Chapter 1

A section from Eric Wieser's thesis, for Zulip

1.1 Scalar actions

Scalar actions (a generalization of group actions) are ubiquitous in mathematics, usually appearing under the guise of multiplication; we write $x+y\mathrm{i}$ when $x, y : \mathbb{R}$ but $i : \mathbb{C}$, or $xi+yj+zk$ to scale unit vectors $i, j, k : \mathbb{R}^3$ by coefficients $x, y, z : \mathbb{R}$, or qvq^{-1} to apply a transformation represented by a quaternion $q : \mathbb{H}$ to a vector $v : \mathbb{R}^3$. Very few programming languages support implicit multiplication-by-juxtaposition like this, but many allow this kind of expression to be written using the regular multiplication operator, $*$. In programming languages for scientific computing like python or Julia, scalar actions fall out as a special case of "broadcasting"[1, fig. 1e], and can be written with the regular multiplication operators. Sadly, lean does not even provide the luxury of using the regular multiplication with the $*$ operator, as this requires the two inputs and the output to all be of the same type[1].

mathlib's solution to these difficulties is to define a new scalar multiplication operator $\bullet$. In this section we explore through examples how Lean's typeclasses are used to implement a flexible range of scalar actions, illustrate some of the problems which come up when using them such as compatibility of actions and non-definitionally-equal diamonds, and note how these problems can be solved. This section is a significantly extended version[2] of [2], notably including more recent work on right actions in section 1.1.7.

[1] While this restriction was lifted in Lean 4, it was preserved in mathlib, and as described in section 1.1.8, simply opens a door to more problems.

[2] The original had a very short page limit.

1.1.1 Basic typeclasses

The typeclass we are most interested in this section is `has_smul M α`, which equips a type `α` with an action by elements of `M` denoted `m • a`. Here, `smul` stands for *s*calar *mul*tiplication. In practice, this is almost always used for monoid or group actions, which are actions that satisfies the additional fields in `mul_action M α`:

```
class has_smul (M : Type*) (α : Type*) := (smul : M → α → α)

infixr ` • `:73 := has_smul.smul

class mul_action (M : Type*) (α : Type*) [monoid M] extends has_smul M α :=
(one_smul : ∀ a : α, (1 : M) • a = a)
(mul_smul : ∀ (x y : M) (a : α), (x * y) • a = x • y • a)
```

Note here that because we use `[monoid M]` instead of `extends monoid M`, we are stating that `mul_action M α` requires `M` to already be equipped with a monoid structure, rather than allowing `mul_action M α` to itself provide that structure.

mathlib extends these two typeclasses with a variety of additional axioms (i.e., fields holding proofs) for when `M` and `α` are themselves equipped with extra structures, such as distributivity over addition and actions by zero. Figure 1.1a shows the majority of these typeclasses, while details of their fields can be found either in [3, section 5.1] or in the mathlib docs.

1.1.2 Elementary actions

Scalar actions can be roughly divided into two types: elementary actions which are intrinsic to a particular family of types, and derived actions which operate elementwise on "bigger" types built out of smaller types. We will start by giving some examples of the former.

Left multiplication

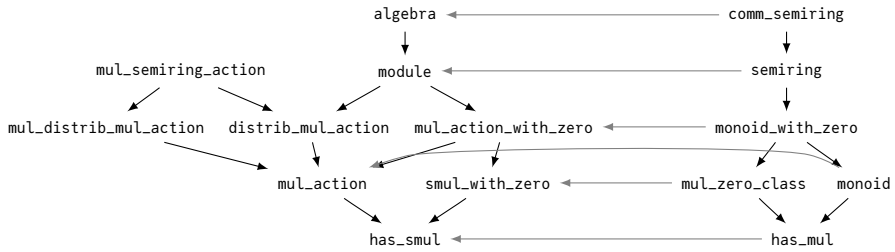
One of the simplest actions we can construct is that of left-multiplication, with `a • b = a * b`, which mathlib provides as follows.

```
instance has_mul.to_has_smul (α : Type*) [has_mul α] : has_smul α α := { smul := (*) }
```

As the properties of the multiplication on `α` becomes stronger, so do those of this scalar action on `α`; for instance when we have `monoid α` we can deduce `mul_action α α`, and when we have `semiring α` we can deduce `module α α`. Figure 1.1b shows these available left multiplication structures, and the corresponding links with fig. 1.1a are shown with grey arrows.

Repeated addition and subtraction

Another simple action we can construct is that of repeated addition (an instance of `module ℕ α`) when `α` is a commutative additive monoid, which can be defined recursively for a natural number as `(0 : ℕ) • x = 0` and `∀ n : ℕ, (n + 1) • x = n • x + x`. A similar approach can be used to



(a) Scalar action typeclasses (b) Multiplicative typeclasses

Figure 1.1: Hierarchy of scalar action and multiplicative typeclasses

Arrows indicate implications. Grey arrows indicate implied left-multiplication actions.

define a `module ℤ α` instance when `α` additionally forms an additive group. These are respectively promoted to `algebra ℕ α` and `algebra ℤ α` structures when `α` forms a `semiring` or `ring`.

Application of endomorphisms and automorphisms

If we have an endomorphism or automorphism `f : E` (a structure-preserving map or equivalence, respectively, from `α` to itself), then we can obtain a `mul_action E α` instance characterized by `f • x = f x`; that is, the action is just function application. Depending on the endomorphism/automorphism in question, this `mul_action` instance can be promoted to a stronger type: for instance, if it is an endomorphism of additive monoids, then this action can be promoted to `module (add_monoid.End α) α` [*mathlib4#8395*]; if it is an automorphism of R-algebras, then this action can be promoted to `mul_semiring_action (A ≃ₐ[R] A) A` [*mathlib#8724*]. In Lean, this latter instance is defined as follows, where `($)` is the function application operator.

```
instance alg_equiv.apply_mul_semiring_action : mul_semiring_action (A₁ ≃ₐ[R] A₁) A₁ :=
{ smul := ($),
  smul_one  := alg_equiv.map_one,   smul_mul := alg_equiv.map_mul,
  smul_zero := alg_equiv.map_zero,  smul_add := alg_equiv.map_add,
  one_smul  := λ _, rfl,            mul_smul := λ _ _ _, rfl }
```

Further instances were added to mathlib in [*mathlib4#8396*].

These actions act as important glue between the typeclass approach of formalizing actions we focus on in this section, and an alternate morphism-based approach. These two approaches are shown in the columns of table 1.1. The typeclass approach is great for when a canonical action is available; but when multiple possible actions are available, such as in representation theory, the morphism approach is more predictable. To go to an entry in the right column from a term `f` whose type is in the left column, we can build the action characterized by `m • x = f m • x`, where the second `•` is the action induced from morphism application we just described. These "actions after composing a function" are generated from the family of `comp_hom` definitions in the center column. The rightwards arrows in table 1.1 show the more direct definitions [*mathlib#8968*] that

Typeclass		Morphism
`mul_action M α`	`mul_actionn` / `mul_actionn`	`M →* function.End α`
`distrib_mul_action M A`	`distrib_mul_actionn` / `distrib_mul_actionn`	`M →* add_monoid.End A`
`mul_distrib_mul_action M A`	`mul_distrib_mul_actionn` / `mul_distrib_mul_actionn`	`M →* monoid.End A`
`module R M`	`modulen` / `modulen`	`R →+* add_monoid.End M`
`mul_semiring_action R S`	`mul_semiring_actionn` / `mul_semiring_actionn`	`R →* (S →+* S)`

Table 1.1: Morphism- vs typeclass-based representations of actions in mathlib, and translations between them.

Note the composition implied by the `comp_hom` name is referring to composition with the endomorphism action in section 1.1.2. For the rightwards arrows, a similar family of maps is available for the *auto*morphisms.

exist to go in the reverse direction.

