UMBC

Chemical Biochemical and Environmental Engineering

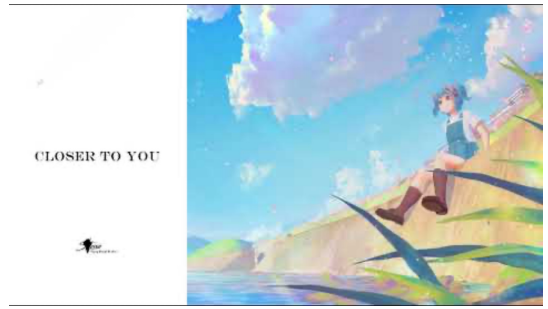
LEAN THEOREM PROVER

Lean for Scientists and Engineers

Tyler R. Josephson

AI & Theory-Oriented Molecular Science (ATOMS) Lab

University of Maryland, Baltimore County



Closer to You
Stessie

Twitter: @trjosephson
Email: tjo@umbc.edu

Lean for Scientists and Engineers 2024

1. Logic and proofs for scientists and engineers
    1. Introduction to theorem proving
    2. Writing proofs in Lean
    3. Formalizing derivations in science and engineering
2. Functional programming in Lean 4
    1. Functional vs. imperative programming
    2. Numerical vs. symbolic mathematics
    3. Writing executable programs in Lean
3. Provably-correct programs for scientific computing

Schedule (tentative)

Logic and proofs for scientists and engineers
Functional programming in Lean 4
Provably-correct programs for scientific computing

July 9, 2024	Introduction to Lean and proofs
July 10, 2024	Equalities and inequalities
July 16, 2024	Proofs with structure
July 17, 2024	Proofs with structure II
July 23, 2024	Proofs about functions; types
July 24, 2024	Calculus-based-proofs
July 30-31, 2024	Prof. Josephson traveling
August 6, 2024	Functions, definitions, structures, recursion
August 8, 2024	Polymorphic functions for floats and reals, compiling Lean to C
August 13, 2024	Input / output, lists, arrays, and indexing
August 14, 2024	Lists, arrays, indexing, and matrices
August 20, 2024	LeanMD & BET Analysis in Lean
August 21, 2024	SciLean tutorial, by Tomáš Skřivan

Content inspired by:
Mechanics of Proof, by Heather Macbeth
Functional Programming in Lean, by David Christiansen

Guest instructor: Tomáš Skřivan

Schedule for today

1. Recap Lectures 1-6
    1. Especially functions
2. Functional vs. imperative programming
3. Recursion basics
    1. Factorial
    2. Head vs. tail
    3. Summation
4. The halting problem
5. Structures

Extra recommended resource:
https://busy-beavers.tigyog.app/proofs-about-programs

