is, start with a `v` followed by a digit.

`description`

Contains: String

A short description for the package (e.g., for Reservoir).

`keywords`

Contains: Array of strings

Custom keywords associated with the package. Reservoir can make use of a package's keywords to group related packages together and make it easier for users to discover them.

Good keywords include the domain (e.g., `math`, `software-verification`, `devtool`), specific subtopics (e.g., `topology`, `cryptology`), and significant implementation details (e.g., `dsl`, `ffi`, `cli`). For instance, Lake's keywords could be `devtool`, `cli`, `dsl`, `package-manager`, and `build-system`.

`homepage`

Contains: String

A URL to information about the package.

Reservoir will already include a link to the package's GitHub repository (if the package is sourced from there). Thus, users are advised to specify something else for this (if anything).

`license`

Contains: String

The package's license (if one). Should be a valid SPDX License Expression.

Reservoir requires that packages uses an OSI-approved license to be included in its index, and currently only supports single identifier SPDX expressions. For, a list of OSI-approved SPDX license identifiers, see the SPDX LIcense List.

`licenseFiles`

Contains: Array of paths

Files containing licensing information for the package.

These should be the license files that users are expected to include when distributing package sources, which may be more then one file for some licenses. For example, the Apache 2.0 license requires the reproduction of a `NOTICE` file along with the license (if such a file exists).

Defaults to `#["LICENSE"]`.

`readmeFile`

Contains: Path

The path to the package's README.

A README should be a Markdown file containing an overview of the package. Reservoir displays the rendered HTML of this file on a package's page. A nonstandard location can be used to provide a different README for Reservoir and GitHub.

Defaults to `README.md`.

`reservoir`

Contains: Boolean

Whether Reservoir should include the package in its index. When set to `false`, Reservoir will not add the package to its index and will remove it if it was already there (when Reservoir is next updated).

Layout:

These options control the top-level directory layout of the package and its build directory. Further paths specified by libraries, executables, and targets within the package are relative to these directories.

`srcDir`

Contains: Path

The directory containing the package's Lean source files. Defaults to the package's directory.

(This will be passed to `lean` as the `-R` option.)

`buildDir`

Contains: Path

The directory to which Lake should output the package's build results. Defaults to `defaultBuildDir` (i.e., `.lake/build`).

`nativeLibDir`

Contains: Path

The build subdirectory to which Lake should output the package's native libraries (e.g., `.a`, `.so`, `.dll` files). Defaults to `defaultNativeLibDir` (i.e., `lib`).

`binDir`

Contains: Path

The build subdirectory to which Lake should output the package's binary executable. Defaults to `defaultBinDir` (i.e., `bin`).

`irDir`

Contains: Path

The build subdirectory to which Lake should output the package's intermediary results (e.g., `.c` and `.o` files). Defaults to `defaultIrDir` (i.e., `ir`).

`packagesDir`

Contains: Path

The directory to which Lake should download remote dependencies. Defaults to `defaultPackagesDir` (i.e., `.lake/packages`).

Building and Running:

These options configure how code is built and run in the package. Libraries, executables, and other targets within a package can further add to parts of this configuration.

`extraDepTargets`

Contains: Array of strings

An `Array` of target names to build whenever the package is used.

`precompileModules`

Contains: Boolean

Whether to compile each of the package's module into a native shared library that is loaded whenever the module is imported. This speeds up evaluation of metaprograms and enables the interpreter to run functions marked `@[extern]`.

Defaults to `false`.

`defaultTargets`

Contains: default targets' names (array)

The names of the package's targets to build by default (i.e., on a bare `lake build` of the package).

`moreGlobalServerArgs`

Contains: Array of strings

Additional arguments to pass to the Lean language server (i.e., `lean --server`) launched by `lake serve`, both for this package and also for any packages browsed from this one in the same session.

`leanLibDir`

Contains: Path

The build subdirectory to which Lake should output the package's binary Lean libraries (e.g., `.olean`, `.ilean` files). Defaults to `defaultLeanLibDir` (i.e., `lib`).

`buildType`

Contains: one of `"debug"`, `"relWithDebInfo"`, `"minSizeRel"`, `"release"`

The mode in which the modules should be built (e.g., `debug`, `release`). Defaults to `release`.

`leanOptions`

Contains: Array of Lean options

An `Array` of additional options to pass to both the Lean language server (i.e., `lean --server`) launched by `lake serve` and to `lean` when compiling a module's Lean source files.

`moreLeanArgs`

Contains: Array of strings

Additional arguments to pass to `lean` when compiling a module's Lean source files.

`weakLeanArgs`

Contains: Array of strings

Additional arguments to pass to `lean` when compiling a module's Lean source files.

Unlike `moreLeanArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLeanArgs`.

`moreLeancArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when compiling a module's C source files generated by `lean`.

Lake already passes some flags based on the `buildType`, but you can change this by, for example, adding `-O0` and `-UDEBUG`.

`moreServerOptions`

Contains: Array of Lean options

Additional options to pass to the Lean language server (i.e., `lean --server`) launched by `lake serve`.

`weakLeancArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when compiling a module's C source files generated by `lean`.

Unlike `moreLeancArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLeancArgs`.

`moreLinkArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of the linked objects.

`weakLinkArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of the linked objects.

Unlike `moreLinkArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLinkArgs`.

`platformIndependent`

Contains: Boolean (optional)

Asserts whether Lake should assume Lean modules are platform-independent.

- If `false`, Lake will add `System.Platform.target` to the module traces within the code unit (e.g., package or library). This will force Lean code to be re-elaborated on different platforms.
- If `true`, Lake will exclude platform-dependent elements (e.g., precompiled modules, external libraries) from a module's trace, preventing re-elaboration on different platforms. Note that this will not effect modules outside the code unit in question. For example, a platform-independent package which depends on a platform-dependent library will still be platform-dependent.
- If `none`, Lake will construct traces as natural. That is, it will include platform-dependent artifacts in the trace if they module depends on them, but otherwise not force modules to be platform-dependent.

There is no check for correctness here, so a configuration can lie and Lake will not catch it. Defaults to `none`.

Testing and Linting:

The CLI commands `lake test` and `lake lint` use definitions configured by the workspace's root package to perform testing and linting. The code that is run to perform tests and lits are referred to as the test or lint driver. In Lean configuration files, these can be specified by applying the `@[test_driver]` or `@[lint_driver]` attributes to a Lake script or an executable or library target. In both Lean and TOML configuration files, they can also be configured by setting these options. A target or script TGT from a dependency PKG can be specified as a test or lint driver using the string `"PKG/TGT"`

`testDriver`

Contains: String

The name of the script, executable, or library by `lake test` when this package is the workspace root. To point to a definition in another package, use the syntax `<pkg>/<def>`.

A script driver will be run by `lake test` with the arguments configured in `testDriverArgs` followed by any specified on the CLI (e.g., via `lake lint -- <args>...`). An executable driver will be built and then run like a script. A library will just be built.

`testDriverArgs`

Contains: Array of strings

Arguments to pass to the package's test driver. These arguments will come before those passed on the command line via `lake test -- <args>...`.

`lintDriver`

Contains: String

The name of the script or executable used by `lake lint` when this package is the workspace root. To point to a definition in another package, use the syntax `<pkg>/<def>`.

A script driver will be run by `lake lint` with the arguments configured in `lintDriverArgs` followed by any specified on the CLI (e.g., via `lake lint -- <args>...`). An executable driver will be built and then run like a script.

`lintDriverArgs`

Contains: Array of strings

Arguments to pass to the package's linter. These arguments will come before those passed on the command line via `lake lint -- <args>...`.

Cloud Releases:

These options define a cloud release for the package, as described in the section on GitHub release builds.

`releaseRepo`

Contains: String (optional)

The URL of the GitHub repository to upload and download releases of this package. If `none` (the default), for downloads, Lake uses the URL the package was download from (if it is a dependency) and for uploads, uses `gh`'s default.

`buildArchive`

Contains: String (optional)

A custom name for the build archive for the GitHub cloud release. If `none` (the default), Lake defaults to `{(pkg-)name}-{System.Platform.target}.tar.gz`.

`preferReleaseBuild`

Contains: Boolean

Whether to prefer downloading a prebuilt release (from GitHub) rather than building this package from the source when this package is used as a dependency.

Other Fields:

`bootstrap`

Contains: Boolean

For internal use. Whether this package is Lean itself.

`enableArtifactCache?`

Contains: Boolean (optional)

`moreLinkObjs`

Contains: Array of paths

Additional target objects to use when linking (both static and shared). These will come *after* the paths of native facets.

`moreLinkLibs`

Contains: Array of dynamic libraries

Additional target libraries to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of other link objects.

▼ *Example: Minimal TOML Package Configuration*

The minimal TOML configuration for a Lean package sets only the package's name, using the default values for all other fields. This package contains no targets, so there is no code to be built.

```
name = "example-package"
```

▼ *Example: Library TOML Package Configuration*

The minimal TOML configuration for a Lean package sets the package's name and defines a library target. This library is named `Sorting`, and its modules are expected under the `Sorting.*` hierarchy.

```
name = "example-package"
defaultTargets = ["Sorting"]

[[lean_lib]]
name = "Sorting"
```

22.1.3.1.2. Dependencies

Dependencies are specified in the `[[require]]` field array of a package configuration, which specifies both the name and the source of each package. There are three kinds of sources:

- Reservoir, or an alternative package registry
- Git repositories, which may be local paths or URLs
- Local paths

TOML table

`Requiring Packages — [[require]]`

A `Dependency` of a package. It specifies a package which another package depends on. This encodes the information contained in the `require` DSL syntax.

The `path` and `git` fields specify an explicit source for a dependency. If neither are provided, then the dependency is fetched from Reservoir, or an alternative registry if one has been configured. The `scope` field is required when fetching a package from Reservoir.

Fields:

`path`

Contains: Path

A dependency on the local filesystem, specified by its path.

`git`

Contains: Git specification

A dependency in a Git repository, specified either by its URL as a string or by a table with the keys:

- `url`: the repository URL
- `subDir`: the subdirectory of the Git repository that contains the package's source code

`rev`

Contains: Git revision

For Git or Reservoir dependencies, this field specifies the Git revision, which may be a branch name, a tag name, or a specific hash. On Reservoir, the `version` field takes precedence over this field.

`source`

Contains: Package Source

A dependency source, specified as a self-contained table, which is used when neither the `git` nor the `path` key is present. The key `type` should be either the string `"git"` or the string `"path"`. If the type is `"path"`, then there must be a further key `"path"`

whose value is a string that provides the location of the package on disk. If the type is `"git"`, then the following keys should be present:

- `url`: the repository URL
- `rev`: the Git revision, which may be a branch name, a tag name, or a specific hash (optional)
- `subDir`: the subdirectory of the Git repository that contains the package's source code

`version`

Contains: version as string

The target version of the dependency. A Git revision can be specified with the syntax `git#<rev>`.

`name`

Contains: String

The package name of the dependency. This name must match the one declared in its configuration file, as that name is used to index its target data types. For this reason, the package name must also be unique across packages in the dependency graph.

`scope`

Contains: String

An additional qualifier used to distinguish packages of the same name in a Lake registry. On Reservoir, this is the package owner.

▼ *Example: Requiring Packages from Reservoir*

The package `example` can be required from Reservoir using this TOML configuration:

```
[[require]]
name = "example"
version = "2.12"
scope = "exampleDev"
```

▼ *Example: Requiring Packages from Git*

The package `example` can be required from a Git repository using this TOML configuration:

```
[[require]]
name = "example"
git = "https://git.example.com/example.git"
rev = "main"
version = "2.12"
```

In particular, the package will be checked out from the `main` branch, and the version number specified in the package's configuration should match `2.12`.

▼ *Example: Requiring Packages from a Git tag*

The package `example` can be required from the tag `v2.12` in a Git repository using this TOML configuration:

```
[[require]]
name = "example"
git = "https://git.example.com/example.git"
rev = "v2.12"
```

The version number specified in the package's configuration is not used.

▼ *Example: Requiring Reservoir Packages from a Git tag*

The package `example`, found using Reservoir, can be required from the tag `v2.12` in its Git repository using this TOML configuration:

```
[[require]]
name = "example"
rev = "v2.12"
scope = "exampleDev"
```

The version number specified in the package's configuration is not used.

▼ *Example: Requiring Packages from Paths*

The package `example` can be required from the local path `../example` using this TOML configuration:

```
[[require]]
name = "example"
path = "../example"
```

Dependencies on local paths are useful when developing multiple packages in a single repository, or when testing whether a change to a dependency fixes a bug in a downstream package.

▼ *Example: Sources as Tables*

The information about the package source can be written in an explicit table.

```
[[require]]
name = "example"
source = {type = "git", url =
"https://example.com/example.git"}
```

22.1.3.1.3. Library Targets

Library targets are expected in the `lean_lib` array of tables.

TOML table

`Library Targets — [[lean_lib]]`

A Lean library's declarative configuration.

Fields:

`srcDir`

Contains: Path

The subdirectory of the package's source directory containing the library's Lean source files. Defaults simply to said `srcDir`.

(This will be passed to `lean` as the `-R` option.)

`roots`

Contains: Array of strings

The root module(s) of the library. Submodules of these roots (e.g., `Lib.Foo` of `Lib`) are considered part of the library. Defaults to a single root of the target's name.

`libName`

Contains: String

The name of the library artifact. Used as a base for the file names of its static and dynamic binaries. Defaults to the mangled name of the target.

`needs`

Contains: Array of targets

An `Array` of targets to build before the executable's modules.

`extraDepTargets`

Contains: Array of strings

Deprecated. Use `needs` instead. An `Array` of target names to build before the library's modules.

`precompileModules`

Contains: Boolean

Whether to compile each of the library's modules into a native shared library that is loaded whenever the module is imported. This speeds up evaluation of metaprograms and enables the interpreter to run functions marked `@[extern]`.

Defaults to `false`.

`defaultFacets`

Contains: Array of strings

An `Array` of library facets to build on a bare `lake build` of the library. For example, `#[LeanLib.sharedLib]` will build the shared library facet.

`buildType`

Contains: one of `"debug"`, `"relWithDebInfo"`, `"minSizeRel"`, `"release"`

The mode in which the modules should be built (e.g., `debug`, `release`). Defaults to `release`.

`leanOptions`

Contains: Array of Lean options

An `Array` of additional options to pass to both the Lean language server (i.e., `lean --server`) launched by `lake serve` and to `lean` when compiling a module's Lean source files.

`moreLeanArgs`

Contains: Array of strings

Additional arguments to pass to `lean` when compiling a module's Lean source files.

`weakLeanArgs`

Contains: Array of strings

Additional arguments to pass to `lean` when compiling a module's Lean source files.

Unlike `moreLeanArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLeanArgs`.

`moreLeancArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when compiling a module's C source files generated by `lean`.

Lake already passes some flags based on the `buildType`, but you can change this by, for example, adding `-O0` and `-UNDEBUG`.

`moreServerOptions`

Contains: Array of Lean options

Additional options to pass to the Lean language server (i.e., `lean --server`) launched by `lake serve`.

`weakLeancArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when compiling a module's C source files generated by `lean`.

Unlike `moreLeancArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLeancArgs`.

`moreLinkObjs`

Contains: Array of paths

Additional target objects to use when linking (both static and shared). These will come *after* the paths of native facets.

`moreLinkLibs`

Contains: Array of dynamic libraries

Additional target libraries to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of other link objects.

`moreLinkArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of the linked objects.

`weakLinkArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of the linked objects.

Unlike `moreLinkArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLinkArgs`.

`platformIndependent`

Contains: Boolean (optional)

Asserts whether Lake should assume Lean modules are platform-independent.

- If `false`, Lake will add `System.Platform.target` to the module traces within the code unit (e.g., package or library). This will force Lean code to be re-elaborated on different platforms.

- If `true`, Lake will exclude platform-dependent elements (e.g., precompiled modules, external libraries) from a module's trace, preventing re-elaboration on different platforms. Note that this will not effect modules outside the code unit in question. For example, a platform-independent package which depends on a platform-dependent library will still be platform-dependent.
- If `none`, Lake will construct traces as natural. That is, it will include platform-dependent artifacts in the trace if they module depends on them, but otherwise not force modules to be platform-dependent.

There is no check for correctness here, so a configuration can lie and Lake will not catch it. Defaults to `none`.

`dynlibs`

Contains: Array of dynamic libraries

`plugins`

Contains: Array of dynamic libraries

▼ *Example: Minimal Library Target*

This library declaration supplies only a name:

```
[[lean_lib]]
name = "TacticTools"
```

The library's source is located in the package's default source directory, in the module hierarchy rooted at `TacticTools`.

▼ *Example: Configured Library Target*

This library declaration supplies more options:

```
[[lean_lib]]
name = "TacticTools"
srcDir = "src"
precompileModules = true
```

The library's source is located in the directory `src`, in the module hierarchy rooted at `TacticTools`. If its modules are accessed at elaboration time, they will be compiled to native code and linked in, rather than run in the interpreter.

22.1.3.1.4. Executable Targets

TOML table

`Executable Targets — [[lean_exe]]`

A Lean executable's declarative configuration.

Fields:

`srcDir`

Contains: Path

The subdirectory of the package's source directory containing the executable's Lean source file. Defaults simply to said `srcDir`.

(This will be passed to `lean` as the `-R` option.)

`root`

Contains: String

The root module of the binary executable. Should include a `main` definition that will serve as the entry point of the program.

The root is built by recursively building its local imports (i.e., fellow modules of the workspace).

Defaults to the name of the target.

`exeName`

Contains: String

The name of the binary executable. Defaults to the target name with any `.` replaced with a `-`.

`needs`

Contains: Array of targets

An `Array` of targets to build before the executable's modules.

`extraDepTargets`

Contains: Array of strings

Deprecated. Use `needs` instead. An `Array` of target names to build before the executable's modules.

`supportInterpreter`

Contains: Boolean

Enables the executable to interpret Lean files (e.g., via `Lean.Elab.runFrontend`) by exposing symbols within the executable to the Lean interpreter.

Implementation-wise, on Windows, the Lean shared libraries are linked to the executable and, on other systems, the executable is linked with `-rdynamic`. This increases the size of the binary on Linux and, on Windows, requires

`libInit_shared.dll` and `libleanshared.dll` to be co-located with the executable or part of PATH (e.g., via `lake exe`). Thus, this feature should only be enabled when necessary.

Defaults to `false`.

`buildType`

Contains: one of `"debug"`, `"relWithDebInfo"`, `"minSizeRel"`, `"release"`

The mode in which the modules should be built (e.g., `debug`, `release`). Defaults to `release`.

`leanOptions`

Contains: Array of Lean options

An `Array` of additional options to pass to both the Lean language server (i.e., `lean --server`) launched by `lake serve` and to `lean` when compiling a module's Lean source files.

`moreLeanArgs`

Contains: Array of strings

Additional arguments to pass to `lean` when compiling a module's Lean source files.

`weakLeanArgs`

Contains: Array of strings

Additional arguments to pass to `lean` when compiling a module's Lean source files.

Unlike `moreLeanArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLeanArgs`.

`moreLeancArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when compiling a module's C source files generated by `lean`.

Lake already passes some flags based on the `buildType`, but you can change this by, for example, adding `-O0` and `-UNDEBUG`.

`moreServerOptions`

Contains: Array of Lean options

Additional options to pass to the Lean language server (i.e., `lean --server`) launched by `lake serve`.

`weakLeancArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when compiling a module's C source files generated by `lean`.

Unlike `moreLeancArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLeancArgs`.

`moreLinkObjs`

Contains: Array of paths

Additional target objects to use when linking (both static and shared). These will come *after* the paths of native facets.

`moreLinkLibs`

Contains: Array of dynamic libraries

Additional target libraries to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of other link objects.

`moreLinkArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of the linked objects.

`weakLinkArgs`

Contains: Array of strings

Additional arguments to pass to `leanc` when linking (e.g., for shared libraries or binary executables). These will come *after* the paths of the linked objects.

Unlike `moreLinkArgs`, these arguments do not affect the trace of the build result, so they can be changed without triggering a rebuild. They come *before* `moreLinkArgs`.

`platformIndependent`

Contains: Boolean (optional)

Asserts whether Lake should assume Lean modules are platform-independent.

- If `false`, Lake will add `System.Platform.target` to the module traces within the code unit (e.g., package or library). This will force Lean code to be re-elaborated on different platforms.
- If `true`, Lake will exclude platform-dependent elements (e.g., precompiled modules, external libraries) from a module's trace, preventing re-elaboration on different platforms. Note that this will not effect modules outside the code unit in question. For example, a platform-independent package which depends on a platform-dependent library will still be platform-dependent.

- If none, Lake will construct traces as natural. That is, it will include platform-dependent artifacts in the trace if they module depends on them, but otherwise not force modules to be platform-dependent.

There is no check for correctness here, so a configuration can lie and Lake will not catch it. Defaults to none.

dynlibs

Contains: Array of dynamic libraries

plugins

Contains: Array of dynamic libraries

▼ *Example: Minimal Executable Target*

This executable declaration supplies only a name:

```
[[lean_exe]]
name = "trustworthytool"
```

The executable's `main` function is expected in a module named `trustworthytool.lean` in the package's default source file path. The resulting executable is named `trustworthytool`.

▼ *Example: Configured Executable Target*

The name `trustworthy-tool` is not a valid Lean name due to the dash (`-`). To use this name for an executable target, an explicit module root must be supplied. Even though `trustworthy-tool` is a perfectly acceptable name for an executable, the target also specifies that the result of compilation and linking should be named `tt`.

```
[[lean_exe]]
name = "trustworthy-tool"
root = "TrustworthyTool"
exeName = "tt"
```

The executable's `main` function is expected in a module named `TrustworthyTool.lean` in the package's default source file path.

22.1.3.2. Lean Format

The Lean format for Lake package configuration files provides a domain-specific language for the declarative features that are supported in the TOML format. Additionally, it provides the ability to write Lean code to implement any necessary build logic that is not expressible declaratively.

Because the Lean format is a Lean source file, it can be edited using all the features of the Lean language server. Additionally, Lean's metaprogramming framework allows elaboration-time side effects to be used to implement features such as configuration steps that are conditional on the current platform. However, a consequence of the Lean configuration format being a Lean file is that it is not feasible to process such files using tools that are not themselves written in Lean.

22.1.3.2.1. Declarative Fields

The declarative subset of the Lean configuration format uses sequences of declaration fields to specify configuration options.

syntax

Declarative Fields

A field assignment in a declarative configuration.

```
declField ::=
    ident := term
```

22.1.3.2.2. Packages

syntax

Package Configuration

```
command ::= ...
    | docComment?
      (@[ attrInstance,* ])?
      package identOrStr

command ::= ...
    | docComment?
      (@[attrInstance,*])?
      package identOrStr where
        declField*

command ::= ...
    | docComment?
      (@[attrInstance,*])?
      package identOrStr {
        declField;*
      }
      (where
        letRecDecl;*)?
```

There can only be one `package` declaration per Lake configuration file. The defined package configuration will be available for reference as `_package`.

syntax

Post-Update Hooks

```
command ::= ...
    | post_update simpleBinder? (declValSimple
      | declValDo)
```

Declare a post-`lake update` hook for the package. Runs the monadic action is after a successful `lake update` execution in this package or one of its downstream dependents.

Example

This feature enables Mathlib to synchronize the Lean toolchain and run `cache get` after a `lake update`:

```
lean_exe cache
post_update pkg do
  let wsToolchainFile := (← getRootPackage).dir / "lean-toolchain"
  let mathlibToolchain ← IO.FS.readFile <| pkg.dir / "lean-
toolchain"
  IO.FS.writeFile wsToolchainFile mathlibToolchain
  let exeFile ← runBuild cache.fetch
  let exitCode ← env exeFile.toString #["get"]
  if exitCode ≠ 0 then
    error s!"{pkg.name}: failed to fetch cache"
```

22.1.3.2.3. Dependencies

Dependencies are specified using the `require` declaration.

syntax

Requiring Packages

```
command ::= ...
    | docComment
      require depName (@ git? term)? fromClause? (with term)?
```

The `@` clause specifies a package version, which is used when requiring a package from Reservoir. The version may either be a string that specifies the version declared in the package's `version` field, or a specific Git revision. Git revisions may be branch names, tag names, or commit hashes.

The optional *`fromClause`* specifies a package source other than Reservoir, which may be either a Git repository or a local path.

The `with` clause specifies a `NameMap String` of Lake options that will be used to configure the dependency. This is equivalent to passing `-K` options to `lake build` when building the dependency on the command line.

syntax

Package Sources

Specifies a specific source from which to draw the package dependency. Dependencies that are downloaded from a remote source will be placed into the workspace's `packagesDir`.

Path Dependencies

```
from <path>
```

Lake loads the package located a fixed `path` relative to the requiring package's directory.

Git Dependencies

```
from git <url> [@ <rev>] [/ <subDir>]
```

Lake clones the Git repository available at the specified fixed Git `url`, and checks out the specified revision `rev`. The revision can be a commit hash, branch, or tag. If none is provided, Lake defaults to `master`. After checkout, Lake loads the package located in `subDir` (or the repository root if no subdirectory is specified).

```
fromClause ::=
    from term

fromClause ::= ...
    | from git term (@ term)? (/ term)?
```

22.1.3.2.4. Targets

Targets are typically added to the set of default targets by applying the `default_target` attribute, rather than by explicitly listing them.

attribute

Specifying Default Targets

```
attr ::= ...
    | default_target
```

Marks a target as a default, to be built when no other target is specified.

22.1.3.2.4.1. Libraries

syntax

Library Targets

To define a library in which all configurable fields have their default values, use `lean_lib` with no further fields.

```
command ::= ...
    | docComment?
      attributes?
      lean_lib identOrStr
```

The default configuration can be modified by providing the new values.

```
command ::= ...
    | docComment?
      attributes?
      lean_lib identOrStr where
        declField*

command ::= ...
    | docComment?
      attributes?
      lean_lib identOrStr {
        declField;*
      }
      (where
        letRecDecl;*)?
```

The fields of `lean_lib` are those of the `LeanLibConfig` structure.

structure

`Lake.LeanLibConfig (name : Lean.Name) : Type`

A Lean library's declarative configuration.

Constructor

`Lake.LeanLibConfig.mk`

Extends

- ☑ `Lake.LeanConfig`

Fields

`buildType : Lake.BuildType`

Inherited from `Lake.LeanConfig`

leanOptions : Array Lean.LeanOption

Inherited from Lake.LeanConfig

moreLeanArgs : Array String

Inherited from Lake.LeanConfig

weakLeanArgs : Array String

Inherited from Lake.LeanConfig

moreLeancArgs : Array String

Inherited from Lake.LeanConfig

moreServerOptions : Array Lean.LeanOption

Inherited from Lake.LeanConfig

weakLeancArgs : Array String

Inherited from Lake.LeanConfig

moreLinkObjs : Lake.TargetArray System.FilePath

Inherited from Lake.LeanConfig

moreLinkLibs : Lake.TargetArray Lake.Dynlib

Inherited from Lake.LeanConfig

moreLinkArgs : Array String

Inherited from Lake.LeanConfig

weakLinkArgs : Array String

Inherited from Lake.LeanConfig

backend : Lake.Backend

Inherited from Lake.LeanConfig

platformIndependent : Option Bool

Inherited from Lake.LeanConfig

dynlibs : Lake.TargetArray Lake.Dynlib

Inherited from Lake.LeanConfig

plugins : Lake.TargetArray Lake.Dynlib

Inherited from Lake.LeanConfig

srcDir : System.FilePath

The subdirectory of the package's source directory containing the library's Lean source files. Defaults simply to said *srcDir*.

(This will be passed to `lean` as the *-R* option.)

roots : Array Lean.Name

The root module(s) of the library. Submodules of these roots (e.g., `Lib.Foo` of `Lib`) are considered part of the library. Defaults to a single root of the target's name.

globs : Array Lake.Glob

An Array of module Globs to build for the library. Defaults to a Glob.one of each of the library's *roots*.

Submodule globs build every source file within their directory. Local imports of glob'ed files (i.e., fellow modules of the workspace) are also recursively built.

`libName : String`

The name of the library artifact. Used as a base for the file names of its static and dynamic binaries. Defaults to the mangled name of the target.

`needs : Array Lake.PartialBuildKey`

An Array of targets to build before the executable's modules.

`extraDepTargets : Array Lean.Name`

Deprecated. Use *needs* **instead.** An Array of target names to build before the library's modules.

`precompileModules : Bool`

Whether to compile each of the library's modules into a native shared library that is loaded whenever the module is imported. This speeds up evaluation of metaprograms and enables the interpreter to run functions marked `@[extern]`.

Defaults to `false`.

`defaultFacets : Array Lean.Name`

An Array of library facets to build on a bare `lake build` of the library. For example, `#[LeanLib.sharedLib]` will build the shared library facet.

`nativeFacets : Bool → Array (Lake.ModuleFacet System.FilePath)`

The module facets to build and combine into the library's static and shared libraries. If `shouldExport` is true, the module facets should export any symbols a user may expect to lookup in the library. For example, the Lean interpreter will use exported symbols in linked libraries.

Defaults to a singleton of `Module.oExportFacet` (if `shouldExport`) or `Module.oFacet`. That is, the object files compiled from the Lean sources, potentially with exported Lean symbols.

22.1.3.2.4.2. Executables

syntax

Executable Targets

To define an executable in which all configurable fields have their default values, use `lean_exe` with no further fields.

```
command ::= ...
    | docComment? attributes?
      lean_exe identOrStr
```

The default configuration can be modified by providing the new values.

```
command ::= ...
    | docComment? attributes?
      lean_exe identOrStr where
        declField*

command ::= ...
    | docComment? attributes?
      lean_exe identOrStr {
        declField;*
      }
      (where
        letRecDecl;*)?
```

The fields of `lean_exe` are those of the `LeanExeConfig` structure.

structure

`Lake.LeanExeConfig (name : Lean.Name) : Type`

A Lean executable's declarative configuration.

Constructor

`Lake.LeanExeConfig.mk`

Extends

- ☑ `Lake.LeanConfig`

Fields

`buildType : Lake.BuildType`

Inherited from `Lake.LeanConfig`

`leanOptions : Array Lean.LeanOption`

Inherited from `Lake.LeanConfig`

`moreLeanArgs : Array String`

Inherited from `Lake.LeanConfig`

`weakLeanArgs : Array String`

Inherited from `Lake.LeanConfig`

`moreLeancArgs : Array String`

Inherited from `Lake.LeanConfig`

`moreServerOptions : Array Lean.LeanOption`

Inherited from `Lake.LeanConfig`

`weakLeancArgs : Array String`

Inherited from `Lake.LeanConfig`

`moreLinkObjs : Lake.TargetArray System.FilePath`

Inherited from `Lake.LeanConfig`

`moreLinkLibs : Lake.TargetArray Lake.Dynlib`

Inherited from `Lake.LeanConfig`

`moreLinkArgs : Array String`

Inherited from `Lake.LeanConfig`

`weakLinkArgs : Array String`

Inherited from `Lake.LeanConfig`

`backend : Lake.Backend`

Inherited from `Lake.LeanConfig`

`platformIndependent : Option Bool`

Inherited from `Lake.LeanConfig`

`dynlibs : Lake.TargetArray Lake.Dynlib`

Inherited from `Lake.LeanConfig`

`plugins : Lake.TargetArray Lake.Dynlib`

Inherited from `Lake.LeanConfig`

`srcDir : System.FilePath`

The subdirectory of the package's source directory containing the executable's Lean source file. Defaults simply to said *`srcDir`*.

(This will be passed to `lean` as the *`-R`* option.)

`root : Lean.Name`

The root module of the binary executable. Should include a `main` definition that will serve as the entry point of the program.

The root is built by recursively building its local imports (i.e., fellow modules of the workspace).

Defaults to the name of the target.

`exeName : String`

The name of the binary executable. Defaults to the target name with any `.` replaced with a `-`.

`needs : Array Lake.PartialBuildKey`

An `Array` of targets to build before the executable's modules.

`extraDepTargets : Array Lean.Name`

Deprecated. Use *needs* **instead.** An `Array` of target names to build before the executable's modules.

`supportInterpreter : Bool`

Enables the executable to interpret Lean files (e.g., via `Lean.Elab.runFrontend`) by exposing symbols within the executable to the Lean interpreter.

Implementation-wise, on Windows, the Lean shared libraries are linked to the executable and, on other systems, the executable is linked with *`-rdynamic`*. This increases the size of the binary on Linux and, on Windows, requires `libInit_shared.dll` and `libleanshared.dll` to be co-located with the executable or part of `PATH` (e.g., via `lake exe`). Thus, this feature should only be enabled when necessary.

Defaults to `false`.

`nativeFacets : Bool → Array (Lake.ModuleFacet System.FilePath)`

The module facets to build and combine into the executable. If `shouldExport` is true, the module facets should export any symbols a user may expect to lookup in the executable. For example, the Lean interpreter will use exported symbols in the executable. Thus, `shouldExport` will be `true` if `supportInterpreter := true`.

Defaults to a singleton of `Module.oExportFacet` (if `shouldExport`) or `Module.oFacet`. That is, the object file compiled from the Lean source, potentially with exported Lean symbols.

22.1.3.2.4.3. External Libraries

Because external libraries may be written in any language and require arbitrary build steps, they are defined as programs written in the `FetchM` monad that produce a `Job`. External library targets should produce a build job that carries out the build and then returns the location of the resulting static library. For the external library to link properly when `precompileModules` is on, the static library produced by an **`extern_lib`** target must follow the platform's naming conventions for libraries (i.e., be named foo.a on Windows or libfoo.a on Unix-like systems). The utility function `Lake.nameToStaticLib` converts a library name into its proper file name for current platform.

syntax

External Library Targets

```
command ::= ...
    | docComment?
      attributes?
      extern_lib identOrStr simpleBinder? := term
      (where letRecDecl*)?
```

Define a new external library target for the package. Has one form:

```
extern_lib «target-name» (pkg : NPackage _package.name) :=
  /- build term of type `FetchM (Job FilePath)` -/
```

The pkg parameter (and its type specifier) is optional. It is of type `NPackage _package.name` to provably demonstrate the package provided is the package in which the target is defined.

The term should build the external library's **static** library.

22.1.3.2.4.4. Custom Targets

Custom targets may be used to define any incrementally-built artifact whatsoever, using the Lake API.

syntax

Custom Targets

```
command ::= ...
    | docComment?
      attributes?
      target identOrStr simpleBinder? : term := term
      (where letRecDecl*)?
```

Define a new external library target for the package. Has one form:

```
extern_lib «target-name» (pkg : NPackage _package.name) :=
  /- build term of type `FetchM (Job FilePath)` -/
```

The pkg parameter (and its type specifier) is optional. It is of type `NPackage _package.name` to provably demonstrate the package provided is the package in which the target is defined.

The term should build the external library's **static** library.

22.1.3.2.4.5. Custom Facets

Custom facets allow additional artifacts to be incrementally built from a module, library, or package.

syntax

Custom Package Facets

Package facets allow the production of an artifact or set of artifacts from a whole package. The Lake API makes it possible to query a package for its libraries; thus, one common use for a package facet is to build a given facet of each library.

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
```

```
package_facet identOrStr simpleBinder? : term := term
(where letRecDecl*)?
```

Define a new package facet. Has one form:

```
package_facet «facet-name» (pkg : Package) : α :=
  /- build term of type `FetchM (Job α)` -/
```

The pkg parameter (and its type specifier) is optional.

syntax

Custom Library Facets

Package facets allow the production of an artifact or set of artifacts from a library. The Lake API makes it possible to query a library for its modules; thus, one common use for a library facet is to build a given facet of each module.

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
      library_facet identOrStr simpleBinder? : term := term
      (where letRecDecl*)?
```

Define a new library facet. Has one form:

```
library_facet «facet-name» (lib : LeanLib) : α :=
  /- build term of type `FetchM (Job α)` -/
```

The `lib` parameter (and its type specifier) is optional.

syntax

Custom Module Facets

Package facets allow the production of an artifact or set of artifacts from a module, typically by invoking a command-line tool.

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
      module_facet identOrStr simpleBinder? : term := term
      (where letRecDecl*)?
```

Define a new module facet. Has one form:

```
module_facet «facet-name» (mod : Module) : α :=
  /- build term of type `FetchM (Job α)` -/
```

The mod parameter (and its type specifier) is optional.

22.1.3.2.5. Configuration Value Types

inductive type

`Lake.BuildType : Type`

Lake equivalent of CMake's CMAKE_BUILD_TYPE.

Constructors

| `debug : Lake.BuildType`

Debug optimization, asserts enabled, custom debug code enabled, and debug info included in executable (so you can step through the code with a debugger and have address to source-file:line-number translation). For example, passes *-Og -g* when compiling C code.

| `relWithDebInfo : Lake.BuildType`

Optimized, *with* debug info, but no debug code or asserts (e.g., passes *-O3 -g -DNDEBUG* when compiling C code).

| `minSizeRel : Lake.BuildType`

Same as `release` but optimizing for size rather than speed (e.g., passes *-Os -DNDEBUG* when compiling C code).

| `release : Lake.BuildType`

High optimization level and no debug info, code, or asserts (e.g., passes *-O3 -DNDEBUG* when compiling C code).

In Lake's DSL, *globs* are patterns that match sets of module names. There is a coercion from names to globs that match the name in question, and there are two postfix operators for constructing further globs.

syntax

Glob Syntax

The glob pattern `N.*` matches N or any submodule for which N is a prefix.

```
term ::= ...
    | name.*
```

The glob pattern `N.*` matches any submodule for which N is a strict prefix, but not N itself.

```
term ::= ...
    | name.+
```

Whitespace is not permitted between the name and `.*` or `.+`.

inductive type

`Lake.Glob : Type`

A specification of a set of module names.

Constructors

`| one : Lean.Name → Lake.Glob`

Selects just the specified module name.

`| submodules : Lean.Name → Lake.Glob`

Selects all submodules of the specified module, but not the module itself.

`| andSubmodules : Lean.Name → Lake.Glob`

Selects the specified module and all submodules.

structure

`Lean.LeanOption : Type`

An option that is used by Lean as if it was passed using *-D*.

Constructor

`Lean.LeanOption.mk`

Fields

`name : Lean.Name`

The option's name.

`value : Lean.LeanOptionValue`

The option's value.

inductive type

`Lake.Backend : Type`

Compiler backend with which to compile Lean.

Constructors

| `c : Lake.Backend`

Force the C backend.

| `llvm : Lake.Backend`

Force the LLVM backend.

| `default : Lake.Backend`

Use the default backend. Can be overridden by more specific configuration.

22.1.3.2.6. Scripts

Lake scripts are used to automate tasks that require access to a package configuration but do not participate in incremental builds of artifacts from code. Scripts run in the `ScriptM` monad, which is `IO` with an additional reader monad transformer that provides access to the package configuration. In particular, a script should have the type `List String → ScriptM UInt32`. Workspace information in scripts is primarily accessed via the `MonadWorkspace ScriptM` instance.

syntax

Script Declarations

```
command ::= ...
    | docComment?
      (@[attrInstance,*])?
      script identOrStr simpleBinder? :=
        term
      (where
        letRecDecl*)?
```

Define a new Lake script for the package.

Example

```
/-- Display a greeting -/
script «script-name» (args) do
  if h : 0 < args.length then
    IO.println s!"Hello, {args[0]'h}!"
  else
    IO.println "Hello, world!"
  return 0
```

def

```
Lake.ScriptM (α : Type) : Type
```

The type of a `Script`'s monad. `IO` equipped information about the Lake configuration.

attribute

Default Scripts

```
attr ::= ...
    | default_script
```

Marks a Lake script as the package's default.

22.1.3.2.7. Utilities

syntax

The Current Directory

```
term ::= ...
    | __dir__
```

A macro that expands to the path of package's directory during the Lakefile's elaboration.

syntax

Configuration Options

```
term ::= ...
    | get_config? ident
```

A macro that expands to the specified configuration option (or none, if the option has not been set) during the Lakefile's elaboration.

Configuration arguments are set either via the Lake CLI (by the `-K` option) or via the **`with`** clause in a **`require`** statement.

syntax

Compile-Time Conditionals

```
command ::= ...
    | meta if term then
        cmdDo
      (else cmdDo)?
```

The **meta if** command has two forms:

```
meta if <c:term> then <a:command>
meta if <c:term> then <a:command> else <b:command>
```

It expands to the command a if the term c evaluates to true (at elaboration time). Otherwise, it expands to command b (if an **else** clause is provided).

One can use this command to specify, for example, external library targets only available on specific platforms:

```
meta if System.Platform.isWindows then
extern_lib winOnlyLib := ...
else meta if System.Platform.isOSX then
extern_lib macOnlyLib := ...
else
extern_lib linuxOnlyLib := ...
```

syntax

Command Sequences

```
cmdDo ::= ...
    | command

cmdDo ::= ...
    | do
        command
        command*
```

The **do** command syntax groups multiple similarly indented commands together. The group can then be passed to another command that usually only accepts a single command (e.g., **meta if**).

syntax

Compile-Time Side Effects

```
term ::= ...
    | run_io doSeq
```

Executes a term of type `IO α` at elaboration-time and produces an expression corresponding to the result via `ToExpr α`.

22.1.4. Script API Reference

In addition to ordinary `IO` effects, Lake scripts have access to the Lake environment (which provides information about the current toolchain, such as the location of the Lean compiler) and the current workspace. This access is provided in `ScriptM`.

def

```
Lake.ScriptM (α : Type) : Type
```

The type of a `Script`'s monad. `IO` equipped information about the Lake configuration.

22.1.4.1. Accessing the Environment

Monads that provide access to information about the current Lake environment (such as the locations of Lean, Lake, and other tools) have `MonadLakeEnv` instances. This is true for all of the monads in the Lake API, including `ScriptM`.

def

```
Lake.MonadLakeEnv.{u} (m : Type → Type u) : Type u
```

A monad equipped with a (read-only) detected environment for Lake.

def

```
Lake.getLakeEnv.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m] :
  m Lake.Env
```

Gets the current Lake environment.

def

```
Lake.getNoCache.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] [Lake.MonadBuild m] : m Bool
```

Get the LAKE_NO_CACHE/ Lake configuration.

def

```
Lake.getTryCache.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] [Lake.MonadBuild m] : m Bool
```

Get whether the LAKE_NO_CACHE/ Lake configuration is **NOT** set.

def

```
Lake.getPkgUrlMap.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m (Lean.NameMap String)
```

Get the LAKE_PACKAGE_URL_MAP for the Lake environment. Empty if none.

def

```
Lake.getElanToolchain.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m String
```

Get the name of Elan toolchain for the Lake environment. Empty if none.

22.1.4.1.1. Search Path Helpers

def

```
Lake.getEnvLeanPath.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m System.SearchPath
```

Get the detected LEAN_PATH value of the Lake environment.

def

```
Lake.getEnvLeanSrcPath.{u_1} {m : Type → Type u_1}
[Lake.MonadLakeEnv m]
  [Functor m] : m System.SearchPath
```

Get the detected LEAN_SRC_PATH value of the Lake environment.

def

```
Lake.getEnvSharedLibPath.{u_1} {m : Type → Type u_1}
  [Lake.MonadLakeEnv m] [Functor m] : m System.SearchPath
```

Get the detected sharedLibPathEnvVar value of the Lake environment.

22.1.4.1.2. Elan Install Helpers

def

```
Lake.getElanInstall?.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m (Option Lake.ElanInstall)
```

Get the detected Elan installation (if one).

def

```
Lake.getElanHome?.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m (Option System.FilePath)
```

Get the root directory of the detected Elan installation (i.e., ELAN_HOME).

def

```
Lake.getElan?.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m (Option System.FilePath)
```

Get the path of the elan binary in the detected Elan installation.

22.1.4.1.3. Lean Install Helpers

def

```
Lake.getLeanInstall.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m Lake.LeanInstall
```

Get the detected Lean installation.

def

```
Lake.getLeanSysroot.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m System.FilePath
```

Get the root directory of the detected Lean installation.

def

```
Lake.getLeanSrcDir.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the Lean source directory of the detected Lean installation.

def

```
Lake.getLeanLibDir.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the Lean library directory of the detected Lean installation.

def

```
Lake.getLeanIncludeDir.{u_1} {m : Type → Type u_1}
[Lake.MonadLakeEnv m]
```

```
  [Functor m] : m System.FilePath
```

Get the C include directory of the detected Lean installation.

def

```
Lake.getLeanSystemLibDir.{u_1} {m : Type → Type u_1}
  [Lake.MonadLakeEnv m] [Functor m] : m System.FilePath
```

Get the system library directory of the detected Lean installation.

def

```
Lake.getLean.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the path of the `lean` binary in the detected Lean installation.

def

```
Lake.getLeanc.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the path of the `leanc` binary in the detected Lean installation.

def

```
Lake.getLeanSharedLib.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m System.FilePath
```

Get the path of the `libleanshared` library in the detected Lean installation.

def

```
Lake.getLeanAr.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the path of the `ar` binary in the detected Lean installation.

def

```
Lake.getLeanCc.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the path of C compiler in the detected Lean installation.

def

```
Lake.getLeanCc?.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m (Option String)
```

Get the optional LEAN_CC compiler override of the detected Lean installation.

22.1.4.1.4. Lake Install Helpers

def

```
Lake.getLakeInstall.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv
m]
  [Functor m] : m Lake.LakeInstall
```

Get the detected Lake installation.

def

```
Lake.getLakeHome.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the root directory of the detected Lake installation (e.g., LAKE_HOME).

def

```
Lake.getLakeSrcDir.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the source directory of the detected Lake installation.

def

```
Lake.getLakeLibDir.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the Lean library directory of the detected Lake installation.

def

```
Lake.getLake.{u_1} {m : Type → Type u_1} [Lake.MonadLakeEnv m]
  [Functor m] : m System.FilePath
```

Get the path of the `lake` binary in the detected Lake installation.

22.1.4.2. Accessing the Workspace

Monads that provide access to information about the current Lake workspace have `MonadWorkspace` instances. In particular, there are instances for `ScriptM` and `LakeM`.

type class

```
Lake.MonadWorkspace.{u} (m : Type → Type u) : Type u
```

A monad equipped with a (read-only) Lake `Workspace`.

Instance Constructor

`Lake.MonadWorkspace.mk.{u}`

Methods

`getWorkspace : m Lake.Workspace`

Gets the current Lake workspace.

def

```
Lake.getRootPackage.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace
m]
  [Functor m] : m Lake.Package
```

Get the root package of the context's workspace.

def

```
Lake.findPackage?.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace
m]
  [Functor m] (name : Lean.Name) : m (Option (Lake.NPackage name))
```

Try to find a package within the workspace with the given name.

def

```
Lake.findModule?.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace m]
  [Functor m] (name : Lean.Name) : m (Option Lake.Module)
```

Locate the named, buildable, importable, local module in the workspace.

def

```
Lake.findLeanExe?.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace
m]
  [Functor m] (name : Lean.Name) : m (Option Lake.LeanExe)
```

Try to find a Lean executable in the workspace with the given name.

def

```
Lake.findLeanLib?.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace
m]
  [Functor m] (name : Lean.Name) : m (Option Lake.LeanLib)
```

Try to find a Lean library in the workspace with the given name.

def

```
Lake.findExternLib?.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace
m]
  [Functor m] (name : Lean.Name) : m (Option Lake.ExternLib)
```

Try to find an external library in the workspace with the given name.

def

```
Lake.getLeanPath.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace m]
  [Functor m] : m System.SearchPath
```

Get the paths added to `LEAN_PATH` by the context's workspace.

def

```
Lake.getLeanSrcPath.{u_1} {m : Type → Type u_1} [Lake.MonadWorkspace
m]
  [Functor m] : m System.SearchPath
```

Get the paths added to `LEAN_SRC_PATH` by the context's workspace.

def

```
Lake.getSharedLibPath.{u_1} {m : Type → Type u_1}
  [Lake.MonadWorkspace m] [Functor m] : m System.SearchPath
```

Get the paths added to the shared library path by the context's workspace.

def

```
Lake.getAugmentedLeanPath.{u_1} {m : Type → Type u_1}
  [Lake.MonadWorkspace m] [Functor m] : m System.SearchPath
```

Get the augmented `LEAN_PATH` set by the context's workspace.

def

```
Lake.getAugmentedLeanSrcPath.{u_1} {m : Type → Type u_1}
  [Lake.MonadWorkspace m] [Functor m] : m System.SearchPath
```

Get the augmented LEAN_SRC_PATH set by the context's workspace.

def

```
Lake.getAugmentedSharedLibPath.{u_1} {m : Type → Type u_1}
  [Lake.MonadWorkspace m] [Functor m] : m System.SearchPath
```

Get the augmented shared library path set by the context's workspace.

def

```
Lake.getAugmentedEnv.{u_1} {m : Type → Type u_1}
[Lake.MonadWorkspace m]
  [Functor m] : m (Array (String × Option String))
```

Get the augmented environment variables set by the context's workspace.

22.2. Managing Toolchains with Elan

Elan is the Lean toolchain manager. It is responsible both for installing toolchains and for running their constituent programs. Elan makes it possible to seamlessly work on a variety of projects, each of which is designed to be built with a particular version of Lean, without having to manually install and select toolchain versions. Each project is typically configured to use a particular version, which is transparently installed as needed, and changes to the Lean version are tracked automatically.

22.2.1. Selecting Toolchains

When using Elan, the version of each tool on the PATH is a proxy that invokes the correct version. The proxy determines the appropriate toolchain version for the current context, ensures that it is installed, and then invokes the underlying tool in the appropriate toolchain installation. These proxies can be instructed to use a specific version by passing it as an argument prefixed with +, so `lake +4.0.0` invokes `lake` version `4.0.0`, after installing it if necessary.

22.2.1.1. Toolchain Identifiers

Toolchains are specified by providing a toolchain identifier that is either a *channel*, which identifies a particular type of Lean release, and optionally an origin, or a *custom toolchain name* established by `elan toolchain link`. Channels may be:

`stable`
: The latest stable Lean release. Elan automatically tracks stable releases and offers to upgrade when a new one is released.

`beta`
: The latest release candidate. Release candidates are builds of Lean that are intended to become the next stable release. They are made available for widespread user testing.

`nightly`

The latest nightly build. Nightly builds are useful for experimenting with new Lean features to provide feedback to the developers.

A version number or specific nightly release

Each Lean version number identifies a channel that contains only that release. The version number may optionally be preceded with a `v`, so `v4.17.0` and `4.17.0` are equivalent. Similarly, `nightly-YYYY-MM-DD` specifies the nightly release from the specified date. A project's toolchain file should typically contain a specific version of Lean, rather than a general channel, to make it easier to coordinate between developers and to build and test older versions of the project. An archive of Lean releases and nightly builds is maintained.

A custom local toolchain

The command `elan toolchain link` can be used to establish a custom toolchain name in Elan for a local build of Lean. This is especially useful when working on the Lean compiler itself.

Specifying an *origin* instructs Elan to install Lean toolchains from a particular source. By default, this is the official project repository on GitHub, identified as `leanprover/lean4`. If specified, an origin should precede the channel, with a colon, so `stable` is equivalent to `leanprover/lean4:stable`. When installing nightly releases, `-nightly` is appended to the origin, so `leanprover/lean4:nightly-2025-03-25` consults the `leanprover/lean4-nightly` repository to download releases. Origins are not used for custom toolchain names.

22.2.1.2. Determining the Current Toolchain

Elan associates toolchains with directories, and uses the toolchain of the most recent parent directory of the current working directory that has a configured toolchain. A directory's toolchain may result from a toolchain file or from an override configured with `elan override`.

The current toolchain is determined by first searching for a configured toolchain for the current directory, walking up through parent directories until a toolchain version is found or there are no more parents. A directory has a configured toolchain if there is a configured toolchain override for the directory or if it contains a `lean-toolchain` file. More recent parents take precedence over their ancestors, and if a directory has both an override and a toolchain file, then the override takes precedence. If no directory toolchain is found, then Elan's configured *default toolchain* is used as a fallback.

The most common way to configure a Lean toolchain is with a *toolchain file*. The toolchain file is a text file named `lean-toolchain` that contains a single line with a valid toolchain identifier. This file is typically located in the root directory of a project and checked in to version control with the code, ensuring that everyone working on the project uses the same version. Updating to a new Lean toolchain requires only editing this file, and the new version is automatically downloaded and run the next time a Lean file is opened or built.

In certain advanced use cases where more flexibility is required, a *toolchain override* can be configured. Like toolchain files, overrides associate a toolchain version with a directory and its children. Unlike toolchain files, overrides are stored in Elan's configuration rather than in a local file. They are typically used when a specific local configuration is required that does not make sense for other developers, such as testing a project with a locally-built Lean compiler.

22.2.2. Toolchain Locations

By default, Elan stores installed toolchains in `.elan/toolchains` in the user's home directory, and its proxies are kept in `.elan/bin`, which is added to the path when Elan is installed. The environment variable `ELAN_HOME` can be used to change this location. It should be set both prior to installing Elan and in all sessions that use Lean in order to ensure that Elan's files are found.

22.2.3. Command-Line Interface

In addition to the proxies that automatically select, install, and invoke the correct versions of Lean tools, Elan provides a command-line interface for querying and configuring its settings. This tool is called `elan`. Like Lake, its command-line interface is structured around subcommands.

Elan can be invoked with following flags:

`--help` or `-h`
: Describes the current subcommand in detail.

`--verbose` or `-v`
: Enables verbose output.

`--version` or `-V`
: Displays the Elan version.

22.2.3.1. Querying Toolchains

The `elan show` command displays the current toolchain (as determined by the current directory) and lists all installed toolchains.

Elan command

`elan show`

Shows the name of the active toolchain and the version of `lean`.

If there are multiple toolchains installed, then they are all listed.

Here is typical output from `elan show` in a project with a `lean-toolchain` file:

```
installed toolchains
--------------------
```

```
leanprover/lean4:nightly-2025-03-25
leanprover/lean4:v4.17.0  (resolved from default 'stable')
leanprover/lean4:v4.16.0
leanprover/lean4:v4.9.0

active toolchain
----------------

leanprover/lean4:v4.9.0 (overridden by '/PATH/TO/PROJECT/lean-
toolchain')
Lean (version 4.9.0, arm64-apple-darwin23.5.0, commit 8f9843a4a5fe,
Release)
```

The `installed toolchains` section lists all the toolchains currently available on the system. The `active toolchain` section identifies the current toolchain and describes how it was selected. In this case, the toolchain was selected due to a `lean-toolchain` file.

22.2.3.2. Setting the Default Toolchain

Elan's configuration file specifies a default toolchain to be used when there is no `lean-toolchain` file or toolchain override for the current directory. Rather than manually editing the file, this value is typically changed using the `elan default` command.

Elan command

`elan default` *`toolchain`*

Sets the default toolchain to *`toolchain`*, which should be a valid toolchain identifier such as `stable`, `nightly`, or `4.17.0`.

22.2.3.3. Managing Installed Toolchains

The `elan toolchain` family of subcommands is used to manage the installed toolchains. Toolchains are stored in Elan's toolchain directory.

Installed toolchains can take up substantial disk space. Elan tracks the Lean projects in which it is invoked, saving a list. This list of projects can be used to determine which toolchains are in active use and automatically delete unused toolchain versions with `elan toolchain gc`.

Elan command

`elan toolchain list`

Lists the currently-installed toolchains. This is a subset of the output of `elan show`.

Elan command

`elan toolchain install` *`toolchain`*

Installs the indicated *toolchain*. The toolchain's name should be an identifier that's suitable for inclusion in a `lean-toolchain` file.

Elan command

elan toolchain uninstall *toolchain*

Uninstalls the indicated *toolchain*. The toolchain's name should the name of an installed toolchain. Use `elan toolchain list` to see the installed toolchains with their names.

Elan command

elan toolchain link *local-name path*

Creates a new local toolchain named *local-name*, using the Lean toolchain found at *path*.

Elan command

elan toolchain gc [--delete] [--json]

This command is still considered experimental.

Determines which of the installed toolchains are in use, offering to delete those that are not. All the installed toolchains are listed, separated into those that are in use and those that are not.

A toolchain is classified as “in use” if

- it is the default toolchain,
- it is registered as an override, or
- there is a directory with a `lean-toolchain` file referencing the toolchain and elan has been used in the directory before.

For safety reasons, `elan toolchain gc` will not actually delete any toolchains unless the `--delete` flag is passed. This may be relaxed in the future when the implementation is deemed sufficiently mature. The `--json` flag causes `elan toolchain gc` to emit the list of used and unused toolchains in a JSON format that's suitable for other tools.

22.2.3.4. Managing Directory Overrides

Directory-specific toolchain overrides are a local configuration that takes precedence over `lean-toolchain` files. The `elan override` commands manage overrides.

Elan command

elan override list

Lists all the currently configured directory overrides in two columns. The left column contains the directories in which the Lean version is overridden, and the right column lists the toolchain version.

Elan command

`elan override set` *`toolchain`*

Sets *`toolchain`* as an override for the current directory.

Elan command

`elan override unset` `[--nonexistent]` `[--path` *`path`*`]`

If `--nonexistent` flag is provided, all overrides that are configured for directories that don't currently exist are removed. If `--path` is provided, then the override set for *`path`* is removed. Otherwise, the override for the current directory is removed.

22.2.3.5. Running Tools and Commands

The commands in this section provide the ability to run a command in a specific toolchain and to locate a tool from a particular toolchain on disk. This can be useful when experimenting with different Lean versions, for cross-version testing, and for integrating Elan with other tools.

Elan command

`elan run` `[--install]` *`toolchain command ...`*

Configures an environment to use the given toolchain and then runs the specified program. The toolchain will be installed if the `--install` flag is provided. The command may be any program; it does not need to be a command that's part of a toolchain such as `lean` or `lake`. This can be used for testing arbitrary toolchains without setting an override.

Elan command

`elan which` *`command`*

Displays the full path to the toolchain-specific binary for *`command`*.

22.2.3.6. Managing Elan

Elan can manage its own installation. It can upgrade itself, remove itself, and help configure tab completion for many popular shells.

Elan command

`elan self update`

Downloads and installs updates to Elan itself.

Elan command

`elan self uninstall`

Uninstalls Elan.

Elan command

`elan completions` *`shell`*

Generates shell completion scripts for Elan, enabling tab completion for Elan commands in a variety of shells. See the output of `elan help completions` for a description of how to install them.

Error Explanations

This section provides explanations of errors and warnings that may be generated by Lean when processing a source file. All error names listed below have the `lean` package prefix.

Name	Summary	Severity	Since
`ctorResultingTypeMismatch`	Resulting type of constructor was not the inductive type being declared.	Error	4.22.0
`dependsOnNoncomputable`	Declaration depends on noncomputable definitions but is not marked as noncomputable	Error	4.22.0
`inductiveParamMismatch`	Invalid parameter in an occurrence of an inductive type in one of its constructors.	Error	4.22.0
`inductiveParamMissing`	Parameter not present in an occurrence of an inductive type in one of its constructors.	Error	4.22.0
`invalidDottedIdent`	Dotted identifier notation used with invalid or uninferrable expected type.	Error	4.22.0
`redundantMatchAlt`	Match alternative will never be reached.	Error	4.22.0

`lean.ctorResultingTypeMismatch`

Resulting type of constructor was not the inductive type being declared.

Severity: Error | **Since:** 4.22.0

In an inductive declaration, the resulting type of each constructor must match the type being declared; if it does not, this error is raised. That is, every constructor of an inductive type must return a value of that type. See the Inductive Types manual section for additional details. Note that it is possible to omit the resulting type for a constructor if the inductive type being defined has no indices.

Examples

▼ *Example: Typo in resulting type*

Original Fixed

```
inductive Tree (α : Type) where
  | leaf : Tree α
  | node : α → Tree α → Treee α
```

```
Unexpected resulting type for constructor 'Tree.node':
Expected an application of
  Tree
but found
  ?m.22
```

▼ *Example: Missing resulting type after constructor parameter*

Original Fixed (resulting type) Fixed (named parameter)

```
inductive Credential where
  | pin      : Nat
  | password : String
```

```
Unexpected resulting type for constructor 'Credential.pin':
Expected
  Credential
but found
  Nat
```

If the type of a constructor is annotated, the full type—including the resulting type—must be provided. Alternatively, constructor parameters can be written using named binders; this allows the omission of the constructor's resulting type because it contains no indices.

`lean.dependsOnNoncomputable`

Declaration depends on noncomputable definitions but is not marked as noncomputable

Severity: Error | **Since:** 4.22.0

This error indicates that the specified definition depends on one or more definitions that do not contain executable code and is therefore required to be marked as **`noncomputable`**. Such definitions can be type-checked but do not contain code that can be executed by Lean.

If you intended for the definition named in the error message to be noncomputable, marking it as **`noncomputable`** will resolve this error. If you did not, inspect the noncomputable definitions on which it depends: they may be noncomputable because they failed to compile, are **`axiom`**s, or were themselves marked as **`noncomputable`**. Making all of your definition's noncomputable dependencies computable will also resolve this error. See the manual section on Modifiers for more information about noncomputable definitions.

Examples

▼ *Example: Necessarily noncomputable function not appropriately marked*

Original Fixed

```
axiom transform : Nat → Nat

def transformIfZero : Nat → Nat
  | 0 => transform 0
  | n => n
```

```
axiom 'transform' not supported by code generator; consider
marking definition as 'noncomputable'
```

In this example, `transformIfZero` depends on the axiom `transform`. Because `transform` is an axiom, it does not contain any executable code; although the value `transform 0` has type `Nat`, there is no way to compute its value. Thus, `transformIfZero` must be marked **`noncomputable`** because its execution would depend on this axiom.

▼ *Example: Noncomputable dependency can be made computable*

Original Fixed (computable)

```
noncomputable def getOrDefault [Nonempty α] : Option α → α
  | some x => x
  | none => Classical.ofNonempty
```

```
def endsOrDefault (ns : List Nat) : Nat × Nat :=
  let head := getOrDefault ns.head?
  let tail := getOrDefault ns.getLast?
  (head, tail)
```

```
failed to compile definition, consider marking it as
'noncomputable' because it depends on 'getOrDefault', which
is 'noncomputable'
```

The original definition of `getOrDefault` is noncomputable due to its use of `Classical.choice`. Unlike in the preceding example, however, it is possible to implement a similar but computable version of `getOrDefault` (using the `Inhabited` type class), allowing `endsOrDefault` to be computable. (The differences between `Inhabited` and `Nonempty` are described in the documentation of inhabited types in the manual section on Basic Classes.)

▼ *Example: Noncomputable instance in namespace*

Original Fixed

```
open Classical in
/--
Returns `y` if it is in the image of `f`,
or an element of the image of `f` otherwise.
-/
def fromImage (f : Nat → Nat) (y : Nat) :=
  if ∃ x, f x = y then
    y
  else
    f 0
```

```
failed to compile definition, consider marking it as
'noncomputable' because it depends on
'Classical.propDecidable', which is 'noncomputable'
```

The `Classical` namespace contains `Decidable` instances that are not computable. These are a common source of noncomputable dependencies that do not explicitly appear in the source code of a definition. In the above example, for instance, a `Decidable` instance for the proposition `∃ x, f x = y` is synthesized using a `Classical` decidability instance; therefore, `fromImage` must be marked **`noncomputable`**.

`lean.inductiveParamMismatch`

Invalid parameter in an occurrence of an inductive type in one of its constructors.

Severity: Error | **Since:** 4.22.0

This error occurs when a parameter of an inductive type is not uniform in an inductive declaration. The parameters of an inductive type (i.e., those that appear before the colon following the **`inductive`** keyword) must be identical in all occurrences of the type being defined in its constructors' types. If a parameter of an inductive type must vary between constructors, make the parameter an index by moving it to the right of the colon. See the manual section on Inductive Types for additional details.

Note that auto-implicit inlay hints always appear left of the colon in an inductive declaration (i.e., as parameters), even when they are actually indices. This means that double-clicking on an inlay hint to insert such parameters may result in this error. If it does, change the inserted parameters to indices.

Examples

▼ *Example: Vector length index as a parameter*

Original Fixed

```
inductive Vec (α : Type) (n : Nat) : Type where
  | nil  : Vec α 0
  | cons : α → Vec α n → Vec α (n + 1)
```

```
Mismatched inductive type parameter in
  Vec α 0
The provided argument
  0
is not definitionally equal to the expected parameter
  n

Note: The value of parameter 'n' must be fixed throughout the
inductive declaration. Consider making this parameter an
index if it must vary.
```

The length argument `n` of the `Vec` type constructor is declared as a parameter, but other values for this argument appear in the `nil` and `cons` constructors (namely, `0` and `n + 1`). An error therefore appears at the first occurrence of such an argument. To correct this, `n` cannot be a parameter of the inductive declaration and must instead be an index, as in the corrected example. On the other hand, `α` remains unchanged throughout all occurrences of `Vec` in the declaration and so is a valid parameter.

`lean.inductiveParamMissing`

Parameter not present in an occurrence of an inductive type in one of its constructors.

Severity: Error | **Since:** 4.22.0

This error occurs when an inductive type constructor is partially applied in the type of one of its constructors such that one or more parameters of the type are omitted. The elaborator requires that all parameters of an inductive type be specified everywhere that type is referenced in its definition, including in the types of its constructors.

If it is necessary to allow the type constructor to be partially applied, without specifying a given type parameter, that parameter must be converted to an index. See the manual section on Inductive Types for further explanation of the difference between indices and parameters.

Examples

▼ *Example: Omitting parameter in argument to higher-order predicate*

Original Fixed

```
inductive List.All {α : Type u} (P : α → Prop) : List α → Prop
  | nil : All P []
  | cons {x xs} : P x → All P xs → All P (x :: xs)

structure RoseTree (α : Type u) where
  val : α
  children : List (RoseTree α)

inductive RoseTree.All {α : Type u} (P : α → Prop) (t : RoseTree
α) : Prop
  | intro : P t.val → List.All (All P) t.children → All P t
```

```
Missing parameter(s) in occurrence of inductive type: In the
expression
  List.All (All P) t.children
found
  All P
but expected all parameters to be specified:
  All P t

Note: All occurrences of an inductive type in the types of
its constructors must specify its fixed parameters. Only
indices can be omitted in a partial application of the type
constructor.
```

Because the `RoseTree.All` type constructor must be partially applied in the argument to `List.All`, the unspecified argument (`t`) must not be a parameter of the `RoseTree.All` predicate. Making it an index to the right of the colon in the header of `RoseTree.All` allows this partial application to succeed.

`lean.invalidDottedIdent`

Dotted identifier notation used with invalid or uninferrable expected type.

Severity: Error | **Since:** 4.22.0

This error indicates that dotted identifier notation was used in an invalid or unsupported context. Dotted identifier notation allows an identifier's namespace to be omitted, provided that it can be inferred by Lean based on type information. Details about this notation can be found in the manual section on identifiers.

This notation can only be used in a term whose type Lean is able to infer. If there is insufficient type information for Lean to do so, this error will be raised. The inferred type must not be a type universe (e.g., `Prop` or `Type`), as dotted-identifier notation is not supported on these types.

Examples

▼ *Example: Insufficient type information*

Original Fixed

```
def reverseDuplicate (xs : List α) :=
  .reverse (xs ++ xs)
```

```
Invalid dotted identifier notation: The expected type of
`.reverse` could not be determined
```

Because the return type of `reverseDuplicate` is not specified, the expected type of `.reverse` cannot be determined. Lean will not use the type of the argument `xs ++ xs` to infer the omitted namespace. Adding the return type `List α` allows Lean to infer the type of `.reverse` and thus the appropriate namespace (`List`) in which to resolve this identifier.

Note that this means that changing the return type of `reverseDuplicate` changes how `.reverse` resolves: if the return type is `T`, then Lean will (attempt to) resolve `.reverse` to a function `T.reverse` whose return type is `T`—even if `T.reverse` does not take an argument of type `List α`.

▼ *Example: Dotted identifier where type universe expected*

Original Fixed

```
example (n : Nat) :=
  match n > 42 with
  | .true  => n - 1
  | .false => n + 1
```

```
Invalid dotted identifier notation: Not supported on type
universe
  Prop
```

The proposition `n > 42` has type `Prop`, which, being a type universe, does not support dotted-identifier notation. As this example demonstrates, attempting to use this notation in such a context is almost always an error. The intent in this example was for `.true` and `.false` to be Booleans, not propositions; however, **`match`** expressions do not automatically perform this coercion for decidable propositions. Explicitly adding `decide` makes the discriminant a `Bool` and allows the dotted-identifier resolution to succeed.

`lean.redundantMatchAlt`

Match alternative will never be reached.

Severity: Error | **Since:** 4.22.0

This error occurs when an alternative in a pattern match can never be reached: any values that would match the provided patterns would also match some preceding alternative. Refer to the Pattern Matching manual section for additional details about pattern matching.

This error may appear in any pattern matching expression, including **`match`** expressions, equational function definitions, **`if let`** bindings, and monadic **`let`** bindings with fallback clauses.

In pattern-matches with multiple arms, this error may occur if a less-specific pattern precedes a more-specific one that it subsumes. Bear in mind that expressions are matched against patterns from top to bottom, so specific patterns should precede generic ones.

In **`if let`** bindings and monadic **`let`** bindings with fallback clauses, in which only one pattern is specified, this error indicates that the specified pattern will always be matched. In this case, the binding in question can be replaced with a standard pattern-matching **`let`**.

One common cause of this error is that a pattern that was intended to match a constructor was instead interpreted as a variable binding. This occurs, for instance, if a constructor name (e.g., `cons`) is written without its prefix (`List`) outside of that type's namespace. The constructor-name-as-variable linter, enabled by default, will display a warning on any variable patterns that resemble constructor names.

This error nearly always indicates an issue with the code where it appears. If needed, however, **`set_option`** `match.ignoreUnusedAlts true` will disable the check for this error and allow pattern matches with redundant alternatives to be compiled by discarding the unused arms.

Examples

▼ *Example: Incorrect ordering of pattern matches*

Original Fixed

```
def seconds : List (List α) → List α
  | [] => []
  | _ :: xss => seconds xss
  | (_ :: x :: _) :: xss => x :: seconds xss
```

```
Redundant alternative: Any expression matching
  (head✝ :: x :: tail✝) :: xss
will match one of the preceding alternatives
```

Since any expression matching `(_ :: x :: _) :: xss` will also match `_ :: xss`, the last alternative in the broken implementation is never reached. We resolve this by moving the more specific alternative before the more general one.

▼ *Example: Unnecessary fallback clause*

Original Fixed

```
example (p : Nat × Nat) : IO Nat := do
  let (m, n) := p
    | return 0
  return m + n
```

```
Redundant alternative: Any expression matching
  x✝
will match one of the preceding alternatives
```

Here, the fallback clause serves as a catch-all for all values of p that do not match `(m, n)`. However, no such values exist, so the fallback clause is unnecessary and can be removed. A similar error arises when using `if let pat := e` when e will always match `pat`.

▼ *Example: Pattern treated as variable, not constructor*

Original Fixed

```
example (xs : List Nat) : Bool :=
  match xs with
  | nil => false
  | _ => true
```

```
Redundant alternative: Any expression matching
  x✝
will match one of the preceding alternatives
```

In the original example, `nil` is treated as a variable, not as a constructor name, since this definition is not within the `List` namespace. Thus, all values of `xs` will match the first pattern, rendering the second unused. Notice that the constructor-name-as-variable linter displays a warning at `nil`, indicating its similarity to a valid constructor name. Using dot-prefix notation, as shown in the fixed example, or specifying the full constructor name `List.nil` achieves the intended behavior.

The Module System

The module system is an experimental feature that allows for more fine-grained control over what information is exported from, and imported into, modules.

The main benefits of doing so are:

Average build times
: Changes to files that affect only non-exported information (e.g. proofs) will not trigger rebuilds outside of these files. Even when dependent files have to be rebuilt, those files that cannot be affected according to the `import` annotations can be skipped.

API evolution
: Library authors can trust that changes to non-exported information will not affect downstream users of their library.

Avoiding accidental unfolding
: Limiting the scope in which definitions can be unfolded allows for avoiding both reductions that should be replaced by application of more specific theorems as well as unproductive reductions that were not in fact necessary.

Smaller executables
: Separating compile-time and run-time code allows for more aggressive dead code elimination.

The module system is activated by prefixing a Lean file with the `module` keyword. `module`s can only import other `module`s so adoption has to be done top-down. Non-`module`s can import `module`s and will ignore all module-system-specific annotations.

At the time of writing, the module system is considered experimental and additionally guarded by the `experimental.module` option that has to be set to `true` in the project's Lake configuration file. Of libraries shipped with Lean, only `Init` is currently fully ported. The language semantics described below are close to final, but not all benefits described above are implemented yet.

Visibility

The main distinction the module system introduces is between the *public scope* that contains all information visible to other modules via `import` and the *private scope* that is not imported by default. Both declarations and imports themselves are scoped in this way.

The default scope is private. The new modifier `public` before a declaration or import puts it into the public scope instead. No information from the private scope can be used in the public scope to ensure information in the latter still makes sense when only it is imported into other modules.

```
module

def priv : Nat := 0

public abbrev pub : Nat := priv  -- error: unknown identifier `priv`
```

`public section` can be used to switch the default scope for declarations, with `private` locally negating it. This is mostly intended to ease porting while avoiding merge conflicts.

Marking a declaration as public at minimum makes its "signature", i.e. its name and type, visible. Some specific declarations/terms may still put other parts in the private scope as follows:

- `by` used in the public scope to prove a proposition puts the resulting proof in the private scope (by wrapping it in a public helper theorem).
- `def` puts its body in the private scope by default. The defined constant cannot be unfolded when used in the public scope. This can be changed with the `@[expose]` attribute. `@[expose] section` can be used to apply the attribute to all `def`s in the section and can locally be negated by `@[no_expose]`.
- `theorem` and `opaque` never expose their body. Consider using `@[expose] def` instead if exposition is absolutely necessary.
- `abbrev` and `instance` always expose their body. For `instance`, individual field values can be marked `private`, which can be useful for programming purposes. For proof fields, `by` already does the trick.

```
module

def myInternalHelper (x : Nat) : String := ...

public instance : ToString Nat where
  toString x := private myInternalHelper x
```

Import Visibility

The basic form `public import M` makes the public scope of `M` available in the public scope of the current module. The private scope of `M` is discarded. Without `public`, the public scope of `M` is instead imported into the private scope. The import thus is irrelevant to downstream modules and ignored by them.

`import all M` behaves like `import M` but additional imports the private scope of `M` into the private scope of the current module. This is only allowed if `M` and the current module have the same module name root, as its main purpose is to allow for separating definitions and proofs into separate modules for internal organization of a library.

```
-- Module M.Defs
module

public def f : Nat := 0
```

```
-- Module M.Lemmas
module

import all M.Defs

public theorem f_eq_zero : f = 0 :=
  -- may access body of `f` imported into the private scope
  rfl
```

Note that the imported private scope includes private `import`s of `M`, including nested `import all`s that then are interpreted likewise. That is, the set of private scopes accessible to the current module is the transitive closure of `import all` declarations.

The module system's `import all` is more powerful than `import` without the module system. It makes imported private definitions accessible directly by name, as if they were defined in the current module. Thus another use case of `import all` is to make declarations available that need to be used in multiple modules but should not leak outside the current library.

`public import all M` behaves like `public import M` followed by `import all M`, i.e. the `all` modifier becomes irrelevant for downstream modules.

The `meta` Phase

When it comes to actual code execution, there is no point to a definition without a body. Thus, in order to eagerly know what definitions *might* be executed at compile time and so need to be available including their bodies (in some executable shape), any definition used as an entry point to compile-time execution has to be tagged with the new `meta` modifier

▼ [meta3]
Semantically unrelated to the modifier of the same name in Lean 3.
. This is automatically done in built-in metaprogramming syntax such as `syntax`, `macro`, and `elab` but may need to be done explicitly when manually applying metaprogramming attributes such as `@[app_delab]`.

A `meta` definition may access (and thus invoke) any `meta` or non-`meta` definition of the current module. For accessing imported definitions, the definition must either have been marked as `meta` when it was declared or the import must be marked as such (`meta import` when the accessing definition is in the private scope and `public meta import` otherwise).

```
module

meta import Std.Data.HashMap

local elab "my_elab" : command => do
  let m : Std.HashMap := {}
  ...
```

Common Errors and Patterns

The following list contains common errors one might encounter when using the module system and especially porting existing files to the module system.

Unknown constant

Check whether you might be trying to access a private definition in the public scope. If so, you might want to make the current declaration private as well or otherwise enter the private scope such as through `private` on a field or `by` for a proof. TODO: improve error message.

If the message is prefixed with `(interpreter)`, this suggests a missing `meta import`. The new import should be placed in the file defining the metaprogram depending on the missing constant, which is not necessarily the file triggering the error. Note that the language server always does `meta import`s for the benefit of `#eval` etc., so the error might only occur in a cmdline build. TODO: better, static `meta` checking.

Definitional equality errors, especially after porting

You are likely missing an **`expose`** attribute on a definition or alternatively, if imported, an `import all`. Prefer the former if anyone outside your library might feasible require the same access. `#reduce` and/or `trace.Meta.isDefEq` can help with finding the blocking definition. You might also see this as a kernel error when a tactic directly emits proof terms referencing specific declarations without going through the elaborator, such as for proof by reflection. In this case, there is no readily available trace for debugging; consider using `@[expose] section`s generously on the closure of relevant modules.

Release Notes

This section provides release notes about recent versions of Lean. When updating to a new version, please read the corresponding release notes. They may contain advice that will help you understand the differences with the previous version and upgrade your projects.

Lean 4.22.0-rc3 (2025-07-04)

For this release, 468 changes landed. In addition to the 185 feature additions and 85 fixes listed below there were 15 refactoring changes, 5 documentation improvements, 4 performance improvements, 0 improvements to the test suite and 174 other changes.

Language

- #6672 filters out all declarations from `Lean.*`, `*.Tactic.*`, and `*.Linter.*` from the results of `exact?` and `rw?`.
- #7395 changes the `show t` tactic to match its documentation. Previously it was a synonym for `change t`, but now it finds the first goal that unifies with the term `t` and moves it to the front of the goal list.
- #7639 changes the generated `below` and `brecOn` implementations for reflexive inductive types to support motives in `Sort u` rather than `Type u`.
- #8337 adjusts the experimental module system to not export any private declarations from modules.
- #8373 enables transforming nondependent `let`s into `have`s in a number of contexts: the bodies of nonrecursive definitions, equation lemmas, smart unfolding definitions, and types of theorems. A motivation for this change is that when zeta reduction is disabled, `simp` can only effectively rewrite `have` expressions (e.g. `split` uses `simp` with zeta reduction disabled), and so we cache the nondependence calculations by transforming `let`s to `have`s. The transformation can be disabled using `set_option cleanup.letToHave false`.
- #8387 improves the error messages produced by `end` and prevents invalid `end` commands from closing scopes on failure.
- #8419 introduces an explicit `defeq` attribute to mark theorems that can be used by `dsimp`. The benefit of an explicit attribute over the prior logic of looking at the proof body is that we can reliably omit theorem bodies across module boundaries. It also helps with intra-file parallelism.
- #8519 makes the equational theorems of non-exposed defs private. If the author of a module chose not to expose the body of their function, then they likely don't want that implementation to leak through equational theorems. Helps with #8419.
- #8543 adds typeclasses for `grind` to embed types into `Int`, for cutsat. This allows, for example, treating `Fin n`, or Mathlib's ℕ+ in a uniform and extensible way.
- #8568 modifies the `structure` elaborator to add local terminfo for structure fields and explicit parent projections, enabling "go to definition" when there are dependent fields.
- #8574 adds an additional diff mode to the error-message hint suggestion widget that displays diffs per word rather than per character.
- #8596 makes `guard_msgs.diff=true` the default. The main usage of `#guard_msgs` is for writing tests, and this makes staring at altered test outputs considerably less tiring.
- #8609 uses `grind` to shorten some proofs in the LRAT checker. The intention is not particularly to improve the quality or maintainability of these proofs (although hopefully this is a side effect), but just to give `grind` a work out.
- #8619 fixes an internalization (aka preprocessing) issue in `grind` when applying injectivity theorems.
- #8621 fixes a bug in the equality-resolution procedure used by `grind`. The procedure now performs a topological sort so that every simplified theorem declaration is emitted **before** any place where it is referenced. Previously, applying equality resolution to

```
h : ∀ x, p x a → ∀ y, p y b → x ≠ y
```

in the example

```
example
  (p : Nat → Nat → Prop)
  (a b c : Nat)
  (h  : ∀ x, p x a → ∀ y, p y b → x ≠ y)
  (h₁ : p c a)
  (h₂ : p c b) :
  False := by
  grind
```

caused `grind` to produce the incorrect term

```
p ?y a → ∀ y, p y b → False
```

The patch eliminates this error, and the following correct simplified theorem is generated

```
∀ y, p y a → p y b → False
```

- #8622 adds a test case / use case example for `grind`, setting up the very basics of `IndexMap`, modelled on Rust's `indexmap`. It is not intended as a complete implementation: just enough to exercise `grind`.
- #8625 improves the diagnostic information produced by `grind` when it succeeds. We now include the list of case-splits performed, and the number of application per function symbol.
- #8633 implements case-split tracking in `grind`. The information is displayed when `grind` fails or diagnostic information is requested. Examples:
  - Failure
- #8637 adds background theorems for normalizing `IntModule` expressions using reflection.
- #8638 improves the diagnostic information produced by `grind`. It now sorts the equivalence classes by generation and then `Expr. lt`.
- #8639 completes the `ToInt` family of typeclasses which `grind` will use to embed types into the integers for `cutsat`. It contains instances for the usual concrete data types (`Fin`, `UIntX`, `IntX`, `BitVec`), and is extensible (e.g. for Mathlib's `PNat`).
- #8641 adds the `#print sig $ident` variant of the `#print` command, which omits the body. This is useful for testing meta-code, in the

  ```
  #guard_msgs (drop trace, all) in #print sig foo
  ```

  idiom. The benefit over `#check` is that it shows the declaration kind, reducibility attributes (and in the future more built-in attributes, like `@[defeq]` in #8419). (One downside is that `#check` shows unused function parameter names, e.g. in induction principles; this could probably be refined.)