1.1.3 Derived actions

A typical example of a module action might be that of a scalar $\mathbb{R}$ on the vector space $\mathbb{R}^3$ (`fin 3 → ℝ`), which multiplies each component separately. After making the obvious generalization to an arbitrary type `α` and index set `ι`, the easy way to write this down would be as follows, where again we can provide a stronger `module α (ι → α)` if we know `α` forms a `semiring`.

```
instance function.has_smul (ι α : Type*) [has_mul α] : has_smul α (ι → α) :=
{ smul := λ r v, (λ i, r * v i) }
```

This definition is perfectly fine for the action we wanted, but we can still generalize it much more. Consider now the action on matrices[3] `ι₁ → ι₂ → R` by their coefficients `R`. We would like to show `has_smul R (ι₁ → ι₂ → R)`, but that doesn't match the `function.has_smul` instance we just defined. While we could obviously define this operation trivially just as we did there, we would have to do so again if working with a vector of matrices or similar; this approach doesn't scale. Instead, we should be *deriving* an action of an arbitrary type `M` on `ι → α` from its action on `α`, such that this exploits the chaining that occurs during typeclass search.

Function types, through their codomain

mathlib defines such a derived action on function types as follows:

```
instance function.has_smul' (ι M α : Type*) [has_smul M α] : has_smul M (ι → α) :=
{ smul := λ r v, (λ i, r • v i) }
```

[3] though not quite mathlib's spelling of them.

This instance is strictly more general than the previous one—typeclass search will recover our original `has_smul α (ι → α)` instance by setting `M := α` and finding `has_smul α α` from `has_mul.to_has_smul`, but can also find the `has_smul R (ι₁ → ι₂ → R)` we wanted by setting `M := R` and `α = (ι₂ → R)`, and finding `has_smul R (ι₂ → α)` by recursive application of this instance.

This action propagates the axioms of the original action of `M` on `α`; we can show that if we additionally have `[module M α]`, then our action above satisfies `module M (ι → α)`, and similarly for all the other typeclasses in fig. 1.1a.

Sets, through their elements

On `set`s, `mathlib` defines a derived action via the action on the elements [mathlib#997], as

```
instance has_smul_set [has_smul α β] : has_smul α (set β) :=
{ smul := λ a s, (λ b, a • b) '' s }
```

Which satisfies `a • {x, y} = {a • x, a • y}`. Once again, the axioms of the original action are propagated; though to a much lesser extent, as `0 • s = 0` (where `0` on the RHS is the set `{0}`) does not hold if `s` is the empty set. An analogous construction exists for `finset`s [mathlib#12865].

For historical reasons, these instance are not globally available by default; they must be requested locally using `open_locale pointwise`.

Morphisms of additive groups, through their codomain

For functions in section 1.1.3 and sets in section 1.1.3, the action we describe contains no proof obligations—we did not need to know any properties of `[has_smul M α]` to define `has_smul M (ι → α)`. This is not always the case; we cannot conclude `has_smul R (M →+ N)` from `[has_smul R N]` as we don't know enough about the action of `R` on `N` to know if additive maps remain additive. Moving away from the root node in fig. 1.1a towards stronger typeclasses is usually enough to resolve this—in this particular case, we can conclude `distrib_mul_action R (M →+ N)` from `[distrib_mul_action R N]` [*mathlib#6891*].

Polynomials, through their coefficients

Another simple example of a derived action is that polynomials $R[X]$ (`polynomial R` or `R[X]`) inherit an action by a type S when S acts upon their coefficients. This is a stronger statement than `has_smul R R[X]` (that polynomials are acted upon *by* their coefficients), as it generalizes in the same way as the instance we saw in section 1.1.3 to allow `R` to act on `R[X][X]` (a polynomial in two variables). Until [mathlib#4784], only this weaker statement was available.

As with the action on additive morphisms in section 1.1.3, we cannot directly conclude `has_smul S R[X]` from `has_smul S R`, this time because if we had our action on the coefficients satisfy `1 • (0 : R) = 1`, then we would end up with `1 • (0 : R[X]) = 1 + X + X^2 + ⋯` which has infinite support and thus is not a polynomial at all! Once again, we can instead start at a stronger typeclass in fig. 1.1a and provide the typeclass instance showing that `distrib_mul_action S R`

implies `distrib_mul_action s R[X]` [*mathlib#7664*], as `distrib_mul_action` provides the crucial proof that `1 • (0 : R) = 0` and ensures that the scaled polynomial is well formed. This instance solves the `has_smul R R[X][X]` case by having us search for `distrib_mul_action R R[X]` and then `distrib_mul_action R R` before finally finishing the search at the instance in section 1.1.2.

Polynomials in mathlib are at the end of a chain of simpler constructions; they are defined as the special case of a "monoid algebra" whose generators are the natural numbers corresponding to the powers of X. A monoid algebras is in turn defined as a "finitely supported function" representing its coefficients; functions which are zero at all but finitely-many points. Before the upgraded `has_smul` instance could be put on polynomials [mathlib#4784], the author had to first upgrade a corresponding instance on monoid algebras [*mathlib#4365*], which in turn relied on an earlier upgrade to the instance on finitely supported functions [mathlib#284]; a sequence spanning over two years!

Multivariate polynomials are treated separately in mathlib, but the handling of scalar actions largely mirrors the univariate case; this time, it was the author's turn to perform the generalization from `has_smul R R[X]` to `has_smul S R[X]`, in [*mathlib#6533*].

Interactions with other actions

The strategy used for additive maps in section 1.1.3 of choosing stronger typeclasses from fig. 1.1a can only take us so far. Once we start working with types that themselves ingrain a preferred action, we need some additional tools. For instance, the closely related types for `R`-linear maps `M →ₗ[R] N` and `R`-submodules `submodule R N` ingrain a preferred `R`-action. For the first of these cases, we can start by attempting to build a general action by an arbitrary type `α`. If we do this we find ourselves left with two proof obligations, indicated by the `show ..., from` syntax.

```
instance {α R M N : Type*}
  [semiring R] [add_comm_monoid M] [add_comm_monoid N] [has_smul α N] [module R M] [module R N] :
  has_smul α (M →ₗ[R] N) :=
{ smul := λ a f, { to_fun := λ m, a • f m,
                   map_add' := λ m₁ m₂, (congr_arg _ $ f.map_add _ _).trans $
                     show a • (f m₁ + f m₂) = a • f m₁ + a • f m₂, from sorry,
                   map_smul' := λ r m, (congr_arg _ $ f.map_smul _ _).trans $
                     show a • r • f m = r • a • f m, from sorry } }
```

The goal in `map_add'` tells us we need to strengthen `[has_smul α N]` to `[monoid α] [distrib_mul_action α N]`, just as we already would have done when building the instance in section 1.1.3.

The goal in `map_smul'` is more troublesome. The easy way out is to replace `α` with a commutative `R` so that our statement becomes

```
instance {α M N : Type*} [comm_semiring R] [add_comm_monoid M] [add_comm_monoid N] [module R M] [module R
↪  N] :
  has_smul R (M →ₗ[R] N) :=
```

and the `sorry` can be closed with `a • r • f m = (a * r) • f m = (r * a) • f m = r • a • f m` which follows from the axioms of `mul_action` and commutativity of `R`. Another approach would

be to require `R` to be an `α`-algebra with `[algebra α R]`, and that the `α`-action on `R` and `N` is compatible with the `R`-action on `N`.