Schedule for Wednesday

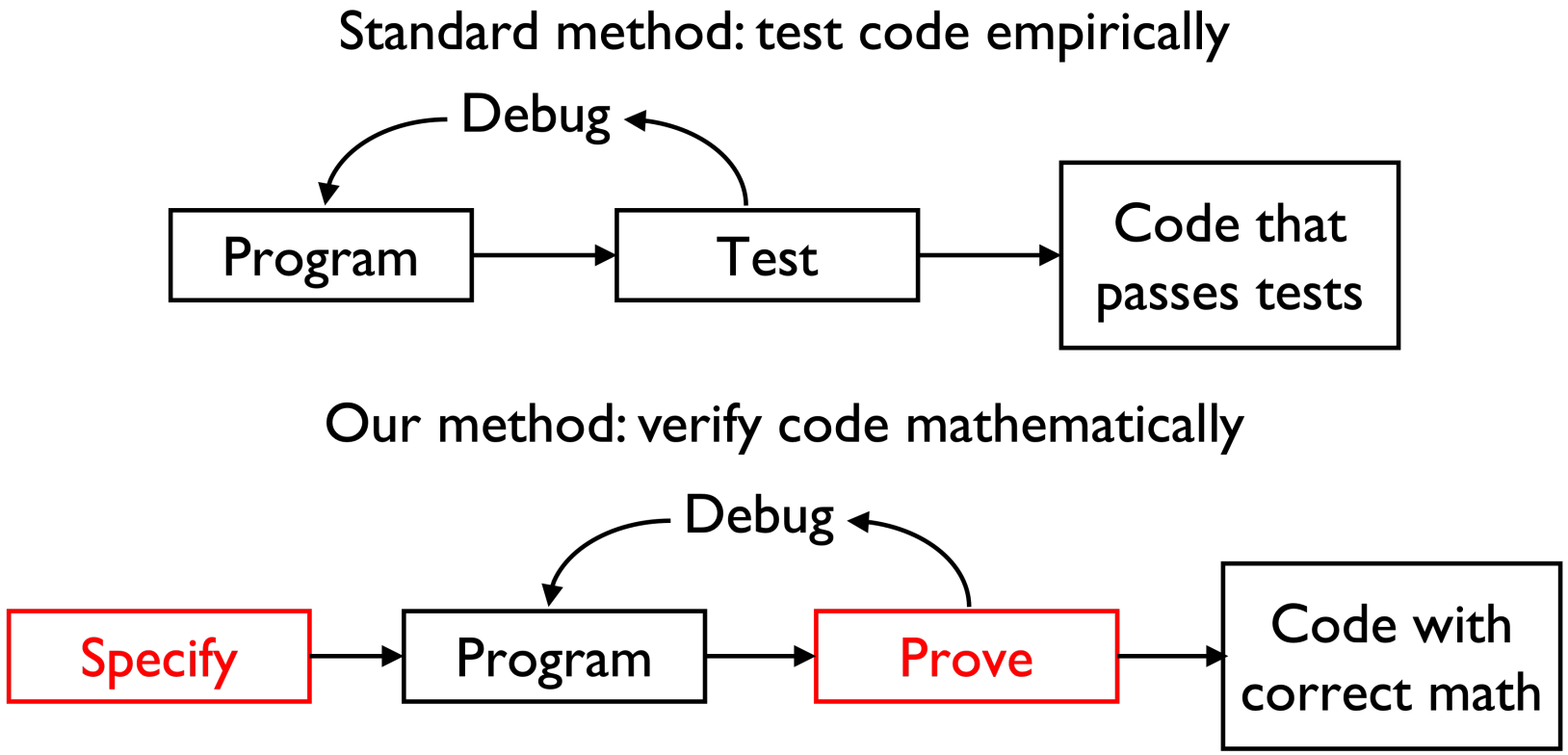
- Recap Lecture 7
- Polymorphic functions
- Lists and arrays
- Recursion over lists

START TIME 2:00 ET, not 1:00 ET

Slide from Lecture 1

A vision for bug-free scientific computing

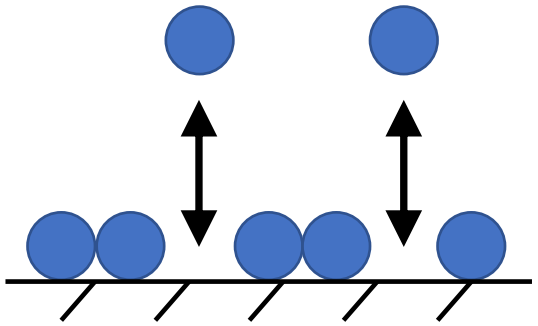
Selsam, Liang, Dill, "Developing Bug-Free Machine Learning Systems with Formal Mathematics," ICML 2017.



Errors in scientific computing software

Category of error	Example	Intervention	Lean
Syntax	Not closing parentheses	Editor	Editor
Runtime	Accessing element in list that doesn't exist	Run the program, program gives error message	Editor
Semantic	Missing a minus sign, transposing tensor indices	Human inspection of the code; test-driven development; observing anomalous behavior	Editor
Floating point / Round off	Subtracting small values from large values	Checking energy conservation	

Derivations in science are math proofs



Langmuir Adsorption

Langmuir, *JACS*, 1918

Proposition

5 premises — imply → conjecture

Site balance: $S_0 = S + S_a$

Adsorption rate model: $r_{ads} = k_{ads} \cdot p \cdot S$

Desorption rate model: $r_{des} = k_{des} \cdot S_a$

Equilibrium assumption: $r_{ads} = r_{des}$

Mass balance $q = S_a$

$$q = \frac{S_0 K_{eq} p}{1 + K_{eq} p}$$

Theorem

Proposition is TRUE

Proof

Derivation using algebraic manipulations (substitution, cancelling terms, etc.)