- #8645 adds many helper theorems for the future `IntModule` linear arithmetic procedure in `grind`. It also adds helper theorems for normalizing input atoms and support for disequality in the new linear arithmetic procedure in `grind`.
- #8650 adds helper theorems for coefficient normalization and equality detection. This theorems are for the linear arithmetic procedure in `grind`.
- #8662 adds a `warn.sorry` option (default true) that logs the "declaration uses 'sorry'" warning when declarations contain `sorryAx`. When false, the warning is not logged.
- #8670 adds helper theorems that will be used to interface the `CommRing` module with the linarith procedure in `grind`.
- #8671 allow structures to have non-bracketed binders, making it consistent with `inductive`.
- #8677 adds the basic infrastructure for the linarith module in `grind`.
- #8680 adds the `reify?` and `denoteExpr` for the new linarith module in `grind`.
- #8682 uses the `CommRing` module to normalize linarith inequalities.
- #8687 implements the infrastructure for constructing proof terms in the linarith procedure in `grind`. It also adds the `ToExpr` instances for the reified objects.
- #8689 implements proof term generation for the `CommRing` and `linarith` interface. It also fixes the `CommRing` helper theorems.
- #8690 implements the main framework of the model search procedure for the linarith component in grind. It currently handles only inequalities. It can already solve simple goals such as

```
example [IntModule α] [Preorder α] [IntModule.IsOrdered α] (a b c
: α)
    : a < b → b < c → c < a → False := by
  grind
```

- #8693 fixes the denotation functions used to interface the ring and linarith modules in grind.
- #8694 implements special support for `One.one` in linarith when the structure is a ordered ring. It also fixes bugs during initialization.
- #8697 implements support for inequalities in the `grind` linear arithmetic procedure and simplifies its design. Some examples that can already be solved:

```
open Lean.Grind
example [IntModule α] [Preorder α] [IntModule.IsOrdered α] (a b c
d : α)
    : a + d < c → b = a + (2:Int)*d → b - d > c → False := by
  grind
```

- #8708 fixes an internalization bug in the interface between linarith and ring modules in `grind`. The `CommRing` module may create new terms during normalization.
- #8713 fixes a bug in the commutative ring module used in `grind`. It was missing simplification opportunities.

- #8715 implements the basic infrastructure for processing disequalities in the `grind linarith` module. We still have to implement backtracking.
- #8723 implements a `finally` section following a (potentially empty) `where` block. `where ... finally` opens a tactic sequence block in which the goals are the unassigned metavariables from the definition body and its auxiliary definitions that arise from use of `let rec` and `where`.
- #8730 adds support for throwing named errors with associated error explanations. In particular, it adds elaborators for the syntax defined in #8649, which use the error-explanation infrastructure added in #8651. This includes completions, hovers, and jump-to-definition for error names.
- #8733 implements disequality splitting and non-chronological backtracking for the `grind` linarith procedure.

  ```
  example [IntModule α] [LinearOrder α] [IntModule.IsOrdered α] (a
  b c d : α)
      : a ≤ b → a - c ≥ 0 + d → d ≤ 0 → d ≥ 0 → b = c → a ≠ b →
  False := by
    grind
  ```
- #8751 adds the nondep field of `Expr.letE` to the C++ data model. Previously this field has been unused, and in followup PRs the elaborator will use it to encode `have` expressions (non-dependent `lets`). The kernel does not verify that nondep is correctly applied during typechecking. The `letE` delaborator now prints `haves` when nondep is true, though `have` still elaborates as `letFun` for now. Breaking change: `Expr.updateLet!` is renamed to `Expr.updateLetE!`.
- #8753 fixes a bug in `simp` where it was not resetting the set of zeta-delta reduced let definitions between `simp` calls. It also fixes a bug where `simp` would report zeta-delta reduced let definitions that weren't given as simp arguments (these extraneous let definitions appear due to certain processes temporarily setting `zetaDelta := true`). This PR also modifies the metaprogramming interface for the zeta-delta tracking functions to be re-entrant and to prevent this kind of no-reset bug from occurring again. Closes #6655.
- #8756 implements counterexamples for grind linarith. Example:

  ```
  example [CommRing α] [LinearOrder α] [Ring.IsOrdered α] (a b c d
  : α)
      : b ≥ 0 → c > b → d > b → a ≠ b + c → a > b + c → a < b + d →
  False := by
    grind
  ```

  produces the counterexample

  ```
  a := 7/2
  b := 1
  c := 2
  d := 3
  ```
- #8759 implements model-based theory combination for grind linarith. Example:

```
example [CommRing α] [LinearOrder α] [Ring.IsOrdered α] (f : α →
α → α) (x y z : α)
    : z ≤ x → x ≤ 1 → z = 1 → f x y = 2 → f 1 y = 2 := by
  grind
```

- #8763 corrects the handling of explicit `monotonicity` proofs for mutual `partial_fixpoint` definitions.
- #8773 implements support for the heterogeneous `(k : Nat) * (a : R)` in ordered modules. Example:

```
variable (R : Type u) [IntModule R] [LinearOrder R]
[IntModule.IsOrdered R]
```

- #8774 adds an option for disabling the cutsat procedure in `grind`. The linarith module takes over linear integer/nat constraints. Example:

```
set_option trace.grind.cutsat.assert true in -- cutsat should
**not** process the following constraints
example (x y z : Int) (h1 : 2 * x < 3 * y) (h2 : -4 * x + 2 * z <
0) : ¬ 12*y - 4* z < 0 := by
  grind -cutsat -- `linarith` module solves it
```

- #8775 adds a `grind` normalization theorem for `Int.negSucc`. Example:

```
example (p : Int) (n : Nat) (hmp : Int.negSucc (n + 1) + 1 = p)
    (hnm : Int.negSucc (n + 1 + 1) + 1 = Int.negSucc (n + 1)) : p
= Int.negSucc n := by
  grind
```

- #8776 ensures that user provided `natCast` application are properly internalized in the grind cutsat module.
- #8777 implements basic `Field` support in the commutative ring module in `grind`. It is just division by numerals for now. Examples:

```
open Lean Grind
```

- #8780 makes Lean code generation respect the module name provided through `lean --setup`.
- #8786 improves the support for fields in `grind`. New supported examples:

```
example [Field α] [IsCharP α 0] (x : α) : x ≠ 0 → (4 / x)⁻¹ * ((3
* x^3) / x)^2 * ((1 / (2 * x))⁻¹)^3 = 18 * x^8 := by grind
example [Field α] (a : α) : 2 * a ≠ 0 → 1 / a + 1 / (2 * a) = 3 /
(2 * a) := by grind
example [Field α] [IsCharP α 0] (a : α) : 1 / a + 1 / (2 * a) = 3
/ (2 * a) := by grind
example [Field α] [IsCharP α 0] (a b : α) : 2*b - a = a + b → 1 /
a + 1 / (2 * a) = 3 / b := by grind
example [Field α] [NoNatZeroDivisors α] (a : α) : 1 / a + 1 / (2
* a) = 3 / (2 * a) := by grind
```

```
example [Field α] {x y z w : α} : x / y = z / w → y ≠ 0 → w ≠ 0 →
x * w = z * y := by grind
example [Field α] (a : α) : a = 0 → a ≠ 1 := by grind
example [Field α] (a : α) : a = 0 → a ≠ 1 - a := by grind
```

- #8789 implements the Rabinowitsch transformation for `Field` disequalities in `grind`. For example, this transformation is necessary for solving:

  ```
  example [Field α] (a : α) : a^2 = 0 → a = 0 := by
    grind
  ```

- #8791 ensures the `grind linarith` module is activated for any type that implements only `IntModule`. That is, the type does not need to be a preorder anymore.
- #8792 makes the `clear_value` tactic preserve the order of variables in the local context. This is done by adding `Lean.MVarId.withRevertedFrom`, which reverts all local variables starting from a given variable, rather than only the ones that depend on it.
- #8794 adds a module `Lean.Util.CollectLooseBVars` with a function `Expr.collectLooseBVars` that collects the set of loose bound variables in an expression. That is, it computes the set of all `i` such that `e.hasLooseBVar i` is true.
- #8795 ensures that auxliary terms are not internalized by the ring and linarith modules.
- #8796 fixes `grind linarith` term internalization and support for `HSMul`.
- #8798 adds the following instance

  ```
  instance [Field α] [LinearOrder α] [Ring.IsOrdered α] : IsCharP α
  0
  ```

  The goal is to ensure we do not perform unnecessary case-splits in our test suite.

- #8804 implements first-class support for nondependent let expressions in the elaborator; recall that a let expression `let x : t := v; b` is called *nondependent* if `fun x : t => b` typechecks, and the notation for a nondependent let expression is `have x := v; b`. Previously we encoded `have` using the `letFun` function, but now we make use of the `nondep` flag in the `Expr.letE` constructor for the encoding. This has been given full support throughout the metaprogramming interface and the elaborator. Key changes to the metaprogramming interface:

  - Local context `ldecls` with `nondep := true` are generally treated as `cdecls`. This is because in the body of a `have` expression the variable is opaque. Functions like `LocalDecl.isLet` by default return `false` for nondependent `ldecls`. In the rare case where it is needed, they take an additional optional `allowNondep : Bool` flag (defaults to `false`) if the variable is being processed in a context where the value is relevant.
  - Functions such as `mkLetFVars` by default generalize nondependent let variables and create lambda expressions for them. The `generalizeNondepLet` flag (default true) can be set to false if `have` expressions should be produced instead. **Breaking change:** Uses of `letLambdaTelescope`/`mkLetFVars` need to use `generalizeNondepLet := false`. See the next item.

  - There are now some mapping functions to make telescoping operations more convenient. See `mapLetTelescope` and `mapLambdaLetTelescope`. There is also `mapLetDecl` as a counterpart to `withLetDecl` for creating `let`/`have` expressions.
  - Important note about the `generalizeNondepLet` flag: it should only be used for variables in a local context that the metaprogram "owns". Since nondependent let variables are treated as constants in most cases, the `value` field might refer to variables that do not exist, if for example those variables were cleared or reverted. Using `mapLetDecl` is always fine.
  - The simplifier will cache its let dependence calculations in the nondep field of let expressions.
  - The `intro` tactic still produces *dependent* local variables. Given that the simplifier will transform lets into haves, it would be surprising if that would prevent `intro` from creating a local variable whose value cannot be used.

- #8809 introduces the basic theory of ordered modules over Nat (i.e. without subtraction), for `grind`. We'll solve problems here by embedding them in the `IntModule` envelope.
- #8810 implements equality elimination in `grind linarith`. The current implementation supports only `IntModule` and `IntModule` + `NoNatZeroDivisors`
- #8813 adds some basic lemmas about `grind` internal notions of modules.
- #8815 refactors the way simp arguments are elaborated: Instead of changing the `SimpTheorems` structure as we go, this elaborates each argument to a more declarative description of what it does, and then apply those. This enables more interesting checks of simp arguments that need to happen in the context of the eventually constructed simp context (the checks in #8688), or after simp has run (unused argument linter #8901).
- #8828 extends the experimental module system to support resolving private names imported (transitively) through `import all`.
- #8835 defines the embedding of a `CommSemiring` into its `CommRing` envelope, injective when the `CommSemiring` is cancellative. This will be used by `grind` to prove results in `Nat`.
- #8836 generalizes #8835 to the noncommutative case, allowing us to embed a `Lean.Grind.Semiring` into a `Lean.Grind.Ring`.
- #8845 implements the proof-by-reflection infrastructure for embedding semiring terms as ring ones.
- #8847 relaxes the assumptions for `Lean.Grind.IsCharP` from `Ring` to `Semiring`, and provides an alternative constructor for rings.
- #8848 generalizes the internal `grind` instance

  ```
  instance [Field α] [LinearOrder α] [Ring.IsOrdered α] : IsCharP α 0
  ```

  to

  ```
  instance [Ring α] [Preorder α] [Ring.IsOrdered α] : IsCharP α 0
  ```

- #8855 refactors `Lean.Grind.NatModule/IntModule/Ring.IsOrdered`.
- #8859 shows the equivalence between `Lean.Grind.NatModule.IsOrdered` and `Lean.Grind.IntModule.IsOrdered` over an `IntModule`.
- #8865 allows `simp` to recognize and warn about simp lemmas that are likely looping in the current simp set. It does so automatically whenever simplification fails with the dreaded "max recursion depth" error fails, but it can be made to do it always with `set_option linter.loopingSimpArgs true`. This check is not on by default because it is somewhat costly, and can warn about simp calls that still happen to work.
- #8874 skips attempting to compute a module name from the file name and root directory (i.e., `lean -R`) if a name is already provided via `lean --setup`.
- #8880 makes `simp` consult its own cache more often, to avoid replicating work.
- #8882 adds `@[expose]` annotations to terms that appear in `grind` proof certificates, so `grind` can be used in the module system. It's possible/likely that I haven't identified all of them yet.
- #8890 adds doc-strings to the `Lean.Grind` algebra typeclasses, as these will appear in the reference manual explaining how to extend `grind` algebra solvers to new types. Also removes some redundant fields.
- #8892 corrects the pretty printing of `grind` modifiers. Previously `@[grind →]` was being pretty printed as `@[grind→ ]` (Space on the right of the symbol, rather than left.) This fixes the pretty printing of attributes, and preserves the presence of spaces after the symbol in the output of `grind?`.
- #8893 fixes a bug in the `dvd` propagation function in cutsat.
- #8901 adds a linter (`linter.unusedSimpArgs`) that complains when a simp argument (`simp [foo]`) is unused. It should do the right thing if the `simp` invocation is run multiple times, e.g. inside `all_goals`. It does not trigger when the `simp` call is inside a macro. The linter message contains a clickable hint to remove the simp argument.
- #8903 make sure that the local instance cache calculation applies more reductions. In #2199 there was an issue where metavariables could prevent local variables from being considered as local instances. We use a slightly different approach that ensures that, for example, `let`s at the ends of telescopes do not cause similar problems. These reductions were already being calculated, so this does not require any additional work to be done.
- #8909 refactors the `NoNatZeroDivisors` to make sure it will work with the new `Semiring` support.
- #8910 adds the `NoNatZeroDivisors` instance for `OfSemiring.Q α`
- #8913 cleans up `grind`'s internal order typeclasses, removing unnecessary duplication.
- #8914 modifies `let` and `have` term syntaxes to be consistent with each other. Adds configuration options; for example, `have` is equivalent to `let +nondep`, for *nondependent* lets. Other options include `+usedOnly` (for `let_tmp`), `+zeta` (for `letI`/`haveI`), and

`+postponeValue` (for `let_delayed`). There is also `let (eq := h) x := v; b` for introducing `h : x = v` when elaborating `b`. The `eq` option works for pattern matching as well, for example `let (eq := h) (x, y) := p; b`.

- #8918 fixes the `guard_msgs.diff` default behavior so that the default specified in the option definition is actually used everywhere.
- #8921 implements support for (commutative) semirings in `grind`. It uses the Grothendieck completion to construct a (commutative) ring `Lean.Grind.Ring.OfSemiring.Q α` from a (commutative) semiring `α`. This construction is mostly useful for semirings that implement `AddRightCancel α`. Otherwise, the function `toQ` is not injective. Examples:

```
example (x y : Nat) : x^2*y = 1 → x*y^2 = y → y*x = 1 := by
  grind
```

- #8935 adds the `+generalize` option to the `let` and `have` syntaxes. For example, `have +generalize n := a + b; body` replaces all instances of `a + b` in the expected type with `n` when elaborating `body`. This can be likened to a term version of the `generalize` tactic. One can combine this with `eq` in `have +generalize (eq := h) n := a + b; body` as an analogue of `generalize h : n = a + b`.
- #8937 changes the output universe of the generated `below` implementation for non-reflexive inductive types to match the implementation for reflexive inductive types in #7639.
- #8940 introduces antitonicity lemmas that support the elaboration of mixed inductive-coinductive predicates defined using the `least_fixpoint` / `greatest_fixpoint` constructs.
- #8943 adds helper theorems for normalizing semirings that do not implement `AddRightCancel`.
- #8953 implements support for normalization for commutative semirings that do not implement `AddRightCancel`. Examples:

```
variable (R : Type u) [CommSemiring R]
```

- #8954 adds a procedure that efficiently transforms `let` expressions into `have` expressions (`Meta.letToHave`). This is exposed as the `let_to_have` tactic.
- #8955 fixes `Lean.MVarId.deltaLocalDecl`, which previously replaced the local definition with the target.
- #8957 adds configuration options to the `let`/`have` tactic syntaxes. For example, `let (eq := h) x := v` adds `h : x = v` to the local context. The configuration options are the same as those for the `let`/`have` term syntaxes.
- #8958 improves the case splitting strategy used in `grind`, and ensures `grind` also considers simple `match`-conditions for case-splitting. Example:

```
example (x y : Nat)
    : 0 < match x, y with
          | 0, 0   => 1
```

```
  | _, _ => x + y := by -- x or y must be greater than 0
grind
```

- #8959 add instances showing that the Grothendieck (i.e. additive) envelope of a semiring is an ordered ring if the original semiring is ordered (and satisfies ExistsAddOfLE), and in this case the embedding is monotone.
- #8963 embeds a NatModule into its IntModule completion, which is injective when we have AddLeftCancel, and monotone when the modules are ordered. Also adds some (failing) grind test cases that can be verified once `grind` uses this embedding.
- #8964 adds `@[expose]` attributes to proof terms constructed by `grind` that need to be evaluated in the kernel.
- #8965 revises @[grind] annotations on Nat bitwise operations.
- #8968 adds the following features to `simp`:
  - A routine for simplifying `have` telescopes in a way that avoids quadratic complexity arising from locally nameless expression representations, like what #6220 did for `letFun` telescopes. Furthermore, simp converts `letFun`s into `have`s (nondependent lets), and we remove the #6220 routine since we are moving away from `letFun` encodings of nondependent lets.
  - A `+letToHave` configuration option (enabled by default) that converts lets into haves when possible, when `-zeta` is set. Previously Lean would need to do a full typecheck of the bodies of `let`s, but the `letToHave` procedure can skip checking some subexpressions, and it modifies the `let`s in an entire expression at once rather than one at a time.
  - A `+zetaHave` configuration option, to turn off zeta reduction of `have`s specifically. The motivation is that dependent `let`s can only be dsimped by let, so zeta reducing just the dependent lets is a reasonable way to make progress. The `+zetaHave` option is also added to the meta configuration.
  - When `simp` is zeta reducing, it now uses an algorithm that avoids complexity quadratic in the depth of the let telescope.
  - Additionally, the zeta reduction routines in `simp`, `whnf`, and `isDefEq` now all are consistent with how they apply the `zeta`, `zetaHave`, and `zetaUnused` configurations.
- #8971 fixes `linter.simpUnusedSimpArgs` to check the syntax kind, to not fire on `simp` calls behind macros. Fixes #8969
- #8973 refactors the juggling of universes in the linear `noConfusionType` construction: Instead of using `PUnit.{…} →` in the to get the branches of `withCtorType` to the same universe level, we use `PULift`.
- #8978 updates the `solveMonoStep` function used in the `monotonicity` tactic to check for definitional equality between the current goal and the monotonicity proof obtained from a recursive call. This ensures soundness by preventing incorrect applications when

`Lean.Order.PartialOrder` instances differ—an issue that can arise with `mutual` blocks defined using the `partial_fixpoint` keyword, where different `Lean.Order.CCPO` structures may be involved.

- #8980 improves the consistency of error message formatting by rendering addenda of several existing error messages as labeled notes and hints.
- #8983 fixes a bug in congruence proof generation in `grind` for over-applied functions.
- #8986 improves the error messages produced by invalid projections and field notation. It also adds a hint to the "function expected" error message noting the argument to which the term is being applied, which can be helpful for debugging spurious "function expected" messages actually caused by syntax errors.
- #8991 adds some missing `ToInt.X` typeclass instances for `grind`.
- #8995 introduces a Hoare logic for monadic programs in `Std.Do.Triple`, and assorted tactics:
  - `mspec` for applying Hoare triple specifications
  - `mvcgen` to turn a Hoare triple proof obligation `⦃P⦄ prog ⦃Q⦄` into pure verification conditoins (i.e., without any traces of Hoare triples or weakest preconditions reminiscent of `prog`). The resulting verification conditions in the stateful logic of `Std.Do.SPred` can be discharged manually with the tactics coming with its custom proof mode or with automation such as `simp` and `grind`.
- #8996 provides the remaining instances for the `Lean.Grind.ToInt` typeclasses.
- #9004 ensures that type-class synthesis failure errors in interpolated strings are displayed at the interpolant at which they occurred.
- #9005 changes the definition of `Lean.Grind.ToInt.OfNat`, introducing a `wrap` on the right-hand-side.
- #9008 implements the basic infrastructure for the generic `ToInt` support in `cutsat`.
- #9022 completes the generic `toInt` infrastructure for embedding terms implementing the `ToInt` type classes into `Int`.
- #9026 implements support for (non strict) `ToInt` inequalities in `grind cutsat`. `grind cutsat` can solve simple problems such as:

```
example (a b c : Fin 11) : a ≤ b → b ≤ c → a ≤ c := by
  grind
```

- #9030 fixes a couple of bootstrapping-related hiccups in the newly added `Std.Do` module. More precisely,
- #9035 extends the list of acceptable characters to all the french ones as well as some others, by adding characters from the Latin-1-Supplement add Latin-Extended-A unicode block.
- #9038 adds test cases for the VC generator and implements a few small and tedious fixes to ensure they pass.

- #9041 makes `mspec` detect more viable assignments by `rfl` instead of generating a VC.
- #9044 adjusts the experimental module system to make `private` the default visibility modifier in `module`s, introducing `public` as a new modifier instead. `public section` can be used to revert the default for an entire section, though this is more intended to ease gradual adoption of the new semantics such as in `Init` (and soon `Std`) where they should be replaced by a future decl-by-decl re-review of visibilities.
- #9045 fixes a type error in `mvcgen` and makes it turn fewer natural goals into synthetic opaque ones, so that tactics such as `trivial` may instantiate them more easily.
- #9048 implements support for strict inequalities in the `ToInt` adapter used in `grind cutsat`. Example:

```
example (a b c : Fin 11) : c ≤ 9 → a ≤ b → b < c → a < c + 1 :=
by
  grind
```

- #9050 ensures the `ToInt` bounds are asserted for every `toInt a` application internalized in `grind cutsat`.
- #9051 implements support for equalities and disequalities in `grind cutsat`. We still have to improve the encoding. Examples:

```
example (a b c : Fin 11) : a ≤ 2 → b ≤ 3 → c = a + b → c ≤ 5 :=
by
  grind
```

- #9057 introduces a simple variable-reordering heuristic for `cutsat`. It is needed by the `ToInt` adapter to support finite types such as `UInt64`. The current encoding into `Int` produces large coefficients, which can enlarge the search space when an unfavorable variable order is used. Example:

```
example (a b c : UInt64) : a ≤ 2 → b ≤ 3 → c - a - b = 0 → c ≤ 5
:= by
  grind
```

- #9059 adds helper theorems for normalizing coefficients in rings of unknown characteristic.
- #9062 implements support for equations `<num> = 0` in rings and fields of unknown characteristic. Examples:

```
example [Field α] (a : α) : (2 * a)⁻¹ = a⁻¹ / 2 := by grind
```

- #9065 improves the counterexamples produced by the `cutsat` procedure in `grind` when using the `ToInt` gadget.
- #9067 adds a docstring for the `grind` tactic.
- #9069 implements support for the type class `LawfulEqCmp`. Examples:

```
example (a b c : Vector (List Nat) n)
    : b = c → a.compareLex (List.compareLex compare) b = o → o =
```

```
.eq → a = c := by
  grind
```

- #9073 copies #9069 to handle `ReflCmp` the same way; we need to call this in propagateUp rather than propagateDown.
- #9074 uses the commutative ring module to normalize nonlinear polynomials in `grind cutsat`. Examples:

  ```
  example (a b : Nat) (h₁ : a + 1 ≠ a * b * a) (h₂ : a * a * b ≤ a
  + 1) : b * a^2 < a + 1 := by
    grind
  ```

- #9076 adds an unexpander for `OfSemiring.toQ`. This an auxiliary function used by the `ring` module in `grind`, but we want to reduce the clutter in the diagnostic information produced by `grind`. Example:

  ```
  example [CommSemiring α] [AddRightCancel α] [IsCharP α 0] (x y :
  α)
      : x^2*y = 1 → x*y^2 = y → x + y = 2 → False := by
    grind
  ```

  produces

  ```
  [ring] Ring `Ring.OfSemiring.Q α` ▼
    [basis] Basis ▼
      [_] ↑x + ↑y + -2 = 0
      [_] ↑y + -1 = 0
  ```

- #9086 deprecates `let_fun` syntax in favor of `have` and removes `letFun` support from WHNF and `simp`.
- #9087 removes the `irreducible` attribute from `letFun`, which is one step toward removing special `letFun` support; part of #9086.

Library

- #8003 adds a new monadic interface for `Async` operations.
- #8072 adds DNS functions to the standard library
- #8109 adds system information functions to the standard library
- #8178 provides a compact formula for the MSB of the sdiv. Most of the work in the PR involves handling the corner cases of division overflowing (e.g. `intMin / -1 = intMin`)
- #8203 adds trichotomy lemmas for unsigned and signed comparisons, stating that only one of three cases may happen: either `x < y`, `x = y`, or `x > y` (for both signed and unsigned comparsions). We use explicit arguments so that users can write `rcases slt_trichotomy x y with hlt | heq | hgt`.
- #8205 adds a simp lemma that simplifies T-division where the numerator is a `Nat` into an E-division:

```
@[simp] theorem ofNat_tdiv_eq_ediv {a : Nat} {b : Int} : (a :
Int).tdiv b = a / b :=
   tdiv_eq_ediv_of_nonneg (by simp)
```

- #8210 adds an equivalence relation to tree maps akin to the existing one for hash maps. In order to get many congruence lemmas to eventually use for defining functions on extensional tree maps, almost all of the remaining tree map functions have also been given lemmas to relate them to list functions, although these aren't currently used to prove lemmas other than congruence lemmas.
- #8253 adds `toInt_smod` and auxilliary lemmas necessary for its proof (`msb_intMin_umod_neg_of_msb_true`, `msb_neg_umod_neg_of_msb_true_of_msb_true`, `toInt_dvd_toInt_iff`, `toInt_dvd_toInt_iff_of_msb_true_msb_false`, `toInt_dvd_toInt_iff_of_msb_false_msb_true`, `neg_toInt_neg_umod_eq_of_msb_true_msb_true`, `toNat_pos_of_ne_zero`, `toInt_umod_neg_add`, `toInt_sub_neg_umod` and `BitVec.[lt_of_msb_false_of_msb_true, msb_umod_of_msb_false_of_ne_zero, neg_toInt_neg]`)
- #8420 provides the iterator combinator `drop` that transforms any iterator into one that drops the first n elements.
- #8534 fixes `IO.FS.realPath` on windows to take symbolic links into account.
- #8545 provides the means to reason about "equivalent" iterators. Simply speaking, two iterators are equivalent if they behave the same as long as consumers do not introspect their states.
- #8546 adds a new `BitVec.clz` operation and a corresponding `clz` circuit to `bv_decide`, allowing to bitblast the count leading zeroes operation. The AIG circuit is linear in the number of bits of the original expression, making the bitblasting convenient wrt. rewriting. `clz` is common in numerous compiler intrinsics (see here) and architectures (see here).
- #8573 avoids the likely unexpected behavior of `removeDirAll` to delete through symlinks and adds the new function `IO.FS.symlinkMetadata`.
- #8585 makes the lemma `BitVec.extractLsb'_append_eq_ite` more usable by using the "simple case" more often, and uses this simplification to make `BitVec.extractLsb'_append_eq_of_add_lt` stronger, renaming it to `BitVec.extractLsb'_append_eq_of_add_le`.
- #8587 adjusts the grind annotation on `Std.HashMap.map_fst_toList_eq_keys` and variants, so `grind` can reason bidirectionally between `m.keys` and `m.toList`.
- #8590 adds `@[grind]` to `getElem?_pos` and variants.
- #8615 provides a special empty iterator type. Although its behavior can be emulated with a list iterator (for example), having a special type has the advantage of being easier to optimize for the compiler.

- #8620 removes the `NatCast (Fin n)` global instance (both the direct instance, and the indirect one via `Lean.Grind.Semiring`), as that instance causes causes `x < n` (for `x : Fin k`, `n : Nat`) to be elaborated as `x < ↑n` rather than `↑x < n`, which is undesirable. Note however that in Mathlib this happens anyway!
- #8629 replaces special, more optimized `IteratorLoop` instances, for which no lawfulness proof has been made, with the verified default implementation. The specialization of the loop/collect implementations is low priority, but having lawfulness instances for all iterators is important for verification.
- #8631 generalizes `Std.Sat.AIG. relabel(Nat)_unsat_iff` to allow the AIG type to be empty. We generalize the proof, by showing that in the case when α is empty, the environment doesn't matter, since all environments `α → Bool` are isomorphic.
- #8640 adds `BitVec.setWidth'_eq` to `bv_normalize` such that `bv_decide` can reduce it and solve lemmas involving `setWidth'_eq`
- #8669 makes `unsafeBaseIO` `noinline`. The new compiler is better at optimizing `Result`-like types, which can cause the final operation in an `unsafeBaseIO` block to be dropped, since `unsafeBaseIO` is discarding the state.
- #8678 makes the LHS of `isSome_finIdxOf?` and `isNone_finIdxOf?` more general.
- #8703 corrects the `IteratorLoop` instance in `DropWhile`, which previously triggered for arbitrary iterator types.
- #8719 adds grind annotations for List/Array/Vector.eraseP/erase/eraseIdx. It also adds some missing lemmas.
- #8721 adds the types `Std.ExtDTreeMap`, `Std.ExtTreeMap` and `Std.ExtTreeSet` of extensional tree maps and sets. These are very similar in construction to the existing extensional hash maps with one exception: extensional tree maps and sets provide all functions from regular tree maps and sets. This is possible because in contrast to hash maps, tree maps are always ordered.
- #8734 adds the missing instance

  ```
  instance decidableExistsFin (P : Fin n → Prop) [DecidablePred P]
  : Decidable (∃ i, P i)
  ```
- #8740 introduces associativity rules and preservation of `(umul, smul, uadd, sadd)Overflow`flags.
- #8741 adds annotations for `List/Array/Vector.find?/findSome?/idxOf?/findIdx?`.
- #8742 fixes a bug where the single-quote character `Char.ofNat 39` would delaborate as `''', which causes a parse error if pasted back in to the source code.
- #8745 adds a logic of stateful predicates `SPred` to `Std.Do` in order to support reasoning about monadic programs. It comes with a dedicated proof mode the tactics of which are accessible by importing `Std.Tactic.Do`.
- #8747 adds grind annotations for `List/Array/Vector.finRange` theorems.

- #8748 adds grind annotations for `Array/Vector.mapIdx` and `mapFinIdx` theorems.
- #8749 adds grind annotations for `List/Array/Vector.ofFn` theorems and additional `List.Impl` find operations.
- #8750 adds grind annotations for the `List/Array/Vector.zipWith/zipWithAll/unzip` functions.
- #8765 adds grind annotations for `List.Perm`; involves a revision of grind annotations for `List.countP/count` as well.
- #8768 introduces a `ForIn'` instance and a `size` function for iterators in a minimal fashion. The `ForIn'` instance is not marked as an instance because it is unclear which `Membership` relation is sufficiently useful. The `ForIn'` instance existing as a `def` and inducing the `ForIn` instance, it becomes possible to provide more specialized `ForIn'` instances, with nice `Membership` relations, for various types of iterators. The `size` function has no lemmas yet.
- #8784 introduces ranges that are polymorphic, in contrast to the existing `Std.Range` which only supports natural numbers.
- #8805 continues adding `grind` annotations for `List/Array/Vector` lemmas.
- #8808 adds the missing `le_of_add_left_le {n m k : Nat} (h : k + n ≤ m) : n ≤ m` and `le_add_left_of_le {n m k : Nat} (h : n ≤ m) : n ≤ k + m`.
- #8811 adds theorems `BitVec.(toNat, toInt, toFin)_shiftLeftZeroExtend`, completing the API for `BitVec.shiftLeftZeroExtend`.
- #8826 corrects the definition of `Lean.Grind.NatModule`, which wasn't previously useful.
- #8827 renames `BitVec.getLsb'` to `BitVec.getLsb`, now that older deprecated definition occupying that name has been removed. (Similarly for `BitVec.getMsb'`.)
- #8829 avoids importing all of `BitVec.Lemmas` and `BitVec.BitBlast` into `UInt.Lemmas`. (They are still imported into `SInt.Lemmas`; this seems much harder to avoid.)
- #8830 rearranges files under `Init.Grind`, moving out instances for concrete algebraic types in `Init.GrindInstances`.
- #8849 adds `grind` annotations for `Sum`.
- #8850 adds `grind` annotations for `Prod`.
- #8851 adds grind annotations for `Function.curry/uncurry`.
- #8852 adds grind annotations for `Nat.testBit` and bitwise operations on `Nat`.
- #8853 adds `grind` annotations relating `Nat.fold/foldRev/any/all` and `Fin.foldl/foldr/foldlM/foldrM` to the corresponding operations on `List.finRange`.
- #8877 adds grind annotations for `List/Array/Vector.attach/attachWith/pmap`.
- #8878 adds grind annotations for List/Array/Vector monadic functions.

- #8886 adds `IO.FS.Stream.readToEnd` which parallels `IO.FS.Handle.readToEnd` along with its upstream definitions (i.e., `readBinToEndInto` and `readBinToEnd`). It also removes an unnecessary `partial` from `IO.FS.Handle.readBinToEnd`.
- #8887 generalizes `IO.FS.lines` with `IO.FS.Handle.lines` and adds the parallel `IO.FS.Stream.lines` for streams.
- #8897 simplifies some `simp` calls.
- #8905 uses the linter from https://github.com/leanprover/lean4/pull/8901 to clean up simp arguments.
- #8920 uses the linter from #8901 to clean up more simp arguments, completing #8905.
- #8928 adds a logic of stateful predicates SPred to Std.Do in order to support reasoning about monadic programs. It comes with a dedicated proof mode the tactics of which are accessible by importing Std.Tactic.Do.
- #8941 adds `BitVec.(getElem, getLsbD, getMsbD)_(smod, sdiv, srem)` theorems to complete the API for `sdiv`, `srem`, `smod`. Even though the rhs is not particularly succint (it's hard to find a meaning for what it means to have "the n-th bit of the result of a signed division/modulo operation"), these lemmas prevent the need to `unfold` of operations.
- #8947 introduces polymorphic slices in their most basic form. They come with a notation similar to the new range notation. `Subarray` is now also a slice and can produce an iterator now. It is intended to migrate more operations of `Subarray` to the `Slice` wrapper type to make them available for slices of other types, too.
- #8950 adds `BitVec.toFin_(sdiv, smod, srem)` and `BitVec.toNat_srem`. The strategy for the `rhs` of the `toFin_*` lemmas is to consider what the corresponding `toNat_*` theorems do and push the `toFin` closerto the operands. For the `rhs` of `BitVec.toNat_srem` I used the same strategy as `BitVec.toNat_smod`.
- #8967 both adds initial `@[grind]` annotations for `BitVec`, and uses `grind` to remove many proofs from `BitVec/Lemmas`.
- #8974 adds `BitVec.msb_(smod, srem)`.
- #8977 adds a generic `MonadLiftT Id m` instance. We do not implement a `MonadLift Id m` instance because it would slow down instance resolution and because it would create more non-canonical instances. This change makes it possible to iterate over a pure iterator, such as `[1, 2, 3].iter`, in arbitrary monads.
- #8992 adds `PULift`, a more general form of `ULift` and `PLift` that subsumes both.
- #8995 introduces a Hoare logic for monadic programs in `Std.Do.Triple`, and assorted tactics:
  - `mspec` for applying Hoare triple specifications
  - `mvcgen` to turn a Hoare triple proof obligation `⦃P⦄ prog ⦃Q⦄` into pure verification conditoins (i.e., without any traces of Hoare triples or weakest preconditions reminiscent of `prog`). The resulting verification conditions in the stateful logic of `Std.Do.SPred`

can be discharged manually with the tactics coming with its custom proof mode or with automation such as `simp` and `grind`.

- #9027 provides an iterator combinator that lifts the emitted values into a higher universe level via `ULift`. This combinator is then used to make the subarray iterators universe-polymorphic. Previously, they were only available for `Subarray α` if `α : Type`.
- #9030 fixes a couple of bootstrapping-related hiccups in the newly added `Std.Do` module. More precisely,
- #9038 adds test cases for the VC generator and implements a few small and tedious fixes to ensure they pass.
- #9049 proves that the default `toList`, `toListRev` and `toArray` functions on slices can be described in terms of the slice iterator. Relying on new lemmas for the `uLift` and `attachWith` iterator combinators, a more concrete description of said functions is given for `Subarray`.
- #9054 corrects some inconsistencies in `TreeMap`/`HashMap` grind annotations, for `isSome_get?_eq_contains` and `empty_eq_emptyc`.
- #9055 renames `Array/Vector.extract_push` to `extract_push_of_le`, and replaces the lemma with one without a side condition.
- #9058 provides a `ToStream` instance for slices so that they can be used in `for i in xs, j in ys do` notation.
- #9075 adds `BEq` instances for `ByteArray` and `FloatArray` (also a `DecidableEq` instance for `ByteArray`).

Compiler

- #8594 removes incorrect optimizations for strictOr/strictAnd from the old compiler, along with deleting an incorrect test. In order to do these optimizations correctly, nontermination analysis is required. Arguably, the correct way to express these optimizations is by exposing the implementation of strictOr/strictAnd to a nontermination-aware phase of the compiler, and then having them follow from more general transformations.
- #8595 wraps the invocation of the new compiler in `withoutExporting`. This is not necessary for the old compiler because it uses more direct access to the kernel environment.
- #8602 adds support to the new compiler for `Eq.recOn` (which is supported by the old compiler but missing a test).
- #8604 adds support for the `compiler.extract_closed` option to the new compiler, since this is used by the definition of `unsafeBaseIO`. We'll revisit this once we switch to the new compiler and rethink its relationship with IO.
- #8614 implements constant folding for `toNat` in the new compiler, which improves parity with the old compiler.
- #8616 adds constant folding for `Nat.pow` to the new compiler, following the same limits as the old compiler.

- #8618 implements LCNF constant folding for `Nat.nextPowerOfTwo`.
- #8634 makes `hasTrivialStructure?` return false for types whose constructors have types that are erased, e.g. if they construct a `Prop`.
- #8636 adds a function called `lean_setup_libuv` that initializes required LIBUV components. It needs to be outside of `lean_initialize_runtime_module` because it requires `argv` and `argc` to work correctly.
- #8647 improves the precision of the new compiler's `noncomputable` check for projections. There is no test included because while this was reduced from Mathlib, the old compiler does not correctly handle the reduced test case. It's not entirely clear to me if the check is passing with the old compiler for correct reasons. A test will be added to the new compiler's branch.
- #8675 increases the precision of the new compiler's non computable check, particularly around irrelevant uses of `noncomputable` defs in applications.
- #8681 adds an optimization to the LCNF simp pass where the discriminant of a `cases` construct will only be mark used if it has a non-default alternative.
- #8683 adds an optimization to the LCNF simp pass where the discriminant of a single-alt cases is only marked as used if any param is used.
- #8709 handles constants with erased types in `toMonoType`. It is much harder to write a test case for this than you would think, because most references to such types get replaced with `lcErased` earlier.
- #8712 optimizes let decls of an erased type to an erased value. Specialization can create local functions that produce a Prop, and there's no point in keeping them around.
- #8716 makes any type application of an erased term to be erased. This comes up a bit more than one would expect in the implementation of Lean itself.
- #8717 uses the fvar substitution mechanism to replace erased code. This isn't entirely satisfactory, since LCNF's `.return` doesn't support a general `Arg` (which has a `.erased` constructor), it only supports an `FVarId`. This is in contrast to the IR `.ret`, which does support a general `Arg`.
- #8729 changes LCNF's `FVarSubst` to use `Arg` rather than `Expr`. This enforces the requirements on substitutions, which match the requirements on `Arg`.
- #8752 fixes an issue where the `extendJoinPointContext` pass can lift join points containing projections to the top level, as siblings of `cases` constructs matching on other projections of the same base value. This prevents the `structProjCases` pass from projecting both at once, extending the lifetime of the parent value and breaking linearity at runtime.
- #8754 changes the implementation of computed fields in the new compiler, which should enable more optimizations (and remove a questionable hack in `toLCNF` that was only suitable for bringup). We convert `casesOn` to `cases` like we do for other inductive types, all constructors get replaced by their real implementations late in the base phase, and then the `cases` expression is rewritten to use the real constructors in `toMono`.

- #8758 adds caching for the `hasTrivialStructure?` function for LCNF types. This is one of the hottest small functions in the new compiler, so adding a cache makes a lot of sense.
- #8764 changes the LCNF pass pipeline so checks are no longer run by default after every pass, only after `init`, `saveBase`, `toMono` and `saveMono`. This is a compile time improvement, and the utility of these checks is decreased a bit after the decision to no longer attempt to preserve types throughout compilation. They have not been a significant way to discover issues during development of the new compiler.
- #8802 fixes a bug in `floatLetIn` where if one decl (e.g. a join point) is floated into a case arm and it uses another decl (e.g. another join point) that does not have any other existing uses in that arm, then the second decl does not get floated in despite this being perfectly legal. This was causing artificial array linearity issues in `Lean.Elab.Tactic.BVDecide.LRAT.trim.useAnalysis`.
- #8816 adds constant folding for Char.ofNat in LCNF simp. This implicitly relies on the representation of `Char` as `UInt32` rather than making a separate `.char` literal type, which seems reasonable as `Char` is erased by the trivial structure optimization in `toMono`.
- #8822 adds a cache for constructor info in toIR. This is called for all constructors, projections, and cases alternatives, so it makes sense to cache.
- #8825 improves IR generation for constructors of inductive types that are represented by scalars. Surprisingly, this isn't required for correctness, because the boxing pass will fix it up. The extra `unbox` operation it inserts shouldn't matter when compiling to native code, because it's trivial for a C compiler to optimize, but it does matter for the interpreter.
- #8831 caches the result of `lowerEnumToScalarType`, which is used heavily in LCNF to IR conversion.
- #8885 removes an old workaround around non-implemented C++11 features in the thread finalization.
- #8923 implements `casesOn` for `Thunk` and `Task`. Since these are builtin types, this needs to be special-cased in `toMono`.
- #8952 fixes the handling of the `never_extract` attribute in the compiler's CSE pass. There is an interesting debate to be had about exactly how hard the compiler should try to avoid duplicating anything that transitively uses `never_extract`, but this is the simplest form and roughly matches the check in the old compiler (although due to different handling of local function decls in the two compilers, the consequences might be slightly different).
- #8956 changes `toLCNF` to stop caching translations of expressions upon seeing an expression marked `never_extract`. This is more coarse-grained than it needs to be, but it is difficult to do any better, as the new compiler's `Expr` cache is based on structural identity (rather than the pointer identity of the old compiler).
- #9003 implements the validity check for the type of `main` in the new compiler. There were no tests for this, so it slipped under the radar.

Pretty Printing

- #7954 improves `pp.oneline`, where it now preserves tags when truncating formatted syntax to a single line. Note that the `[...]` continuation does not yet have any functionality to enable seeing the untruncated syntax. Closes #3681.
- #8617 fixes (1) an issue where private names are not unresolved when they are pretty printed, (2) an issue where in `pp.universes` mode names were allowed to shadow local names, (3) an issue where in `match` patterns constants shadowing locals wouldn't use `_root_`, and (4) an issue where tactics might have an incorrect "try this" when `pp.fullNames` is set. Adds more delaboration tests for name unresolution.
- #8626 closes #3791, making sure that the Syntax formatter inserts whitespace before and after comments in the leading and trailing text of Syntax to avoid having comments comment out any following syntax, and to avoid comments' lexical syntax from being interpreted as being part of another syntax. If the text contains newlines before or after any comments, they are formatted as hard newlines rather than soft newlines. For example, `--` comments will have a hard newline after them. Note: metaprograms generating Syntax with comments should be sure to include newlines at the ends of `--` comments.

Documentation

- #8934 adds explanations for a few errors concerning noncomputability, redundant match alternatives, and invalid inductive declarations.
- #8990 adds missing doc-strings for grind's internal algebra typeclasses, for inclusion in the reference manual.
- #8998 makes the docstrings related to `Format` and `Repr` have consistent formatting and style, and adds missing docstrings.

Server

- #8105 adds support for server-sided `RpcRef` reuse and fixes a bug where trace nodes in the InfoView would close while the file was still being processed.
- #8511 implements signature help support. When typing a function application, editors with support for signature help will now display a popup that designates the current (remaining) function type. This removes the need to remember the function signature while typing the function application, or having to constantly cycle between hovering over the function identifier and typing the application. In VS Code, the signature help can be triggered manually using `Ctrl+Shift+Space`.
- #8654 adds server-side support for a new module hierarchy component in VS Code that can be used to navigate both the import tree of a module and the imported-by tree of a module. Specifically, it implements new requests `$/lean/prepareModuleHierarchy`, `$/lean/moduleHierarchy/imports` and `$/lean/moduleHierarchy/importedBy`. These requests are not supported by standard LSP. Companion PR at leanprover/vscode-lean4#620.
- #8699 adds support to the server for the new module setup process by changing how `lake setup-file` is used.

- #8868 ensures that code actions do not have to wait for the full file to elaborate. This regression was accidentally introduced in #7665.
- #9019 fixes a bug where semantic highlighting would only highlight keywords that started with an alphanumeric character. Now, it uses `Lean.isIdFirst`.

Lake

- #7738 makes memoization of built-in facets toggleable through a `memoize` option on the facet configuration. Built-in facets which are essentially aliases (e.g., `default`, `o`) have had memoization disabled.
- #8447 makes use of `lean --setup` in Lake builds of Lean modules and adds Lake support for the new `.olean` artifacts produced by the module system.
- #8613 changes the Lake version syntax (to `5.0.0-src+<commit>`) to ensure it is a well-formed SemVer,
- #8656 enables auto-implicits in the Lake math template. This resolves an issue where new users sometimes set up a new project for math formalization and then quickly realize that none of the code samples in our official books and docs that use auto-implicits work in their projects. With the introduction of inlay hints for auto-implicits, we consider the auto-implicit UX to be sufficiently usable that they can be enabled by default in the math template. Notably, this change does not affect Mathlib itself, which will proceed to disable auto-implicits.
- #8701 exports `LeanOption` in the `Lean` namespace from the `Lake` namespace. `LeanOption` was moved from `Lean` to `Lake` in #8447, which can cause unnecessary breakage without this.
- #8736 partially reverts #8024 which introduced a significant Lake performance regression during builds. Once the cause is discovered and fixed, a similar PR will be made to revert this.
- #8846 reintroduces the basics of `lean --setup` integration into Lake without the module computation which is still undergoing performance debugging in #8787.
- #8866 upgrades the `math` template for `lake init` and `lake new` to configures the new project to meet rigorous Mathlib maintenance standards. In comparison with the previous version (now available as `lake new ... math-lax`), this automatically provides:
  - Strict linting options matching Mathlib.
  - GitHub workflow for automatic upgrades to newer Lean and Mathlib releases.
  - Automatic release tagging for toolchain upgrades.
  - API documentation generated by doc-gen4 and hosted on `github.io`.
  - README with some GitHub-specific instructions.
- #8922 introduces a local artifact cache for Lake. When enabled, Lake will shared build artifacts (built files) across different instances of the same package using an input- and content-addressed cache.

- #8981 removes Lake's usage of `lean -R` and `moduleNameOfFileName` to pass module names to Lean. For workspace names, it now relies on directly passing the module name through `lean --setup`. For non-workspace modules passed to `lake lean` or `lake setup-file`, it uses a fixed module name of `_unknown`.
- #9068 fixes some bugs with the local Lake artifact cache and cleans up the surrounding API. It also adds the ability to opt-in to the cache on packages without `enableArtifactCache` set using the `LAKE_ARTIFACT_CACHE` environment variable.
- #9081 fixes a bug with Lake where the job monitor would sit on a top-level build (e.g., `mathlib/Mathlib:default`) instead of reporting module build progress.
- #9101 fixes a bug introduce by #9081 where the source file was dropped from the module input trace and some entries were dropped from the module job log.

Other

- #8702 enhances the PR release workflow to create both short format and SHA-suffixed release tags. Creates both pr-release-{PR_NUMBER} and pr-release-{PR_NUMBER}-{SHORT_SHA} tags, generates separate releases for both formats, adds separate GitHub status checks, and updates Batteries/Mathlib testing branches to use SHA-suffixed tags for exact commit traceability.
- #8710 pins the precise hash of softprops/action-gh-release to
- #9033 adds a Mathlib-like testing and feedback system for the reference manual. Lean PRs will receive comments that reflect the status of the language reference with respect to the PR.
- #9092 further updates release automation. The per-repository update scripts `script/release_steps.py` now actually performs the tests, rather than outputting a script for the release manager to run line by line. It's been tested on `v4.21.0` (i.e. the easy case of a stable release), and we'll debug its behaviour on `v4.22.0-rc1` tonight.

Lean 4.21.0 (2025-06-30)

For this release, 295 changes landed. In addition to the 100 feature additions and 83 fixes listed below there were 2 refactoring changes, 4 documentation improvements, 6 performance improvements, 2 improvements to the test suite and 98 other changes.

Highlights

'Unknown identifier' code actions

- #7665 and #8180 add support for code actions that resolve 'unknown identifier' errors by either importing the missing declaration or by changing the identifier to one from the environment.

New Language Features

- #8449 and #8516 upstream and extend the Mathlib `clear_value` tactic. Given a local definition `x : T := v`, the tactic `clear_value x` replaces it with a hypothesis `x : T`, or throws an error if the goal does not depend on the value `v`. The syntax `clear_value (h :`

`x = _`) creates a hypothesis `h : x = _` before clearing the value of `x`. Any expression definitionally equal to `x` can be used in place of the underscore. Furthermore, `clear_value *` clears all values that can be cleared, or throws an error if none can be cleared.

- #8512 adds a `value_of% ident` term that elaborates to the value of the local or global constant `ident`. This is useful for creating definition hypotheses:

```
let x := ... complicated expression ...
have hx : x = value_of% x := rfl
```

- #8450 adds a feature to the `subst` tactic so that when `x : X := v` is a local definition, `subst x` substitutes `v` for `x` in the goal and removes `x`. Previously the tactic would throw an error.
- #8037 introduces a `noConfusionType` construction that's sub-quadratic in size, and reduces faster. The previous `noConfusion` construction with two nested `match` statements is quadratic in size and reduction behavior. Using some helper definitions, a linear size construction is possible.
- #8104 makes `fun_induction` and `fun_cases` (try to) unfold the function application of interest in the goal. The old behavior can be enabled with `set_option tactic.fun_induction.unfolding false`. For `fun_cases` this does not work yet when the function's result type depends on one of the arguments, see issue #8296.
- #8171 omits cases from functional induction/cases principles that are implemented by `contradiction` (or, more generally, `False.elim`, `absurd` or `noConfusion). **Breaking change** in the sense that there are fewer goals to prove after using functional induction.
- #8106 adds a `register_linter_set` command for declaring linter sets. The `getLinterValue` function now checks if the present linter is contained in a set that has been enabled (using the `set_option` command or on the command line).
- #8267 makes `#guard_msgs` to treat `trace` messages separate from `info`, `warning` and `error`. It also introduces the ability to say `#guard_msgs (pass info)`, like `(drop info)` so far, and also adds `(check info)` as the explicit form of `(info)`, for completeness.

Library Highlights

- #8358 introduces a very minimal version of the new iterator library. It comes with list iterators and various consumers, namely `toArray`, `toList`, `toListRev`, `ForIn`, `fold`, `foldM` and `drain`. All consumers also come in a partial variant that can be used without any proofs. This limited version of the iterator library generates decent code, even with the old code generator.
- #7352 reworks the `simp` set around the `Id` monad, to not elide or unfold `pure` and `Id.run`
- #8313 changes the definition of `Vector` so it no longer extends `Array`. This prevents `Array` API from "leaking through".

Other Highlights

- Performance optimizations in `dsimp`:

  - #6973 stops `dsimp` from visiting proof terms, which should make `simp` and `dsimp` more efficient.
  - #7428 adds a `dsimp` cache to `simp`. Previously each `dsimp` call from `simp` started with a fresh cache. As a result, time spent in `simp` while compiling Mathlib is reduced by over 45%, giving an overall 8% speedup to Mathlib compilation.

- #8221 adjusts the experimental module system to not export the bodies of `defs` unless opted out by the new attribute `@[expose]` on the `def` or on a surrounding `section`.
- #8559 and #8560 fix an adversarial soundness attack described in #8554. The attack exploits the fact that `assert!` no longer aborts execution, and that users can redirect error messages.

Language

- #6973 stops `dsimp` from visiting proof terms, which should make `simp` and `dsimp` more efficient.
- #7428 adds a `dsimp` cache to `simp`. Previously each `dsimp` call from `simp` started with a fresh cache. As a result, time spent in `simp` while compiling Mathlib is reduced by over 45%, giving an overall 8% speedup to Mathlib compilation.
- #7631 fixes `Lean.Level.mkIMaxAux` (`mk_imax` in the kernel) such that `imax 1 u` reduces to `u`.
- #7977 adds basic support for eta-reduction to `grind`.
- #8002 fixes an issue where "go to definition" for variables generalized by the `induction` and `cases` tactic did not work. Closes #2873.
- #8024 adds the `--setup` option to the `lean` CLI. It takes a path to a JSON file containing information about a module's imports and configuration, superseding that in the module's own file header. This will be used by Lake to specify paths to module artifacts (e.g., oleans and ileans) separate from the `LEAN_PATH` schema.
- #8037 introduces a `noConfusionType` construction that's sub-quadratic in size, and reduces faster.
- #8104 makes `fun_induction` and `fun_cases` (try to) unfold the function application of interest in the goal. The old behavior can be enabled with `set_option tactic.fun_induction.unfolding false`. For `fun_cases` this does not work yet when the function's result type depends on one of the arguments, see issue #8296.
- #8106 adds a `register_linter_set` command for declaring linter sets. The `getLinterValue` function now checks if the present linter is contained in a set that has been enabled (using the `set_option` command or on the command line).
- #8169 makes the whitespace handling in the syntax of `omit` and `include` consistent with `variable`.
- #8171 omits cases from functional induction/cases principles that are implemented by `contradiction` (or, more generally, `False.elim`, `absurd` or `noConfusion`). Breaking change in the sense that there are fewer goals to prove after using functional induction.

- #8196 improves the E-matching pattern inference procedure in `grind`. Consider the following theorem:

  ```
  @[grind →]
  theorem eq_empty_of_append_eq_empty {xs ys : Array α} (h : xs ++
  ys = #[]) : xs = #[] ∧ ys = #[] :=
    append_eq_empty_iff.mp h
  ```

  Before this PR, `grind` inferred the following pattern:

  ```
  @HAppend.hAppend _ _ _ _ #2 #1
  ```

  Note that this pattern would match any ++ application, even if it had nothing to do with arrays. With this PR, the inferred pattern becomes:

  ```
  @HAppend.hAppend (Array #3) (Array _) (Array _) _ #2 #1
  ```

  With the new pattern, the theorem will not be considered by `grind` for goals that do not involve `Array`s.

- #8198 fixes an issue in the theory propagation used in `grind`. When two equivalence classes are merged, the core may need to push additional equalities or disequalities down to the satellite theory solvers (e.g., `cutsat`, `comm ring`, etc). Some solvers (e.g. `cutsat`) assume that all of the core's invariants hold before they receive those facts. Propagating immediately therefore risks violating a solver's pre-conditions midway through the merge. To decouple the merge operation from propagation and to keep the core solver-agnostic, this PR adds the helper type `PendingTheoryPropagation`.

- #8208 reduces the need for defeq in frequently used bv_decide rewrite by turning them into simprocs that work on structural equality instead. As the intended meaning of these rewrites is to simply work with structural equality anyways this should not change the proving power of `bv_decide`'s rewriter but just make it faster on certain very large problems.

- #8209 fixes a nondeterminism issue in the `grind` tactic. It was a bug in the model-based theory combination module.

- #8221 adjusts the experimental module system to not export the bodies of `def`s unless opted out by the new attribute `@[expose]` on the `def` or on a surrounding `section`.

- #8224 adds diagnostic information for the commutative ring procedure in `grind`.

- #8226 fixes the `simplifyBasis` procedure in the commutative ring procedure in `grind`.

- #8231 changes the behaviour of `apply?` so that the `sorry` it uses to close the goal is non-synthetic. (Recall that correct use of synthetic sorries requires that the tactic also generates an error message, which we don't want to do in this situation.) This change defends against the problem reported in #8212.

- #8232 fixes elaboration of constants in the `rewrite` tactic. previously, `rw [eq_self]` would elaborate `eq_self` twice, and add it to the infotree twice. This would lead to the "Expected type" being delaborated with an unknown universe metavariable.

- #8241 changes the behavior of the `rename` tactic to skip over implementation detail hypotheses when finding a hypothesis to rename.
- #8254 fixes unintended inlining of `ToJson`, `FromJson`, and `Repr` instances, which was causing exponential compilation times in `deriving` clauses for large structures.
- #8259 clarifies the invalid field notation error when projected value type is a metavariable.
- #8260 clarifies the invalid dotted identifier notation error when the type is a sort.
- #8261 adjusts the error message when `apply` fails to unify. It is clearer about distinguishing the term being applied and the goal, as well as distinguishing the "conclusion" of the given term and the term itself.
- #8262 improves the type-as-hole error message. Type-as-hole error for theorem declarations should not admit the possibility of omitting the type entirely.
- #8264 rewords the `application type mismatch` error message by more specifically mentioning that the problem is with the final argument. This is useful when the same argument is passed to the function multiple times.
- #8267 makes `#guard_msgs` to treat `trace` messages separate from `info`, `warning` and `error`. It also introduce the ability to say `#guard_msgs (pass info`, like `(drop info)` so far, and also adds `(check info)` as the explicit form of `(info)`, for completeness.
- #8270 makes the enum pass of `bv_decide` handle enum types that are universe polymorphic.
- #8271 changes `addPPExplicitToExposeDiff` to show universe differences and to visit into projections, e.g.:

  ```
  error: tactic 'rfl' failed, the left-hand side
    (Test.mk (∀ (x : PUnit.{1}), True)).1
  is not definitionally equal to the right-hand side
    (Test.mk (∀ (x : PUnit.{2}), True)).1
  ```

  for

  ```
  inductive Test where
    | mk (x : Prop)
  ```

- #8275 ensures the congruence closure in `grind` and find non-dependent arrow congruences. That is, it can apply the `implies_congr` theorem.
- #8276 adds the instances `Grind.CommRing (Fin n)` and `Grind.IsCharP (Fin n) n`. New tests:

  ```
  example (x y z : Fin 13) :
      (x + y + z) ^ 2 = x ^ 2 + y ^ 2 + z ^ 2 + 2 * (x * y + y * z
  + z * x) := by
    grind +ring
  ```

- #8277 improves the generation of `.induct_unfolding` by rewriting `match` statements more reliably, using the new "congruence equations" introduced in #8284. Fixes #8195.
- #8280 the support for arrows in the congruence closure procedure used in `grind`.
- #8281 improves the module used to prove auxiliary type cast equalities in `grind`.
- #8284 adds a new variant of equations for matchers, namely "congruence equations" that generalize the normal matcher equations. They have unrestricted left-hand-sides, extra equality assumptions relating the discriminiants with the patterns and thus prove heterogenous equalities. In that sense they combine congruence with rewriting. They can be used to rewrite matcher applications where, due to dependencies, `simp` would fail to rewrite the discriminants, and will be used when producing the unfolding induction theorems.
- #8285 fixes "declaration has free variables" errors when generating a splitter for a match statement with named patterns. Fixes #8274.
- #8299 implements a missing preprocessing step in `grind`: abstract metavariables in the goal
- #8301 unfolds functions in the unfolding induction principle properly when they use `bif` (a.k.a. `Bool.cond`).
- #8302 lets `cases` fail gracefully when the motive has an complex argument whose type is dependent type on the targets. While the `induction` tactic can handle this well, `cases` does not. This change at least gracefully degrades to not instantiating that motive parameter. See issue #8296 for more details on this issue.
- #8303 fixes missing occurrences of `foldProjs` in `grind`.
- #8306 makes it possible for `bv_decide` to tackle situations for its enum type preprocessing where the enums themselves are use in a dependently type context (for example inside of a `GetElem` body) and thus not trivially accessible to `simp` for rewriting. To do this we drop`GetElem` on `BitVec` as well as `dite` as early as possible in the pipeline.
- #8321 lets the termination argument inference consider negations of Nat comparisons. Fixes #8257.
- #8323 adds support for bv_decide to understand `BitVec.reverse` in bitblasting.
- #8330 improves support for structure extensionality in `grind`. It now uses eta expansion for structures instead of the extensionality theorems generated by `[ext]`. Examples:

  ```
  opaque f (a : Nat) : Nat × Bool
  ```

- #8338 improves the error messages displayed in `inductive` declarations when type parameters are invalid or absent.
- #8341 fixes the `propagateCtor` constraint propagator used in `grind`.
- #8343 splits `Lean.Grind.CommRing` into 4 typeclasses, for semirings and noncommutative rings. This does not yet change the behaviour of `grind`, which expects to find all 4 typeclasses. Later we will make some generalizations.
- #8344 fixes term normalization issues in `grind`, and the new option `grind +etaStruct`.

- #8347 adds draft typeclasses for `grind` to process facts about ordered modules. These interfaces will evolve as the implementation develops.
- #8354 makes sure that when generating the unfolding functional induction theorem, `mdata` does not get in the way.
- #8356 tries harder to clean internals of the argument packing of n-ary functions from the functional induction theorem, in particular the unfolding variant
- #8359 improves the functional cases principles, by making a more educated guess which function parameters should be targets and which should remain parameters (or be dropped). This simplifies the principles, and increases the chance that `fun_cases` can unfold the function call.
- #8361 fixes a bug in the `cases` tacic introduced in #3188 that arises when cases (not induction) is used with a non-atomic expression in using and the argument indexing gets confused.
- #8363 unifies various ways of naming auxiliary declarations in a conflict-free way and ensures the method is compatible with diverging branches of elaboration such as parallelism or Aesop-like backtracking+replaying search.
- #8365 fixes the transparency mode for ground patterns. This is important for implicit instances. Here is a mwe for an issue detected while testing `grind` in Mathlib.

  ```
  example (a : Nat) : max a a = a := by
    grind
  ```

- #8368 improves the error messages produced by invalid pattern-match alternatives and improves parity in error placement between pattern-matching tactics and elaborators.
- #8369 fixes a type error at `instantiateTheorem` function used in `grind`. It was failing to instantiate theorems such as

  ```
  theorem getElem_reverse {xs : Array α} {i : Nat} (hi : i <
  xs.reverse.size)
      : (xs.reverse)[i] = xs[xs.size - 1 - i]'(by simp at hi;
  omega)
  ```

  in examples such as

  ```
  example (xs : Array Nat) (w : xs.reverse = xs) (j : Nat) (hj : 0
  ≤ j) (hj' : j < xs.size / 2)
      : xs[j] = xs[xs.size - 1 - j]
  ```

  generating the issue

  ```
    [issue] type error constructing proof for Array.getElem_reverse
        when assigning metavariable ?hi with
          ‹j < xs.toList.length›
        has type
          j < xs.toList.length : Prop
  ```

```
but is expected to have type
  j < xs.reverse.size : Prop
```

- #8375 ensures that using `mapError` to expand an error message uses `addMessageContext` to include the current context, so that expressions are rendered correctly. Also adds a `preprendError` variant with a more convenient argument order for the common cases of prepending-and-indenting.
- #8403 adds missing monotonicity lemmas for universal quantifiers, that are used in defining (co)inductive predicates.
- #8410 fixes a case-splitting heuristic in `grind` and simplifies the proof for test `grind_palindrome2.lean`.
- #8412 fixes the `markNestedProofs` preprocessor used in `grind`. There was a missing case (e.g., `Expr.mdata`)
- #8413 implements normalization rules that pull universal quantifiers across disjunctions. This is a common normalization step performed by first-order theorem provers.
- #8417 introduces `Lean.Grind.Field`, proves that a `IsCharP 0` field satisfies `NoNatZeroDivisors`, and sets up some basic (currently failing) tests for `grind`.
- #8426 adds the attribute `[grind?]`. It is like `[grind]` but displays inferred E-matching patterns. It is a more convinient than writing. Thanks @kim-em for suggesting this feature.

  ```
  set_option trace.grind.ematch.pattern true
  ```

  also improves some tests, and adds helper function `ENode.isRoot`.
- #8429 adds `Lean.Grind.Ring.IsOrdered`, and cleans up the ring/module grind API. These typeclasses are at present unused, but will support future algorithmic improvements in `grind`.
- #8437 fixes `split` in the presence of metavariables in the target.
- #8438 ensures that `grind` diagnostics are obtained even when `maxHeartbeats` is reached. also removes some dead code.
- #8440 implements non-chronological backtracking for the `grind` tactic. This feature ensures that `grind` does not need to process irrelevant branches after performing a case-split that is not relevant. It is not just about performance, but also the size of the final proof term. The new test demonstrates this feature in practice.

  ```
  -- In the following test, the first 8 case-splits are irrelevant,
  -- and non-choronological backtracking is used to avoid searching
  -- (2^8 - 1) irrelevant branches
  /--
  trace:
  [grind.split] p8 ∨ q8, generation: 0
  [grind.split] p7 ∨ q7, generation: 0
  [grind.split] p6 ∨ q6, generation: 0
  [grind.split] p5 ∨ q5, generation: 0
  [grind.split] p4 ∨ q4, generation: 0
  ```

```
[grind.split] p3 ∨ q3, generation: 0
[grind.split] p2 ∨ q2, generation: 0
[grind.split] p1 ∨ q1, generation: 0
[grind.split] ¬p ∨ ¬q, generation: 0
-/
#guard_msgs (trace) in
set_option trace.grind.split true in
theorem ex
    : p ∨ q →
      ¬ p ∨ q →
      p ∨ ¬ q →
      ¬ p ∨ ¬ q →
      p1 ∨ q1 →
      p2 ∨ q2 →
      p3 ∨ q3 →
      p4 ∨ q4 →
      p5 ∨ q5 →
      p6 ∨ q6 →
      p7 ∨ q7 →
      p8 ∨ q8 →
      False := by
  grind (splits := 10)
```

- #8443 adds the lemmas about ordered rings and ordered fields which will be needed by the new algebraic normalization components of `grind`.
- #8449 upstreams and extends the Mathlib `clear_value` tactic. Given a local definition `x : T := v`, the tactic `clear_value x` replaces it with a hypothesis `x : T`, or throws an error if the goal does not depend on the value `v`. The syntax `clear_value x with h` creates a hypothesis `h : x = v` before clearing the value of `x`. Furthermore, `clear_value *` clears all values that can be cleared, or throws an error if none can be cleared.
- #8450 adds a feature to the `subst` tactic so that when `x : X := v` is a local definition, `subst x` substitutes `v` for `x` in the goal and removes `x`. Previously the tactic would throw an error.
- #8466 fixes another instance of the `grind` issue "unexpected kernel projection term during internalization".
- #8472 avoids name resolution blocking on the elaboration of a theorem's proof when looking up the theorem name.
- #8479 implements hash-consing for `grind` that takes alpha equivalence into account.
- #8483 ensures `grind` reuses the `simp` cache between different calls. Recall that `grind` uses `simp` to normalize terms during internalization.
- #8491 fixes the behavior of `simp_all?` and `simp_all?!`, aligning them with `simp_all` and `simp_all!` respectively.

- #8506 implements `match`-expressions in `grind` using `match` congruence equations. The goal is to minimize the number of `cast` operations that need to be inserted, and avoid `cast` over functions. The new approach support `match`-expressions of the form `match h : ... with ...`.
- #8512 adds a `value_of% ident` term that elaborates to the value of the local or global constant `ident`. This is useful for creating definition hypotheses:

  ```
  let x := ... complicated expression ...
  have hx : x = value_of% x := rfl
  ```

- #8516 is a followup to #8449 to refine the syntax of `clear_value`. The syntax for adding equality hypotheses before clearing values is now `clear_value (h : x = _)`. Any expression definitionally equal to `x` can be used in place of the underscore.
- #8536 fixes the support for `LawfulBEq` and `BEq` in `grind`.
- #8541 ensures that for any nested proof `h : p` in a goal, we propagate that `p` is true in the `grind` tactic.
- #8542 fixes two inappropriate uses of `whnfD` in `grind`. They were potential performance foot guns, and were producing unexpected errors since `whnfD` is not consistently used (and it should not be) in all modules.
- #8544 implements support for over-applied `ite` and `dite` applications in the `grind` tactic. It adds support for propagation and case-split.
- #8549 fixes the hash function used to implement congruence closure in `grind`. The hash of an `Expr` must not depend on whether the expression has been internalized or not.
- #8564 simplifies the interface between the `grind` core and the cutsat procedure. Before this PR, core would try to minimize the number of numeric literals that have to be internalized in cutsat. This optimization was buggy (see `grind_cutsat_zero.lean` test), and produced counterintuitive counterexamples.
- #8569 adds support for generalized E-match patterns to arbitrary theorems.
- #8570 fixes some issues in the E-matching generalized pattern support after the update stage0.
- #8572 adds some generalized `Option` theorems for `grind` . The avoid `casts` operations during E-matching.
- #8576 sets `ring := true` by default in `grind`. It also fixes a bug in the reification procedure, and improves the term internalization in the ring and cutsat modules.

Library

- #7352 reworks the `simp` set around the `Id` monad, to not elide or unfold `pure` and `Id.run`
- #7995 adds a verification of `Array.qsort` properties, trying to use `grind` and `fun_induction` where possible. Currently this is in the `tests/` folder, but once `grind` is ready for production use we will move it out into the library.
- #8182 adds `ofList_eq_insertMany_empty` lemmas for all the hash/tree map types, with the exception of `Std.HashSet.Raw.ofList_eq_insertMany_empty`.

- #8188 takes the existing `getElem_map` statements for `HashMap` variants (also `getElem?`, `getElem!`, and `getD` statements), adds a prime to their name and an explanatory comment, and replaces the unprimed statement with a simpler statement that is only true with `LawfulBEq` present. The original statements which were simp lemmas are now low priority simp lemmas, so the nicer statements should fire when `LawfulBEq` is available.
- #8202 adds an inference that was repeatedly needed when proving `BitVec.msb_sdiv`, and is the symmetric version of `BitVec.one_eq_zero_iff`
- #8206 shows that negating a bitvector created from a natural number equals creating a bitvector from the the negative of that number (as an integer).
- #8216 completes adding `@[grind]` annotations for `Option` lemmas, and incidentally fills in some `Option` API gaps/defects.
- #8218 continues adding `@[grind]` attributes for List/Array/Vector, particularly to the lemmas involving the `toList`/`toArray` functions.
- #8246 add `@[grind]` annotations for HashMap and variants.
- #8272 adds lemmas about the length and use of `[]?` on results of `List.intersperse`.
- #8291 changes the statements of `Fin` lemmas to use `[NeZero n] (i : Fin n)` rather than `(i : Fin (n+1))` where possible.
- #8298 adds various `Option` lemmas and defines `Option.filterM` for applicative functors.
- #8313 changes the definition of `Vector` so it no longer extends `Array`. This prevents `Array` API from "leaking through".
- #8315 splits `Std.Classes.Ord` into `Std.Classes.Ord.Basic` (with few imports) and `Std.Classes.Ord.SInt` and `Std.Classes.Ord.Vector`. These changes avoid importing `Init.Data.BitVec.Lemmas` unnecessarily into various basic files. As the new import-only file `Std.Classes.Ord` imports all three of these, end-users are not affected.
- #8318 is follow-up to #8272, combining the conditional lemmas for `getElem_intersperse` into a single lemma with an `if` on the RHS.
- #8327 adds `@[grind]` annotations to the generic `getElem?_eq_none_iff`, `isSome_getElem?`, and `get_getElem?`.
- #8328 adds the `@[grind =]` attribute to all `contains_iff_mem` lemmas.
- #8331 improves the docstring for `PlainDateTime.now` and its variants.
- #8346 adds some missing lemmas about consequences of positivity/non-negativity of `a * b : Int`.
- #8349 fixes the signature of the intended `Inhabited` instance for `ExtDHashMap`.
- #8357 adds variants of `dite_eq_left_iff` that will be useful in a future PR.
- #8358 introduces a very minimal version of the new iterator library. It comes with list iterators and various consumers, namely `toArray`, `toList`, `toListRev`, `ForIn`, `fold`, `foldM` and `drain`. All consumers also come in a partial variant that can be used without any proofs. This limited version of the iterator library generates decent code, even with the old code generator.

- #8378 improves and extends the api around `Ord` and `Ordering`.
- #8379 adds missing `Option` lemmas.
- #8380 provides simple lemmas about `toArray`, `toList` and `toListRev` for the iterator library.
- #8384 provides lemmas about the behavior of `step`, `toArray`, `toList` and `toListRev` on list iterators created with `List.iter` and `List.iterM`.
- #8389 adds the `List/Array/Vector.ofFnM`, the monadic analogues of `ofFn`, along with basic theory.
- #8392 corrects some `Array` lemmas to be about `Array` not `List`.
- #8397 cleans up many duplicate instances (or, in some cases, needlessly duplicated `def X := ...; instance Y := X`).
- #8399 adds variants of `HashMap.getElem?_filter` that assume `LawfulBEq` and have a simpler right-hand-side. `simp` can already achieve these, via rewriting with `getKey_eq` under the lambda. However `grind` can not, and these lemmas help `grind` work with `HashMap` goals. There are variants for all variants of `HashMap`, `getElem?`/`getElem`/`getElem!`/`getD`, and for `filter` and `filterMap`.
- #8405 provides lemmas about the loop constructs `ForIn`, `fold`, `foldM` and `drain` and their relation to each other in the context of iterators.
- #8418 provides the `take` iterator combinator that transforms any iterator into an iterator that stops after a given number of steps. The change contains the implementation and lemmas.
- #8422 adds `LT` and `Decidable LT` instances for `Std.Time.Timestamp` and `Std.Time.Duration`.
- #8434 adds the equivalent of `List.take_cons` about `List.drop`.
- #8435 upstreams the `LawfulMonadLift(T)` classes, lemmas and instances from Batteries into Core because the iterator library needs them in order to prove lemmas about the `mapM` operator, which relies on `MonadLiftT`.
- #8445 adds a `@[simp]` lemma, and comments explaining that there is intentionally no verification API for `Vector.take`, `Vector.drop`, or `Vector.tail`, which should all be rewritten in terms of `Vector.extract`.
- #8446 adds basic `@[grind]` annotations for `TreeMap` and its variants. Likely more annotations will be added after we've explored some examples.
- #8451 provides the iterator combinator `filterMap` in a pure and monadic version and specializations `map` and `filter`. This new combinator allows to apply a function to the emitted values of a stream while filtering out certain elements.
- #8460 adds further `@[grind]` annotations for `Option`, as follow-up to the recent additions to the `Option` API in #8379 and #8298.
- #8465 adds further lemmas about `LawfulGetElem`, including marking some with `@[grind]`.

- #8470 adds `@[simp]` to `getElem_pos/neg` (similarly for `getElem!`). These are often already simp lemmas for concrete types.
- #8482 adds preliminary `@[grind]` annotations for `List.Pairwise` and `List.Nodup`.
- #8484 provides the iterator combinator `zip` in a pure and monadic version.
- #8492 adds `simp` lemmas for `toInt_*` and `toNat_*` with arithmetic operation given the hypothesis of no-overflow (`toNat_add_of_not_uaddOverflow`, `toInt_add_of_not_saddOverflow`, `toNat_sub_of_not_usubOverflow`, `toInt_sub_of_not_ssubOverflow`, `toInt_neg_of_not_negOverflow`, `toNat_mul_of_not_umulOverflow`, `toInt_mul_of_not_smulOverflow`). In particular, these are `simp` since (1) the `rhs` is strictly simpler than the `lhs` and (2) this version is also simpler than the standard operation when the hypothesis is available.
- #8493 provides the iterator combinators `takeWhile` (forwarding all emitted values of another iterator until a predicate becomes false) `dropWhile` (dropping values until some predicate on these values becomes false, then forwarding all the others).
- #8497 adds preliminary grind annotations for `List.Sublist/IsInfix/IsPrefix/IsSuffix`, along with test cases.
- #8499 changes the definition of `Array.ofFn.go` to use recursion on `Nat` (rather than well-founded recursion). This resolves a problem reported on zulip).
- #8513 removes the `@[reducible]` annotation on `Array.size`. This is probably best gone anyway in order to keep separation between the `List` and `Array` APIs, but it also helps avoid uselessly instantiating `Array` theorems when `grind` is working on `List` problems.
- #8515 removes the prime from `Fin.ofNat'`: the old `Fin.ofNat` has completed its 6 month deprecation cycle and is being removed.
- #8527 adds `grind` annotations for theorems about `List.countP` and `List.count`.
- #8552 provides array iterators (`Array.iter(M)`, `Array.iterFromIdx(M)`), infinite iterators produced by a step function (`Iter.repeat`), and a `ForM` instance for finite iterators that is implemented in terms of `ForIn`.
- #8620 removes the `NatCast (Fin n)` global instance (both the direct instance, and the indirect one via `Lean.Grind.Semiring`), as that instance causes causes `x < n` (for `x : Fin k`, `n : Nat`) to be elaborated as `x < ↑n` rather than `↑x < n`, which is undesirable. Note however that in Mathlib this happens anyway!

Compiler

- #8211 adds support for generating IR from the LCNF representation of the new compiler.
- #8236 fixes an issue where the combination of `extern_lib` and `precompileModules` would lead to "symbol not found" errors.
- #8268 optimizes lean_nat_shiftr for scalar operands. The new compiler converts Nat divisions into right shifts, so this now shows up as hot in some profiles.

- #8308 makes the new compiler's specialization pass compute closures the same way as the old compiler, in particular when it comes to variables captured by lambdas.
- #8367 adds a new `structProjCases` pass to the new compiler, analogous to the `struct_cases_on` pass in the old compiler, which converts all projections from structs into `cases` expressions. When lowered to IR, this causes all of the projections from a single structure to be grouped together, which is an invariant relied upon by the IR RC passes (at least for linearity, if not general correctness).
- #8409 adds support to LCNF for native UInt8/UInt16/UInt32/UInt64 literals.
- #8456 adds support for primitive USize literals in LCNF.
- #8458 adds closed term extraction to the new compiler, closely following the approach in the old compiler. In the future, we will explore some ideas to improve upon this approach.
- #8462 enables the LCNF extractClosed pass by default.
- #8468 switches the LCNF baseExt/monoExt environment extensions to use a custom environment extension that uses a PersistentHashMap. The optimizer relies upon the ability to update a decl multiple times, which does not work with SimplePersistentEnvExtension.
- #8502 changes the new compiler to use the kernel environment to find definitions, which causes compilation to be skipped when the decl had a kernel error (e.g. due to an unresolved metavariable). This matches the behavior of the old compiler.
- #8521 makes LCNF.toMono recursively process jmp args.
- #8523 moves the new compiler's noncomputable check into toMono, matching the recent change in the old compiler. This is mildly more complicated because we can't throw an error at the mere use of a constant, we need to check for a later relevant use. This is still a bit more conservative than it could theoretically be around join points and local functions, but it's hard to imagine that mattering in practice (and we can easily enable it if it does).
- #8535 extracts more Nats (and their downstream users) in extractClosed by fixing a silly oversight in the logic.
- #8540 changes the LCNF specialize pass to allow ground variables to depend on local fun decls (with no non-ground free variables). This enables specialization of Monad instances that depend on local lambdas.
- #8559 fixes an adversarial soundness attack described in #8554. The attack exploits the fact that `assert!` no longer aborts execution, and that users can redirect error messages. Another PR will implement the same fix for `Expr.Data`.
- #8560 is similar to #8559 but for `Expr.mkData`. This vulnerability has not been exploited yet, but adversarial users may find a way.
- #8561 increases maxHeartbeats in the isDefEqProjIssue test, because when running under the new compiler the `run_meta` call includes the allocations of the compiler itself. With the old compiler, many of the corresponding allocations were internal to C++ code and would not increase the heartbeat count.

- #8565 makes the LCNF specialization pass only treat type/instance params as ground vars. The current policy was too liberal and would result on computations being floated into specialized loops.
- #8566 changes the LCNF constant folding pass to not convert Nat multiplication to a left shift by a power of 2. The fast path test for this is sufficiently complex that it's simpler to just use the fast path for multiplication.
- #8575 makes LCNF's simpAppApp? bail out on trivial aliases as intended. It seems that there was a typo in the original logic, and this PR also extends it to include aliases of global constants rather than just local vars.
- #8582 fixes an accidental dropping of state in Param.toMono. When this code was originally written, there was no other state besides `typeParams`.

Pretty Printing

- #8041 changes the behavior of `pp.showLetValues` to use a hoverable ⋯ to hide let values. This is now false by default, and there is a new option `pp.showLetValues.threshold` for allowing small expressions to be shown anyway. For tactic metavariables, there is an additional option `pp.showLetValues.tactic.threshold`, which by default is set to the maximal value, since in tactic states local values are usually significant.
- #8372 modifies the pretty printer to use `have` syntax instead of `let_fun` syntax.
- #8457 fixes an issue when including a hard line break in a `Format` that caused subsequent (ordinary) line breaks to be erroneously flattened to spaces.
- #8504 modifies the pretty printer so that dot notation is used for class parent projections. Previously, dot notation was never used for classes.