To best solve this problem, mathlib provides two additional typeclasses about scalar actions. The first expresses the compatibility condition we would need to use `[algebra α R]` as mentioned above, as

```
class is_scalar_tower (M N α : Type*) [has_smul M N] [has_smul N α] [has_smul M α] : Prop :=
(smul_assoc : ∀ (x : M) (y : N) (z : α), (x • y) • z = x • (y • z))
```

The name alludes to towers of algebras, which is described in more detail in [4, Section 4.2]. Our particular problem can be solved more directly with the second typeclass, `[smul_comm_class α R N]`, which expresses exactly the condition we require:

```
class smul_comm_class (M N α : Type*) [has_smul M α] [has_smul N α] : Prop :=
(smul_comm : ∀ (m : M) (n : N) (a : α), m • n • a = n • m • a)
```

After this typeclass was introduced in [mathlib#4770], the author contributed and drove the review of a large number of instances of it [*mathlib#6534*; *mathlib#6614*; *mathlib#8965*; *mathlib#15876*; *mathlib#10262*], most notably those for polynomials [*mathlib#6542*; *mathlib#6592*], product types [*mathlib#6139*], and the repeated addition actions in section 1.1.2 [*mathlib#5205*; *mathlib#5369*; *mathlib#5509*; *mathlib#13174*].

1.1.4 Algebras and not-quite algebras

The mathlib `algebra R A` describes an associative unital `R`-algebra over `A` given a `comm_semiring R` and `semiring A`. The definition is roughly

```
class algebra (R A : Type*) [comm_semiring R] [semiring A] extends has_smul R A :=
(algebra_map : R →+* A)
(commutes : ∀ r x, algebra_map r * x = x * algebra_map r)
(smul_def : ∀ r x, r • x = algebra_map r * x)
```

which states that there is a canonical ring homomorphism from `R` to `A` which agrees with `•` and sends `R` to the center of `A`. This parameterization of the axioms is difficult to generalize to A being a nonunital and non-associative ring. However, mathlib also provides a definition to construct an algebra from an alternate set of axioms:

```
def algebra.of_module (R A : Type*) [comm_semiring R] [semiring A] [module R A]
  (h₁ : ∀ (r : R) (x y : A), (r • x) * y = r • (x * y))
  (h₂ : ∀ (r : R) (x y : A), x * (r • y) = r • (x * y)) : algebra R A := sorry
```

If we look carefully, we note that `h₁` and `h₂` closely resemble `smul_assoc` and `smul_comm` from section 1.1.3, but with some `*`s substituted for `•`s. But if we look back to section 1.1.2, we remember that when `x` and `y` are the same type, `x * y = x • y` by definition! This means that `h₁` and `h₂` correspond directly with `is_scalar_tower R A A` and `smul_comm_class R A A`, respectively.

This is a valuable insight, because it allows us to use the follow sequences of typeclass arguments interchangeably:

```
variables [comm_semiring R] [semiring A] [algebra R A]
```

```
variables [comm_semiring R] [semiring A] [module R A] [is_scalar_tower R A A] [smul_comm_class R A A]
```

Knowing this, it becomes immediately obvious how to generalize various statements to non-unital algebras (which were needed in [mathlib#7932]); we switch from from the first form to the second form, and then replace `[semiring A]` with `[non_unital_semiring A]`, something which was not permitted on the unexpanded version. Another generalization this permits is one that allows putting "most of" an `R'`-algebra structure on `A` when `R'` is only a monoid, which comes up for instance when `R' := units R`. In this case, we replace `[comm_semiring R] [semiring A] [module R A]` with `[monoid R] [semiring A] [distrib_mul_action R A]`. This generalization was used when proving intermediate results needed for Sylvester's law of inertia [mathlib#7416].

1.1.5 Typeclass diamonds

Frequently, there are multiple ways for Lean to construct a typeclass. One example arises when considering how the ring of module endomorphisms, `M →ₗ[R] M` with `*` as composition, acts on itself; `module (M →ₗ[R] M) (M →ₗ[R] M)`. There are two routes to find this instance, as shown in fig. 1.2; we call this situation a "typeclass diamond", in light of the shape of the figure[4]. Once route is to use the action from section 1.1.2, where `•` is just defined as `*`; this gives an action characterized by `(f • g) x = (f * g) x = f (g x)`. The other route is to combine the codomain-wise action from section 1.1.3 with the endomorphism action from section 1.1.2; this gives an action characterized by `(f • g) x = f • g x = f (g x)`. Depending on the order of the search, Lean could take either of these two paths; though as the result is the same (the paths "commute"), we do not care here.

Non-commuting diamonds

While Lean does not care about the existence of multiple paths and will happily just pick one, for the typeclass to be useful it needs to be predictable to the user—all they see is a `•` in the goal

[4] Although in fact not all such situations actually resemble diamonds!

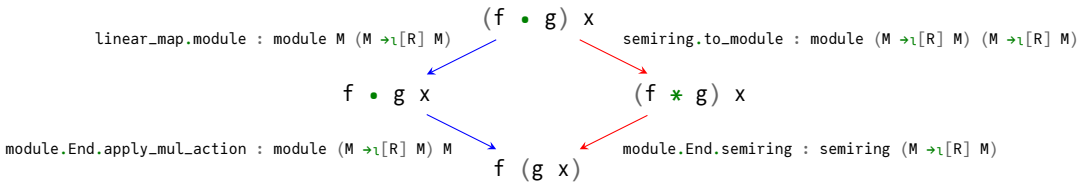


Figure 1.2: A commuting diamond in typeclass search

Here the nodes show the expression, while the edges represent using a certain instance to populate the `•` or `*`, and point towards the definition implied by that instance.

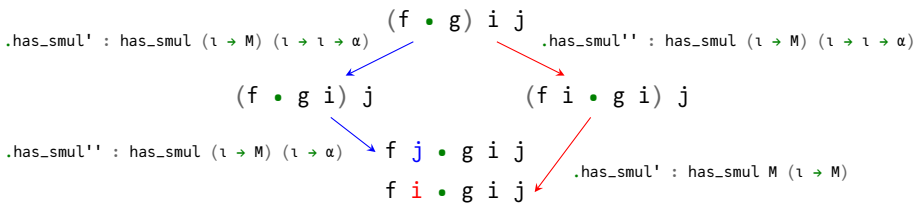


Figure 1.3: A non-commuting diamond in typeclass search

state. This means that whenever we have a diamond, we want all the paths to produce the same • such that the actual path taken does not matter. In this section, we shall give an example of an instance that violates this rule.

mathlib contains the following variant of the `function.has_smul'` that we saw earlier, which applies scalar multiplication pointwise between two families (or vectors) of elements[5].

```
instance function.has_smul'' (ι M α : Type*) [has_smul M α] : has_smul (ι → M) (ι → α) :=
{ smul := λ r v, (λ i, r i • v i) }
```

Mathematically this is a very reasonable operation, but in the context of typeclass diamonds we shall see it is not. In particular, consider a typeclass search for `has_smul (ι → M) (ι → ι → α)` in the presence of `has_smul M α`, as shown in fig. 1.3. Here, we find that one path gives `(f • g) i j = f i • g i j`, while the other gives `(f • g) i j = f j • g i j`, where the argument to `f` is different. We say these paths are not "propositionally equal" (as it can be proved in most cases[6] that they do not agree), and call this instance diamond "non-commuting".

Definitional equality

There is a reason that section 1.1.5 specifically refers to issues with *propositional* equality in typeclass resolution; dependent type theory leaves us with another important kind of equality, *definitional* equality, which holds only if things are true by construction.

The following example, which shows that a family of additive maps are a module under scaling by natural numbers, is vulnerable to issues around definitional equality.

```
example {ι A B} [add_comm_monoid A] [add_comm_monoid B] : module ℕ (ι → A →+ B) := by apply_instance
```

The possible typeclass-resolution paths available are shown in fig. 1.4.

From the viewpoint of *propositional* equality, we are safe from non-commuting paths; it can be proven that `∀ (M : Type*) [add_comm_monoid M], subsingleton (module ℕ M)` (that is that all ℕ-module structures are equal), and thus we can conclude the paths must be equal from the type of their endpoint alone. To a user of Lean, this means that they can be confident that the • carries the right mathematical meaning.