Documentation

- #8199 adds a style guide for documentation, including both general principles and docstring-specific concerns.

Server

- #7665 and #8180 add support for code actions that resolve 'unknown identifier' errors by either importing the missing declaration or by changing the identifier to one from the environment.
- #8091 improves the performance of the workspace symbol request.
- #8242 fixes the 'goals accomplished' diagnostics. They were accidentally broken in #7902.
- #8350 changes namespace completion to use the same algorithm as declaration identifier completion, which makes it use the short name (last name component) for completions instead of the full name, avoiding namespace duplications.
- #8362 fixes a bug where the unknown identifier code actions wouldn't work correctly for some unknown identifier error spans and adjusts several unknown identifier spans to actually end on the identifier in question.

Lake

- #8383 fixes the use of `import Lake` with precompiled modules, which was previously broken on MacOS.
- #8411 fixes a doc bug in the Resolve.lean; in reverse order, B comes before A
- #8528 fixes the heuristic Lake uses to determine whether a `lean_lib` can be loaded via `lean --plugin` rather than `lean --load-dynlib`. Previously, a mismatch between the single root's name and the library's name would not be caught and cause loading to fail.
- #8529 changes `lake lean` and `lake setup-file` to precompile the imports of non-workspace files using the the import's whole library. This ensures that additional link objects are linked and available during elaboration.
- #8539 changes Lake to use relative path for the Lean messages produced by a module build. This makes the message portable across different machines, which is useful for Mathlib's cache.

Other

- #8192 includes upgrades to the `release_checklist.py` script prepared while releasing v4.20.0-rc1.
- #8366 adds the `expose` attribute to `Ordering.then`. This is required for building with the new compiler, but works fine with the old compiler because it silently ignores the missing definition.

Lean 4.20.1 (2025-06-04)

The 4.20.1 point release addresses a metaprogramming regression in `Lean.Environment.addDeclCore` (#8610).

Lean 4.20.0 (2025-06-02)

For this release, 346 changes landed. In addition to the 108 feature additions and 85 fixes listed below there were 6 refactoring changes, 7 documentation improvements, 8 performance improvements, 4 improvements to the test suite and 126 other changes.

Highlights

The Lean v4.20.0 release brings multiple new features, bug fixes, improvements to Lake, and groundwork for the module system.

Language Features

- #6432 implements tactics called `extract_lets` and `lift_lets` that manipulate `let`/`let_fun` expressions. The `extract_lets` tactic creates new local declarations extracted from any `let` and `let_fun` expressions in the main goal. For top-level lets in the target, it is like the `intros` tactic, but in general it can extract lets from deeper subexpressions

as well. The `lift_lets` tactic moves `let` and `let_fun` expressions as far out of an expression as possible, but it does not extract any new local declarations. The option `extract_lets +lift` combines these behaviors.

- #7806 modifies the syntaxes of the `ext`, `intro` and `enter` conv tactics to accept _. The introduced binder is an inaccessible name.
- #7830 modifies the syntax of `induction`, `cases`, and other tactics that use `Lean.Parser.Tactic.inductionAlts`. If a case omits `=> ...` then it is assumed to be `=> ?_`. Example:

```
example (p : Nat × Nat) : p.1 = p.1 := by
  cases p with | _ p1 p2
  /-
  case mk
  p1 p2 : Nat
  ⊢ (p1, p2).fst = (p1, p2).fst
  -/
```

This works with multiple cases as well. Example:

```
example (n : Nat) : n + 1 = 1 + n := by
  induction n with | zero | succ n ih
  /-
  case zero
  ⊢ 0 + 1 = 1 + 0

  case succ
  n : Nat
  ih : n + 1 = 1 + n
  ⊢ n + 1 + 1 = 1 + (n + 1)
  -/
```

The `induction n with | zero | succ n ih` is short for `induction n with | zero | succ n ih => ?_`, which is short for `induction n with | zero => ?_ | succ n ih => ?_`. Note that a consequence of parsing is that only the last alternative can omit =>. Any =>-free alternatives before an alternative with => will be a part of that alternative.

- #7831 adds extensibility to the `evalAndSuggest` procedure used to implement `try?`. Users can now implement their own handlers for any tactic.

```
-- Install a `TryTactic` handler for `assumption`
@[try_tactic assumption]
def evalTryApply : TryTactic := fun tac => do
  -- We just use the default implementation, but return a
different tactic.
  evalAssumption tac
  `(tactic| (trace "worked"; assumption))

/-- info: Try this: · trace "worked"; assumption -/
#guard_msgs (info) in
```

```
example (h : False) : False := by
  try? (max := 1) -- at most one solution

-- `try?` uses `evalAndSuggest` the attribute `[try_tactic]` is
used to extend `evalAndSuggest`.
-- Let's define our own `try?` that uses `evalAndSuggest`
elab stx:"my_try?" : tactic => do
  -- Things to try
  let toTry ← `(tactic| attempt_all | assumption | apply True |
rfl)
  evalAndSuggest stx toTry

/--
info: Try these:
• · trace "worked"; assumption
• rfl
-/
#guard_msgs (info) in
example (a : Nat) (h : a = a) : a = a := by
  my_try?
```

- #8055 adds an implementation of an async IO multiplexing framework as well as an implementation of it for the `Timer` API in order to demonstrate it.
- #8088 adds the "unfolding" variant of the functional induction and functional cases principles, under the name `foo.induct_unfolding` resp. `foo.fun_cases_unfolding`. These theorems combine induction over the structure of a recursive function with the unfolding of that function, and should be more reliable, easier to use and more efficient than just case-splitting and then rewriting with equational theorems.

  For example instead of

  ```
  ackermann.induct
    (motive : Nat → Nat → Prop)
    (case1 : ∀ (m : Nat), motive 0 m)
    (case2 : ∀ (n : Nat), motive n 1 → motive (Nat.succ n) 0)
    (case3 : ∀ (n m : Nat), motive (n + 1) m → motive n (ackermann
  (n + 1) m) → motive (Nat.succ n) (Nat.succ m))
    (x x : Nat) : motive x x
  ```

  one gets

  ```
  ackermann.fun_cases_unfolding
    (motive : Nat → Nat → Nat → Prop)
    (case1 : ∀ (m : Nat), motive 0 m (m + 1))
    (case2 : ∀ (n : Nat), motive n.succ 0 (ackermann n 1))
    (case3 : ∀ (n m : Nat), motive n.succ m.succ (ackermann n
  (ackermann (n + 1) m)))
    (x✝ x✝¹ : Nat) : motive x✝ x✝¹ (ackermann x✝ x✝¹)
  ```

- #8097 adds support for inductive and coinductive predicates defined using lattice theoretic structures on `Prop`. These are syntactically defined using `greatest_fixpoint` or `least_fixpoint` termination clauses for recursive `Prop`-valued functions. The

functionality relies on `partial_fixpoint` machinery and requires function definitions to be monotone. For non-mutually recursive predicates, an appropriate (co)induction proof principle (given by Park induction) is generated.

Library Highlights

#8004 adds extensional hash maps and hash sets under the names `Std.ExtDHashMap`, `Std.ExtHashMap` and `Std.ExtHashSet`. Extensional hash maps work like regular hash maps, except that they have extensionality lemmas which make them easier to use in proofs. This however makes it also impossible to regularly iterate over its entries.

Other notable library developments in this release include:

- Updates to the `Option` API,
- Async runtime developments: added support for multiplexing via UDP and TCP sockets, as well as channels,
- New `BitVec` definitions related to overflow handling,
- New lemmas for `Nat.lcm`, and `Int` variants for `Nat.gcd` and `Nat.lcm`,
- Upstreams from Mathlib related to `Nat` and `Int`,
- Additions to numeric types APIs, such as `UIntX.ofInt`, `Fin.ofNat'_mul` and `Fin.mul_ofNat'`, `Int.toNat_sub''`,
- Updates to `Perm` API in `Array`, `List`, and added support for `Vector`,
- Additional lemmas for `Array`/`List`/`Vector`.

Lake

- #7909 adds Lake support for building modules given their source file path. This is made use of in both the CLI and the server.

Breaking Changes

- #7474 updates `rw?`, `show_term`, and other tactic-suggesting tactics to suggest `expose_names` when necessary and validate tactics prior to suggesting them, as `exact?` already did, and it also ensures all such tactics produce hover info in the messages showing tactic suggestions.

  This introduces a **breaking change** in the `TryThis` API: the `type?` parameter of `addRewriteSuggestion` is now an `LOption`, not an `Option`, to obviate the need for a hack we previously used to indicate that a rewrite closed the goal.

- #7789 fixes `lean` potentially changing or interpreting arguments after `--run`.

  Breaking change: The Lean file to run must now be passed directly after `--run`, which accidentally was not enforced before.

- #7813 fixes an issue where `let n : Nat := sorry` in the Infoview pretty prints as `n : ℕ := sorry `«Foo:17:17». This was caused by top-level expressions being pretty printed with the same rules as Infoview hovers. Closes #6715. Refactors `Lean.Widget.ppExprTagged`; now it takes a delaborator, and downstream users should configure their own pretty printer option overrides if necessary if they used the `explicit` argument (see `Lean.Widget.makePopup.ppExprForPopup` for an example). **Breaking change:** `ppExprTagged` does not set `pp.proofs` on the root expression.
- #7855 moves `ReflBEq` to `Init.Core` and changes `LawfulBEq` to extend `ReflBEq`.

  Breaking changes:

  - The `refl` field of `ReflBEq` has been renamed to `rfl` to match `LawfulBEq`
  - `LawfulBEq` extends `ReflBEq`, so in particular `LawfulBEq.rfl` is no longer valid
- #7873 fixes a number of bugs related to the handling of the source search path in the language server, where deleting files could cause several features to stop functioning and both untitled files and files that don't exist on disc could have conflicting module names.

  See the PR description for the details on changes in URI <-> module name conversion.

  Breaking changes:

  - `Server.documentUriFromModule` has been renamed to `Server.documentUriFromModule?` and doesn't take a `SearchPath` argument anymore, as the `SearchPath` is now computed from the `LEAN_SRC_PATH` environment variable. It has also been moved from `Lean.Server.GoTo` to `Lean.Server.Utils`.
  - `Server.moduleFromDocumentUri` does not take a `SearchPath` argument anymore and won't return an `Option` anymore. It has also been moved from `Lean.Server.GoTo` to `Lean.Server.Utils`.
  - The `System.SearchPath.searchModuleNameOfUri` function has been removed. It is recommended to use `Server.moduleFromDocumentUri` instead.
  - The `initSrcSearchPath` function has been renamed to `getSrcSearchPath` and has been moved from `Lean.Util.Paths` to `Lean.Util.Path`. It also doesn't need to take a `pkgSearchPath` argument anymore.
- #7967 adds a `bootstrap` option to Lake which is used to identify the core Lean package. This enables Lake to use the current stage's include directory rather than the Lean toolchains when compiling Lean with Lean in core.

  Breaking change: The Lean library directory is no longer part of `getLeanLinkSharedFlags`. FFI users should provide this option separately when linking to Lean (e.g.. via `s!"-L{(←getLeanLibDir).toString}"`). See the FFI example for a demonstration.

Language

- #6325 ensures that environments can be loaded, repeatedly, without executing arbitrary code
- #6432 implements tactics called `extract_lets` and `lift_lets` that manipulate `let`/`let_fun` expressions. The `extract_lets` tactic creates new local declarations extracted from any `let` and `let_fun` expressions in the main goal. For top-level lets in the target, it is like the `intros` tactic, but in general it can extract lets from deeper subexpressions as well. The `lift_lets` tactic moves `let` and `let_fun` expressions as far out of an expression as possible, but it does not extract any new local declarations. The option `extract_lets +lift` combines these behaviors.
- #7474 updates `rw?`, `show_term`, and other tactic-suggesting tactics to suggest `expose_names` when necessary and validate tactics prior to suggesting them, as `exact?` already did, and it also ensures all such tactics produce hover info in the messages showing tactic suggestions.
- #7797 adds a monolithic `CommRing` class, for internal use by `grind`, and includes instances for `Int`/`BitVec`/`IntX`/`UIntX`.
- #7803 adds normalization rules for function composition to `grind`.
- #7806 modifies the syntaxes of the `ext`, `intro` and `enter` conv tactics to accept `_`. The introduced binder is an inaccessible name.
- #7808 adds missing forall normalization rules to `grind`.
- #7816 fixes an issue where `x.f.g` wouldn't work but `(x.f).g` would when `x.f` is generalized field notation. The problem was that `x.f.g` would assume `x : T` should be the first explicit argument to `T.f`. Now it uses consistent argument insertion rules. Closes #6400.
- #7825 improves support for `Nat` in the `cutsat` procedure used in `grind`:
  - `cutsat` no longer *pollutes* the local context with facts of the form `-1 * NatCast.natCast x <= 0` for each `x : Nat`. These facts are now stored internally in the `cutsat` state.
  - A single context is now used for all `Nat` terms.
- #7829 fixes an issue in the cutsat counterexamples. It removes the optimization (`Cutsat.State.terms`) that was used to avoid the new theorem `eq_def`. In the two new tests, prior to this PR, `cutsat` produced a bogus counterexample with `b := 2`.
- #7830 modifies the syntax of `induction`, `cases`, and other tactics that use `Lean.Parser.Tactic.inductionAlts`. If a case omits `=> ...` then it is assumed to be `=> ?_`. Example:

```
example (p : Nat × Nat) : p.1 = p.1 := by
  cases p with | _ p1 p2
  /-
  case mk
  p1 p2 : Nat
  ⊢ (p1, p2).fst = (p1, p2).fst
  -/
```

  This works with multiple cases as well. Example:

```
example (n : Nat) : n + 1 = 1 + n := by
  induction n with | zero | succ n ih
  /-
  case zero
  ⊢ 0 + 1 = 1 + 0

  case succ
  n : Nat
  ih : n + 1 = 1 + n
  ⊢ n + 1 + 1 = 1 + (n + 1)
  -/
```

The `induction n with | zero | succ n ih` is short for `induction n with | zero | succ n ih => ?_`, which is short for `induction n with | zero => ?_ | succ n ih => ?_`. Note that a consequence of parsing is that only the last alternative can omit `=>`. Any `=>`-free alternatives before an alternative with `=>` will be a part of that alternative.

- #7831 adds extensibility to the `evalAndSuggest` procedure used to implement `try?`. Users can now implement their own handlers for any tactic. The new test demonstrates how this feature works.
- #7859 allows the LRAT parser to accept any proof that derives the empty clause at somepoint, not necessarily in the last line. Some tools like lrat-trim occasionally include deletions after the derivation of the empty clause but the proof is sound as long as it soundly derives the empty clause somewhere.
- #7861 fixes an issue that prevented theorems from being activated in `grind`.
- #7862 improves the normalization of `Bool` terms in `grind`. Recall that `grind` currently does not case split on Boolean terms to reduce the size of the search space.
- #7864 adds support to `grind` for case splitting on implications of the form `p -> q` and `(h : p) -> q h`. See the new option `(splitImp := true)`.
- #7865 adds a missing propagation rule for implication in `grind`. It also avoids unnecessary case-splits on implications.
- #7870 adds a mixin typeclass for `Lean.Grind.CommRing` recording the characteristic of the ring, and constructs instances for `Int`, `IntX`, `UIntX`, and `BitVec`.
- #7885 fixes the counterexamples produced by the cutsat procedure in `grind` for examples containing `Nat` terms.
- #7892 improves the support for `funext` in `grind`. We will push another PR to minimize the number of case-splits later.
- #7902 introduces a dedicated option for checking whether elaborators are running in the language server.
- #7905 fixes an issue introduced bug #6125 where an `inductive` or `structure` with an autoimplicit parameter with a type that has a metavariable would lead to a panic. Closes #7788.

- #7907 fixes two bugs in `grind`.

    1. Model-based theory combination was creating type incorrect terms.
    2. `Nat.cast` vs `NatCast.natCast` issue during normalization.

- #7914 adds a function hook `PersistentEnvExtension.saveEntriesFn` that can be used to store server-only metadata such as position information and docstrings that should not affect (re)builds.
- #7920 introduces a fast path based on comparing the (cached) hash value to the `DecidableEq` instance of the core expression data type in `bv_decide`'s bitblaster.
- #7926 fixes two issues that were preventing `grind` to solve `getElem?_eq_some_iff`.

    1. Missing propagation rule for `Exists p = False`
    2. Missing conditions at `isCongrToPrevSplit` a filter for discarding unnecessary case-splits.

- #7937 implements a lookahead feature to reduce the size of the search space in `grind`. It is currently effective only for arithmetic atoms.
- #7949 adds the attribute `[grind ext]`. It is used to select which `[ext]` theorems should be used by `grind`. The option `grind +extAll` instructs `grind` to use all `[ext]` theorems available in the environment. After update stage0, we need to add the builtin `[grind ext]` annotations to key theorems such as `funext`.
- #7950 modifies `all_goals` so that in recovery mode it commits changes to the state only for those goals for which the tactic succeeds (while preserving the new message log state). Before, we were trusting that failing tactics left things in a reasonable state, but now we roll back and admit the goal. The changes also fixes a bug where we were rolling back only the metacontext state and not the tactic state, leading to an inconsistent state (a goal list with metavariables not in the metacontext). Closes #7883
- #7952 makes two improvements to the local context when there are autobound implicits in `variables`. First, the local context no longer has two copies of every variable (the local context is rebuilt if the types of autobound implicits have metavariables). Second, these metavariables get names using the same algorithm used by binders that appear in declarations (with `mkForallFVars'` instead of `mkForallFVars`).
- #7957 ensures that mkAppM can be used to construct terms that are only type-correct at at default transparency, even if we are in `withReducible` (e.g. in simp), so that simp does not stumble over simplifying `let` expression with simplifiable type.reliable.
- #7961 fixes a bug in bv_decide where if it was presented with a match on an enum with as many arms as constructors but the last arm being a default match it would (wrongly) give up on the match.
- #7975 reduces the priority of the parent projections of `Lean.Grind.CommRing`, to avoid these being used in typeclass inference in Mathlib.
- #7976 ensure that `bv_decide` can handle the simp normal form of a shift.

- #7978 adds a repro for a non-determinism problem in `grind`.
- #7980 adds a simple type for representing monomials in a `CommRing`. This is going to be used in `grind`.
- #7986 implements reverse lexicographical and graded reverse lexicographical orders for `CommRing` monomials.
- #7989 adds functions and theorems for `CommRing` multivariate polynomials.
- #7992 add a function for converting `CommRing` expressions into multivariate polynomials.
- #7997 removes all type annotations (optional paramters, auto parameters, out params, semi-out params, not just optional parameters as before) from the type of functional induction principles.
- #8011 adds `IsCharP` support to the multivariate-polynomial library in `CommRing`.
- #8012 adds the option `debug.terminalTacticsAsSorry`. When enabled, terminal tactics such as `grind` and `omega` are replaced with `sorry`. Useful for debugging and fixing bootstrapping issues.
- #8014 makes `RArray` universe polymorphic.
- #8016 fixes several issues in the `CommRing` multivariate polynomial library:

  1. Replaces the previous array type with the universe polymorphic `RArray`.
  2. Properly eliminates cancelled monomials.
  3. Sorts monomials in decreasing order.
  4. Marks the parameter p of the `IsCharP` class as an output parameter.
  5. Adds `LawfulBEq` instances for the types `Power`, `Mon`, and `Poly`.

- #8025 simplifies the `CommRing` monomials, and adds

  1. Monomial `lcm`
  2. Monomial division
  3. S-polynomials

- #8029 implements basic support for `CommRing` in `grind`. Terms are already being reified and normalized. We still need to process the equations, but `grind` can already prove simple examples such as:

  ```
  open Lean.Grind in
  example [CommRing α] (x : α) : (x + 1)*(x - 1) = x^2 - 1 := by
    grind +ring
  ```

- #8032 adds support to `grind` for detecting unsatisfiable commutative ring equations when the ring characteristic is known. Examples:

  ```
  example (x : Int) : (x + 1)*(x - 1) = x^2 → False := by
    grind +ring
  ```

- #8033 adds functions for converting `CommRing` reified terms back into Lean expressions.
- #8036 fixes a linearity issue in `bv_decide`'s bitblaster, caused by the fact that the higher order combinators `AIG.RefVec.zip` and `AIG.RefVec.fold` were not being properly specialised.
- #8042 makes `IntCast` a field of `Lean.Grind.CommRing`, along with additional axioms relating it to negation of `OfNat`. This allows use to use existing instances which are not definitionally equal to the previously given construction.
- #8043 adds `NullCert` type for representing Nullstellensatz certificates that will be produced by the new commutative ring procedure in `grind`.
- #8050 fixes missing trace messages when produced inside `realizeConst`
- #8055 adds an implementation of an async IO multiplexing framework as well as an implementation of it for the `Timer` API in order to demonstrate it.
- #8064 adds a failing `grind` test, showing a bug where grind is trying to assign a metavariable incorrectly.
- #8065 adds a (failing) test case for an obstacle I've been running into setting up `grind` for `HashMap`.
- #8068 ensures that for modules opted into the experimental module system, we do not import module docstrings or declaration ranges.
- #8076 fixes `simp?!`, `simp_all?!` and `dsimp?!` to do auto-unfolding.
- #8077 adds simprocs to simplify appends of non-overlapping Bitvector adds. We add a simproc instead of just a `simp` lemma to ensure that we correctly rewrite bitvector appends. Since bitvector appends lead to computation at the bitvector width level, it seems to be more stable to write a simproc.
- #8083 fixes #8081.
- #8086 makes sure that the functional induction priciples for mutually recursive structural functions with extra parameters are split deeply, as expected.
- #8088 adds the “unfolding” variant of the functional induction and functional cases principles, under the name `foo.induct_unfolding` resp. `foo.fun_cases_unfolding`. These theorems combine induction over the structure of a recursive function with the unfolding of that function, and should be more reliable, easier to use and more efficient than just case-splitting and then rewriting with equational theorems.
- #8090 adjusts the experimental module system to elide theorem bodies (i.e. proofs) from being imported into other modules.
- #8094 fixes the generation of functional induction principles for functions with nested nested well-founded recursion and late fixed parameters. This is a follow-up for #7166. Fixes #8093.
- #8096 lets `induction` accept eliminator where the motive application in the conclusion has complex arguments; these are abstracted over using `kabstract` if possible. This feature will go well with unfolding induction principles (#8088).

- #8097 adds support for inductive and coinductive predicates defined using lattice theoretic structures on `Prop`. These are syntactically defined using `greatest_fixpoint` or `least_fixpoint` termination clauses for recursive `Prop`-valued functions. The functionality relies on `partial_fixpoint` machinery and requires function definitions to be monotone. For non-mutually recursive predicates, an appropriate (co)induction proof principle (given by Park induction) is generated.
- #8101 fixes a parallelism regression where linters that e.g. check for errors in the command would no longer find such messages.
- #8102 allows ASCII `<-` in `if let` clauses, for consistency with bind, where both are allowed. Fixes #8098.
- #8111 adds the helper type class `NoZeroNatDivisors` for the commutative ring procedure in `grind`. Core only implements it for `Int`. It can be instantiated in Mathlib for any type `A` that implements `NoZeroSMulDivisors Nat A`. See `findSimp?` and `PolyDerivation` for details on how this instance impacts the commutative ring procedure.
- #8122 implements the generation of compact proof terms for Nullstellensatz certificates in the new commutative ring procedure in `grind`. Some examples:

```
example [CommRing α] (x y : α) : x = 1 → y = 2 → 2*x + y = 4 :=
by
  grind +ring
```

- #8126 implements the main loop of the new commutative ring procedure in `grind`. In the main loop, for each polynomial p in the todo queue, the procedure:
  - Simplifies it using the current basis.
  - Computes critical pairs with polynomials already in the basis and adds them to the queue.
- #8128 implements equality propagation in the new commutative ring procedure in `grind`. The idea is to propagate implied equalities back to the `grind` core module that does congruence closure. In the following example, the equalities: `x^2*y = 1` and `x*y^2 - y = 0` imply that `y*x` is equal to `y*x*y`, which implies by congruence that `f (y*x) = f (y*x*y)`.

```
example [CommRing α] (x y : α) (f : α → Nat) : x^2*y = 1 → x*y^2
- y = 0 → f (y*x) = f (y*x*y) := by
  grind +ring
```

- #8129 updates the If-Normalization example, to separately give an implementation and subsequently prove the spec (using fun_induction), instead of previously building a term in the subtype directly. At the same time, adds a (failing) `grind` test case illustrating a problem with unused match witnesses.
- #8131 adds a configuration option that controls the maximum number of steps the commutative-ring procedure in `grind` performs.

- #8133 fixes the monomial order used by the commutative ring procedure in `grind`. The following new test now terminates quickly.

```
example [CommRing α] (a b c : α)
  : a + b + c = 3 →
    a^2 + b^2 + c^2 = 5 →
    a^3 + b^3 + c^3 = 7 →
    a^4 + b^4 + c^4 = 9 := by
  grind +ring
```

- #8134 ensures that `set_option grind.debug true` works properly when using `grind +ring`. It also adds the helper functions `mkPropEq` and `mkExpectedPropHint`.
- #8137 improves equality propagation (also known as theory combination) and polynomial simplification for rings that do not implement the `NoZeroNatDivisors` class. With these fixes, `grind` can now solve:

```
example [CommRing α] (a b c : α) (f : α → Nat)
  : a + b + c = 3 →
    a^2 + b^2 + c^2 = 5 →
    a^3 + b^3 + c^3 = 7 →
    f (a^4 + b^4) + f (9 - c^4) ≠ 1 := by
  grind +ring
```

  This example uses the commutative ring procedure, the linear integer arithmetic solver, and congruence closure. For rings that implement `NoZeroNatDivisors`, a polynomial is now also divided by the greatest common divisor (gcd) of its coefficients when it is inserted into the basis.
- #8157 fixes an incompatibility of `replayConst` as used by e.g. `aesop` with `native_decide`-using tactics such as `bv_decide`
- #8158 fixes the `grind +splitImp` and the arrow propagator. Given `p : Prop`, the propagator was incorrectly assuming `A` was always a proposition in an arrow `A -> p`. also adds a missing normalization rule to `grind`.
- #8159 adds support for the following import variants to the experimental module system:
  - `private import`: Makes the imported constants available only in non-exported contexts such as proofs. In particular, the import will not be loaded, or required to exist at all, when the current module is imported into other modules.
  - `import all`: Makes non-exported information such as proofs of the imported module available in non-exported contexts in the current module. Main purpose is to allow for reasoning about imported definitions when they would otherwise be opaque. TODO: adjust name resolution so that imported `private` decls are accessible through syntax.
- #8161 changes `Lean.Grind.CommRing` to inline the `NatCast` instance (i.e. to be provided by the user) rather than constructing one from the existing data. Without this change we can't construct instances in Mathlib that `grind` can use.
- #8163 adds some currently failing tests for `grind +ring`, resulting in either kernel type mismatches (bugs) or a kernel deep recursion (perhaps just a too-large problem).

- #8167 improves the heuristics used to compute the basis and simplify polynomials in the commutative procedure used in `grind`.
- #8168 fixes a bug when constructing the proof term for a Nullstellensatz certificate produced by the new commutative ring procedure in `grind`. The kernel was rejecting the proof term.
- #8170 adds the infrastructure for creating stepwise proof terms in the commutative procedure used in `grind`.
- #8189 implements **stepwise proof terms** in the commutative ring procedure used by `grind`. These terms serve as an alternative representation to the traditional Nullstellensatz certificates, aiming to address the **exponential worst-case complexity** often associated with certificate construction.
- #8231 changes the behaviour of `apply?` so that the `sorry` it uses to close the goal is non-synthetic. (Recall that correct use of synthetic sorries requires that the tactic also generates an error message, which we don't want to do in this situation.) Either this PR or #8230 are sufficient to defend against the problem reported in #8212.
- #8254 fixes unintended inlining of `ToJson`, `FromJson`, and `Repr` instances, which was causing exponential compilation times in `deriving` clauses for large structures.

Library

- #6081 adds an `inheritEnv` field to `IO.Process.SpawnArgs`. If `false`, the spawned process does not inherit its parent's environment.
- #7108 proves `List.head_of_mem_head?` and the analogous `List.getLast_of_mem_getLast?`.
- #7400 adds lemmas for the `filter`, `map` and `filterMap` functions of the hash map.
- #7659 adds SMT-LIB operators to detect overflow `BitVec.(umul_overflow, smul_overflow)`, according to the definitions here, and the theorems proving equivalence of such definitions with the `BitVec` library functions (`umulOverflow_eq`, `smulOverflow_eq`). Support theorems for these proofs are `BitVec.toInt_one_of_lt`, `BitVec.toInt_mul_toInt_lt`, `BitVec.le_toInt_mul_toInt`, `BitVec.toNat_mul_toNat_lt`, `BitVec.two_pow_le_toInt_mul_toInt_iff`, `BitVec.toInt_mul_toInt_lt_neg_two_pow_iff` and `Int.neg_mul_le_mul`, `Int.bmod_eq_self_of_le_mul_two`, `Int.mul_le_mul_of_natAbs_le`, `Int.mul_le_mul_of_le_of_le_of_nonneg_of_nonpos`, `Int.pow_lt_pow`. The PR also includes a set of tests.
- #7671 contains the theorem proving that signed division x.toInt / y.toInt only overflows when `x = intMin w` and `y = allOnes w` (for `0 < w`). To show that this is the *only* case in which overflow happens, we refer to overflow for negation (`BitVec.sdivOverflow_eq_negOverflow_of_neg_one`): in fact, `x.toInt/(allOnes w).toInt = - x.toInt`, i.e., the overflow conditions are the

same as `negOverflow` for `x`, and then reason about the signs of the operands with the respective theorems. These BitVec theorems themselves rely on numerous `Int.ediv_*` theorems, that carefully set the bounds of signed division for integers.

- #7761 implements the core theorem for the Bitwuzla rewrites NORM_BV_NOT_OR_SHL and BV_ADD_SHL, which convert the mixed-boolean-arithmetic expression into a purely arithmetic expression:

```
theorem add_shiftLeft_eq_or_shiftLeft {x y : BitVec w} :
    x + (y <<< x) =  x ||| (y <<< x)
```

- #7770 adds a shared mutex (or read-write lock) as `Std.SharedMutex`.
- #7774 adds `Option.pfilter`, a variant of `Option.filter` and several lemmas for it and other `Option` functions. These lemmas are split off from #7400.
- #7791 adds lemmas about `Nat.lcm`.
- #7802 adds `Int.gcd` and `Int.lcm` variants of all `Nat.gcd` and `Nat.lcm` lemmas.
- #7818 deprecates `Option.merge` and `Option.liftOrGet` in favor of `Option.zipWith`.
- #7819 extends `Std.Channel` to provide a full sync and async API, as well as unbounded, zero sized and bounded channels.
- #7835 adds `BitVec.[toInt_append|toFin_append]`.
- #7847 removes `@[simp]` from all deprecated theorems. `simp` will still use such lemmas, without any warning message.
- #7851 partially reverts #7818, because the function called `Option.zipWith` in that PR does not actually correspond to `List.zipWith`. We choose `Option.merge` as the name instead.
- #7855 moves `ReflBEq` to `Init.Core` and changes `LawfulBEq` to extend `ReflBEq`.
- #7856 changes definitions and theorems not to use the membership instance on `Option` unless the theorem is specifically about the membership instance.
- #7869 fixes a regression introduced in #7445 where the new `Array.emptyWithCapacity` was accidentally not tagged with the correct function to actually allocate the capacity.
- #7871 generalizes the typeclass assumptions on monadic `Option` functions.
- #7879 adds `Int.toNat_emod`, analogous to `Int.toNat_add/mul`.
- #7880 adds the functions `UIntX.ofInt`, and basic lemmas.
- #7886 adds `UIntX.pow` and `Pow UIntX Nat` instances, and similarly for signed fixed-width integers. These are currently only the naive implementation, and will need to be subsequently replaced via `@[extern]` with fast implementations (tracked at #7887).
- #7888 adds `Fin.ofNat'_mul` and `Fin.mul_ofNat'`, parallel to the existing lemmas about add.

- #7889 adds `Int.toNat_sub''` a variant of `Int.toNat_sub` taking inequality hypotheses, rather than expecting the arguments to be casts of natural numbers. This is parallel to the existing `toNat_add` and `toNat_mul`.
- #7890 adds missing lemmas about `Int.bmod`, parallel to lemmas about the other `mod` variants.
- #7891 adds the rfl simp lemma `Int.cast x = x` for `x : Int`.
- #7893 adds `BitVec.pow` and `Pow (BitVec w) Nat`. The implementation is the naive one, and should later be replaced by an `@[extern]`. This is tracked at https://github.com/leanprover/lean4/issues/7887.
- #7897 cleans up the `Option` development, upstreaming some results from mathlib in the process.
- #7899 shuffles some results about integers around to make sure that all material that currently exists about `Int.bmod` is located in `DivMod/Lemmas.lean` and not downstream of that.
- #7901 adds `instance [Pure f] : Inhabited (OptionT f α)`, so that `Inhabited (OptionT Id Empty)` synthesizes.
- #7912 adds `List.Perm.take/drop`, and `Array.Perm.extract`, restricting permutations to sublist / subarrays when they are constant elsewhere.
- #7913 adds some missing `List/Array/Vector` lemmas about `isSome_idxOf?`, `isSome_finIdxOf?`, `isSome_findFinIdx?`, isSome_findIdx?and the `correspondingisNone` versions.
- #7933 adds lemmas about `Int.bmod` to achieve parity between `Int.bmod` and `Int.emod`/`Int.fmod`/`Int.tmod`. Furthermore, it adds missing lemmas for emod/fmod/tmod and performs cleanup on names and statements for all four operations, also with a view towards increasing consistency with the corresponding `Nat.mod` lemmas.
- #7938 adds lemmas about `List/Array/Vector.countP/count` interacting with `replace`. (Specializing to `_self` and _ne lemmas doesn't seem useful, as there will still be an `if` on the RHS.)
- #7939 adds `Array.count_erase` and specializations.
- #7953 generalizes some typeclass hypotheses in the `List.Perm` API (away from `DecidableEq`), and reproduces `List.Perm.mem_iff` for `Array`, and fixes a mistake in the statement of `Array.Perm.extract`.
- #7971 upstreams much of the material from `Mathlib/Data/Nat/Init.lean` and `Mathlib/Data/Nat/Basic.lean`.
- #7983 upstreams many of the results from `Mathlib/Data/Int/Init.lean`.
- #7994 reproduces the `Array.Perm` API for `Vector`. Both are still significantly less developed than the API for `List.Perm`.
- #7999 replaces `Array.Perm` and `Vector.Perm` with one-field structures. This avoids dot notation for `List` to work like e.g. `h.cons 3` where h is an `Array.Perm`.
- #8000 deprecates some `Int.ofNat_*` lemmas in favor of `Int.natCast_*`.

- #8004 adds extensional hash maps and hash sets under the names `Std.ExtDHashMap`, `Std.ExtHashMap` and `Std.ExtHashSet`. Extensional hash maps work like regular hash maps, except that they have extensionality lemmas which make them easier to use in proofs. This however makes it also impossible to regularly iterate over its entries.
- #8030 adds some missing lemmas about `List/Array/Vector.findIdx?/findFinIdx?/findSome?/idxOf?`.
- #8044 introduces the modules `Std.Data.DTreeMap.Raw`, `Std.Data.TreeMap.Raw` and `Std.Data.TreeSet.Raw` and imports them into `Std.Data`. All modules related to the raw tree maps are imported into these new modules so that they are now a transitive dependency of `Std`.
- #8067 fixes the behavior of `Substring.isNat` to disallow empty strings.
- #8078 is a follow up to #8055 and implements a `Selector` for async TCP in order to allow IO multiplexing using TCP sockets.
- #8080 fixes `Json.parse` to handle surrogate pairs correctly.
- #8085 moves the coercion `α → Option α` to the new file `Init.Data.Option.Coe`. This file may not be imported anywhere in `Init` or `Std`.
- #8089 adds optimized division functions for `Int` and `Nat` when the arguments are known to be divisible (such as when normalizing rationals). These are backed by the gmp functions `mpz_divexact` and `mpz_divexact_ui`. See also leanprover-community/batteries#1202.
- #8136 adds an initial set of `@[grind]` annotations for `List/Array/Vector`, enough to set up some regression tests using `grind` in proofs about `List`. More annotations to follow.
- #8139 is a follow up to #8055 and implements a `Selector` for async UDP in order to allow IO multiplexing using UDP sockets.
- #8144 changes the predicate for `Option.guard` to be `p : α → Bool` instead of `p : α → Prop`. This brings it in line with other comparable functions like `Option.filter`.
- #8147 adds `List.findRev?` and `List.findSomeRev?`, for parity with the existing Array API, and simp lemmas converting these into existing operations.
- #8148 generalises `List.eraseDups` to allow for an arbitrary comparison relation. Further, it proves `eraseDups_append : (as ++ bs).eraseDups = as.eraseDups ++ (bs.removeAll as).eraseDups`.
- #8150 is a follow up to #8055 and implements a Selector for `Std.Channel` in order to allow multiplexing using channels.
- #8154 adds unconditional lemmas for `HashMap.getElem?_insertMany_list`, alongside the existing ones that have quite strong preconditions. Also for TreeMap (and dependent/extensional variants).
- #8175 adds simp/grind lemmas about `List/Array/Vector.contains`. In the presence of `LawfulBEq` these effectively already held, via simplifying `contains` to `mem`, but now these also fire without `LawfulBEq`.
- #8184 adds the `insertMany_append` lemma for all map variants.

Compiler

- #6063 updates the version of LLVM and clang used by and shipped with Lean to 19.1.2
- #7824 fixes an issue where uses of 'noncomputable' definitions can get incorrectly compiled, while also removing the use of 'noncomputable' definitions altogether. Some uses of 'noncomputable' definitions (e.g. Classical.propDecidable) do not get compiled correctly by type erasure. Running the optimizer on the result can lead to them being optimized away, eluding the later IR-level check for uses of noncomputable definitions.
- #7838 adds support for mpz objects (i.e., big nums) to the `shareCommon` functions.
- #7854 introduces fundamental API to distribute module data across multiple files in preparation for the module system.
- #7945 fixes a potential race between `IO.getTaskState` and the task in question finishing, resulting in undefined behavior.
- #7958 ensures that after `main` is finished we still wait on dedicated tasks instead of exiting forcefully. If users wish to violently kill their dedicated tasks at the end of main instead they can run `IO.Process.exit` at the end of `main` instead.
- #7990 adopts lcAny in more cases of type erasure in the new code generator.
- #7996 disables CSE of local function declarations in the base phase of the new compiler. This was introducing sharing between lambdas to bind calls w/ do notation, which caused them to later no longer be inlined.
- #8006 changes the inlining heuristics of the new code generator to match the old one, which ensures that monadic folds get sufficiently inlined for their tail recursion to be exposed to the code generator.
- #8007 changes eager lambda lifting heuristics in the new compiler to match the old compiler, which ensures that inlining/specializing monadic code does not accidentally create mutual tail recursion that the code generator can't handle.
- #8008 changes specialization in the new code generator to consider callee params to be ground variables, which improves the specialization of polymorphic functions.
- #8009 restricts lifting outside of cases expressions on values of a Decidable type, since we can't correctly represent the dependency on the erased proposition in the later stages of the compiler.
- #8010 fixes caseOn expressions with an implemented_by to work correctly with hash consing, even when the elaborator produces terms that reconstruct the discriminant rather than just reusing a variable.
- #8015 fixes the IR elim_dead_branches pass to correctly handle join points with no params, which currently get considered unreachable. I was not able to find an easy repro of this with the old compiler, but it occurs when bootstrapping Lean with the new compiler.
- #8017 makes the IR elim_dead_branches pass correctly handle extern functions by considering them as having a top return value. This fix is required to bootstrap the Init/ directory with the new compiler.

- #8023 fixes the IR expand_reset_reuse pass to correctly handle duplicate projections from the same base/index. This does not occur (at least easily) with the old compiler, but it occurs when bootstrapping Lean with the new compiler.
- #8124 correctly handles escaping functions in the LCNF elimDeadBranches pass, by setting all params to top instead of potentially leaving them at their default bottom value.
- #8125 adds support for the `init` attribute to the new compiler.
- #8127 adds support for borrowed params in the new compiler, which requires adding support for .mdata expressions to LCNF type handling.
- #8132 adds support for lowering `casesOn` for builtin types in the new compiler.
- #8156 fixes a bug where the old compiler's lcnf conversion expr cache was not including all of the relevant information in the key, leading to terms inadvertently being erased. The `root` variable is used to determine whether lambda arguments to applications should get let bindings or not, which in turn affects later decisions about type erasure (erase_irrelevant assumes that any non-atomic argument is irrelevant).
- #8236 fixes an issue where the combination of `extern_lib` and `precompileModules` would lead to "symbol not found" errors.

Pretty Printing

- #7805 modifies the pretty printing of raw natural number literals; now both `pp.explicit` and `pp.natLit` enable the `nat_lit` prefix. An effect of this is that the hover on such a literal in the Infoview has the `nat_lit` prefix.
- #7812 modifies the pretty printing of pi types. Now ∀ will be preferred over → for propositions if the domain is not a proposition. For example, `∀ (n : Nat), True` pretty prints as `∀ (n : Nat), True` rather than as `Nat → True`. There is also now an option `pp.foralls` (default true) that when false disables using ∀ at all, for pedagogical purposes. also adjusts instance implicit binder pretty printing — nondependent pi types won't show the instance binder name. Closes #1834.
- #7813 fixes an issue where `let n : Nat := sorry` in the Infoview pretty prints as `n : ℕ := sorry` `` `«Foo:17:17» ``. This was caused by top-level expressions being pretty printed with the same rules as Infoview hovers. Closes #6715. Refactors `Lean.Widget.ppExprTagged`; now it takes a delaborator, and downstream users should configure their own pretty printer option overrides if necessary if they used the `explicit` argument (see `Lean.Widget.makePopup.ppExprForPopup` for an example). Breaking change: `ppExprTagged` does not set `pp.proofs` on the root expression.
- #7840 causes structure instance notation to be tagged with the constructor when `pp.tagAppFns` is true. This will make docgen will have { and } be links to the structure constructor.
- #8022 fixes a bug where pretty printing is done in a context with cleared local instances. These were cleared since the local context is updated during a name sanitization step, but preserving local instances is valid since the modification to the local context only affects user names.

Documentation

- #7947 adds some docstrings to clarify the functions of `Lean.mkFreshId`, `Lean.Core.mkFreshUserName`, `Lean.Elab.Term.mkFreshBinderName`, and `Lean.Meta.mkFreshBinderNameForTactic`.
- #8018 adjusts the RArray docstring to the new reality from #8014.

Server

- #7610 adjusts the `TryThis` widget to also work in widget messages rather than only as a panel widget. It also adds additional documentation explaining why this change was needed.
- #7873 fixes a number of bugs related to the handling of the source search path in the language server, where deleting files could cause several features to stop functioning and both untitled files and files that don't exist on disc could have conflicting module names.
- #7882 fixes a regression where elaboration of a previous document version is not cancelled on changes to the document.
- #8242 fixes the 'goals accomplished' diagnostics. They were accidentally broken in #7902.

Lake

- #7796 moves Lean's shared library path before the workspace's in Lake's augmented environment (e.g., `lake env`).
- #7809 fixes the order of libraries when loading them via `--load-dynlib` or `--plugin` in `lean` and when linking them into a shared library or executable. A `Dynlib` now tracks its dependencies and they are topologically sorted before being passed to either linking or loading.
- #7822 changes Lake to use normalized absolute paths for its various files and directories.
- #7860 restores the use of builtins (e.g., initializer, elaborators, and macros) for DSL features and the use of the Lake plugin in the server.
- #7906 changes Lake build traces to track their mixed inputs. The tracked inputs are saved as part of the `.trace` file, which can significantly assist in debugging trace issues. In addition, this PR tweaks some existing Lake traces. Most significant, module olean traces no longer incorporate their module's source trace.
- #7909 adds Lake support for building modules given their source file path. This is made use of in both the CLI and the sever.
- #7963 adds helper functions to convert between `Lake.EStateT` and `EStateM`.
- #7967 adds a `bootstrap` option to Lake which is used to identify the core Lean package. This enables Lake to use the current stage's include directory rather than the Lean toolchains when compiling Lean with Lean in core.
- #7987 fixes a bug in #7967 that broke external library linking.
- #8026 fixes bugs in #7809 and #7909 that were not caught partially because the `badImport` test had been disabled.

- #8048 moves the Lake DSL syntax into a dedicated module with minimal imports.
- #8152 fixes a regression where non-precompiled module builds would `--load-dynlib` package `extern_lib` targets.
- #8183 makes Lake tests much more verbose in output. It also fixes some bugs that had been missed due to disabled tests. Most significantly, the target specifier `@pkg` (e.g., in `lake build`) is now always interpreted as a package. It was previously ambiguously interpreted due to changes in #7909.
- #8190 adds documentation for native library options (e.g., `dynlibs`, `plugins`, `moreLinkObjs`, `moreLinkLibs`) and `needs` to the Lake README. It is also includes information about specifying targets on the Lake CLI and in Lean and TOML configuration files.

Other

- #7785 adds further automation to the release process, taking care of tagging, and creating new `bump/v4.X.0` branches automatically, and fixing some bugs.
- #7789 fixes `lean` potentially changing or interpreting arguments after `--run`.
- #8060 fixes a bug in the Lean kernel. During reduction of `Nat.pow`, the kernel did not validate that the WHNF of the first argument is a `Nat` literal before interpreting it as an `mpz` number. adds the missing check.

Lean 4.19.0 (2025-05-01)

For this release, 420 changes landed. In addition to the 164 feature additions and 78 fixes listed below there were 13 refactoring changes, 29 documentation improvements, 31 performance improvements, 9 improvements to the test suite and 94 other changes.

Highlights

Lean v4.19.0 introduces a number of features, bug fixes, performance gains, library developments, along with quality-of-life improvements across documentation, the language server, and Lake.

New Decorations in VS Code

Visual feedback in VS Code has been improved, with the extension now featuring:

- Gutter decorations for errors and warnings. These make the full range of errors/warnings clear, which is especially useful when the corresponding squigglies are small.
- End-of-line markers for 'unsolved goals'. These are displayed at the line where 'unsolved goals' error ends and indicate where the proof needs to be continued.
- 'Goals accomplished!' message. When a theorem or a `Prop`-typed `example` contains no errors or `sorry`s anymore, two blue checkmarks appear next to the start of the declaration as a gutter decoration. Additionally, a 'Goals accomplished!' message appears under 'Messages' in the InfoView.

Gutter decorations for errors and warnings are available for all Lean 4 versions. Decorations for 'unsolved goals' and 'goals accomplished' rely on server-side support, which is added in this version via #7366.

All of these features can be disabled, and 'Goals accomplished!' icon can be configured in VS Code extension settings. See leanprover/vscode-lean4#585 for the details.

Parallel Elaboration

- #7084 enables the elaboration of theorem bodies, i.e. proofs, to happen in parallel to each other as well as to other elaboration tasks.

Language Features

- #5182 makes functions defined by well-founded recursion use an `opaque` well-founded proof by default. This reliably prevents kernel reduction of such definitions and proofs, which tends to be prohibitively slow (fixes #2171), and which regularly causes hard-to-debug kernel type-checking failures. This change renders `unseal` ineffective for such definitions. To avoid the opaque proof, annotate the function definition with `@[semireducible]`.
- #7166 extends the notion of “fixed parameter” of a recursive function also to parameters that come after varying function. The main benefit is that we get nicer induction principles.

  Before, the definition

  ```
  def app (as : List α) (bs : List α) : List α :=
    match as with
    | [] => bs
    | a::as => a :: app as bs
  ```

  produced

  ```
  app.induct.{u_1} {α : Type u_1} (motive : List α → List α → Prop)
  (case1 : ∀ (bs : List α), motive [] bs)
    (case2 : ∀ (bs : List α) (a : α) (as : List α), motive as bs →
  motive (a :: as) bs) (as bs : List α) : motive as bs
  ```

  and now you get

  ```
  app.induct.{u_1} {α : Type u_1} (motive : List α → Prop) (case1 :
  motive [])
    (case2 : ∀ (a : α) (as : List α), motive as → motive (a :: as))
  (as : List α) : motive as
  ```

  because `bs` is fixed throughout the recursion (and can completely be dropped from the principle).

  This is a **breaking change** when such an induction principle is used explicitly. Using `fun_induction` makes proof tactics robust against this change.

  See the PR description for the rules for when a parameter is considered fixed.

Note that in a definition like

```
def app : List α → List α → List α
  | [], bs => bs
  | a::as, bs => a :: app as bs
```

the `bs` is not considered fixed, as it goes through the matcher machinery.

- #7431 changes the syntax of location modifiers for tactics like `simp` and `rw` (e.g., `simp at h ⊢`) to allow the turnstile ⊢ to appear anywhere in the sequence of locations.
- #7457 ensures info tree users such as linters and request handlers have access to info subtrees created by async elab task by introducing API to leave holes filled by such tasks.

  Breaking change: other metaprogramming users of `Command.State.infoState` may need to call `InfoState.substituteLazy` on it manually to fill all holes.

Updates to structures and classes

- #7302 changes how fields are elaborated in the `structure`/`class` commands and also makes default values respect the structure resolution order when there is diamond inheritance. Before, the details of subobjects were exposed during elaboration, and in the local context any fields that came from a subobject were defined to be projections of the subobject field. Now, every field is represented as a local variable. All parents (not just subobject parents) are now represented in the local context, and they are now local variables defined to be parent constructors applied to field variables (inverting the previous relationship). See the PR description for further details.
- #7640 implements the main logic for inheriting and overriding autoParam fields in the `structure`/`class` commands, pending being enabled in the structure instance notation elaborator. Adds term info to overridden fields, so they now can be hovered over, and "go to definition" goes to the structure the field is originally defined in.
- #7717 changes how `{...}`/`where` notation ("structure instance notation") elaborates. The notation now tries to simulate a flat representation as much as possible, without exposing the details of subobjects. This is a **breaking change**, see the PR description for further details and mitigation strategies.
- #7742 adds a feature to `structure`/`class` where binders without types on a field definition are interpreted as overriding the type's parameters binder kinds in that field's projection function. See the PR description for further details.

Library Updates

- Developments in the async machinery;
- Standardization of the integer division API;
- Conversions between finite types;
- API expansion of `BitVec` and tree maps;
- Proofs of Bitwuzla rewrite rules;

- Improvements to `List`/`Array`/`Vector`, as well as `HashMap` and `Int`/`Nat`.

See the Library section below for details.

Other Highlights

- Documentation has been significantly expanded. See the Documentation section below for details.
- #7185 refactors Lake's build internals to enable the introduction of targets and facets beyond packages, modules, and libraries. Facets, build keys, build info, and CLI commands have been generalized to arbitrary target types.

Language

- #5182 makes functions defined by well-founded recursion use an `opaque` well-founded proof by default; see highlights section for details.
- #5998 lets `omega` always abstract its own proofs into an auxiliary definition. The size of the olean of Vector.Extract goes down from 20MB to 5MB with this, overall stdlib olean size and build instruction count go down 5%.
- #6325 ensures that environments can be loaded, repeatedly, without executing arbitrary code
- #7075 ensures that names suggested by tactics like `simp?` are not shadowed by auxiliary declarations in the local context and that names of `let rec` and `where` declarations are correctly resolved in tactic blocks.
- #7166 extends the notion of “fixed parameter” of a recursive function also to parameters that come after varying function; see highlights section for details.
- #7256 introduces the `assert!` variant `debug_assert!` that is activated when compiled with `buildType debug`.
- #7304 fixes an issue where nested `let rec` declarations within `match` expressions or tactic blocks failed to compile if they were nested within, and recursively called, a `let rec` that referenced a variable bound by a containing declaration.
- #7324 changes the internal construction of well-founded recursion, to not change the type of `fix`’s induction hypothesis in non-defeq ways.
- #7333 allows aux decls (like generated by `match`) to be generated by decreasing_by tactics.
- #7335 modifies `elabTerminationByHints` in a way that the type of the recursive function used for elaboration of the termination measure is striped of from optional parameters. It prevents introducing dependencies between the default values for arguments, that can cause the termination checker to fail.
- #7353 changes `abstractNestedProofs` so that it also visits the subterms in the head of an application.
- #7362 allows simp dischargers to add aux decls to the environment. This enables tactics like `native_decide` to be used here, and unblocks improvements to omega in #5998.

- #7387 uses `-implicitDefEqProofs` in `bv_omega` to ensure it is not affected by the change in #7386.
- #7397 ensures that `Poly.mul p 0` always returns `Poly.num 0`.
- #7409 allows the use of `dsimp` during preprocessing of well-founded definitions. This fixes regressions when using `if-then-else` without giving a name to the condition, but where the condition is needed for the termination proof, in cases where that subexpression is reachable only by dsimp, but not by simp (e.g. inside a dependent let)
- #7431 changes the syntax of location modifiers for tactics like `simp` and `rw` (e.g., `simp at h ⊢`) to allow the turnstile ⊢ to appear anywhere in the sequence of locations.
- #7509 disables the `implicitDefEqProofs` simp option in the preprocessor of `bv_decide` in order to account for regressions caused by #7387.
- #7511 fixes two bugs in `simp +arith` that were preventing specific subterms from being normalized.
- #7515 fixes another bug in `simp +arith`. This bug was affecting `grind`. See new test for an example.
- #7551 changes `isNatCmp` to ignore optional arguments annotations, when checking for <-like comparison between elements of `Nat`. That previously caused `guessLex` to fail when checking termination of a function, whose signature involved an optional argument of the type `Nat`.
- #7560 ensures that we use the same ordering to normalize linear `Int` terms and relations. This change affects `simp +arith` and `grind` normalizer.
- #7622 fixes `fun_induction` when used on structurally recursive functions where there are targets occurring before fixed parameters.
- #7630 fixes a performance issue in the `whnfCore` procedure.
- #7728 fixes an issue in `abstractNestedProofs`. We should abstract proofs occurring in the inferred proposition too.

Structures

- #7302 changes how fields are elaborated in the `structure`/`class` commands and also makes default values respect the structure resolution order when there is diamond inheritance. Before, the details of subobjects were exposed during elaboration, and in the local context any fields that came from a subobject were defined to be projections of the subobject field. Now, every field is represented as a local variable. All parents (not just subobject parents) are now represented in the local context, and they are now local variables defined to be parent constructors applied to field variables (inverting the previous relationship). Other notes:
  - The entire collection of parents is processed, and all parent projection names are checked for consistency. Every parent appears in the local context now.
  - For classes, every parent now contributes an instance, not just the parents represented as subobjects.

- Default values are now processed according to the parent resolution order. Default value definition/override auxiliary definitions are stored at `StructName.fieldName._default`, and inherited values are stored at `StructName.fieldName._inherited_default`. Metaprograms no longer need to look at parents when doing calculations on default values.
- Default value omission for structure instance notation pretty printing has been updated in consideration of this.
- Now the elaborator generates a `_flat_ctor` constructor that will be used for structure instance elaboration. All types in this constructor are put in "field normal form" (projections of parent constructors are reduced, and parent constructors are eta reduced), and all fields with autoParams are annotated as such. This is not meant for users, but it may be useful for metaprogramming.
- While elaborating fields, any metavariables whose type is one of the parents is assigned to that parent. The hypothesis is that, for the purpose of elaborating structure fields, parents are fixed: there is only *one* instance of any given parent under consideration. See the `Magma` test for an example of this being necessary. The hypothesis may not be true when there are recursive structures, since different values of the structure might not agree on parent fields.

- #7314 changes elaboration of `structure` parents so that each must be fully elaborated before the next one is processed.
- #7640 implements the main logic for inheriting and overriding autoParam fields in the `structure`/`class` commands, pending being enabled in the structure instance notation elaborator. Adds term info to overridden fields, so they now can be hovered over, and "go to definition" goes to the structure the field is originally defined in.
- #7652 gives `#print` for structures the ability to show the default values and auto-param tactics for fields.
- #7717 changes how `{...}`/`where` notation ("structure instance notation") elaborates. The notation now tries to simulate a flat representation as much as possible, without exposing the details of subobjects. Features:
  - When fields are elaborated, their expected types now have a couple reductions applied. For all projections and constructors associated to the structure and its parents, projections of constructors are reduced and constructors of projections are eta reduced, and also implementation detail local variables are zeta reduced in propositions (so tactic proofs should never see them anymore). Furthermore, field values are beta reduced automatically in successive field types. The example in mathlib4#12129 now shows a goal of `0 = 0` rather than `{ toFun := fun x => x }.toFun 0 = 0`.
  - All parents can now be used as field names, not just the subobject parents. These are like additional sources but with three constraints: every field of the value must be used, the fields must not overlap with other provided fields, and every field of the specified parent

must be provided for. Similar to sources, the values are hoisted to `let`s if they are not already variables, to avoid multiple evaluation. They are implementation detail local variables, so they get unfolded for successive fields.

- All class parents are now used to fill in missing fields, not just the subobject parents. Closes #6046. Rules: (1) only those parents whose fields are a subset of the remaining fields are considered, (2) parents are considered only before any fields are elaborated, and (3) only those parents whose type can be computed are considered (this can happen if a parent depends on another parent, which is possible since #7302).
- Default values and autoparams now respect the resolution order completely: each field has at most one default value definition that can provide for it. The algorithm that tries to unstick default values by walking up the subobject hierarchy has been removed. If there are applications of default value priorities, we might consider it in a future release.
- The resulting constructors are now fully packed. This is implemented by doing structure eta reduction of the elaborated expressions.
- "Magic field definitions" (as reported on Zulip) have been eliminated. This was where fields were being solved for by unification, tricking the default value system into thinking they had actually been provided. Now the default value system keeps track of which fields it has actually solved for, and which fields the user did not provide. Explicit structure fields (the default kind) without any explicit value definition will result in an error. If it was solved for by unification, the error message will include the inferred value, like "field 'f' must be explicitly provided, its synthesized value is v"
- When the notation is used in patterns, it now no longer inserts fields using class parents, and it no longer applies autoparams or default values. The motivation is that one expects patterns to match only the given fields. This is still imperfect, since fields might be solved for indirectly.
- Elaboration now attempts error recovery. Extraneous fields log errors and are ignored, missing fields are filled with `sorry`.

- #7742 adds a feature to `structure`/`class` where binders without types on a field definition are interpreted as overriding the type's parameters binder kinds in that field's projection function. The rules are (1) only a prefix of the binders are interpreted this way, (2) multi-identifier binders are allowed but they must all be for parameters, (3) only parameters that appear in the declaration itself (not from `variables`) can be overridden and (4) the updates will be applied after parameter binder kind inference is done. Binder updates are not allowed in default value redefinitions. Example application: In the following, `(R p)` causes the R and p parameters to be explicit, where normally they would be implicit.

```
class CharP (R : Type u) [AddMonoidWithOne R] (p : Nat) : Prop
where
  cast_eq_zero_iff (R p) : ∀ x : Nat, (x : R) = 0 ↔ p ∣ x

#guard_msgs in #check CharP.cast_eq_zero_iff
/-
info: CharP.cast_eq_zero_iff.{u} (R : Type u) {inst✝ :
AddMonoidWithOne R} (p : Nat) [self : CharP R p] (x : Nat) :
```

```
  ↑x = 0 ↔ p ∣ x
-/
```

- #7746 adds declaration ranges to structure fields that were copied from parents that aren't represented as subobjects, supporting "go to definition". The declaration range is the parent in the `extends` clause.

Parallel Elaboration

- #7084 enables the elaboration of theorem bodies, i.e. proofs, to happen in parallel to each other as well as to other elaboration tasks.
- #7247 makes generation of `match` equations and splitters compatible with parallelism.
- #7261 ensures all equation, unfold, induction, and partial fixpoint theorem generators in core are compatible with parallelism.
- #7348 ensures all equation and unfold theorem generators in core are compatible with parallelism.
- #7457 ensures info tree users such as linters and request handlers have access to info subtrees created by async elab task by introducing API to leave holes filled by such tasks.
- #8101 fixes a parallelism regression where linters that e.g. check for errors in the command would no longer find such messages.

bv_decide

- #7298 adds rewrites to bv_decide's preprocessing that concern combinations of if-then-else and operation such as multiplication or negation.
- #7309 fixes a bug where bv_decide's new structure support would sometimes not case split on all available structure fvars as their type was an mvar.
- #7329 adds support to bv_decide for simple pattern matching on enum inductives. By simple we mean non dependent match statements with all arms written out.
- #7347 upgrades the CaDiCal we ship and use for bv_decide to version 2.1.2. Additionally it enables binary LRAT proofs on windows by default as https://github.com/arminbiere/cadical/issues/112 has been fixed.
- #7381 refactors the AIG datastructures that underly bv_decide in order to allow a better tracking of negations in the circuit. This refactor has two effects, for one adding full constant folding to the AIG framework and secondly enabling us to add further simplifications from the Brummayer Biere paper in the future which was previously architecturally impossible.
- #7390 makes bv_decide's preprocessing handle casts, as we are in the constant BitVec fragment we should be able to always remove them using BitVec.cast_eq.
- #7407 adds rules for `-1#w * a = -a` and `a * -1#w = -a` to bv_normalize as seen in Bitwuzla's BV_MUL_SPECIAL_CONST.
- #7417 adds support for enum inductive matches with default branches to bv_decide.
- #7429 adds the BV_EXTRACT_FULL preprocessing rule from Bitwuzla to bv_decide.

- #7436 adds simprocs that turn left and right shifts by constants into extracts to bv_decide.
- #7438 adds the EQUAL_CONST_BV_ADD and BV_AND_CONST rules to bv_decide's preprocessor.
- #7441 adds the BV_CONCAT_CONST, BV_CONCAT_EXTRACT and ELIM_ZERO_EXTEND rule from Bitwuzla to bv_decide.
- #7477 ensures that bv_decide doesn't accidentally operate on terms underneath binders. As there is currently no binder construct that is in the supported fragment of bv_decide this changes nothing about the proof power.
- #7480 adds the necessary rewrites for the Bitwuzla rules BV_ULT_SPECIAL_CONST, BV_SIGN_EXTEND_ELIM, TODO.
- #7486 adds the BitVec.add_neg_mul rule introduced in #7481 to bv_decide's preprocessor.
- #7491 achieves a speed up in bv_decide's LRAT checker by improving its input validation.
- #7521 adds the equivalent of `Array.emptyWithCapacity` to the AIG framework and applies it to `bv_decide`. This is particularly useful as we are only working with capacities that are always known at run time so we should never have to reallocate a `RefVec`.
- #7527 adds the BV_EXTRACT_CONCAT_LHS_RHS, NORM_BV_ADD_MUL and NORM_BV_SHL_NEG rewrite from Bitwuzla as well as a reduction from getLsbD to extractLsb' to bv_decide.
- #7615 adds the ADD part of bitwuzlas BV_EXTRACT_ADD_MUL rule to bv_decide's preprocessor.
- #7617 adds the known bits optimization from the multiplication circuit to the add one, allowing us to discover potentially even more symmetries before going to the SAT solver.
- #7636 makes sure that the expression level cache in bv_decide is maintained across the entire bitblaster instead of just locally per BitVec expression.
- #7644 adds a cache to the reflection procedure of bv_decide.
- #7649 changes the AIG representation of constants from `const (b : Bool)` to a single constructor `false`. Since #7381 `Ref` contains an `invert` flag meaning the constant `true` can be represented as a `Ref` to `false` with `invert` set, so no expressivity is lost.
- #7655 adds the preprocessing rule for extraction over multiplication to bv_decide.
- #7663 uses computed fields to store the hash code and pointer equality to increase performance of comparison and hashmap lookups on the core data structure used by the bitblaster.
- #7670 improves the caching computation of the atoms assignment in bv_decide's reflection procedure.
- #7698 adds more sharing and caching procedures to bv_decide's reflection step.
- #7720 compresses the AIG representation by storing the inverter bit in the lowest bit of the gate descriptor instead of as a separate `Bool`.
- #7727 avoids some unnecessary allocations in the CNF to dimacs conversion

- #7733 ensures that in the AIG the constant circuit node is always stored at the first spot. This allows us to skip performing a cache lookup when we require a constant node.

Grind

- #7355 fixes a bug in the `markNestedProofs` preprocessor used in the `grind` tactic.
- #7392 fixes an issue in the `grind` tactic when case splitting on if-then-else expressions.
- #7510 ensures that `grind` can be used as a more powerful `contradiction` tactic, sparing the user from having to type `exfalso; grind` or `intros; exfalso; grind`.
- #7512 adds missing normalization rules for `Nat` div and mod to the `grind` tactic.
- #7514 adds more missing normalization rules for `div` and `mod` to `grind`.
- #7532 fixes the procedure for putting new facts into the `grind` "to-do" list. It ensures the new facts are preprocessed. also removes some of the clutter in the `Nat.sub` support.
- #7540 adds `[grind cases eager]` attribute to `Subtype`. See new test.
- #7553 removes a bad normalization rule in `grind`, and adds a missing dsimproc.
- #7641 implements basic model-based theory combination in `grind`. `grind` can now solve examples such as

```
example (f : Int → Int) (x : Int)
    : 0 ≤ x → x ≠ 0 → x ≤ 1 → f x = 2 → f 1 = 2 := by
  grind
```

- #7712 ensures `grind` always abstract its own proofs into an auxiliary definition/theorem. This is similar to #5998 but for `grind`
- #7714 fixes an assertion violation in the `grind` model-based theory combination module.
- #7723 adds the configuration options `zeta` and `zetaDelta` in `grind`. Both are set to `true` by default.
- #7724 adds `dite_eq_ite` normalization rule to `grind`. This rule is important to adjust mismatches between a definition and its function induction principle.
- #7726 fixes the `markNestedProofs` procedure used in `grind`. It was missing the case where the type of a nested proof may contain other nested proofs.
- #7760 ensures `grind` is using the default transparency setting when computing auxiliary congruence lemmas.
- #7765 improves how `grind` normalizes dependent implications during introduction. Previously, `grind` would introduce a hypothesis `h : p` for a goal of the form `.. ⊢ (h : p) → q h`, and then normalize and assert a non-dependent copy of p. As a result, the local context would contain both `h : p` and a separate `h' : p'`, where `p'` is the normal form of p. Moreover, q would still depend on the original h.
- #7776 improves the equality proof discharger used by the E-matching procedure in `grind`.
- #7777 fixes the introduction procedure used in `grind`. It was not registering local instances that are also propositions. See new test.

- #7778 adds missing propagation rules for `LawfulBEq A` to `grind`. They are needed in a context where the instance `DecidableEq A` is not available. See new test.
- #7781 adds a new propagation rule for `Bool` disequalities to `grind`. It now propagates `x = true` (`x = false`) from the disequality `x = false` (`x = true`). It ensures we don't have to perform case analysis on `x` to learn this fact. See tests.

CutSat

- #7312 implements proof term generation for `cooper_dvd_left` and its variants in the cutsat procedure for linear integer arithmetic.
- #7315 implements the Cooper conflict resolution in cutsat. We still need to implement the backtracking and disequality case.
- #7339 implements cooper conflict resolution in the cutsat procedure. It also fixes several bugs in the proof term construction. We still need to add more tests, but we can already solve the following example that `omega` fails to solve:

```
example (x y : Int) :
    27 ≤ 11*x + 13*y →
    11*x + 13*y ≤ 45 →
    -10 ≤ 7*x - 9*y →
    7*x - 9*y ≤ 4 → False := by
  grind
```

- #7351 ensures cutsat does not have to perform case analysis in the univariate polynomial case. That it, it can close a goal whenever there is no solution for a divisibility constraint in an interval. Example of theorem that is now proved in a single step by cutsat:

```
example (x : Int) : 100 ≤ x → x ≤ 10000 → 20000 ∣ 3*x → False :=
by
  grind
```

- #7357 adds support for `/` and `%` to the cutsat procedure.
- #7369 uses `let`-declarations for each polynomial occurring in a proof term generated by the cutsat procedure.
- #7370 simplifies the proof term due to the Cooper's conflict resolution in cutsat.
- #7373 implements the last missing case for the cutsat procedure and fixes a bug. During model construction, we may encounter a bounded interval containing integer solutions that satisfy the divisibility constraint but fail to satisfy known disequalities.
- #7394 adds infrastructure necessary for supporting `Nat` in the cutsat procedure. It also makes the `grind` more robust.
- #7396 fixes a bug in the cutsat model construction. It was searching for a solution in the wrong direction.
- #7401 improves the cutsat model search procedure by tightening inequalities using divisibility constraints.
- #7494 implements support for `Nat` inequalities in the cutsat procedure.

- #7495 implements support for `Nat` divisibility constraints in the cutsat procedure.
- #7501 implements support for `Nat` equalities and disequalities in the cutsat procedure.
- #7502 implements support for `Nat` div and mod in the cutsat procedure.
- #7503 implements support for `Nat.sub` in cutsat
- #7536 implements support for `¬ d ∣ p` in the cutsat procedure.
- #7537 implements support for `Int.natAbs` and `Int.toNat` in the cutsat procedure.
- #7538 fixes a bug in the cutsat model construction. It was not resetting the decision stack at the end of the search.
- #7561 fixes the support for nonlinear `Nat` terms in cutsat. For example, cutsat was failing in the following example

  ```
  example (i j k l : Nat) : i / j + k + l - k = i / j + l := by grind
  ```

  because we were not adding the fact that `i / j` is non negative when we inject the `Nat` expression into `Int`.
- #7579 improves the counterexamples produced by the cutsat procedure, and adds proper support for `Nat`. Before this PR, the assignment for an natural variable `x` would be represented as `NatCast.natCast x`.

Library

- #6496 adds short-circuit support to bv_decide to accelerate multiplications with shared coefficients. In particular, `a * x = b * x` can be extended to `a = b ∨ (a * x = b * x)`. The latter is faster if `a = b` is true, as `a = b` may be evaluated without considering the multiplication circuit. On the other hand, we require the multiplication circuit, as `a * x = b * x -> a = b` is not always true due to two's complement wrapping.
- #7141 generalizes `cond` to allow the motive to be in `Sort u`, not just `Type u`.
- #7289 adds `getKey_beq`, `getKey_congr` and variants to the hashmap api.
- #7319 continues alignment of lemmas about `Int.ediv`/`fdiv`/`tdiv`, including adding notes about "missing" lemmas that do not apply in one case. Also lemmas about `emod`/`fmod`/`tmod`. There's still more to do.
- #7338 adds @[simp] to `Int.neg_inj`.
- #7341 adds an equivalence relation to the hash map with several lemmas for it.
- #7356 adds lemmas reducing monadic operations with `pure` to the non-monadic counterparts.
- #7358 fills further gaps in the integer division API, and mostly achieves parity between the three variants of integer division. There are still some inequality lemmas about `tdiv` and `fdiv` that are missing, but as they would have quite awkward statements I'm hoping that for now no one is going to miss them.
- #7378 adds lemmas about `Int` that will be required in #7368.

- #7380 moves `DHashMap.Raw.foldRev(M)` into `DHashMap.Raw.Internal`.
- #7406 makes the instance for `Subsingleton (Squash α)` work for `α : Sort u`.
- #7418 renames several hash map lemmas (`get` -> `getElem`) and uses `m[k]?` instead of `get? m k` (and also for `get!` and `get`).
- #7432 adds a consequence of `Nat.add_div` using a divisibility hypothesis.
- #7433 makes `simp` able to simplify basic `for` loops in monads other than `Id`.
- #7435 reviews the `Nat` and `Int` API, making the interfaces more consistent.
- #7445 renames `Array.mkEmpty` to `emptyWithCapacity`. (Similarly for `ByteArray` and `FloatArray`.)
- #7446 prefers using ∅ instead of `.empty` functions. We may later rename `.empty` functions to avoid the naming clash with `EmptyCollection`, and to better express semantics of functions which take an optional capacity argument.
- #7451 renames the member `insert_emptyc_eq` of the `LawfulSingleton` typeclass to `insert_empty_eq` to conform to the recommended spelling of ∅ as `empty`.
- #7466 further cleans up simp lemmas for `Int`.
- #7516 changes the order of arguments for `List.modify` and `List.insertIdx`, making them consistent with `Array`.
- #7522 splits off the required theory about `Nat`, `Fin` and `BitVec` from #7484.
- #7529 upstreams `bind_congr` from Mathlib and proves that the minimum of a sorted list is its head and weakens the antisymmetry condition of `min?_eq_some_iff`. Instead of requiring an `Std.Antisymm` instance, `min?_eq_some_iff` now only expects a proof that the relation is antisymmetric *on the elements of the list*. If the new premise is left out, an autoparam will try to derive it from `Std.Antisymm`, so existing usages of the theorem will most likely continue to work.
- #7541 corrects names of a number of lemmas, where the incorrect name was identified automatically by a tool written by @Rob23oba.
- #7554 adds SMT-LIB operators to detect overflow `BitVec.negOverflow`, according to the SMTLIB standard, and the theorem proving equivalence of such definition with the `BitVec` library functions (`negOverflow_eq`).
- #7558 changes the definition of `Nat.div` and `Nat.mod` to use a structurally recursive, fuel-based implementation rather than well-founded recursion. This leads to more predicable reduction behavior in the kernel.
- #7565 adds `BitVec.toInt_sdiv` plus a lot of related bitvector theory around divisions.
- #7614 marks `Nat.div` and `Nat.modCore` as `irreducible`, to recover the behavior from from before #7558.
- #7672 reviews the implicitness of arguments across List/Array/Vector, generally trying to make arguments implicit where possible, although sometimes correcting propositional arguments which were incorrectly implicit to explicit.

- #7687 provides `Inhabited`, `Ord` (if missing), `TransOrd`, `LawfulEqOrd` and `LawfulBEqOrd` instances for various types, namely `Bool`, `String`, `Nat`, `Int`, `UIntX`, `Option`, `Prod` and date/time types. It also adds a few related theorems, especially about how the `Ord` instance for `Int` relates to `LE` and `LT`.
- #7692 upstreams a small number of ordering lemmas for `Fin` from mathlib.
- #7700 provides `Ord`-related instances such as `TransOrd` for `IntX`, `Ordering`, `BitVec`, `Array`, `List` and `Vector`.
- #7704 adds lemmas about the modulo operation defined on signed bounded integers.
- #7706 performs various cleanup tasks on `Init/Data/UInt/*` and `Init/Data/SInt/*`.
- #7729 replaces `assert!` with `assertBEq` to fix issues where asserts didn't trigger the `ctest` due to being in a separate task. This was caused by panics not being caught in tasks, while IO errors were handled by the `AsyncTask` if we use the `block` function on them.
- #7756 adds lemmas about `Nat.gcd` (some of which are currently present in mathlib).

  BREAKING CHANGE: While many lemmas were renamed and the lemma with the old signature was simply deprecated, some lemmas were changed without renaming them. They now use the `getElem` variants instead of `get`.

Async

- #6683 introduces TCP socket support using the LibUV library, enabling asynchronous I/O operations with it.
- #7571 fixes #7478 by modifying `number` specifiers from `atLeast size` to `flexible size` for parsing. This change allows:
  - 1 repetition to accept 1 or more characters
  - More than 1 repetition to require exactly that many characters
- #7574 introduces UDP socket support using the LibUV library, enabling asynchronous I/O operations with it.
- #7578 introduces a function called `interfaceAddresses` that retrieves an array of system's network interfaces.
- #7584 introduces a structure called `FormatConfig`, which provides additional configuration options for `GenericFormat`, such as whether leap seconds should be allowed during parsing. By default, this option is set to `false`.
- #7751 adds `Std.BaseMutex.tryLock` and `Std.Mutex.tryAtomically` as well as unit tests for our locking and condition variable primitives.
- #7755 adds `Std.RecursiveMutex` as a recursive/reentrant equivalent to `Std.Mutex`.
- #7771 adds a barrier primitive as `Std.Barrier`.

Finite Types

- #7228 adds simprocs to reduce expressions involving `IntX`.

- #7274 adds lemmas about iterated conversions between finite types, starting with something of type `IntX`.
- #7340 adds lemmas for iterated conversions between finite types which start with `Nat`/`Int`/`Fin`/`BitVec` and then go through `UIntX`.
- #7368 adds lemmas for iterated conversions between finite types, starting with something of type `Nat`/`Int`/`Fin`/`BitVec` and going through `IntX`.
- #7414 adds the remaining lemmas about iterated conversions of finite type that go through signed or unsigned bounded integers.
- #7484 adds some lemmas about operations defined on `UIntX`
- #7487 adds the instance `Neg UInt8`.
- #7592 adds theory about signed finite integers relating operations and conversion functions.
- #7598 adds miscellaneous results about `Nat` and `BitVec` that will be required for `IntX` theory (#7592).
- #7685 contains additional material about `BitVec` and `Int` spun off from #7592.
- #7694 contains additional material on `BitVec`, `Int` and `Nat`, split off from #7592.

Tree Map

- #7270 provides lemmas about the tree map functions `foldlM`, `foldl`, `foldrM` and `foldr` and their interactions with other functions for which lemmas already exist. Additionally, it generalizes the `fold*`/`keys` lemmas to arbitrary tree maps, which were previously stated only for the `DTreeMap α Unit` case.
- #7331 provides lemmas about the tree map function `insertMany` and its interaction with other functions for which lemmas already exist. Most lemmas about `ofList`, which is related to `insertMany`, are not included.
- #7360 provides lemmas about the tree map function `ofList` and interactions with other functions for which lemmas already exist.
- #7367 provides lemmas for the tree map functions `alter` and `modify` and their interactions with other functions for which lemmas already exist.

  BREAKING CHANGE: The signature of `size_alter` was corrected for all four hash map types. Instead of relying on the boolean operations `contains` and && in the if statements, we now use the `Prop`-based operations `Membership` and `And`.
- #7412 provides lemmas about the tree map that have been introduced to the hash map in #7289.
- #7419 provides lemmas about the tree map function `modify` and its interactions with other functions for which lemmas already exist.
- #7437 provides (some but not all) lemmas about the tree map function `minKey?`.
- #7556 provides lemmas about the tree map function `minKey?` and its interaction with other functions for which lemmas already exist.

- #7600 provides lemmas about the tree map function `minKey!` and its interactions with other functions for which lemmas already exist.
- #7626 provides lemmas for the tree map function `minKeyD` and its interations with other functions for which lemmas already exist.
- #7657 provides lemmas for the tree map function `maxKey?` and its interations with other functions for which lemmas already exist.
- #7660 provides lemmas for the tree map function `minKey` and its interations with other functions for which lemmas already exist.
- #7664 fixes a bug in the definition of the tree map functions `maxKey` and `maxEntry`. Moreover, it provides lemmas for this function and its interactions with other function for which lemmas already exist.
- #7674 add missing lemmas about the tree map: `minKey*` variants return the head of `keys`, `keys` and `toList` are ordered and `getKey* t.minKey?` equals the minimum.
- #7675 provides lemmas about the tree map function `maxKeyD` and its interactions with other functions for which lemmas already exist.
- #7686 provides lemmas for the tree map function `maxKey!` and its interactions with other functions for which lemmas already exist.
- #7695 removes simp lemmas about the tree map with a metavariable in the head of the discrimination pattern.
- #7697 is a follow-up to #7695, which removed `simp` attributes from tree map lemmas with bad discrimination patterns. In this PR, we introduce some `Ord`-based lemmas that are more simp-friendly.

BitVec API

- #7104 adds `BitVec.[toNat|toFin|toInt]_[sshiftRight|sshiftRight']` plus variants with `of_msb_*`. While at it, we also add `toInt_zero_length` and `toInt_of_zero_length`. In support of our main theorem we add `toInt_shiftRight_lt` and `le_toInt_shiftRight`, which make the main theorem automatically derivable via omega.
- #7225 contains `BitVec.(toInt, toFin)_twoPow` theorems, completing the API for `BitVec.*_twoPow`. It also expands the `toNat_twoPow` API with `toNat_twoPow_of_le`, `toNat_twoPow_of_lt`, as well as `toNat_twoPow_eq_if` and moves `msb_twoPow` up, as it is used in the `toInt_msb` proof.
- #7415 adds a few lemmas about the interactions of `BitVec` with `Fin` and `Nat`.
- #7420 generalizes `BitVec.toInt_[lt|le]'` to not require `0 < w`.
- #7465 adds the theorem:

  ```
  theorem lt_allOnes_iff {x : BitVec w} : x < allOnes w ↔ x ≠
  allOnes w
  ```

  to simplify comparisons against `-1#w`. This is a corollary of the existing lemma:

```
theorem allOnes_le_iff {x : BitVec w} : allOnes w ≤ x ↔ x =
allOnes w
```

- #7599 adds SMT-LIB operators to detect overflow `BitVec.(usubOverflow, ssubOverflow)`, according to the SMTLIB standard, and the theorems proving equivalence of such definition with the `BitVec` library functions `BittVec.(usubOverflow_eq, ssubOverflow_eq)`.
- #7604 adds bitvector theorems that to push negation into other operations, following Hacker's Delight: Ch2.1.
- #7605 adds theorems `BitVec.[(toInt, toFin)_(extractLsb, extractLsb')]`, completing the API for `BitVec.(extractLsb, extractLsb')`.
- #7616 introduces `BitVec.(toInt, toFin)_rotate(Left, Right)`, completing the API for `BitVec.rotate(Left, Right)`
- #7658 introduces `BitVec.(toFin_signExtend_of_le, toFin_signExtend)`, completing the API for `BitVec.signExtend`.
- #7661 adds theorems `BitVec.[(toFin, toInt)_setWidth', msb_setWidth'_of_lt, toNat_lt_twoPow_of_le, toInt_setWidth'_of_lt]`, completing the API for `BitVec.setWidth'`.
- #7699 adds the `BitVec.toInt_srem` lemma, relating `BitVec.srem` with `Int.tmod`.

Bitwuzla Rewrite Rules

- #7424 proves Bitwuzla's rule `BV_ZERO_EXTEND_ELIM`:

```
theorem setWidth_eq_append {v : Nat} {x : BitVec v} {w : Nat} (h
: v ≤ w) :
    x.setWidth w = ((0#(w - v)) ++ x).cast (by omega) := by
```

- #7426 adds the Bitwuzla rewrite rule `BV_EXTRACT_FULL`, which is useful for the bitblaster to simplify `extractLsb'` based expressions.
- #7427 implements the bitwuzla rule `BV_CONCAT_EXTRACT`. This will be used by the bitblaster to simplify adjacent `extracts` into a single `extract`.
- #7454 implements the bitwuzla rule `BV_SIGN_EXTEND_ELIM`, which rewrites a `signExtend x` as an append of the appropriate sign bits, followed by the bits of `x`.
- #7461 introduces a bitvector associativity/commutativity normalization on bitvector terms of the form `(a * b) = (c * d)` for `a, b, c, d` bitvectors. This mirrors Bitwuzla's `PassNormalize::process`'s `PassNormalize::normalize_eq_add_mul`.
- #7481 implements the Bitwuzla rewrites `BV_ADD_NEG_MUL`, and associated lemmas to make the proof streamlined. `bvneg (bvadd a (bvmul a b)) = (bvmul a (bvnot b))`, or spelled as lean:

```
theorem neg_add_mul_eq_mul_not {x y : BitVec w} :
    - (x + x * y) = (x * ~~~ y)
```

- #7482 implements the BV_EXTRACT_CONCAT rule from Bitwuzla, which explains how to extract bits from an append. We first prove a 'master theorem' which has the full case analysis, from which we rapidly derive the necessary BV_EXTRACT_CONCAT theorems:

```
theorem extractLsb'_append_eq_ite {v w} {xhi : BitVec v} {xlo :
BitVec w} {start len : Nat} :
    extractLsb' start len (xhi ++ xlo) =
    if hstart : start < w
    then
      if hlen : start + len < w
      then extractLsb' start len xlo
      else
        (((extractLsb' (start - w) (len - (w - start)) xhi) ++
            extractLsb' start (w - start) xlo)).cast (by omega)
    else
      extractLsb' (start - w) len xhi
```

- #7493 implements the Bitwuzla rewrite rule NORM_BV_ADD_MUL, and the associated lemmas to allow for expedient rewriting:

```
theorem neg_add_mul_eq_mul_not {x y : BitVec w} : - (x + x * y) =
x * ~~~ y
```

- #7508 shows that negation commutes with left shift, which is the Bitwuzla rewrite NORM_BV_SHL_NEG.
- #7594 implements the Bitwuzla rewrites BV_EXTRACT_ADD_MUL, which witness that the high bits at `i >= len` do not affect the bits of the product upto `len`.
- #7595 implements the addition rewrite from the Bitwuzla rewrite BV_EXTRACT_ADD_MUL, which witness that the high bits at `i >= len` do not affect the bits of the sum upto `len`:

```
theorem extractLsb'_add {w len} {x y : BitVec w} (hlen : len ≤ w)
:
    (x + y).extractLsb' 0 len = x.extractLsb' 0 len +
y.extractLsb' 0 len
```

- #7757 adds the Bitwuzla rewrite NORM_BV_ADD_CONCAT for symbolic simplification of add-of-append.

Compiler

- #7398 fixes a scoping error in the cce (Common Case Elimination) pass of the old code generator. This pass would create a join point for common minor premises even if some of those premises were in the bodies of locally defined functions, which results in an improperly scoped reference to a join point. The fix is to save/restore candidates when visiting a lambda.
- #7710 improves memory use of Lean, especially for longer-running server processes, by up to 60%

Pretty Printing

- #7589 changes the structure instance notation pretty printer so that fields are omitted if their value is definitionally equal to the default value for the field (up to reducible transparancy). Setting `pp.structureInstances.defaults` to true forces such fields to be pretty printed anyway.

Documentation

- #7198 makes the docstrings in the `Char` namespace follow the documentation conventions.
- #7204 adds docstrings for the `Id` monad.
- #7246 updates existing docstrings for Bool and adds the missing ones.
- #7288 fixes the doc of `List.removeAll`
- #7365 updates docstrings and adds some that are missing.
- #7452 makes the style of all `List` docstrings that appear in the language reference consistent.
- #7476 adds missing docstrings for `IO` and related code and makes the style of the existing docstrings consistent.
- #7492 adds missing `Array` docstrings and makes their style consistent.
- #7506 adds missing `String` docstrings and makes the existing ones consistent in style.
- #7523 adds missing docstrings and makes docstring style consistent for `System` and `System.FilePath`.
- #7528 makes the docstrings for Thunk consistent with the style of the others.
- #7534 adds missing `Syntax`-related docstrings and makes the existing ones consistent in style with the others.
- #7535 revises the docstring for `funext`, making it more concise and adding a reference to the manual for more details.
- #7548 adds missing monad transformer docstrings and makes their style consistent.
- #7552 adds missing `Nat` docstrings and makes their style consistent.
- #7564 updates the docstrings for `ULift` and `PLift`, making their style consistent with the others.
- #7568 adds missing `Int` docstrings and makes the style of all of them consistent.
- #7602 adds missing docstrings for fixed-width integer operations and makes their style consistent.
- #7607 adds docstrings for `String.drop` and `String.dropRight`.
- #7613 adds a variety of docstrings for names that appear in the manual.
- #7635 adds missing docstrings for `Substring` and makes the style of `Substring` docstrings consistent.
- #7642 reviews the docstrings for `Float` and `Float32`, adding missing ones and making their format consistent.

- #7645 adds missing docstrings and makes docstring style consistent for `ForM`, `ForIn`, `ForIn'`, `ForInStep`, `IntCast`, and `NatCast`.
- #7711 adds the last few missing docstrings that appear in the manual.
- #7713 makes the BitVec docstrings match each other and the rest of the API in style.

Server

- #7178 fixes a race condition in the language server that would sometimes cause it to drop requests and never respond to them when editing the header of a file. This in turn could cause semantic highlighting to stop functioning in VS Code, as VS Code would stop emitting requests when a prior request was dropped, and also cause the InfoView to become defective. It would also cause import auto-completion to feel a bit wonky, since these requests were sometimes dropped. This race condition has been present in the language server since its first version in 2020.
- #7223 implements parallel watchdog request processing so that requests that are processed by the watchdog cannot block the main thread of the watchdog anymore.
- #7240 adds a canonical syntax for linking to sections in the language reference along with formatting of examples in docstrings according to the docstring style guide.
- #7343 mitigates an issue where inserting an inlay hint in VS Code by double-clicking would insert the inlay hint at the wrong position right after an edit.
- #7344 combines the auto-implicit inlay hint tooltips into a single tooltip. This works around an issue in VS Code where VS Code fails to update hovers for tooltips in adjacent inlay hint parts when moving the mouse.
- #7346 fixes an issue where the language server would run into an inlay hint assertion violation when deleting a file that is still open in the language server.
- #7366 adds server-side support for dedicated 'unsolved goals' and 'goals accomplished' diagnostics that will have special support in the Lean 4 VS Code extension. The special 'unsolved goals' diagnostic is adapted from the 'unsolved goals' error diagnostic, while the 'goals accomplished' diagnostic is issued when a `theorem` or `Prop`-typed `example` has no errors or `sorry`s. The Lean 4 VS Code extension companion PR is at leanprover/vscode-lean4#585.
- #7376 ensures `weak` options do not have to be repeated in both Lake `leanOptions` and `moreServerOptions`.
- #7882 fixes a regression where elaboration of a previous document version is not cancelled on changes to the document.

Lake

- #7185 refactors Lake's build internals to enable the introduction of targets and facets beyond packages, modules, and libraries. Facets, build keys, build info, and CLI commands have been generalized to arbitrary target types.

- #7393 adds autocompletion support for Lake configuration fields in the Lean DSL at the indented whitespace after an existing field. Autocompletion in the absence of any fields is currently still not supported.

  Breaking change: The nonstandard braced configuration syntax now uses a semicolon `;` rather than a comma `,` as a separator. Indentation can still be used as an alternative to the separator.

- #7399 reverts the new builtin initializers, elaborators, and macros in Lake back to non-builtin.
- #7504 augments the Lake configuration data structures declarations (e.g., `PackageConfig`, `LeanLibConfig`) to produce additional metadata which is used to automatically generate the Lean & TOML encoders and decoders via metaprograms.
- #7543 unifies the configuration declarations of dynamic targets, external libraries, Lean libraries, and Lean executables into a single data type stored in a unified map within a package.

  Breaking change: Users can no longer define multiple targets with the same name but different kinds (e.g., a Lean executable and a Lean library both named `foo`). This should not effect most users as the Lake DSL already discouraged this.

- #7576 changes Lake to produce and use response files on Windows when building executables and libraries (static and shared). This is done to avoid potentially exceeding Windows command line length limits.
- #7586 changes the `static.export` facet for Lean libraries to produce thin static libraries.
- #7608 removes the use of the Lake plugin in the Lake build and in configuration files.
- #7667 changes Lake to log messages from a Lean configuration the same way it logs message from a Lean build. This, for instance, removes redundant severity captions.
- #7703 adds `input_file` and `input_dir` as new target types. It also adds the `needs` configuration option for Lean libraries and executables. This option generalizes `extraDepTargets` (which will be deprecated in the future), providing much richer support for declaring dependencies across package and target type boundaries.
- #7716 adds the `moreLinkObjs` and `moreLinkLibs` options for Lean packages, libraries, and executables. These serves as functional replacements for `extern_lib` and provided additional flexibility.

  Breaking change: `precompileModules` now only loads modules of the current library individually. Modules of other libraries are loaded together via that library's shared library.

- #7732 deprecates `extraDepTargets` and fixes a bug caused by the configuration refactor.
- #7758 removes the `-lstdcpp` extra link argument from the FFI example. It is not actually necessary.
- #7763 corrects build key fetches to produce jobs with the proper data kinds and fixes a failed coercion from key literals to targets.

Other

- #7326 updates the release notes script to better indent PR descriptions.
- #7453 adds "(kernel)" to the message for the kernel-level application type mismatch error.
- #7769 fixes a number of bugs in the release automation scripts, adds a script to merge tags into remote `stable` branches, and makes the main `release_checklist.py` script give suggestions to call the `merge_remote.py` and `release_steps.py` scripts when needed.

Lean 4.18.0 (2025-04-02)

For this release, 344 changes landed. In addition to the 166 feature additions and 38 fixes listed below there were 13 refactoring changes, 10 documentation improvements, 3 performance improvements, 4 improvements to the test suite and 109 other changes.

Highlights

Lean v4.18 release brings a number of exciting new features:

- Inlay hints for auto implicits

  The Language Server now uses inlay hints to show which variables are brought into scope implicitly, and where. The hint reveals its type upon hover, and double-clicking the hint will insert the variable binding explicitly.

  See the description of #6768 for a screenshort.

  Note that this feature is only visible when `set_option autoImplicit true`, which is the default in plain Lean projects, but not in mathlib

- #6935 adds the tactic `expose_names`. It creates a new goal whose local context has been "exposed" so that every local declaration has a clear, accessible name. If no local declarations require renaming, the original goal is returned unchanged.

```
/--
info: α : Sort u_1
a b : α
h_1 : a = b
h_2 : True
h_3 : True ∨ False
h : b = a
⊢ b = a
-/
#guard_msgs (info) in
example (a b : α) (h : a = b) (_ : True) (_ : True ∨ False) (h :
b = a) : b = a := by
  expose_names
  trace_state
  rw [h]
```

This tactic intended for use in auto-generated tactic suggestions, and can also be useful during proof exploration. It is still best practice to name variables where they are brought into scope (`intro`, `case` etc.), and not use `expose_names` in a finished, polished proof.

- #7069 adds the `fun_induction` and `fun_cases` tactics, which add convenience around using functional induction and functional cases principles.

  `fun_induction foo  x y z`

  elaborates `foo x y z`, then looks up `foo.induct`, and then essentially does

  `induction z using foo.induct y`

  including and in particular figuring out which arguments are parameters, targets or dropped. This only works for non-mutual functions so far.

  Likewise there is the `fun_cases` tactic using `foo.fun_cases`.

- #6744 extend the preprocessing of well-founded recursive definitions to bring assumptions like `h✝ : x ∈ xs`, if a recursive call is in the argument of a higher-order function like `List.map`, into scope automatically. In many cases this removes the need to use functions like `List.attach`.

  This feature can be disabled with `set_option wf.preprocess false`.

- #6634 adds support for changing the binder annotations of existing variables to and from strict-implicit and instance-implicit using the `variable` command.

- #7100 modifies the `structure` syntax so that parents can be named, like in

  `structure S extends toParent : P`

  Breaking change: The syntax is also modified so that the resultant type comes *before* the `extends` clause, for example `structure S : Prop extends P`. This is necessary to prevent a parsing ambiguity, but also this is the natural place for the resultant type. Implements RFC #7099.

- #7103 gives the `induction` tactic the ability to name hypotheses to use when generalizing targets, just like in `cases`. For example, `induction h : xs.length` leads to goals with hypotheses `h : xs.length = 0` and `h : xs.length = n + 1`. Target handling is also slightly modified for multi-target induction principles: it used to be that if any target was not a free variable, all of the targets would be generalized (thus causing free variables to lose their connection to the local hypotheses they appear in); now only the non-free-variable targets are generalized.

- #6869 adds a `recommended_spelling` command, which can be used for recording the recommended spelling of a notation (for example, that the recommended spelling of ∧ in identifiers is and). This information is then appended to the relevant docstrings for easy lookup.

- #6893 adds support for plugins to the frontend and server.

- #7061 provides a basic API for a premise selection tool, which can be provided in downstream libraries. It does not implement premise selection itself!

And many more! Check out the *Language* section below.

Notably, a line of work has been carried out on the following (see the corresponding subsections in the *Language* section for the details):

- the `try?` tactic, which has been re-implemented using `evalAndSuggest` tactic. `try?` now supports referencing inaccessible local names and can provide more complex suggestions, involving `exact?` and `fun_induction` tactics. New configuration options have been added: `-only`, `+missing`, and `max:=<num>`, as well as `merge`.
- `bv_decide` tactic. There are new features in preprocessing, added support for enum inductives, `IntX` and `ISize`, and improved performance in LRAT trimming.
- normalization for linear integer arithmetic expressions has been implemented and connected to `simp +arith`. #7043 deprecates the tactics `simp_arith`, `simp_arith!`, `simp_all_arith` and `simp_all_arith!` in favor of `simp +arith`.

Important Library updates include:

- #6914 introduces ordered map data structures, namely `DTreeMap`, `TreeMap`, `TreeSet` and their `.Raw` variants, into the standard library. A collection of lemmas about the operations on these data structures has been added in the subsequent PRs.
- #7255 fixes the definition of `Min (Option α)`. This is a **breaking change**. This treats `none` as the least element, so `min none x = min x none = none` for all `x : Option α`. Prior to nightly-2025-02-27, we instead had `min none (some x) = min (some x) none = some x`. Also adds basic lemmas relating `min`, `max`, ≤ and < on `Option`.

Significant development has been made in the verification APIs of `BitVec` and fixed-width integer types (`IntX`), along with ongoing work to align `List`/`Array`/`Vector` APIs. Several lemmas about `Int.ediv`/`fdiv`/`tdiv` have been strengthened.

#6950 adds a style guide and a naming convention for the standard library.

This summary of highlights was contributed by Violetta Sim.

Language

- #6634 adds support for changing the binder annotations of existing variables to and from strict-implicit and instance-implicit using the `variable` command.
- #6744 extends the preprocessing of well-founded recursive definitions, see the highlights section for details.
- #6823 adds a builtin tactic and a builtin attribute that are required for the tree map. The tactic, `as_aux_lemma`, can generally be used to wrap the proof term generated by a tactic sequence into a separate auxiliary lemma in order to keep the proof term small. This can, in rare cases,

be necessary if the proof term will appear multiple times in the encompassing term. The new attribute, `Std.Internal.tree_tac`, is internal and should not be used outside of `Std`.

- #6853 adds support for anonymous equality proofs in `match` expressions of the form `match _ : e with ...`.
- #6869 adds a `recommended_spelling` command; see the highlights section for details.
- #6891 modifies `rewrite`/`rw` to abort rewriting if the elaborated lemma has any immediate elaboration errors (detected by presence of synthetic sorries). Rewriting still proceeds if there are elaboration issues arising from pending synthetic metavariables, like instance synthesis failures. The purpose of the change is to avoid obscure "tactic 'rewrite' failed, equality or iff proof expected ?m.5" errors when for example a lemma does not exist.
- #6893 adds support for plugins to the frontend and server.
- #6935 adds the tactic `expose_names`; see the highlights section for details.
- #6936 fixes the `#discr_tree_simp_key` command, because it displays the keys for just `lhs` in `lhs ≠ rhs`, but it should be `lhs = rhs`, since that is what simp indexes.
- #6939 adds error messages for `inductive` declarations with conflicting constructor names and `mutual` declarations with conflicting names.
- #6947 adds the `binderNameHint` gadget. It can be used in rewrite and simp rules to preserve a user-provided name where possible.

  The expression `binderNameHint v binder e` defined to be `e`.

  If it is used on the right-hand side of an equation that is applied by a tactic like `rw` or `simp`, and `v` is a local variable, and `binder` is an expression that (after beta-reduction) is a binder (so `fun w => …` or `∀ w, …`), then it will rename `v` to the name used in the binder, and remove the `binderNameHint`.

  A typical use of this gadget would be as follows; the gadget ensures that after rewriting, the local variable is still `name`, and not `x`:

  ```
  theorem all_eq_not_any_not (l : List α) (p : α → Bool) :
      l.all p = !l.any fun x => binderNameHint x p (!p x) := sorry

  example (names : List String) : names.all (fun name =>
  "Waldo".isPrefixOf name) = true := by
    rw [all_eq_not_any_not]
    -- ⊢ (!names.any fun name => !"Waldo".isPrefixOf name) = true
  ```

  This gadget is supported by `simp`, `dsimp` and `rw` in the right-hand-side of an equation, but not in hypotheses or by other tactics.
- #6951 adds line breaks and indentations to simp's trace messages to make them easier to read (IMHO).
- #6964 adds a convenience command `#info_trees in`, which prints the info trees generated by the following command. It is useful for debugging or learning about `InfoTree`.

- #7039 improves the well-founded definition preprocessing to propagate `wfParam` through let expressions.
- #7053 makes `simp` heed the `binderNameHint` also in the assumptions of congruence rules. Fixes #7052.
- #7055 improves array and vector literal syntax by allowing trailing commas. For example, `#[1, 2, 3,]`.
- #7061 provides a basic API for a premise selection tool; see the highlights section for details.
- #7078 implements simprocs for `Int` and `Nat` divides predicates.
- #7088 fixes the behavior of the indexed-access notation `xs[i]` in cases where the proof of `i`'s validity is filled in during unification.
- #7090 strips `lib` prefixes and `_shared` suffixes from plugin names. It also moves most of the dynlib processing code to Lean to make such preprocessing more standard.
- #7100 modifies the `structure` syntax; see the highlights section for details.
- #7103 gives the `induction` tactic the ability to name hypotheses; see the highlights section for details.
- #7119 makes linter names clickable in the `trace.profiler` output.
- #7191 fixes the indentation of "Try this" suggestions in widget-less multiline messages, as they appear in `#guard_msgs` outputs.
- #7192 prevents `exact?` and `apply?` from suggesting tactics that correspond to correct proofs but do not elaborate, and it allows these tactics to suggest `expose_names` when needed.
- #7200 allows the debug form of DiscrTree.Key to line-wrap.
- #7213 adds `SetConsoleOutputCP(CP_UTF8)` during runtime initialization to properly display Unicode on the Windows console. This effects both the Lean executable itself and user executables (including Lake).
- #7224 make `induction … using` and `cases … using` complain if more targets were given than expected by that eliminator.
- #7294 fixes bugs in `Std.Internal.Rat.floor` and `Std.Internal.Rat.ceil`.

Updates to the `try?` Tactic

- #6961 adds the auxiliary tactic `evalAndSuggest`. It will be used to refactor `try?`.
- #6965 re-implements the `try?` tactic using the new `evalAndSuggest` infrastructure.
- #6967 ensures `try?` can suggest tactics that need to reference inaccessible local names. Example:

```
/--
info: Try these:
• · expose_names; induction as, bs_1 using app.induct <;> grind
[= app]
```

```
• · expose_names; induction as, bs_1 using app.induct <;> grind
only [app]
-/
#guard_msgs (info) in
example : app (app as bs) cs = app as (app bs cs) := by
  have bs := 20 -- shadows `bs` in the target
  try?
```

- #6979 adds support for more complex suggestions in `try?`. Example:

  ```
  example (as : List α) (a : α) : concat as a = as ++ [a] := by
    try?
  ```

  suggestion

  ```
  Try this: · induction as, a using concat.induct
    · rfl
    · simp_all
  ```

- #6980 improves the `try?` tactic runtime validation and error messages. It also simplifies the implementation, and removes unnecessary code.
- #6981 adds new configuration options to `try?`.
  - `try? -only` omits `simp only` and `grind only` suggestions
  - `try? +missing` enables partial solutions where some subgoals are "solved" using `sorry`, and must be manually proved by the user.
  - `try? (max:=<num>)` sets the maximum number of suggestions produced (default is 8).
- #6991 improves how suggestions for the `<;>` combinator are generated.
- #6994 adds the `Try.Config.merge` flag (`true` by default) to the `try?` tactic. When set to `true`, `try?` compresses suggestions such as:

  ```
  · induction xs, ys using bla.induct
      · grind only [List.length_reverse]
      · grind only [bla]
  ```

  into:

  ```
  induction xs, ys using bla.induct <;> grind only
  [List.length_reverse, bla]
  ```

- #6995 implements support for `exact?` in the `try?` tactic.
- #7082 makes `try?` use `fun_induction` instead of `induction … using foo.induct`. It uses the argument-free short-hand `fun_induction foo` if that is unambiguous. Avoids `expose_names` if not necessary by simply trying without first.

Functional Induction Tactic

- #7069 adds the `fun_induction` and `fun_cases` tactics, which add convenience around using functional induction and functional cases principles.

- #7101 implements `fun_induction foo`, which is like `fun_induction foo x y z`, only that it picks the arguments to use from a unique suitable call to `foo` in the goal.
- #7127 follows up on #7103 which changes the generaliziation behavior of `induction`, to keep `fun_induction` in sync. Also fixes a `Syntax` indexing off-by-one error.

`bv_decide` Tactic

- #6741 implements two rules for bv_decide's preprocessor, lowering `|||` to `&&&` in order to enable more term sharing + application of rules about `&&&` as well as rewrites of the form `(a &&& b == -1#w) = (a == -1#w && b == -1#w)` in order to preserve rewriting behavior that already existed before this lowering.
- #6924 adds the EQUAL_ITE rules from Bitwuzla to the preprocessor of bv_decide.
- #6926 adds the BV_EQUAL_CONST_NOT rules from Bitwuzla to the preprocessor of bv_decide.
- #6946 implements basic support for handling of enum inductives in `bv_decide`. It now supports equality on enum inductive variables (or other uninterpreted atoms) and constants.
- #7009 ensures users get an error message saying which module to import when they try to use `bv_decide`.
- #7019 properly spells out the trace nodes in bv_decide so they are visible with just `trace.Meta.Tactic.bv` and `trace.Meta.Tactic.sat` instead of always having to enable the profiler.
- #7021 adds theorems for interactions of extractLsb with `&&&`, `^^^`, `~~~` and `bif` to bv_decide's preprocessor.
- #7029 adds simprocs to bv_decide's preprocessor that rewrite multiplication with powers of two to constant shifts.
- #7033 improves presentation of counter examples for UIntX and enum inductives in bv_decide.
- #7242 makes sure bv_decide can work with projections applied to `ite` and `cond` in its structures pass.
- #7257 improves performance of LRAT trimming in bv_decide.
- #7269 implements support for `IntX` and `ISize` in `bv_decide`.
- #7275 adds all level 1 rewrites from Bitwuzla to the preprocessor of bv_decide.

Parallelizing Elaboration

- #6770 enables code generation to proceed in parallel to further elaboration.
- #6988 ensures interrupting the kernel does not lead to wrong, sticky error messages in the editor
- #7047 removes the `save` and `checkpoint` tactics that have been superseded by incremental elaboration
- #7076 introduces the central parallelism API for ensuring that helper declarations can be generated lazily without duplicating work or creating conflicts across threads.

Linear Integer Normalization in `simp +arith`

- #7000 adds helper theorems for justifying the linear integer normalizer.
- #7002 implements the normalizer for linear integer arithmetic expressions. It is not connect to `simp +arith` yet because of some spurious `[simp]` attributes.
- #7011 adds `simp +arith` for integers. It uses the new `grind` normalizer for linear integer arithmetic. We still need to implement support for dividing the coefficients by their GCD. It also fixes several bugs in the normalizer.
- #7015 makes sure `simp +arith` normalizes coefficients in linear integer polynomials. There is still one todo: tightening the bound of inequalities.
- #7030 adds completes the linear integer inequality normalizer for `grind`. The missing normalization step replaces a linear inequality of the form `a_1*x_1 + ... + a_n*x_n + b <= 0` with `a_1/k * x_1 + ... + a_n/k * x_n + ceil(b/k) <= 0` where `k = gcd(a_1, ..., a_n)`. `ceil(b/k)` is implemented using the helper `cdiv b k`.
- #7040 ensures that terms such as `f (2*x + y)` and `f (y + x + x)` have the same normal form when using `simp +arith`
- #7043 deprecates the tactics `simp_arith`, `simp_arith!`, `simp_all_arith` and `simp_all_arith!`. Users can just use the `+arith` option.

`grind` Tactic

The `grind` tactic is still is experimental and still under development. Avoid using it in production projects

- #6902 ensures `simp` diagnostic information in included in the `grind` diagnostic message.
- #6937 improves `grind` error and trace messages by cleaning up local declaration names.
- #6940 improves how the `grind` tactic performs case splits on `p <-> q`.
- #7102 modifies `grind` to run with the `reducible` transparency setting. We do not want `grind` to unfold arbitrary terms during definitional equality tests. also fixes several issues introduced by this change. The most common problem was the lack of a hint in proofs, particularly in those constructed using proof by reflection. also introduces new sanity checks when `set_option grind.debug true` is used.
- #7231 implements functions for constructing disequality proofs in `grind`.

Cutsat Procedure (Solver for Linear Integer Arithmetic Problems)

- #7077 proves the helper theorems for justifying the "Div-Solve" rule in the cutsat procedure.
- #7091 adds helper theorems for normalizing divisibility constraints. They are going to be used to implement the cutsat procedure in the `grind` tactic.
- #7092 implements divisibility constraint normalization in `simp +arith`.
- #7097 implements several modifications for the cutsat procedure in `grind`.

  - The maximal variable is now at the beginning of linear polynomials.
  - The old `LinearArith.Solver` was deleted, and the normalizer was moved to `Simp`.
  - cutsat first files were created, and basic infrastructure for representing divisibility constraints was added.
- #7122 implements the divisibility constraint solver for the cutsat procedure in the `grind` tactic.
- #7124 adds the helper theorems for justifying the divisibility constraint solver in the cutsat procedure used by the `grind` tactic.
- #7138 implements proof generation for the divisibility constraint solver in `grind`.
- #7139 uses a `let`-expression for storing the (shared) context in proofs produced by the cutsat procedure in `grind`.
- #7152 implements the infrastructure for supporting integer inequality constraints in the cutsat procedure.
- #7155 implements some infrastructure for the model search procedure in cutsat.
- #7156 adds a helper theorem that will be used in divisibility constraint conflict resolution during model construction.
- #7176 implements model construction for divisibility constraints in the cutsat procedure.
- #7183 improves the cutsat model search procedure.
- #7186 simplifies the proofs and data structures used by cutsat.
- #7193 adds basic infrastructure for adding support for equalities in cutsat.
- #7194 adds support theorems for solving equality in cutsat.
- #7202 adds support for internalizing terms relevant to the cutsat module. This is required to implement equality propagation.
- #7203 improves the support for equalities in cutsat. It also simplifies a few support theorems used to justify cutsat rules.
- #7217 improves the support for equalities in cutsat.
- #7220 implements the missing cases for equality propagation from the `grind` core to the cutsat module.
- #7234 implements dIsequality propagation from `grind` core module to cutsat.
- #7244 adds support for disequalities in the cutsat procedure used in `grind`.
- #7248 implements simple equality propagation in cutsat `p <= 0 -> -p <= 0 -> p = 0`
- #7252 implements inequality refinement using disequalities. It minimizes the number of case splits cutsat will have to perform.
- #7267 improves the cutsat search procedure. It adds support for find an approximate rational solution, checks disequalities, and adds stubs for all missing cases.

- #7278 adds counterexamples for linear integer constraints in the `grind` tactic. This feature is implemented in the cutsat procedure.
- #7279 adds support theorems for the **Cooper-Dvd-Left** conflict resolution rule used in the cutsat procedure. During model construction, when attempting to extend the model to a variable `x`, cutsat may find a conflict that involves two inequalities (the lower and upper bounds for `x`) and a divisibility constraint:

  ```
  a * x + p ≤ 0
  b * x + q ≤ 0
  d ∣ c * x + s
  ```

- #7284 implements non-choronological backtracking for the cutsat procedure. The procedure has two main kinds of case-splits: disequalities and Cooper resolvents. focus on the first kind.
- #7290 adds support theorems for the **Cooper-Left** conflict resolution rule used in the cutsat procedure. During model construction,when attempting to extend the model to a variable `x`, cutsat may find a conflict that involves two inequalities (the lower and upper bounds for `x`). This is a special case of Cooper-Dvd-Left when there is no divisibility constraint.
- #7292 adds support theorems for the **Cooper-Dvd-Right** conflict resolution rule used in the cutsat procedure. During model construction, when attempting to extend the model to a variable `x`, cutsat may find a conflict that involves two inequalities (the lower and upper bounds for `x`) and a divisibility constraint.
- #7293 adds support theorems for the Cooper-Right conflict resolution rule used in the cutsat procedure. During model construction, when attempting to extend the model to a variable `x`, cutsat may find a conflict that involves two inequalities (the lower and upper bounds for `x`). This is a special case of Cooper-Dvd-Right when there is no divisibility constraint.
- #7409 allows the use of `dsimp` during preprocessing of well-founded definitions. This fixes regressions when using `if-then-else` without giving a name to the condition, but where the condition is needed for the termination proof, in cases where that subexpression is reachable only by dsimp, but not by simp (e.g. inside a dependent let)

Library

- #5498 makes `BitVec.getElem` the simp normal form in case a proof is available and changes `ext` to return `x[i]` + a hypothesis that proves that we are in-bounds. This aligns `BitVec` further with the API conventions of the Lean standard datatypes.
- #6326 adds `BitVec.(getMsbD, msb)_replicate, replicate_one` theorems, corrects a non-terminal `simp` in `BitVec.getLsbD_replicate` and simplifies the proof of `BitVec.getElem_replicate` using the `cases` tactic.
- #6628 adds SMT-LIB operators to detect overflow `BitVec.(uadd_overflow, sadd_overflow)`, according to the definitions here, and the theorems proving equivalence of such definitions with the `BitVec` library functions (`uaddOverflow_eq`, `saddOverflow_eq`). Support theorems for these proofs are `BitVec.toNat_mod_cancel_of_lt, BitVec.toInt_lt, BitVec.le_toInt, Int.bmod_neg_iff`. The PR also includes a set of tests.

- #6792 adds theorems `BitVec.(getMsbD, msb)_(extractLsb', extractLsb), getMsbD_extractLsb'_eq_getLsbD`.
- #6795 adds theorems `BitVec.(getElem_umod_of_lt, getElem_umod, getLsbD_umod, getMsbD_umod)`. For the defiition of these theorems we rely on `divRec`, excluding the case where `d=0#w`, which is treated separately because there is no infrastructure to reason about this case within `divRec`. In particular, our implementation follows the mathlib standard where division by 0 yields 0, while in SMTLIB this yields `allOnes`.
- #6830 improves some files separation and standardize error messages in UV modules
- #6850 adds some lemmas about the new tree map. These lemmas are about the interactions of `empty`, `isEmpty`, `insert`, `contains`. Some lemmas about the interaction of `contains` with the others will follow in a later PR.
- #6866 adds missing `Hashable` instances for `PUnit` and `PEmpty`.
- #6914 introduces ordered map data structures, namely `DTreeMap`, `TreeMap`, `TreeSet` and their `.Raw` variants, into the standard library. There are still some operations missing that the hash map has. As of now, the operations are unverified, but the corresponding lemmas will follow in subsequent PRs. While the tree map has already been optimized, more micro-optimization will follow as soon as the new code generator is ready.
- #6922 adds `LawfulBEq` instances for `Array` and `Vector`.
- #6948 completes the alignment of `List/Array/Vectors` lemmas for `insertIdx`.
- #6954 verifies the `toList`function for hash maps and dependent hash maps.
- #6958 improves the `Promise` API by considering how dropped promises can lead to never-finished tasks.
- #6966 adds an internal-use-only strict linter for the variable names of `List/Array/Vector` variables, and begins cleaning up.
- #6982 improves some lemmas about monads and monadic operations on Array/Vector, using @Rob23oa's work in https://github.com/leanprover-community/batteries/pull/1109, and adding/generalizing some additional lemmas.
- #7013 makes improvements to the simp set for List/Array/Vector/Option to improve confluence, in preparation for `simp_lc`.
- #7017 renames the simp set `boolToPropSimps` to `bool_to_prop` and `bv_toNat` to `bitvec_to_nat`. I'll be adding more similarly named simp sets.
- #7034 adds `wf_preprocess` theorems for `{List,Array}.{foldlM,foldrM,mapM,filterMapM,flatMapM}`
- #7036 adds some deprecated function aliases to the tree map in order to ease the transition from the RBMap to the tree map.
- #7046 renames `UIntX.mk` to `UIntX.ofBitVec` and adds deprecations.
- #7048 adds the functions `IntX.ofBitVec`.

- #7050 renames the functions `UIntX.val` to `UIntX.toFin`.
- #7051 implements the methods `insertMany`, `ofList`, `ofArray`, `foldr` and `foldrM` on the tree map.
- #7056 adds the `UIntX.ofFin` conversion functions.
- #7057 adds the function `UIntX.ofNatLT`. This is supposed to be a replacement for `UIntX.ofNatCore` and `UIntX.ofNat'`, but for bootstrapping reasons we need this function to exist in stage0 before we can proceed with the renaming and deprecations, so this PR just adds the function.
- #7059 moves away from using `List.get` / `List.get?` / `List.get!` and `Array.get!`, in favour of using the `GetElem` mediated getters. In particular it deprecates `List.get?`, `List.get!` and `Array.get?`. Also adds `Array.back`, taking a proof, matching `List.getLast`.
- #7062 introduces the functions `UIntX.toIntX` as the public API to obtain the `IntX` that is 2's complement equivalent to a given `UIntX`.
- #7063 adds `ISize.toInt8`, `ISize.toInt16`, `Int8.toISize`, `Int16.toISize`.
- #7064 renames `BitVec.ofNatLt` to `BitVec.ofNatLT` and sets up deprecations for the old name.
- #7066 renames `IntX.toNat` to `IntX.toNatClampNeg` (to reduce surprises) and sets up a deprecation.
- #7068 is a follow-up to #7057 and adds a builtin dsimproc for `UIntX.ofNatLT` which it turns out we need in stage0 before we can get the deprecation of `UIntX.ofNatCore` in favor of `UIntX.ofNatLT` off the ground.
- #7070 implements the methods `min`, `max`, `minKey`, `maxKey`, `atIndex`, `getEntryLE`, `getKeyLE` and consorts on the tree map.
- #7071 unifies the existing functions `UIntX.ofNatCore` and `UIntX.ofNat'` under a new name, `UIntX.ofNatLT`.
- #7079 introduces `Fin.toNat` as an alias for `Fin.val`. We add this function for discoverability and consistency reasons. The normal form for proofs remains `Fin.val`, and there is a `simp` lemma rewriting `Fin.toNat` to `Fin.val`.
- #7080 adds the functions `UIntX.ofNatTruncate` (the version for `UInt32` already exists).
- #7081 adds functions `IntX.ofIntLE`, `IntX.ofIntTruncate`, which are analogous to the unsigned counterparts `UIntX.ofNatLT` and `UInt.ofNatTruncate`.
- #7083 adds (value-based, not bitfield-based) conversion functions between `Float`/`Float32` and `IntX`/`UIntX`.
- #7105 completes aligning `Array`/`Vector.extract` lemmas with the lemmas for `List.take` and `List.drop`.
- #7106 completes the alignment of `List`/`Array`/`Vector.finRange` lemmas.

- #7109 implements the `getThenInsertIfNew?` and `partition` functions on the tree map.
- #7114 implements the methods `values` and `valuesArray` on the tree map.
- #7116 implements the `getKey` functions on the tree map. It also fixes the naming of the `entryAtIdx` function on the tree set, which should have been called `atIdx`.
- #7118 implements the functions `modify` and `alter` on the tree map.
- #7128 adds `Repr` and `Hashable` instances for `IntX`.
- #7131 adds `IntX.abs` functions. These are specified by `BitVec.abs`, so they map `IntX.minValue` to `IntX.minValue`, similar to Rust's `i8::abs`. In the future we might also have versions which take values in `UIntX` and/or `Nat`.
- #7137 verifies the various fold and for variants for hashmaps.
- #7151 fixes a memory leak in `IO.FS.createTempFile`
- #7158 strengthens `Int.tdiv_eq_ediv`, by dropping an unnecessary hypothesis, in preparation for further work on `ediv`/`tdiv`/`fdiv` lemmas.
- #7161 adds all missing tree map lemmas about the interactions of the functions `empty`, `isEmpty`, `contains`, `size`, `insert(IfNew)` and `erase`.
- #7162 splits `Int.DivModLemmas` into a `Bootstrap` and `Lemmas` file, where it is possible to use `omega` in `Lemmas`.
- #7163 gives an unconditional theorem expressing `Int.tdiv` in terms of `Int.ediv`, not just for non-negative arguments.
- #7165 provides tree map lemmas about the interaction of `containsThenInsert(IfNew)` with `contains` and `insert(IfNew)`.
- #7167 provides tree map lemmas for the interaction of `get?` with the other operations for which lemmas already exist.
- #7174 adds the first batch of lemmas about iterated conversions between finite types starting with something of type `UIntX`.
- #7199 adds theorems comparing `Int.ediv` with `tdiv` and `fdiv`, for all signs of arguments. (Previously we just had the statements about the cases in which they agree.)
- #7201 adds `Array`/`Vector.left`/`rightpad`. These will not receive any verification theorems; simp just unfolds them to an ++ operation.
- #7205 completes alignment of `List.getLast`/`List.getLast!`/`List.getLast?` lemmas with the corresponding lemmas for Array and Vector.
- #7206 adds theorem `BitVec.toFin_abs`, completing the API for `BitVec.*_abs`.
- #7207 provides lemmas for the tree map functions `get`, `get!` and `getD` in relation to the other operations for which lemmas already exist.
- #7208 aligns lemmas for `List.dropLast` / `Array.pop` / `Vector.pop`.

- #7210 adds the remaining lemmas about iterated conversions between finite types starting with something of type `UIntX`.
- #7214 adds a `ForIn` instance for the `PersistentHashSet` type.
- #7221 provides lemmas about the tree map functions `getKey?`, `getKey`, `getKey!`, `getKeyD` and `insertIfNew` and their interaction with other functions for which lemmas already exist.
- #7222 removes the `simp` attribute from `ReflCmp.compare_self` because it matches arbitrary function applications. Instead, a new `simp` lemma `ReflOrd.compare_self` is introduced, which only matches applications of `compare`.
- #7229 provides lemmas for the tree map function `getThenInsertIfNew?`.
- #7235 adds `Array.replace` and `Vector.replace`, proves the correspondences with `List.replace`, and reproduces the basic API. In order to do so, it fills in some gaps in the `List.findX` APIs.
- #7237 provides proofs that the raw tree map operations are well-formed and refactors the file structure of the tree map, introducing new modules `Std.{DTreeMap,TreeMap,TreeSet}.Raw` and splittting `AdditionalOperations` into separate files for bundled and raw types.
- #7245 adds missing `@[specialize]` annotations to the `alter` and `modify` functions in `Std.Data.DHashMap.Internal.AssocList`, which are used by the corresponding hash map functions.
- #7249 completes alignment of theorems about `List/Array/Vector.any/all`.
- #7255 fixes the definition of `Min (Option α)`. This is a breaking change. This treats none as the least element, so `min none x = min x none = none` for all `x : Option α`. Prior to nightly-2025-02-27, we instead had `min none (some x) = min (some x) none = some x`. Also adds basic lemmas relating `min`, `max`, ≤ and < on `Option`.
- #7259 contains theorems about `IntX` that are required for `bv_decide` and the `IntX` simprocs.
- #7260 provides lemmas about the tree map functions `keys` and `toList` and their interactions with other functions for which lemmas already exist. Moreover, a bug in `foldr` (calling `foldlM` instead of `foldrM`) is fixed.
- #7266 begins the alignment of `Int.ediv/fdiv/tdiv` theorems.
- #7268 implements `Lean.ToExpr` for finite signed integers.
- #7271 changes the order of arguments of the folding function expected by the tree map's `foldr` and `foldrM` functions so that they are consistent with the API of `List`.
- #7273 fixes the statement of a `UIntX` conversion lemma.
- #7277 fixes a bug in Float32.ofInt, which previously returned a Float(64).

Compiler

- #6928 makes extern decls evaluate as ⊤ rather than the default value of ⊥ in the LCNF elimDeadBranches analysis.
- #6930 changes the name generation of specialized LCNF decls so they don't strip macro scopes. This avoids name collisions for specializations created in distinct macro scopes. Since the normal Name.append function checks for the presence of macro scopes, we need to use appendCore.
- #6976 extends the behavior of the `sync` flag for `Task.map/bind` etc. to encompass synchronous execution even when they first have to wait on completion of the first task, drastically lowering the overhead of such tasks. Thus the flag is now equivalent to e.g. .NET's `TaskContinuationOptions.ExecuteSynchronously`.
- #7037 relaxes the minimum required glibc version for Lean and Lean executables to 2.26 on x86-64 Linux
- #7041 marks several LCNF-specific environment extensions as having an asyncMode of .sync rather than the default of .mainOnly, so they work correctly even in async contexts.
- #7086 makes the arity reduction pass in the new code generator match the old one when it comes to the behavior of decls with no used parameters. This is important, because otherwise we might create a top-level decl with no params that contains unreachable code, which would get evaluated unconditionally during initialization. This actually happens when initializing Init.Core built with the new code generator.

Pretty Printing

- #7074 modifies the signature pretty printer to add hover information for parameters in binders. This makes the binders be consistent with the hovers in pi types.

Documentation

- #6886 adds recommended spellings for many notations defined in Lean core, using the `recommended_spelling` command from #6869.
- #6950 adds a style guide and a naming convention for the standard library.
- #6962 improves the doc-string for `List.toArray`.
- #6998 modifies the `Prop` docstring to point out that every proposition is propositionally equal to either `True` or `False`. This will help point users toward seeing that `Prop` is like `Bool`.
- #7026 clarifies the styling of do blocks, and enhanes the naming conventions with information about the `ext` and `mono` name components as well as advice about primed names and naming of simp sets.
- #7111 extends the standard library style guide with guidance on universe variables, notations and Unicode usage, and structure definitions.

Server

- #6329 enables the language server to present multiple disjoint line ranges as being worked on. Even before parallelism lands, we make use of this feature to show post-elaboration tasks such as kernel checking on the first line of a declaration to distinguish them from the final tactic step.
- #6768 adds preliminary support for inlay hints, as well as support for inlay hints that denote the auto-implicits of a function. Hovering over an auto-implicit displays its type and double-clicking the auto-implicit inserts it into the text document.

  Breaking Change: The semantic highlighting request handler is not a pure request handler anymore, but a stateful one. Notably, this means that clients that extend the semantic highlighting of the Lean language server with the `chainLspRequestHandler` function must now use the `chainStatefulLspRequestHandler` function instead.
- #6887 fixes a bug where the goal state selection would sometimes select incomplete incremental snapshots on whitespace, leading to an incorrect "no goals" response. Fixes #6594, a regression that was originally introduced in 4.11.0 by #4727.
- #6959 implements a number of refinements for the auto-implicit inlay hints implemented in #6768. Specifically:
  - In #6768, there was a bug where the inlay hint edit delay could accumulate on successive edits, which meant that it could sometimes take much longer for inlay hints to show up. implements the basic infrastructure for request cancellation and implements request cancellation for semantic tokens and inlay hints to resolve the issue. With this edit delay bug fixed, it made more sense to increase the edit delay slightly from 2000ms to 3000ms.
  - In #6768, we applied the edit delay to every single inlay hint request in order to reduce the amount of inlay hint flickering. This meant that the edit delay also had a significant effect on how far inlay hints would lag behind the file progress bar. adjusts the edit delay logic so that it only affects requests sent directly after a corresponding `didChange` notification. Once the edit delay is used up, all further semantic token requests are responded to without delay, so that the only latency that affects how far the inlay hints lag behind the progress bar is how often we emit refresh requests and how long VS Code takes to respond to them.
  - For inlay hints, refresh requests are now emitted 500ms after a response to an inlay hint request, not 2000ms, which means that after the edit delay, inlay hints should only lag behind the progress bar by about up to 500ms. This is justifiable for inlay hints because the response should be much smaller than e.g. is the case for semantic tokens.
  - In #6768, 'Restart File' did not prompt a refresh, but it does now.
  - VS Code does not immediately remove old inlay hints from the document when they are applied. In #6768, this meant that inlay hints would linger around for a bit once applied. To mitigate this issue, this PR adjusts the inlay hint edit delay logic to identify edits sent from the client as being inlay hint applications, and sets the edit delay to 0ms for the inlay hint requests following it. This means that inlay hints are now applied immediately.

  - In #6768, hovering over single-letter auto-implicit inlay hints was a bit finicky because VS Code uses the regular cursor icon on inlay hints, not the thin text cursor icon, which means that it is easy to put the cursor in the wrong spot. We now add the separation character (or {) preceding an auto-implicit to the hover range as well, which makes hovering over inlay hints much smoother.

- #6978 fixes a bug where both the inlay hint change invalidation logic and the inlay hint edit delay logic were broken in untitled files. Thanks to @Julian for spotting this!
- #7054 adds language server support for request cancellation to the following expensive requests: Code actions, auto-completion, document symbols, folding ranges and semantic highlighting. This means that when the client informs the language server that a request is stale (e.g. because it belongs to a previous state of the document), the language server will now prematurely cancel the computation of the response in order to reduce the CPU load for requests that will be discarded by the client anyways.
- #7087 ensures that all tasks in the language server either use dedicated tasks or reuse an existing thread from the thread pool. This ensures that elaboration tasks cannot prevent language server tasks from being scheduled. This is especially important with parallelism right around the corner and elaboration becoming more likely to starve the language server of computation, which could drive up language server latencies significantly on machines with few cores.
- #7112 adds a tooltip describing what the auto-implicit inlay hints denote, as well as auto-implicit inlay hints for instances.
- #7134 significantly improves the performance of auto-completion by optimizing individual requests by a factor of ~2 and by giving language clients like VS Code the opportunity to reuse the state of previous completion requests, thus greatly reducing the latency for the auto-completion list to update when adding more characters to an identifier.
- #7143 makes the server consistently not report newlines between trace nodes to the info view, enabling it to render them on dedicates lines without extraneous spacing between them in all circumstances.
- #7149 adds a fast path to the inlay hint request that makes it re-use already computed inlay hints from previous requests instead of re-computing them. This is necessary because for some reason VS Code emits an inlay hint request for every line you scroll, so we need to be able to respond to these requests against the same document state quickly. Otherwise, every single scrolled line would result in a request that can take a few dozen ms to be responded to in long files, putting unnecessary pressure on the CPU. It also filters the result set by the inlay hints that have been requested.
- #7153 changes the server to run `lake setup-file` on Lake configuration files (e.g., `lakefile.lean`).
- #7175 fixes an `Elab.async` regression where elaboration tasks are cancelled on document edit even though their result may be reused in the new document version, reporting an incomplete result.

Lake

- #6829 changes the error message for Lake configuration failure to reflect that issues do not always arise from an invalid lakefile, but sometimes arise from other issues like network errors. The new error message encompasses all of these possibilities.
- #6929 passes the shared library of the previous stage's Lake as a plugin to the next stage's Lake in the CMake build. This enables Lake to use its own builtin elaborators / initializers at build time.
- #7001 adds support for plugins to Lake. Precompiled modules are now loaded as plugins rather than via `--load-dynlib`.
- #7024 documents how to use Elan's + option with `lake new|init`. It also provides an more informative error message if a + option leaks into Lake (e.g., if a user provides the option to a Lake run without Elan).
- #7157 changes `lake setup-file` to now use Lake as a plugin for files which import Lake (or one of its submodules). Thus, the server will now load Lake as a plugin when editing a Lake configuration written in Lean. This further enables the use of builtin language extensions in Lake.
- #7171 changes the Lake DSL to use builtin elaborators, macros, and initializers.
- #7182 makes `lake setup-file` succeed on an invalid Lean configuration file.
- #7209 fixes broken Lake tests on Windows' new MSYS2. As of MSYS2 0.0.20250221, OSTYPE is now reported as `cygwin` instead of `msys`, which must be accounted for in a few Lake tests.
- #7211 changes the job monitor to perform run job computation itself as a separate job. Now progress will be reported eagerly, even before all outstanding jobs have been discovered. Thus, the total job number reported can now grow while jobs are still being computed (e.g., the Y in `[X/Y[` may increase).
- #7233 uses the Lake plugin when Lake is built with Lake via USE_LAKE.
- #7291 changes the Lake job monitor to display the last (i.e., newest) running/unfinished job rather than the first. This avoids the monitor focusing too long on any one job (e.g., "Running job computation").
- #7399 reverts the new builtin initializers, elaborators, and macros in Lake back to non-builtin.
- #7608 removes the use of the Lake plugin in the Lake build and in configuration files.

Other

- #7129 optimizes the performance of the unused variables linter in the case of a definition with a huge `Expr` representation
- #7173 introduces a trace node for each deriving handlers invocation for the benefit of `trace.profiler`
- #7184 adds support for LEAN_BACKTRACE on macOS. This previously only worked with glibc, but it can not be enabled for all Unix-like systems, since e.g. Musl does not support it.

- #7190 makes the stage2 Leanc build use the stage2 oleans rather than stage1 oleans. This was happening because Leanc's own OLEAN_OUT is at the build root rather than the lib/lean subdirectory, so when the build added this OLEAN_OUT to LEAN_PATH no oleans were found there and the search fell back to the stage1 installation location.

Lean 4.17.0 (2025-03-03)

For this release, 319 changes landed. In addition to the 168 feature additions and 57 fixes listed below there were 12 refactoring changes, 13 documentation improvements and 56 chores.

Highlights

The Lean v4.17 release brings a range of new features, performance improvements, and bug fixes. Notable user-visible updates include:

- #6368 implements executing kernel checking in parallel to elaboration, which is a prerequisite for parallelizing elaboration itself.
- #6711 adds support for `UIntX` and `USize` in `bv_decide` by adding a preprocessor that turns them into `BitVec` of their corresponding size.
- #6505 implements a basic async framework as well as asynchronously running timers using libuv.
- improvements to documentation with `docgen`, which now links dot notations (#6703), coerced functions (#6729), and tokens (#6730).
- extensive library development, in particular, expanding verification APIs of `BitVec`, making APIs of `List` / `Array` / `Vector` consistent, and adding lemmas describing the behavior of `UInt`.
- #6597 fixes the indentation of nested traces nodes in the info view.

New Language Features

- **Partial Fixpoint**

  #6355 adds the ability to define possibly non-terminating functions and still be able to reason about them equationally, as long as they are tail-recursive, or operate within certain monads such as `Option`

  Typical examples:

```
def ack : (n m : Nat) → Option Nat
  | 0,   y   => some (y+1)
  | x+1, 0   => ack x 1
  | x+1, y+1 => do ack x (← ack (x+1) y)
partial_fixpoint

def whileSome (f : α → Option α) (x : α) : α :=
  match f x with
  | none => x
```

```
  | some x' => whileSome f x'
partial_fixpoint

def computeLfp {α : Type u} [DecidableEq α] (f : α → α) (x : α) :
α :=
  let next := f x
  if x ≠ next then
    computeLfp f next
  else
    x
partial_fixpoint
```

See the reference manual for more details.

- #6905 adds a first draft of the `try?` interactive tactic, which tries various tactics, including induction:

```
@[simp] def revAppend : List Nat → List Nat → List Nat
| [],    ys => ys
| x::xs, ys => revAppend xs (x::ys)

example : (revAppend xs ys).length = xs.length + ys.length := by
  try?
  /-
  Try these:
  • · induction xs, ys using revAppend.induct
      · simp
      · simp +arith [*]
  • · induction xs, ys using revAppend.induct
      · simp only [revAppend, List.length_nil, Nat.zero_add]
      · simp +arith only [revAppend, List.length_cons, *]
  -/
```

- **`induction` with zero alternatives**

  #6486 modifies the `induction`/`cases` syntax so that the `with` clause does not need to be followed by any alternatives. This improves friendliness of these tactics, since this lets them surface the names of the missing alternatives:

```
example (n : Nat) : True := by
  induction n with
/-            ~~~~
alternative 'zero' has not been provided
alternative 'succ' has not been provided
-/
```

- **`simp?` and `dsimp?` tactics in conversion mode**

  #6593 adds support for the `simp?` and `dsimp?` tactics in conversion mode.

- **`fun_cases`**

#6261 adds `foo.fun_cases`, an automatically generated theorem that splits the goal according to the branching structure of `foo`, much like the Functional Induction Principle, but for all functions (not just recursive ones), and without providing inductive hypotheses.

New CLI Features

- #6427 adds the Lean CLI option `--src-deps` which parallels `--deps`. It parses the Lean code's header and prints out the paths to the (transitively) imported modules' source files (deduced from `LEAN_SRC_PATH`).
- #6323 adds a new Lake CLI command, `lake query`, that both builds targets and outputs their results. It can produce raw text or JSON -formatted output (with `--json` / `-J`).

Breaking Changes

- #6602 allows the dot ident notation to resolve to the current definition, or to any of the other definitions in the same mutual block. Existing code that uses dot ident notation may need to have `nonrec` added if the ident has the same name as the definition.
- Introduction of the `zetaUnused` simp and reduction option (#6755) is a breaking change in rare cases: the `split` tactic no longer removes unused `let` and `have` expressions as a side-effect. `dsimp only` can be used to remove unused `have` and `let` expressions.

This highlights section was contributed by Violetta Sim.

Language

- #5145 splits the environment used by the kernel from that used by the elaborator, providing the foundation for tracking of asynchronously elaborated declarations, which will exist as a concept only in the latter.
- #6261 adds `foo.fun_cases`, an automatically generated theorem that splits the goal according to the branching structure of `foo`, much like the Functional Induction Principle, but for all functions (not just recursive ones), and without providing inductive hypotheses.
- #6355 adds the ability to define possibly non-terminating functions and still be able to reason about them equationally, as long as they are tail-recursive or monadic.
- #6368 implements executing kernel checking in parallel to elaboration, which is a prerequisite for parallelizing elaboration itself.
- #6427 adds the Lean CLI option `--src-deps` which parallels `--deps`. It parses the Lean code's header and prints out the paths to the (transitively) imported modules' source files (deduced from `LEAN_SRC_PATH`).
- #6486 modifies the `induction`/`cases` syntax so that the `with` clause does not need to be followed by any alternatives. This improves friendliness of these tactics, since this lets them surface the names of the missing alternatives:

```
example (n : Nat) : True := by
  induction n with
/-            ~~~~
```

```
alternative 'zero' has not been provided
alternative 'succ' has not been provided
-/
```

- #6505 implements a basic async framework as well as asynchronously running timers using libuv.
- #6516 enhances the `cases` tactic used in the `grind` tactic and ensures that it can be applied to arbitrary expressions.
- #6521 adds support for activating relevant `match`-equations as E-matching theorems. It uses the `match`-equation lhs as the pattern.
- #6528 adds a missing propagation rule to the (WIP) `grind` tactic.
- #6529 adds support for `let`-declarations to the (WIP) `grind` tactic.
- #6530 fixes nondeterministic failures in the (WIP) `grind` tactic.
- #6531 fixes the support for `let_fun` in `grind`.
- #6533 adds support to E-matching offset patterns. For example, we want to be able to E-match the pattern `f (#0 + 1)` with term `f (a + 2)`.
- #6534 ensures that users can utilize projections in E-matching patterns within the `grind` tactic.
- #6536 fixes different thresholds for controlling E-matching in the `grind` tactic.
- #6538 ensures patterns provided by users are normalized. See new test to understand why this is needed.
- #6539 introduces the `[grind_eq]` attribute, designed to annotate equational theorems and functions for heuristic instantiations in the `grind` tactic. When applied to an equational theorem, the `[grind_eq]` attribute instructs the `grind` tactic to automatically use the annotated theorem to instantiate patterns during proof search. If applied to a function, it marks all equational theorems associated with that function.
- #6543 adds additional tests for `grind`, demonstrating that we can automate some manual proofs from Mathlib's basic category theory library, with less reliance on Mathlib's `@[reassoc]` trick.
- #6545 introduces the parametric attribute `[grind]` for annotating theorems and definitions. It also replaces `[grind_eq]` with `[grind =]`. For definitions, `[grind]` is equivalent to `[grind =]`.
- #6556 adds propagators for implication to the `grind` tactic. It also disables the normalization rule: `(p → q) = (¬ p ∨ q)`
- #6559 adds a basic case-splitting strategy for the `grind` tactic. We still need to add support for user customization.
- #6565 fixes the location of the error emitted when the `rintro` and `intro` tactics cannot introduce the requested number of binders.
- #6566 adds support for erasing the `[grind]` attribute used to mark theorems for heuristic instantiation in the `grind` tactic.

- #6567 adds support for erasing the `[grind]` attribute used to mark theorems for heuristic instantiation in the `grind` tactic.
- #6568 adds basic support for cast-like operators to the grind tactic. Example:

```
example (α : Type) (β : Type) (a₁ a₂ : α) (b₁ b₂ : β)
        (h₁ : α = β)
        (h₂ : h₁ ▸ a₁ = b₁)
        (h₃ : a₁ = a₂)
        (h₄ : b₁ = b₂)
        : HEq a₂ b₂ := by
  grind
```

- #6569 adds support for case splitting on `match`-expressions in `grind`. We still need to add support for resolving the antecedents of `match`-conditional equations.
- #6575 ensures tactics are evaluated incrementally in the body of `classical`.
- #6578 fixes and improves the propagator for forall-expressions in the `grind` tactic.
- #6581 adds the following configuration options to `Grind.Config`: `splitIte`, `splitMatch`, and `splitIndPred`.
- #6582 adds support for creating local E-matching theorems for universal propositions known to be true. It allows `grind` to automatically solve examples such as:
- #6584 adds helper theorems to implement offset constraints in grind.
- #6585 fixes a bug in the `grind` canonicalizer.
- #6588 improves the `grind` canonicalizer diagnostics.
- #6593 adds support for the `simp?` and `dsimp?` tactics in conversion mode.
- #6595 improves the theorems used to justify the steps performed by the inequality offset module. See new test for examples of how they are going to be used.
- #6600 removes functions from compiling decls from Environment, and moves all users to functions on CoreM. This is required for supporting the new code generator, since its implementation uses CoreM.
- #6602 allows the dot ident notation to resolve to the current definition, or to any of the other definitions in the same mutual block. Existing code that uses dot ident notation may need to have `nonrec` added if the ident has the same name as the definition.
- #6603 implements support for offset constraints in the `grind` tactic. Several features are still missing, such as constraint propagation and support for offset equalities, but `grind` can already solve examples like the following:
- #6606 fixes a bug in the pattern selection in the `grind`.
- #6607 adds support for case-splitting on `<->` (and `@Eq Prop`) in the `grind` tactic.
- #6608 fixes a bug in the `simp_arith` tactic. See new test.
- #6609 improves the case-split heuristic used in grind, prioritizing case-splits with fewer cases.

- #6610 fixes a bug in the `grind` core module responsible for merging equivalence classes and propagating constraints.
- #6611 fixes one of the sanity check tests used in `grind`.
- #6613 improves the case split heuristic used in the `grind` tactic, ensuring it now avoids unnecessary case-splits on `Iff`.
- #6614 improves the usability of the `[grind =]` attribute by automatically handling forbidden pattern symbols. For example, consider the following theorem tagged with this attribute:

  ```
  getLast?_eq_some_iff {xs : List α} {a : α} : xs.getLast? = some a
  ↔ ∃ ys, xs = ys ++ [a]
  ```

  Here, the selected pattern is `xs.getLast? = some a`, but `Eq` is a forbidden pattern symbol. Instead of producing an error, this function converts the pattern into a multi-pattern, allowing the attribute to be used conveniently.
- #6615 adds two auxiliary functions `mkEqTrueCore` and `mkOfEqTrueCore` that avoid redundant proof terms in proofs produced by `grind`.
- #6618 implements exhaustive offset constraint propagation in the `grind` tactic. This enhancement minimizes the number of case splits performed by `grind`. For instance, it can solve the following example without performing any case splits:
- #6633 improves the failure message produced by the `grind` tactic. We now include information about asserted facts, propositions that are known to be true and false, and equivalence classes.
- #6636 implements model construction for offset constraints in the `grind` tactic.
- #6639 puts the `bv_normalize` simp set into simp_nf and splits up the bv_normalize implementation across multiple files in preparation for upcoming changes.
- #6641 implements several optimisation tricks from Bitwuzla's preprocessing passes into the Lean equivalent in `bv_decide`. Note that these changes are mostly geared towards large proof states as for example seen in SMT-Lib.
- #6645 implements support for offset equality constraints in the `grind` tactic and exhaustive equality propagation for them. The `grind` tactic can now solve problems such as the following:
- #6648 adds support for numerals, lower & upper bounds to the offset constraint module in the `grind` tactic. `grind` can now solve examples such as:

  ```
  example (f : Nat → Nat) :
          f 2 = a →
          b ≤ 1 → b ≥ 1 →
          c = b + 1 →
          f c = a := by
    grind
  ```

  In the example above, the literal 2 and the lower&upper bounds, `b ≤ 1` and `b ≥ 1`, are now processed by offset constraint module.

- #6649 fixes a bug in the term canonicalizer used in the `grind` tactic.
- #6652 adds the `int_toBitVec` simp set to convert UIntX and later IntX propositions to BitVec ones. This will be used as a preprocessor for `bv_decide` to provide UIntX/IntX `bv_decide` support.
- #6653 improves the E-matching pattern selection heuristics in the `grind` tactic. They now take into account type predicates and transformers.
- #6654 improves the support for partial applications in the E-matching procedure used in `grind`.
- #6656 improves the diagnostic information provided in `grind` failure states. We now include the list of issues found during the search, and all search thresholds that have been reached. also improves its formatting.
- #6657 improves the `grind` search procedure, and adds the new configuration option: `failures`.
- #6658 ensures that `grind` avoids case-splitting on terms congruent to those that have already been case-split.
- #6659 fixes a bug in the `grind` term preprocessor. It was abstracting nested proofs **before** reducible constants were unfolded.
- #6662 improves the canonicalizer used in the `grind` tactic and the diagnostics it produces. It also adds a new configuration option, `canonHeartbeats`, to address (some of) the issues. Here is an example illustrating the new diagnostics, where we intentionally create a problem by using a very small number of heartbeats.
- #6663 implements a basic equality resolution procedure for the `grind` tactic.
- #6669 adds a workaround for the discrepancy between Terminal/Emacs and VS Code when displaying info trees.
- #6675 adds `simp`-like parameters to `grind`, and `grind only` similar to `simp only`.
- #6679 changes the identifier parser to allow for the ¡ unicode character which was forgotten as it lives by itself in a codeblock with coptic characters.
- #6682 adds support for extensionality theorems (using the `[ext]` attribute) to the `grind` tactic. Users can disable this functionality using `grind -ext`. Below are examples that demonstrate problems now solvable by `grind`.
- #6685 fixes the issue that `#check_failure`'s output is warning
- #6686 fixes parameter processing, initialization, and attribute handling issues in the `grind` tactic.
- #6691 introduces the central API for making parallel changes to the environment
- #6692 removes the `[grind_norm]` attribute. The normalization theorems used by `grind` are now fixed and cannot be modified by users. We use normalization theorems to ensure the built-in procedures receive term wish expected "shapes". We use it for types that have built-in support in grind. Users could misuse this feature as a simplification rule. For example, consider the following example:

- #6700 adds support for beta reduction in the `grind` tactic. `grind` can now solve goals such as

  ```
  example (f : Nat → Nat) : f = (fun x : Nat => x + 5) → f 2 > 5 :=
  by
    grind
  ```

- #6702 adds support for equality backward reasoning to `grind`. We can illustrate the new feature with the following example. Suppose we have a theorem:

  ```
  theorem inv_eq {a b : α} (w : a * b = 1) : inv a = b
  ```

  and we want to instantiate the theorem whenever we are tying to prove `inv t = s` for some terms `t` and `s` The attribute `[grind ←]` is not applicable in this case because, by default, = is not eligible for E-matching. The new attribute `[grind ←=]` instructs `grind` to use the equality and consider disequalities in the `grind` proof state as candidates for E-matching.

- #6705 adds the attributes `[grind cases]` and `[grind cases eager]` for controlling case splitting in `grind`. They will replace the `[grind_cases]` and the configuration option `splitIndPred`.

- #6711 adds support for `UIntX` and `USize` in `bv_decide` by adding a preprocessor that turns them into `BitVec` of their corresponding size.

- #6717 introduces a new feature that allows users to specify which inductive datatypes the `grind` tactic should perform case splits on. The configuration option `splitIndPred` is now set to `false` by default. The attribute `[grind cases]` is used to mark inductive datatypes and predicates that `grind` may case split on during the search. Additionally, the attribute `[grind cases eager]` can be used to mark datatypes and predicates for case splitting both during pre-processing and the search.

- #6718 adds BitVec lemmas required to cancel multiplicative negatives, and plumb support through to `bv_normalize` to make use of this result in the normalized twos-complement form.

- #6719 fixes a bug in the equational theorem generator for `match`-expressions. See new test for an example.

- #6724 adds support for `bv_decide` to automatically split up non-recursive structures that contain information about supported types. It can be controlled using the new `structures` field in the `bv_decide` config.

- #6733 adds better support for overlapping `match` patterns in `grind`. `grind` can now solve examples such as

  ```
  inductive S where
    | mk1 (n : Nat)
    | mk2 (n : Nat) (s : S)
    | mk3 (n : Bool)
    | mk4 (s1 s2 : S)

  def f (x y : S) :=
    match x, y with
    | .mk1 _, _ => 2
  ```

```
  | _, .mk2 1 (.mk4 _ _) => 3
  | .mk3 _, _ => 4
  | _, _ => 5

example : b = .mk2 y1 y2 → y1 = 2 → a = .mk4 y3 y4 → f a b = 5 :=
by
  unfold f
  grind (splits := 0)
```

- #6735 adds support for case splitting on `match`-expressions with overlapping patterns to the `grind` tactic. `grind` can now solve examples such as:

```
inductive S where
  | mk1 (n : Nat)
  | mk2 (n : Nat) (s : S)
  | mk3 (n : Bool)
  | mk4 (s1 s2 : S)

def g (x y : S) :=
  match x, y with
  | .mk1 a, _ => a + 2
  | _, .mk2 1 (.mk4 _ _) => 3
  | .mk3 _, .mk4 _ _ => 4
  | _, _ => 5

example : g a b > 1 := by
  grind [g.eq_def]
```

- #6736 ensures the canonicalizer used in `grind` does not waste time checking whether terms with different types are definitionally equal.
- #6737 ensures that the branches of an `if-then-else` term are internalized only after establishing the truth value of the condition. This change makes its behavior consistent with the `match`-expression and dependent `if-then-else` behavior in `grind`. This feature is particularly important for recursive functions defined by well-founded recursion and `if-then-else`. Without lazy `if-then-else` branch internalization, the equation theorem for the recursive function would unfold until reaching the generation depth threshold, and before performing any case analysis. See new tests for an example.
- #6739 adds a fast path for bitblasting multiplication with constants in `bv_decide`.
- #6740 extends `bv_decide`'s structure reasoning support for also reasoning about equalities of supported structures.
- #6745 supports rewriting `ushiftRight` in terms of `extractLsb'`. This is the companion PR to #6743 which adds the similar lemmas about `shiftLeft`.
- #6746 ensures that conditional equation theorems for function definitions are handled correctly in `grind`. We use the same infrastructure built for `match`-expression equations. Recall that in both cases, these theorems are conditional when there are overlapping patterns.
- #6748 fixes a few bugs in the `grind` tactic: missing issues, bad error messages, incorrect threshold in the canonicalizer, and bug in the ground pattern internalizer.

- #6750 adds support for fixing type mismatches using `cast` while instantiating quantifiers in the E-matching module used by the grind tactic.
- #6754 adds the `+zetaUnused` option.
- #6755 implements the `zetaUnused` simp and reduction option (added in #6754).
- #6761 fixes issues in `grind` when processing `match`-expressions with indexed families.
- #6773 fixes a typo that prevented `Nat.reduceAnd` from working correctly.
- #6777 fixes a bug in the internalization of offset terms in the `grind` tactic. For example, `grind` was failing to solve the following example because of this bug.

```
example (f : Nat → Nat) : f (a + 1) = 1 → a = 0 → f 1 = 1 := by
  grind
```

- #6778 fixes the assignment produced by `grind` to satisfy the offset constraints in a goal.
- #6779 improves the support for `match`-expressions in the `grind` tactic.
- #6781 fixes the support for case splitting on data in the `grind` tactic. The following example works now:

```
inductive C where
  | a | b | c

def f : C → Nat
  | .a => 2
  | .b => 3
  | .c => 4

example : f x > 1 := by
  grind [
      f, -- instructs `grind` to use `f`-equation theorems,
      C -- instructs `grind` to case-split on free variables of
type `C`
  ]
```

- #6783 adds support for closing goals using `match`-expression conditions that are known to be true in the `grind` tactic state. `grind` can now solve goals such as:

```
def f : List Nat → List Nat → Nat
  | _, 1 :: _ :: _ => 1
  | _, _ :: _ => 2
  | _, _  => 0

example : z = a :: as → y = z → f x y > 0
```

- #6785 adds infrastructure for the `grind?` tactic. It also adds the new modifier `usr` which allows users to write `grind only [use thmName]` to instruct `grind` to only use theorem `thmName`, but using the patterns specified with the command `grind_pattern`.
- #6788 teaches bv_normalize that `!(x < x)` and `!(x < 0)`.
- #6790 fixes an issue with the generation of equational theorems from `partial_fixpoint` when case-splitting is necessary. Fixes #6786.

- #6791 fixes #6789 by ensuring metadata generated for inaccessible variables in pattern-matches is consumed in `casesOnStuckLHS` accordingly.
- #6801 fixes a bug in the exhaustive offset constraint propagation module used in `grind`.
- #6810 implements a basic `grind?` tactic companion for `grind`. We will add more bells and whistles later.
- #6822 adds a few builtin case-splits for `grind`. They are similar to builtin `simp` theorems. They reduce the noise in the tactics produced by `grind?`.
- #6824 introduces the auxiliary command `%reset_grind_attrs` for debugging purposes. It is particularly useful for writing self-contained tests.
- #6834 adds "performance" counters (e.g., number of instances per theorem) to `grind`. The counters are always reported on failures, and on successes when `set_option diagnostics true`.
- #6839 ensures `grind` can use constructors and axioms for heuristic instantiation based on E-matching. It also allows patterns without pattern variables for theorems such as `theorem evenz : Even 0`.
- #6851 makes `bv_normalize` rewrite shifts by `BitVec` constants to shifts by `Nat` constants. This is part of the greater effort in providing good support for constant shift simplification in `bv_normalize`.
- #6852 allows environment extensions to opt into access modes that do not block on the entire environment up to this point as a necessary prerequisite for parallel proof elaboration.
- #6854 adds a convenience for inductive predicates in `grind`. Now, give an inductive predicate C, `grind [C]` marks C terms as case-split candidates **and** C constructors as E-matching theorems. Here is an example:

  ```
  example {B S T s t} (hcond : B s) : (ifThenElse B S T, s) ==> t →
  (S, s) ==> t := by
    grind [BigStep]
  ```

  Users can still use `grind [cases BigStep]` to only mark C as a case split candidate.
- #6858 adds new propagation rules for `decide` and equality in `grind`. It also adds new tests and cleans old ones
- #6861 adds propagation rules for `Bool.and`, `Bool.or`, and `Bool.not` to the `grind` tactic.
- #6870 adds two new normalization steps in `grind` that reduces `a != b` and `a == b` to `decide (¬ a = b)` and `decide (a = b)`, respectively.
- #6879 fixes a bug in `mkMatchCondProf?` used by the `grind` tactic. This bug was introducing a failure in the test `grind_constProp.lean`.
- #6880 improves the E-matching pattern selection heuristic used in `grind`.
- #6881 improves the `grind` error message by including a trace of the terms on which `grind` applied `cases`-like operations.
- #6882 ensures `grind` auxiliary gadgets are "hidden" in error and diagnostic messages.

- #6888 adds the `[grind intro]` attribute. It instructs `grind` to mark the introduction rules of an inductive predicate as E-matching theorems.
- #6889 inlines a few functions in the `bv_decide` circuit cache.
- #6892 fixes a bug in the pattern selection heuristic used in `grind`. It was unfolding definitions/abstractions that were not supposed to be unfolded. See `grind_constProp.lean` for examples affected by this bug.
- #6895 fixes a few `grind` issues exposed by the `grind_constProp.lean` test.
    - Support for equational theorem hypotheses created before invoking `grind`. Example: applying an induction principle.s
    - Support of `Unit`-like types.
    - Missing recursion depth checks.
- #6897 adds the new attributes `[grind =>]` and `[grind <=]` for controlling pattern selection and minimizing the number of places where we have to use verbose `grind_pattern` command. It also fixes a bug in the new pattern selection procedure, and improves the automatic pattern selection for local lemmas.
- #6904 adds the `grind` configuration option `verbose`. For example, `grind -verbose` disables all diagnostics. We are going to use this flag to implement `try?`.
- #6905 adds the `try?` tactic; see above

Library

- #6177 implements `BitVec.*_fill`.
- #6211 verifies the `insertMany` method on `HashMap`s for the special case of inserting lists.
- #6346 completes the toNat/Int/Fin family for `shiftLeft`.
- #6347 adds `BitVec.toNat_rotateLeft` and `BitVec.toNat_rotateLeft`.
- #6402 adds a `toFin` and `msb` lemma for unsigned bitvector division. We *don't* have `toInt_udiv`, since the only truly general statement we can make does no better than unfolding the definition, and it's not uncontroversially clear how to unfold `toInt` (see `toInt_eq_msb_cond`/`toInt_eq_toNat_cond`/`toInt_eq_toNat_bmod` for a few options currently provided). Instead, we do have `toInt_udiv_of_msb` that's able to provide a more meaningful rewrite given an extra side-condition (that `x.msb = false`).
- #6404 adds a `toFin` and `msb` lemma for unsigned bitvector modulus. Similar to #6402, we don't provide a general `toInt_umod` lemmas, but instead choose to provide more specialized rewrites, with extra side-conditions.
- #6431 fixes the `Repr` instance of the `Timestamp` type and changes the `PlainTime` type so that it always represents a clock time that may be a leap second.
- #6476 defines `reverse` for bitvectors and implements a first subset of theorems (`getLsbD_reverse, getMsbD_reverse, reverse_append, reverse_replicate, reverse_cast, msb_reverse`). We also include some

necessary related theorems (`cons_append, cons_append_append, append_assoc, replicate_append_self, replicate_succ'`) and deprecate theorems`replicate_zero_eq` and `replicate_succ_eq`.

- #6494 proves the basic theorems about the functions `Int.bdiv` and `Int.bmod`.
- #6507 adds the subtraction equivalents for `Int.emod_add_emod` (`(a % n + b) % n = (a + b) % n`) and `Int.add_emod_emod` (`(a + b % n) % n = (a + b) % n`). These are marked @[simp] like their addition equivalents.
- #6524 upstreams some remaining `List.Perm` lemmas from Batteries.
- #6546 continues aligning `Array` and `Vector` lemmas with `List`, working on `fold` and `map` operations.
- #6563 implements `Std.Net.Addr` which contains structures around IP and socket addresses.
- #6573 replaces the existing implementations of `(D)HashMap.alter` and `(D)HashMap.modify` with primitive, more efficient ones and in particular provides proofs that they yield well-formed hash maps (`WF` typeclass).
- #6586 continues aligning `List/Array/Vector` lemmas, finishing up lemmas about `map`.
- #6587 adds decidable instances for the `LE` and `LT` instances for the `Offset` types defined in `Std.Time`.
- #6589 continues aligning `List/Array` lemmas, finishing `filter` and `filterMap`.
- #6591 adds less-than and less-than-or-equal-to relations to `UInt32`, consistent with the other `UIntN` types.
- #6612 adds lemmas about `Array.append`, improving alignment with the `List` API.
- #6617 completes alignment of `List/Array/Vector` append lemmas.
- #6620 adds lemmas about HashMap.alter and .modify. These lemmas describe the interaction of alter and modify with the read methods of the HashMap. The additions affect the HashMap, the DHashMap and their respective raw versions. Moreover, the raw versions of alter and modify are defined.
- #6625 adds lemmas describing the behavior of `UIntX.toBitVec` on `UIntX` operations.
- #6630 adds theorems `Nat.[shiftLeft_or_distrib`, shiftLeft_xor_distrib`, shiftLeft_and_distrib`, `testBit_mul_two_pow`, `bitwise_mul_two_pow`, `shiftLeft_bitwise_distrib]`, to prove `Nat.shiftLeft_or_distrib` by emulating the proof strategy of `shiftRight_and_distrib`.
- #6640 completes aligning `List/Array/Vector` lemmas about `flatten`. `Vector.flatten` was previously missing, and has been added (for rectangular sizes only). A small number of missing `Option` lemmas were also need to get the proofs to go through.
- #6660 defines `Vector.flatMap`, changes the order of arguments in `List.flatMap` for consistency, and aligns the lemmas for `List/Array/Vector` `flatMap`.

- #6661 adds array indexing lemmas for `Vector.flatMap`. (These were not available for `List` and `Array` due to variable lengths.)
- #6667 aligns `List.replicate`/`Array.mkArray`/`Vector.mkVector` lemmas.
- #6668 fixes negative timestamps and `PlainDateTimes` before 1970.
- #6674 adds theorems `BitVec.[getMsbD_mul, getElem_udiv, getLsbD_udiv, getMsbD_udiv]`
- #6695 aligns `List/Array/Vector.reverse` lemmas.
- #6697 changes the arguments of `List/Array.mapFinIdx` from `(f : Fin as.size → α → β)` to `(f : (i : Nat) → α → (h : i < as.size) → β)`, in line with the API design elsewhere for `List/Array`.
- #6701 completes aligning `mapIdx` and `mapFinIdx` across `List/Array/Vector`.
- #6707 completes aligning lemmas for `List` / `Array` / `Vector` about `foldl`, `foldr`, and their monadic versions.
- #6708 deprecates `List.iota`, which we make no essential use of. `iota n` can be replaced with `(range' 1 n).reverse`. The verification lemmas for `range'` already have better coverage than those for `iota`. Any downstream projects using it (I am not aware of any) are encouraged to adopt it.
- #6712 aligns `List/Array/Vector` theorems for `countP` and `count`.
- #6723 completes the alignment of {List/Array/Vector}.{attach,attachWith,pmap} lemmas. I had to fill in a number of gaps in the List API.
- #6728 removes theorems `Nat.mul_one` to simplify a rewrite in the proof of `BitVec.getMsbD_rotateLeft_of_lt`
- #6742 adds the lemmas that show what happens when multiplying by `twoPow` to an arbitrary term, as well to another `twoPow`.
- #6743 adds rewrites that normalizes left shifts by extracting bits and concatenating zeroes. If the shift amount is larger than the bit-width, then the resulting bitvector is zero.
- #6747 adds the ability to push `BitVec.extractLsb` and `BitVec.extractLsb'` with bitwise operations. This is useful for constant-folding extracts.
- #6767 adds lemmas to rewrite `BitVec.shiftLeft,shiftRight,sshiftRight'` by a `BitVec.ofNat` into a shift-by-natural number. This will be used to canonicalize shifts by constant bitvectors into shift by constant numbers, which have further rewrites on them if the number is a power of two.
- #6799 adds a number of simple comparison lemmas to the top/bottom element for BitVec. Then they are applied to teach bv_normalize that `(a<1) = (a==0)` and to remove an intermediate proof that is no longer necessary along the way.
- #6800 uniformizes the naming of `enum`/`enumFrom` (on `List`) and `zipWithIndex` (on `Array` on `Vector`), replacing all with `zipIdx`. At the same time, we generalize to add an optional `Nat` parameter for the initial value of the index (which previously existed, only for `List`, as the separate function `enumFrom`).