[5] In some sense, providing a one-sided "broadcasting" multiplication like that found in [1]; though only for when the right array has more dimensions than the left.

[6] The exception being when `subsingleton ι` (i.e. when there is only a single inhabitant of `ι`, and so `i = j`) or similar!

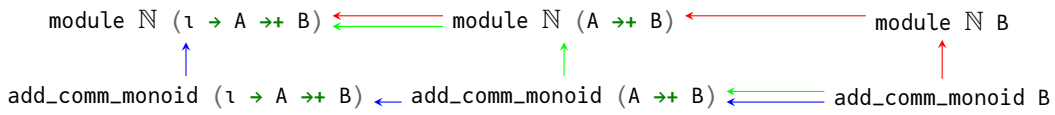


Figure 1.4: Compounding diamonds in typeclass search

Three possible paths to resolve `module ℕ (ι → A →+ B)`, with arrows showing implications. Diamonds are created by the choice between inheriting a module structure (horizontal edges), or deriving it from the additive structure (vertical edges).

When it comes to applying lemmas about `•`, it is not sufficient that `•` carry the right mathematical meaning; the `•` in the lemma statement needs to unify with the `•` in the target. In practice, this means that the instances found for each need to be *definitionally* equal, otherwise users are left with baffling error messages about how `n • x` does not match `n • x`.

In older versions of `mathlib`, the paths taken around the diamonds in fig. 1.4 resulted in instances that were propositionally equal, but not definitionally equal, as shown in fig. 1.5. The underlying reason was that the recursor `@nat.rec` for natural numbers (which underpins the `^[n]` notation in fig. 1.5) does not commute definitionally with lambda introduction `λ a,`; that is, the following example fails:

```
example {α β} (f₀ : α → β) (f : α → ℕ → β → β) (n : ℕ) :
  @nat.rec (λ n, α → β) (λ a, f₀ a) (λ n ih a, f a n (ih a)) n =
  λ a, @nat.rec (λ n, β) (f₀ a) (λ n ih, f a n ih) n :=
rfl
```

This meant that lemmas about the natural ℕ-action (blue path, fig. 1.4) such as `∑ x in s, c = s.card • c` would fail to match goals containing a derived ℕ-action (green and red paths, fig. 1.4). This was fixed in [mathlib#7084] by requiring the definition of `add_comm_monoid M` to include an implementation of the ℕ-module structure; namely, a new `nsmul` field, and a proof that it coincides propositionally with the naïve recursive implementation.

While mathematically it is bizarre to say "a commutative additive monoid has a zero, addition, *and a scalar-multiplication by naturals*, such that …", in Lean this is crucial to allow manual control of definitional equality such that the green and blue paths in fig. 1.4 can be made definitionally equal to the red path. This is analogous to the situation described in [3, section 4.1] for topologies associated with metric spaces, and follows the "forgetful inheritance" pattern described in [5]. Further discussion of this `nsmul` field can be found in [6, §7].

A similar place in which this comes up is when constructing `algebra ℕ S`, `algebra ℤ R`, and `algebra ℚ K` instances for a semiring, ring, or characteristic-zero division ring, respectively. In each case, the type of the instance is a subsingleton and so instance paths can be seen trivially to commute propositionally. The danger arises when constructing the `algebra_map` fields; the "obvious" way is to do so recursively, by recursing structurally on `n : ℕ/z : ℤ/q : ℚ` and setting `algebra_map _ _ (n + 1) = algebra_map _ _ n + 1`, etc. This approach not only fails on a very similar example to fig. 1.4, but also fails in the case when `S = ℕ/R = ℤ/K =`

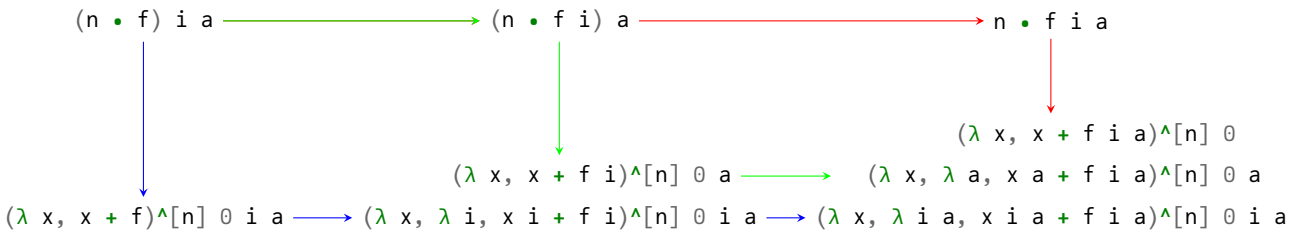


Figure 1.5: Non-commuting diamonds in repeated addition actions

Prior to [mathlib#7084], the expressions shown here are the ones found through the paths in fig. 1.4, where the `g^[n] x` operation is iterated function application, `g^[2] x = g (g x)`. The arrows mean "unfolds to" as they did in figs. 1.2 and 1.3.

$\mathbb{Q}$, as the identity function is certainly not equal by definition to such a recursive scheme. The solution in [mathlib#12182; mathlib#14894] was also similar; adding `of_nat`/`of_int`/`of_rat` field to the `semiring`/`ring`/`division_ring` typeclasses[7] and proof fields demonstrating that they are well-behaved.

This is not the last time we shall have to concern ourselves with definitional equality; we shall see the relevance of it again in ????.

1.1.6 Conjugation, via type synonyms

While section 1.1.5 gives some more involved examples of how derived actions can induce conflicting definitions, we can obtain conflicting actions much more simply. Consider for instance the ways in which a group can act on itself: both $g \cdot h = gh$ and $g \cdot h = ghg^{-1}$ are reasonable actions, but we clearly can't use the same notation for both without causing confusion[8], and so only the former (section 1.1.2) gets the privilege of the `has_smul G G` typeclass.

This is of no help to us if we really want to work with the latter conjugation action. In some cases we can avoid this by working with `mul_distrib_mul_action G H` for arbitrary groups `G` and `H` instead, which is a more abstract representation of actions (like conjugation) which distribute over multiplication. Even then, this abstraction only postpones the inevitable; it cannot be specialized to the concrete case of $g \cdot h = ghg^{-1}$ until we solve the original issue.

The solution used by mathlib is that of "type synonyms", which can be written as either of the following forms

[7] Or rather, suitable ancestors of these classes, for reasons related to non-associative algebras

[8] Lean may be happy for us to overload the notation in this way, but the humans writing proofs who see this notation in their goal are unlikely to be!

```
-- nominal types
@[deriving group]
def conj_act (G : Type*) : Type* := G

def of_conj_act : conj_act G ≃* G :=
  mul_equiv.refl _

def to_conj_act : G ≃* conj_act G :=
  of_conj_act.symm
```

```
-- one-field structures
structure conj_act (G : Type*) : Type* :=
  to_conj_act :: (of_conj_act : G)

instance [has_mul G] : has_mul (conj_act G) :=
{ mul := λ g h, to_conj_act (g.of_conj_act * h.of_conj_act) }

instance [Group G] : group (conj_act G) :=
function.injective.group to_conj_act
  sorry sorry sorry sorry sorry sorry sorry
```

where in both cases, we define a new type `conj_act G` which is a copy of `G` with the same group structure, and a pair of functions `to_conj_act` and `of_conj_act` to translate from `G` to `conj_act G` and back. The trade-offs between the two approaches are not particularly relevant to this section[9], and in this particular case mathlib opts for the left approach.