- #6808 adds simp lemmas replacing `BitVec.setWidth'` with `setWidth`, and conditionally simplifying `setWidth v (setWidth w v)`.
- #6818 adds a BitVec lemma that `(x >> x) = 0` and plumbs it through to bv_normalize. I also move some theorems I found useful to the top of the ushiftRight section.
- #6821 adds basic lemmas about `Ordering`, describing the interaction of `isLT`/`isLE`/`isGE`/`isGT`, `swap` and the constructors. Additionally, it refactors the instance derivation code such that a `LawfulBEq Ordering` instance is also derived automatically.
- #6826 adds injectivity theorems for inductives that did not get them automatically (because they are defined too early) but also not yet manuall later.
- #6828 adds add/sub injectivity lemmas for BitVec, and then adds specialized forms with additional symmetries for the `bv_normalize` normal form.
- #6831 completes the alignment of `List`/`Array`/`Vector` lemmas about `isEqv` and `==`.
- #6833 makes the signatures of `find` functions across `List`/`Array`/`Vector` consistent. Verification lemmas will follow in subsequent PRs.
- #6835 fills some gaps in the `Vector` API, adding `mapM`, `zip`, and `ForIn'` and `ToStream` instances.
- #6838 completes aligning the (limited) verification API for `List`/`Array`/`Vector.ofFn`.
- #6840 completes the alignment of `List`/`Array`/`Vector.zip`/`zipWith`/`zipWithAll`/`unzip` lemmas.
- #6845 adds missing monadic higher order functions on `List`/`Array`/`Vector`. Only the most basic verification lemmas (relating the operations on the three container types) are provided for now.
- #6848 adds simp lemmas proving `x + y = x ↔ x = 0` for BitVec, along with symmetries, and then adds these to the bv_normalize simpset.
- #6860 makes `take`/`drop`/`extract` available for each of `List`/`Array`/`Vector`. The simp normal forms differ, however: in `List`, we simplify `extract` to `take`+`drop`, while in `Array` and `Vector` we simplify `take` and `drop` to `extract`. We also provide `Array`/`Vector.shrink`, which simplifies to `take`, but is implemented by repeatedly popping. Verification lemmas for `Array`/`Vector.extract` to follow in a subsequent PR.
- #6862 defines Cooper resolution with a divisibility constraint as formulated in "Cutting to the Chase: Solving Linear Integer Arithmetic" by Dejan Jovanović and Leonardo de Moura, DOI 10.1007/s10817-013-9281-x.
- #6863 allows fixing regressions in mathlib introduced in nightly-2024-02-25 by allowing the use of `x * y` in match patterns. There are currently 11 instances in mathlib explicitly flagging the lack of this match pattern.
- #6864 adds lemmas relating the operations on findIdx?/findFinIdx?/idxOf?/findIdxOf?/eraseP/erase on List and on Array. It's preliminary to aligning the verification lemmas for `find...` and `erase...`.

- #6868 completes the alignment across `List`/`Array`/`Vector` of lemmas about the `eraseP`/`erase`/`eraseIdx` operations.
- #6872 adds lemmas for xor injectivity and when and/or/xor equal allOnes or zero. Then I plumb support for the new lemmas through to bv_normalize.
- #6875 adds a lemma relating `msb` and `getMsbD`, and three lemmas regarding `getElem` and `shiftConcat`. These lemmas were needed in Batteries#1078 and the request to upstream was made in the review of that PR.
- #6878 completes alignments of `List`/`Array`/`Vector` lemmas about `range`, `range'`, and `zipIdx`.
- #6883 completes the alignment of lemmas about monadic functions on `List`/`Array`/`Vector`. Amongst other changes, we change the simp normal form from `List.forM` to `ForM.forM`, and correct the definition of `List.flatMapM`, which previously was returning results in the incorrect order. There remain many gaps in the verification lemmas for monadic functions; this PR only makes the lemmas uniform across `List`/`Array`/`Vector`.
- #6890 teaches bv_normalize to replace subtractions on one side of an equality with an addition on the other side, this re-write eliminates a not + addition in the normalized form so it is easier on the solver.
- #6912 aligns current coverage of `find`-type theorems across `List`/`Array`/`Vector`. There are still quite a few holes in this API, which will be filled later.

Compiler

- #6535 avoids a linker warning on Windows.
- #6547 should prevent Lake from accidentally picking up other linkers installed on the machine.
- #6574 actually prevents Lake from accidentally picking up other toolchains installed on the machine.
- #6664 changes the toMono pass to longer filter out type class instances, because they may actually be needed for later compilation.
- #6665 adds a new lcAny constant to Prelude, which is meant for use in LCNF to represent types whose dependency on another term has been erased during compilation. This is in addition to the existing lcErased constant, which represents types that are irrelevant.
- #6678 modifies LCNF.toMonoType to use a more refined type erasure scheme, which distinguishes between irrelevant/erased information (represented by lcErased) and erased type dependencies (represented by lcAny). This corresponds to the irrelevant/object distinction in the old code generator.
- #6680 makes the new code generator skip generating code for decls with an implemented_by decl, just like the old code generator.
- #6757 adds support for applying crimp theorems in toLCNF.

- #6758 prevents deadlocks from non-cyclical task waits that may otherwise occur during parallel elaboration with small threadpool sizes.
- #6837 adds Float32 to the LCNF builtinRuntimeTypes list. This was missed during the initial Float32 implementation, but this omission has the side effect of lowering Float32 to obj in the IR.

Pretty Printing

- #6703 modifies the delaborator so that in `pp.tagAppFns` mode, generalized field notation is tagged with the head constant. The effect is that docgen documentation will linkify dot notation. Internal change: now formatted `rawIdent` can be tagged.
- #6716 renames the option `infoview.maxTraceChildren` to `maxTraceChildren` and applies it to the cmdline driver and language server clients lacking an info view as well. It also implements the common idiom of the option value `0` meaning "unlimited".
- #6729 makes the pretty printer for `.coeFun`-tagged functions respect `pp.tagAppFns`. The effect is that in docgen, when an expression pretty prints as `f x y z` with `f` a coerced function, then if `f` is a constant it will be linkified.
- #6730 changes how app unexpanders are invoked. Before the ref was `.missing`, but now the ref is the head constant's delaborated syntax. This way, when `pp.tagAppFns` is true, then tokens in app unexpanders are annotated with the head constant. The consequence is that in docgen, tokens will be linkified. This new behavior is consistent with how `notation` defines app unexpanders.

Documentation

- #6549 fixes #6548.
- #6638 correct the docstring of theorem `Bitvec.toNat_add_of_lt`
- #6643 changes the macOS docs to indicate that Lean now requires pkgconf to build.
- #6646 changes the ubuntu docs to indicate that Lean now requires pkgconf to build.
- #6738 updates our lexical structure documentation to mention the newly supported ¡ which lives in a separate unicode block and is thus not captured by the current ranges.
- #6885 fixes the name of the truncating integer division function in the `HDiv.hDiv` docstring (which is shown when hovering over `/`). It was changed from `Int.div` to `Int.tdiv` in #5301.

Server

- #6597 fixes the indentation of nested traces nodes in the info view.
- #6794 fixes a significant auto-completion performance regression that was introduced in #5666, i.e. v4.14.0.

Lake

- #6290 uses `StateRefT` instead of `StateT` to equip the Lake build monad with a build store.

- #6323 adds a new Lake CLI command, `lake query`, that both builds targets and outputs their results. It can produce raw text or JSON -formatted output (with `--json` / `-J`).
- #6418 alters all builtin Lake facets to produce `Job` objects.
- #6627 aims to fix the trace issues reported by Mathlib that are breaking `lake exe cache` in downstream projects.
- #6631 sets `MACOSX_DEPLOYMENT_TARGET` for shared libraries (it was previously only set for executables).
- #6771 enables `FetchM` to be run from `JobM` / `SpawnM` and vice-versa. This allows calls of `fetch` to asynchronously depend on the outputs of other jobs.
- #6780 makes all targets and all `fetch` calls produce a `Job` of some value. As part of this change, facet definitions (e.g., `library_data`, `module_data`, `package_data`) and Lake type families (e.g., `FamilyOut`) should no longer include `Job` in their types (as this is now implicit).
- #6798 deprecates the `-U` shorthand for the `--update` option.
- #7209 fixes broken Lake tests on Windows' new MSYS2. As of MSYS2 0.0.20250221, `OSTYPE` is now reported as `cygwin` instead of `msys`, which must be accounted for in a few Lake tests.

Other

- #6479 speeds up JSON serialisation by using a lookup table to check whether a string needs to be escaped.
- #6519 adds a script to automatically generate release notes using the new `changelog-*` labels and "..." conventions.
- #6542 introduces a script that automates checking whether major downstream repositories have been updated for a new toolchain release.

Lean 4.16.0 (2025-02-03)

Highlights

Unique `sorry`s

#5757 makes it harder to create "fake" theorems about definitions that are stubbed-out with `sorry` by ensuring that each `sorry` is not definitionally equal to any other. For example, this now fails:

```
example : (sorry : Nat) = sorry := rfl -- fails
```

However, this still succeeds, since the `sorry` is a single indeterminate `Nat`:

```
def f (n : Nat) : Nat := sorry
example : f 0 = f 1 := rfl -- succeeds
```

One can be more careful by putting parameters to the right of the colon:

```
def f : (n : Nat) → Nat := sorry
example : f 0 = f 1 := rfl -- fails
```

Most sources of synthetic sorries (recall: a sorry that originates from the elaborator) are now unique, except for elaboration errors, since making these unique tends to cause a confusing cascade of errors. In general, however, such sorries are labeled. This enables "go to definition" on `sorry` in the Infoview, which brings you to its origin. The option `set_option pp.sorrySource true` causes the pretty printer to show source position information on sorries.

Separators in numeric literals

#6204 lets _ be used in numeric literals as a separator. For example, `1_000_000`, `0xff_ff` or `0b_10_11_01_00`. New lexical syntax:

```
numeral10 : [0-9]+ ("_"+ [0-9]+)*
numeral2  : "0" [bB] ("_"* [0-1]+)+
numeral8  : "0" [oO] ("_"* [0-7]+)+
numeral16 : "0" [xX] ("_"* hex_char+)+
float     : numeral10 "." numeral10? [eE[+-]numeral10]
```

Additional new featues

- #6300 adds the `debug.proofAsSorry` option. When enabled, the proofs of theorems are ignored and replaced with `sorry`.
- #6362 adds the `--error=kind` option (shorthand: `-Ekind`) to the `lean` CLI. When set, messages of `kind` (e.g., `linter.unusedVariables`) will be reported as errors. This setting does nothing in interactive contexts (e.g., the server).
- #6366 adds support for `Float32` and fixes a bug in the runtime.

Library updates

The Lean 4 library saw many changes that improve arithmetic reasoning, enhance data structure APIs, and refine library organization. Key changes include better support for bitwise operations, shifts, and conversions, expanded lemmas for `Array`, `Vector`, and `List`, and improved ordering definitions. Some modules have been reorganized for clarity, and internal refinements ensure greater consistency and correctness.

Breaking changes

#6330 removes unnecessary parameters from the functional induction principles. This is a breaking change; broken code can typically be adjusted simply by passing fewer parameters.

This highlights section was contributed by Violetta Sim.

For this release, 201 changes landed. In addition to the 74 feature additions and 44 fixes listed below there were 7 refactoring changes, 5 documentation improvements and 62 chores.

Language

- #3696 makes all message constructors handle pretty printer errors.
- #4460 runs all linters for a single command (together) on a separate thread from further elaboration, making a first step towards parallelizing the elaborator.
- #5757, see the highlights section above for details.
- #6123 ensures that the configuration in `Simp.Config` is used when reducing terms and checking definitional equality in `simp`.
- #6204, see the highlights section above for details.
- #6270 fixes a bug that could cause the `injectivity` tactic to fail in reducible mode, which could cause unfolding lemma generation to fail (used by tactics such as `unfold`). In particular, `Lean.Meta.isConstructorApp'?` was not aware that `n + 1` is equivalent to `Nat.succ n`.
- #6273 modifies the "foo has been deprecated: use betterFoo instead" warning so that foo and betterFoo are hoverable.
- #6278 enables simp configuration options to be passed to `norm_cast`.
- #6286 ensure `bv_decide` uses definitional equality in its reflection procedure as much as possible. Previously it would build up explicit congruence proofs for the kernel to check. This reduces the size of proof terms passed to kernel speeds up checking of large reflection proofs.
- #6288 uses Lean.RArray in bv_decide's reflection proofs. Giving speedups on problems with lots of variables.
- #6295 sets up simprocs for all the remaining operations defined in `Init.Data.Fin.Basic`
- #6300, see the highlights section above for details.
- #6330, see the highlights section above for details.
- #6362, see the highlights section above for details.
- #6366, see the highlights section above for details.
- #6375 fixes a bug in the simplifier. It was producing terms with loose bound variables when eliminating unused `let_fun` expressions.
- #6378 adds an explanation to the error message when `cases` and `induction` are applied to a term whose type is not an inductive type. For `Prop`, these tactics now suggest the `by_cases` tactic. Example:

```
tactic 'cases' failed, major premise type is not an inductive type
  Prop
```

- #6381 fixes a bug in `withTrackingZetaDelta` and `withTrackingZetaDeltaSet`. The `MetaM` caches need to be reset. See new test.
- #6385 fixes a bug in `simp_all?` that caused some local declarations to be omitted from the `Try this:` suggestions.
- #6386 ensures that `revertAll` clears auxiliary declarations when invoked directly by users.
- #6387 fixes a type error in the proof generated by the `contradiction` tactic.

- #6397 ensures that `simp` and `dsimp` do not unfold definitions that are not intended to be unfolded by the user. See issue #5755 for an example affected by this issue.
- #6398 ensures `Meta.check` check projections.
- #6412 adds reserved names for congruence theorems used in the simplifier and `grind` tactics. The idea is prevent the same congruence theorems to be generated over and over again.
- #6413 introduces the following features to the WIP `grind` tactic:
- `Expr` internalization.
- Congruence theorem cache.
- Procedure for adding new facts
- New tracing options
- New preprocessing steps: fold projections and eliminate dangling `Expr.mdata`
- #6414 fixes a bug in `Lean.Meta.Closure` that would introduce under-applied delayed assignment metavariables, which would keep them from ever getting instantiated. This bug affected `match` elaboration when the expected type contained postponed elaboration problems, for example tactic blocks.
- #6419 fixes multiple bugs in the WIP `grind` tactic. It also adds support for printing the `grind` internal state.
- #6428 adds a new preprocessing step to the `grind` tactic: universe-level normalization. The goal is to avoid missing equalities in the congruence closure module.
- #6430 adds the predicate `Expr.fvarsSet a b`, which returns `true` if and only if the free variables in `a` are a subset of the free variables in `b`.
- #6433 adds a custom type and instance canonicalizer for the (WIP) `grind` tactic. The `grind` tactic uses congruence closure but disregards types, type formers, instances, and proofs. Proofs are ignored due to proof irrelevance. Types, type formers, and instances are considered supporting elements and are not factored into congruence detection. Instead, `grind` only checks whether elements are structurally equal, which, in the context of the `grind` tactic, is equivalent to pointer equality. See new tests for examples where the canonicalizer is important.
- #6435 implements the congruence table for the (WIP) `grind` tactic. It also fixes several bugs, and adds a new preprocessing step.
- #6437 adds support for detecting congruent terms in the (WIP) `grind` tactic. It also introduces the `grind.debug` option, which, when set to `true`, checks many invariants after each equivalence class is merged. This option is intended solely for debugging purposes.
- #6438 ensures `norm_cast` doesn't fail to act in the presence of `no_index` annotations
- #6441 adds basic truth value propagation rules to the (WIP) `grind` tactic.
- #6442 fixes the `checkParents` sanity check in `grind`.
- #6443 adds support for propagating the truth value of equalities in the (WIP) `grind` tactic.

- #6447 refactors `grind` and adds support for invoking the simplifier using the `GrindM` monad.
- #6448 declares the command `builtin_grind_propagator` for registering equation propagator for `grind`. It also declares the auxiliary the attribute.
- #6449 completes the implementation of the command `builtin_grind_propagator`.
- #6452 adds support for generating (small) proofs for any two expressions that belong to the same equivalence class in the `grind` tactic state.
- #6453 improves bv_decide's performance in the presence of large literals.
- #6455 fixes a bug in the equality proof generator in the (WIP) `grind` tactic.
- #6456 fixes another bug in the equality proof generator in the (WIP) `grind` tactic.
- #6457 adds support for generating congruence proofs for congruences detected by the `grind` tactic.
- #6458 adds support for compact congruence proofs in the (WIP) `grind` tactic. The `mkCongrProof` function now verifies whether the congruence proof can be constructed using only `congr`, `congrFun`, and `congrArg`, avoiding the need to generate the more complex `hcongr` auxiliary theorems.
- #6459 adds the (WIP) `grind` tactic. It currently generates a warning message to make it clear that the tactic is not ready for production.
- #6461 adds a new propagation rule for negation to the (WIP) `grind` tactic.
- #6463 adds support for constructors to the (WIP) `grind` tactic. When merging equivalence classes, `grind` checks for equalities between constructors. If they are distinct, it closes the goal; if they are the same, it applies injectivity.
- #6464 completes support for literal values in the (WIP) `grind` tactic. `grind` now closes the goal whenever it merges two equivalence classes with distinct literal values.
- #6465 adds support for projection functions to the (WIP) `grind` tactic.
- #6466 completes the implementation of `addCongrTable` in the (WIP) `grind` tactic. It also adds a new test to demonstrate why the extra check is needed. It also updates the field `cgRoot` (congruence root).
- #6468 fixes issue #6467
- #6469 adds support code for implementing e-match in the (WIP) `grind` tactic.
- #6470 introduces a command for specifying patterns used in the heuristic instantiation of global theorems in the `grind` tactic. Note that this PR only adds the parser.
- #6472 implements the command `grind_pattern`. The new command allows users to associate patterns with theorems. These patterns are used for performing heuristic instantiation with e-matching. In the future, we will add the attributes `@[grind_eq]`, `@[grind_fwd]`, and `@[grind_bwd]` to compute the patterns automatically for theorems.
- #6473 adds a deriving handler for the `ToExpr` class. It can handle mutual and nested inductive types, however it falls back to creating `partial` instances in such cases. This is upstreamed from the Mathlib deriving handler written by @kmill, but has fixes to handle

autoimplicit universe level variables.

- #6474 adds pattern validation to the `grind_pattern` command. The new `checkCoverage` function will also be used to implement the attributes `@[grind_eq]`, `@[grind_fwd]`, and `@[grind_bwd]`.
- #6475 adds support for activating relevant theorems for the (WIP) `grind` tactic. We say a theorem is relevant to a `grind` goal if the symbols occurring in its patterns also occur in the goal.
- #6478 internalize nested ground patterns when activating ematch theorems in the (WIP) `grind` tactic.
- #6481 implements E-matching for the (WIP) `grind` tactic. We still need to finalize and internalize the new instances.
- #6484 addresses a few error messages where diffs weren't being exposed.
- #6485 implements `Grind.EMatch.instantiateTheorem` in the (WIP) `grind` tactic.
- #6487 adds source position information for `structure` parent projections, supporting "go to definition". Closes #3063.
- #6488 fixes and refactors the E-matching module for the (WIP) `grind` tactic.
- #6490 adds basic configuration options for the `grind` tactic.
- #6492 fixes a bug in the theorem instantiation procedure in the (WIP) `grind` tactic.
- #6497 fixes another theorem instantiation bug in the `grind` tactic. It also moves new instances to be processed to `Goal`.
- #6498 adds support in the `grind` tactic for propagating dependent forall terms `forall (h : p), q[h]` where p is a proposition.
- #6499 fixes the proof canonicalizer for `grind`.
- #6500 fixes a bug in the `markNestedProofs` used in `grind`. See new test.
- #6502 fixes a bug in the proof assembly procedure utilized by the `grind` tactic.
- #6503 adds a simple strategy to the (WIP) `grind` tactic. It just keeps internalizing new theorem instances found by E-matching.
- #6506 adds the `monotonicity` tactic, intended to be used inside the `partial_fixpoint` feature.
- #6508 fixes a bug in the sanity checkers for the `grind` tactic. See the new test for an example of a case where it was panicking.
- #6509 fixes a bug in the congruence closure data structure used in the `grind` tactic. The new test includes an example that previously caused a panic. A similar panic was also occurring in the test `grind_nested_proofs.lean`.
- #6510 adds a custom congruence rule for equality in `grind`. The new rule takes into account that Eq is a symmetric relation. In the future, we will add support for arbitrary symmetric relations. The current rule is important for propagating disequalities effectively in `grind`.

- #6512 introduces support for user-defined fallback code in the `grind` tactic. The fallback code can be utilized to inspect the state of failing `grind` subgoals and/or invoke user-defined automation. Users can now write `grind on_failure <code>`, where `<code>` should have the type `GoalM Unit`. See the modified tests in this PR for examples.
- #6513 adds support for (dependent) if-then-else terms (i.e., `ite` and `dite` applications) in the `grind` tactic.
- #6514 enhances the assertion of new facts in `grind` by avoiding the creation of unnecessary metavariables.

Library

- #6182 adds `BitVec.[toInt|toFin]_concat` and moves a couple of theorems into the concat section, as `BitVec.msb_concat` is needed for the `toInt_concat` proof.
- #6188 completes the `toNat` theorems for the bitwise operations (`and`, `or`, `xor`, `shiftLeft`, `shiftRight`) of the UInt types and adds `toBitVec` theorems as well. It also renames `and_toNat` to `toNat_and` to fit with the current naming convention.
- #6238 adds theorems characterizing the value of the unsigned shift right of a bitvector in terms of its 2s complement interpretation as an integer. Unsigned shift right by at least one bit makes the value of the bitvector less than or equal to `2^(w-1)`, makes the interpretation of the bitvector `Int` and `Nat` agree. In the case when `n = 0`, then the shift right value equals the integer interpretation.
- #6244 changes the implementation of `HashMap.toList`, so the ordering agrees with `HashMap.toArray`.
- #6272 introduces the basic theory of permutations of `Array`s and proves `Array.swap_perm`.
- #6282 moves `IO.Channel` and `IO.Mutex` from `Init` to `Std.Sync` and renames them to `Std.Channel` and `Std.Mutex`.
- #6294 upstreams `List.length_flatMap`, `countP_flatMap` and `count_flatMap` from Mathlib. These were not possible to state before we upstreamed `List.sum`.
- #6315 adds `protected` to `Fin.cast` and `BitVec.cast`, to avoid confusion with `_root_.cast`. These should mostly be used via dot-notation in any case.
- #6316 adds lemmas simplifying `for` loops over `Option` into `Option.pelim`, giving parity with lemmas simplifying `for` loops of `List` into `List.fold`.
- #6317 completes the basic API for BitVec.ofBool.
- #6318 generalizes the universe level for `Array.find?`, by giving it a separate implementation from `Array.findM?`.
- #6324 adds `GetElem` lemmas for the basic `Vector` operations.
- #6333 generalizes the panic functions to a type of `Sort u` rather than `Type u`. This better supports universe polymorphic types and avoids confusing errors.

- #6334 adds `Nat` theorems for distributing `>>>` over bitwise operations, paralleling those of `BitVec`.
- #6338 adds `BitVec.[toFin|getMsbD]_setWidth` and `[getMsb|msb]_signExtend` as well as `ofInt_toInt`.
- #6341 generalizes `DecidableRel` to allow a heterogeneous relation.
- #6353 reproduces the API around `List.any/all` for `Array.any/all`.
- #6364 makes fixes suggested by the Batteries environment linters, particularly `simpNF`, and `unusedHavesSuffices`.
- #6365 expands the `Array.set` and `Array.setIfInBounds` lemmas to match existing lemmas for `List.set`.
- #6367 brings Vector lemmas about membership and indexing to parity with List and Array.
- #6369 adds lemmas about `Vector.set`, `anyM`, `any`, `allM`, and `all`.
- #6376 adds theorems about == on `Vector`, reproducing those already on `List` and `Array`.
- #6379 replaces the inductive predicate `List.lt` with an upstreamed version of `List.Lex` from Mathlib. (Previously `Lex.lt` was defined in terms of <; now it is generalized to take an arbitrary relation.) This subtly changes the notion of ordering on `List α`.

  `List.lt` was a weaker relation: in particular if $l_1 < l_2$, then $a :: l_1 < b :: l_2$ may hold according to `List.lt` even if a and b are merely incomparable (either neither $a < b$ nor $b < a$), whereas according to `List.Lex` this would require $a = b$.

  When < is total, in the sense that $\neg \cdot < \cdot$ is antisymmetric, then the two relations coincide.

  Mathlib was already overriding the order instances for `List α`, so this change should not be noticed by anyone already using Mathlib.

  We simultaneously add the boolean valued `List.lex` function, parameterised by a BEq typeclass and an arbitrary `lt` function. This will support the flexibility previously provided for `List.lt`, via a == function which is weaker than strict equality.
- #6390 redefines `Range.forIn'` and `Range.forM`, in preparation for writing lemmas about them.
- #6391 requires that the step size in `Std.Range` is positive, to avoid ill-specified behaviour.
- #6396 adds lemmas reducing for loops over `Std.Range` to for loops over `List.range'`.
- #6399 adds basic lemmas about lexicographic order on Array and Vector, achieving parity with List.
- #6423 adds missing lemmas about lexicographic order on List/Array/Vector.
- #6477 adds the necessary domain theory that backs the `partial_fixpoint` feature.

Compiler

- #6311 adds support for HEq to the new code generator.

- #6348 adds support for `Float32` to the Lean runtime.
- #6350 adds missing features and fixes bugs in the `Float32` support
- #6383 ensures the new code generator produces code for `opaque` definitions that are not tagged as `@[extern]`. Remark: This is the behavior of the old code generator.
- #6405 adds support for erasure of `Decidable.decide` to the new code generator. It also adds a new `Probe.runOnDeclsNamed` function, which is helpful for writing targeted single-file tests of compiler internals.
- #6415 fixes a bug in the `sharecommon` module, which was returning incorrect results for objects that had already been processed by `sharecommon`. See the new test for an example that triggered the bug.
- #6429 adds support for extern LCNF decls, which is required for parity with the existing code generator.
- #6535 avoids a linker warning on Windows.
- #6547 should prevent Lake from accidentally picking up other linkers installed on the machine.
- #6574 actually prevents Lake from accidentally picking up other toolchains installed on the machine.

Pretty Printing

- #5689 adjusts the way the pretty printer unresolves names. It used to make use of all `exports` when pretty printing, but now it only uses `exports` that put names into parent namespaces (heuristic: these are "API exports" that are intended by the library author), rather than "horizontal exports" that put the names into an unrelated namespace, which the dot notation feature in #6189 now incentivizes.
- #5757, aside from introducing labeled sorries, fixes the bug that the metadata attached to the pretty-printed representation of arguments with a borrow annotation (for example, the second argument of `String.append`), is inconsistent with the metadata attached to the regular arguments.

Documentation

- #6450 adds a docstring to the `@[app_delab]` attribute.

Server

- #6279 fixes a bug in structure instance field completion that caused it to not function correctly for bracketed structure instances written in Mathlib style.
- #6408 fixes a regression where goals that don't exist were being displayed. The regression was triggered by #5835 and originally caused by #4926.

Lake

- #6176 changes Lake's build process to no longer use `leanc` for compiling C files or linking shared libraries and executables. Instead, it directly invokes the bundled compiler (or the native compiler if none) using the necessary flags.
- #6289 adapts Lake modules to use `prelude` and includes them in the `check-prelude` CI.
- #6291 ensures the the log error position is properly preserved when prepending stray log entries to the job log. It also adds comparison support for `Log.Pos`.
- #6388 merges `BuildJob` and `Job`, deprecating the former. `Job` now contains a trace as part of its state which can be interacted with monadically. also simplifies the implementation of `OpaqueJob`.
- #6411 adds the ability to override package entries in a Lake manifest via a separate JSON file. This file can be specified on the command line with `--packages` or applied persistently by placing it at `.lake/package-overrides.json`.
- #6422 fixes a bug in #6388 where the `Package.afterBuildCahe*` functions would produce different traces depending on whether the cache was fetched.
- #6627 aims to fix the trace issues reported by Mathlib that are breaking `lake exe cache` in downstream projects.
- #6631 sets `MACOSX_DEPLOYMENT_TARGET` for shared libraries (it was previously only set for executables).

Other

- #6285 upstreams the `ToLevel` typeclass from mathlib and uses it to fix the existing `ToExpr` instances so that they are truly universe polymorphic (previously it generated malformed expressions when the universe level was nonzero). We improve on the mathlib definition of `ToLevel` to ensure the class always lives in `Type`, irrespective of the universe parameter.
- #6363 fixes errors at load time in the comparison mode of the Firefox profiler.

Lean 4.15.0 (2025-01-04)

Language

- #4595 implements `Simp.Config.implicitDefEqsProofs`. When `true` (default: `true`), `simp` will **not** create a proof term for a rewriting rule associated with an `rfl`-theorem. Rewriting rules are provided by users by annotating theorems with the attribute `@[simp]`. If the proof of the theorem is just `rfl` (reflexivity), and `implicitDefEqProofs := true`, `simp` will **not** create a proof term which is an application of the annotated theorem.
- #5429 avoid negative environment lookup
- #5501 ensure `instantiateMVarsProfiling` adds a trace node
- #5856 adds a feature to the the mutual def elaborator where the `instance` command yields theorems instead of definitions when the class is a `Prop`.
- #5907 unset trailing for `simpa?` "try this" suggestion

- #5920 changes the rule for which projections become instances. Before, all parents along with all indirect ancestors that were represented as subobject fields would have their projections become instances. Now only projections for direct parents become instances.
- #5934 make `all_goals` admit goals on failure
- #5942 introduce synthetic atoms in bv_decide
- #5945 adds a new definition `Message.kind` which returns the top-level tag of a message. This is serialized as the new field `kind` in `SerialMessaege` so that i can be used by external consumers (e.g., Lake) to identify messages via `lean --json`.
- #5968 `arg` conv tactic misreported number of arguments on error
- #5979 BitVec.twoPow in bv_decide
- #5991 simplifies the implementation of `omega`.
- #5992 fix style in bv_decide normalizer
- #5999 adds configuration options for `decide`/`decide!`/`native_decide` and refactors the tactics to be frontends to the same backend. Adds a `+revert` option that cleans up the local context and reverts all local variables the goal depends on, along with indirect propositional hypotheses. Makes `native_decide` fail at elaboration time on failure without sacrificing performance (the decision procedure is still evaluated just once). Now `native_decide` supports universe polymorphism.
- #6010 changes `bv_decide`'s configuration from lots of `set_option` to an elaborated config like `simp` or `omega`. The notable exception is `sat.solver` which is still a `set_option` such that users can configure a custom SAT solver globally for an entire project or file. Additionally it introduces the ability to set `maxSteps` for the simp preprocessing run through the new config.
- #6012 improves the validation of new syntactic tokens. Previously, the validation code had inconsistencies: some atoms would be accepted only if they had a leading space as a pretty printer hint. Additionally, atoms with internal whitespace are no longer allowed.
- #6016 removes the `decide!` tactic in favor of `decide +kernel` (breaking change).
- #6019 removes @[specilize] from `MkBinding.mkBinding`, which is a function that cannot be specialized (as none of its arguments are functions). As a result, the specializable function `Nat.foldRevM.loop` doesn't get specialized, which leads to worse performing code.
- #6022 makes the `change` tactic and conv tactic use the same elaboration strategy. It works uniformly for both the target and local hypotheses. Now `change` can assign metavariables, for example:

```
example (x y z : Nat) : x + y = z := by
  change ?a = _
  let w := ?a
  -- now `w : Nat := x + y`
```

- #6024 fixes a bug where the monad lift coercion elaborator would partially unify expressions even if they were not monads. This could be taken advantage of to propagate information that could help elaboration make progress, for example the first `change` worked because the monad lift coercion elaborator was unifying `@Eq _ _` with `@Eq (Nat × Nat) p`:

```
example (p : Nat × Nat) : p = p := by
  change _ = ⟨_, _⟩ -- used to work (yielding `p = (p.fst, p.snd)`),
now it doesn't
  change ⟨_, _⟩ = _ -- never worked
```

As such, this is a breaking change; you may need to adjust expressions to include additional implicit arguments.

- #6029 adds a normalization rule to `bv_normalize` (which is used by `bv_decide`) that converts `x / 2^k` into `x >>> k` under suitable conditions. This allows us to simplify the expensive division circuits that are used for bitblasting into much cheaper shifting circuits. Concretely, it allows for the following canonicalization:
- #6030 fixes `simp only [· ∈ ·]` after #5020.
- #6035 introduces the and flattening pre processing pass from Bitwuzla to `bv_decide`. It splits hypotheses of the form `(a && b) = true` into `a = true` and `b = true` which has synergy potential with the already existing embedded constraint substitution pass.
- #6037 fixes `bv_decide`'s embedded constraint substitution to generate correct counter examples in the corner case where duplicate theorems are in the local context.
- #6045 add `LEAN_ALWAYS_INLINE` to some functions
- #6048 fixes `simp?` suggesting output with invalid indentation
- #6051 mark `Meta.Context.config` as private
- #6053 fixes the caching infrastructure for `whnf` and `isDefEq`, ensuring the cache accounts for all relevant configuration flags. It also cleans up the `WHNF.lean` module and improves the configuration of `whnf`.
- #6061 adds a simp_arith benchmark.
- #6062 optimize Nat.Linear.Expr.toPoly
- #6064 optimize Nat.Linear.Poly.norm
- #6068 improves the asymptotic performance of `simp_arith` when there are many variables to consider.
- #6077 adds options to `bv_decide`'s configuration structure such that all non mandatory preprocessing passes can be disabled.
- #6082 changes how the canonicalizer handles `forall` and `lambda`, replacing bvars with temporary fvars. Fixes a bug reported by @hrmacbeth on zulip.
- #6093 use mkFreshUserName in ArgsPacker

- #6096 improves the `#print` command for structures to show all fields and which parents the fields were inherited from, hiding internal details such as which parents are represented as subobjects. This information is still present in the constructor if needed. The pretty printer for private constants is also improved, and it now handles private names from the current module like any other name; private names from other modules are made hygienic.
- #6098 modifies `Lean.MVarId.replaceTargetDefEq` and `Lean.MVarId.replaceLocalDeclDefEq` to use `Expr.equal` instead of `Expr.eqv` when determining whether the expression has changed. This is justified on the grounds that binder names and binder infos are user-visible and affect elaboration.
- #6105 fixes a stack overflow caused by a cyclic assignment in the metavariable context. The cycle is unintentionally introduced by the structure instance elaborator.
- #6108 turn off pp.mvars in apply? results
- #6109 fixes an issue in the `injection` tactic. This tactic may execute multiple sub-tactics. If any of them fail, we must backtrack the partial assignment. This issue was causing the error: "`mvarId` is already assigned" in issue #6066. The issue is not yet resolved, as the equation generator for the match expressions is failing in the example provided in this issue.
- #6112 makes stricter requirements for the `@[deprecated]` attribute, requiring either a replacement identifier as `@[deprecated bar]` or suggestion text `@[deprecated "Past its use by date"]`, and also requires a `since := "..."` field.
- #6114 liberalizes atom rules by allowing `''` to be a prefix of an atom, after #6012 only added an exception for `''` alone, and also adds some unit tests for atom validation.
- #6116 fixes a bug where structural recursion did not work when indices of the recursive argument appeared as function parameters in a different order than in the argument's type's definition.
- #6125 adds support for `structure` in `mutual` blocks, allowing inductive types defined by `inductive` and `structure` to be mutually recursive. The limitations are (1) that the parents in the `extends` clause must be defined before the `mutual` block and (2) mutually recursive classes are not allowed (a limitation shared by `class inductive`). There are also improvements to universe level inference for inductive types and structures. Breaking change: structure parents now elaborate with the structure in scope (fix: use qualified names or rename the structure to avoid shadowing), and structure parents no longer elaborate with autoimplicits enabled.
- #6128 does the same fix as #6104, but such that it doesn't break the test/the file in `Plausible`. This is done by not creating unused let binders in metavariable types that are made by `elimMVar`. (This is also a positive thing for users looking at metavariable types, for example in error messages)
- #6129 fixes a bug at `isDefEq` when `zetaDelta := false`. See new test for a small example that exposes the issue.
- #6131 fixes a bug at the definitional equality test (`isDefEq`). At unification constraints of the form `c.{u} =?= c.{v}`, it was not trying to unfold `c`. This bug did not affect the kernel.
- #6141 make use of recursive structures in snapshot types

- #6145 fixes the `revert` tactic so that it creates a `syntheticOpaque` metavariable as the new goal, instead of a `natural` metavariable
- #6146 fixes a non-termination bug that occurred when generating the match-expression splitter theorem. The bug was triggered when the proof automation for the splitter theorem repeatedly applied `injection` to the same local declaration, as it could not be removed due to forward dependencies. See issue #6065 for an example that reproduces this issue.
- #6165 modifies structure instance notation and `where` notation to use the same notation for fields. Structure instance notation now admits binders, type ascriptions, and equations, and `where` notation admits full structure lvals. Examples of these for structure instance notation:

```
structure PosFun where
  f : Nat → Nat
  pos : ∀ n, 0 < f n
```

- #6168 extends the "motive is not type correct" error message for the rewrite tactic to explain what it means. It also pretty prints the type-incorrect motive and reports the type error.
- #6170 adds core metaprogramming functions for forking off background tasks from elaboration such that their results are visible to reporting and the language server
- #6175 fixes a bug with the `structure`/`class` command where if there are parents that are not represented as subobjects but which used other parents as instances, then there would be a kernel error. Closes #2611.
- #6180 fixes a non-termination bug that occurred when generating the match-expression equation theorems. The bug was triggered when the proof automation for the equation theorem repeatedly applied `injection` (to the same local declaration, as it could not be removed due to forward dependencies. See issue #6067 for an example that reproduces this issue.
- #6189 changes how generalized field notation ("dot notation") resolves the function. The new resolution rule is that if `x : S`, then `x.f` resolves the name `S.f` relative to the root namespace (hence it now affected by `export` and `open`). Breaking change: aliases now resolve differently. Before, if `x : S`, and if `S.f` is an alias for `S'.f`, then `x.f` would use `S'.f` and look for an argument of type `S'`. Now, it looks for an argument of type `S`, which is more generally useful behavior. Code making use of the old behavior should consider defining `S` or `S'` in terms of the other, since dot notation can unfold definitions during resolution.
- #6206 makes it possible to write `rw (occs := [1,2]) ...` instead of `rw (occs := .pos [1,2]) ...` by adding a coercion from `List.Nat` to `Lean.Meta.Occurrences`.
- #6220 adds proper support for `let_fun` in `simp`.
- #6236 fixes an issue where edits to a command containing a nested docstring fail to reparse the entire command.

Library

- #4904 introduces date and time functionality to the Lean 4 Std.

- #5616 is a follow-up to https://github.com/leanprover/lean4/pull/5609, where we add lemmas characterizing `smtUDiv` and `smtSDiv`'s behavior when the denominator is zero.
- #5866 verifies the `keys` function on `Std.HashMap`.
- #5885 add Int16/Int32/Int64
- #5926 add `Option.or_some'`
- #5927 `List.pmap_eq_self`
- #5937 upstream lemmas about Fin.foldX
- #5938 upstream List.ofFn and relate to Array.ofFn
- #5941 List.mapFinIdx, lemmas, relate to Array version
- #5949 consolidate `decide_True` and `decide_true_eq_true`
- #5950 relate Array.takeWhile with List.takeWhile
- #5951 remove @[simp] from BitVec.ofFin_sub and sub_ofFin
- #5952 relate Array.eraseIdx with List.eraseIdx
- #5961 define ISize and basic operations on it
- #5969 upstream List.insertIdx from Batteries, lemmas from Mathlib, and revise lemmas
- #5970 deprecate Array.split in favour of identical Array.partition
- #5971 relate Array.isPrefixOf with List.isPrefixOf
- #5972 relate Array.zipWith/zip/unzip with List versions
- #5974 add another List.find?_eq_some lemma
- #5981 names the default SizeOf instance `instSizeOfDefault`
- #5982 minor lemmas about List.ofFn
- #5984 adds lemmas for `List` for the interactions between {`foldl`, `foldr`, `foldlM`, `foldlrM`} and {`filter`, `filterMap`}.
- #5985 relates the operations `findSomeM?`, `findM?`, `findSome?`, and `find?` on `Array` with the corresponding operations on `List`, and also provides simp lemmas for the `Array` operations `findSomeRevM?`, `findRevM?`, `findSomeRev?`, `findRev?` (in terms of `reverse` and the usual forward find operations).
- #5987 BitVec.getMsbD in bv_decide
- #5988 changes the signature of `Array.set` to take a `Nat`, and a tactic-provided bound, rather than a `Fin`.
- #5995 BitVec.sshiftRight' in bv_decide
- #6007 List.modifyTailIdx naming fix
- #6008 missing @[ext] attribute on monad transformer ext lemmas
- #6023 variants of List.forIn_eq_foldlM
- #6025 deprecate duplicated Fin.size_pos

- #6032 changes the signature of `Array.get` to take a Nat and a proof, rather than a `Fin`, for consistency with the rest of the (planned) Array API. Note that because of bootstrapping issues we can't provide `get_elem_tactic` as an autoparameter for the proof. As users will mostly use the `xs[i]` notation provided by `GetElem`, this hopefully isn't a problem.
- #6041 modifies the order of arguments for higher-order `Array` functions, preferring to put the `Array` last (besides positional arguments with defaults). This is more consistent with the `List` API, and is more flexible, as dot notation allows two different partially applied versions.
- #6049 adds a primitive for accessing the current thread ID
- #6052 adds `Array.pmap`, as well as a `@[csimp]` lemma in terms of the no-copy `Array.attachWith`.
- #6055 adds lemmas about for loops over `Array`, following the existing lemmas for `List`.
- #6056 upstream some NameMap functions
- #6060 implements conversion functions from `Bool` to all `UIntX` and `IntX` types.
- #6070 adds the Lean.RArray data structure.
- #6074 allow `Sort u` in `Squash`
- #6094 adds raw transmutation of floating-point numbers to and from `UInt64`. Floats and UInts share the same endianness across all supported platforms. The IEEE 754 standard precisely specifies the bit layout of floats. Note that `Float.toBits` is distinct from `Float.toUInt64`, which attempts to preserve the numeric value rather than the bitwise value.
- #6095 generalize `List.get_mem`
- #6097 naming convention and `NaN` normalization
- #6102 moves `IO.rand` and `IO.setRandSeed` to be in the `BaseIO` monad.
- #6106 fix naming of left/right injectivity lemmas
- #6111 fills in the API for `Array.findSome?` and `Array.find?`, transferring proofs from the corresponding List statements.
- #6120 adds theorems `BitVec.(getMsbD, msb)_(rotateLeft, rotateRight)`.
- #6126 adds lemmas for extracting a given bit of a `BitVec` obtained via `sub/neg/sshiftRight'/abs`.
- #6130 adds `Lean.loadPlugin` which exposes functionality similar to the `lean` executable's `--plugin` option to Lean code.
- #6132 duplicates the verification API for `List.attach/attachWith/pmap` over to `Array`.
- #6133 replaces `Array.feraseIdx` and `Array.insertAt` with `Array.eraseIdx` and `Array.insertIdx`, both of which take a `Nat` argument and a tactic-provided proof that it is in bounds. We also have `eraseIdxIfInBounds` and `insertIdxIfInBounds` which are noops if the index is out of bounds. We also provide a `Fin` valued version of `Array.findIdx?`. Together, these quite ergonomically improve the array indexing safety at a number of places in the compiler/elaborator.

- #6136 fixes the run-time evaluation of `(default : Float)`.
- #6139 modifies the signature of the functions `Nat.fold`, `Nat.foldRev`, `Nat.any`, `Nat.all`, so that the function is passed the upper bound. This allows us to change runtime array bounds checks to compile time checks in many places.
- #6148 adds a primitive for creating temporary directories, akin to the existing functionality for creating temporary files.
- #6149 completes the elementwise accessors for `ofNatLt`, `allOnes`, and `not` by adding their implementations of `getMsbD`.
- #6151 completes the `toInt` interface for `BitVec` bitwise operations.
- #6154 implements `BitVec.toInt_abs`.
- #6155 adds `toNat` theorems for `BitVec.signExtend`.
- #6157 adds toInt theorems for BitVec.signExtend.
- #6160 adds theorem `mod_eq_sub`, makes theorem `sub_mul_eq_mod_of_lt_of_le` not private anymore and moves its location within the `rotate*` section to use it in other proofs.
- #6184 uses `Array.findFinIdx?` in preference to `Array.findIdx?` where it allows converting a runtime bounds check to a compile time bounds check.
- #6188 completes the `toNat` theorems for the bitwise operations (`and`, `or`, `xor`, `shiftLeft`, `shiftRight`) of the UInt types and adds `toBitVec` theorems as well. It also renames `and_toNat` to `toNat_and` to fit with the current naming convention.
- #6190 adds the builtin simproc `USize.reduceToNat` which reduces the `USize.toNat` operation on literals less than `UInt32.size` (i.e., `4294967296`).
- #6191 adds `Array.zipWithAll`, and the basic lemmas relating it to `List.zipWithAll`.
- #6192 adds deprecations for `Lean.HashMap` functions which did not receive deprecation attributes initially.
- #6193 completes the TODO in `Init.Data.Array.BinSearch`, removing the `partial` keyword and converting runtime bounds checks to compile time bounds checks.
- #6194 changes the signature of `Array.swap`, so it takes `Nat` arguments with tactic provided bounds checking. It also renames `Array.swap!` to `Array.swapIfInBounds`.
- #6195 renames `Array.setD` to `Array.setIfInBounds`.
- #6197 upstreams the definition of `Vector` from Batteries, along with the basic functions.
- #6200 upstreams `Nat.lt_pow_self` and `Nat.lt_two_pow` from Mathlib and uses them to prove the simp theorem `Nat.mod_two_pow`.
- #6202 makes `USize.toUInt64` a regular non-opaque definition.
- #6203 adds the theorems `le_usize_size` and `usize_size_le`, which make proving inequalities about `USize.size` easier.

- #6205 upstreams some UInt theorems from Batteries and adds more `toNat`-related theorems. It also adds the missing `UInt8` and `UInt16` to/from `USize` conversions so that the the interface is uniform across the UInt types.
- #6207 ensures the `Fin.foldl` and `Fin.foldr` are semireducible. Without this the defeq `example (f : Fin 3 → ℕ) : List.ofFn f = [f 0, f 1, f 2] := rfl` was failing.
- #6208 fix Vector.indexOf?
- #6217 adds `simp` lemmas about `List`'s == operation.
- #6221 fixes:
- Problems in other linux distributions that the default `tzdata` directory is not the same as previously defined by ensuring it with a fallback behavior when directory is missing.
- Trim unnecessary characters from local time identifier.
- #6222 changes the definition of `HashSet.insertMany` and `HashSet.Raw.insertMany` so that it is equivalent to repeatedly calling `HashSet.insert`/`HashSet.Raw.insert`. It also clarifies the docstrings of all the `insert` and `insertMany` functions.
- #6230 copies some lemmas about `List.foldX` to `Array`.
- #6233 upstreams lemmas about `Vector` from Batteries.
- #6234 upstreams the definition and basic lemmas about `List.finRange` from Batteries.
- #6235 relates that operations `Nat.fold`/`foldRev`/`any`/`all` to the corresponding List operations over `List.finRange`.
- #6241 refactors `Array.qsort` to remove runtime array bounds checks, and avoids the use of `partial`. We use the `Vector` API, along with auto_params, to avoid having to write any proofs. The new code benchmarks indistinguishably from the old.
- #6242 deprecates `Fin.ofNat` in favour of `Fin.ofNat'` (which takes an `[NeZero]` instance, rather than returning an element of `Fin (n+1)`).
- #6247 adds the theorems `numBits_pos`, `le_numBits`, `numBits_le` , which make proving inequalities about `System.Platform.numBits` easier.

Compiler

- #5840 changes `lean_sharecommon_{eq,hash}` to only consider the salient bytes of an object, and not any bytes of any unspecified/uninitialized unused capacity.
- #6087 fixes a bug in the constant folding for the `Nat.ble` and `Nat.blt` function in the old code generator, leading to a miscompilation.
- #6143 should make lean better-behaved around sanitizers, per https://github.com/google/sanitizers/issues/1688. As far as I can tell, https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterReturn#algorithm replaces local variables with heap allocations, and so taking the address of a local is not effective at producing a monotonic measure of stack usage.

- #6209 documents under which conditions `Runtime.markPersistent` is unsafe and adjusts the elaborator accordingly
- #6257 harden `markPersistent` uses

Pretty Printing

- #2934 adds the option `pp.parens` (default: false) that causes the pretty printer to eagerly insert parentheses, which can be useful for teaching and for understanding the structure of expressions. For example, it causes `p → q → r` to pretty print as `p → (q → r)`.
- #6014 prevents `Nat.succ ?_` from pretty printing as `?_.succ`, which should make `apply?` be more usable.
- #6085 improves the term info for coercions marked with `CoeFnType.coeFun` (such as `DFunLike.coe` in Mathlib), making "go to definition" on the function name work. Hovering over such a coerced function will show the coercee rather than the coercion expression. The coercion expression can still be seen by hovering over the whitespace in the function application.
- #6096 improves the `#print` command for structures to show all fields and which parents the fields were inherited from, hiding internal details such as which parents are represented as subobjects. This information is still present in the constructor if needed. The pretty printer for private constants is also improved, and it now handles private names from the current module like any other name; private names from other modules are made hygienic.
- #6119 adds a new delab option `pp.coercions.types` which, when enabled, will display all coercions with an explicit type ascription.
- #6161 ensures whitespace is printed before `+opt` and `-opt` configuration options when pretty printing, improving the experience of tactics such as `simp?`.
- #6181 fixes a bug where the signature pretty printer would ignore the current setting of `pp.raw`. This fixes an issue where `#check ident` would not heed `pp.raw`. Closes #6090.
- #6213 exposes the difference in "synthesized type class instance is not definitionally equal" errors.

Documentation

- #6009 fixes a typo in the docstring for prec and makes the text slightly more precise.
- #6040 join → flatten in docstring
- #6110 does some mild refactoring of the `Lean.Elab.StructInst` module while adding documentation.
- #6144 converts 3 doc-string to module docs since it seems that this is what they were intended to be!
- #6150 refine kernel code comments
- #6158 adjust file reference in Data.Sum
- #6239 explains the order in which `Expr.abstract` introduces de Bruijn indices.

Server

- #5835 adds auto-completion for the fields of structure instance notation. Specifically, querying the completions via `Ctrl+Space` in the whitespace of a structure instance notation will now bring up the full list of fields. Whitespace structure completion can be enabled for custom syntax by wrapping the parser for the list of fields in a `structInstFields` parser.
- #5837 fixes an old auto-completion bug where `x.` would issue nonsensical completions when `x.` could not be elaborated as a dot completion.
- #5996 avoid max heartbeat error in completion
- #6031 fixes a regression with go-to-definition and document highlight misbehaving on tactic blocks.
- #6246 fixes a performance issue where the Lean language server would walk the full project file tree every time a file was saved, blocking the processing of all other requests and notifications and significantly increasing overall language server latency after saving.

Lake

- #5684 update toolchain on `lake update`
- #6026 adds a newline at end of each Lean file generated by `lake new` templates.
- #6218 makes Lake no longer automatically fetch GitHub cloud releases if the package build directory is already present (mirroring the behavior of the Reservoir cache). This prevents the cache from clobbering existing prebuilt artifacts. Users can still manually fetch the cache and clobber the build directory by running `lake build <pkg>:release`.
- #6225 makes `lake build` also eagerly print package materialization log lines. Previously, only a `lake update` performed eager logging.
- #6231 improves the errors Lake produces when it fails to fetch a dependency from Reservoir. If the package is not indexed, it will produce a suggestion about how to require it from GitHub.

Other

- #6137 adds support for displaying multiple threads in the trace profiler output.
- #6138 fixes `trace.profiler.pp` not using the term pretty printer.
- #6259 ensures that nesting trace nodes are annotated with timing information iff `trace.profiler` is active.

Lean 4.14.0 (2024-12-02)

Full Changelog: https://github.com/leanprover/lean4/compare/v4.13.0...v4.14.0

Language features, tactics, and metaprograms

- `structure` and `inductive` commands

- #5517 improves universe level inference for the resulting type of an `inductive` or `structure`. Recall that a `Prop`-valued inductive type is a syntactic subsingleton if it has at most one constructor and all the arguments to the constructor are in `Prop`. Such types have large elimination, so they could be defined in `Type` or `Prop` without any trouble. The way inference has changed is that if a type is a syntactic subsingleton with exactly one constructor, and the constructor has at least one parameter/field, then the `inductive`/`structure` command will prefer creating a `Prop` instead of a `Type`. The upshot is that the `: Prop` in `structure S : Prop` is often no longer needed. (With @arthur-adjedj).
- #5842 and #5783 implement a feature where the `structure` command can now define recursive inductive types:

```
structure Tree where
  n : Nat
  children : Fin n → Tree

def Tree.size : Tree → Nat
  | {n, children} => Id.run do
    let mut s := 0
    for h : i in [0 : n] do
      s := s + (children ⟨i, h.2⟩).size
    pure s
```

- #5814 fixes a bug where Mathlib's `Type*` elaborator could lead to incorrect universe parameters with the `inductive` command.
- #3152 and #5844 fix bugs in default value processing for structure instance notation (with @arthur-adjedj).
- #5399 promotes instance synthesis order calculation failure from a soft error to a hard error.
- #5542 deprecates `:=` variants of `inductive` and `structure` (see breaking changes).

- **Application elaboration improvements**
  - #5671 makes `@[elab_as_elim]` require at least one discriminant, since otherwise there is no advantage to this alternative elaborator.
  - #5528 enables field notation in explicit mode. The syntax `@x.f` elaborates as `@S.f` with `x` supplied to the appropriate parameter.
  - #5692 modifies the dot notation resolution algorithm so that it can apply `CoeFun` instances. For example, Mathlib has `Multiset.card : Multiset α →+ Nat`, and now with `m : Multiset α`, the notation `m.card` resolves to `⇑Multiset.card m`.
  - #5658 fixes a bug where 'don't know how to synthesize implicit argument' errors might have the incorrect local context when the eta arguments feature is activated.
  - #5933 fixes a bug where `..` ellipses in patterns made use of optparams and autoparams.

- #5770 makes dot notation for structures resolve using *all* ancestors. Adds a *resolution order* for generalized field notation. This is the order of namespaces visited during resolution when trying to resolve names. The algorithm to compute a resolution order is the commonly used C3 linearization (used for example by Python), which when successful ensures that immediate parents' namespaces are considered before more distant ancestors' namespaces. By default we use a relaxed version of the algorithm that tolerates inconsistencies, but using `set_option structure.strictResolutionOrder true` makes inconsistent parent orderings into warnings.

- **Recursion and induction principles**
  - #5619 fixes functional induction principle generation to avoid over-eta-expanding in the preprocessing step.
  - #5766 fixes structural nested recursion so that it is not confused when a nested type appears first.
  - #5803 fixes a bug in functional induction principle generation when there are `let` bindings.
  - #5904 improves functional induction principle generation to unfold aux definitions more carefully.
  - #5850 refactors code for `Predefinition.Structural`.

- **Error messages**
  - #5276 fixes a bug in "type mismatch" errors that would structurally assign metavariables during the algorithm to expose differences.
  - #5919 makes "type mismatch" errors add type ascriptions to expose differences for numeric literals.
  - #5922 makes "type mismatch" errors expose differences in the bodies of functions and pi types.
  - #5888 improves the error message for invalid induction alternative names in `match` expressions (@josojo).
  - #5719 improves `calc` error messages.

- #5627 and #5663 improve the **`#eval` command** and introduce some new features.
  - Now results can be pretty printed if there is a `ToExpr` instance, which means **hoverable output**. If `ToExpr` fails, it then tries looking for a `Repr` or `ToString` instance like before. Setting `set_option eval.pp false` disables making use of `ToExpr` instances.
  - There is now **auto-derivation** of `Repr` instances, enabled with the `pp.derive.repr` option (default to **true**). For example:

```
inductive Baz
| a | b
```

```
#eval Baz.a
-- Baz.a
```

It simply does `deriving instance Repr for Baz` when there's no way to represent `Baz`.

- The option `eval.type` controls whether or not to include the type in the output. For now the default is false.
- Now expressions such as `#eval do return 2`, where monad is unknown, work. It tries unifying the monad with `CommandElabM`, `TermElabM`, or `IO`.
- The classes `Lean.Eval` and `Lean.MetaEval` have been removed. These each used to be responsible for adapting monads and printing results. Now the `MonadEval` class is responsible for adapting monads for evaluation (it is similar to `MonadLift`, but instances are allowed to use default data when initializing state), and representing results is handled through a separate process.
- Error messages about failed instance synthesis are now more precise. Once it detects that a `MonadEval` class applies, then the error message will be specific about missing `ToExpr`/`Repr`/`ToString` instances.
- Fixes bugs where evaluating `MetaM` and `CoreM` wouldn't collect log messages.
- Fixes a bug where `let rec` could not be used in `#eval`.

- `partial` definitions
  - #5780 improves the error message when `partial` fails to prove a type is inhabited. Add delta deriving.
  - #5821 gives `partial` inhabitation the ability to create local `Inhabited` instances from parameters.
- **New tactic configuration syntax.** The configuration syntax for all core tactics has been given an upgrade. Rather than `simp (config := { contextual := true, maxSteps := 22})`, one can now write `simp +contextual (maxSteps := 22)`. Tactic authors can migrate by switching from `(config)?` to `optConfig` in tactic syntaxes and potentially deleting `mkOptionalNode` in elaborators. #5883, #5898, #5928, and #5932. (Tactic authors, see breaking changes.)
- `simp` tactic
  - #5632 fixes the simpproc for `Fin` literals to reduce more consistently.
  - #5648 fixes a bug in `simpa ... using t` where metavariables in `t` were not properly accounted for, and also improves the type mismatch error.
  - #5838 fixes the docstring of `simp!` to actually talk about `simp!`.
  - #5870 adds support for `attribute [simp ←]` (note the reverse direction). This adds the reverse of a theorem as a global simp theorem.
- `decide` tactic

- #5665 adds `decide!` tactic for using kernel reduction (warning: this is renamed to `decide +kernel` in a future release).

- `bv_decide` tactic
  - #5714 adds inequality regression tests (@alexkeizer).
  - #5608 adds `bv_toNat` tag for `toNat_ofInt` (@bollu).
  - #5618 adds support for `at` in `ac_nf` and uses it in `bv_normalize` (@tobiasgrosser).
  - #5628 adds udiv support.
  - #5635 adds auxiliary bitblasters for negation and subtraction.
  - #5637 adds more `getLsbD` bitblaster theory.
  - #5652 adds umod support.
  - #5653 adds performance benchmark for modulo.
  - #5655 reduces error on `bv_check` to warning.
  - #5670 adds `~~~(-x)` support.
  - #5673 disables `ac_nf` by default.
  - #5675 fixes context tracking in `bv_decide` counter example.
  - #5676 adds an error when the LRAT proof is invalid.
  - #5781 introduces uninterpreted symbols everywhere.
  - #5823 adds `BitVec.sdiv` support.
  - #5852 adds `BitVec.ofBool` support.
  - #5855 adds `if` support.
  - #5869 adds support for all the SMTLIB BitVec divison/remainder operations.
  - #5886 adds embedded constraint substitution.
  - #5918 fixes loose mvars bug in `bv_normalize`.
  - Documentation:
    - #5636 adds remarks about multiplication.

- `conv` mode
  - #5861 improves the `congr` conv tactic to handle "over-applied" functions.
  - #5894 improves the `arg` conv tactic so that it can access more arguments and so that it can handle "over-applied" functions (it generates a specialized congruence lemma for the specific argument in question). Makes `arg 1` and `arg 2` apply to pi types in more situations. Adds negative indexing, for example `arg -2` is equivalent to the `lhs` tactic. Makes the `enter [...]` tactic show intermediate states like `rw`.

- **Other tactics**

- #4846 fixes a bug where `generalize ... at *` would apply to implementation details (@ymherklotz).
- #5730 upstreams the `classical` tactic combinator.
- #5815 improves the error message when trying to unfold a local hypothesis that is not a local definition.
- #5862 and #5863 change how `apply` and `simp` elaborate, making them not disable error recovery. This improves hovers and completions when the term has elaboration errors.

- `deriving` clauses
  - #5899 adds declaration ranges for delta-derived instances.
  - #5265 removes unused syntax in `deriving` clauses for providing arguments to deriving handlers (see breaking changes).
- #5065 upstreams and updates `#where`, a command that reports the current scope information.
- **Linters**
  - #5338 makes the unused variables linter ignore variables defined in tactics by default now, avoiding performance bottlenecks.
  - #5644 ensures that linters in general do not run on `#guard_msgs` itself.
- **Metaprogramming interface**
  - #5720 adds `pushGoal`/`pushGoals` and `popGoal` for manipulating the goal state. These are an alternative to `replaceMainGoal` and `getMainGoal`, and with them you don't need to worry about making sure nothing clears assigned metavariables from the goal list between assigning the main goal and using `replaceMainGoal`. Modifies `closeMainGoalUsing`, which is like a `TacticM` version of `liftMetaTactic`. Now the callback is run in a context where the main goal is removed from the goal list, and the callback is free to modify the goal list. Furthermore, the `checkUnassigned` argument has been replaced with `checkNewUnassigned`, which checks whether the value assigned to the goal has any *new* metavariables, relative to the start of execution of the callback. Modifies `withCollectingNewGoalsFrom` to take the `parentTag` argument explicitly rather than indirectly via `getMainTag`. Modifies `elabTermWithHoles` to optionally take `parentTag?`.
  - #5563 fixes `getFunInfo` and `inferType` to use `withAtLeastTransparency` rather than `withTransparency`.
  - #5679 fixes `RecursorVal.getInduct` to return the name of major argument's type. This makes "structure eta" work for nested inductives.
  - #5681 removes unused `mkRecursorInfoForKernelRec`.
  - #5686 makes discrimination trees index the domains of foralls, for better performance of the simplify and type class search.

- #5760 adds `Lean.Expr.name?` recognizer for `Name` expressions.
- #5800 modifies `liftCommandElabM` to preserve more state, fixing an issue where using it would drop messages.
- #5857 makes it possible to use dot notation in `m!` strings, for example `m!"{.ofConstName n}"`.
- #5841 and #5853 record the complete list of `structure` parents in the `StructureInfo` environment extension.

- **Other fixes or improvements**
  - #5566 fixes a bug introduced in #4781 where heartbeat exceptions were no longer being handled properly. Now such exceptions are tagged with `runtime.maxHeartbeats` (@eric-wieser).
  - #5708 modifies the proof objects produced by the proof-by-reflection tactics `ac_nf0` and `simp_arith` so that the kernel is less prone to reducing expensive atoms.
  - #5768 adds a `#version` command that prints Lean's version information.
  - #5822 fixes elaborator algorithms to match kernel algorithms for primitive projections (`Expr.proj`).
  - #5811 improves the docstring for the `rwa` tactic.

Language server, widgets, and IDE extensions

- #5224 fixes `WorkspaceClientCapabilities` to make `applyEdit` optional, in accordance with the LSP specification (@pzread).
- #5340 fixes a server deadlock when shutting down the language server and a desync between client and language server after a file worker crash.
- #5560 makes `initialize` and `builtin_initialize` participate in the call hierarchy and other requests.
- #5650 makes references in attributes participate in the call hierarchy and other requests.
- #5666 add auto-completion in tactic blocks without having to type the first character of the tactic, and adds tactic completion docs to tactic auto-completion items.
- #5677 fixes several cases where goal states were not displayed in certain text cursor positions.
- #5707 indicates deprecations in auto-completion items.
- #5736, #5752, #5763, #5802, and #5805 fix various performance issues in the language server.
- #5801 distinguishes theorem auto-completions from non-theorem auto-completions.

Pretty printing

- #5640 fixes a bug where goal states in messages might print newlines as spaces.

- #5643 adds option `pp.mvars.delayed` (default false), which when false causes delayed assignment metavariables to pretty print with what they are assigned to. Now `fun x : Nat => ?a` pretty prints as `fun x : Nat => ?a` rather than `fun x ↦ ?m.7 x`.
- #5711 adds options `pp.mvars.anonymous` and `pp.mvars.levels`, which when false respectively cause expression metavariables and level metavariables to pretty print as `?_`.
- #5710 adjusts the ⋯ elaboration warning to mention `pp.maxSteps`.
- #5759 fixes the app unexpander for `sorryAx`.
- #5827 improves accuracy of binder names in the signature pretty printer (like in output of `#check`). Also fixes the issue where consecutive hygienic names pretty print without a space separating them, so we now have `(x✝ y✝ : Nat)` rather than `(x✝y✝ : Nat)`.
- #5830 makes sure all the core delaborators respond to `pp.explicit` when appropriate.
- #5639 makes sure name literals use escaping when pretty printing.
- #5854 adds delaborators for `<|>`, `<*>`, `>>=`, `<*`, and `*>`.

Library

- `Array`
  - #5687 deprecates `Array.data`.
  - #5705 uses a better default value for `Array.swapAt!`.
  - #5748 moves `Array.mapIdx` lemmas to a new file.
  - #5749 simplifies signature of `Array.mapIdx`.
  - #5758 upstreams `Array.reduceOption`.
  - #5786 adds simp lemmas for `Array.isEqv` and `BEq`.
  - #5796 renames `Array.shrink` to `Array.take`, and relates it to `List.take`.
  - #5798 upstreams `List.modify`, adds lemmas, relates to `Array.modify`.
  - #5799 relates `Array.forIn` and `List.forIn`.
  - #5833 adds `Array.forIn'`, and relates to `List`.
  - #5848 fixes deprecations in `Init.Data.Array.Basic` to not recommend the deprecated constant.
  - #5895 adds `LawfulBEq (Array α) ↔ LawfulBEq α`.
  - #5896 moves `@[simp]` from `back_eq_back?` to `back_push`.
  - #5897 renames `Array.back` to `back!`.
- `List`
  - #5605 removes `List.redLength`.
  - #5696 upstreams `List.mapIdx` and adds lemmas.
  - #5697 upstreams `List.foldxM_map`.

- #5701 renames `List.join` to `List.flatten`.
- #5703 upstreams `List.sum`.
- #5706 marks `prefix_append_right_inj` as a simp lemma.
- #5716 fixes `List.drop_drop` addition order.
- #5731 renames `List.bind` and `Array.concatMap` to `flatMap`.
- #5732 renames `List.pure` to `List.singleton`.
- #5742 upstreams `ne_of_mem_of_not_mem`.
- #5743 upstreams `ne_of_apply_ne`.
- #5816 adds more `List.modify` lemmas.
- #5879 renames `List.groupBy` to `splitBy`.
- #5913 relates `for` loops over `List` with `foldlM`.

- `Nat`
  - #5694 removes `instBEqNat`, which is redundant with `instBEqOfDecidableEq` but not defeq.
  - #5746 deprecates `Nat.sum`.
  - #5785 adds `Nat.forall_lt_succ` and variants.
- Fixed width integers
  - #5323 redefine unsigned fixed width integers in terms of `BitVec`.
  - #5735 adds `UIntX.[val_ofNat, toBitVec_ofNat]`.
  - #5790 defines `Int8`.
  - #5901 removes native code for `UInt8.modn`.
- `BitVec`
  - #5604 completes `BitVec.[getMsbD|getLsbD|msb]` for shifts (@luisacicolini).
  - #5609 adds lemmas for division when denominator is zero (@bollu).
  - #5620 documents Bitblasting (@bollu)
  - #5623 moves `BitVec.udiv/umod/sdiv/smod` after `add/sub/mul/lt` (@tobiasgrosser).
  - #5645 defines `udiv` normal form to be `/`, resp. `umod` and `%` (@bollu).
  - #5646 adds lemmas about arithmetic inequalities (@bollu).
  - #5680 expands relationship with `toFin` (@tobiasgrosser).
  - #5691 adds `BitVec.(getMSbD, msb)_(add, sub)` and `BitVec.getLsbD_sub` (@luisacicolini).
  - #5712 adds `BitVec.[udiv|umod]_[zero|one|self]` (@tobiasgrosser).

- #5718 adds `BitVec.sdiv_[zero|one|self]` (@tobiasgrosser).
- #5721 adds `BitVec.(msb, getMsbD, getLsbD)_(neg, abs)` (@luisacicolini).
- #5772 adds `BitVec.toInt_sub`, simplifies `BitVec.toInt_neg` (@tobiasgrosser).
- #5778 prove that `intMin` the smallest signed bitvector (@alexkeizer).
- #5851 adds `(msb, getMsbD)_twoPow` (@luisacicolini).
- #5858 adds `BitVec.[zero_ushiftRight|zero_sshiftRight|zero_mul]` and cleans up BVDecide (@tobiasgrosser).
- #5865 adds `BitVec.(msb, getMsbD)_concat` (@luisacicolini).
- #5881 adds `Hashable (BitVec n)`

- `String`/`Char`
  - #5728 upstreams `String.dropPrefix?`.
  - #5745 changes `String.dropPrefix?` signature.
  - #5747 adds `Hashable Char` instance
- `HashMap`
  - #5880 adds interim implementation of `HashMap.modify`/`alter`
- **Other**
  - #5704 removes `@[simp]` from `Option.isSome_eq_isSome`.
  - #5739 upstreams material on `Prod`.
  - #5740 moves `Antisymm` to `Std.Antisymm`.
  - #5741 upstreams basic material on `Sum`.
  - #5756 adds `Nat.log2_two_pow` (@spinylobster).
  - #5892 removes duplicated `ForIn` instances.
  - #5900 removes `@[simp]` from `Sum.forall` and `Sum.exists`.
  - #5812 removes redundant `Decidable` assumptions (@FR-vdash-bot).

Compiler, runtime, and FFI

- #5685 fixes help message flags, removes the `-f` flag and adds the `-g` flag (@James-Oswald).
- #5930 adds `--short-version` (`-V`) option to display short version (@juhp).
- #5144 switches all 64-bit platforms over to consistently using GMP for bignum arithmetic.
- #5753 raises the minimum supported Windows version to Windows 10 1903 (released May 2019).

Lake

- #5715 changes `lake new math` to use `autoImplicit false` (@eric-wieser).

- #5688 makes `Lake` not create core aliases in the `Lake` namespace.
- #5924 adds a `text` option for `buildFile*` utilities.
- #5789 makes `lake init` not `git init` when inside git work tree (@haoxins).
- #5684 has Lake update a package's `lean-toolchain` file on `lake update` if it finds the package's direct dependencies use a newer compatible toolchain. To skip this step, use the `--keep-toolchain` CLI option. (See breaking changes.)
- #6218 makes Lake no longer automatically fetch GitHub cloud releases if the package build directory is already present (mirroring the behavior of the Reservoir cache). This prevents the cache from clobbering existing prebuilt artifacts. Users can still manually fetch the cache and clobber the build directory by running `lake build <pkg>:release`.
- #6231 improves the errors Lake produces when it fails to fetch a dependency from Reservoir. If the package is not indexed, it will produce a suggestion about how to require it from GitHub.

Documentation

- #5617 fixes MSYS2 build instructions.
- #5725 points out that `OfScientific` is called with raw literals (@eric-wieser).
- #5794 adds a stub for application ellipsis notation (@eric-wieser).

Breaking changes

- The syntax for providing arguments to deriving handlers has been removed, which was not used by any major Lean projects in the ecosystem. As a result, the `applyDerivingHandlers` now takes one fewer argument, `registerDerivingHandlerWithArgs` is now simply `registerDerivingHandler`, `DerivingHandler` no longer includes the unused parameter, and `DerivingHandlerNoArgs` has been deprecated. To migrate code, delete the unused none argument and use `registerDerivingHandler` and `DerivingHandler`. (#5265)
- The minimum supported Windows version has been raised to Windows 10 1903, released May 2019. (#5753)
- The `--lean` CLI option for `lake` was removed. Use the `LEAN` environment variable instead. (#5684)
- The `inductive ... :=`, `structure ... :=`, and `class ... :=` syntaxes have been deprecated in favor of the `... where` variants. The old syntax produces a warning, controlled by the `linter.deprecated` option. (#5542)
- The generated tactic configuration elaborators now land in `TacticM` to make use of the current recovery state. Commands that wish to elaborate configurations should now use `declare_command_config_elab` instead of `declare_config_elab` to get an elaborator landing in `CommandElabM`. Syntaxes should migrate to `optConfig` instead of `(config)?`, but the elaborators are reverse compatible. (#5883)

Lean 4.13.0 (2024-11-01)

Full Changelog: https://github.com/leanprover/lean4/compare/v4.12.0...v4.13.0

Language features, tactics, and metaprograms

- `structure` command
  - #5511 allows structure parents to be type synonyms.
  - #5531 allows default values for structure fields to be noncomputable.
- `rfl` and `apply_rfl` tactics
  - #3714, #3718 improve the `rfl` tactic and give better error messages.
  - #3772 makes `rfl` no longer use kernel defeq for ground terms.
  - #5329 tags `Iff.refl` with `@[refl]` (@Parcly-Taxel)
  - #5359 ensures that the `rfl` tactic tries `Iff.rfl` (@Parcly-Taxel)
- `unfold` tactic
  - #4834 let `unfold` do zeta-delta reduction of local definitions, incorporating functionality of the Mathlib `unfold_let` tactic.
- `omega` tactic
  - #5382 fixes spurious error in #5315
  - #5523 supports `Int.toNat`
- `simp` tactic
  - #5479 lets `simp` apply rules with higher-order patterns.
- `induction` tactic
  - #5494 fixes `induction`'s "pre-tactic" block to always be indented, avoiding unintended uses of it.
- `ac_nf` tactic
  - #5524 adds `ac_nf`, a counterpart to `ac_rfl`, for normalizing expressions with respect to associativity and commutativity. Tests it with `BitVec` expressions.
- `bv_decide`
  - #5211 makes `extractLsb'` the primitive `bv_decide` understands, rather than `extractLsb` (@alexkeizer)
  - #5365 adds `bv_decide` diagnoses.
  - #5375 adds `bv_decide` normalization rules for `ofBool (a.getLsbD i)` and `ofBool a[i]` (@alexkeizer)

- #5423 enhances the rewriting rules of `bv_decide`
- #5433 presents the `bv_decide` counterexample at the API
- #5484 handles `BitVec.ofNat` with `Nat` fvars in `bv_decide`
- #5506, #5507 add `bv_normalize` rules.
- #5568 generalize the `bv_normalize` pipeline to support more general preprocessing passes
- #5573 gets `bv_normalize` up-to-date with the current `BitVec` rewrites
- Cleanups: #5408, #5493, #5578

- Elaboration improvements
  - #5266 preserve order of overapplied arguments in `elab_as_elim` procedure.
  - #5510 generalizes `elab_as_elim` to allow arbitrary motive applications.
  - #5283, #5512 refine how named arguments suppress explicit arguments. Breaking change: some previously omitted explicit arguments may need explicit _ arguments now.
  - #5376 modifies projection instance binder info for instances, making parameters that are instance implicit in the type be implicit.
  - #5402 localizes universe metavariable errors to `let` bindings and `fun` binders if possible. Makes "cannot synthesize metavariable" errors take precedence over unsolved universe level errors.
  - #5419 must not reduce `ite` in the discriminant of `match`-expression when reducibility setting is `.reducible`
  - #5474 have autoparams report parameter/field on failure
  - #5530 makes automatic instance names about types with hygienic names be hygienic.
- Deriving handlers
  - #5432 makes `Repr` deriving instance handle explicit type parameters
- Functional induction
  - #5364 adds more equalities in context, more careful cleanup.
- Linters
  - #5335 fixes the unused variables linter complaining about match/tactic combinations
  - #5337 fixes the unused variables linter complaining about some wildcard patterns
- Other fixes
  - #4768 fixes a parse error when `..` appears with a `.` on the next line
- Metaprogramming
  - #3090 handles level parameters in `Meta.evalExpr` (@eric-wieser)

- #5401 instance for `Inhabited (TacticM α)` (@alexkeizer)
- #5412 expose Kernel.check for debugging purposes
- #5556 improves the "invalid projection" type inference error in `inferType`.
- #5587 allows `MVarId.assertHypotheses` to set `BinderInfo` and `LocalDeclKind`.
- #5588 adds `MVarId.tryClearMany'`, a variant of `MVarId.tryClearMany`.

Language server, widgets, and IDE extensions

- #5205 decreases the latency of auto-completion in tactic blocks.
- #5237 fixes symbol occurrence highlighting in VS Code not highlighting occurrences when moving the text cursor into the identifier from the right.
- #5257 fixes several instances of incorrect auto-completions being reported.
- #5299 allows auto-completion to report completions for global identifiers when the elaborator fails to provide context-specific auto-completions.
- #5312 fixes the server breaking when changing whitespace after the module header.
- #5322 fixes several instances of auto-completion reporting non-existent namespaces.
- #5428 makes sure to always report some recent file range as progress when waiting for elaboration.

Pretty printing

- #4979 make pretty printer escape identifiers that are tokens.
- #5389 makes formatter use the current token table.
- #5513 use breakable instead of unbreakable whitespace when formatting tokens.

Library

- #5222 reduces allocations in `Json.compress`.
- #5231 upstreams `Zero` and `NeZero`
- #5292 refactors `Lean.Elab.Deriving.FromToJson` (@arthur-adjedj)
- #5415 implements `Repr Empty` (@TomasPuverle)
- #5421 implements `To/FromJSON Empty` (@TomasPuverle)
- Logic
  - #5263 allows simplifying `dite_not`/`decide_not` with only `Decidable (¬p)`.
  - #5268 fixes binders on `ite_eq_left_iff`
  - #5284 turns off `Inhabited (Sum α β)` instances
  - #5355 adds simp lemmas for `LawfulBEq`

- #5374 add `Nonempty` instances for products, allowing more `partial` functions to elaborate successfully
- #5447 updates Pi instance names
- #5454 makes some instance arguments implicit
- #5456 adds `heq_comm`
- #5529 moves `@[simp]` from `exists_prop'` to `exists_prop`

- `Bool`
  - #5228 fills gaps in Bool lemmas
  - #5332 adds notation `^^` for Bool.xor
  - #5351 removes `_root_.and` (and or/not/xor) and instead exports/uses `Bool.and` (etc.).
- `BitVec`
  - #5240 removes BitVec simps with complicated RHS
  - #5247 `BitVec.getElem_zeroExtend`
  - #5248 simp lemmas for BitVec, improving confluence
  - #5249 removes `@[simp]` from some BitVec lemmas
  - #5252 changes `BitVec.intMin/Max` from abbrev to def
  - #5278 adds `BitVec.getElem_truncate` (@tobiasgrosser)
  - #5281 adds udiv/umod bitblasting for `bv_decide` (@bollu)
  - #5297 `BitVec` unsigned order theoretic results
  - #5313 adds more basic BitVec ordering theory for UInt
  - #5314 adds `toNat_sub_of_le` (@bollu)
  - #5357 adds `BitVec.truncate` lemmas
  - #5358 introduces `BitVec.setWidth` to unify zeroExtend and truncate (@tobiasgrosser)
  - #5361 some BitVec GetElem lemmas
  - #5385 adds `BitVec.ofBool_[and|or|xor]_ofBool` theorems (@tobiasgrosser)
  - #5404 more of `BitVec.getElem_*` (@tobiasgrosser)
  - #5410 BitVec analogues of `Nat.{mul_two, two_mul, mul_succ, succ_mul}` (@bollu)
  - #5411 `BitVec.toNat_{add,sub,mul_of_lt}` for BitVector non-overflow reasoning (@bollu)
  - #5413 adds `_self`, `_zero`, and `_allOnes` for `BitVec.[and|or|xor]` (@tobiasgrosser)

- #5416 adds LawCommIdentity + IdempotentOp for `BitVec.[and|or|xor]` (@tobiasgrosser)
- #5418 decidable quantifers for BitVec
- #5450 adds `BitVec.toInt_[intMin|neg|neg_of_ne_intMin]` (@tobiasgrosser)
- #5459 missing BitVec lemmas
- #5469 adds `BitVec.[not_not, allOnes_shiftLeft_or_shiftLeft, allOnes_shiftLeft_and_shiftLeft]` (@luisacicolini)
- #5478 adds `BitVec.(shiftLeft_add_distrib, shiftLeft_ushiftRight)` (@luisacicolini)
- #5487 adds `sdiv_eq`, `smod_eq` to allow `sdiv`/`smod` bitblasting (@bollu)
- #5491 adds `BitVec.toNat_[abs|sdiv|smod]` (@tobiasgrosser)
- #5492 `BitVec.(not_sshiftRight, not_sshiftRight_not, getMsb_not, msb_not)` (@luisacicolini)
- #5499 `BitVec.Lemmas` - drop non-terminal simps (@tobiasgrosser)
- #5505 unsimps `BitVec.divRec_succ'`
- #5508 adds `BitVec.getElem_[add|add_add_bool|mul|rotateLeft|rotateRight…` (@tobiasgrosser)
- #5554 adds `Bitvec.[add, sub, mul]_eq_xor` and `width_one_cases` (@luisacicolini)

- `List`
  - #5242 improve naming for `List.mergeSort` lemmas
  - #5302 provide `mergeSort` comparator autoParam
  - #5373 fix name of `List.length_mergeSort`
  - #5377 upstream `map_mergeSort`
  - #5378 modify signature of lemmas about `mergeSort`
  - #5245 avoid importing `List.Basic` without List.Impl
  - #5260 review of List API
  - #5264 review of List API
  - #5269 remove HashMap's duplicated Pairwise and Sublist
  - #5271 remove @[simp] from `List.head_mem` and similar
  - #5273 lemmas about `List.attach`
  - #5275 reverse direction of `List.tail_map`
  - #5277 more `List.attach` lemmas

- #5285 `List.count` lemmas
- #5287 use boolean predicates in `List.filter`
- #5289 `List.mem_ite_nil_left` and analogues
- #5293 cleanup of `List.findIdx` / `List.take` lemmas
- #5294 switch primes on `List.getElem_take`
- #5300 more `List.findIdx` theorems
- #5310 fix `List.all`/`any` lemmas
- #5311 fix `List.countP` lemmas
- #5316 `List.tail` lemma
- #5331 fix implicitness of `List.getElem_mem`
- #5350 `List.replicate` lemmas
- #5352 `List.attachWith` lemmas
- #5353 `List.head_mem_head?`
- #5360 lemmas about `List.tail`
- #5391 review of `List.erase` / `List.find` lemmas
- #5392 `List.fold` / `attach` lemmas
- #5393 `List.fold` relators
- #5394 lemmas about `List.maximum?`
- #5403 theorems about `List.toArray`
- #5405 reverse direction of `List.set_map`
- #5448 add lemmas about `List.IsPrefix` (@Command-Master)
- #5460 missing `List.set_replicate_self`
- #5518 rename `List.maximum?` to `max?`
- #5519 upstream `List.fold` lemmas
- #5520 restore `@[simp]` on `List.getElem_mem` etc.
- #5521 List simp fixes
- #5550 `List.unattach` and simp lemmas
- #5594 induction-friendly `List.min?_cons`

- `Array`
  - #5246 cleanup imports of Array.Lemmas
  - #5255 split Init.Data.Array.Lemmas for better bootstrapping
  - #5288 rename `Array.data` to `Array.toList`
  - #5303 cleanup of `List.getElem_append` variants

- #5304 `Array.not_mem_empty`
- #5400 reorganization in Array/Basic
- #5420 make `Array` functions either semireducible or use structural recursion
- #5422 refactor `DecidableEq (Array α)`
- #5452 refactor of Array
- #5458 cleanup of Array docstrings after refactor
- #5461 restore `@[simp]` on `Array.swapAt!_def`
- #5465 improve Array GetElem lemmas
- #5466 `Array.foldX` lemmas
- #5472 @[simp] lemmas about `List.toArray`
- #5485 reverse simp direction for `toArray_concat`
- #5514 `Array.eraseReps`
- #5515 upstream `Array.qsortOrd`
- #5516 upstream `Subarray.empty`
- #5526 fix name of `Array.length_toList`
- #5527 reduce use of deprecated lemmas in Array
- #5534 cleanup of Array GetElem lemmas
- #5536 fix `Array.modify` lemmas
- #5551 upstream `Array.flatten` lemmas
- #5552 switch obvious cases of array "bang" `[]!` indexing to rely on hypothesis (@TomasPuverle)
- #5577 add missing simp to `Array.size_feraseIdx`
- #5586 `Array/Option.unattach`

- `Option`
  - #5272 remove @[simp] from `Option.pmap/pbind` and add simp lemmas
  - #5307 restoring Option simp confluence
  - #5354 remove @[simp] from `Option.bind_map`
  - #5532 `Option.attach`
  - #5539 fix explicitness of `Option.mem_toList`
- `Nat`
  - #5241 add @[simp] to `Nat.add_eq_zero_iff`
  - #5261 Nat bitwise lemmas
  - #5262 `Nat.testBit_add_one` should not be a global simp lemma

- #5267 protect some Nat bitwise theorems
- #5305 rename Nat bitwise lemmas
- #5306 add `Nat.self_sub_mod` lemma
- #5503 restore @[simp] to upstreamed `Nat.lt_off_iff`

- `Int`
  - #5301 rename `Int.div/mod` to `Int.tdiv/tmod`
  - #5320 add `ediv_nonneg_of_nonpos_of_nonpos` to DivModLemmas (@sakehl)
- `Fin`
  - #5250 missing lemma about `Fin.ofNat'`
  - #5356 `Fin.ofNat'` uses `NeZero`
  - #5379 remove some @[simp]s from Fin lemmas
  - #5380 missing Fin @[simp] lemmas
- `HashMap`
  - #5244 `(DHashMap|HashMap|HashSet).(getKey?|getKey|getKey!|getKeyD)`
  - #5362 remove the last use of `Lean.(HashSet|HashMap)`
  - #5369 `HashSet.ofArray`
  - #5370 `HashSet.partition`
  - #5581 `Singleton/Insert/Union` instances for `HashMap/Set`
  - #5582 `HashSet.all/any`
  - #5590 adding `Insert/Singleton/Union` instances for `HashMap/Set.Raw`
  - #5591 `HashSet.Raw.all/any`
- `Monads`
  - #5463 upstream some monad lemmas
  - #5464 adjust simp attributes on monad lemmas
  - #5522 more monadic simp lemmas
- Simp lemma cleanup
  - #5251 remove redundant simp annotations
  - #5253 remove Int simp lemmas that can't fire
  - #5254 variables appearing on both sides of an iff should be implicit
  - #5381 cleaning up redundant simp lemmas

Compiler, runtime, and FFI

- #4685 fixes a typo in the C `run_new_frontend` signature

- #4729 has IR checker suggest using `noncomputable`
- #5143 adds a shared library for Lake
- #5437 removes (syntactically) duplicate imports (@euprunin)
- #5462 updates `src/lake/lakefile.toml` to the adjusted Lake build process
- #5541 removes new shared libs before build to better support Windows
- #5558 make `lean.h` compile with MSVC (@kant2002)
- #5564 removes non-conforming size-0 arrays (@eric-wieser)

Lake

- Reservoir build cache. Lake will now attempt to fetch a pre-built copy of the package from Reservoir before building it. This is only enabled for packages in the leanprover or leanprover-community organizations on versions indexed by Reservoir. Users can force Lake to build packages from the source by passing --no-cache on the CLI or by setting the LAKE_NO_CACHE environment variable to true. #5486, #5572, #5583, #5600, #5641, #5642.
- #5504 lake new and lake init now produce TOML configurations by default.
- #5878 fixes a serious issue where Lake would delete path dependencies when attempting to cleanup a dependency required with an incorrect name.
- **Breaking changes**
  - #5641 A Lake build of target within a package will no longer build a package's dependencies package-level extra target dependencies. At the technical level, a package's extraDep facet no longer transitively builds its dependencies' extraDep facets (which include their extraDepTargets).

Documentation fixes

- #3918 `@[builtin_doc]` attribute (@digama0)
- #4305 explains the borrow syntax (@eric-wieser)
- #5349 adds documentation for `groupBy.loop` (@vihdzp)
- #5473 fixes typo in `BitVec.mul` docstring (@llllvvuu)
- #5476 fixes typos in `Lean.MetavarContext`
- #5481 removes mention of `Lean.withSeconds` (@alexkeizer)
- #5497 updates documentation and tests for `toUIntX` functions (@TomasPuverle)
- #5087 mentions that `inferType` does not ensure type correctness
- Many fixes to spelling across the doc-strings, (@euprunin): #5425 #5426 #5427 #5430 #5431 #5434 #5435 #5436 #5438 #5439 #5440 #5599

Changes to CI

- #5343 allows addition of `release-ci` label via comment (@thorimur)

- #5344 sets check level correctly during workflow (@thorimur)
- #5444 Mathlib's `lean-pr-testing-NNNN` branches should use Batteries' `lean-pr-testing-NNNN` branches
- #5489 commit `lake-manifest.json` when updating `lean-pr-testing` branches
- #5490 use separate secrets for commenting and branching in `pr-release.yml`

Lean 4.12.0 (2024-10-01)

Language features, tactics, and metaprograms

- `bv_decide` tactic. This release introduces a new tactic for proving goals involving `BitVec` and `Bool`. It reduces the goal to a SAT instance that is refuted by an external solver, and the resulting LRAT proof is checked in Lean. This is used to synthesize a proof of the goal by reflection. As this process uses verified algorithms, proofs generated by this tactic use `Lean.ofReduceBool`, so this tactic includes the Lean compiler as part of the trusted code base. The external solver CaDiCaL is included with Lean and does not need to be installed separately to make use of `bv_decide`.

  For example, we can use `bv_decide` to verify that a bit twiddling formula leaves at most one bit set:

  ```
  def popcount (x : BitVec 64) : BitVec 64 :=
    let rec go (x pop : BitVec 64) : Nat → BitVec 64
      | 0 => pop
      | n + 1 => go (x >>> 2) (pop + (x &&& 1)) n
    go x 0 64

  example (x : BitVec 64) : popcount ((x &&& (x - 1)) ^^^ x) ≤ 1 :=
  by
    simp only [popcount, popcount.go]
    bv_decide
  ```

  When the external solver fails to refute the SAT instance generated by `bv_decide`, it can report a counterexample:

  ```
  /--
  error: The prover found a counterexample, consider the following
  assignment:
  x = 0xffffffffffffffff#64
  -/
  #guard_msgs in
  example (x : BitVec 64) : x < x + 1 := by
    bv_decide
  ```

  See `Lean.Elab.Tactic.BVDecide` for a more detailed overview, and look in `tests/lean/run/bv_*` for examples.

  #5013, #5074, #5100, #5113, #5137, #5203, #5212, #5220.

- `simp` tactic
  - #4988 fixes a panic in the `reducePow` simproc.
  - #5071 exposes the `index` option to the `dsimp` tactic, introduced to `simp` in #4202.
  - #5159 fixes a panic at `Fin.isValue` simproc.
  - #5167 and #5175 rename the `simpCtorEq` simproc to `reduceCtorEq` and makes it optional. (See breaking changes.)
  - #5187 ensures `reduceCtorEq` is enabled in the `norm_cast` tactic.
  - #5073 modifies the simp debug trace messages to tag with "dpre" and "dpost" instead of "pre" and "post" when in definitional rewrite mode. #5054 explains the `reduce` steps for `trace.Debug.Meta.Tactic.simp` trace messages.
- `ext` tactic
  - #4996 reduces default maximum iteration depth from 1000000 to 100.
- `induction` tactic
  - #5117 fixes a bug where `let` bindings in minor premises wouldn't be counted correctly.
- `omega` tactic
  - #5157 fixes a panic.
- `conv` tactic
  - #5149 improves `arg n` to handle subsingleton instance arguments.
- #5044 upstreams the `#time` command.
- #5079 makes `#check` and `#reduce` typecheck the elaborated terms.
- **Incrementality**
  - #4974 fixes regression where we would not interrupt elaboration of previous document versions.
  - #5004 fixes a performance regression.
  - #5001 disables incremental body elaboration in presence of `where` clauses in declarations.
  - #5018 enables infotrees on the command line for ilean generation.
  - #5040 and #5056 improve performance of info trees.
  - #5090 disables incrementality in the `case .. | ..` tactic.
  - #5312 fixes a bug where changing whitespace after the module header could break subsequent commands.
- **Definitions**

- #5016 and #5066 add `clean_wf` tactic to clean up tactic state in `decreasing_by`. This can be disabled with `set_option debug.rawDecreasingByGoal false`.
- #5055 unifies equational theorems between structural and well-founded recursion.
- #5041 allows mutually recursive functions to use different parameter names among the "fixed parameter prefix"
- #4154 and #5109 add fine-grained equational lemmas for non-recursive functions. See breaking changes.
- #5129 unifies equation lemmas for recursive and non-recursive definitions. The `backward.eqns.deepRecursiveSplit` option can be set to `false` to get the old behavior. See breaking changes.
- #5141 adds `f.eq_unfold` lemmas. Now Lean produces the following zoo of rewrite rules:

  ```
  Option.map.eq_1      : Option.map f none = none
  Option.map.eq_2      : Option.map f (some x) = some (f x)
  Option.map.eq_def    : Option.map f p = match o with | none
  => none | (some x) => some (f x)
  Option.map.eq_unfold : Option.map = fun f p => match o with |
  none => none | (some x) => some (f x)
  ```

  The `f.eq_unfold` variant is especially useful to rewrite with `rw` under binders.
- #5136 fixes bugs in recursion over predicates.

- **Variable inclusion**
  - #5206 documents that `include` currently only applies to theorems.
- **Elaboration**
  - #4926 fixes a bug where autoparam errors were associated to an incorrect source position.
  - #4833 fixes an issue where cdot anonymous functions (e.g. `(· + ·)`) would not handle ambiguous notation correctly. Numbers the parameters, making this example expand as `fun x1 x2 => x1 + x2` rather than `fun x x_1 => x + x_1`.
  - #5037 improves strength of the tactic that proves array indexing is in bounds.
  - #5119 fixes a bug in the tactic that proves indexing is in bounds where it could loop in the presence of mvars.
  - #5072 makes the structure type clickable in "not a field of structure" errors for structure instance notation.
  - #4717 fixes a bug where mutual `inductive` commands could create terms that the kernel rejects.
  - #5142 fixes a bug where `variable` could fail when mixing binder updates and declarations.
- **Other fixes or improvements**

- #5118 changes the definition of the `syntheticHole` parser so that hovering over _ in ?_ gives the docstring for synthetic holes.
- #5173 uses the emoji variant selector for ✅,❌,💥 in messages, improving fonts selection.
- #5183 fixes a bug in `rename_i` where implementation detail hypotheses could be renamed.

Language server, widgets, and IDE extensions

- #4821 resolves two language server bugs that especially affect Windows users. (1) Editing the header could result in the watchdog not correctly restarting the file worker, which would lead to the file seemingly being processed forever. (2) On an especially slow Windows machine, we found that starting the language server would sometimes not succeed at all. This PR also resolves an issue where we would not correctly emit messages that we received while the file worker is being restarted to the corresponding file worker after the restart.
- #5006 updates the user widget manual.
- #5193 updates the quickstart guide with the new display name for the Lean 4 extension ("Lean 4").
- #5185 fixes a bug where over time "import out of date" messages would accumulate.
- #4900 improves ilean loading performance by about a factor of two. Optimizes the JSON parser and the conversion from JSON to Lean data structures; see PR description for details.
- **Other fixes or improvements**
  - #5031 localizes an instance in `Lsp.Diagnostics`.

Pretty printing

- #4976 introduces `@[app_delab]`, a macro for creating delaborators for particular constants. The `@[app_delab ident]` syntax resolves `ident` to its constant name `name` and then expands to `@[delab app.name]`.
- #4982 fixes a bug where the pretty printer assumed structure projections were type correct (such terms can appear in type mismatch errors). Improves hoverability of `#print` output for structures.
- #5218 and #5239 add `pp.exprSizes` debugging option. When true, each pretty printed expression is prefixed with `[size a/b/c]`, where `a` is the size without sharing, `b` is the actual size, and `c` is the size with the maximum possible sharing.

Library

- #5020 swaps the parameters to `Membership.mem`. A purpose of this change is to make set-like `CoeSort` coercions to refer to the eta-expanded function `fun x => Membership.mem s x`, which can reduce in many computations. Another is that having the `s` argument first leads to better discrimination tree keys. (See breaking changes.)

- `Array`
  - #4970 adds `@[ext]` attribute to `Array.ext`.
  - #4957 deprecates `Array.get_modify`.
- `List`
  - #4995 upstreams `List.findIdx` lemmas.
  - #5029, #5048 and #5132 add `List.Sublist` lemmas, some upstreamed. #5077 fixes implicitness in refl/rfl lemma binders. add `List.Sublist` theorems.
  - #5047 upstreams `List.Pairwise` lemmas.
  - #5053, #5124, and #5161 add `List.find?/findSome?/findIdx?` theorems.
  - #5039 adds `List.foldlRecOn` and `List.foldrRecOn` recursion principles to prove things about `List.foldl` and `List.foldr`.
  - #5069 upstreams `List.Perm`.
  - #5092 and #5107 add `List.mergeSort` and a fast `@[csimp]` implementation.
  - #5103 makes the simp lemmas for `List.subset` more aggressive.
  - #5106 changes the statement of `List.getLast?_cons`.
  - #5123 and #5158 add `List.range` and `List.iota` lemmas.
  - #5130 adds `List.join` lemmas.
  - #5131 adds `List.append` lemmas.
  - #5152 adds `List.erase(|P|Idx)` lemmas.
  - #5127 makes miscellaneous lemma updates.
  - #5153 and #5160 add lemmas about `List.attach` and `List.pmap`.
  - #5164, #5177, and #5215 add `List.find?` and `List.range'/range/iota` lemmas.
  - #5196 adds `List.Pairwise_erase` and related lemmas.
  - #5151 and #5163 improve confluence of `List` simp lemmas. #5105 and #5102 adjust `List` simp lemmas.
  - #5178 removes `List.getLast_eq_iff_getLast_eq_some` as a simp lemma.
  - #5210 reverses the meaning of `List.getElem_drop` and `List.getElem_drop'`.
  - #5214 moves `@[csimp]` lemmas earlier where possible.
- `Nat` and `Int`
  - #5104 adds `Nat.add_left_eq_self` and relatives.
  - #5146 adds missing `Nat.and_xor_distrib_(left|right)`.
  - #5148 and #5190 improve `Nat` and `Int` simp lemma confluence.
  - #5165 adjusts `Int` simp lemmas.

- #5166 adds `Int` lemmas relating `neg` and `emod`/`mod`.
- #5208 reverses the direction of the `Int.toNat_sub` simp lemma.
- #5209 adds `Nat.bitwise` lemmas.
- #5230 corrects the docstrings for integer division and modulus.

- `Option`
  - #5128 and #5154 add `Option` lemmas.
- `BitVec`
  - #4889 adds `sshiftRight` bitblasting.
  - #4981 adds `Std.Associative` and `Std.Commutative` instances for `BitVec.[and|or|xor]`.
  - #4913 enables `missingDocs` error for `BitVec` modules.
  - #4930 makes parameter names for `BitVec` more consistent.
  - #5098 adds `BitVec.intMin`. Introduces `boolToPropSimps` simp set for converting from boolean to propositional expressions.
  - #5200 and #5217 rename `BitVec.getLsb` to `BitVec.getLsbD`, etc., to bring naming in line with `List`/`Array`/etc.
  - **Theorems:** #4977, #4951, #4667, #5007, #4997, #5083, #5081, #4392
- `UInt`
  - #4514 fixes naming convention for `UInt` lemmas.
- `Std.HashMap` and `Std.HashSet`
  - #4943 deprecates variants of hash map query methods. (See breaking changes.)
  - #4917 switches the library and Lean to `Std.HashMap` and `Std.HashSet` almost everywhere.
  - #4954 deprecates `Lean.HashMap` and `Lean.HashSet`.
  - #5023 cleans up lemma parameters.
- `Std.Sat` (for `bv_decide`)
  - #4933 adds definitions of SAT and CNF.
  - #4953 defines "and-inverter graphs" (AIGs) as described in section 3 of Davis-Swords 2013.
- **Parsec**
  - #4774 generalizes the `Parsec` library, allowing parsing of iterable data beyond `String` such as `ByteArray`. (See breaking changes.)
  - #5115 moves `Lean.Data.Parsec` to `Std.Internal.Parsec` for bootstrappng reasons.

- Thunk
  - #4969 upstreams `Thunk.ext`.
- **IO**
  - #4973 modifies `IO.FS.lines` to handle `\r\n` on all operating systems instead of just on Windows.
  - #5125 adds `createTempFile` and `withTempFile` for creating temporary files that can only be read and written by the current user.
- **Other fixes or improvements**
  - #4945 adds `Array`, `Bool` and `Prod` utilities from LeanSAT.
  - #4960 adds `Relation.TransGen.trans`.
  - #5012 states `WellFoundedRelation Nat` using `<`, not `Nat.lt`.
  - #5011 uses ≠ instead of `Not (Eq ...)` in `Fin.ne_of_val_ne`.
  - #5197 upstreams `Fin.le_antisymm`.
  - #5042 reduces usage of `refine'`.
  - #5101 adds about `if-then-else` and `Option`.
  - #5112 adds basic instances for `ULift` and `PLift`.
  - #5133 and #5168 make fixes from running the simpNF linter over Lean.
  - #5156 removes a bad simp lemma in omega theory.
  - #5155 improves confluence of `Bool` simp lemmas.
  - #5162 improves confluence of `Function.comp` simp lemmas.
  - #5191 improves confluence of `if-then-else` simp lemmas.
  - #5147 adds `@[elab_as_elim]` to `Quot.rec`, `Nat.strongInductionOn` and `Nat.casesStrongInductionOn`, and also renames the latter two to `Nat.strongRecOn` and `Nat.casesStrongRecOn` (deprecated in #5179).
  - #5180 disables some simp lemmas with bad discrimination tree keys.
  - #5189 cleans up internal simp lemmas that had leaked.
  - #5198 cleans up `allowUnsafeReducibility`.
  - #5229 removes unused lemmas from some `simp` tactics.
  - #5199 removes >6 month deprecations.

Lean internals

- **Performance**
  - Some core algorithms have been rewritten in C++ for performance.
    - #4910 and #4912 reimplement `instantiateLevelMVars`.

    - #4915, #4922, and #4931 reimplement `instantiateExprMVars`, 30% faster on a benchmark.
  - #4934 has optimizations for the kernel's `Expr` equality test.
  - #4990 fixes bug in hashing for the kernel's `Expr` equality test.
  - #4935 and #4936 skip some `PreDefinition` transformations if they are not needed.
  - #5225 adds caching for visited exprs at `CheckAssignmentQuick` in `ExprDefEq`.
  - #5226 maximizes term sharing at `instantiateMVarDeclMVars`, used by `runTactic`.
- **Diagnostics and profiling**
  - #4923 adds profiling for `instantiateMVars` in `Lean.Elab.MutualDef`, which can be a bottleneck there.
  - #4924 adds diagnostics for large theorems, controlled by the `diagnostics.threshold.proofSize` option.
  - #4897 improves display of diagnostic results.
- **Other fixes or improvements**
  - #4921 cleans up `Expr.betaRev`.
  - #4940 fixes tests by not writing directly to stdout, which is unreliable now that elaboration and reporting are executed in separate threads.
  - #4955 documents that `stderrAsMessages` is now the default on the command line as well.
  - #4647 adjusts documentation for building on macOS.
  - #4987 makes regular mvar assignments take precedence over delayed ones in `instantiateMVars`. Normally delayed assignment metavariables are never directly assigned, but on errors Lean assigns `sorry` to unassigned metavariables.
  - #4967 adds linter name to errors when a linter crashes.
  - #5043 cleans up command line snapshots logic.
  - #5067 minimizes some imports.
  - #5068 generalizes the monad for `addMatcherInfo`.
  - f71a1f adds missing test for #5126.
  - #5201 restores a test.
  - #3698 fixes a bug where label attributes did not pass on the attribute kind.
  - Typos: #5080, #5150, #5202

Compiler, runtime, and FFI

- #3106 moves frontend to new snapshot architecture. Note that `Frontend.processCommand` and `FrontendM` are no longer used by Lean core, but they will be preserved.
- #4919 adds missing include in runtime for `AUTO_THREAD_FINALIZATION` feature on Windows.
- #4941 adds more `LEAN_EXPORT`s for Windows.
- #4911 improves formatting of CLI help text for the frontend.
- #4950 improves file reading and writing.
  - `readBinFile` and `readFile` now only require two system calls (`stat` + `read`) instead of one `read` per 1024 byte chunk.
  - `Handle.getLine` and `Handle.putStr` no longer get tripped up by NUL characters.
- #4971 handles the SIGBUS signal when detecting stack overflows.
- #5062 avoids overwriting existing signal handlers, like in rust-lang/rust#69685.
- #4860 improves workarounds for building on Windows. Splits `libleanshared` on Windows to avoid symbol limit, removes the `LEAN_EXPORT` denylist workaround, adds missing `LEAN_EXPORT`s.
- #4952 output panics into Lean's redirected stderr, ensuring panics ARE visible as regular messages in the language server and properly ordered in relation to other messages on the command line.
- #4963 links LibUV.

Lake

- #5030 removes dead code.
- #4770 adds additional fields to the package configuration which will be used by Reservoir. See the PR description for details.

DevOps/CI

- #4914 and #4937 improve the release checklist.
- #4925 ignores stale leanpkg tests.
- #5003 upgrades `actions/cache` in CI.
- #5010 sets `save-always` in cache actions in CI.
- #5008 adds more libuv search patterns for the speedcenter.
- #5009 reduce number of runs in the speedcenter for "fast" benchmarks from 10 to 3.
- #5014 adjusts lakefile editing to use new `git` syntax in `pr-release` workflow.
- #5025 has `pr-release` workflow pass `--retry` to `curl`.
- #5022 builds MacOS Aarch64 release for PRs by default.

- #5045 adds libuv to the required packages heading in macos docs.
- #5034 fixes the install name of `libleanshared_1` on macOS.
- #5051 fixes Windows stage 0.
- #5052 fixes 32bit stage 0 builds in CI.
- #5057 avoids rebuilding `leanmanifest` in each build.
- #5099 makes `restart-on-label` workflow also filter by commit SHA.
- #4325 adds CaDiCaL.

Breaking changes

- LibUV is now required to build Lean. This change only affects developers who compile Lean themselves instead of obtaining toolchains via `elan`. We have updated the official build instructions with information on how to obtain LibUV on our supported platforms. (#4963)
- Recursive definitions with a `decreasing_by` clause that begins with `simp_wf` may break. Try removing `simp_wf` or replacing it with `simp`. (#5016)
- The behavior of `rw [f]` where `f` is a non-recursive function defined by pattern matching changed.

  For example, preciously, `rw [Option.map]` would rewrite `Option.map f o` to `match o with …` . Now this rewrite fails because it will use the equational lemmas, and these require constructors – just like for `List.map`.

  Remedies:

  - Split on `o` before rewriting.
  - Use `rw [Option.map.eq_def]`, which rewrites any (saturated) application of `Option.map`.
  - Use `set_option backward.eqns.nonrecursive false` when *defining* the function in question. (#4154)
- The unified handling of equation lemmas for recursive and non-recursive functions can break existing code, as there now can be extra equational lemmas:

  - Explicit uses of `f.eq_2` might have to be adjusted if the numbering changed.
  - Uses of `rw [f]` or `simp [f]` may no longer apply if they previously matched (and introduced a `match` statement), when the equational lemmas got more fine-grained.

    In this case either case analysis on the parameters before rewriting helps, or setting the option `backward.eqns.deepRecursiveSplit false` while *defining* the function.

  (#5129, #5207)

- The `reduceCtorEq` simproc is now optional, and it might need to be included in lists of simp lemmas, like `simp only [reduceCtorEq]`. This simproc is responsible for reducing equalities of constructors. (#5167)
- `Nat.strongInductionOn` is now `Nat.strongRecOn` and `Nat.caseStrongInductionOn` to `Nat.caseStrongRecOn`. (#5147)
- The parameters to `Membership.mem` have been swapped, which affects all `Membership` instances. (#5020)
- The meanings of `List.getElem_drop` and `List.getElem_drop'` have been reversed and the first is now a simp lemma. (#5210)
- The `Parsec` library has moved from `Lean.Data.Parsec` to `Std.Internal.Parsec`. The `Parsec` type is now more general with a parameter for an iterable. Users parsing strings can migrate to `Parser` in the `Std.Internal.Parsec.String` namespace, which also includes string-focused parsing combinators. (#4774)
- The `Lean` module has switched from `Lean.HashMap` and `Lean.HashSet` to `Std.HashMap` and `Std.HashSet` (#4943). `Lean.HashMap` and `Lean.HashSet` are now deprecated (#4954) and will be removed in a future release. Users of `Lean` APIs that interact with hash maps, for example `Lean.Environment.const2ModIdx`, might encounter minor breakage due to the following changes from `Lean.HashMap` to `Std.HashMap`:
  - query functions use the term `get` instead of `find`, (#4943)
  - the notation `map[key]` no longer returns an optional value but instead expects a proof that the key is present in the map. The previous behavior is available via the `map[key]?` notation.

Lean 4.11.0 (2024-09-02)

Language features, tactics, and metaprograms

- The variable inclusion mechanism has been changed. Like before, when a definition mentions a variable, Lean will add it as an argument of the definition, but now in theorem bodies, variables are not included based on usage in order to ensure that changes to the proof cannot change the statement of the overall theorem. Instead, variables are only available to the proof if they have been mentioned in the theorem header or in an **`include` command** or are instance implicit and depend only on such variables. The **`omit` command** can be used to omit included variables.

  See breaking changes below.

  PRs: #4883, #4814, #5000, #5036, #5138, 0edf1b.
- **Recursive definitions**
  - Structural recursion can now be explicitly requested using

    ```
    termination_by structural x
    ```

in analogy to the existing `termination_by x` syntax that causes well-founded recursion to be used. #4542

- #4672 fixes a bug that could lead to ill-typed terms.
- The `termination_by?` syntax no longer forces the use of well-founded recursion, and when structural recursion is inferred, it will print the result using the `termination_by structural` syntax.
- **Mutual structural recursion** is now supported. This feature supports both mutual recursion over a non-mutual data type, as well as recursion over mutual or nested data types:

```
mutual
def Even : Nat → Prop
  | 0 => True
  | n+1 => Odd n

def Odd : Nat → Prop
  | 0 => False
  | n+1 => Even n
end

mutual
inductive A
| other : B → A
| empty
inductive B
| other : A → B
| empty
end

mutual
def A.size : A → Nat
| .other b => b.size + 1
| .empty => 0

def B.size : B → Nat
| .other a => a.size + 1
| .empty => 0
end

inductive Tree where | node : List Tree → Tree

mutual
def Tree.size : Tree → Nat
| node ts => Tree.list_size ts

def Tree.list_size : List Tree → Nat
| [] => 0
| t::ts => Tree.size t + Tree.list_size ts
end
```

Functional induction principles are generated for these functions as well (`A.size.induct`, `A.size.mutual_induct`).

Nested structural recursion is still not supported.

PRs: #4639, #4715, #4642, #4656, #4684, #4715, #4728, #4575, #4731, #4658, #4734, #4738, #4718, #4733, #4787, #4788, #4789, #4807, #4772

- #4809 makes unnecessary `termination_by` clauses cause warnings, not errors.
- #4831 improves handling of nested structural recursion through non-recursive types.
- #4839 improves support for structural recursive over inductive predicates when there are reflexive arguments.

- `simp` tactic
  - #4784 sets configuration `Simp.Config.implicitDefEqProofs` to `true` by default.
- `omega` tactic
  - #4612 normalizes the order that constraints appear in error messages.
  - #4695 prevents pushing casts into multiplications unless it produces a non-trivial linear combination.
  - #4989 fixes a regression.
- `decide` tactic
  - #4711 switches from using default transparency to *at least* default transparency when reducing the `Decidable` instance.
  - #4674 adds detailed feedback on `decide` tactic failure. It tells you which `Decidable` instances it unfolded, if it get stuck on `Eq.rec` it gives a hint about avoiding tactics when defining `Decidable` instances, and if it gets stuck on `Classical.choice` it gives hints about classical instances being in scope. During this process, it processes `Decidable.recs` and matches to pin blame on a non-reducing instance.
- `@[ext]` attribute
  - #4543 and #4762 make `@[ext]` realize `ext_iff` theorems from user `ext` theorems. Fixes the attribute so that `@[local ext]` and `@[scoped ext]` are usable. The `@[ext (iff := false)]` option can be used to turn off `ext_iff` realization.
  - #4694 makes "go to definition" work for the generated lemmas. Also adjusts the core library to make use of `ext_iff` generation.
  - #4710 makes `ext_iff` theorem preserve inst implicit binder types, rather than making all binder types implicit.
- `#eval` command

  - #4810 introduces a safer `#eval` command that prevents evaluation of terms that contain `sorry`. The motivation is that failing tactics, in conjunction with operations such as array accesses, can lead to the Lean process crashing. Users can use the new `#eval!` command to use the previous unsafe behavior. (#4829 adjusts a test.)

- #4447 adds `#discr_tree_key` and `#discr_tree_simp_key` commands, for helping debug discrimination tree failures. The `#discr_tree_key t` command prints the discrimination tree keys for a term `t` (or, if it is a single identifier, the type of that constant). It uses the default configuration for generating keys. The `#discr_tree_simp_key` command is similar to `#discr_tree_key`, but treats the underlying type as one of a simp lemma, that is it transforms it into an equality and produces the key of the left-hand side.

  For example,

```
#discr_tree_key (∀ {a n : Nat}, bar a (OfNat.ofNat n))
-- bar _ (@OfNat.ofNat Nat _ _)

#discr_tree_simp_key Nat.add_assoc
-- @HAdd.hAdd Nat Nat Nat _ (@HAdd.hAdd Nat Nat Nat _ _ _) _
```

- #4741 changes option parsing to allow user-defined options from the command line. Initial options are now re-parsed and validated after importing. Command line option assignments prefixed with `weak.` are silently discarded if the option name without the prefix does not exist.

- **Deriving handlers**

  - 7253ef and a04f3c improve the construction of the `BEq` deriving handler.
  - 86af04 makes `BEq` deriving handler work when there are dependently typed fields.
  - #4826 refactors the `DecidableEq` deriving handle to use `termination_by structural`.

- **Metaprogramming**

  - #4593 adds `unresolveNameGlobalAvoidingLocals`.
  - #4618 deletes deprecated functions from 2022.
  - #4642 adds `Meta.lambdaBoundedTelescope`.
  - #4731 adds `Meta.withErasedFVars`, to enter a context with some fvars erased from the local context.
  - #4777 adds assignment validation at `closeMainGoal`, preventing users from circumventing the occurs check for tactics such as `exact`.
  - #4807 introduces `Lean.Meta.PProdN` module for packing and projecting nested `PProd`s.
  - #5170 fixes `Syntax.unsetTrailing`. A consequence of this is that "go to definition" now works on the last module name in an `import` block (issue #4958).

Language server, widgets, and IDE extensions

- #4727 makes it so that responses to info view requests come as soon as the relevant tactic has finished execution.
- #4580 makes it so that whitespace changes do not invalidate imports, and so starting to type the first declaration after imports should no longer cause them to reload.
- #4780 fixes an issue where hovering over unimported builtin names could result in a panic.

Pretty printing

- #4558 fixes the `pp.instantiateMVars` setting and changes the default value to `true`.
- #4631 makes sure syntax nodes always run their formatters. Fixes an issue where if `ppSpace` appears in a `macro` or `elab` command then it does not format with a space.
- #4665 fixes a bug where pretty printed signatures (for example in `#check`) were overly hoverable due to `pp.tagAppFns` being set.
- #4724 makes `match` pretty printer be sensitive to `pp.explicit`, which makes hovering over a `match` in the Infoview show the underlying term.
- #4764 documents why anonymous constructor notation isn't pretty printed with flattening.
- #4786 adjusts the parenthesizer so that only the parentheses are hoverable, implemented by having the parentheses "steal" the term info from the parenthesized expression.
- #4854 allows arbitrarily long sequences of optional arguments to be omitted from the end of applications, versus the previous conservative behavior of omitting up to one optional argument.

Library

- `Nat`
  - #4597 adds bitwise lemmas `Nat.and_le_(left|right)`.
  - #4874 adds simprocs for simplifying bit expressions.
- `Int`
  - #4903 fixes performance of `HPow Int Nat Int` synthesis by rewriting it as a `NatPow Int` instance.
- `UInt*` and `Fin`
  - #4605 adds lemmas.
  - #4629 adds `*.and_toNat`.
- `Option`
  - #4599 adds `get` lemmas.
  - #4600 adds `Option.or`, a version of `Option.orElse` that is strict in the second argument.
- `GetElem`

- #4603 adds `getElem_congr` to help with rewriting indices.

- `List` and `Array`
  - Upstreamed from Batteries: #4586 upstreams `List.attach` and `Array.attach`, #4697 upstreams `List.Subset` and `List.Sublist` and API, #4706 upstreams basic material on `List.Pairwise` and `List.Nodup`, #4720 upstreams more `List.erase` API, #4836 and #4837 upstream `List.IsPrefix`/`List.IsSuffix`/`List.IsInfix` and add `Decidable` instances, #4855 upstreams `List.tail`, `List.findIdx`, `List.indexOf`, `List.countP`, `List.count`, and `List.range'`, #4856 upstreams more List lemmas, #4866 upstreams `List.pairwise_iff_getElem`, #4865 upstreams `List.eraseIdx` lemmas.
  - #4687 adjusts `List.replicate` simp lemmas and simprocs.
  - #4704 adds characterizations of `List.Sublist`.
  - #4707 adds simp normal form tests for `List.Pairwise` and `List.Nodup`.
  - #4708 and #4815 reorganize lemmas on list getters.
  - #4765 adds simprocs for literal array accesses such as `#[1,2,3,4,5][2]`.
  - #4790 removes typeclass assumptions for `List.Nodup.eraseP`.
  - #4801 adds efficient `usize` functions for array types.
  - #4820 changes `List.filterMapM` to run left-to-right.
  - #4835 fills in and cleans up gaps in List API.
  - #4843, #4868, and #4877 correct `List.Subset` lemmas.
  - #4863 splits `Init.Data.List.Lemmas` into function-specific files.
  - #4875 fixes statement of `List.take_takeWhile`.
  - Lemmas: #4602, #4627, #4678 for `List.head` and `list.getLast`, #4723 for `List.erase`, #4742

- `ByteArray`
  - #4582 eliminates `partial` from `ByteArray.toList` and `ByteArray.findIdx?`.

- `BitVec`
  - #4568 adds recurrence theorems for bitblasting multiplication.
  - #4571 adds `shiftLeftRec` lemmas.
  - #4872 adds `ushiftRightRec` and lemmas.
  - #4873 adds `getLsb_replicate`.

- `Std.HashMap` added:
  - #4583 **adds `Std.HashMap`** as a verified replacement for `Lean.HashMap`. See the PR for naming differences, but #4725 renames `HashMap.remove` to `HashMap.erase`.
  - #4682 adds `Inhabited` instances.

- #4732 improves BEq argument order in hash map lemmas.
- #4759 makes lemmas resolve instances via unification.
- #4771 documents that hash maps should be used linearly to avoid expensive copies.
- #4791 removes `bif` from hash map lemmas, which is inconvenient to work with in practice.
- #4803 adds more lemmas.

- SMap
  - #4690 upstreams `SMap.foldM`.
- BEq
  - #4607 adds `PartialEquivBEq`, `ReflBEq`, `EquivBEq`, and `LawfulHashable` classes.
- IO
  - #4660 adds `IO.Process.Child.tryWait`.
- #4747, #4730, and #4756 add `×'` syntax for `PProd`. Adds a delaborator for `PProd` and `MProd` values to pretty print as flattened angle bracket tuples.
- **Other fixes or improvements**
  - #4604 adds lemmas for cond.
  - #4619 changes some definitions into theorems.
  - #4616 fixes some names with duplicated namespaces.
  - #4620 fixes simp lemmas flagged by the simpNF linter.
  - #4666 makes the `Antisymm` class be a `Prop`.
  - #4621 cleans up unused arguments flagged by linter.
  - #4680 adds imports for orphaned `Init` modules.
  - #4679 adds imports for orphaned `Std.Data` modules.
  - #4688 adds forward and backward directions of `not_exists`.
  - #4689 upstreams `eq_iff_true_of_subsingleton`.
  - #4709 fixes precedence handling for `Repr` instances for negative numbers for `Int` and `Float`.
  - #4760 renames TC ("transitive closure") to `Relation.TransGen`.
  - #4842 fixes `List` deprecations.
  - #4852 upstreams some Mathlib attributes applied to lemmas.
  - 93ac63 improves proof.
  - #4862 and #4878 generalize the universe for `PSigma.exists` and rename it to `Exists.of_psigma_prop`.

- Typos: #4737, 7d2155
- Docs: #4782, #4869, #4648

Lean internals

- **Elaboration**
  - #4596 enforces `isDefEqStuckEx` at `unstuckMVar` procedure, causing isDefEq to throw a stuck defeq exception if the metavariable was created in a previous level. This results in some better error messages, and it helps `rw` succeed in synthesizing instances (see issue #2736).
  - #4713 fixes deprecation warnings when there are overloaded symbols.
  - `elab_as_elim` algorithm:
    - #4722 adds check that inferred motive is type-correct.
    - #4800 elaborates arguments for parameters appearing in the types of targets.
    - #4817 makes the algorithm correctly handle eliminators with explicit motive arguments.
  - #4792 adds term elaborator for `Lean.Parser.Term.namedPattern` (e.g. `n@(n' + 1)`) to report errors when used in non-pattern-matching contexts.
  - #4818 makes anonymous dot notation work when the expected type is a pi-type-valued type synonym.
- **Typeclass inference**
  - #4646 improves `synthAppInstances`, the function responsible for synthesizing instances for the `rw` and `apply` tactics. Adds a synthesis loop to handle functions whose instances need to be synthesized in a complex order.
- **Inductive types**
  - #4684 (backported as 98ee78) refactors `InductiveVal` to have a `numNested : Nat` field instead of `isNested : Bool`. This modifies the kernel.
- **Definitions**
  - #4776 improves performance of `Replacement.apply`.
  - #4712 fixes `.eq_def` theorem generation with messy universes.
  - #4841 improves success of finding `T.below x` hypothesis when transforming `match` statements for `IndPredBelow`.
- **Diagnostics and profiling**
  - #4611 makes kernel diagnostics appear when `diagnostics` is enabled even if it is the only section.
  - #4753 adds missing `profileitM` functions.

- #4754 adds `Lean.Expr.numObjs` to compute the number of allocated sub-expressions in a given expression, primarily for diagnosing performance issues.
- #4769 adds missing `withTraceNodes` to improve `trace.profiler` output.
- #4781 and #4882 make the "use `set_option diagnostics true`" message be conditional on current setting of `diagnostics`.

- **Performance**
  - #4767, #4775, and #4887 add `ShareCommon.shareCommon'` for sharing common terms. In an example with 16 million subterms, it is 20 times faster than the old `shareCommon` procedure.
  - #4779 ensures `Expr.replaceExpr` preserves DAG structure in `Exprs`.
  - #4783 documents performance issue in `Expr.replaceExpr`.
  - #4794, #4797, #4798 make `for_each` use precise cache.
  - #4795 makes `Expr.find?` and `Expr.findExt?` use the kernel implementations.
  - #4799 makes `Expr.replace` use the kernel implementation.
  - #4871 makes `Expr.foldConsts` use a precise cache.
  - #4890 makes `expr_eq_fn` use a precise cache.
- **Utilities**
  - #4453 upstreams `ToExpr FilePath` and `compile_time_search_path%`.
- **Module system**
  - #4652 fixes handling of `const2ModIdx` in `finalizeImport`, making it prefer the original module for a declaration when a declaration is re-declared.
- **Kernel**
  - #4637 adds a check to prevent large `Nat` exponentiations from evaluating. Elaborator reduction is controlled by the option `exponentiation.threshold`.
  - #4683 updates comments in `kernel/declaration.h`, making sure they reflect the current Lean 4 types.
  - #4796 improves performance by using `replace` with a precise cache.
  - #4700 improves performance by fixing the implementation of move constructors and move assignment operators. Expression copying was taking 10% of total runtime in some workloads. See issue #4698.
  - #4702 improves performance in `replace_rec_fn::apply` by avoiding expression copies. These copies represented about 13% of time spent in `save_result` in some workloads. See the same issue.
- **Other fixes or improvements**
  - #4590 fixes a typo in some constants and `trace.profiler.useHeartbeats`.

- #4617 add 'since' dates to `deprecated` attributes.
- #4625 improves the robustness of the constructor-as-variable test.
- #4740 extends test with nice example reported on Zulip.
- #4766 moves `Syntax.hasIdent` to be available earlier and shakes dependencies.
- #4881 splits out `Lean.Language.Lean.Types`.
- #4893 adds `LEAN_EXPORT` for `sharecommon` functions.
- Typos: #4635, #4719, af40e6
- Docs: #4748 (`Command.Scope`)

Compiler, runtime, and FFI

- #4661 moves `Std` from `libleanshared` to much smaller `libInit_shared`. This fixes the Windows build.
- #4668 fixes initialization, explicitly initializing `Std` in `lean_initialize`.
- #4746 adjusts `shouldExport` to exclude more symbols to get below Windows symbol limit. Some exceptions are added by #4884 and #4956 to support Verso.
- #4778 adds `lean_is_exclusive_obj` (`Lean.isExclusiveUnsafe`) and `lean_set_external_data`.
- #4515 fixes calling programs with spaces on Windows.

Lake

- #4735 improves a number of elements related to Git checkouts, cloud releases, and related error handling.
  - On error, Lake now prints all top-level logs. Top-level logs are those produced by Lake outside of the job monitor (e.g., when cloning dependencies).
  - When fetching a remote for a dependency, Lake now forcibly fetches tags. This prevents potential errors caused by a repository recreating tags already fetched.
  - Git error handling is now more informative.
  - The builtin package facets `release`, `optRelease`, `extraDep` are now captions in the same manner as other facets.
  - `afterReleaseSync` and `afterReleaseAsync` now fetch `optRelease` rather than `release`.
  - Added support for optional jobs, whose failure does not cause the whole build to failure. Now `optRelease` is such a job.
- #4608 adds draft CI workflow when creating new projects.
- #4847 adds CLI options to control log levels. The `--log-level=<lv>` controls the minimum log level Lake should output. For instance, `--log-level=error` will only print errors (not warnings or info). Also, adds an analogous `--fail-level` option to control the minimum

log level for build failures. The existing `--iofail` and `--wfail` options are respectively equivalent to `--fail-level=info` and `--fail-level=warning`.

- Docs: #4853

DevOps/CI

- **Workflows**
  - #4531 makes release trigger an update of `release.lean-lang.org`.
  - #4598 adjusts `pr-release` to the new `lakefile.lean` syntax.
  - #4632 makes `pr-release` use the correct tag name.
  - #4638 adds ability to manually trigger nightly release.
  - #4640 adds more debugging output for `restart-on-label` CI.
  - #4663 bumps up waiting for 10s to 30s for `restart-on-label`.
  - #4664 bumps versions for `actions/checkout` and `actions/upload-artifacts`.
  - 582d6e bumps version for `actions/download-artifact`.
  - 6d9718 adds back dropped `check-stage3`.
  - 0768ad adds Jira sync (for FRO).
  - #4830 adds support to report CI errors on FRO Zulip.
  - #4838 adds trigger for `nightly_bump_toolchain` on mathlib4 upon nightly release.
  - abf420 fixes msys2.
  - #4895 deprecates Nix-based builds and removes interactive components. Users who prefer the flake build should maintain it externally.
- #4693, #4458, and #4876 update the **release checklist**.
- #4669 fixes the "max dynamic symbols" metric per static library.
- #4691 improves compatibility of `tests/list_simp` for retesting simp normal forms with Mathlib.
- #4806 updates the quickstart guide.
- c02aa9 documents the **triage team** in the contribution guide.

Breaking changes

- For `@[ext]`-generated `ext` and `ext_iff` lemmas, the `x` and `y` term arguments are now implicit. Furthermore these two lemmas are now protected (#4543).
- Now `trace.profiler.useHearbeats` is `trace.profiler.useHeartbeats` (#4590).

- A bugfix in the structural recursion code may in some cases break existing code, when a parameter of the type of the recursive argument is bound behind indices of that type. This can usually be fixed by reordering the parameters of the function (#4672).
- Now `List.filterMapM` sequences monadic actions left-to-right (#4820).
- The effect of the `variable` command on proofs of `theorem`s has been changed. Whether such section variables are accessible in the proof now depends only on the theorem signature and other top-level commands, not on the proof itself. This change ensures that
  - the statement of a theorem is independent of its proof. In other words, changes in the proof cannot change the theorem statement.
  - tactics such as `induction` cannot accidentally include a section variable.
  - the proof can be elaborated in parallel to subsequent declarations in a future version of Lean.

  The effect of `variables` on the theorem header as well as on other kinds of declarations is unchanged.

  Specifically, section variables are included if they
  - are directly referenced by the theorem header,
  - are included via the new `include` command in the current section and not subsequently mentioned in an `omit` statement,
  - are directly referenced by any variable included by these rules, OR
  - are instance-implicit variables that reference only variables included by these rules.

  For porting, a new option `deprecated.oldSectionVars` is included to locally switch back to the old behavior.

Lean 4.10.0 (2024-07-31)

Language features, tactics, and metaprograms

- `split` tactic:
  - #4401 improves the strategy `split` uses to generalize discriminants of matches and adds `trace.split.failure` trace class for diagnosing issues.
- `rw` tactic:
  - #4385 prevents the tactic from claiming pre-existing goals are new subgoals.
  - dac1da adds configuration for ordering new goals, like for `apply`.
- `simp` tactic:
  - #4430 adds `dsimprocs` for `if` expressions (`ite` and `dite`).

- #4434 improves heuristics for unfolding. Equational lemmas now have priorities where more-specific equationals lemmas are tried first before a possible catch-all.
- #4481 fixes an issue where function-valued `OfNat` numeric literals would become denormalized.
- #4467 fixes an issue where dsimp theorems might not apply to literals.
- #4484 fixes the source position for the warning for deprecated simp arguments.
- #4258 adds docstrings for `dsimp` configuration.
- #4567 improves the accuracy of used simp lemmas reported by `simp?`.
- fb9727 adds (but does not implement) the simp configuration option `implicitDefEqProofs`, which will enable including `rfl`-theorems in proof terms.

- `omega` tactic:
  - #4360 makes the tactic generate error messages lazily, improving its performance when used in tactic combinators.
- `bv_omega` tactic:
  - #4579 works around changes to the definition of `Fin.sub` in this release.
- #4490 sets up groundwork for a tactic index in generated documentation, as there was in Lean 3. See PR description for details.
- **Commands**
  - #4370 makes the `variable` command fully elaborate binders during validation, fixing an issue where some errors would be reported only at the next declaration.
  - #4408 fixes a discrepancy in universe parameter order between `theorem` and `def` declarations.
  - #4493 and #4482 fix a discrepancy in the elaborators for `theorem`, `def`, and `example`, making `Prop`-valued `example`s and other definition commands elaborate like `theorem`s.
  - 8f023b, 3c4d6b and 0783d0 change the `#reduce` command to be able to control what gets reduced. For example, `#reduce (proofs := true) (types := false) e` reduces both proofs and types in the expression `e`. By default, neither proofs or types are reduced.
  - #4489 fixes an elaboration bug in `#check_tactic`.
  - #4505 adds support for `open _root_.<namespace>`.
- **Options**
  - #4576 adds the `debug.byAsSorry` option. Setting `set_option debug.byAsSorry true` causes all `by ...` terms to elaborate as `sorry`.

- 7b56eb and d8e719 add the `debug.skipKernelTC` option. Setting `set_option debug.skipKernelTC true` turns off kernel typechecking. This is meant for temporarily working around kernel performance issues, and it compromises soundness since buggy tactics may produce invalid proofs, which will not be caught if this option is set to true.

- #4301 adds a linter to flag situations where a local variable's name is one of the argumentless constructors of its type. This can arise when a user either doesn't open a namespace or doesn't add a dot or leading qualifier, as in the following:

  ```
  inductive Tree (α : Type) where
    | leaf
    | branch (left : Tree α) (val : α) (right : Tree α)

  def depth : Tree α → Nat
    | leaf => 0
  ```

  With this linter, the `leaf` pattern is highlighted as a local variable whose name overlaps with the constructor `Tree.leaf`.

  The linter can be disabled with `set_option linter.constructorNameAsVariable false`.

  Additionally, the error message that occurs when a name in a pattern that takes arguments isn't valid now suggests similar names that would be valid. This means that the following definition:

  ```
  def length (list : List α) : Nat :=
    match list with
    | nil => 0
    | cons x xs => length xs + 1
  ```

  now results in the following warning:

  ```
  warning: Local variable 'nil' resembles constructor 'List.nil' -
  write '.nil' (with a dot) or 'List.nil' to use the constructor.
  note: this linter can be disabled with `set_option
  linter.constructorNameAsVariable false`
  ```

  and error:

  ```
  invalid pattern, constructor or constant marked with
  '[match_pattern]' expected

  Suggestion: 'List.cons' is similar
  ```

- **Metaprogramming**

  - #4454 adds public `Name.isInternalDetail` function for filtering declarations using naming conventions for internal names.

- **Other fixes or improvements**

- #4416 sorts the output of `#print axioms` for determinism.
- #4528 fixes error message range for the cdot focusing tactic.

Language server, widgets, and IDE extensions

- #4443 makes the watchdog be more resilient against badly behaving clients.

Pretty printing

- #4433 restores fallback pretty printers when context is not available, and documents `addMessageContext`.
- #4556 introduces `pp.maxSteps` option and sets the default value of `pp.deepTerms` to `false`. Together, these keep excessively large or deep terms from overwhelming the Infoview.

Library

- #4560 splits `GetElem` class into `GetElem` and `GetElem?`. This enables removing `Decidable` instance arguments from `GetElem.getElem?` and `GetElem.getElem!`, improving their rewritability. See the docstrings for these classes for more information.
- `Array`
  - #4389 makes `Array.toArrayAux_eq` be a `simp` lemma.
  - #4399 improves robustness of the proof for `Array.reverse_data`.
- `List`
  - #4469 and #4475 improve the organization of the `List` API.
  - #4470 improves the `List.set` and `List.concat` API.
  - #4472 upstreams lemmas about `List.filter` from Batteries.
  - #4473 adjusts `@[simp]` attributes.
  - #4488 makes `List.getElem?_eq_getElem` be a simp lemma.
  - #4487 adds missing `List.replicate` API.
  - #4521 adds lemmas about `List.map`.
  - #4500 changes `List.length_cons` to use `as.length + 1` instead of `as.length.succ`.
  - #4524 fixes the statement of `List.filter_congr`.
  - #4525 changes binder explicitness in `List.bind_map`.
  - #4550 adds `maximum?_eq_some_iff'` and `minimum?_eq_some_iff?`.
- #4400 switches the normal forms for indexing `List` and `Array` to `xs[n]` and `xs[n]?`.
- `HashMap`
  - #4372 fixes linearity in `HashMap.insert` and `HashMap.erase`, leading to a 40% speedup in a replace-heavy workload.

- `Option`
  - #4403 generalizes type of `Option.forM` from `Unit` to `PUnit`.
  - #4504 remove simp attribute from `Option.elim` and instead adds it to individual reduction lemmas, making unfolding less aggressive.
- `Nat`
  - #4242 adds missing theorems for `n + 1` and `n - 1` normal forms.
  - #4486 makes `Nat.min_assoc` be a simp lemma.
  - #4522 moves `@[simp]` from `Nat.pred_le` to `Nat.sub_one_le`.
  - #4532 changes various `Nat.succ n` to `n + 1`.
- `Int`
  - #3850 adds complete div/mod simprocs for `Int`.
- `String`/`Char`
  - #4357 make the byte size interface be `Nat`-valued with functions `Char.utf8Size` and `String.utf8ByteSize`.
  - #4438 upstreams `Char.ext` from Batteries and adds some `Char` documentation to the manual.
- `Fin`
  - #4421 adjusts `Fin.sub` to be more performant in definitional equality checks.
- `Prod`
  - #4526 adds missing `Prod.map` lemmas.
  - #4533 fixes binder explicitness in lemmas.
- `BitVec`
  - #4428 adds missing `simproc` for `BitVec` equality.
  - #4417 adds `BitVec.twoPow` and lemmas, toward bitblasting multiplication for LeanSAT.
- `Std` library
  - #4499 introduces `Std`, a library situated between `Init` and `Lean`, providing functionality not in the prelude both to Lean's implementation and to external users.
- **Other fixes or improvements**
  - #3056 standardizes on using `(· == a)` over `(a == ·)`.
  - #4502 fixes errors reported by running the library through the the Batteries linters.

Lean internals

- #4391 makes `getBitVecValue?` recognize `BitVec.ofNatLt`.
- #4410 adjusts `instantiateMVars` algorithm to zeta reduce `let` expressions while beta reducing instantiated metavariables.
- #4420 fixes occurs check for metavariable assignments to also take metavariable types into account.
- #4425 fixes `forEachModuleInDir` to iterate over each Lean file exactly once.
- #3886 adds support to build Lean core oleans using Lake.
- **Defeq and WHNF algorithms**
  - #4387 improves performance of `isDefEq` by eta reducing lambda-abstracted terms during metavariable assignments, since these are beta reduced during metavariable instantiation anyway.
  - #4388 removes redundant code in `isDefEqQuickOther`.
- **Typeclass inference**
  - #4530 fixes handling of metavariables when caching results at `synthInstance?`.
- **Elaboration**
  - #4426 makes feature where the "don't know how to synthesize implicit argument" error reports the name of the argument more reliable.
  - #4497 fixes a name resolution bug for generalized field notation (dot notation).
  - #4536 blocks the implicit lambda feature for `(e :)` notation.
  - #4562 makes it be an error for there to be two functions with the same name in a `where`/`let rec` block.
- Recursion principles
  - #4549 refactors `findRecArg`, extracting `withRecArgInfo`. Errors are now reported in parameter order rather than the order they are tried (non-indices are tried first). For every argument, it will say why it wasn't tried, even if the reason is obvious (e.g. a fixed prefix or is `Prop`-typed, etc.).
- Porting core C++ to Lean
  - #4474 takes a step to refactor `constructions` toward a future port to Lean.
  - #4498 ports `mk_definition_inferring_unsafe` to Lean.
  - #4516 ports `recOn` construction to Lean.
  - #4517, #4653, and #4651 port `below` and `brecOn` construction to Lean.
- Documentation
  - #4501 adds a more-detailed docstring for `PersistentEnvExtension`.
- **Other fixes or improvements**

  - #4382 removes `@[inline]` attribute from `NameMap.find?`, which caused respecialization at each call site.
  - 5f9ded improves output of `trace.Elab.snapshotTree`.
  - #4424 removes "you might need to open '{dir}' in your editor" message that is now handled by Lake and the VS Code extension.
  - #4451 improves the performance of `CollectMVars` and `FindMVar`.
  - #4479 adds missing `DecidableEq` and `Repr` instances for intermediate structures used by the `BitVec` and `Fin` simprocs.
  - #4492 adds tests for a previous `isDefEq` issue.
  - 9096d6 removes `PersistentHashMap.size`.
  - #4508 fixes `@[implemented_by]` for functions defined by well-founded recursion.
  - #4509 adds additional tests for `apply?` tactic.
  - d6eab3 fixes a benchmark.
  - #4563 adds a workaround for a bug in `IndPredBelow.mkBelowMatcher`.
- **Cleanup:** #4380, #4431, #4494, e8f768, de2690, d3a756, #4404, #4537.

Compiler, runtime, and FFI

- d85d3d fixes criterion for tail-calls in ownership calculation.
- #3963 adds validation of UTF-8 at the C++-to-Lean boundary in the runtime.
- #4512 fixes missing unboxing in interpreter when loading initialized value.
- #4477 exposes the compiler flags for the bundled C compiler (clang).

Lake

- #4384 deprecates `inputFile` and replaces it with `inputBinFile` and `inputTextFile`. Unlike `inputBinFile` (and `inputFile`), `inputTextFile` normalizes line endings, which helps ensure text file traces are platform-independent.
- #4371 simplifies dependency resolution code.
- #4439 touches up the Lake configuration DSL and makes other improvements: string literals can now be used instead of identifiers for names, avoids using French quotes in `lake new` and `lake init` templates, changes the `exe` template to use `Main` for the main module, improves the `math` template error if `lean-toolchain` fails to download, and downgrades unknown configuration fields from an error to a warning to improve cross-version compatibility.
- #4496 tweaks `require` syntax and updates docs. Now `require` in TOML for a package name such as `doc-gen4` does not need French quotes.
- #4485 fixes a bug where package versions in indirect dependencies would take precedence over direct dependencies.

- #4478 fixes a bug where Lake incorrectly included the module dynamic library in a platform-independent trace.
- #4529 fixes some issues with bad import errors. A bad import in an executable no longer prevents the executable's root module from being built. This also fixes a problem where the location of a transitive bad import would not been shown. The root module of the executable now respects `nativeFacets`.
- #4564 fixes a bug where non-identifier script names could not be entered on the CLI without French quotes.
- #4566 addresses a few issues with precompiled libraries.
  - Fixes a bug where Lake would always precompile the package of a module.
  - If a module is precompiled, it now precompiles its imports. Previously, it would only do this if imported.
- #4495, #4692, #4849 add a new type of `require` that fetches package metadata from a registry API endpoint (e.g. Reservoir) and then clones a Git package using the information provided. To require such a dependency, the new syntax is:

  ```
  require <scope> / <pkg-name> [@ git <rev>]
  -- Examples:
  require "leanprover" / "doc-gen4"
  require "leanprover-community" / "proofwidgets" @ git "v0.0.39"
  ```

  Or in TOML:

  ```
  [[require]]
  name = "<pkg-name>"
  scope = "<scope>"
  rev = "<rev>"
  ```

  Unlike with Git dependencies, Lake can make use of the richer information provided by the registry to determine the default branch of the package. This means for repositories of packages like `doc-gen4` which have a default branch that is not `master`, Lake will now use said default branch (e.g., in `doc-gen4`'s case, `main`).

  Lake also supports configuring the registry endpoint via an environment variable: `RESERVIOR_API_URL`. Thus, any server providing a similar interface to Reservoir can be used as the registry. Further configuration options paralleling those of Cargo's Alternative Registries and Source Replacement will come in the future.

DevOps/CI

- #4427 uses Namespace runners for CI for `leanprover/lean4`.
- #4440 fixes speedcenter tests in CI.
- #4441 fixes that workflow change would break CI for unrebased PRs.
- #4442 fixes Wasm release-ci.

- 6d265b fixes `github.event.pull_request.merge_commit_sha` sometimes not being available.
- 16cad2 adds optimization for CI to not fetch complete history.
- #4544 causes releases to be marked as prerelease on GitHub.
- #4446 switches Lake to using `src/lake/lakefile.toml` to avoid needing to load a version of Lake to build Lake.
- Nix
  - 5eb5fa fixes `update-stage0-commit` for Nix.
  - #4476 adds gdb to Nix shell.
  - e665a0 fixes `update-stage0` for Nix.
  - 4808eb fixes `cacheRoots` for Nix.
  - #3811 adds platform-dependent flag to lib target.
  - #4587 adds linking of `-lStd` back into nix build flags on darwin.

Breaking changes

- `Char.csize` is replaced by `Char.utf8Size` (#4357).
- Library lemmas now are in terms of `(· == a)` over `(a == ·)` (#3056).
- Now the normal forms for indexing into `List` and `Array` is `xs[n]` and `xs[n]?` rather than using functions like `List.get` (#4400).
- Sometimes terms created via a sequence of unifications will be more eta reduced than before and proofs will require adaptation (#4387).
- The `GetElem` class has been split into two; see the docstrings for `GetElem` and `GetElem?` for more information (#4560).

Lean 4.9.0 (2024-07-01)

Language features, tactics, and metaprograms

- **Definition transparency**
  - #4053 adds the `seal` and `unseal` commands, which make definitions locally be irreducible or semireducible.
  - #4061 marks functions defined by well-founded recursion with `@[irreducible]` by default, which should prevent the expensive and often unfruitful unfolding of such definitions (see breaking changes below).
- **Incrementality**
  - #3940 extends incremental elaboration into various steps inside of declarations: definition headers, bodies, and tactics.

Screen recording.

- 250994 and 67338b add `@[incremental]` attribute to mark an elaborator as supporting incremental elaboration.
- #4259 improves resilience by ensuring incremental commands and tactics are reached only in supported ways.
- #4268 adds special handling for `:= by` so that stray tokens in tactic blocks do not inhibit incrementality.
- #4308 adds incremental `have` tactic.
- #4340 fixes incorrect info tree reuse.
- #4364 adds incrementality for careful command macros such as `set_option in theorem`, `theorem foo.bar`, and `lemma`.
- #4395 adds conservative fix for whitespace handling to avoid incremental reuse leading to goals in front of the text cursor being shown.
- #4407 fixes non-incremental commands in macros blocking further incremental reporting.
- #4436 fixes incremental reporting when there are nested tactics in terms.
- #4459 adds incrementality support for `next` and `if` tactics.
- #4554 disables incrementality for tactics in terms in tactics.

- **Functional induction**
  - #4135 ensures that the names used for functional induction are reserved.
  - #4327 adds support for structural recursion on reflexive types. For example,

```
inductive Many (α : Type u) where
  | none : Many α
  | more : α → (Unit → Many α) → Many α

def Many.map {α β : Type u} (f : α → β) : Many α → Many β
  | .none => .none
  | .more x xs => .more (f x) (fun _ => (xs ()).map f)

#check Many.map.induct
/-
Many.map.induct {α β : Type u} (f : α → β) (motive : Many α →
Prop)
  (case1 : motive Many.none)
  (case2 : ∀ (x : α) (xs : Unit → Many α), motive (xs ()) →
motive (Many.more x xs)) :
  ∀ (a : Many α), motive a
-/
```

- #3903 makes the Lean frontend normalize all line endings to LF before processing. This lets Lean be insensitive to CRLF vs LF line endings, improving the cross-platform experience and making Lake hashes be faithful to what Lean processes.

- #4130 makes the tactic framework be able to recover from runtime errors (for example, deterministic timeouts or maximum recursion depth errors).
- `split` tactic
  - #4211 fixes `split at h` when `h` has forward dependencies.
  - #4349 allows `split` for `if`-expressions to work on non-propositional goals.
- `apply` tactic
  - #3929 makes error message for `apply` show implicit arguments in unification errors as needed. Modifies `MessageData` type (see breaking changes below).
- `cases` tactic
  - #4224 adds support for unification of offsets such as `x + 20000 = 20001` in `cases` tactic.
- `omega` tactic
  - #4073 lets `omega` fall back to using classical `Decidable` instances when setting up contradiction proofs.
  - #4141 and #4184 fix bugs.
  - #4264 improves `omega` error message if no facts found in local context.
  - #4358 improves expression matching in `omega` by using `match_expr`.
- `simp` tactic
  - #4176 makes names of erased lemmas clickable.
  - #4208 adds a pretty printer for discrimination tree keys.
  - #4202 adds `Simp.Config.index` configuration option, which controls whether to use the full discrimination tree when selecting candidate simp lemmas. When `index := false`, only the head function is taken into account, like in Lean 3. This feature can help users diagnose tricky simp failures or issues in code from libraries developed using Lean 3 and then ported to Lean 4.

    In the following example, it will report that `foo` is a problematic theorem.

    ```
    opaque f : Nat → Nat → Nat

    @[simp] theorem foo : f x (x, y).2 = y := by sorry

    example : f a b ≤ b := by
      set_option diagnostics true in
      simp (config := { index := false })
    /-
    [simp] theorems with bad keys
      foo, key: f _ (@Prod.mk ℕ ℕ _ _).2
    -/
    ```

With the information above, users can annotate theorems such as `foo` using `no_index` for problematic subterms. Example:

```
opaque f : Nat → Nat → Nat

@[simp] theorem foo : f x (no_index (x, y).2) = y := by sorry

example : f a b ≤ b := by
  simp -- `foo` is still applied with `index := true`
```

- #4274 prevents internal `match` equational theorems from appearing in simp trace.
- #4177 and #4359 make `simp` continue even if a simp lemma does not elaborate, if the tactic state is in recovery mode.
- #4341 fixes panic when applying `@[simp]` to malformed theorem syntax.
- #4345 fixes `simp` so that it does not use the forward version of a user-specified backward theorem.
- #4352 adds missing `dsimp` simplifications for fixed parameters of generated congruence theorems.
- #4362 improves trace messages for `simp` so that constants are hoverable.

- **Elaboration**
  - #4046 makes subst notation (`he ▸ h`) try rewriting in both directions even when there is no expected type available.
  - #3328 adds support for identifiers in autoparams (for example, `rfl` in `(h : x = y := by exact rfl)`).
  - #4096 changes how the type in `let` and `have` is elaborated, requiring that any tactics in the type be evaluated before proceeding, improving performance.
  - #4215 ensures the expression tree elaborator commits to the computed "max type" for the entire arithmetic expression.
  - #4267 cases signature elaboration errors to show even if there are parse errors in the body.
  - #4368 improves error messages when numeric literals fail to synthesize an `OfNat` instance, including special messages warning when the expected type of the numeral can be a proposition.
  - #4643 fixes issue leading to nested error messages and info trees vanishing, where snapshot subtrees were not restored on reuse.
  - #4657 calculates error suppression per snapshot, letting elaboration errors appear even when there are later parse errors (RFC #3556).

- **Metaprogramming**
  - #4167 adds `Lean.MVarId.revertAll` to revert all free variables.

- #4169 adds `Lean.MVarId.ensureNoMVar` to ensure the goal's target contains no expression metavariables.
- #4180 adds `cleanupAnnotations` parameter to `forallTelescope` methods.
- #4307 adds support for parser aliases in syntax quotations.

- Work toward implementing `grind` tactic
  - 0a515e and #4164 add `grind_norm` and `grind_norm_proc` attributes and `@[grind_norm]` theorems.
  - #4170, #4221, and #4249 create `grind` preprocessor and core module.
  - #4235 and d6709e add special `cases` tactic to `grind` along with `@[grind_cases]` attribute to mark types that this `cases` tactic should automatically apply to.
  - #4243 adds special `injection?` tactic to `grind`.
- **Other fixes or improvements**
  - #4065 fixes a bug in the `Nat.reduceLeDiff` simproc.
  - #3969 makes deprecation warnings activate even for generalized field notation ("dot notation").
  - #4132 fixes the `sorry` term so that it does not activate the implicit lambda feature
  - 9803c5 and 47c8e3 move `cdot` and `calc` parsers to `Lean` namespace.
  - #4252 fixes the `case` tactic so that it is usable in macros by having it erase macro scopes from the tag.
  - 26b671 and cc33c3 extract `haveId` syntax.
  - #4335 fixes bugs in partial `calc` tactic when there is mdata or metavariables.
  - #4329 makes `termination_by?` report unused each unused parameter as `_`.
- **Docs:** #4238, #4294, #4338.

Language server, widgets, and IDE extensions

- #4066 fixes features like "Find References" when browsing core Lean sources.
- #4254 allows embedding user widgets in structured messages. Companion PR is vscode-lean4#449.
- #4445 makes watchdog more resilient against badly behaving clients.

Library

- #4059 upstreams many `List` and `Array` operations and theorems from Batteries.
- #4055 removes the unused `Inhabited` instance for `Subtype`.
- #3967 adds dates in existing `@[deprecated]` attributes.
- #4231 adds boilerplate `Char`, `UInt`, and `Fin` theorems.

- #4205 fixes the `MonadStore` type classes to use `semiOutParam`.
- #4350 renames `IsLawfulSingleton` to `LawfulSingleton`.
- `Nat`
  - #4094 swaps `Nat.zero_or` and `Nat.or_zero`.
  - #4098 and #4145 change the definition of `Nat.mod` so that `n % (m + n)` reduces when `n` is literal without relying on well-founded recursion, which becomes irreducible by default in #4061.
  - #4188 redefines `Nat.testBit` to be more performant.
  - Theorems: #4199.
- `Array`
  - #4074 improves the functional induction principle `Array.feraseIdx.induct`.
- `List`
  - #4172 removes `@[simp]` from `List.length_pos`.
- `Option`
  - #4037 adds theorems to simplify `Option`-valued dependent if-then-else.
  - #4314 removes `@[simp]` from `Option.bind_eq_some`.
- `BitVec`
  - Theorems: #3920, #4095, #4075, #4148, #4165, #4178, #4200, #4201, #4298, #4299, #4257, #4179, #4321, #4187.
  - #4193 adds simprocs for reducing `x >>> i` and `x <<< i` where `i` is a bitvector literal.
  - #4194 adds simprocs for reducing `(x <<< i) <<< j` and `(x >>> i) >>> j` where `i` and `j` are natural number literals.
  - #4229 redefines `rotateLeft`/`rotateRight` to use modulo reduction of shift offset.
  - 0d3051 makes `<num>#<term>` bitvector literal notation global.
- `Char`/`String`
  - #4143 modifies `String.substrEq` to avoid linter warnings in downstream code.
  - #4233 adds simprocs for `Char` and `String` inequalities.
  - #4348 upstreams Mathlib lemmas.
  - #4354 upstreams basic `String` lemmas.
- `HashMap`
  - #4248 fixes implicitness of typeclass arguments in `HashMap.ofList`.
- `IO`

- #4036 adds `IO.Process.getCurrentDir` and `IO.Process.setCurrentDir` for adjusting the current process's working directory.

- **Cleanup:** #4077, #4189, #4304.
- **Docs:** #4001, #4166, #4332.

Lean internals

- **Defeq and WHNF algorithms**
  - #4029 remove unnecessary `checkpointDefEq`
  - #4206 fixes `isReadOnlyOrSyntheticOpaque` to respect metavariable depth.
  - #4217 fixes missing occurs check for delayed assignments.
- **Definition transparency**
  - #4052 adds validation to application of `@[reducible]`/`@[semireducible]`/`@[irreducible]` attributes (with `local`/`scoped` modifiers as well). Setting `set_option allowUnsafeReductibility true` turns this validation off.
- **Inductive types**
  - #3591 fixes a bug where indices could be incorrectly promoted to parameters.
  - #3398 fixes a bug in the injectivity theorem generator.
  - #4342 fixes elaboration of mutual inductives with instance parameters.
- **Diagnostics and profiling**
  - #3986 adds option `trace.profiler.useHeartbeats` to switch `trace.profiler.threshold` to being in terms of heartbeats instead of milliseconds.
  - #4082 makes `set_option diagnostics true` report kernel diagnostic information.
- **Typeclass resolution**
  - #4119 fixes multiple issues with TC caching interacting with `synthPendingDepth`, adds `maxSynthPendingDepth` option with default value `1`.
  - #4210 ensures local instance cache does not contain multiple copies of the same instance.
  - #4216 fix handling of metavariables, to avoid needing to set the option `backward.synthInstance.canonInstances` to `false`.
- **Other fixes or improvements**
  - #4080 fixes propagation of state for `Lean.Elab.Command.liftCoreM` and `Lean.Elab.Command.liftTermElabM`.

- #3944 makes the `Repr` deriving handler be consistent between `structure` and `inductive` for how types and proofs are erased.
- #4113 propagates `maxHeartbeats` to kernel to control "(kernel) deterministic timeout" error.
- #4125 reverts #3970 (monadic generalization of `FindExpr`).
- #4128 catches stack overflow in auto-bound implicits feature.
- #4129 adds `tryCatchRuntimeEx` combinator to replace `catchRuntimeEx` reader state.
- #4155 simplifies the expression canonicalizer.
- #4151 and #4369 add many missing trace classes.
- #4185 makes congruence theorem generators clean up type annotations of argument types.
- #4192 fixes restoration of infotrees when auto-bound implicit feature is activated, fixing a pretty printing error in hovers and strengthening the unused variable linter.
- dfb496 fixes `declareBuiltin` to allow it to be called multiple times per declaration.
- #4569 fixes an issue introduced in a merge conflict, where the interrupt exception was swallowed by some `tryCatchRuntimeEx` uses.
- #4584 (backported as b056a0) adapts kernel interruption to the new cancellation system.
- Cleanup: #4112, #4126, #4091, #4139, #4153.
- Tests: 030406, #4133.

Compiler, runtime, and FFI

- #4100 improves reset/reuse algorithm; it now runs a second pass relaxing the constraint that reused memory cells must only be for the exact same constructor.
- #2903 fixes segfault in old compiler from mishandling `noConfusion` applications.
- #4311 fixes bug in constant folding.
- #3915 documents the runtime memory layout for inductive types.

Lake

- #4518 makes trace reading more robust. Lake now rebuilds if trace files are invalid or unreadable and is backwards compatible with previous pure numeric traces.
- #4057 adds support for docstrings on `require` commands.
- #4088 improves hovers for `family_def` and `library_data` commands.
- #4147 adds default `README.md` to package templates
- #4261 extends `lake test` help page, adds help page for `lake check-test`, adds `lake lint` and tag `@[lint_driver]`, adds support for specifying test and lint drivers from dependencies, adds `testDriverArgs` and `lintDriverArgs` options, adds support for

library test drivers, makes `lake check-test` and `lake check-lint` only load the package without dependencies.

- #4270 adds `lake pack` and `lake unpack` for packing and unpacking Lake build artifacts from an archive.
- #4083 Switches the manifest format to use `major.minor.patch` semantic versions. Major version increments indicate breaking changes (e.g., new required fields and semantic changes to existing fields). Minor version increments (after `0.x`) indicate backwards-compatible extensions (e.g., adding optional fields, removing fields). This change is backwards compatible. Lake will still successfully read old manifests with numeric versions. It will treat the numeric version N as semantic version `0.N.0`. Lake will also accept manifest versions with `-` suffixes (e.g., `x.y.z-foo`) and then ignore the suffix.
- #4273 adds a lift from `JobM` to `FetchM` for backwards compatibility reasons.
- #4351 fixes `LogIO`-to-`CliM`-lifting performance issues.
- #4343 make Lake store the dependency trace for a build in the cached build long and then verifies that it matches the trace of the current build before replaying the log.
- #4402 moves the cached log into the trace file (no more `.log.json`). This means logs are no longer cached on fatal errors and this ensures that an out-of-date log is not associated with an up-to-date trace. Separately, `.hash` file generation was changed to be more reliable as well. The `.hash` files are deleted as part of the build and always regenerate with `--rehash`.
- **Other fixes or improvements**
  - #4056 cleans up tests
  - #4244 fixes `noRelease` test when Lean repo is tagged
  - #4346 improves `tests/serve`
  - #4356 adds build log path to the warning for a missing or invalid build log.

DevOps

- #3984 adds a script (`script/rebase-stage0.sh`) for `git rebase -i` that automatically updates each stage0.
- #4108 finishes renamings from transition to Std to Batteries.
- #4109 adjusts the Github bug template to mention testing using live.lean-lang.org.
- #4136 makes CI rerun only when `full-ci` label is added or removed.
- #4175 and 72b345 switch to using `#guard_msgs` to run tests as much as possible.
- #3125 explains the Lean4 `pygments` lexer.
- #4247 sets up a procedure for preparing release notes.
- #4032 modernizes build instructions and workflows.
- #4255 moves some expensive checks from merge queue to releases.
- #4265 adds aarch64 macOS as native compilation target for CI.

- f05a82 restores macOS aarch64 install suffix in CI
- #4317 updates build instructions for macOS.
- #4333 adjusts workflow to update Batteries in manifest when creating `lean-pr-testing-NNNN` Mathlib branches.
- #4355 simplifies `lean4checker` step of release checklist.
- #4361 adds installing elan to `pr-release` CI step.
- #4628 fixes the Windows build, which was missing an exported symbol.

Breaking changes

While most changes could be considered to be a breaking change, this section makes special note of API changes.

- `Nat.zero_or` and `Nat.or_zero` have been swapped (#4094).
- `IsLawfulSingleton` is now `LawfulSingleton` (#4350).
- The `BitVec` literal notation is now `<num>#<term>` rather than `<term>#<term>`, and it is global rather than scoped. Use `BitVec.ofNat w x` rather than `x#w` when `x` is a not a numeric literal (0d3051).
- `BitVec.rotateLeft` and `BitVec.rotateRight` now take the shift modulo the bitwidth (#4229).
- These are no longer simp lemmas: `List.length_pos` (#4172), `Option.bind_eq_some` (#4314).
- Types in `let` and `have` (both the expressions and tactics) may fail to elaborate due to new restrictions on what sorts of elaboration problems may be postponed (#4096). In particular, tactics embedded in the type will no longer make use of the type of `value` in expressions such as `let x : type := value; body`.
- Now functions defined by well-founded recursion are marked with `@[irreducible]` by default (#4061). Existing proofs that hold by definitional equality (e.g. `rfl`) can be rewritten to explicitly unfold the function definition (using `simp`, `unfold`, `rw`), or the recursive function can be temporarily made semireducible (using `unseal f in` before the command), or the function definition itself can be marked as `@[semireducible]` to get the previous behavior.
- Due to #3929:
  - The `MessageData.ofPPFormat` constructor has been removed. Its functionality has been split into two:
    - for lazy structured messages, please use `MessageData.lazy`;
    - for embedding `Format` or `FormatWithInfos`, use `MessageData.ofFormatWithInfos`.

    An example migration can be found in #3929.

- The `MessageData.ofFormat` constructor has been turned into a function. If you need to inspect `MessageData`, you can pattern-match on `MessageData.ofFormatWithInfos`.

Lean 4.8.0 (2024-06-05)

Language features, tactics, and metaprograms

- **Functional induction principles.** #3432, #3620, #3754, #3762, #3738, #3776, #3898.

  Derived from the definition of a (possibly mutually) recursive function, a **functional induction principle** is created that is tailored to proofs about that function.

  For example from:

  ```
  def ackermann : Nat → Nat → Nat
    | 0, m => m + 1
    | n+1, 0 => ackermann n 1
    | n+1, m+1 => ackermann n (ackermann (n + 1) m)
  ```

  we get

  ```
  ackermann.induct (motive : Nat → Nat → Prop) (case1 : ∀ (m :
  Nat), motive 0 m)
    (case2 : ∀ (n : Nat), motive n 1 → motive (Nat.succ n) 0)
    (case3 : ∀ (n m : Nat), motive (n + 1) m → motive n (ackermann
  (n + 1) m) → motive (Nat.succ n) (Nat.succ m))
    (x x : Nat) : motive x x
  ```

  It can be used in the `induction` tactic using the `using` syntax:

  ```
  induction n, m using ackermann.induct
  ```

- The termination checker now recognizes more recursion patterns without an explicit `termination_by`. In particular the idiom of counting up to an upper bound, as in

  ```
  def Array.sum (arr : Array Nat) (i acc : Nat) : Nat :=
    if _ : i < arr.size then
      Array.sum arr (i+1) (acc + arr[i])
    else
      acc
  ```

  is recognized without having to say `termination_by arr.size - i`.

  - #3630 makes `termination_by?` not use `sizeOf` when not needed
  - #3652 improves the `termination_by` syntax.
  - #3658 changes how termination arguments are elaborated.
  - #3665 refactors GuessLex to allow inferring more complex termination arguments
  - #3666 infers termination arguments such as `xs.size - i`

- #3629, #3655, #3747: Adds `@[induction_eliminator]` and `@[cases_eliminator]` attributes to be able to define custom eliminators for the `induction` and `cases` tactics, replacing the `@[eliminator]` attribute. Gives custom eliminators for `Nat` so that `induction` and `cases` put goal states into terms of `0` and `n + 1` rather than `Nat.zero` and `Nat.succ n`. Added option `tactic.customEliminators` to control whether to use custom eliminators. Added a hack for `rcases`/`rintro`/`obtain` to use the custom eliminator for `Nat`.
- **Shorter instances names.** There is a new algorithm for generating names for anonymous instances. Across Std and Mathlib, the median ratio between lengths of new names and of old names is about 72%. With the old algorithm, the longest name was 1660 characters, and now the longest name is 202 characters. The new algorithm's 95th percentile name length is 67 characters, versus 278 for the old algorithm. While the new algorithm produces names that are 1.2% less unique, it avoids cross-project collisions by adding a module-based suffix when it does not refer to declarations from the same "project" (modules that share the same root). #3089 and #3934.
- 8d2adf Importing two different files containing proofs of the same theorem is no longer considered an error. This feature is particularly useful for theorems that are automatically generated on demand (e.g., equational theorems).
- 84b091 Lean now generates an error if the type of a theorem is **not** a proposition.
- **Definition transparency.** 47a343. `@[reducible]`, `@[semireducible]`, and `@[irreducible]` are now scoped and able to be set for imported declarations.
- `simp`/`dsimp`
  - #3607 enables kernel projection reduction in `dsimp`
  - b24fbf and acdb00: `dsimproc` command to define defeq-preserving simplification procedures.
  - #3624 makes `dsimp` normalize raw nat literals as `OfNat.ofNat` applications.
  - #3628 makes `simp` correctly handle `OfScientific.ofScientific` literals.
  - #3654 makes `dsimp?` report used simprocs.
  - dee074 fixes equation theorem handling in `simp` for non-recursive definitions.
  - #3819 improved performance when simp encounters a loop.
  - #3821 fixes discharger/cache interaction.
  - #3824 keeps `simp` from breaking `Char` literals.
  - #3838 allows `Nat` instances matching to be more lenient.
  - #3870 documentation for `simp` configuration options.
  - #3972 fixes simp caching.
  - #4044 improves cache behavior for "well-behaved" dischargers.
- `omega`

- #3639, #3766, #3853, #3875: introduces a term canonicalizer.
- #3736 improves handling of positivity for the modulo operator for `Int`.
- #3828 makes it work as a `simp` discharger.
- #3847 adds helpful error messages.