So far, the type synonym has achieved nothing; it behaves in exactly the same way as `G`! To give it purpose, we give it a special action on `G` that is the conjugation action

```
instance : has_smul (conj_act G) G :=
{ smul := λ cg h, of_conj_act cg * h * (of_conj_act cg)⁻¹ }
```

where we convert from `conj_act G` back to `G` before implementing the expected expression. This allows us to write the conjugation of `g` on `h` as `to_conj_act g • h`, and prove the various axioms of the stronger typeclasses in fig. 1.1a. The synonym is placed on the type doing the acting (`has_smul (conj_act G) G`) rather than the type being acted upon (`has_smul G (conj_act G)`) as this permits us to use both the multiplication and conjugation action simultaneously[10] (via different spellings).

The verbosity of this spelling might make it seem unappealing; if a spelling this long is acceptable, it would tempting to conclude we could have just used the spelling `conj g h` where `conj : G →* monoid.End G` is the appropriate morphism taken from the right column of table 1.1. The reason we avoid making this choice is that it would exclude the conjugation action from incorporation into the derived instances in section 1.1.3. One action in particular is of interest here; the pointwise action [*mathlib#8945*] (matching section 1.1.3) induced by elements `g : G` on a subgroup `S : subgroup H` when `mul_distrib_mul_action G H`. Combined with the conjugation action we just saw, this gives us the usual conjugation action of an element on a subgroup! The type synonym in this section was added in [mathlib#8627] by Chris Hughes, in response to review the author's review of Chris' previous approach in an early version of [mathlib#8592], which aimed to define only the conjugation action.

[9] They come down to the left version being easier to misuse (since Lean is willing to confuse the types `G` and `conj_act G` in some situations), and the right version being much more work to set up (since Lean is *not* ever willing to confuse the types and we must therefore rebuild the group structure from scratch, hence all the `sorry`s). If used correctly, the two approaches are in practice equivalent to the end user.

[10] Though in fact there is a third option, explored in section 1.1.9.

1.1.7 Right actions

The scalar action typeclass `has_smul` in mathlib is intended for left actions (those where $a(bX) = (ab)X$), which is apparent both in the definition of the `mul_smul` axiom of `mul_action`, and in the order in which the arguments appear in the notation. However, this does not mean that right actions (those where $(Xb)a = X(ba)$) are impossible.

The trick is to use another type synonym (section 1.1.6) from mathlib, `mul_opposite α`, which is built similarly to the pedagogical example in **??** and which reverses the multiplication order. With this in mind, we can require a right action by writing `[mul_action (mul_opposite M) α]`, which permits us to write `op a • X` as a messy spelling for a right action on `X` by `a`.

The author introduced mathlib's first right action in [*mathlib#7630*], via the instance

```
instance monoid.to_opposite_mul_action [monoid α] : mul_action (mul_opposite α) α :=
{ smul := λ c x, x * c.unop,
  one_smul := mul_one,
  mul_smul := λ x y r, (mul_assoc _ _ _).symm }

lemma op_smul_eq_mul [monoid α] {a b : α} : op a • b = b * a := rfl
```

which mirrors the left-multiplication action in section 1.1.2. Similar instances were introduced for the other stronger typeclasses in fig. 1.1a.

One big advantage of this design over introducing a new `right_mul_action` type is that the vast majority of the derived actions from section 1.1.3 come for free: as an example, `mul_action (mul_opposite α) (ι → α)` is found automatically and corresponds to the action `(op a • f) i = op a • f i = f i * a`,. This is precisely the action we want, if for example we want to multiply a family of quaternions (or indeed, any object with non-commutative multiplci by a constant on the right.

Bimodules

Often, we wish to consider simultaneous right and left actions, such as an `R-S`-bimodule , where `R` acts on the left of `M`, and `S` on the right. Crucially, these actions must be compatible; $(rm)s = r(ms)$. Mathematically, this looks like associativity, and so we might hope to capture it using the `smul_assoc` from `is_scalar_tower`. In actuality, to Lean the statement is `r • op s • m = op s • r • m`, and so this is commutativity! We can thus capture the structure of the bimodule `M` as follow:

```
variables [module R M] [module (mul_opposite S) M] [smul_comm_class R (mul_opposite S) M]
```

When `R` is commutative, we typically don't bother with talking about left and right actions, as they are usually equivalent[11]. However, it would be a bad idea to make `module R M` imply `module (mul_opposite R) M`, as even though the commutativity of `R` means we need not worry about

[11] and if they weren't, mathlib would typically force the use of a type synonym (section 1.1.6) to declare the less-canonical action

propositionally equal typeclass diamonds arising, we are almost guaranteed to run into definitional ones. The solution was to introduce an `is_central_scalar R m` typeclass [*mathlib#10543*], which instead of producing a new instance and risking diamond issues, simply asserts that two existing instances interact in the desired way; namely that `op_smul_eq_smul : op r • m = r • m`.

Adding this typeclass is just the tip of the iceberg; the real work is to saturate mathlib with instances of this typeclass. [*mathlib#10543*] claimed the low-hanging fruit, providing instances for commutative monoids, product types (`M × N`, `Π i, M i`), finitely supported functions (`ι →₀ M`, `Π₀ i : ι, M i`), `ULift M`, polynomials and their generalizations (`monoid_algebra R M`, `add_monoid_algebra R M`, `R[X]`, `mv_polynomial σ R`), `matrix m n M`, morphisms (`M →+ N`, `M →ₗ[R] N`), complex numbers, and pointwise instances[12] (`set M`, `submonoid M`, `add_submonoid M`, `subgroup M`, `add_subgroup M`, `subsemiring M`, `subring M`, `submodule M`). Further contributions [*mathlib#10720*; *mathlib#11291*; *mathlib#11297*; *mathlib#12248*; *mathlib#12272*; *mathlib#12434*; *mathlib#13710*; *mathlib#15359*; mathlib#18682] brought the total up to more than 60 `is_central_scalar` instances across mathlib.

Interaction with `algebra`

The addition of all these `is_central_scalar` instances was not simply motivated by completeness: it was a prerequisite for solving a larger issue, the fact that we want every `algebra R A` to automatically be a left- and right- `R`-module.

We already saw in section 1.1.4 that mathlib knows that an `algebra R A` implies a left `module R A` structure, and forces it to agree with left-multiplication. To make it also imply a right-module (`module (mul_opposite R) A`) structure, we need it to carry an additional `to_has_op_smul : has_smul (mul_opposite R) A` field (the right action), and a proof that it coincides with *right*-multiplication, `op_smul_def' x r : op r • x = x * to_fun r`. Such a refactor is a rather herculean task; there are at least 130 instances of `algebra` in mathlib[13], and adding fields to `algebra` requires every instance to provide values for these fields. To make matters worse, mathlib is a moving (and growing) target, and once you think you're almost done, you merge in the latest changes from other contributors and have even more `algebra` instances to fix!

The exercise of adding a large number of `is_central_scalar` instances was a means to slowly close the gap; the act of adding these results often involved adding associated `has_smul (mul_opposite R) A` instances, getting half the work out of the way. Additionally, the `op_smul_eq_smul` lemma provided by these instances paves a quick path to proving `op_smul_def'`. Getting these intermediate results merged into mathlib before the full project was a way of offloading the work of keeping up: while results are only in your local modifications, it falls primarily on you to deal with conflicts and proof breakages arising from others' changes; once results are in mathlib, the responsibility transfers to the community at large. As it turned out, this approach of

[12] elementwise actions on the elements of a set or sub-object
[13] According to the generated documentation.

contributing early and often had further benefits; the author's prototype of a complete refactor in [*mathlib#10716*] was "caught in the porting tide", which is to say that the translation ("port") from Lean 3 to Lean 4 (which starting from the simplest files and rose up through the import hierarchy) caught up with the files it modified, forcing it to be abandoned. The same fate did not befall the `is_central_scalar` instances, as these were already merged before the port began.