- `rfl`
  - #3671, #3708: upstreams the `@[refl]` attribute and the `rfl` tactic.
  - #3751 makes `apply_rfl` not operate on `Eq` itself.
  - #4067 improves error message when there are no goals.
- #3719 upstreams the `rw?` tactic, with fixes and improvements in #3783, #3794, #3911.
- `conv`
  - #3659 adds a `conv` version of the `calc` tactic.
  - #3763 makes `conv` clean up using `try with_reducible rfl` instead of `try rfl`.
- `#guard_msgs`
  - #3617 introduces whitespace protection using the ⏎ character.
  - #3883: The `#guard_msgs` command now has options to change whitespace normalization and sensitivity to message ordering. For example, `#guard_msgs (whitespace := lax) in cmd` collapses whitespace before checking messages, and `#guard_msgs (ordering := sorted) in cmd` sorts the messages in lexicographic order before checking.
  - #3931 adds an unused variables ignore function for `#guard_msgs`.
  - #3912 adds a diff between the expected and actual outputs. This feature is currently disabled by default, but can be enabled with `set_option guard_msgs.diff true`. Depending on user feedback, this option may default to `true` in a future version of Lean.
- `do` **notation**
  - #3820 makes it an error to lift `(<- ...)` out of a pure `if ... then ... else ...`
- **Lazy discrimination trees**
  - #3610 fixes a name collision for `LazyDiscrTree` that could lead to cache poisoning.
  - #3677 simplifies and fixes `LazyDiscrTree` handling for `exact?`/`apply?`.
  - #3685 moves general `exact?`/`apply?` functionality into `LazyDiscrTree`.
  - #3769 has lemma selection improvements for `rw?` and `LazyDiscrTree`.
  - #3818 improves ordering of matches.

- #3590 adds `inductive.autoPromoteIndices` option to be able to disable auto promotion of indices in the `inductive` command.
- **Miscellaneous bug fixes and improvements**
  - #3606 preserves `cache` and `dischargeDepth` fields in `Lean.Meta.Simp.Result.mkEqSymm`.
  - #3633 makes `elabTermEnsuringType` respect `errToSorry`, improving error recovery of the `have` tactic.
  - #3647 enables `noncomputable unsafe` definitions, for deferring implementations until later.
  - #3672 adjust namespaces of tactics.
  - #3725 fixes `Ord` derive handler for indexed inductive types with unused alternatives.
  - #3893 improves performance of derived `Ord` instances.
  - #3771 changes error reporting for failing tactic macros. Improves `rfl` error message.
  - #3745 fixes elaboration of generalized field notation if the object of the notation is an optional parameter.
  - #3799 makes commands such as `universe`, `variable`, `namespace`, etc. require that their argument appear in a later column. Commands that can optionally parse an `ident` or parse any number of `ident`s generally should require that the `ident` use `colGt`. This keeps typos in commands from being interpreted as identifiers.
  - #3815 lets the `split` tactic be used for writing code.
  - #3822 adds missing info in `induction` tactic for `with` clauses of the form `| cstr a b c => ?_`.
  - #3806 fixes `withSetOptionIn` combinator.
  - #3844 removes unused `trace.Elab.syntax` option.
  - #3896 improves hover and go-to-def for `attribute` command.
  - #3989 makes linter options more discoverable.
  - #3916 fixes go-to-def for syntax defined with `@[builtin_term_parser]`.
  - #3962 fixes how `solveByElim` handles `symm` lemmas, making `exact?`/`apply?` usable again.
  - #3968 improves the `@[deprecated]` attribute, adding `(since := "<date>")` field.
  - #3768 makes `#print` command show structure fields.
  - #3974 makes `exact?%` behave like `by exact?` rather than `by apply?`.
  - #3994 makes elaboration of `he ▸ h` notation more predictable.
  - #3991 adjusts transparency for `decreasing_trivial` macros.
  - #4092 improves performance of `binop%` and `binrel%` expression tree elaborators.

- **Docs:** #3748, #3796, #3800, #3874, #3863, #3862, #3891, #3873, #3908, #3872.

Language server and IDE extensions

- #3602 enables `import` auto-completions.
- #3608 fixes issue leanprover/vscode-lean4#392. Diagnostic ranges had an off-by-one error that would misplace goal states for example.
- #3014 introduces snapshot trees, foundational work for incremental tactics and parallelism. #3849 adds basic incrementality API.
- #3271 adds support for server-to-client requests.
- #3656 fixes jump to definition when there are conflicting names from different files. Fixes issue #1170.
- #3691, #3925, #3932 keep semantic tokens synchronized (used for semantic highlighting), with performance improvements.
- #3247 and #3730 add diagnostics to run "Restart File" when a file dependency is saved.
- #3722 uses the correct module names when displaying references.
- #3728 makes errors in header reliably appear and makes the "Import out of date" warning be at "hint" severity. #3739 simplifies the text of this warning.
- #3778 fixes #3462, where info nodes from before the cursor would be used for computing completions.
- #3985 makes trace timings appear in Infoview.

Pretty printing

- #3797 fixes the hovers over binders so that they show their types.
- #3640 and #3735: Adds attribute `@[pp_using_anonymous_constructor]` to make structures pretty print as `⟨x, y, z⟩` rather than as `{a := x, b := y, c := z}`. This attribute is applied to `Sigma`, `PSigma`, `PProd`, `Subtype`, `And`, and `Fin`.
- #3749 Now structure instances pretty print with parent structures' fields inlined. That is, if B extends A, then `{ toA := { x := 1 }, y := 2 }` now pretty prints as `{ x := 1, y := 2 }`. Setting option `pp.structureInstances.flatten` to false turns this off.
- #3737, #3744 and #3750: Option `pp.structureProjections` is renamed to `pp.fieldNotation`, and there is now a suboption `pp.fieldNotation.generalized` to enable pretty printing function applications using generalized field notation (defaults to true). Field notation can be disabled on a function-by-function basis using the `@[pp_nodot]` attribute. The notation is not used for theorems.
- #4071 fixes interaction between app unexpanders and `pp.fieldNotation.generalized`
- #3625 makes `delabConstWithSignature` (used by `#check`) have the ability to put arguments "after the colon" to avoid printing inaccessible names.

- #3798, #3978, #3798: Adds options `pp.mvars` (default: true) and `pp.mvars.withType` (default: false). When `pp.mvars` is false, expression metavariables pretty print as `?_` and universe metavariables pretty print as `_`. When `pp.mvars.withType` is true, expression metavariables pretty print with a type ascription. These can be set when using `#guard_msgs` to make tests not depend on the particular names of metavariables.
- #3917 makes binders hoverable and gives them docstrings.
- #4034 makes hovers for RHS terms in `match` expressions in the Infoview reliably show the correct term.

Library

- `Bool`/`Prop`
  - #3508 improves `simp` confluence for `Bool` and `Prop` terms.
  - Theorems: #3604
- `Nat`
  - #3579 makes `Nat.succ_eq_add_one` be a simp lemma, now that `induction`/`cases` uses `n + 1` instead of `Nat.succ n`.
  - #3808 replaces `Nat.succ` simp rules with simprocs.
  - #3876 adds faster `Nat.repr` implementation in C.
- `Int`
  - Theorems: #3890
- `UInts`
  - #3960 improves performance of upcasting.
- `Array` and `Subarray`
  - #3676 removes `Array.eraseIdxAux`, `Array.eraseIdxSzAux`, and `Array.eraseIdx'`.
  - #3648 simplifies `Array.findIdx?`.
  - #3851 renames fields of `Subarray`.
- `List`
  - #3785 upstreams tail-recursive List operations and `@[csimp]` lemmas.
- `BitVec`
  - Theorems: #3593, #3593, #3597, #3598, #3721, #3729, #3880, #4039.
  - #3884 protects `Std.BitVec`.
- `String`
  - #3832 fixes `String.splitOn`.

- #3959 adds `String.Pos.isValid`.
- #3959 UTF-8 string validation.
- #3961 adds a model implementation for UTF-8 encoding and decoding.

- `IO`
  - #4097 adds `IO.getTaskState` which returns whether a task is finished, actively running, or waiting on other Tasks to finish.

- **Refactors**
  - #3605 reduces imports for `Init.Data.Nat` and `Init.Data.Int`.
  - #3613 reduces imports for `Init.Omega.Int`.
  - #3634 upstreams `Std.Data.Nat` and #3635 upstreams `Std.Data.Int`.
  - #3790 reduces more imports for `omega`.
  - #3694 extends `GetElem` interface with `getElem!` and `getElem?` to simplify containers like `RBMap`.
  - #3865 renames `Option.toMonad` (see breaking changes below).
  - #3882 unifies `lexOrd` with `compareLex`.

- **Other fixes or improvements**
  - #3765 makes `Quotient.sound` be a `theorem`.
  - #3645 fixes `System.FilePath.parent` in the case of absolute paths.
  - #3660 `ByteArray.toUInt64LE!` and `ByteArray.toUInt64BE!` were swapped.
  - #3881, #3887 fix linearity issues in `HashMap.insertIfNew`, `HashSet.erase`, and `HashMap.erase`. The `HashMap.insertIfNew` fix improves `import` performance.
  - #3830 ensures linearity in `Parsec.many*Core`.
  - #3930 adds `FS.Stream.isTty` field.
  - #3866 deprecates `Option.toBool` in favor of `Option.isSome`.
  - #3975 upstreams `Data.List.Init` and `Data.Array.Init` material from Std.
  - #3942 adds instances that make `ac_rfl` work without Mathlib.
  - #4010 changes `Fin.induction` to use structural induction.
  - 02753f fixes bug in `reduceLeDiff` simproc.
  - #4097 adds `IO.TaskState` and `IO.getTaskState` to get the task from the Lean runtime's task manager.

- **Docs:** #3615, #3664, #3707, #3734, #3868, #3861, #3869, #3858, #3856, #3857, #3867, #3864, #3860, #3859, #3871, #3919.

Lean internals

- **Defeq and WHNF algorithms**
  - #3616 gives better support for reducing `Nat.rec` expressions.
  - #3774 add tracing for "non-easy" WHNF cases.
  - #3807 fixes an `isDefEq` performance issue, now trying structure eta *after* lazy delta reduction.
  - #3816 fixes `.yesWithDeltaI` behavior to prevent increasing transparency level when reducing projections.
  - #3837 improves heuristic at `isDefEq`.
  - #3965 improves `isDefEq` for constraints of the form `t.i =?= s.i`.
  - #3977 improves `isDefEqProj`.
  - #3981 adds universe constraint approximations to be able to solve `u =?= max u ?v` using `?v = u`. These approximations are only applied when universe constraints cannot be postponed anymore.
  - #4004 improves `isDefEqProj` during typeclass resolution.
  - #4012 adds `backward.isDefEq.lazyProjDelta` and `backward.isDefEq.lazyWhnfCore` backwards compatibility flags.
- **Kernel**
  - #3966 removes dead code.
  - #4035 fixes mismatch for `TheoremVal` between Lean and C++.
- **Discrimination trees**
  - 423fed and 3218b2: simplify handling of dependent/non-dependent pi types.
- **Typeclass instance synthesis**
  - #3638 eta-reduces synthesized instances
  - ce350f fixes a linearity issue
  - 917a31 improves performance by considering at most one answer for subgoals not containing metavariables. #4008 adds `backward.synthInstance.canonInstances` backward compatibility flag.
- **Definition processing**
  - #3661, #3767 changes automatically generated equational theorems to be named using suffix `.eq_<idx>` instead of `._eq_<idx>`, and `.eq_def` instead of `._unfold`. (See breaking changes below.) #3675 adds a mechanism to reserve names. #3803 fixes reserved name resolution inside namespaces and fixes handling of `matcher` declarations and equation lemmas.
  - #3662 causes auxiliary definitions nested inside theorems to become `defs` if they are not proofs.

- #4006 makes proposition fields of `structures` be theorems.
- #4018 makes it an error for a theorem to be `extern`.
- #4047 improves performance making equations for well-founded recursive definitions.

- **Refactors**
  - #3614 avoids unfolding in `Lean.Meta.evalNat`.
  - #3621 centralizes functionality for `Fix`/`GuessLex`/`FunInd` in the `ArgsPacker` module.
  - #3186 rewrites the UnusedVariable linter to be more performant.
  - #3589 removes coercion from `String` to `Name` (see breaking changes below).
  - #3237 removes the `lines` field from `FileMap`.
  - #3951 makes msg parameter to `throwTacticEx` optional.
- **Diagnostics**
  - #4016, #4019, #4020, #4030, #4031, c3714b, #4049 adds `set_option diagnostics true` for diagnostic counters. Tracks number of unfolded declarations, instances, reducible declarations, used instances, recursor reductions, `isDefEq` heuristic applications, among others. This option is suggested in exceptional situations, such as at deterministic timeout and maximum recursion depth.
  - 283587 adds diagnostic information for `simp`.
  - #4043 adds diagnostic information for congruence theorems.
  - #4048 display diagnostic information for `set_option diagnostics true in <tactic>` and `set_option diagnostics true in <term>`.
- **Other features**
  - #3800 adds environment extension to record which definitions use structural or well-founded recursion.
  - #3801 `trace.profiler` can now export to Firefox Profiler.
  - #3918, #3953 adds `@[builtin_doc]` attribute to make docs and location of a declaration available as a builtin.
  - #3939 adds the `lean --json` CLI option to print messages as JSON.
  - #3075 improves `test_extern` command.
  - #3970 gives monadic generalization of `FindExpr`.
- **Docs:** #3743, #3921, #3954.
- **Other fixes:** #3622, #3726, #3823, #3897, #3964, #3946, #4007, #4026.

Compiler, runtime, and FFI

- #3632 makes it possible to allocate and free thread-local runtime resources for threads not started by Lean itself.
- #3627 improves error message about compacting closures.
- #3692 fixes deadlock in `IO.Promise.resolve`.
- #3753 catches error code from `MoveFileEx` on Windows.
- #4028 fixes a double `reset` bug in `ResetReuse` transformation.
- 6e731b removes `interpreter` copy constructor to avoid potential memory safety issues.

Lake

- **TOML Lake configurations**. #3298, #4104.

  Lake packages can now use TOML as a alternative configuration file format instead of Lean. If the default `lakefile.lean` is missing, Lake will also look for a `lakefile.toml`. The TOML version of the configuration supports a restricted set of the Lake configuration options, only including those which can easily mapped to a TOML data structure. The TOML syntax itself fully compiles with the TOML v1.0.0 specification.

  As part of the introduction of this new feature, we have been helping maintainers of some major packages within the ecosystem switch to this format. For example, the following is Aesop's new `lakefile.toml`:

  leanprover-community/aesop/lakefile.toml

  ```
  name = "aesop"
  defaultTargets = ["Aesop"]
  testRunner = "test"
  precompileModules = false

  [[require]]
  name = "batteries"
  git = "https://github.com/leanprover-community/batteries"
  rev = "main"

  [[lean_lib]]
  name = "Aesop"

  [[lean_lib]]
  name = "AesopTest"
  globs = ["AesopTest.+"]
  leanOptions = {linter.unusedVariables = false}

  [[lean_exe]]
  name = "test"
  srcDir = "scripts"
  ```

  To assist users who wish to transition their packages between configuration file formats, there is also a new `lake translate-config` command for migrating to/from TOML.

Running `lake translate-config toml` will produce a `lakefile.toml` version of a package's `lakefile.lean`. Any configuration options unsupported by the TOML format will be discarded during translation, but the original `lakefile.lean` will remain so that you can verify the translation looks good before deleting it.

- **Build progress overhaul.** #3835, #4115, #4127, #4220, #4232, #4236.

  Builds are now managed by a top-level Lake build monitor, this makes the output of Lake builds more standardized and enables producing prettier and more configurable progress reports.

  As part of this change, job isolation has improved. Stray I/O and other build related errors in custom targets are now properly isolated and caught as part of their job. Import errors no longer cause Lake to abort the entire build and are instead localized to the build jobs of the modules in question.

  Lake also now uses ANSI escape sequences to add color and produce progress lines that update in-place; this can be toggled on and off using `--ansi` / `--no-ansi`.

  `--wfail` and `--iofail` options have been added that causes a build to fail if any of the jobs log a warning (`--wfail`) or produce any output or log information messages (`--iofail`). Unlike some other build systems, these options do **NOT** convert these logs into errors, and Lake does not abort jobs on such a log (i.e., dependent jobs will still continue unimpeded).

- `lake test`. #3779.

  Lake now has a built-in `test` command which will run a script or executable labelled `@[test_runner]` (in Lean) or defined as the `testRunner` (in TOML) in the root package.

  Lake also provides a `lake check-test` command which will exit with code `0` if the package has a properly configured test runner or error with `1` otherwise.

- `lake lean`. #3793.

  The new command `lake lean <file> [-- <args...>]` functions like `lake env lean <file> <args...>`, except that it builds the imports of `file` before running `lean`. This makes it very useful for running test or example code that imports modules that are not guaranteed to have been built beforehand.

- **Miscellaneous bug fixes and improvements**

  - #3609 `LEAN_GITHASH` environment variable to override the detected Git hash for Lean when computing traces, useful for testing custom builds of Lean.
  - #3795 improves relative package directory path normalization in the pre-rename check.
  - #3957 fixes handling of packages that appear multiple times in a dependency tree.
  - #3999 makes it an error for there to be a mismatch between a package name and what it is required as. Also adds a special message for the `std`-to-`batteries` rename.
  - #4033 fixes quiet mode.

- **Docs:** #3704.

DevOps

- #3536 and #3833 add a checklist for the release process.
- #3600 runs nix-ci more uniformly.
- #3612 avoids argument limits when building on Windows.
- #3682 builds Lean's `.o` files in parallel to rest of core.
- #3601 changes the way Lean is built on Windows (see breaking changes below). As a result, Lake now dynamically links executables with `supportInterpreter := true` on Windows to `libleanshared.dll` and `libInit_shared.dll`. Therefore, such executables will not run unless those shared libraries are co-located with the executables or part of PATH. Running the executable via `lake exe` will ensure these libraries are part of PATH.

  In a related change, the signature of the `nativeFacets` Lake configuration options has changed from a static `Array` to a function `(shouldExport : Bool) → Array`. See its docstring or Lake's README for further details on the changed option.
- #3690 marks "Build matrix complete" as canceled if the build is canceled.
- #3700, #3702, #3701, #3834, #3923: fixes and improvements for std and mathlib CI.
- #3712 fixes `nix build .` on macOS.
- #3717 replaces `shell.nix` in devShell with `flake.nix`.
- #3715 and #3790 add test result summaries.
- #3971 prevents stage0 changes via the merge queue.
- #3979 adds handling for `changes-stage0` label.
- #3952 adds a script to summarize GitHub issues.
- 18a699 fixes asan linking

Breaking changes

- Due to the major Lake build refactor, code using the affected parts of the Lake API or relying on the previous output format of Lake builds is likely to have been broken. We have tried to minimize the breakages and, where possible, old definitions have been marked `@[deprecated]` with a reference to the new alternative.
- Executables configured with `supportInterpreter := true` on Windows should now be run via `lake exe` to function properly.
- Automatically generated equational theorems are now named using suffix `.eq_<idx>` instead of `._eq_<idx>`, and `.eq_def` instead of `._unfold`. Example:

```
def fact : Nat → Nat
  | 0 => 1
  | n+1 => (n+1) * fact n
```

```
theorem ex : fact 0 = 1 := by unfold fact; decide

#check fact.eq_1
-- fact.eq_1 : fact 0 = 1

#check fact.eq_2
-- fact.eq_2 (n : Nat) : fact (Nat.succ n) = (n + 1) * fact n

#check fact.eq_def
/-
fact.eq_def :
  ∀ (x : Nat),
    fact x =
      match x with
      | 0 => 1
      | Nat.succ n => (n + 1) * fact n
-/
```

- The coercion from `String` to `Name` was removed. Previously, it was `Name.mkSimple`, which does not separate strings at dots, but experience showed that this is not always the desired coercion. For the previous behavior, manually insert a call to `Name.mkSimple`.
- The `Subarray` fields `as`, h_1 and h_2 have been renamed to `array`, `start_le_stop`, and `stop_le_array_size`, respectively. This more closely follows standard Lean conventions. Deprecated aliases for the field projections were added; these will be removed in a future release.
- The change to the instance name algorithm (described above) can break projects that made use of the auto-generated names.
- `Option.toMonad` has been renamed to `Option.getM` and the unneeded `[Monad m]` instance argument has been removed.

Lean 4.7.0 (2024-04-03)

- `simp` and `rw` now use instance arguments found by unification, rather than always resynthesizing. For backwards compatibility, the original behaviour is available via `set_option tactic.skipAssignedInstances false`. #3507 and #3509.
- When the `pp.proofs` is false, now omitted proofs use ⋯ rather than `_`, which gives a more helpful error message when copied from the Infoview. The `pp.proofs.threshold` option lets small proofs always be pretty printed. #3241.
- `pp.proofs.withType` is now set to false by default to reduce noise in the info view.
- The pretty printer for applications now handles the case of over-application itself when applying app unexpanders. In particular, the `| `($_ $a $b $xs*) => `(($a + $b) $xs*)` case of an `app_unexpander` is no longer necessary. #3495.
- New `simp` (and `dsimp`) configuration option: `zetaDelta`. It is `false` by default. The `zeta` option is still `true` by default, but their meaning has changed.

- When `zeta := true`, `simp` and `dsimp` reduce terms of the form `let x := val; e[x]` into `e[val]`.
- When `zetaDelta := true`, `simp` and `dsimp` will expand let-variables in the context. For example, suppose the context contains `x := val`. Then, any occurrence of `x` is replaced with `val`.

See issue #2682 for additional details. Here are some examples:

```
example (h : z = 9) : let x := 5; let y := 4; x + y = z := by
  intro x
  simp
  /-
  New goal:
  h : z = 9; x := 5 |- x + 4 = z
  -/
  rw [h]

example (h : z = 9) : let x := 5; let y := 4; x + y = z := by
  intro x
  -- Using both `zeta` and `zetaDelta`.
  simp (config := { zetaDelta := true })
  /-
  New goal:
  h : z = 9; x := 5 |- 9 = z
  -/
  rw [h]

example (h : z = 9) : let x := 5; let y := 4; x + y = z := by
  intro x
  simp [x] -- asks `simp` to unfold `x`
  /-
  New goal:
  h : z = 9; x := 5 |- 9 = z
  -/
  rw [h]

example (h : z = 9) : let x := 5; let y := 4; x + y = z := by
  intro x
  simp (config := { zetaDelta := true, zeta := false })
  /-
  New goal:
  h : z = 9; x := 5 |- let y := 4; 5 + y = z
  -/
  rw [h]
```

- When adding new local theorems to `simp`, the system assumes that the function application arguments have been annotated with `no_index`. This modification, which addresses issue #2670, restores the Lean 3 behavior that users expect. With this modification, the following examples are now operational:

```
example {α β : Type} {f : α × β → β → β} (h : ∀ p : α × β, f p
p.2 = p.2)
  (a : α) (b : β) : f (a, b) b = b := by
  simp [h]

example {α β : Type} {f : α × β → β → β}
  (a : α) (b : β) (h : f (a,b) (a,b).2 = (a,b).2) : f (a, b) b =
b := by
  simp [h]
```

In both cases, h is applicable because `simp` does not index f-arguments anymore when adding h to the `simp`-set. It's important to note, however, that global theorems continue to be indexed in the usual manner.

- Improved the error messages produced by the `decide` tactic. #3422
- Improved auto-completion performance. #3460
- Improved initial language server startup performance. #3552
- Changed call hierarchy to sort entries and strip private header from names displayed in the call hierarchy. #3482
- There is now a low-level error recovery combinator in the parsing framework, primarily intended for DSLs. #3413
- You can now write `termination_by?` after a declaration to see the automatically inferred termination argument, and turn it into a `termination_by …` clause using the "Try this" widget or a code action. #3514
- A large fraction of `Std` has been moved into the Lean repository. This was motivated by:

  1. Making universally useful tactics such as `ext`, `by_cases`, `change at`, `norm_cast`, `rcases`, `simpa`, `simp?`, `omega`, and `exact?` available to all users of Lean, without imports.
  2. Minimizing the syntactic changes between plain Lean and Lean with `import Std`.
  3. Simplifying the development process for the basic data types `Nat`, `Int`, `Fin` (and variants such as `UInt64`), `List`, `Array`, and `BitVec` as we begin making the APIs and simp normal forms for these types more complete and consistent.
  4. Laying the groundwork for the Std roadmap, as a library focused on essential datatypes not provided by the core language (e.g. `RBMap`) and utilities such as basic IO. While we have achieved most of our initial aims in `v4.7.0-rc1`, some upstreaming will continue over the coming months.

- The `/` and `%` notations in `Int` now use `Int.ediv` and `Int.emod` (i.e. the rounding conventions have changed). Previously `Std` overrode these notations, so this is no change for users of `Std`. There is now kernel support for these functions. #3376.
- `omega`, our integer linear arithmetic tactic, is now available in the core language.
  - It is supplemented by a preprocessing tactic `bv_omega` which can solve goals about `BitVec` which naturally translate into linear arithmetic problems. #3435.

- omega now has support for `Fin` #3427, the `<<<` operator #3433.
- During the port omega was modified to no longer identify atoms up to definitional equality (so in particular it can no longer prove `id x ≤ x`). #3525. This may cause some regressions. We plan to provide a general purpose preprocessing tactic later, or an `omega!` mode.
- omega is now invoked in Lean's automation for termination proofs #3503 as well as in array indexing proofs #3515. This automation will be substantially revised in the medium term, and while omega does help automate some proofs, we plan to make this much more robust.

- The library search tactics `exact?` and `apply?` that were originally in Mathlib are now available in Lean itself. These use the implementation using lazy discrimination trees from `Std`, and thus do not require a disk cache but have a slightly longer startup time. The order used for selection lemmas has changed as well to favor goals purely based on how many terms in the head pattern match the current goal.
- The `solve_by_elim` tactic has been ported from `Std` to Lean so that library search can use it.
- New `#check_tactic` and `#check_simp` commands have been added. These are useful for checking tactics (particularly `simp`) behave as expected in test suites.
- Previously, app unexpanders would only be applied to entire applications. However, some notations produce functions, and these functions can be given additional arguments. The solution so far has been to write app unexpanders so that they can take an arbitrary number of additional arguments. However this leads to misleading hover information in the Infoview. For example, while `HAdd.hAdd f g 1` pretty prints as `(f + g) 1`, hovering over `f + g` shows `f`. There is no way to fix the situation from within an app unexpander; the expression position for `HAdd.hAdd f g` is absent, and app unexpanders cannot register TermInfo.

  This commit changes the app delaborator to try running app unexpanders on every prefix of an application, from longest to shortest prefix. For efficiency, it is careful to only try this when app delaborators do in fact exist for the head constant, and it also ensures arguments are only delaborated once. Then, in `(f + g) 1`, the `f + g` gets TermInfo registered for that subexpression, making it properly hoverable.

  #3375

Breaking changes:

- `Lean.withTraceNode` and variants got a stronger `MonadAlwaysExcept` assumption to fix trace trees not being built on elaboration runtime exceptions. Instances for most elaboration monads built on `EIO Exception` should be synthesized automatically.
- The `match ... with.` and `fun.` notations previously in Std have been replaced by `nomatch ...` and `nofun`. #3279 and #3286

Other improvements:

- several bug fixes for `simp`:
  - we should not crash when `simp` loops #3269
  - `simp` gets stuck on `autoParam` #3315
  - `simp` fails when custom discharger makes no progress #3317
  - `simp` fails to discharge `autoParam` premises even when it can reduce them to `True` #3314
  - `simp?` suggests generated equations lemma names, fixes #3547 #3573
- fixes for `match` expressions:
  - fix regression with builtin literals #3521
  - accept `match` when patterns cover all cases of a `BitVec` finite type #3538
  - fix matching `Int` literals #3504
  - patterns containing int values and constructors #3496
- improve `termination_by` error messages #3255
- fix `rename_i` in macros, fixes #3553 #3581
- fix excessive resource usage in `generalize`, fixes #3524 #3575
- an equation lemma with autoParam arguments fails to rewrite, fixing #2243 #3316
- `add_decl_doc` should check that declarations are local #3311
- instantiate the types of inductives with the right parameters, closing #3242 #3246
- New simprocs for many basic types. #3407

Lake fixes:

- Warn on fetch cloud release failure #3401
- Cloud release trace & `lake build :release` errors #3248

Lean 4.6.0 (2024-02-29)

- Add custom simplification procedures (aka `simprocs`) to `simp`. Simprocs can be triggered by the simplifier on a specified term-pattern. Here is an small example:

```
import Lean.Meta.Tactic.Simp.BuiltinSimprocs.Nat

def foo (x : Nat) : Nat :=
  x + 10

/--
The `simproc` `reduceFoo` is invoked on terms that match the
pattern `foo _`.
-/
simproc reduceFoo (foo _) :=
```

```
  /- A term of type `Expr → SimpM Step -/
  fun e => do
    /-
    The `Step` type has three constructors: `.done`, `.visit`,
`.continue`.
    * The constructor `.done` instructs `simp` that the result
does
      not need to be simplified further.
    * The constructor `.visit` instructs `simp` to visit the
resulting expression.
    * The constructor `.continue` instructs `simp` to try other
simplification procedures.

    All three constructors take a `Result`. The `.continue`
constructor may also take `none`.
    `Result` has two fields `expr` (the new expression), and
`proof?` (an optional proof).
     If the new expression is definitionally equal to the input
one, then `proof?` can be omitted or set to `none`.
    -/
    /- `simp` uses matching modulo reducibility. So, we ensure
the term is a `foo`-application. -/
    unless e.isAppOfArity ``foo 1 do
      return .continue
    /- `Nat.fromExpr?` tries to convert an expression into a
`Nat` value -/
    let some n ← Nat.fromExpr? e.appArg!
      | return .continue
    return .done { expr := Lean.mkNatLit (n+10) }
```

We disable simprocs support by using the command `set_option simprocs false`. This command is particularly useful when porting files to v4.6.0. Simprocs can be scoped, manually added to `simp` commands, and suppressed using `-`. They are also supported by `simp?`. `simp only` does not execute any `simproc`. Here are some examples for the `simproc` defined above.

```
example : x + foo 2 = 12 + x := by
  set_option simprocs false in
    /- This `simp` command does not make progress since
`simproc`s are disabled. -/
    fail_if_success simp
  simp_arith

example : x + foo 2 = 12 + x := by
  /- `simp only` must not use the default simproc set. -/
  fail_if_success simp only
  simp_arith

example : x + foo 2 = 12 + x := by
  /-
  `simp only` does not use the default simproc set,
  but we can provide simprocs as arguments. -/
```

```
  simp only [reduceFoo]
  simp_arith

example : x + foo 2 = 12 + x := by
  /- We can use `-` to disable `simproc`s. -/
  fail_if_success simp [-reduceFoo]
  simp_arith
```

The command `register_simp_attr <id>` now creates a `simp` **and** a `simproc` set with the name `<id>`. The following command instructs Lean to insert the `reduceFoo` simplification procedure into the set `my_simp`. If no set is specified, Lean uses the default `simp` set.

```
simproc [my_simp] reduceFoo (foo _) := ...
```

- The syntax of the `termination_by` and `decreasing_by` termination hints is overhauled:
  - They are now placed directly after the function they apply to, instead of after the whole `mutual` block.
  - Therefore, the function name no longer has to be mentioned in the hint.
  - If the function has a `where` clause, the `termination_by` and `decreasing_by` for that function come before the `where`. The functions in the `where` clause can have their own termination hints, each following the corresponding definition.
  - The `termination_by` clause can only bind "extra parameters", that are not already bound by the function header, but are bound in a lambda (`:= fun x y z =>`) or in patterns (`| x, n + 1 => …`). These extra parameters used to be understood as a suffix of the function parameters; now it is a prefix.

  Migration guide: In simple cases just remove the function name, and any variables already bound at the header.

```
 def foo : Nat → Nat → Nat := …
-termination_by foo a b => a - b
+termination_by a b => a - b
```

  or

```
 def foo : Nat → Nat → Nat := …
-termination_by _ a b => a - b
+termination_by a b => a - b
```

  If the parameters are bound in the function header (before the `:`), remove them as well:

```
 def foo (a b : Nat) : Nat := …
-termination_by foo a b => a - b
+termination_by a - b
```

  Else, if there are multiple extra parameters, make sure to refer to the right ones; the bound variables are interpreted from left to right, no longer from right to left:

```
 def foo : Nat → Nat → Nat → Nat
   | a, b, c => …
-termination_by foo b c => b
+termination_by a b => b
```

In the case of a `mutual` block, place the termination arguments (without the function name) next to the function definition:

```
-mutual
-def foo : Nat → Nat → Nat := …
-def bar : Nat → Nat := …
-end
-termination_by
-  foo a b => a - b
-  bar a => a
+mutual
+def foo : Nat → Nat → Nat := …
+termination_by a b => a - b
+def bar : Nat → Nat := …
+termination_by a => a
+end
```

Similarly, if you have (mutual) recursion through `where` or `let rec`, the termination hints are now placed directly after the function they apply to:

```
-def foo (a b : Nat) : Nat := …
-  where bar (x : Nat) : Nat := …
-termination_by
-  foo a b => a - b
-  bar x => x
+def foo (a b : Nat) : Nat := …
+termination_by a - b
+  where
+    bar (x : Nat) : Nat := …
+    termination_by x

-def foo (a b : Nat) : Nat :=
-  let rec bar (x : Nat) :  Nat := …
-  …
-termination_by
-  foo a b => a - b
-  bar x => x
+def foo (a b : Nat) : Nat :=
+  let rec bar (x : Nat) : Nat := …
+    termination_by x
+  …
+termination_by a - b
```

In cases where a single `decreasing_by` clause applied to multiple mutually recursive functions before, the tactic now has to be duplicated.

- The semantics of `decreasing_by` changed; the tactic is applied to all termination proof goals together, not individually.

  This helps when writing termination proofs interactively, as one can focus each subgoal individually, for example using `·`. Previously, the given tactic script had to work for *all* goals, and one had to resort to tactic combinators like `first`:

  ```
   def foo (n : Nat) := … foo e1 … foo e2 …
  -decreasing_by
  -simp_wf
  -first | apply something_about_e1; …
  -      | apply something_about_e2; …
  +decreasing_by
  +all_goals simp_wf
  +· apply something_about_e1; …
  +· apply something_about_e2; …
  ```

  To obtain the old behaviour of applying a tactic to each goal individually, use `all_goals`:

  ```
   def foo (n : Nat) := …
  -decreasing_by some_tactic
  +decreasing_by all_goals some_tactic
  ```

  In the case of mutual recursion each `decreasing_by` now applies to just its function. If some functions in a recursive group do not have their own `decreasing_by`, the default `decreasing_tactic` is used. If the same tactic ought to be applied to multiple functions, the `decreasing_by` clause has to be repeated at each of these functions.

- Modify `InfoTree.context` to facilitate augmenting it with partial contexts while elaborating a command. This breaks backwards compatibility with all downstream projects that traverse the `InfoTree` manually instead of going through the functions in `InfoUtils.lean`, as well as those manually creating and saving `InfoTrees`. See PR #3159 for how to migrate your code.

- Add language server support for call hierarchy requests (PR #3082). The change to the .ilean format in this PR means that projects must be fully rebuilt once in order to generate .ilean files with the new format before features like "find references" work correctly again.

- Structure instances with multiple sources (for example `{a, b, c with x := 0}`) now have their fields filled from these sources in strict left-to-right order. Furthermore, the structure instance elaborator now aggressively use sources to fill in subobject fields, which prevents unnecessary eta expansion of the sources, and hence greatly reduces the reliance on costly structure eta reduction. This has a large impact on mathlib, reducing total CPU instructions by 3% and enabling impactful refactors like leanprover-community/mathlib4#8386 which reduces the build time by almost 20%. See PR #2478 and RFC #2451.

- Add pretty printer settings to omit deeply nested terms (`pp.deepTerms false` and `pp.deepTerms.threshold`) (PR #3201)

- Add pretty printer options `pp.numeralTypes` and `pp.natLit`. When `pp.numeralTypes` is true, then natural number literals, integer literals, and rational number literals are pretty printed with type ascriptions, such as `(2 : Rat)`, `(-2 : Rat)`, and `(-2 / 3 : Rat)`. When `pp.natLit` is true, then raw natural number literals are pretty printed as `nat_lit 2`. PR #2933 and RFC #3021.

Lake updates:

- improved platform information & control #3226
- `lake update` from unsupported manifest versions #3149

Other improvements:

- make `intro` be aware of `let_fun` #3115
- produce simpler proof terms in `rw` #3121
- fuse nested `mkCongrArg` calls in proofs generated by `simp` #3203
- `induction using` followed by a general term #3188
- allow generalization in `let` #3060, fixing #3065
- reducing out-of-bounds `swap!` should return `a`, not ``default`` #3197, fixing #3196
- derive `BEq` on structure with `Prop`-fields #3191, fixing #3140
- refine through more `casesOnApp`/`matcherApp` #3176, fixing #3175
- do not strip dotted components from lean module names #2994, fixing #2999
- fix `deriving` only deriving the first declaration for some handlers #3058, fixing #3057
- do not instantiate metavariables in kabstract/rw for disallowed occurrences #2539, fixing #2538
- hover info for `cases h : ...` #3084

Lean 4.5.0 (2024-02-01)

- Modify the lexical syntax of string literals to have string gaps, which are escape sequences of the form `"\" newline whitespace*`. These have the interpretation of an empty string and allow a string to flow across multiple lines without introducing additional whitespace. The following is equivalent to `"this is a string"`.

  ```
  "this is \
     a string"
  ```

  PR #2821 and RFC #2838.

- Add raw string literal syntax. For example, `r"\n"` is equivalent to `"\\n"`, with no escape processing. To include double quote characters in a raw string one can add sufficiently many # characters before and after the bounding "s, as in `r#"the "the" is in quotes"#` for `"the \"the\" is in quotes"`. PR #2929 and issue #1422.

- The low-level `termination_by'` clause is no longer supported.

  Migration guide: Use `termination_by` instead, e.g.:

  ```
  -termination_by' measure (fun ⟨i, _⟩ => as.size - i)
  +termination_by i _ => as.size - i
  ```

  If the well-founded relation you want to use is not the one that the `WellFoundedRelation` type class would infer for your termination argument, you can use `WellFounded.wrap` from the std library to explicitly give one:

  ```
  -termination_by' ⟨r, hwf⟩
  +termination_by x => hwf.wrap x
  ```

- Support snippet edits in LSP `TextEdits`. See `Lean.Lsp.SnippetString` for more details.
- Deprecations and changes in the widget API.
  - `Widget.UserWidgetDefinition` is deprecated in favour of `Widget.Module`. The annotation `@[widget]` is deprecated in favour of `@[widget_module]`. To migrate a definition of type `UserWidgetDefinition`, remove the `name` field and replace the type with `Widget.Module`. Removing the `name` results in a title bar no longer being drawn above your panel widget. To add it back, draw it as part of the component using `<details open=true><summary class='mv2 pointer'>{name}</summary>{rest_of_widget}</details>`. See an example migration here.
  - The new command `show_panel_widgets` allows displaying always-on and locally-on panel widgets.
  - `RpcEncodable` widget props can now be stored in the infotree.
  - See RFC 2963 for more details and motivation.
- If no usable lexicographic order can be found automatically for a termination proof, explain why. See feat: GuessLex: if no measure is found, explain why.
- Option to print inferred termination argument. With `set_option showInferredTerminationBy true` you will get messages like

  ```
  Inferred termination argument:
  termination_by
  ackermann n m => (sizeOf n, sizeOf m)
  ```

  for automatically generated `termination_by` clauses.
- More detailed error messages for invalid mutual blocks.
- Multiple improvements to the output of `simp?` and `simp_all?`.
- Tactics with `withLocation *` no longer fail if they close the main goal.
- Implementation of a `test_extern` command for writing tests for `@[extern]` and `@[implemented_by]` functions. Usage is

```
import Lean.Util.TestExtern

test_extern Nat.add 17 37
```

The head symbol must be the constant with the `@[extern]` or `@[implemented_by]` attribute. The return type must have a `DecidableEq` instance.

Bug fixes for #2853, #2953, #2966, #2971, #2990, #3094.

Bug fix for eager evaluation of default value in `Option.getD`. Avoid panic in `leanPosToLspPos` when file source is unavailable. Improve short-circuiting behavior for `List.all` and `List.any`.

Several Lake bug fixes: #3036, #3064, #3069.

Lean 4.4.0 (2023-12-31)

- Lake and the language server now support per-package server options using the `moreServerOptions` config field, as well as options that apply to both the language server and `lean` using the `leanOptions` config field. Setting either of these fields instead of `moreServerArgs` ensures that viewing files from a dependency uses the options for that dependency. Additionally, `moreServerArgs` is being deprecated in favor of the `moreGlobalServerArgs` field. See PR #2858.

  A Lakefile with the following deprecated package declaration:

  ```
  def moreServerArgs := #[
    "-Dpp.unicode.fun=true"
  ]
  def moreLeanArgs := moreServerArgs

  package SomePackage where
    moreServerArgs := moreServerArgs
    moreLeanArgs := moreLeanArgs
  ```

  ... can be updated to the following package declaration to use per-package options:

  ```
  package SomePackage where
    leanOptions := #[⟨`pp.unicode.fun, true⟩]
  ```

- Rename request handler.
- Import auto-completion.
- `pp.beta`` to apply beta reduction when pretty printing.
- Embed and check githash in .olean.
- Guess lexicographic order for well-founded recursion.
- Allow trailing comma in tuples, lists, and tactics.

Bug fixes for #2628, #2883, #2810, #2925, and #2914.

Lake:

- `lake init .` and a bare `lake init` and will now use the current directory as the package name. #2890
- `lake new` and `lake init` will now produce errors on invalid package names such as `..`, `foo/bar`, `Init`, `Lean`, `Lake`, and `Main`. See issue #2637 and PR #2890.
- `lean_lib` no longer converts its name to upper camel case (e.g., `lean_lib bar` will include modules named `bar.*` rather than `Bar.*`). See issue #2567 and PR #2889.
- Lean and Lake now properly support non-identifier library names (e.g., `lake new 123-hello` and `import «123Hello»` now work correctly). See issue #2865 and PR #2889.
- Lake now filters the environment extensions loaded from a compiled configuration (`lakefile.olean`) to include only those relevant to Lake's workspace loading process. This resolves segmentation faults caused by environment extension type mismatches (e.g., when defining custom elaborators via `elab` in configurations). See issue #2632 and PR #2896.
- Cloud releases will now properly be re-unpacked if the build directory is removed. See PR #2928.
- Lake's `math` template has been simplified. See PR #2930.
- `lake exe <target>` now parses `target` like a build target (as the help text states it should) rather than as a basic name. For example, `lake exe @mathlib/runLinter` should now work. See PR #2932.
- `lake new foo.bar [std]` now generates executables named `foo-bar` and `lake new foo.bar exe` properly creates `foo/bar.lean`. See PR #2932.
- Later packages and libraries in the dependency tree are now preferred over earlier ones. That is, the later ones "shadow" the earlier ones. Such an ordering is more consistent with how declarations generally work in programming languages. This will break any package that relied on the previous ordering. See issue #2548 and PR #2937.
- Executable roots are no longer mistakenly treated as importable. They will no longer be picked up by `findModule?`. See PR #2937.

Lean 4.3.0 (2023-11-30)

- `simp [f]` does not unfold partial applications of `f` anymore. See issue #2042. To fix proofs affected by this change, use `unfold f` or `simp (config := { unfoldPartialApp := true }) [f]`.
- By default, `simp` will no longer try to use Decidable instances to rewrite terms. In particular, not all decidable goals will be closed by `simp`, and the `decide` tactic may be useful in such cases. The `decide` simp configuration option can be used to locally restore the old `simp` behavior, as in `simp (config := {decide := true})`; this includes using Decidable instances to verify side goals such as numeric inequalities.
- Many bug fixes:

- Add left/right actions to term tree coercion elaborator and make `^` a right action
- Fix for #2775, don't catch max recursion depth errors
- Reduction of `Decidable` instances very slow when using `cases` tactic
- `simp` not rewriting in binder
- `simp` unfolding `let` even with `zeta := false` option
- `simp` (with beta/zeta disabled) and discrimination trees
- unknown free variable introduced by `rw ... at h`
- `dsimp` doesn't use `rfl` theorems which consist of an unapplied constant
- `dsimp` does not close reflexive equality goals if they are wrapped in metadata
- `rw [h]` uses h from the environment in preference to h from the local context
- missing `withAssignableSyntheticOpaque` for `assumption` tactic
- ignoring default value for field warning

- Cancel outstanding tasks on document edit in the language server.
- Remove unnecessary `%` operations in `Fin.mod` and `Fin.div`
- Avoid `DecidableEq` in `Array.mem`
- Ensure `USize.size` unifies with `?m + 1`
- Improve compatibility with emacs eglot client

Lake:

- Sensible defaults for `lake new MyProject math`
- Changed `postUpdate?` configuration option to a `post_update` declaration. See the `post_update` syntax docstring for more information on the new syntax.
- A manifest is automatically created on workspace load if one does not exists..
- The `:=` syntax for configuration declarations (i.e., `package`, `lean_lib`, and `lean_exe`) has been deprecated. For example, `package foo := {...}` is deprecated.
- support for overriding package URLs via `LAKE_PKG_URL_MAP`
- Moved the default build directory (e.g., `build`), default packages directory (e.g., `lake-packages`), and the compiled configuration (e.g., `lakefile.olean`) into a new dedicated directory for Lake outputs, `.lake`. The cloud release build archives are also stored here, fixing #2713.
- Update manifest format to version 7 (see lean4#2801 for details on the changes).
- Deprecate the `manifestFile` field of a package configuration.
- There is now a more rigorous check on `lakefile.olean` compatibility (see #2842 for more details).

Lean 4.2.0 (2023-10-31)

- isDefEq cache for terms not containing metavariables..
- Make `Environment.mk` and `Environment.add` private, and add `replay` as a safer alternative.
- `IO.Process.output` no longer inherits the standard input of the caller.
- Do not inhibit caching of default-level `match` reduction.
- List the valid case tags when the user writes an invalid one.
- The derive handler for `DecidableEq` now handles mutual inductive types.
- Show path of failed import in Lake.
- Fix linker warnings on macOS.
- **Lake:** Add `postUpdate?` package configuration option. Used by a package to specify some code which should be run after a successful `lake update` of the package or one of its downstream dependencies. (lake#185)
- Improvements to Lake startup time (#2572, #2573)
- `refine e` now replaces the main goal with metavariables which were created during elaboration of `e` and no longer captures pre-existing metavariables that occur in `e` (#2502).
    - This is accomplished via changes to `withCollectingNewGoalsFrom`, which also affects `elabTermWithHoles`, `refine'`, `calc` (tactic), and `specialize`. Likewise, all of these now only include newly-created metavariables in their output.
    - Previously, both newly-created and pre-existing metavariables occurring in `e` were returned inconsistently in different edge cases, causing duplicated goals in the infoview (issue #2495), erroneously closed goals (issue #2434), and unintuitive behavior due to `refine e` capturing previously-created goals appearing unexpectedly in `e` (no issue; see PR).

Lean 4.1.0 (2023-09-26)

- The error positioning on missing tokens has been improved. In particular, this should make it easier to spot errors in incomplete tactic proofs.
- After elaborating a configuration file, Lake will now cache the configuration to a `lakefile.olean`. Subsequent runs of Lake will import this OLean instead of elaborating the configuration file. This provides a significant performance improvement (benchmarks indicate that using the OLean cuts Lake's startup time in half), but there are some important details to keep in mind:
    - Lake will regenerate this OLean after each modification to the `lakefile.lean` or `lean-toolchain`. You can also force a reconfigure by passing the new `--reconfigure` / `-R` option to `lake`.

  - Lake configuration options (i.e., `-K`) will be fixed at the moment of elaboration. Setting these options when `lake` is using the cached configuration will have no effect. To change options, run `lake` with `-R` / `--reconfigure`.
  - **The `lakefile.olean` is a local configuration and should not be committed to Git. Therefore, existing Lake packages need to add it to their `.gitignore`.**
- The signature of `Lake.buildO` has changed, `args` has been split into `weakArgs` and `traceArgs`. `traceArgs` are included in the input trace and `weakArgs` are not. See Lake's FFI example for a demonstration of how to adapt to this change.
- The signatures of `Lean.importModules`, `Lean.Elab.headerToImports`, and `Lean.Elab.parseImports`
- There is now an `occs` field in the configuration object for the `rewrite` tactic, allowing control of which occurrences of a pattern should be rewritten. This was previously a separate argument for `Lean.MVarId.rewrite`, and this has been removed in favour of an additional field of `Rewrite.Config`. It was not previously accessible from user tactics.

Lean 4.0.0 (2023-09-08)

- `Lean.Meta.getConst?` has been renamed. We have renamed `getConst?` to `getUnfoldableConst?` (and `getConstNoEx?` to `getUnfoldableConstNoEx?`). These were not intended to be part of the public API, but downstream projects had been using them (sometimes expecting different behaviour) incorrectly instead of `Lean.getConstInfo`.
- `dsimp` / `simp` / `simp_all` now fail by default if they make no progress.

  This can be overridden with the `(config := { failIfUnchanged := false })` option. This change was made to ease manual use of `simp` (with complicated goals it can be hard to tell if it was effective) and to allow easier flow control in tactics internally using `simp`. See the summary discussion on zulip for more details.
- `simp_all` now preserves order of hypotheses.

  In order to support the `failIfUnchanged` configuration option for `dsimp` / `simp` / `simp_all` the way `simp_all` replaces hypotheses has changed. In particular it is now more likely to preserve the order of hypotheses. See `simp_all` reorders hypotheses unnecessarily. (Previously all non-dependent propositional hypotheses were reverted and reintroduced. Now only such hypotheses which were changed, or which come after a changed hypothesis, are reverted and reintroduced. This has the effect of preserving the ordering amongst the non-dependent propositional hypotheses, but now any dependent or non-propositional hypotheses retain their position amongst the unchanged non-dependent propositional hypotheses.) This may affect proofs that use `rename_i`, `case ... =>`, or `next ... =>`.
- New `have this` implementation.

`this` is now a regular identifier again that is implicitly introduced by anonymous `have :=` for the remainder of the tactic block. It used to be a keyword that was visible in all scopes and led to unexpected behavior when explicitly used as a binder name.

- Show typeclass and tactic names in profile output.
- Make `calc` require the sequence of relation/proof-s to have the same indentation, and add `calc` alternative syntax allowing underscores `_` in the first relation.

  The flexible indentation in `calc` was often used to align the relation symbols:

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y +
y * y :=
  calc
      (x + y) * (x + y) = (x + y) * x + (x + y) * y       := by
rw [Nat.mul_add]
                        -- improper indentation
                        _ = x * x + y * x + (x + y) * y     := by
rw [Nat.add_mul]
                        _ = x * x + y * x + (x * y + y * y) := by
rw [Nat.add_mul]
                        _ = x * x + y * x + x * y + y * y   := by
rw [←Nat.add_assoc]
```

  This is no longer legal. The new syntax puts the first term right after the `calc` and each step has the same indentation:

```
example (x y : Nat) : (x + y) * (x + y) = x * x + y * x + x * y +
y * y :=
  calc (x + y) * (x + y)
    _ = (x + y) * x + (x + y) * y       := by rw [Nat.mul_add]
    _ = x * x + y * x + (x + y) * y     := by rw [Nat.add_mul]
    _ = x * x + y * x + (x * y + y * y) := by rw [Nat.add_mul]
    _ = x * x + y * x + x * y + y * y   := by rw [←Nat.add_assoc]
```

- Update Lake to latest prerelease.
- Make go-to-definition on a typeclass projection application go to the instance(s).
- Include timings in trace messages when `profiler` is true.
- Pretty-print signatures in hover and `#check <ident>`.
- Introduce parser memoization to avoid exponential behavior.
- feat: allow doSeq in `let x <- e | seq`.
- Add hover/go-to-def/refs for options.
- Add empty type ascription syntax `(e :)`.
- Make tokens in `<|>` relevant to syntax match.
- Add `linter.deprecated` option to silence deprecation warnings.
- Improve fuzzy-matching heuristics.
- Implementation-detail hypotheses.

- Hover information for `cases`/`induction` case names.
- Prefer longer parse even if unsuccessful.
- Show declaration module in hover.
- New `conv` mode structuring tactics.
- `simp` can track information and can print an equivalent `simp only`. PR #1626.
- Enforce uniform indentation in tactic blocks / do blocks. See issue #1606.
- Moved `AssocList`, `HashMap`, `HashSet`, `RBMap`, `RBSet`, `PersistentArray`, `PersistentHashMap`, `PersistentHashSet` to the Lean package. The standard library contains versions that will evolve independently to simplify bootstrapping process.
- Standard library moved to the std4 GitHub repository.
- `InteractiveGoals` now has information that a client infoview can use to show what parts of the goal have changed after applying a tactic. PR #1610.
- Add `[inheritDoc]` attribute. PR #1480.
- Expose that `panic = default`. PR #1614.
- New code generator project has started.
- Remove description argument from `register_simp_attr`. PR #1566.
- Additional concurrency primitives.
- Collapsible traces with messages.
- Hygienic resolution of namespaces.
- New `Float` functions.
- Many new doc strings have been added to declarations at `Init`.

Lean 4.0.0-m5 (2022-08-22)

This is the fifth milestone release of Lean 4. It contains many improvements and many new features. We had 1495 commits since the last milestone.

Contributors:

```
 885  Leonardo de Moura
 310  Sebastian Ullrich
  69  E.W.Ayers
  66  Wojciech Nawrocki
  49  Gabriel Ebner
  38  Mario Carneiro
  22  larsk21
  10  tydeu
   6  Ed Ayers
   6  Mariana Alanis
   4  Chris Lovett
   3  Jannis Limperg
```

```
 2  François G. Dorais
 2  Henrik Böving
 2  Jakob von Raumer
 2  Scott Morrison
 2  Siddharth
 1  Andrés Goens
 1  Arthur Paulino
 1  Connor Baker
 1  Joscha
 1  KaseQuark
 1  Lars
 1  Mac
 1  Marcus Rossel
 1  Patrick Massot
 1  Siddharth Bhat
 1  Timo
 1  Vincent de Haan
 1  William Blake
 1  Yuri de Wit
 1  ammkrn
 1  asdasd1dsadsa
 1  kzvi
```

- Update Lake to v4.0.0. See the v4.0.0 release notes for detailed changes.
- Mutual declarations in different namespaces are now supported. Example:

```
mutual
  def Foo.boo (x : Nat) :=
    match x with
    | 0 => 1
    | x + 1 => 2*Boo.bla x

  def Boo.bla (x : Nat) :=
    match x with
    | 0 => 2
    | x+1 => 3*Foo.boo x
end
```

A `namespace` is automatically created for the common prefix. Example:

```
mutual
  def Tst.Foo.boo (x : Nat) := ...
  def Tst.Boo.bla (x : Nat) := ...
end
```

expands to

```
namespace Tst
mutual
  def Foo.boo (x : Nat) := ...
  def Boo.bla (x : Nat) := ...
```

```
end
end Tst
```

- Allow users to install their own `deriving` handlers for existing type classes. See example at Simple.lean.
- Add tactic `congr (num)?`. See doc string for additional details.
- Missing doc linter
- `match`-syntax notation now checks for unused alternatives. See issue #1371.
- Auto-completion for structure instance fields. Example:

```
example : Nat × Nat := {
  f -- HERE
}
```

  `fst` now appears in the list of auto-completion suggestions.

- Auto-completion for dotted identifier notation. Example:

```
example : Nat :=
  .su -- HERE
```

  `succ` now appears in the list of auto-completion suggestions.

- `nat_lit` is not needed anymore when declaring `OfNat` instances. See issues #1389 and #875. Example:

```
inductive Bit where
  | zero
  | one

instance inst0 : OfNat Bit 0 where
  ofNat := Bit.zero

instance : OfNat Bit 1 where
  ofNat := Bit.one

example : Bit := 0
example : Bit := 1
```

- Add `[elabAsElim]` attribute (it is called `elab_as_eliminator` in Lean 3). Motivation: simplify the Mathlib port to Lean 4.
- `Trans` type class now accepts relations in `Type u`. See this Zulip issue.
- Accept unescaped keywords as inductive constructor names. Escaping can often be avoided at use sites via dot notation.

```
inductive MyExpr
  | let : ...

def f : MyExpr → MyExpr
  | .let ... => .let ...
```

- Throw an error message at parametric local instances such as `[Nat -> Decidable p]`. The type class resolution procedure cannot use this kind of local instance because the parameter does not have a forward dependency. This check can be disabled using `set_option checkBinderAnnotations false`.
- Add option `pp.showLetValues`. When set to `false`, the info view hides the value of `let`-variables in a goal. By default, it is `true` when visualizing tactic goals, and `false` otherwise. See issue #1345 for additional details.
- Add option `warningAsError`. When set to true, warning messages are treated as errors.
- Support dotted notation and named arguments in patterns. Example:

```
def getForallBinderType (e : Expr) : Expr :=
  match e with
  | .forallE (binderType := type) .. => type
  | _ => panic! "forall expected"
```

- "jump-to-definition" now works for function names embedded in the following attributes `@[implementedBy funName]`, `@[tactic parserName]`, `@[termElab parserName]`, `@[commandElab parserName]`, `@[builtinTactic parserName]`, `@[builtinTermElab parserName]`, and `@[builtinCommandElab parserName]`. See issue #1350.
- Improve `MVarId` methods discoverability. See issue #1346. We still have to add similar methods for `FVarId`, `LVarId`, `Expr`, and other objects. Many existing methods have been marked as deprecated.
- Add attribute `[deprecated]` for marking deprecated declarations. Examples:

```
def g (x : Nat) := x + 1

-- Whenever `f` is used, a warning message is generated
suggesting to use `g` instead.
@[deprecated g]
def f (x : Nat) := x + 1

#check f 0 -- warning: `f` has been deprecated, use `g` instead

-- Whenever `h` is used, a warning message is generated.
@[deprecated]
def h (x : Nat) := x + 1

#check h 0 -- warning: `h` has been deprecated
```

- Add type `LevelMVarId` (and abbreviation `LMVarId`) for universe level metavariable ids. Motivation: prevent meta-programmers from mixing up universe and expression metavariable ids.
- Improve `calc` term and tactic. See issue #1342.
- Relaxed antiquotation parsing further reduces the need for explicit `$x:p` antiquotation kind annotations.

- Add support for computed fields in inductives. Example:

```
inductive Exp
  | var (i : Nat)
  | app (a b : Exp)
with
  @[computedField] hash : Exp → Nat
    | .var i => i
    | .app a b => a.hash * b.hash + 1
```

The result of the `Exp.hash` function is then stored as an extra "computed" field in the `.var` and `.app` constructors; `Exp.hash` accesses this field and thus runs in constant time (even on dag-like values).

- Update `a[i]` notation. It is now based on the typeclass

```
class GetElem (cont : Type u) (idx : Type v) (elem : outParam
(Type w)) (dom : outParam (cont → idx → Prop)) where
  getElem (xs : cont) (i : idx) (h : dom xs i) : Elem
```

The notation `a[i]` is now defined as follows

```
macro:max x:term noWs "[" i:term "]" : term => `(getElem $x $i
(by get_elem_tactic))
```

The proof that `i` is a valid index is synthesized using the tactic `get_elem_tactic`. For example, the type `Array α` has the following instances

```
instance : GetElem (Array α) Nat α fun xs i => LT.lt i xs.size
where ...
instance : GetElem (Array α) USize α fun xs i => LT.lt i.toNat
xs.size where ...
```

You can use the notation `a[i]'h` to provide the proof manually. Two other notations were introduced: `a[i]!` and `a[i]?`, For `a[i]!`, a panic error message is produced at runtime if `i` is not a valid index. `a[i]?` has type `Option α`, and `a[i]?` evaluates to none if the index `i` is not valid. The three new notations are defined as follows:

```
@[inline] def getElem' [GetElem cont idx elem dom] (xs : cont) (i
: idx) (h : dom xs i) : elem :=
getElem xs i h

@[inline] def getElem! [GetElem cont idx elem dom] [Inhabited
elem] (xs : cont) (i : idx) [Decidable (dom xs i)] : elem :=
  if h : _ then getElem xs i h else panic! "index out of bounds"

@[inline] def getElem? [GetElem cont idx elem dom] (xs : cont) (i
: idx) [Decidable (dom xs i)] : Option elem :=
  if h : _ then some (getElem xs i h) else none

macro:max x:term noWs "[" i:term "]" noWs "?" : term =>
`(getElem? $x $i)
```

```
macro:max x:term noWs "[" i:term "]" noWs "!" : term =>
`(getElem! $x $i)
macro x:term noWs "[" i:term "]'" h:term:max : term => `(getElem'
$x $i $h)
```

See discussion on Zulip. Examples:

```
example (a : Array Int) (i : Nat) : Int :=
  a[i] -- Error: failed to prove index is valid ...

example (a : Array Int) (i : Nat) (h : i < a.size) : Int :=
  a[i] -- Ok

example (a : Array Int) (i : Nat) : Int :=
  a[i]! -- Ok

example (a : Array Int) (i : Nat) : Option Int :=
  a[i]? -- Ok

example (a : Array Int) (h : a.size = 2) : Int :=
  a[0]'(by rw [h]; decide) -- Ok

example (a : Array Int) (h : a.size = 2) : Int :=
  have : 0 < a.size := by rw [h]; decide
  have : 1 < a.size := by rw [h]; decide
  a[0] + a[1] -- Ok

example (a : Array Int) (i : USize) (h : i.toNat < a.size) : Int
:=
  a[i] -- Ok
```

The `get_elem_tactic` is defined as

```
macro "get_elem_tactic" : tactic =>
  `(first
    | get_elem_tactic_trivial
    | fail "failed to prove index is valid, ..."
   )
```

The `get_elem_tactic_trivial` auxiliary tactic can be extended using `macro_rules`. By default, it tries `trivial`, `simp_arith`, and a special case for `Fin`. In the future, it will also try `linarith`. You can extend `get_elem_tactic_trivial` using `my_tactic` as follows

```
macro_rules
| `(tactic| get_elem_tactic_trivial) => `(tactic| my_tactic)
```

Note that `Idx`'s type in `GetElem` does not depend on `Cont`. So, you cannot write the instance `instance : GetElem (Array α) (Fin ??) α fun xs i => ...`, but the Lean library comes equipped with the following auxiliary instance:

```
instance [GetElem cont Nat elem dom] : GetElem cont (Fin n) elem
fun xs i => dom xs i where
  getElem xs i h := getElem xs i.1 h
```

and helper tactic

```
macro_rules
| `(tactic| get_elem_tactic_trivial) => `(tactic| apply
Fin.val_lt_of_le; get_elem_tactic_trivial; done)
```

Example:

```
example (a : Array Nat) (i : Fin a.size) :=
  a[i] -- Ok

example (a : Array Nat) (h : n ≤ a.size) (i : Fin n) :=
  a[i] -- Ok
```

- Better support for qualified names in recursive declarations. The following is now supported:

```
namespace Nat
  def fact : Nat → Nat
  | 0 => 1
  | n+1 => (n+1) * Nat.fact n
end Nat
```

- Add support for `CommandElabM` monad at `#eval`. Example:

```
import Lean

open Lean Elab Command

#eval do
  let id := mkIdent `foo
  elabCommand (← `(def $id := 10))

#eval foo -- 10
```

- Try to elaborate do notation even if the expected type is not available. We still delay elaboration when the expected type is not available. This change is particularly useful when writing examples such as

```
#eval do
  IO.println "hello"
  IO.println "world"
```

That is, we don't have to use the idiom `#eval show IO _ from do ...` anymore. Note that auto monadic lifting is less effective when the expected type is not available. Monadic polymorphic functions (e.g., `ST.Ref.get`) also require the expected type.

- On Linux, panics now print a backtrace by default, which can be disabled by setting the environment variable `LEAN_BACKTRACE` to `0`. Other platforms are TBD.

- The `group(·) syntax` combinator is now introduced automatically where necessary, such as when using multiple parsers inside `(...)+`.
- Add "Typed Macros": syntax trees produced and accepted by syntax antiquotations now remember their syntax kinds, preventing accidental production of ill-formed syntax trees and reducing the need for explicit `:kind` antiquotation annotations. See PR for details.
- Aliases of protected definitions are protected too. Example:

```
protected def Nat.double (x : Nat) := 2*x

namespace Ex
export Nat (double) -- Add alias Ex.double for Nat.double
end Ex

open Ex
#check Ex.double -- Ok
#check double -- Error, `Ex.double` is alias for `Nat.double`
which is protected
```

- Use `IO.getRandomBytes` to initialize random seed for `IO.rand`. See discussion at this PR.
- Improve dot notation and aliases interaction. See discussion on Zulip for additional details. Example:

```
def Set (α : Type) := α → Prop
def Set.union (s₁ s₂ : Set α) : Set α := fun a => s₁ a ∨ s₂ a
def FinSet (n : Nat) := Fin n → Prop

namespace FinSet
  export Set (union) -- FinSet.union is now an alias for
`Set.union`
end FinSet

example (x y : FinSet 10) : FinSet 10 :=
  x.union y -- Works
```

- `ext` and `enter` conv tactics can now go inside let-declarations. Example:

```
example (g : Nat → Nat) (y : Nat) (h : let x := y + 1; g (0+x) =
x) : g (y + 1) = y + 1 := by
  conv at h => enter [x, 1, 1]; rw [Nat.zero_add]
  /-
    g : Nat → Nat
    y : Nat
    h : let x := y + 1;
        g x = x
    ⊢ g (y + 1) = y + 1
  -/
  exact h
```

- Add `zeta` conv tactic to expand let-declarations. Example:

```
example (h : let x := y + 1; 0 + x = y) : False := by
  conv at h => zeta; rw [Nat.zero_add]
  /-
    y : Nat
    h : y + 1 = y
    ⊢ False
  -/
  simp_arith at h
```

- Improve namespace resolution. See issue #1224. Example:

```
import Lean
open Lean Parser Elab
open Tactic -- now opens both `Lean.Parser.Tactic` and
`Lean.Elab.Tactic`
```

- Rename `constant` command to `opaque`. See discussion at Zulip.
- Extend `induction` and `cases` syntax: multiple left-hand-sides in a single alternative. This extension is very similar to the one implemented for `match` expressions. Examples:

```
inductive Foo where
  | mk1 (x : Nat) | mk2 (x : Nat) | mk3

def f (v : Foo) :=
  match v with
  | .mk1 x => x + 1
  | .mk2 x => 2*x + 1
  | .mk3   => 1

theorem f_gt_zero : f v > 0 := by
  cases v with
  | mk1 x | mk2 x => simp_arith!  -- New feature used here!
  | mk3 => decide
```

- `let/if` indentation in do blocks in now supported.
- Add unnamed antiquotation `$_` for use in syntax quotation patterns.
- Add unused variables linter. Feedback welcome!
- Lean now generates an error if the body of a declaration body contains a universe parameter that does not occur in the declaration type, nor is an explicit parameter. Examples:

```
/-
The following declaration now produces an error because `PUnit`
is universe polymorphic,
but the universe parameter does not occur in the function type
`Nat → Nat`
-/
def f (n : Nat) : Nat :=
  let aux (_ : PUnit) : Nat := n + 1
  aux ⟨⟩

/-
```

```
The following declaration is accepted because the universe
parameter was explicitly provided in the
function signature.
-/
def g.{u} (n : Nat) : Nat :=
  let aux (_ : PUnit.{u}) : Nat := n + 1
  aux ⟨⟩
```

- Add `subst_vars` tactic.
- Fix `autoParam` in structure fields lost in multiple inheritance..
- Add `[eliminator]` attribute. It allows users to specify default recursor/eliminators for the `induction` and `cases` tactics. It is an alternative for the `using` notation. Example:

```
@[eliminator] protected def recDiag {motive : Nat → Nat → Sort u}
    (zero_zero : motive 0 0)
    (succ_zero : (x : Nat) → motive x 0 → motive (x + 1) 0)
    (zero_succ : (y : Nat) → motive 0 y → motive 0 (y + 1))
    (succ_succ : (x y : Nat) → motive x y → motive (x + 1) (y +
1))
    (x y : Nat) :  motive x y :=
  let rec go : (x y : Nat) → motive x y
    | 0,     0 => zero_zero
    | x+1, 0   => succ_zero x (go x 0)
    | 0,   y+1 => zero_succ y (go 0 y)
    | x+1, y+1 => succ_succ x y (go x y)
  go x y
termination_by go x y => (x, y)

def f (x y : Nat) :=
  match x, y with
  | 0,   0   => 1
  | x+1, 0   => f x 0
  | 0,   y+1 => f 0 y
  | x+1, y+1 => f x y
termination_by f x y => (x, y)

example (x y : Nat) : f x y > 0 := by
  induction x, y <;> simp [f, *]
```

- Add support for `casesOn` applications to structural and well-founded recursion modules. This feature is useful when writing definitions using tactics. Example:

```
inductive Foo where
  | a | b | c
  | pair: Foo × Foo → Foo

def Foo.deq (a b : Foo) : Decidable (a = b) := by
  cases a <;> cases b
  any_goals apply isFalse Foo.noConfusion
  any_goals apply isTrue rfl
  case pair a b =>
    let (a₁, a₂) := a
```

```
    let (b₁, b₂) := b
    exact match deq a₁ b₁, deq a₂ b₂ with
    | isTrue h₁, isTrue h₂ => isTrue (by rw [h₁,h₂])
    | isFalse h₁, _ => isFalse (fun h => by cases h; cases (h₁
rfl))
    | _, isFalse h₂ => isFalse (fun h => by cases h; cases (h₂
rfl))
```

- `Option` is again a monad. The auxiliary type `OptionM` has been removed. See Zulip thread.
- Improve `split` tactic. It used to fail on `match` expressions of the form `match h : e with ...` where `e` is not a free variable. The failure used to occur during generalization.
- New encoding for `match`-expressions that use the `h :` notation for discriminants. The information is not lost during delaboration, and it is the foundation for a better `split` tactic. at delaboration time. Example:

  ```
  #print Nat.decEq
  /-
  protected def Nat.decEq : (n m : Nat) → Decidable (n = m) :=
  fun n m =>
    match h : Nat.beq n m with
    | true => isTrue (_ : n = m)
    | false => isFalse (_ : ¬n = m)
  -/
  ```

- `exists` tactic is now takes a comma separated list of terms.
- Add `dsimp` and `dsimp!` tactics. They guarantee the result term is definitionally equal, and only apply `rfl`-theorems.
- Fix binder information for `match` patterns that use definitions tagged with `[matchPattern]` (e.g., `Nat.add`). We now have proper binder information for the variable `y` in the following example.

  ```
  def f (x : Nat) : Nat :=
    match x with
    | 0 => 1
    | y + 1 => y
  ```

- (Fix) the default value for structure fields may now depend on the structure parameters. Example:

  ```
  structure Something (i: Nat) where
  n1: Nat := 1
  n2: Nat := 1 + i

  def s : Something 10 := {}
  example : s.n2 = 11 := rfl
  ```

- Apply `rfl` theorems at the `dsimp` auxiliary method used by `simp`. `dsimp` can be used anywhere in an expression because it preserves definitional equality.
- Refine auto bound implicit feature. It does not consider anymore unbound variables that have the same name of a declaration being defined. Example:

```
def f : f → Bool := -- Error at second `f`
  fun _ => true

inductive Foo : List Foo → Type -- Error at second `Foo`
  | x : Foo []
```

Before this refinement, the declarations above would be accepted and the second `f` and `Foo` would be treated as auto implicit variables. That is, `f : {f : Sort u} → f → Bool`, and `Foo : {Foo : Type u} → List Foo → Type`.

- Fix syntax highlighting for recursive declarations. Example

```
inductive List (α : Type u) where
  | nil : List α  -- `List` is not highlighted as a variable
anymore
  | cons (head : α) (tail : List α) : List α

def List.map (f : α → β) : List α → List β
  | []    => []
  | a::as => f a :: map f as -- `map` is not highlighted as a
variable anymore
```

- Add `autoUnfold` option to `Lean.Meta.Simp.Config`, and the following macros
  - `simp!` for `simp (config := { autoUnfold := true })`
  - `simp_arith!` for `simp (config := { autoUnfold := true, arith := true })`
  - `simp_all!` for `simp_all (config := { autoUnfold := true })`
  - `simp_all_arith!` for `simp_all (config := { autoUnfold := true, arith := true })`

  When the `autoUnfold` is set to true, `simp` tries to unfold the following kinds of definition
  - Recursive definitions defined by structural recursion.
  - Non-recursive definitions where the body is a `match`-expression. This kind of definition is only unfolded if the `match` can be reduced. Example:

```
def append (as bs : List α) : List α :=
  match as with
  | [] => bs
  | a :: as => a :: append as bs

theorem append_nil (as : List α) : append as [] = as := by
  induction as <;> simp_all!

theorem append_assoc (as bs cs : List α) : append (append as bs)
cs = append as (append bs cs) := by
  induction as <;> simp_all!
```

- Add `save` tactic for creating checkpoints more conveniently. Example:

```
example : <some-proposition> := by
  tac_1
  tac_2
  save
  tac_3
  ...
```

is equivalent to

```
example : <some-proposition> := by
  checkpoint
    tac_1
    tac_2
  tac_3
  ...
```

- Remove support for `{}` annotation from inductive datatype constructors. This annotation was barely used, and we can control the binder information for parameter bindings using the new inductive family indices to parameter promotion. Example: the following declaration using `{}`

  ```
  inductive LE' (n : Nat) : Nat → Prop where
    | refl {} : LE' n n -- Want `n` to be explicit
    | succ  : LE' n m → LE' n (m+1)
  ```

  can now be written as

  ```
  inductive LE' : Nat → Nat → Prop where
    | refl (n : Nat) : LE' n n
    | succ : LE' n m → LE' n (m+1)
  ```

  In both cases, the inductive family has one parameter and one index. Recall that the actual number of parameters can be retrieved using the command `#print`.
- Remove support for `{}` annotation in the `structure` command.
- Several improvements to LSP server. Examples: "jump to definition" in mutually recursive sections, fixed incorrect hover information in "match"-expression patterns, "jump to definition" for pattern variables, fixed auto-completion in function headers, etc.
- In `macro ... xs:p* ...` and similar macro bindings of combinators, `xs` now has the correct type `Array Syntax`
- Identifiers in syntax patterns now ignore macro scopes during matching.
- Improve binder names for constructor auto implicit parameters. Example, given the inductive datatype

  ```
  inductive Member : α → List α → Type u
    | head : Member a (a::as)
    | tail : Member a bs → Member a (b::bs)
  ```

  before:

  ```
  #check @Member.head
  -- @Member.head : {x : Type u_1} → {a : x} → {as : List x} →
  ```

```
Member a (a :: as)

now:

#check @Member.head
-- @Member.head : {α : Type u_1} → {a : α} → {as : List α} →
Member a (a :: as)
```

- Improve error message when constructor parameter universe level is too big.
- Add support for `for h : i in [start:stop] do ..` where `h : i ∈ [start:stop]`. This feature is useful for proving termination of functions such as:

```
inductive Expr where
  | app (f : String) (args : Array Expr)

def Expr.size (e : Expr) : Nat := Id.run do
  match e with
  | app f args =>
    let mut sz := 1
    for h : i in [: args.size] do
      -- h.upper : i < args.size
      sz := sz + size (args.get ⟨i, h.upper⟩)
    return sz
```

- Add tactic `case'`. It is similar to `case`, but does not admit the goal on failure. For example, the new tactic is useful when writing tactic scripts where we need to use `case'` at `first | ... | ...`, and we want to take the next alternative when `case'` fails.
- Add tactic macro

```
macro "stop" s:tacticSeq : tactic => `(repeat sorry)
```

  See discussion on Zulip.

- When displaying goals, we do not display inaccessible proposition names if they do not have forward dependencies. We still display their types. For example, the goal

```
case node.inl.node
β : Type u_1
b : BinTree β
k : Nat
v : β
left : Tree β
key : Nat
value : β
right : Tree β
ihl : BST left → Tree.find? (Tree.insert left k v) k = some v
ihr : BST right → Tree.find? (Tree.insert right k v) k = some v
h✝ : k < key
a✝³ : BST left
a✝² : ForallTree (fun k v => k < key) left
a✝¹ : BST right
```

```
a✝ : ForallTree (fun k v => key < k) right
⊢ BST left
```

is now displayed as

```
case node.inl.node
β : Type u_1
b : BinTree β
k : Nat
v : β
left : Tree β
key : Nat
value : β
right : Tree β
ihl : BST left → Tree.find? (Tree.insert left k v) k = some v
ihr : BST right → Tree.find? (Tree.insert right k v) k = some v
 : k < key
 : BST left
 : ForallTree (fun k v => k < key) left
 : BST right
 : ForallTree (fun k v => key < k) right
⊢ BST left
```

- The hypothesis name is now optional in the `by_cases` tactic.
- Fix inconsistency between `syntax` and kind names. The node kinds `numLit`, `charLit`, `nameLit`, `strLit`, and `scientificLit` are now called `num`, `char`, `name`, `str`, and `scientific` respectively. Example: we now write

  ```
  macro_rules | `($n:num) => `("hello")
  ```

  instead of

  ```
  macro_rules | `($n:numLit) => `("hello")
  ```
- (Experimental) New `checkpoint <tactic-seq>` tactic for big interactive proofs.
- Rename tactic `nativeDecide` => `native_decide`.
- Antiquotations are now accepted in any syntax. The `incQuotDepth` `syntax` parser is therefore obsolete and has been removed.
- Renamed tactic `nativeDecide` => `native_decide`.
- "Cleanup" local context before elaborating a `match` alternative right-hand-side. Examples:

  ```
  example (x : Nat) : Nat :=
    match g x with
    | (a, b) => _ -- Local context does not contain the auxiliary
  `_discr := g x` anymore

  example (x : Nat × Nat) (h : x.1 > 0) : f x > 0 := by
    match x with
    | (a, b) => _ -- Local context does not contain the `h✝ : x.fst
  > 0` anymore
  ```

- Improve `let`-pattern (and `have`-pattern) macro expansion. In the following example,

  ```
  example (x : Nat × Nat) : f x > 0 := by
    let (a, b) := x
    done
  ```

  The resulting goal is now `... |- f (a, b) > 0` instead of `... |- f x > 0`.

- Add cross-compiled aarch64 Linux and aarch64 macOS releases.

- Add tutorial-like examples to our documentation, rendered using LeanInk+Alectryon.

Lean 4.0.0-m4 (2022-03-27)

This is the fourth milestone release of Lean 4. It contains many improvements and many new features. We had more than 600 commits since the last milestone.

Contributors:

```
$ git shortlog -s -n v4.0.0-m3..v4.0.0-m4
   501  Leonardo de Moura
    65  Sebastian Ullrich
    11  Daniel Fabian
    10  larsk21
     5  Gabriel Ebner
     2  E.W.Ayers
     2  Jonathan Coates
     2  Joscha
     2  Mario Carneiro
     2  ammkrn
     1  Chris Lovett
     1  François G. Dorais
     1  Jakob von Raumer
     1  Lars
     1  Patrick Stevens
     1  Wojciech Nawrocki
     1  Xubai Wang
     1  casavaca
     1  zygi
```

- `simp` now takes user-defined simp-attributes. You can define a new `simp` attribute by creating a file (e.g., `MySimp.lean`) containing

  ```
  import Lean
  open Lean.Meta

  initialize my_ext : SimpExtension ← registerSimpAttr `my_simp "my
  own simp attribute"
  ```

  If you don't need to access `my_ext`, you can also use the macro

```
import Lean

register_simp_attr my_simp "my own simp attribute"
```

Recall that the new `simp` attribute is not active in the Lean file where it was defined. Here is a small example using the new feature.

```
import MySimp

def f (x : Nat) := x + 2
def g (x : Nat) := x + 1

@[my_simp] theorem f_eq : f x = x + 2 := rfl
@[my_simp] theorem g_eq : g x = x + 1 := rfl

example : f x + g x = 2*x + 3 := by
  simp_arith [my_simp]
```

- Extend `match` syntax: multiple left-hand-sides in a single alternative. Example:

```
def fib : Nat → Nat
| 0 | 1 => 1
| n+2 => fib n + fib (n+1)
```

This feature was discussed at issue 371. It was implemented as a macro expansion. Thus, the following is accepted.