As discussed by the author in mathlib4/issues/7152, this refactor introduces further complications almost exactly analogous to the ones we saw in fig. 1.4. If every algebra implies a right action, then we need $\mathbb{N}$-algebras to imply right-$\mathbb{N}$-actions. We saw in section 1.1.5 that to make this work without definitional typeclass diamonds for left-actions, we needed to add an `nsmul` field to additive monoids (and `zsmul` to additive groups); for right-actions, we need to do the same for `op_nsmul` and `op_zsmul`, making the mathlib `add_comm_group` even further from the expected mathematical definition!

Other compatibility concerns

In section 1.1.3, we saw how when working even just with left-actions, needs arise for compatibility typeclasses like `is_scalar_tower` and `smul_comm_class`. In section 1.1.7, we saw that the latter can be repurposed to provide a compatibility between left and right actions. However, there are other common interactions of left and right actions for which mathlib has no compatibility typeclass.

Introducing briefly for clarity the notation[14] `a •> b` for `a • b` and `a <• b` for `op a • b`, there is no typeclass in mathlib capable of expressing `(a <• b) •> c = a •> (b •> c)`. Some examples of when this situation arises are `[monoid M] (a b c : M)` (all three variables belong to the same non-commutative monoid), `[monoid M] (a c : M) (S : submonoid M) (b : S)` (the second belongs to a submonoid of the monoid containing the other two), and `[monoid M] (a b : M) (c : ι → M)` (the third variable is a coordinate vector).

On functions, through their domains

In section 1.1.6, we remarked that sometime the left action we want is not the one by left-multiplication, and that to resolve this we must introduce a type synonym. The same issue arises for right actions. A particularly typical example is the right action on function types (or in general, morphisms) that acts through their domain; the action where for $f : G \to G$, $g, h : G$, we define the right action fg such that $(fg)(h) = f(gh)$. This is problematic, because the derived action in section 1.1.3 gives us the action where $(fg)(h) = f(h)g$, which when combined with the instance we wanted produces a non-commuting instance diamond.

Type synonyms again provide a solution here; in [mathlib4#5368], after some discusion with the author, Yury Kudryashov introduces a `DomMulAct M` type synonym which induces precisely this $(fg)(h) = f(gh)$ action, which Lean characterizes as:

[14] Which is proposed for inclusion in mathlib in [*mathlib4#8909*].

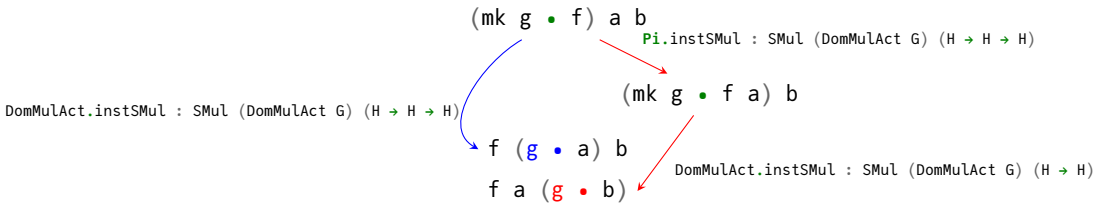


Figure 1.6: A non-commuting diamond caused by `DomMulAct`

Here we have an action `SMul G H`, variables `g : G`, `a b : H` and `f : H → H → H`. The ambiguity arises through choosing which argument of `f` to act upon.

```
example [SMul M α] (c : M) (f : α → β) (a : α) : (mk c • f) a = f (c • a) := rfl
```

which for our example specializes to

```
example [Group G] (g : G) (f : H → H) (h : H) : (mk g • f) a = f (g * h) := rfl
```

With this synonym, Lean cannot take a wrong turn towards using the instance in section 1.1.3 with $(fg)(h) = f(h)g$, as there is no action of `SMul (DomMulAct G) G` available.

Unfortunately, this solution only postpones the diamonds to one step further down the road. When faced with an action on a function taking two arguments, Lean now faces an ambiguity about which of the two domains the action should act upon (thanks to the action in section 1.1.3), as shown in fig. 1.6. This isn't quite as bad as the diamond in fig. 1.3, as it only impacts users of `DomMulAct`; but it is indicative that type synonyms are not a silver bullet.

We could resolve this by removing the `Pi.instSMul` instance in section 1.1.3 that formed one of the offending edges, but this would be prevent writing `r • ![x, y, z]` to scale a vector of coefficients. A compromise is available through the use of even more type synonyms; we could demote the instance for actions through the codomain to a `CodMulAct` synonym, and then the two actions in fig. 1.6 would be spelt as `(DomMulAct.mk g • f) a b = f (g * a) b` and `(CodMulAct.mk (DomMulAct.mk g) • f) a b = f a (g * b)`, and `r • ![x, y, z]` would need to be written as `CodMulAct.mk r • ![x, y, z]`. The verbosity could be somewhat reduced by introducing some shorthand notation, which for these three examples could resemble `g •D f`, `g •CD f`, and `r •C ![x, y, z]`.

Such a compromise would involve a substantial refactor to mathlib, and for now the costs of the instance diamond in fig. 1.6 do not in the author's opinion outweigh the effort involved in *performing* the refactor, let alone the negative impact of the increased verbosity.

1.1.8 Lean 4's new `HMul` typeclass

Lean 4 generalizes the meaning of the `*` syntax, allowing it to be used to multiply elements of different types. This is done by means of a new `HMul α β γ` typeclass that provides the

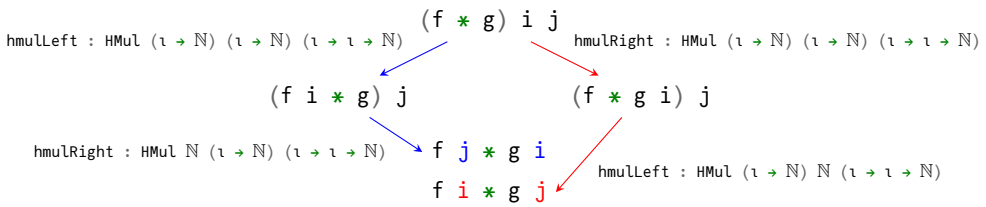


Figure 1.7: A non-commuting diamond caused by `HMul` for `f g : ι → ℕ`

operator `*` through the function `HMul.hMul : α → β → γ`, where the `H` stands for "heterogenous". This generalization is very useful for multiplication of matrices (which `mathlib` switched to using in [*mathlib4#6487*]), as here multiplication is naturally homogenous in the dimensions of the matrices; we have `HMul.hMul : Matrix l m R → Matrix m n R → Matrix l n R`.

At first sight, the introduction of `HMul` would appear to make `has_scalar.smul : α → β → β` redundant, as it falls out as a special case. Similarly, we could even see `HMul` as the answer to the right actions in section 1.1.7, since we can also recover `β → α → β` as a special case. Even `smul_comm_class` and `is_scalar_tower` could likely be subsumed by a typeclass for generalized heterogenous commutativity or associativity, respectively.