```
inductive StrOrNum where
  | S (s : String)
  | I (i : Int)

def StrOrNum.asString (x : StrOrNum) :=
  match x with
  | I a | S a => toString a
```

- Improve `#eval` command. Now, when it fails to synthesize a `Lean.MetaEval` instance for the result type, it reduces the type and tries again. The following example now works without additional annotations

```
def Foo := List Nat

def test (x : Nat) : Foo :=
  [x, x+1, x+2]

#eval test 4
```

- `rw` tactic can now apply auto-generated equation theorems for a given definition. Example:

```
example (a : Nat) (h : n = 1) : [a].length = n := by
  rw [List.length]
  trace_state -- .. |- [].length + 1 = n
  rw [List.length]
  trace_state -- .. |- 0 + 1 = n
  rw [h]
```

- Fuzzy matching for auto completion
- Extend dot-notation `x.field` for arrow types. If type of `x` is an arrow, we look up for `Function.field`. For example, given `f : Nat → Nat` and `g : Nat → Nat`, `f.comp g` is now notation for `Function.comp f g`.
- The new `.<identifier>` notation is now also accepted where a function type is expected.

```
example (xs : List Nat) : List Nat := .map .succ xs
example (xs : List α) : Std.RBTree α ord := xs.foldl .insert ∅
```

- Add code folding support to the language server.
- Support notation `let <pattern> := <expr> | <else-case>` in do blocks.
- Remove support for "auto" `pure`. In the Zulip thread, the consensus seemed to be that "auto" `pure` is more confusing than it's worth.
- Remove restriction in `congr` theorems that all function arguments on the left-hand-side must be free variables. For example, the following theorem is now a valid `congr` theorem.

```
@[congr]
theorem dep_congr [DecidableEq ι] {p : ι → Set α} [∀ i, Inhabited
(p i)] :
                  ∀ {i j} (h : i = j) (x : p i) (y : α) (hx : x =
y), Pi.single (f := (p ·)) i x = Pi.single (f := (p ·)) j ⟨y, hx
▸ h ▸ x.2⟩ :=
```

- Partially applied congruence theorems.
- Improve elaboration postponement heuristic when expected type is a metavariable. Lean now reduces the expected type before performing the test.
- Remove deprecated leanpkg in favor of Lake now bundled with Lean.
- Various improvements to go-to-definition & find-all-references accuracy.
- Auto generated congruence lemmas with support for casts on proofs and `Decidable` instances (see wishlist).
- Rename option `autoBoundImplicitLocal` => `autoImplicit`.
- Relax auto-implicit restrictions. The command `set_option relaxedAutoImplicit false` disables the relaxations.
- `contradiction` tactic now closes the goal if there is a `False.elim` application in the target.
- Renamed tatic `byCases` => `by_cases` (motivation: enforcing naming convention).
- Local instances occurring in patterns are now considered by the type class resolution procedure. Example:

```
def concat : List ((α : Type) × ToString α × α) → String
  | [] => ""
  | ⟨_, _, a⟩ :: as => toString a ++ concat as
```

- Notation for providing the motive for `match` expressions has changed. before:

```
match x, rfl : (y : Nat) → x = y → Nat with
| 0,   h => ...
| x+1, h => ...
```

now:

```
match (motive := (y : Nat) → x = y → Nat) x, rfl with
| 0,   h => ...
| x+1, h => ...
```

With this change, the notation for giving names to equality proofs in `match`-expressions is not whitespace sensitive anymore. That is, we can now write

```
match h : sort.swap a b with
| (r₁, r₂) => ... -- `h : sort.swap a b = (r₁, r₂)`
```

- `(generalizing := true)` is the default behavior for `match` expressions even if the expected type is not a proposition. In the following example, we used to have to include `(generalizing := true)` manually.

```
inductive Fam : Type → Type 1 where
  | any : Fam α
  | nat : Nat → Fam Nat

example (a : α) (x : Fam α) : α :=
  match x with
  | Fam.any   => a
  | Fam.nat n => n
```

- We now use PSum (instead of Sum) when compiling mutually recursive definitions using well-founded recursion.
- Better support for parametric well-founded relations. See [issue #1017](). This change affects the low-level `termination_by'` hint because the fixed prefix of the function parameters in not "packed" anymore when constructing the well-founded relation type. For example, in the following definition, `as` is part of the fixed prefix, and is not packed anymore. In previous versions, the `termination_by'` term would be written as `measure fun ⟨as, i, _⟩ => as.size - i`

```
def sum (as : Array Nat) (i : Nat) (s : Nat) : Nat :=
  if h : i < as.size then
    sum as (i+1) (s + as.get ⟨i, h⟩)
  else
    s
termination_by' measure fun ⟨i, _⟩ => as.size - i
```

- Add `while <cond> do <do-block>`, `repeat <do-block>`, and `repeat <do-block> until <cond>` macros for do-block. These macros are based on `partial` definitions, and consequently are useful only for writing programs we don't want to prove anything about.

- Add `arith` option to `Simp.Config`, the macro `simp_arith` expands to `simp (config := { arith := true })`. Only `Nat` and linear arithmetic is currently supported. Example:

  ```
  example : 0 < 1 + x ∧ x + y + 2 ≥ y + 1 := by
    simp_arith
  ```

- Add `fail <string>?` tactic that always fail.
- Add support for acyclicity at dependent elimination. See issue #1022.
- Add `trace <string>` tactic for debugging purposes.
- Add nontrivial `SizeOf` instance for types `Unit → α`, and add support for them in the auto-generated `SizeOf` instances for user-defined inductive types. For example, given the inductive datatype

  ```
  inductive LazyList (α : Type u) where
    | nil                                : LazyList α
    | cons (hd : α) (tl : LazyList α)    : LazyList α
    | delayed (t : Thunk (LazyList α))   : LazyList α
  ```

  we now have `sizeOf (LazyList.delayed t) = 1 + sizeOf t` instead of `sizeOf (LazyList.delayed t) = 2`.

- Add support for guessing (very) simple well-founded relations when proving termination. For example, the following function does not require a `termination_by` annotation anymore.

  ```
  def Array.insertAtAux (i : Nat) (as : Array α) (j : Nat) : Array
  α :=
    if h : i < j then
      let as := as.swap! (j-1) j;
      insertAtAux i as (j-1)
    else
      as
  ```

- Add support for `for h : x in xs do ...` notation where `h : x ∈ xs`. This is mainly useful for showing termination.
- Auto implicit behavior changed for inductive families. An auto implicit argument occurring in inductive family index is also treated as an index (IF it is not fixed, see next item). For example

  ```
  inductive HasType : Index n → Vector Ty n → Ty → Type where
  ```

  is now interpreted as

  ```
  inductive HasType : {n : Nat} → Index n → Vector Ty n → Ty → Type
  where
  ```

- To make the previous feature more convenient to use, we promote a fixed prefix of inductive family indices to parameters. For example, the following declaration is now accepted by Lean

  ```
  inductive Lst : Type u → Type u
    | nil  : Lst α
    | cons : α → Lst α → Lst α
  ```

and α in `Lst α` is a parameter. The actual number of parameters can be inspected using the command `#print Lst`. This feature also makes sure we still accept the declaration

```
inductive Sublist : List α → List α → Prop
  | slnil : Sublist [] []
  | cons l₁ l₂ a : Sublist l₁ l₂ → Sublist l₁ (a :: l₂)
  | cons2 l₁ l₂ a : Sublist l₁ l₂ → Sublist (a :: l₁) (a :: l₂)
```

- Added auto implicit "chaining". Unassigned metavariables occurring in the auto implicit types now become new auto implicit locals. Consider the following example:

  ```
  inductive HasType : Fin n → Vector Ty n → Ty → Type where
    | stop : HasType 0 (ty :: ctx) ty
    | pop  : HasType k ctx ty → HasType k.succ (u :: ctx) ty
  ```

  `ctx` is an auto implicit local in the two constructors, and it has type `ctx : Vector Ty ?m`. Without auto implicit "chaining", the metavariable `?m` will remain unassigned. The new feature creates yet another implicit local `n : Nat` and assigns n to `?m`. So, the declaration above is shorthand for

  ```
  inductive HasType : {n : Nat} → Fin n → Vector Ty n → Ty → Type
  where
    | stop : {ty : Ty} → {n : Nat} → {ctx : Vector Ty n} → HasType
  0 (ty :: ctx) ty
    | pop  : {n : Nat} → {k : Fin n} → {ctx : Vector Ty n} → {ty :
  Ty} → HasType k ctx ty → HasType k.succ (u :: ctx) ty
  ```

- Eliminate auxiliary type annotations (e.g, `autoParam` and `optParam`) from recursor minor premises and projection declarations. Consider the following example

  ```
  structure A :=
    x : Nat
    h : x = 1 := by trivial

  example (a : A) : a.x = 1 := by
    have aux := a.h
    -- `aux` has now type `a.x = 1` instead of `autoParam (a.x = 1)
  auto✝`
    exact aux

  example (a : A) : a.x = 1 := by
    cases a with
    | mk x h =>
      -- `h` has now type `x = 1` instead of `autoParam (x = 1)
  auto✝`
      assumption
  ```

- We now accept overloaded notation in patterns, but we require the set of pattern variables in each alternative to be the same. Example:

  ```
  inductive Vector (α : Type u) : Nat → Type u
    | nil : Vector α 0
    | cons : α → Vector α n → Vector α (n+1)
  ```

```
infix:67 " :: " => Vector.cons -- Overloading the `::` notation

def head1 (x : List α) (h : x ≠ []) : α :=
  match x with
  | a :: as => a -- `::` is `List.cons` here

def head2 (x : Vector α (n+1)) : α :=
  match x with
  | a :: as => a -- `::` is `Vector.cons` here
```

- New notation `.<identifier>` based on Swift. The namespace is inferred from the expected type. See issue #944. Examples:

```
def f (x : Nat) : Except String Nat :=
  if x > 0 then
    .ok x
  else
    .error "x is zero"

namespace Lean.Elab
open Lsp

def identOf : Info → Option (RefIdent × Bool)
  | .ofTermInfo ti => match ti.expr with
    | .const n .. => some (.const n, ti.isBinder)
    | .fvar id .. => some (.fvar id, ti.isBinder)
    | _ => none
  | .ofFieldInfo fi => some (.const fi.projName, false)
  | _ => none

def isImplicit (bi : BinderInfo) : Bool :=
  bi matches .implicit

end Lean.Elab
```

Lean 4.0.0-m3 (2022-01-31)

This is the third milestone release of Lean 4, and the last planned milestone before an official release. With almost 3000 commits improving and extending many parts of the system since the last milestone, we are now close to completing all main features we have envisioned for Lean 4.

Contributors:

```
$ git shortlog -s -n v4.0.0-m2..v4.0.0-m3
  1719  Leonardo de Moura
   725  Sebastian Ullrich
   149  Wojciech Nawrocki
    93  Daniel Selsam
    82  Gabriel Ebner
    36  Joscha
```

```
35  Daniel Fabian
21  tydeu
14  Mario Carneiro
13  larsk21
12  Jannis Limperg
11  Chris Lovett
 8  Henrik Böving
 4  François G. Dorais
 4  Siddharth
 3  Joe Hendrix
 3  Scott Morrison
 3  ammkrn
 2  Josh Levine
 2  Mac
 2  Mac Malone
 2  Simon Hudon
 2  pcpthm
 1  Anders Christiansen Sørby
 1  Andrei Cheremskoy
 1  Arthur Paulino
 1  Christian Pehle
 1  Formally Verified Waffle Maker
 1  Hunter Monroe
 1  Jan Hrcek
 1  Joshua Seaton
 1  Kevin Buzzard
 1  Lorenz Leutgeb
 1  Mauricio Collares
 1  Michael Burge
 1  Paul Brinkmeier
 1  Reijo Jaakkola
 1  Severen Redwood
 1  Siddharth Bhat
 1  Tom Ball
 1  Varun Gandhi
 1  WojciechKarpiel
 1  Xavier Noria
 1  gabriel-doriath-dohler
 1  zygi
 1  Бакиновский Максим
```

Lean 4.0.0-m2 (2021-03-02)

This is the second milestone release of Lean 4. With too many improvements and bug fixes in almost all parts of the system to list, we would like to single out major improvements to `simp` and other built-in tactics as well as support for a goal view that make the proving experience more comfortable.

Lean 4.0.0-m1 (2021-01-04)

The Lean development team is proud to announce the first milestone release of Lean 4. This release is aimed at experimentation with the new features of Lean 4, eventually leading to a full release of 4.0.0 ready for general use.

This release is the result of almost three years of work since the release of Lean 3.4.0, reworking, extending, and improving almost all aspects of Lean. More information about Lean 4 can be found in the official documentation as well as in the introductory talk "An overview of Lean 4" at Lean Together 2021.

Leonardo de Moura & Sebastian Ullrich

Acknowledgements

- Daniel Selsam - type class resolution, feedback, design discussions
- Marc Huisinga and Wojciech Nawrocki - Lean Server
- Joe Hendrix, Andrew Kent, Rob Dockins, Simon Winwood (Galois Inc) - early adopters, suggestions, feedback
- Daan Leijen, Simon Peyton Jones, Nikhil Swamy, Sebastian Graf, Max Wagner - design discussions, feedback, suggestions

Index

Symbols | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Z

Symbols

(. . .)

. . . .

< ; > (0) (1)

{ . . . }

·

· . . .

Σ-types

ι-reduction

A

AR (environment variable)

Acc

Acc.intro
  Constructor of Acc

AccessRight
  IO.AccessRight

Add

Add.mk
  Instance constructor of Add

AddRightCancel
  Lean.Grind.AddRightCancel

Alternative

Alternative.mk
  Instance constructor of Alternative

And

And.elim

And.intro
  Constructor of And

Append

Append.mk
  Instance constructor of Append

Applicative

Applicative.mk
  Instance constructor of Applicative

Array

Array.all

Array.allDiff

Array.allM

Array.any

Array.anyM

Array.append

Array.appendList

Array.attach

Array.attachWith

Array.back

Array.back!

Array.back?

Array.binInsert

Array.binInsertM

Array.binSearch

Array.binSearchContains

Array.contains

Array.count

Array.countP

Array.drop

Array.elem

Array.empty

Array.emptyWithCapacity

Array.erase

Array.eraseIdx

Array.eraseIdx!

Array.eraseIdxIfInBounds

Array.eraseP

Array.eraseReps

Array.extract

Array.filter

Array.filterM

Array.filterMap

Array.filterMapM

Array.filterRevM

Array.filterSepElems

Array.filterSepElemsM

Array.finIdxOf?

Array.finRange

Array.find?

Array.findFinIdx?

Array.findIdx

Array.findIdx?

Array.findIdxM?

Array.findM?

Array.findRev?

Array.findRevM?

Array.findSome!

Array.findSome?

Array.findSomeM?

Array.findSomeRev?

Array.findSomeRevM?

Array.firstM

Array.flatMap

Array.flatMapM

Array.flatten

Array.foldl

Array.foldlM

Array.foldr

Array.foldrM

Array.forM

Array.forRevM

Array.getD

Array.getEvenElems

Array.getMax?

Array.groupByKey

Array.idxOf

Array.idxOf?

Array.insertIdx

- Array.insertIdx!
- Array.insertIdxIfInBounds
- Array.insertionSort
- Array.isEmpty
- Array.isEqv
- Array.isPrefixOf
- Array.leftpad
- Array.lex
- Array.map
- Array.mapFinIdx
- Array.mapFinIdxM
- Array.mapIdx
- Array.mapIdxM
- Array.mapM
- Array.mapM'
- Array.mapMono
- Array.mapMonoM
- Array.mk
  - Constructor of Array
- Array.modify
- Array.modifyM
- Array.modifyOp
- Array.ofFn
- Array.ofSubarray

Array.partition

Array.pmap

Array.pop

Array.popWhile

Array.push

Array.qsort

Array.qsortOrd

Array.range

Array.range'

Array.replace

Array.replicate

Array.reverse

Array.rightpad

Array.set

Array.set!

Array.setIfInBounds

Array.shrink

Array.singleton

Array.size

Array.sum

Array.swap

Array.swapAt

Array.swapAt!

Array.swapIfInBounds

Array.take

Array.takeWhile

Array.toList

Array.toListAppend

Array.toListRev

Array.toSubarray

Array.toVector

Array.uget

Array.unattach

Array.unzip

Array.uset

Array.usize

Array.zip

Array.zipIdx

Array.zipWith

Array.zipWithAll

AtomicT
  Std.AtomicT

abs
  BitVec.abs

abs
  Float.abs

abs
  Float32.abs

abs
ISize.abs

abs
Int16.abs

abs
Int32.abs

abs
Int64.abs

abs
Int8.abs

absurd

ac_nf

ac_nf0

ac_rfl

accessed
IO.FS.Metadata.accessed (structure field)

acos
Float.acos

acos
Float32.acos

acosh
Float.acosh

acosh
Float32.acosh

adapt
ExceptT.adapt

adapt

ReaderT.adapt

adaptExcept

EStateM.adaptExcept

adc

BitVec.adc

adcb

BitVec.adcb

add

Add.add (class method)

add

BitVec.add

add

Fin.add

add

Float.add

add

Float32.add

add

ISize.add

add

Int.add

add

Int16.add

add

Int32.add

add

Int64.add

add

Int8.add

add

Nat.add

add

UInt16.add

add

UInt32.add

add

UInt64.add

add

UInt8.add

add

USize.add

addAppParen

Repr.addAppParen

addCases

Fin.addCases

addExtension

System.FilePath.addExtension

addHeartbeats

IO.addHeartbeats

addMacroScope

Lean.Macro.addMacroScope

addNat

Fin.addNat

add_assoc

Lean.Grind.Semiring.nsmul (class method)

add_comm

Lean.Grind.Semiring.ofNat (class method)

add_le_left_iff

Lean.Grind.OrderedAdd.add_le_left_iff (class method)

add_nsmul

[anonymous] (class method)

add_right_cancel

Lean.Grind.AddRightCancel.add_right_cancel (class method)

add_zero

Lean.Grind.Semiring.natCast (class method)

add_zsmul

[anonymous] (class method)

admit

all

Array.all

all

List.all

all

Nat.all

all

Option.all

all

Std.HashSet.all

all

Std.TreeMap.all

all

Std.TreeSet.all

all

String.all

all

Subarray.all

all

Substring.all

allDiff

Array.allDiff

allEq

Subsingleton.allEq (class method)

allI

Prod.allI

allM

Array.allM

allM

List.allM

allM

Nat.allM

allM

Subarray.allM

allOnes

BitVec.allOnes

allTR

Nat.allTR

all_goals (0) (1)

allowUnsafeReducibility

alter

Std.DHashMap.alter

alter

Std.DTreeMap.alter

alter
Std.ExtDHashMap.alter

alter
Std.ExtHashMap.alter

alter
Std.HashMap.alter

alter
Std.TreeMap.alter

and
BitVec.and

and
Bool.and

and
List.and

andM

and_intros

any
Array.any

any
List.any

any
Nat.any

any
Option.any

any
Std.HashSet.any

any
Std.TreeMap.any

any
Std.TreeSet.any

any
String.any

any
Subarray.any

any
Substring.any

anyI
Prod.anyI

anyM
Array.anyM

anyM
List.anyM

anyM
Nat.anyM

anyM
Subarray.anyM

anyTR
Nat.anyTR

any_goals (0) (1)

appDir
IO.appDir

appPath
IO.appPath

append
Append.append (class method)

append

Array.append

append

BitVec.append

append

List.append

append

String.append

appendList

Array.appendList

appendTR

List.appendTR

apply (0) (1)

apply?

apply_assumption

apply_ext_theorem

apply_mod_cast

apply_rfl

apply_rules

arg_[@]i

args

args

[anonymous] (structure field)

arith

Lean.Meta.Simp.Config.arith (structure field)

asString

List.asString

asTask

BaseIO.asTask

asTask

EIO.asTask

asTask

IO.asTask

as_aux_lemma

asin

Float.asin

asin

Float32.asin

asinh

Float.asinh

asinh

Float32.asinh

assumption

assumption

inaccessible

assumption_mod_cast

atEnd

String.Iterator.atEnd

atEnd

String.atEnd

atEnd

Substring.atEnd

atIdx!

Std.TreeSet.atIdx!

atIdx

Std.TreeSet.atIdx

atIdx?

Std.TreeSet.atIdx?

atIdxD

Std.TreeSet.atIdxD

atan

Float.atan

atan

Float32.atan

atan2

Float.atan2

atan2

Float32.atan2

atanh

Float.atanh

atanh

Float32.atanh

atomically

Std.Mutex.atomically

atomicallyOnce

Std.Mutex.atomicallyOnce

attach

Array.attach

attach

List.attach

attach

Option.attach

attachWith

Array.attachWith

attachWith

List.attachWith

attachWith

Option.attachWith

autoImplicit

autoLift

autoParam

autoPromoteIndices

inductive.autoPromoteIndices

autoUnfold

Lean.Meta.DSimp.Config.autoUnfold (structure field)

autoUnfold

Lean.Meta.Simp.Config.autoUnfold (structure field)

B

BEq

BEq.mk

Instance constructor of BEq

Backend

Lake.Backend

Backtrackable

EStateM.Backtrackable

BaseIO

BaseIO.asTask

BaseIO.bindTask

BaseIO.chainTask

BaseIO.mapTask

BaseIO.mapTasks

BaseIO.toEIO

BaseIO.toIO

Bind

Bind.bindLeft

Bind.kleisliLeft

Bind.kleisliRight

Bind.mk

Instance constructor of Bind

BitVec

BitVec.abs

BitVec.adc

BitVec.adcb

BitVec.add

BitVec.allOnes

BitVec.and

BitVec.append

BitVec.carry

BitVec.cast

BitVec.concat

BitVec.cons

BitVec.decEq

BitVec.divRec

BitVec.divSubtractShift

BitVec.extractLsb

BitVec.extractLsb'

BitVec.fill

BitVec.getLsb

BitVec.getLsb?

BitVec.getLsbD

BitVec.getMsb

BitVec.getMsb?

BitVec.getMsbD

BitVec.hash

BitVec.intMax

BitVec.intMin

BitVec.iunfoldr

BitVec.iunfoldr_replace

BitVec.msb

BitVec.mul

BitVec.mulRec

BitVec.neg

BitVec.nil

BitVec.not

BitVec.ofBool

BitVec.ofBoolListBE

BitVec.ofBoolListLE

BitVec.ofFin

Constructor of BitVec

BitVec.ofInt

BitVec.ofNat

BitVec.ofNatLT

BitVec.or

BitVec.replicate

BitVec.reverse

BitVec.rotateLeft

BitVec.rotateRight

BitVec.saddOverflow

BitVec.sdiv

BitVec.setWidth

BitVec.setWidth'

BitVec.shiftConcat

BitVec.shiftLeft

BitVec.shiftLeftRec

BitVec.shiftLeftZeroExtend

BitVec.signExtend

BitVec.sle

BitVec.slt

BitVec.smod

BitVec.smtSDiv

BitVec.smtUDiv

BitVec.srem

BitVec.sshiftRight

BitVec.sshiftRight'

BitVec.sshiftRightRec

BitVec.ssubOverflow

BitVec.sub

BitVec.toHex

BitVec.toInt

BitVec.toNat

BitVec.truncate

BitVec.twoPow

BitVec.uaddOverflow

BitVec.udiv

BitVec.ule

BitVec.ult

BitVec.umod

BitVec.ushiftRight

BitVec.ushiftRightRec

BitVec.usubOverflow

BitVec.xor

BitVec.zero

BitVec.zeroExtend

Bool

Bool.and

Bool.dcond

Bool.decEq

Bool.false

Constructor of Bool

Bool.not

Bool.or

Bool.toISize

Bool.toInt

Bool.toInt16

Bool.toInt32

Bool.toInt64

Bool.toInt8

Bool.toNat

Bool.toUInt16

Bool.toUInt32

Bool.toUInt64

Bool.toUInt8

Bool.toUSize

Bool.true
  Constructor of Bool

Bool.xor

Buffer
  IO.FS.Stream.Buffer

BuildType
  Lake.BuildType

back!
  Array.back!

back
  Array.back

back
  String.back

back?
  Array.back?

backward.synthInstance.canonInstances

bdiv
  Int.bdiv

beq
  BEq.beq (class method)

beq

Float.beq

beq

Float32.beq

beq

List.beq

beq

Nat.beq

beq

Substring.beq

beta

Lean.Meta.DSimp.Config.beta (structure field)

beta

Lean.Meta.Simp.Config.beta (structure field)

binInsert

Array.binInsert

binInsertM

Array.binInsertM

binSearch

Array.binSearch

binSearchContains

Array.binSearchContains

bind

Bind.bind (class method)

bind

EStateM.bind

bind

Except.bind

bind

ExceptT.bind

bind

Option.bind

bind

OptionT.bind

bind

ReaderT.bind

bind

StateT.bind

bind

Task.bind

bind

Thunk.bind

bindCont

ExceptT.bindCont

bindLeft

Bind.bindLeft

bindM

Option.bindM

bindTask

BaseIO.bindTask

bindTask

EIO.bindTask

bindTask

IO.bindTask

bind_assoc

[anonymous] (class method)

bind_map

[anonymous] (class method)

bind_pure_comp

[anonymous] (class method)

binderNameHint

bitwise

Nat.bitwise

ble

Nat.ble

blt

Nat.blt

bmod

Int.bmod

bootstrap.inductiveCheckResultingUniverse

bracket

Std.Format.bracket

bracketFill

Std.Format.bracketFill

bsize

Substring.bsize

buckets

Std.DHashMap.Raw.buckets (structure field)

build (Lake command)

bv_check

bv_decide

bv_decide?

bv_normalize

bv_omega

byCases
  Decidable.byCases

by_cases

by_cases'
  Or.by_cases'

by_cases
  Or.by_cases

byteIdx
  String.Pos.byteIdx (structure field)

byteSize
  IO.FS.Metadata.byteSize (structure field)

C

CC (environment variable)

CCPO
  Lean.Order.CCPO

Channel
  Std.Channel

Char

Char.isAlpha

Char.isAlphanum

Char.isDigit

Char.isLower

Char.isUpper

Char.isValidCharNat

Char.isWhitespace

Char.le

Char.lt

Char.mk
  Constructor of Char

Char.ofNat

Char.ofUInt8

Char.quote

Char.toLower

Char.toNat

Char.toString

Char.toUInt8

Char.toUpper

Char.utf16Size

Char.utf8Size

CharLit

Lean.Syntax.CharLit

Child

IO.Process.Child

CloseableChannel

Std.CloseableChannel

Coe

Coe.mk

Instance constructor of Coe

CoeDep

CoeDep.mk

Instance constructor of CoeDep

CoeFun

CoeFun.mk

Instance constructor of CoeFun

CoeHTC

CoeHTC.mk

Instance constructor of CoeHTC

CoeHTCT

CoeHTCT.mk

Instance constructor of CoeHTCT

CoeHead

CoeHead.mk

Instance constructor of CoeHead

CoeOTC

CoeOTC.mk

Instance constructor of CoeOTC

CoeOut

CoeOut.mk

Instance constructor of CoeOut

CoeSort

CoeSort.mk

Instance constructor of CoeSort

CoeT

CoeT.mk

Instance constructor of CoeT

CoeTC

CoeTC.mk

Instance constructor of CoeTC

CoeTail

CoeTail.mk

Instance constructor of CoeTail

CommRing

Lean.Grind.CommRing

CommSemiring

Lean.Grind.CommSemiring

Command

Lean.Syntax.Command

Condvar

Std.Condvar

Config

Lean.Meta.DSimp.Config

Config

Lean.Meta.Rewrite.Config

Config

Lean.Meta.Simp.Config

calc

call-by-need

cancel

IO.cancel

canonInstances

backward.synthInstance.canonInstances

capitalize

String.capitalize

carry

BitVec.carry

case

case ... => ...

case'

case' ... => ...

caseStrongRecOn

Nat.caseStrongRecOn

cases

cases

Fin.cases

casesAuxOn

Nat.casesAuxOn

cast

cast
BitVec.cast

cast
Fin.cast

cast
Int.cast

cast
Nat.cast

castAdd
Fin.castAdd

castLE
Fin.castLE

castLT
Fin.castLT

castSucc
Fin.castSucc

cast_heq

catchExceptions
EIO.catchExceptions

catchRuntime
Lean.Meta.Simp.Config.catchRuntime (structure field)

cbrt
Float.cbrt

cbrt
Float32.cbrt

ceil
Float.ceil

ceil
Float32.ceil

chain
of coercions

chainTask
BaseIO.chainTask

chainTask
EIO.chainTask

chainTask
IO.chainTask

change (0) (1)

change ... with ...

charLitKind
Lean.charLitKind

check-build (Lake command)

check-lint (Lake command)

check-test (Lake command)

checkCanceled
IO.checkCanceled

choice
Option.choice

choiceKind
Lean.choiceKind

choose
Exists.choose

classical

clean (Lake command)

clear

cmd

[anonymous] (structure field)

coe

Coe.coe (class method)

coe

CoeDep.coe (class method)

coe

CoeFun.coe (class method)

coe

CoeHTC.coe (class method)

coe

CoeHTCT.coe (class method)

coe

CoeHead.coe (class method)

coe

CoeOTC.coe (class method)

coe

CoeOut.coe (class method)

coe

CoeSort.coe (class method)

coe

CoeT.coe (class method)

coe

CoeTC.coe (class method)

coe

CoeTail.coe (class method)

colEq

colGe

colGt

comment
block

comment
line

commonPrefix
Substring.commonPrefix

commonSuffix
Substring.commonSuffix

comp
Function.comp

comp_map
LawfulFunctor.comp_map (class method)

compare
Ord.compare (class method)

compareLex

compareOfLessAndBEq

compareOfLessAndEq

compareOn

complement
ISize.complement

complement
Int16.complement

complement
Int32.complement

complement
Int64.complement

complement
Int8.complement

complement
UInt16.complement

complement
UInt32.complement

complement
UInt64.complement

complement
UInt8.complement

complement
USize.complement

completions (Elan command)

components
System.FilePath.components

concat
BitVec.concat

concat
List.concat

cond

configuration
of tactics

congr (0) (1) (2)

congrArg

congrFun

cons

BitVec.cons

const

Function.const

constructor (0) (1)

contains

Array.contains

contains

List.contains

contains

Std.DHashMap.contains

contains

Std.DTreeMap.contains

contains

Std.ExtDHashMap.contains

contains

Std.ExtHashMap.contains

contains

Std.ExtHashSet.contains

contains

Std.HashMap.contains

contains

Std.HashSet.contains

contains

Std.TreeMap.contains

contains

Std.TreeSet.contains

contains

String.contains

contains

Substring.contains

containsThenInsert

Std.DHashMap.containsThenInsert

containsThenInsert

Std.DTreeMap.containsThenInsert

containsThenInsert

Std.ExtDHashMap.containsThenInsert

containsThenInsert

Std.ExtHashMap.containsThenInsert

containsThenInsert

Std.ExtHashSet.containsThenInsert

containsThenInsert

Std.HashMap.containsThenInsert

containsThenInsert

Std.HashSet.containsThenInsert

containsThenInsert

Std.TreeMap.containsThenInsert

containsThenInsert

Std.TreeSet.containsThenInsert

containsThenInsertIfNew

Std.DHashMap.containsThenInsertIfNew

containsThenInsertIfNew

Std.DTreeMap.containsThenInsertIfNew

containsThenInsertIfNew

Std.ExtDHashMap.containsThenInsertIfNew

containsThenInsertIfNew
  Std.ExtHashMap.containsThenInsertIfNew

containsThenInsertIfNew
  Std.HashMap.containsThenInsertIfNew

containsThenInsertIfNew
  Std.TreeMap.containsThenInsertIfNew

contextual
  Lean.Meta.Simp.Config.contextual (structure field)

contradiction

control

controlAt

conv

conv => ...

conv' (0) (1)

cos
  Float.cos

cos
  Float32.cos

cosh
  Float.cosh

cosh
  Float32.cosh

count
  Array.count

count
  List.count

countP

Array.countP

countP

List.countP

createDir

IO.FS.createDir

createDirAll

IO.FS.createDirAll

createTempDir

IO.FS.createTempDir

createTempFile

IO.FS.createTempFile

crlfToLf

String.crlfToLf

csup

[anonymous] (class method)

csup_spec

[anonymous] (class method)

cumulativity

curr

String.Iterator.curr

currColumn

Std.Format.MonadPrettyFormat.currColumn (class method)

currentDir

IO.currentDir

curry

Function.curry

customEliminators

tactic.customEliminators

cwd

[anonymous] (structure field)

D

DHashMap

Std.DHashMap

DTreeMap

Std.DTreeMap

Decidable

Decidable.byCases

Decidable.decide

Decidable.isFalse

Constructor of Decidable

Decidable.isTrue

Constructor of Decidable

DecidableEq

DecidableLE

DecidableLT

DecidablePred

DecidableRel

DirEntry

IO.FS.DirEntry

Div

Div.mk

Instance constructor of Div

Dvd

Dvd.mk

Instance constructor of Dvd

data

IO.FS.Stream.Buffer.data (structure field)

data

String.data (structure field)

dbgTraceIfShared

dbg_trace

dcond

Bool.dcond

decEq

BitVec.decEq

decEq

Bool.decEq

decEq

ISize.decEq

decEq

Int.decEq

decEq

Int16.decEq

decEq

Int32.decEq

decEq

Int64.decEq

decEq

Int8.decEq

decEq

Nat.decEq

decEq

String.decEq

decEq

UInt16.decEq

decEq

UInt32.decEq

decEq

UInt64.decEq

decEq

UInt8.decEq

decEq

USize.decEq

decLe

Float.decLe

decLe

Float32.decLe

decLe

ISize.decLe

decLe

Int16.decLe

decLe

Int32.decLe

decLe

Int64.decLe

decLe

Int8.decLe

decLe

Nat.decLe

decLe

UInt16.decLe

decLe

UInt32.decLe

decLe

UInt64.decLe

decLe

UInt8.decLe

decLe

USize.decLe

decLt

Float.decLt

decLt

Float32.decLt

decLt

ISize.decLt

decLt

Int16.decLt

decLt

Int32.decLt

decLt

Int64.decLt

decLt

Int8.decLt

decLt

Nat.decLt

decLt

UInt16.decLt

decLt

UInt32.decLt

decLt

UInt64.decLt

decLt

UInt8.decLt

decLt

USize.decLt

decapitalize

String.decapitalize

decidable

decidableEqNone

Option.decidableEqNone

decide

decide

Decidable.decide

decide

Lean.Meta.DSimp.Config.decide (structure field)

decide

Lean.Meta.Simp.Config.decide (structure field)

decreasing_tactic

decreasing_trivial

decreasing_with

dedicated

Task.Priority.dedicated

deepTerms

pp.deepTerms

defIndent

Std.Format.defIndent

defWidth

Std.Format.defWidth

default (Elan command)

default

Inhabited.default (class method)

default

Task.Priority.default

defaultFacets

[anonymous] (structure field)

delta (0) (1)

diff

guard_msgs.diff

digitChar

Nat.digitChar

discard

Functor.discard

discharge

trace.Meta.Tactic.simp.discharge

div

Div.div (class method)

div

Fin.div

div

Float.div

div

Float32.div

div

ISize.div

div

Int16.div

div

Int32.div

div

Int64.div

div

Int8.div

div

Nat.div

div

UInt16.div

div

UInt32.div

div

UInt64.div

div

UInt8.div

div

USize.div

div2Induction

Nat.div2Induction

divRec

BitVec.divRec

divSubtractShift

BitVec.divSubtractShift

div_eq_mul_inv

[anonymous] (class method)

done (0) (1)

down

PLift.down (structure field)

down

ULift.down (structure field)

drop

Array.drop

drop

List.drop

drop

String.drop

drop

Subarray.drop

drop

Substring.drop

dropLast

List.dropLast

dropLastTR

List.dropLastTR

dropPrefix?

String.dropPrefix?

dropPrefix?

Substring.dropPrefix?

dropRight

String.dropRight

dropRight

Substring.dropRight

dropRightWhile

String.dropRightWhile

dropRightWhile

Substring.dropRightWhile

dropSuffix?

String.dropSuffix?

dropSuffix?

Substring.dropSuffix?

dropWhile

List.dropWhile

dropWhile

String.dropWhile

dropWhile

Substring.dropWhile

dsimp (0) (1)

dsimp!

dsimp

Lean.Meta.Simp.Config.dsimp (structure field)

dsimp?

dsimp?!

dvd

Dvd.dvd (class method)

E

EIO

EIO.asTask

EIO.bindTask

EIO.catchExceptions

EIO.chainTask

EIO.mapTask

EIO.mapTasks

EIO.toBaseIO

EIO.toIO

EIO.toIO'

ELAN (environment variable)

ELAN_HOME (environment variable) (0) (1)

EST

EStateM

EStateM.Backtrackable

EStateM.Backtrackable.mk
  Instance constructor of EStateM.Backtrackable

EStateM.Result

EStateM.Result.error
  Constructor of EStateM.Result

EStateM.Result.ok
  Constructor of EStateM.Result

EStateM.adaptExcept

EStateM.bind

EStateM.fromStateM

EStateM.get

EStateM.map

EStateM.modifyGet

EStateM.nonBacktrackable

EStateM.orElse

EStateM.orElse'

EStateM.pure

EStateM.run

EStateM.run'

EStateM.seqRight

EStateM.set

EStateM.throw

EStateM.tryCatch

Empty

Empty.elim

Eq

Eq.mp

Eq.mpr

Eq.refl

Constructor of Eq

Eq.subst

Eq.symm

Eq.trans

Equiv

HasEquiv.Equiv (class method)

Equiv

Std.DHashMap.Equiv

Equiv

Std.HashMap.Equiv

Equiv

Std.HashSet.Equiv

EquivBEq

EquivBEq.mk

Instance constructor of EquivBEq

Equivalence

Equivalence.mk

Constructor of Equivalence

Error

IO.Error

Even

Even.plusTwo

Constructor of Even

Even.zero

Constructor of Even

Except

Except.bind

Except.error

Constructor of Except

Except.isOk

Except.map

Except.mapError

Except.ok

Constructor of Except

Except.orElseLazy

Except.pure

Except.toBool

Except.toOption

Except.tryCatch

ExceptCpsT

ExceptCpsT.lift

ExceptCpsT.run

ExceptCpsT.runCatch

ExceptCpsT.runK

ExceptT

ExceptT.adapt

ExceptT.bind

ExceptT.bindCont

ExceptT.lift

ExceptT.map

ExceptT.mk

ExceptT.pure

ExceptT.run

ExceptT.tryCatch

Exists

Exists.choose

Exists.intro
: Constructor of Exists

ExtDHashMap
: Std.ExtDHashMap

ExtHashMap
: Std.ExtHashMap

ExtHashSet
: Std.ExtHashSet

ediv
: Int.ediv

elem
: Array.elem

elem
: List.elem

elemsAndSeps
: Lean.Syntax.TSepArray.elemsAndSeps (structure field)

elim
: And.elim

elim
: Empty.elim

elim
: False.elim

elim
: HEq.elim

elim

Iff.elim

elim

Not.elim

elim

Option.elim

elim

PEmpty.elim

elim

Subsingleton.elim

elim

Sum.elim

elim0

Fin.elim0

elimM

Option.elimM

emod

Int.emod

empty

Array.empty

empty

Std.DTreeMap.empty

empty

Std.TreeMap.empty

empty

Std.TreeSet.empty

empty

Subarray.empty

emptyWithCapacity

Array.emptyWithCapacity

emptyWithCapacity

Std.DHashMap.emptyWithCapacity

emptyWithCapacity

Std.ExtDHashMap.emptyWithCapacity

emptyWithCapacity

Std.ExtHashMap.emptyWithCapacity

emptyWithCapacity

Std.ExtHashSet.emptyWithCapacity

emptyWithCapacity

Std.HashMap.emptyWithCapacity

emptyWithCapacity

Std.HashSet.emptyWithCapacity

endPos

String.endPos

endTags

Std.Format.MonadPrettyFormat.endTags (class method)

endsWith

String.endsWith

enter

entryAtIdx!

Std.TreeMap.entryAtIdx!

entryAtIdx

Std.TreeMap.entryAtIdx

entryAtIdx?

Std.TreeMap.entryAtIdx?

entryAtIdxD

Std.TreeMap.entryAtIdxD

env (Lake command)

env
  IO.Process.SpawnArgs.cmd (structure field)

environment variables

eprint
  IO.eprint

eprintln
  IO.eprintln

eqRec_heq

eq_of_beq
  [anonymous] (class method)

eq_of_heq

eq_refl

erase
  Array.erase

erase
  List.erase

erase
  Std.DHashMap.erase

erase
  Std.DTreeMap.erase

erase
  Std.ExtDHashMap.erase

erase
  Std.ExtHashMap.erase

erase
  Std.ExtHashSet.erase

erase

Std.HashMap.erase

erase

Std.HashSet.erase

erase

Std.TreeMap.erase

erase

Std.TreeSet.erase

eraseDups

List.eraseDups

eraseIdx!

Array.eraseIdx!

eraseIdx

Array.eraseIdx

eraseIdx

List.eraseIdx

eraseIdxIfInBounds

Array.eraseIdxIfInBounds

eraseIdxTR

List.eraseIdxTR

eraseMany

Std.TreeMap.eraseMany

eraseMany

Std.TreeSet.eraseMany

eraseP

Array.eraseP

eraseP

List.eraseP

erasePTR

List.erasePTR

eraseReps

Array.eraseReps

eraseReps

List.eraseReps

eraseTR

List.eraseTR

erw (0) (1)

eta

Lean.Meta.DSimp.Config.eta (structure field)

eta

Lean.Meta.Simp.Config.eta (structure field)

etaStruct

Lean.Meta.DSimp.Config.etaStruct (structure field)

etaStruct

Lean.Meta.Simp.Config.etaStruct (structure field)

eval.derive.repr

eval.pp

eval.type

exact

exact

Quotient.exact

exact?

exact_mod_cast

exe (Lake command)

exeExtension

System.FilePath.exeExtension

exeName

[anonymous] (structure field)

execution

IO.AccessRight.execution (structure field)

exfalso

exists

exit

IO.Process.exit

exitCode

IO.Process.Output.exitCode (structure field)

exp

Float.exp

exp

Float32.exp

exp2

Float.exp2

exp2

Float32.exp2

expandMacro?

Lean.Macro.expandMacro?

expose_names

ext (0) (1)

ext1

extSeparator

System.FilePath.extSeparator

extension

System.FilePath.extension

extensionality

of propositions

extraDepTargets

[anonymous] (structure field) (0) (1)

extract

Array.extract

extract

List.extract

extract

String.Iterator.extract

extract

String.extract

extract

Substring.extract

extractLsb'

BitVec.extractLsb'

extractLsb

BitVec.extractLsb

F

False

False.elim

Field
  Lean.Grind.Field

FilePath
  System.FilePath

FileRight
  IO.FileRight

Fin

Fin.add

Fin.addCases

Fin.addNat

Fin.cases

Fin.cast

Fin.castAdd

Fin.castLE

Fin.castLT

Fin.castSucc

Fin.div

Fin.elim0

Fin.foldl

Fin.foldlM

Fin.foldr

Fin.foldrM

Fin.hIterate

Fin.hIterateFrom

Fin.induction

Fin.inductionOn

Fin.land

Fin.last

Fin.lastCases

Fin.log2

Fin.lor

Fin.mk

Constructor of Fin

Fin.mod

Fin.modn

Fin.mul

Fin.natAdd

Fin.ofNat

Fin.pred

Fin.rev

Fin.reverseInduction

Fin.shiftLeft

Fin.shiftRight

Fin.sub

Fin.subNat

Fin.succ

Fin.succRec

Fin.succRecOn

Fin.toNat

Fin.xor

FlattenBehavior

Std.Format.FlattenBehavior

Float

Float.abs

Float.acos

Float.acosh

Float.add

Float.asin

Float.asinh

Float.atan

Float.atan2

Float.atanh

Float.beq

Float.cbrt

Float.ceil

Float.cos

Float.cosh

Float.decLe

Float.decLt

Float.div

Float.exp

Float.exp2

Float.floor

Float.frExp

Float.isFinite

Float.isInf

Float.isNaN

Float.le

Float.log

Float.log10

Float.log2

Float.lt

Float.mul

Float.neg

Float.ofBinaryScientific

Float.ofBits

Float.ofInt

Float.ofNat

Float.ofScientific

Float.pow

Float.round

Float.scaleB

Float.sin

Float.sinh

Float.sqrt

Float.sub

Float.tan

Float.tanh

Float.toBits

Float.toFloat32

Float.toISize

Float.toInt16

Float.toInt32

Float.toInt64

Float.toInt8

Float.toString

Float.toUInt16

Float.toUInt32

Float.toUInt64

Float.toUInt8

Float.toUSize

Float32

Float32.abs

Float32.acos

Float32.acosh

Float32.add

Float32.asin

Float32.asinh

Float32.atan

Float32.atan2

Float32.atanh

Float32.beq

Float32.cbrt

Float32.ceil

Float32.cos

Float32.cosh

Float32.decLe

Float32.decLt

Float32.div

Float32.exp

Float32.exp2

Float32.floor

Float32.frExp

Float32.isFinite

Float32.isInf

Float32.isNaN

Float32.le

Float32.log

Float32.log10

Float32.log2

Float32.lt

Float32.mul

Float32.neg

Float32.ofBinaryScientific

Float32.ofBits

Float32.ofInt

Float32.ofNat

Float32.ofScientific

Float32.pow

Float32.round

Float32.scaleB

Float32.sin

Float32.sinh

Float32.sqrt

Float32.sub

Float32.tan

Float32.tanh

Float32.toBits

Float32.toFloat

Float32.toISize

Float32.toInt16

Float32.toInt32

Float32.toInt64

Float32.toInt8

Float32.toString

Float32.toUInt16

Float32.toUInt32

Float32.toUInt64

Float32.toUInt8

Float32.toUSize

ForIn

ForIn'

ForIn'.mk

Instance constructor of ForIn'

ForIn.mk

Instance constructor of ForIn

ForInStep

ForInStep.done

Constructor of ForInStep

ForInStep.value

ForInStep.yield

Constructor of ForInStep

ForM

ForM.forIn

ForM.mk

Instance constructor of ForM

Format

Std.Format

Function.comp

Function.const

Function.curry

Function.uncurry

Functor

Functor.discard

Functor.mapRev

Functor.mk

Instance constructor of Functor

fail

fail

OptionT.fail

failIfUnchanged

Lean.Meta.DSimp.Config.failIfUnchanged (structure field)

failIfUnchanged

Lean.Meta.Simp.Config.failIfUnchanged (structure field)

fail_if_success (0) (1)

failure

ReaderT.failure

failure

StateT.failure

failure

[anonymous] (class method)

false_or_by_contra

fdiv

Int.fdiv

fieldIdxKind

Lean.fieldIdxKind

fieldNotation

pp.fieldNotation

fileName

IO.FS.DirEntry.fileName (structure field)

fileName

System.FilePath.fileName

fileStem

System.FilePath.fileStem

fill

BitVec.fill

fill

Std.Format.fill

filter

Array.filter

filter

List.filter

filter

Option.filter

filter

Std.DHashMap.filter

filter

Std.DTreeMap.filter

filter

Std.ExtDHashMap.filter

filter

Std.ExtHashMap.filter

filter

Std.ExtHashSet.filter

filter

Std.HashMap.filter

filter

Std.HashSet.filter

filter

Std.TreeMap.filter

filter

Std.TreeSet.filter

filterM

Array.filterM

filterM

List.filterM

filterMap

Array.filterMap

filterMap

List.filterMap

filterMap

Std.DHashMap.filterMap

filterMap

Std.DTreeMap.filterMap

filterMap

Std.ExtDHashMap.filterMap

filterMap

Std.ExtHashMap.filterMap

filterMap

Std.HashMap.filterMap

filterMap

Std.TreeMap.filterMap

filterMapM

Array.filterMapM

filterMapM

List.filterMapM

filterMapTR

List.filterMapTR

filterRevM

Array.filterRevM

filterRevM

List.filterRevM

filterSepElems

Array.filterSepElems

filterSepElemsM

Array.filterSepElemsM

filterTR

List.filterTR

finIdxOf?

Array.finIdxOf?

finIdxOf?

List.finIdxOf?

finRange

Array.finRange

finRange

List.finRange

find

String.find

find?

Array.find?

find?

List.find?

findExternLib?

Lake.findExternLib?

findFinIdx?

Array.findFinIdx?

findFinIdx?

List.findFinIdx?

findIdx

Array.findIdx

findIdx

List.findIdx

findIdx?

Array.findIdx?

findIdx?

List.findIdx?

findIdxM?

Array.findIdxM?

findLeanExe?

Lake.findLeanExe?

findLeanLib?

Lake.findLeanLib?

findLineStart

String.findLineStart

findM?

Array.findM?

findM?

List.findM?

findModule?

Lake.findModule?

findPackage?

Lake.findPackage?

findRev?

Array.findRev?

findRev?

Subarray.findRev?

findRevM?

Array.findRevM?

findRevM?

Subarray.findRevM?

findSome!

Array.findSome!

findSome?

Array.findSome?

findSome?

List.findSome?

findSomeM?

Array.findSomeM?

findSomeM?

List.findSomeM?

findSomeRev?

Array.findSomeRev?

findSomeRevM?

Array.findSomeRevM?

findSomeRevM?

Subarray.findSomeRevM?

first (0) (1)

firstDiffPos

String.firstDiffPos

firstM

Array.firstM

firstM

List.firstM

fix

Lean.Order.fix

fix

WellFounded.fix

fix_eq

Lean.Order.fix_eq

flags

IO.AccessRight.flags

flags

IO.FileRight.flags

flatMap

Array.flatMap

flatMap

List.flatMap

flatMapM

Array.flatMapM

flatMapM

List.flatMapM

flatMapTR

List.flatMapTR

flatten

Array.flatten

flatten

List.flatten

flattenTR

List.flattenTR

floor

Float.floor

floor

Float32.floor

flush

IO.FS.Handle.flush

flush

IO.FS.Stream.flush (structure field)

fmod

Int.fmod

fn

Thunk.fn (structure field)

focus (0) (1)

fold

Nat.fold

fold

Std.DHashMap.fold

fold

Std.HashMap.fold

fold

Std.HashSet.fold

foldI

Prod.foldI

foldM

Nat.foldM

foldM

Std.DHashMap.foldM

foldM

Std.HashMap.foldM

foldM

Std.HashSet.foldM

foldRev

Nat.foldRev

foldRevM

Nat.foldRevM

foldTR

Nat.foldTR

foldl

Array.foldl

foldl

Fin.foldl

foldl

List.foldl

foldl

Std.DTreeMap.foldl

foldl

Std.TreeMap.foldl

foldl

Std.TreeSet.foldl

foldl

String.foldl

foldl

Subarray.foldl

foldl

Substring.foldl

foldlM

Array.foldlM

foldlM

Fin.foldlM

foldlM

List.foldlM

foldlM

Std.DTreeMap.foldlM

foldlM

Std.TreeMap.foldlM

foldlM

Std.TreeSet.foldlM

foldlM

Subarray.foldlM

foldlRecOn

List.foldlRecOn

foldr

Array.foldr

foldr

Fin.foldr

foldr

List.foldr

foldr

Std.TreeMap.foldr

foldr

Std.TreeSet.foldr

foldr

String.foldr

foldr

Subarray.foldr

foldr

Substring.foldr

foldrM

Array.foldrM

foldrM

Fin.foldrM

foldrM

List.foldrM

foldrM

Std.TreeMap.foldrM

foldrM

Std.TreeSet.foldrM

foldrM

Subarray.foldrM

foldrRecOn

List.foldrRecOn

foldrTR

List.foldrTR

forA

List.forA

forAsync

Std.Channel.forAsync

forIn'

ForIn'.forIn' (class method)

forIn

ForIn.forIn (class method)

forIn

ForM.forIn

forIn

Std.DHashMap.forIn

forIn

Std.DTreeMap.forIn

forIn

Std.HashMap.forIn

forIn

Std.HashSet.forIn

forIn

Std.TreeMap.forIn

forIn

Std.TreeSet.forIn

forIn

Subarray.forIn

forM

Array.forM

forM

ForM.forM (class method)

forM

List.forM

forM

Nat.forM

forM

Option.forM

forM

Std.DHashMap.forM

forM

Std.DTreeMap.forM

forM

Std.HashMap.forM

forM

Std.HashSet.forM

forM

Std.TreeMap.forM

forM

Std.TreeSet.forM

forM

Subarray.forM

forRevM

Array.forRevM

forRevM

Nat.forRevM

forRevM

Subarray.forRevM

format

Option.format

format

Std.ToFormat.format (class method)

forward

String.Iterator.forward

frExp

Float.frExp

frExp

Float32.frExp

fromStateM

EStateM.fromStateM

fromUTF8!

String.fromUTF8!

fromUTF8

String.fromUTF8

fromUTF8?

String.fromUTF8?

front

String.front

front

Substring.front

fst

MProd.fst (structure field)

fst

PProd.fst (structure field)

fst

PSigma.fst (structure field)

fst

Prod.fst (structure field)

fst

Sigma.fst (structure field)

fun

fun_cases

fun_induction

funext (0) (1)

G

GetElem

GetElem.mk

Instance constructor of GetElem

GetElem?

GetElem?.mk

Instance constructor of GetElem?

Glob

Lake.Glob

gcd

Int.gcd

gcd

Nat.gcd

generalize

get!

Option.get!

get!

Std.DHashMap.get!

get!

Std.DTreeMap.get!

get!

Std.ExtDHashMap.get!

get!

Std.ExtHashMap.get!

get!

Std.ExtHashSet.get!

get!

Std.HashMap.get!

get!

Std.HashSet.get!

get!

Std.TreeMap.get!

get!

Std.TreeSet.get!

get!

String.get!

get!

Subarray.get!

get'

String.get'

get

EStateM.get

get

List.get

get

MonadState.get

get

MonadState.get (class method)

get

MonadStateOf.get (class method)

get

Option.get

get

ST.Ref.get

get

StateRefT'.get

get

StateT.get

get

Std.DHashMap.get

get

Std.DTreeMap.get

get

Std.ExtDHashMap.get

get

Std.ExtHashMap.get

get

Std.ExtHashSet.get

get

Std.HashMap.get

get

Std.HashSet.get

get

Std.TreeMap.get

get

Std.TreeSet.get

get

String.get

get

Subarray.get

get

Substring.get

get

Task.get

get

Thunk.get

get?

Std.DHashMap.get?

get?

Std.DTreeMap.get?

get?

Std.ExtDHashMap.get?

get?

Std.ExtHashMap.get?

get?

Std.ExtHashSet.get?

get?

Std.HashMap.get?

get?

Std.HashSet.get?

get?

Std.TreeMap.get?

get?

Std.TreeSet.get?

get?

String.get?

getAugmentedEnv

Lake.getAugmentedEnv

getAugmentedLeanPath

Lake.getAugmentedLeanPath

getAugmentedLeanSrcPath

Lake.getAugmentedLeanSrcPath

getAugmentedSharedLibPath

Lake.getAugmentedSharedLibPath

getChar

Lean.TSyntax.getChar

getCurrNamespace

Lean.Macro.getCurrNamespace

getCurrentDir

IO.Process.getCurrentDir

getD

Array.getD

getD

List.getD

getD

Option.getD

getD

Std.DHashMap.getD

getD

Std.DTreeMap.getD

getD

Std.ExtDHashMap.getD

getD

Std.ExtHashMap.getD

getD

Std.ExtHashSet.getD

getD

Std.HashMap.getD

getD

Std.HashSet.getD

getD

Std.TreeMap.getD

getD

Std.TreeSet.getD

getD

Subarray.getD

getDM

Option.getDM

getElan?

Lake.getElan?

getElanHome?

Lake.getElanHome?

getElanInstall?

Lake.getElanInstall?

getElanToolchain

Lake.getElanToolchain

getElem!

GetElem?.getElem? (class method)

getElem!_def

LawfulGetElem.getElem!_def (class method)

getElem

GetElem.getElem (class method)

getElem?

[anonymous] (class method)

getElem?_def

LawfulGetElem.getElem?_def (class method)

getEntryGE!

Std.TreeMap.getEntryGE!

getEntryGE

Std.TreeMap.getEntryGE

getEntryGE?

Std.TreeMap.getEntryGE?

getEntryGED

Std.TreeMap.getEntryGED

getEntryGT!

Std.TreeMap.getEntryGT!

getEntryGT

Std.TreeMap.getEntryGT

getEntryGT?

Std.TreeMap.getEntryGT?

getEntryGTD

Std.TreeMap.getEntryGTD

getEntryLE!

Std.TreeMap.getEntryLE!

getEntryLE

Std.TreeMap.getEntryLE

getEntryLE?

Std.TreeMap.getEntryLE?

getEntryLED

Std.TreeMap.getEntryLED

getEntryLT!

Std.TreeMap.getEntryLT!

getEntryLT

Std.TreeMap.getEntryLT

getEntryLT?

Std.TreeMap.getEntryLT?

getEntryLTD

Std.TreeMap.getEntryLTD

getEnv

IO.getEnv

getEnvLeanPath

Lake.getEnvLeanPath

getEnvLeanSrcPath

Lake.getEnvLeanSrcPath

getEnvSharedLibPath

Lake.getEnvSharedLibPath

getEvenElems

Array.getEvenElems

getGE!

Std.TreeSet.getGE!

getGE

Std.TreeSet.getGE

getGE?

Std.TreeSet.getGE?

getGED

Std.TreeSet.getGED

getGT!

Std.TreeSet.getGT!

getGT

Std.TreeSet.getGT

getGT?

Std.TreeSet.getGT?

getGTD

Std.TreeSet.getGTD

getHygieneInfo

Lean.TSyntax.getHygieneInfo

getId

Lean.TSyntax.getId

getKey!

Std.DHashMap.getKey!

getKey!

Std.DTreeMap.getKey!

getKey!

Std.ExtDHashMap.getKey!

getKey!

Std.ExtHashMap.getKey!

getKey!

Std.HashMap.getKey!

getKey!

Std.TreeMap.getKey!

getKey

Std.DHashMap.getKey

getKey

Std.DTreeMap.getKey

getKey

Std.ExtDHashMap.getKey

getKey

Std.ExtHashMap.getKey

getKey

Std.HashMap.getKey

getKey

Std.TreeMap.getKey

getKey?

Std.DHashMap.getKey?

getKey?

Std.DTreeMap.getKey?

getKey?

Std.ExtDHashMap.getKey?

getKey?

Std.ExtHashMap.getKey?

getKey?

Std.HashMap.getKey?

getKey?

Std.TreeMap.getKey?

getKeyD

Std.DHashMap.getKeyD

getKeyD

Std.DTreeMap.getKeyD

getKeyD

Std.ExtDHashMap.getKeyD

getKeyD

Std.ExtHashMap.getKeyD

getKeyD

Std.HashMap.getKeyD

getKeyD

Std.TreeMap.getKeyD

getKeyGE!

Std.TreeMap.getKeyGE!

getKeyGE

Std.TreeMap.getKeyGE

getKeyGE?

Std.TreeMap.getKeyGE?

getKeyGED

Std.TreeMap.getKeyGED

getKeyGT!

Std.TreeMap.getKeyGT!

getKeyGT

Std.TreeMap.getKeyGT

getKeyGT?

Std.TreeMap.getKeyGT?

getKeyGTD

Std.TreeMap.getKeyGTD

getKeyLE!

Std.TreeMap.getKeyLE!

getKeyLE

Std.TreeMap.getKeyLE

getKeyLE?

Std.TreeMap.getKeyLE?

getKeyLED

Std.TreeMap.getKeyLED

getKeyLT!

Std.TreeMap.getKeyLT!

getKeyLT

Std.TreeMap.getKeyLT

getKeyLT?

Std.TreeMap.getKeyLT?

getKeyLTD

Std.TreeMap.getKeyLTD

getKind

Lean.Syntax.getKind

getLE!

Std.TreeSet.getLE!

getLE

Std.TreeSet.getLE

getLE?

Std.TreeSet.getLE?

getLED

Std.TreeSet.getLED

getLT!

Std.TreeSet.getLT!

getLT

Std.TreeSet.getLT

getLT?

Std.TreeSet.getLT?

getLTD

Std.TreeSet.getLTD

getLake

Lake.getLake

getLakeEnv

Lake.getLakeEnv

getLakeHome

Lake.getLakeHome

getLakeInstall

Lake.getLakeInstall

getLakeLibDir

Lake.getLakeLibDir

getLakeSrcDir

Lake.getLakeSrcDir

getLast!

List.getLast!

getLast

List.getLast

getLast?

List.getLast?

getLastD

List.getLastD

getLean

Lake.getLean

getLeanAr

Lake.getLeanAr

getLeanCc

Lake.getLeanCc

getLeanCc?

Lake.getLeanCc?

getLeanIncludeDir

Lake.getLeanIncludeDir

getLeanInstall

Lake.getLeanInstall

getLeanLibDir

Lake.getLeanLibDir

getLeanPath

Lake.getLeanPath

getLeanSharedLib

Lake.getLeanSharedLib

getLeanSrcDir

Lake.getLeanSrcDir

getLeanSrcPath

Lake.getLeanSrcPath

getLeanSysroot

Lake.getLeanSysroot

getLeanSystemLibDir

Lake.getLeanSystemLibDir

getLeanc

Lake.getLeanc

getLeft

Sum.getLeft

getLeft?

Sum.getLeft?

getLine

IO.FS.Handle.getLine

getLine

IO.FS.Stream.getLine (structure field)

getLsb

BitVec.getLsb

getLsb?

BitVec.getLsb?

getLsbD

BitVec.getLsbD

getM

Option.getM

getMax?

Array.getMax?

getModify

getMsb

BitVec.getMsb

getMsb?

BitVec.getMsb?

getMsbD

BitVec.getMsbD

getName

Lean.TSyntax.getName

getNat

Lean.TSyntax.getNat

getNoCache

Lake.getNoCache

getNumHeartbeats

IO.getNumHeartbeats

getPID

IO.Process.getPID

getPkgUrlMap

Lake.getPkgUrlMap

getRandomBytes

IO.getRandomBytes

getRight

Sum.getRight

getRight?

Sum.getRight?

getRootPackage

Lake.getRootPackage

getScientific

Lean.TSyntax.getScientific

getSharedLibPath

Lake.getSharedLibPath

getStderr

IO.getStderr

getStdin

IO.getStdin

getStdout

IO.getStdout

getString

Lean.TSyntax.getString

getTID

IO.getTID

getTaskState

IO.getTaskState

getThe

getThenInsertIfNew?

Std.DHashMap.getThenInsertIfNew?

getThenInsertIfNew?

Std.DTreeMap.getThenInsertIfNew?

getThenInsertIfNew?

Std.ExtDHashMap.getThenInsertIfNew?

getThenInsertIfNew?

Std.ExtHashMap.getThenInsertIfNew?

getThenInsertIfNew?

Std.HashMap.getThenInsertIfNew?

getThenInsertIfNew?

Std.TreeMap.getThenInsertIfNew?

getTryCache

Lake.getTryCache

getUtf8Byte

String.getUtf8Byte

getWorkspace

Lake.MonadWorkspace.getWorkspace (class method)

get_elem_tactic

get_elem_tactic_trivial

globs

[anonymous] (structure field)

goal

main

ground

Lean.Meta.Simp.Config.ground (structure field)

group

IO.FileRight.group (structure field)

groupByKey

Array.groupByKey

groupByKey

List.groupByKey

groupKind

Lean.groupKind

guard

guard

Option.guard

guard_expr

guard_hyp

guard_msgs.diff

guard_target

H

HAdd

HAdd.mk

Instance constructor of HAdd

HAnd

HAnd.mk

Instance constructor of HAnd

HAppend

HAppend.mk

Instance constructor of HAppend

HDiv

HDiv.mk

Instance constructor of HDiv

HEq

HEq.elim

HEq.ndrec

HEq.ndrecOn

HEq.refl

Constructor of HEq

HEq.rfl

HEq.subst

HMod

HMod.mk

Instance constructor of HMod

HMul

HMul.mk

Instance constructor of HMul

HOr

HOr.mk

Instance constructor of HOr

HPow

HPow.mk

Instance constructor of HPow

HShiftLeft

HShiftLeft.mk

Instance constructor of HShiftLeft

HShiftRight

HShiftRight.mk

Instance constructor of HShiftRight

HSub

HSub.mk

Instance constructor of HSub

HXor

HXor.mk

Instance constructor of HXor

Handle

IO.FS.Handle

HasEquiv

HasEquiv.mk

Instance constructor of HasEquiv

HashMap

Std.HashMap

HashSet

Std.HashSet

Hashable

Hashable.mk

Instance constructor of Hashable

HomogeneousPow

HomogeneousPow.mk

Instance constructor of HomogeneousPow

HygieneInfo

Lean.Syntax.HygieneInfo

hAdd

HAdd.hAdd (class method)

hAnd

HAnd.hAnd (class method)

hAppend

HAppend.hAppend (class method)

hDiv

HDiv.hDiv (class method)

hIterate

Fin.hIterate

hIterateFrom

Fin.hIterateFrom

hMod

HMod.hMod (class method)

hMul

HMul.hMul (class method)

hOr

HOr.hOr (class method)

hPow

HPow.hPow (class method)

hShiftLeft

HShiftLeft.hShiftLeft (class method)

hShiftRight

HShiftRight.hShiftRight (class method)

hSub

HSub.hSub (class method)

hXor

HXor.hXor (class method)

hasDecl

Lean.Macro.hasDecl

hasFinished

IO.hasFinished

hasNext

String.Iterator.hasNext

hasPrev

String.Iterator.hasPrev

hash

BitVec.hash

hash

Hashable.hash (class method)

hash

String.hash

hash_eq

hash_eq

LawfulHashable.hash_eq (class method)

have (0) (1)

[have']

head!
  List.head!

head
  List.head

head?
  List.head?

headD
  List.headD

helim
  Subsingleton.helim

heq_of_eq

heq_of_eqRec_eq

heq_of_heq_of_eq

hrecOn
  Quot.hrecOn

hrecOn
  Quotient.hrecOn

hygiene
  in tactics

hygieneInfoKind
  Lean.hygieneInfoKind

hygienic
  tactic.hygienic

I

IO

IO.AccessRight

IO.AccessRight.flags

IO.AccessRight.mk
  Constructor of IO.AccessRight

IO.Error

IO.Error.alreadyExists
  Constructor of IO.Error

IO.Error.hardwareFault
  Constructor of IO.Error

IO.Error.illegalOperation
  Constructor of IO.Error

IO.Error.inappropriateType
  Constructor of IO.Error

IO.Error.interrupted
  Constructor of IO.Error

IO.Error.invalidArgument
  Constructor of IO.Error

IO.Error.noFileOrDirectory
  Constructor of IO.Error

IO.Error.noSuchThing
  Constructor of IO.Error

IO.Error.otherError
  Constructor of IO.Error

IO.Error.permissionDenied
  Constructor of IO.Error

IO.Error.protocolError

Constructor of IO.Error

IO.Error.resourceBusy

Constructor of IO.Error

IO.Error.resourceExhausted

Constructor of IO.Error

IO.Error.resourceVanished

Constructor of IO.Error

IO.Error.timeExpired

Constructor of IO.Error

IO.Error.toString

IO.Error.unexpectedEof

Constructor of IO.Error

IO.Error.unsatisfiedConstraints

Constructor of IO.Error

IO.Error.unsupportedOperation

Constructor of IO.Error

IO.Error.userError

Constructor of IO.Error

IO.FS.DirEntry

IO.FS.DirEntry.mk

Constructor of IO.FS.DirEntry

IO.FS.DirEntry.path

IO.FS.Handle

IO.FS.Handle.flush

IO.FS.Handle.getLine

IO.FS.Handle.isTty

IO.FS.Handle.lock

IO.FS.Handle.mk

IO.FS.Handle.putStr

IO.FS.Handle.putStrLn

IO.FS.Handle.read

IO.FS.Handle.readBinToEnd

IO.FS.Handle.readBinToEndInto

IO.FS.Handle.readToEnd

IO.FS.Handle.rewind

IO.FS.Handle.truncate

IO.FS.Handle.tryLock

IO.FS.Handle.unlock

IO.FS.Handle.write

IO.FS.Metadata

IO.FS.Metadata.mk

Constructor of IO.FS.Metadata

IO.FS.Mode

IO.FS.Mode.append

Constructor of IO.FS.Mode

IO.FS.Mode.read

Constructor of IO.FS.Mode

IO.FS.Mode.readWrite

Constructor of IO.FS.Mode

IO.FS.Mode.write

Constructor of IO.FS.Mode

IO.FS.Mode.writeNew

Constructor of IO.FS.Mode

IO.FS.Stream

IO.FS.Stream.Buffer

IO.FS.Stream.Buffer.mk

Constructor of IO.FS.Stream.Buffer

IO.FS.Stream.mk

Constructor of IO.FS.Stream

IO.FS.Stream.ofBuffer

IO.FS.Stream.ofHandle

IO.FS.Stream.putStrLn

IO.FS.createDir

IO.FS.createDirAll

IO.FS.createTempDir

IO.FS.createTempFile

IO.FS.lines

IO.FS.readBinFile

IO.FS.readFile

IO.FS.realPath

IO.FS.removeDir

IO.FS.removeDirAll

IO.FS.removeFile

IO.FS.rename

IO.FS.withFile

IO.FS.withIsolatedStreams

IO.FS.withTempDir

IO.FS.withTempFile

IO.FS.writeBinFile

IO.FS.writeFile

IO.FileRight

IO.FileRight.flags

IO.FileRight.mk

Constructor of IO.FileRight

IO.Process.Child

IO.Process.Child.kill

IO.Process.Child.takeStdin

IO.Process.Child.tryWait

IO.Process.Child.wait

IO.Process.Output

IO.Process.Output.mk

Constructor of IO.Process.Output

IO.Process.SpawnArgs

IO.Process.SpawnArgs.mk

Constructor of IO.Process.SpawnArgs

IO.Process.Stdio

IO.Process.Stdio.inherit

Constructor of IO.Process.Stdio

IO.Process.Stdio.null

Constructor of IO.Process.Stdio

IO.Process.Stdio.piped

Constructor of IO.Process.Stdio

IO.Process.Stdio.toHandleType

IO.Process.StdioConfig

IO.Process.StdioConfig.mk

Constructor of IO.Process.StdioConfig

IO.Process.exit

IO.Process.getCurrentDir

IO.Process.getPID

IO.Process.output

IO.Process.run

IO.Process.setCurrentDir

IO.Process.spawn

IO.Promise

IO.Promise.isResolved

IO.Promise.new

IO.Promise.resolve

IO.Promise.result!

IO.Promise.result?

IO.Promise.resultD

IO.Ref

IO.TaskState

IO.TaskState.finished

Constructor of IO.TaskState

IO.TaskState.running

Constructor of IO.TaskState

IO.TaskState.waiting

Constructor of IO.TaskState

IO.addHeartbeats

IO.appDir

IO.appPath

IO.asTask

IO.bindTask

IO.cancel

IO.chainTask

IO.checkCanceled

IO.currentDir

IO.eprint

IO.eprintln

IO.getEnv

IO.getNumHeartbeats

IO.getRandomBytes

IO.getStderr

IO.getStdin

IO.getStdout

IO.getTID

IO.getTaskState

IO.hasFinished

IO.iterate

IO.lazyPure

IO.mapTask

IO.mapTasks

IO.mkRef

IO.monoMsNow

IO.monoNanosNow

IO.ofExcept

IO.print

IO.println

IO.rand

IO.setAccessRights

IO.setRandSeed

IO.setStderr

IO.setStdin

IO.setStdout

IO.sleep

IO.toEIO

IO.userError

IO.wait

IO.waitAny

IO.withStderr

IO.withStdin

IO.withStdout

ISize

ISize.abs

ISize.add

ISize.complement

ISize.decEq

ISize.decLe

ISize.decLt

ISize.div

ISize.land

ISize.le

ISize.lor

ISize.lt

ISize.maxValue

ISize.minValue

ISize.mod

ISize.mul

ISize.neg

ISize.ofBitVec

ISize.ofInt

ISize.ofIntLE

ISize.ofIntTruncate

ISize.ofNat

ISize.shiftLeft

ISize.shiftRight

ISize.size

ISize.sub

ISize.toBitVec

ISize.toFloat

ISize.toFloat32

ISize.toInt

ISize.toInt16

ISize.toInt32

ISize.toInt64

ISize.toInt8

ISize.toNatClampNeg

ISize.xor

Id

Id.run

Ident
  Lean.Syntax.Ident

Iff

Iff.elim

Iff.intro
  Constructor of Iff

Inhabited

Inhabited.mk

Instance constructor of Inhabited

Int

Int.add

Int.bdiv

Int.bmod

Int.cast

Int.decEq

Int.ediv

Int.emod

Int.fdiv

Int.fmod

Int.gcd

Int.lcm

Int.le

Int.lt

Int.mul

Int.natAbs

Int.neg

Int.negOfNat

Int.negSucc

Constructor of Int

Int.not

Int.ofNat

Constructor of Int

Int.pow

Int.repr

Int.shiftRight

Int.sign

Int.sub

Int.subNatNat

Int.tdiv

Int.tmod

Int.toISize

Int.toInt16

Int.toInt32

Int.toInt64

Int.toInt8

Int.toNat

Int.toNat?

Int16

Int16.abs

Int16.add

Int16.complement

Int16.decEq

Int16.decLe

Int16.decLt

Int16.div

Int16.land

Int16.le

Int16.lor

Int16.lt

Int16.maxValue

Int16.minValue

Int16.mod

Int16.mul

Int16.neg

Int16.ofBitVec

Int16.ofInt

Int16.ofIntLE

Int16.ofIntTruncate

Int16.ofNat

Int16.shiftLeft

Int16.shiftRight

Int16.size

Int16.sub

Int16.toBitVec

Int16.toFloat

Int16.toFloat32

Int16.toISize

Int16.toInt

Int16.toInt32

Int16.toInt64

Int16.toInt8

Int16.toNatClampNeg

Int16.xor

Int32

Int32.abs

Int32.add

Int32.complement

Int32.decEq

Int32.decLe

Int32.decLt

Int32.div

Int32.land

Int32.le

Int32.lor

Int32.lt

Int32.maxValue

Int32.minValue

Int32.mod

Int32.mul

Int32.neg

Int32.ofBitVec

Int32.ofInt

Int32.ofIntLE

Int32.ofIntTruncate

Int32.ofNat

Int32.shiftLeft

Int32.shiftRight

Int32.size

Int32.sub

Int32.toBitVec

Int32.toFloat

Int32.toFloat32

Int32.toISize

Int32.toInt

Int32.toInt16

Int32.toInt64

Int32.toInt8

Int32.toNatClampNeg

Int32.xor

Int64

Int64.abs

Int64.add

Int64.complement

Int64.decEq

Int64.decLe

Int64.decLt

Int64.div

Int64.land

Int64.le

Int64.lor

Int64.lt

Int64.maxValue

Int64.minValue

Int64.mod

Int64.mul

Int64.neg

Int64.ofBitVec

Int64.ofInt

Int64.ofIntLE

Int64.ofIntTruncate

Int64.ofNat

Int64.shiftLeft

Int64.shiftRight

Int64.size

Int64.sub

Int64.toBitVec

Int64.toFloat

Int64.toFloat32

Int64.toISize

Int64.toInt

Int64.toInt16

Int64.toInt32

Int64.toInt8

Int64.toNatClampNeg

Int64.xor

Int8

Int8.abs

Int8.add

Int8.complement

Int8.decEq

Int8.decLe

Int8.decLt

Int8.div

Int8.land

Int8.le

Int8.lor

Int8.lt

Int8.maxValue

Int8.minValue

Int8.mod

Int8.mul

Int8.neg

Int8.ofBitVec

Int8.ofInt

Int8.ofIntLE

Int8.ofIntTruncate

Int8.ofNat

Int8.shiftLeft

Int8.shiftRight

Int8.size

Int8.sub

Int8.toBitVec

Int8.toFloat

Int8.toFloat32

Int8.toISize

Int8.toInt

Int8.toInt16

Int8.toInt32

Int8.toInt64

Int8.toNatClampNeg

Int8.xor

IntCast

IntCast.mk

Instance constructor of IntCast

IntModule

Lean.Grind.IntModule

IsCharP

Lean.Grind.IsCharP

IsInfix

List.IsInfix

IsPrefix

List.IsPrefix

IsSuffix

List.IsSuffix

Iterator

String.Iterator

i

String.Iterator.i (structure field)

id_map

LawfulFunctor.id_map (class method)

identKind

Lean.identKind

identifier

identifier

raw

idxOf

Array.idxOf

idxOf

List.idxOf

idxOf?

Array.idxOf?

idxOf?

List.idxOf?

if ... then ... else ...

if h : ... then ... else ...

implicitDefEqProofs

Lean.Meta.Simp.Config.implicitDefEqProofs (structure field)

impredicative

inaccessible

ind

Quot.ind

ind

Quotient.ind

ind

Squash.ind

indentD

Std.Format.indentD

index

Lean.Meta.DSimp.Config.index (structure field)

index

Lean.Meta.Simp.Config.index (structure field)

index

of inductive type

indexed family

of types

induction

induction

Fin.induction

inductionOn

Fin.inductionOn

inductionOn

Nat.div.inductionOn

inductionOn

Nat.mod.inductionOn

inductive.autoPromoteIndices

inductiveCheckResultingUniverse

bootstrap.inductiveCheckResultingUniverse

inferInstance

inferInstanceAs

infer_instance

inhabitedLeft

PSum.inhabitedLeft

inhabitedLeft

Sum.inhabitedLeft

inhabitedRight

PSum.inhabitedRight

inhabitedRight

Sum.inhabitedRight

inheritEnv

IO.Process.SpawnArgs.args (structure field)

init (Lake command)

injection

injections

inner

Std.DHashMap.Equiv.inner (structure field)

inner

Std.DTreeMap.Raw.inner (structure field)

inner

Std.ExtHashSet.inner (structure field)

inner

Std.HashMap.Equiv.inner (structure field)

inner

Std.HashMap.Raw.inner (structure field)

inner

Std.HashSet.Equiv.inner (structure field)

inner

Std.HashSet.Raw.inner (structure field)

inner

Std.HashSet.inner (structure field)

inner

Std.TreeMap.Raw.inner (structure field)

inner

Std.TreeSet.Raw.inner (structure field)

insert

List.insert

insert

Std.DHashMap.insert

insert

Std.DTreeMap.insert

insert

Std.ExtDHashMap.insert

insert

Std.ExtHashMap.insert

insert

Std.ExtHashSet.insert

insert

Std.HashMap.insert

insert

Std.HashSet.insert

insert

Std.TreeMap.insert

insert

Std.TreeSet.insert

insertIdx!

Array.insertIdx!

insertIdx

Array.insertIdx

insertIdx

List.insertIdx

insertIdxIfInBounds

Array.insertIdxIfInBounds

insertIdxTR

List.insertIdxTR

insertIfNew

Std.DHashMap.insertIfNew

insertIfNew

Std.DTreeMap.insertIfNew

insertIfNew

Std.ExtDHashMap.insertIfNew

insertIfNew

Std.ExtHashMap.insertIfNew

insertIfNew

Std.HashMap.insertIfNew

insertIfNew

Std.TreeMap.insertIfNew

insertMany

Std.DHashMap.insertMany

insertMany

Std.DTreeMap.insertMany

insertMany

Std.ExtDHashMap.insertMany

insertMany

Std.ExtHashMap.insertMany

insertMany

Std.ExtHashSet.insertMany

insertMany

Std.HashMap.insertMany

insertMany

Std.HashSet.insertMany

insertMany

Std.TreeMap.insertMany

insertMany

Std.TreeSet.insertMany

insertManyIfNewUnit

Std.ExtHashMap.insertManyIfNewUnit

insertManyIfNewUnit

Std.HashMap.insertManyIfNewUnit

insertManyIfNewUnit

Std.TreeMap.insertManyIfNewUnit

insertionSort

Array.insertionSort

instance synthesis

intCast

IntCast.intCast (class method)

intCast

[anonymous] (class method)

intCast_neg

[anonymous] (class method)

intCast_ofNat

[anonymous] (class method)

intMax

BitVec.intMax

intMin

BitVec.intMin

intercalate

List.intercalate

intercalate

String.intercalate

intercalateTR

List.intercalateTR

interpolatedStrKind

Lean.interpolatedStrKind

interpolatedStrLitKind

Lean.interpolatedStrLitKind

intersperse

List.intersperse

intersperseTR

List.intersperseTR

intro (0) (1)

intro | ... => ... | ... => ...

intros

invImage

inv_zero

[anonymous] (class method)

iota

Lean.Meta.DSimp.Config.iota (structure field)

iota

Lean.Meta.Simp.Config.iota (structure field)

isAbsolute

System.FilePath.isAbsolute

isAlpha

Char.isAlpha

isAlphanum

Char.isAlphanum

isDigit

Char.isDigit

isDir

System.FilePath.isDir

isEmpty

Array.isEmpty

isEmpty

List.isEmpty

isEmpty

Std.DHashMap.isEmpty

isEmpty

Std.DTreeMap.isEmpty

isEmpty

Std.ExtDHashMap.isEmpty

isEmpty

Std.ExtHashMap.isEmpty

isEmpty

Std.ExtHashSet.isEmpty

isEmpty

Std.Format.isEmpty

isEmpty

Std.HashMap.isEmpty

isEmpty

Std.HashSet.isEmpty

isEmpty

Std.TreeMap.isEmpty

isEmpty

Std.TreeSet.isEmpty

isEmpty

String.isEmpty

isEmpty

Substring.isEmpty

isEmscripten

System.Platform.isEmscripten

isEq

Ordering.isEq

isEqSome

Option.isEqSome

isEqv

Array.isEqv

isEqv

List.isEqv

isExclusiveUnsafe

isFinite

Float.isFinite

isFinite

Float32.isFinite

isGE

Ordering.isGE

isGT

Ordering.isGT

isInf

Float.isInf

isInf

Float32.isInf

isInt

String.isInt

isLE

Ordering.isLE

isLT

Ordering.isLT

isLeft

Sum.isLeft

isLower

Char.isLower

isLt

Fin.isLt (structure field)

isNaN