Unfortunately, for our use-case the flexibility of `HMul` is also its downfall; attempting to build basic left- and right- actions for "vectors" (as in section 1.1.3) using it almost immediately leads to instance diamonds in the style of section 1.1.5. The two basic instances in question are:

```
-- HMul generalizes to families on the right
instance hmulRight [inst : HMul α β γ] : HMul α (ι → β) (ι → γ) where
  hMul a b := fun i => a * b i

-- HMul generalizes to families on the left
instance hmulLeft [inst : HMul α β γ] : HMul (ι → α) β (ι → γ) where
  hMul a b := fun i => a i * b
```

and the diamond is formed when querying for `HMul (ι → ℕ) (ι → ℕ) (ι → ι → ℕ)`, as shown in fig. 1.7. The following code can be used to verify the diagram is correct.

```
variable (ι)
abbrev blue : HMul (ι → ℕ) (ι → ℕ) (ι → ι → ℕ) := hmulLeft (inst := hmulRight)
abbrev red  : HMul (ι → ℕ) (ι → ℕ) (ι → ι → ℕ) := hmulRight (inst := hmulLeft)

-- the two paths give conflicting results, swapping the placements of `i` and `j`
example (f g : ι → ℕ) (i j : ι) : letI := blue ι; (f * g) i j = f i * g j := rfl
example (f g : ι → ℕ) (i j : ι) : letI := red ι;  (f * g) i j = f j * g i := rfl
```

Strictly speaking, our problem is not that `HMul` is too general, but that `hmulLeft` and `hmulRight` are. We can resolve this by eliminating `γ`, and using repeated type variables to distinguish right and left actions:

```
-- right-HMul generalizes to families on the right
instance hmulRight [inst : HMul α β β] : HMul α (ι → β) (ι → β) where
```

```
  hMul a b := fun i => a * b i

-- left-HMul generalizes to families on the left
instance hmulLeft [inst : HMul α β α] : HMul (ι → α) β (ι → α) where
  hMul a b := fun i => a i * b
```

At this point though, we may as well have defined `LeftSMul α β := HMul α β β` and `RightSMul α β := HMul α β α`, an approach which is not all that different from the current mathlib approach which is effectively using `LeftSMul α β := SMul α β` and `RightSMul α β := SMul (MulOpposite α) β`. The only real difference is the choice of symbol, `*` vs `•`.

1.1.9 Alternatives to type synonyms

In sections 1.1.6 and 1.1.7 we explored how mathlib uses `DomMulAct` and `ConjAct` synonyms that wraps the type doing the acting, in order to distinguish these action from other "more canonical" actions. However, this comes at the cost of introducing some annoying boilerplace in the form of `DomMulAct.mk` and `ConjAct.toConjAct`, which we use to juggle between `DomMulAct G`, `ConjAct G`, and `G`. While section 1.1.7 suggests that the pain of this boilerplate can be reduced with clever notation, it still ends up being something that has to be manipulated within proofs. This section briefly outlines an alternative design that has no prior use in mathlib.

A possible redesign of the `SMul` typeclass could be

```
def SMul.Discr := Type
class SMul (M : Type*) (α : Type*) (discr : SMul.Discr) where
  smul : M → α → α
notation3:72 a:72 " •[" discr "] " b:72 => @SMul.smul _ _ discr _ a b
```

where the `discr` parameter acts as a discriminator to implement "tag dispatching" (to borrow the term from C++) and contains information relevant only at typeclass-search-time about *which* action to use; information that was previously tracked by attaching it to type synonyms wrapping `M` and `α`. The `discr` parameter would be copied to all the other typeclasses in fig. 1.1a.

The actions in sections 1.1.2, 1.1.3 and 1.1.7 can then respectively be written

```
inductive SMul.Discr.leftMul : SMul.Discr
instance (M : Type*) [Mul M] : SMul M M .leftMul where
  smul m n := m * n

inductive SMul.Discr.domAct (_ : SMul.Discr) : SMul.Discr
instance (ι α M : Type*) (discr) [SMul α ι discr] : SMul α (ι → M) discr.domAct where
  smul a f := fun i => f (a •[discr] i)

inductive SMul.Discr.codAct (_ : SMul.Discr) : SMul.Discr
instance (ι α M : Type*) (discr) [SMul α M discr] : SMul α (ι → M) discr.codAct where
  smul a f := fun i => a •[discr] (f i)
```

where the `domAct` and `codAct` discriminators are parameterized to record the discriminator they inherit from.

With this infrastructure in place, the diamonds in fig. 1.6 are avoided by forcing the user to spell which action they want:

```
variable {G H} (discr) [SMul G H discr] (g : G) (f : H → H → H) (a b : H)
example : (g •[discr.domAct] f) a b = f (g •[discr] a) b := rfl
example : (g •[discr.domAct.codAct] f) a b = f a (g •[discr] b) := rfl
```

This strategy is not without its downsides. It makes cases where the typeclass diamonds do commute (such as fig. 1.2) more awkward to work with, requiring an additional compatibility typeclass that states that two discriminators describe the same action; Additionally, bundled linear maps would now need to take two extra arguments to specify the discriminator for their source and domain. Determining whether these tradeoffs are acceptable would require an attempt at refactoring large pieces of mathlib, which the author will leave to a particularly interested reader.

References

[1] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array Programming with NumPy". June 17, 2020. arXiv: 2006.10256 [cs, stat]. URL: http://arxiv.org/abs/2006.10256 (visited on 06/23/2020) (cit. on pp. 1, 9).

[2] Eric Wieser. "Scalar Actions in Lean's Mathlib". In: *CICM 2021*. Timisoara, Romania, Aug. 10, 2021. URL: http://arxiv.org/abs/2108.10700 (visited on 05/25/2022) (cit. on p. 1).

[3] The mathlib Community. "The Lean Mathematical Library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. POPL '20: 47th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. New Orleans LA USA: ACM, Jan. 20, 2020, pp. 367–381. ISBN: 978-1-4503-7097-4. DOI: 10.1145/3372885.3373824. URL: https://dl.acm.org/doi/10.1145/3372885.3373824 (visited on 08/21/2020) (cit. on pp. 2, 10).

[4] Anne Baanen, Sander R. Dahmen, Ashvni Narayanan, and Filippo A. E. Nuccio Mortarino Majno Di Capriglio. "A Formalization of Dedekind Domains and Class Groups of Global Fields". In: *Journal of Automated Reasoning* 66.4 (Nov. 2022), pp. 611–637. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-022-09644-0. URL: https://link.springer.com/10.1007/s10817-022-09644-0 (visited on 12/14/2023) (cit. on p. 7).

[5] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. "Competing Inheritance Paths in Dependent Type Theory: A Case Study in Functional Analysis". In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Vol. 12167. Cham: Springer International Publishing, 2020, pp. 3–20. ISBN: 978-3-030-51053-4 978-3-030-51054-1. DOI: 10.1007/978-3-030-51054-1_1. URL: http://link.springer.com/10.1007/978-3-030-51054-1_1 (visited on 07/06/2022) (cit. on p. 10).

[6] Anne Baanen. "Use and Abuse of Instance Parameters in the Lean Mathematical Library". In: *ITP 2022*. Haifa, Israel, May 2, 2022. URL: http://arxiv.org/abs/2202.01629 (visited on 05/25/2022) (cit. on p. 10).

Github references

This section contain references to issues and pull requests on GitHub, which is the main place that development of community-owned software such as Lean, mathlib, and Numba occurs. While not peer-reviewed in a conventional sense, the pull requests (i.e. code contributions) in this section have been reviewed by other contributors, and are only "merged" if they pass this review process. This process can be anything from an instant approval in seconds to a long discussion iterating on code changes that spans months! *Italicised keys* indicate self-references.

[*mathlib4#8395*] Eric Wieser. *feat(Algebra/Module/Hom): `AddMonoid.End` application forms a `Module`*. Nov. 2023 (cit. on p. 3).

[*mathlib#8724*] Eric Wieser. *feat(group_theory/group_action): generalize `mul_action.function_End` to other endomorphisms*. Aug. 2021 (cit. on p. 3).

[*mathlib4#8396*] Eric Wieser. *feat(Algebra/GroupRingAction/Basic): `RingHom` application forms a `MulDistribMulAction`*. Nov. 2023 (cit. on p. 3).

[*mathlib#8968*] Eric Wieser. *feat(algebra/module/basic): add `module.to_add_monoid_End`*. Sept. 2021 (cit. on p. 3).

[mathlib#997] Johan Commelin. *feat(algebra/pointwise): More lemmas on pointwise multiplication*. May 2019 (cit. on p. 5).

[mathlib#12865] Yaël Dillies. *feat(data/finset/pointwise): Missing operations*. Mar. 2022 (cit. on p. 5).

[*mathlib#6891*] Eric Wieser. *feat(algebra/module/hom): Add missing smul instances on `add_monoid_hom`*. Mar. 2021 (cit. on p. 5).

[mathlib#4784] Yury G. Kudryashov. *refactor(data/polynomial): use `linear_map` for `monomial`, review `degree`*. Oct. 2020 (cit. on pp. 5, 6).

[*mathlib#7664*] Eric Wieser. *feat(data/polynomial): generalize `polynomial.has_scalar` to require only `distrib_mul_action` instead of `semimodule`*. May 2021 (cit. on p. 6).

[*mathlib#4365*] Eric Wieser. *feat(data/monoid_algebra): Allow R k in semimodule R (`monoid_algebra` k G)*. Oct. 2020 (cit. on p. 6).

[mathlib#284] Scott Morrison. *generalise finsupp.to_module*. Aug. 2018 (cit. on p. 6).

[*mathlib#6533*] Eric Wieser. *feat(data/mv_polynomial/basic): a polynomial ring over an R-algebra is also an R-algebra*. Mar. 2021 (cit. on p. 6).

[mathlib#4770] Yury G. Kudryashov. *chore(group_theory/group_action): introduce* `smul_comm_class`. Oct. 2020 (cit. on p. 7).

[*mathlib#6534*] Eric Wieser. *feat(data/finsupp, algebra/monoid_algebra): add* `is_scalar_tower` *and* `smul_comm_class`. Mar. 2021 (cit. on p. 7).

[*mathlib#6614*] Eric Wieser. *feat(data/dfinsupp): add* `is_scalar_tower` *and* `smul_comm_class`. Mar. 2021 (cit. on p. 7).

[*mathlib#8965*] Eric Wieser. *chore(field_theory/fixed): reuse existing* `mul_semiring_action.to_alg_hom` *by providing* `smul_comm_class`. Sept. 2021 (cit. on p. 7).

[*mathlib#15876*] Eric Wieser. *feat(group_theory/group_action/conj_act):* `smul_comm_class` *instances*. Aug. 2022 (cit. on p. 7).

[*mathlib#10262*] Eric Wieser. *feat(linear_algebra/pi_tensor_prod): generalize actions and provide missing* `smul_comm_class` *and* `is_scalar_tower` *instance*. Nov. 2021 (cit. on p. 7).

[*mathlib#6542*] Eric Wieser. *feat(data/mv_polynomial/basic): add* `is_scalar_tower` *and* `smul_comm_class` *instances*. Mar. 2021 (cit. on p. 7).

[*mathlib#6592*] Eric Wieser. *chore(data/polynomial/basic): add missing* `is_scalar_tower` *and* `smul_comm_class` *instances*. Mar. 2021 (cit. on p. 7).

[*mathlib#6139*] Eric Wieser. *chore(algebra/module/pi): add missing* `smul_comm_class` *instances*. Feb. 2021 (cit. on p. 7).

[*mathlib#5205*] Eric Wieser. *feat(algebra/module/basic): Add* `smul_comm_class` *instances*. Dec. 2020 (cit. on p. 7).

[*mathlib#5369*] Eric Wieser. *feat(algebra/module/basic): Add symmetric* `smul_comm_class` *instances for int and nat*. Dec. 2020 (cit. on p. 7).

[*mathlib#5509*] Eric Wieser. *fix(algebra/module/basic): Do not attach the* $\mathbb{N}$ *and* `is_scalar_tower` *and* `smul_comm_class` *instances to a specific definition of smul*. Dec. 2020 (cit. on p. 7).

[*mathlib#13174*] Eric Wieser. *fix(algebra/module/basic,group_theory/group_action/defs): generalize nat and int* `smul_comm_class` *instances*. Apr. 2022 (cit. on p. 7).

[mathlib#7932] Oliver Nash. *feat(algebra/monoid_algebra): adjointness for the functor* `G ↦ monoid_algebra k G` *when* `G` *carries only* `has_mul`. June 2021 (cit. on p. 8).

[mathlib#7416] Kexing Ying. *feat(linear_algebra/quadratic_form): Complex version of Sylvester's law of inertia*. Apr. 2021 (cit. on p. 8).

[mathlib#7084] sgouezel. *refactor(*): kill nat multiplication diamonds*. Apr. 2021 (cit. on pp. 10, 11).

[mathlib#12182] Gabriel Ebner. *feat:* `add_monoid_with_one`, `add_group_with_one`. Feb. 2022 (cit. on p. 11).

[mathlib#14894] Anne Baanen. *chore({algebra,data/rat}): use forgetful inheritance for* `algebra_rat`. June 2022 (cit. on p. 11).

[*mathlib#8945*] Eric Wieser. *feat(group_theory/sub{monoid,group}): pointwise actions on* `add_sub{monoid,group}s` *and* `sub{monoid,group,module,semiring,ring,algebra}s`. Sept. 2021 (cit. on p. 12).

[mathlib#8627] Chris Hughes. *feat(group_theory/group_action/conj_act): conjugation action of groups*. Aug. 2021 (cit. on p. 12).

[mathlib#8592] Chris Hughes. *feat(group_theory/group_action/subgroup): Conjugation action on subgroups of a group*. Aug. 2021 (cit. on p. 12).

[*mathlib#7630*] Eric Wieser. *feat(algebra/opposites): add* `has_scalar (opposite α) α` *instances*. May 2021 (cit. on p. 13).

[*mathlib#10543*] Eric Wieser. *feat(group_theory/group_action/defs): add a typeclass to show that an action is central (aka symmetric)*. Nov. 2021 (cit. on p. 14).

[*mathlib#10720*] Eric Wieser. *chore(algebra/module/submodule): missing* `is_central_scalar` *instance*. Dec. 2021 (cit. on p. 14).

[*mathlib#11291*] Eric Wieser. *chore(topology/algebra/module/basic): add* `continuous_linear_map.is_central_scalar`. Jan. 2022 (cit. on p. 14).

[*mathlib#11297*] Eric Wieser. *feat(algebra,linear_algebra,ring_theory): more* `is_central_scalar` *instances*. Jan. 2022 (cit. on p. 14).

[*mathlib#12248*] Eric Wieser. *feat(measure_theory/function/ae_eq_fun): generalize scalar actions*. Feb. 2022 (cit. on p. 14).

[*mathlib#12272*] Eric Wieser. *chore(topology/continuous_function/bounded): add an* `is_central_scalar` *instance*. Feb. 2022 (cit. on p. 14).

[*mathlib#12434*] Eric Wieser. *chore(topology/algebra/uniform_mul_action): add* `has_uniform_continuous_const_smul.op`. Mar. 2022 (cit. on p. 14).

[*mathlib#13710*] Eric Wieser. *chore(topology/continuous_function/zero_at_infty): add* `is_central_scalar` *instance*. Apr. 2022 (cit. on p. 14).

[*mathlib#15359*] Eric Wieser. *chore(linear_algebra/alternating): add an* `is_central_scalar` *instance*. July 2022 (cit. on p. 14).

[mathlib#18682] Yaël Dillies. *feat(data/finset/pointwise):* `a • (s ∩ t) = a • s ∩ a • t`. Mar. 2023 (cit. on p. 14).

[*mathlib#10716*] Eric Wieser. *feat(algebra/algebra/basic): Algebras are bimodules*. Dec. 2021 (cit. on p. 15).

[*mathlib4#8909*] Eric Wieser. *feat(GroupTheory/GroupAction/Opposite): add notation for right and left actions*. Dec. 2023 (cit. on p. 15).

[mathlib4#5368] Yury G. Kudryashov. *feat: define a type synonym for right action on the domain of a function*. June 2023 (cit. on p. 15).

[*mathlib4#6487*] Eric Wieser. *refactor(Data/Matrix): Eliminate · notation in favor of* `HMul`. Aug. 2023 (cit. on p. 17).