✓ _____

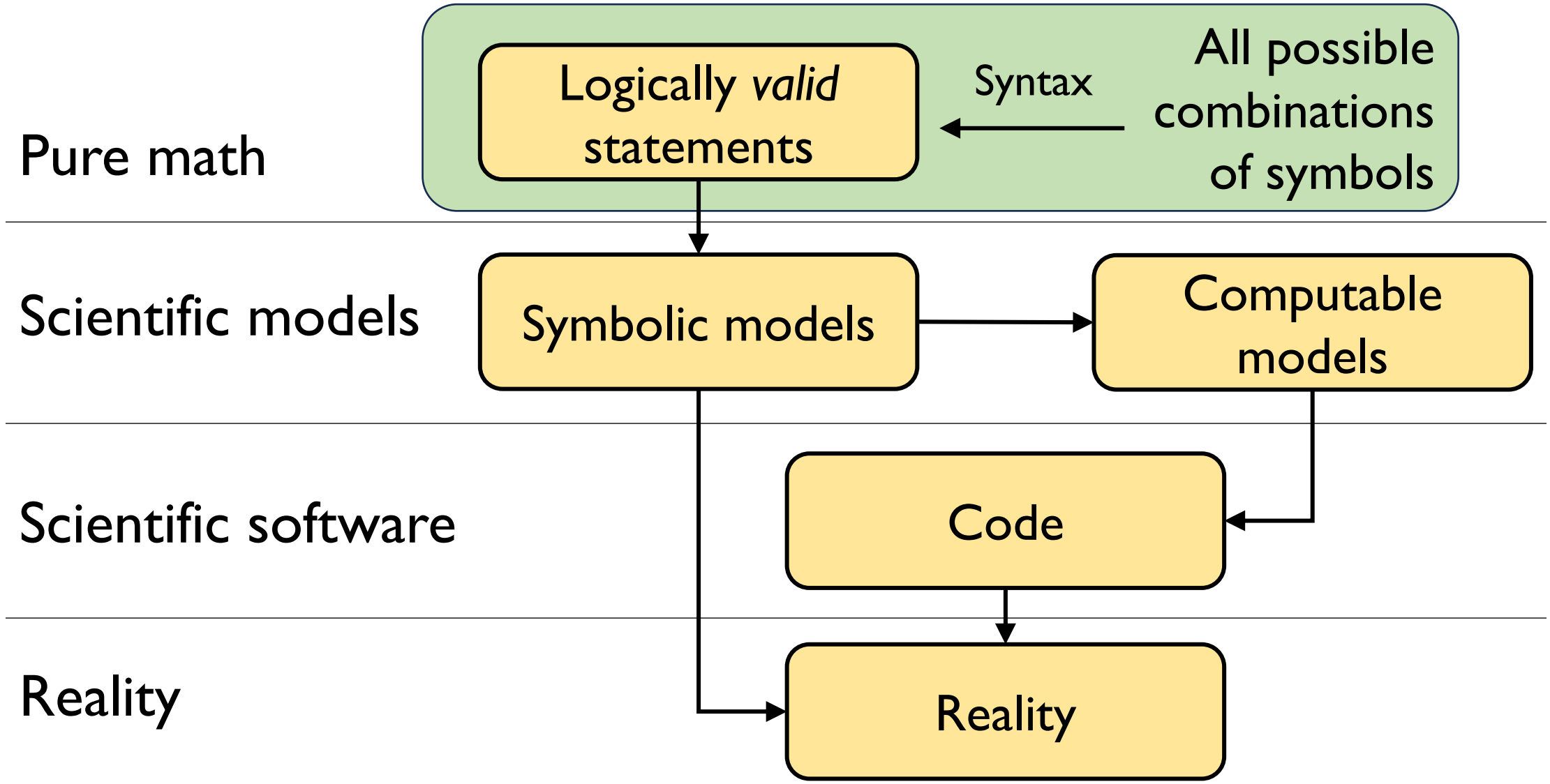
✓ _____

✓ _____

✓ _____

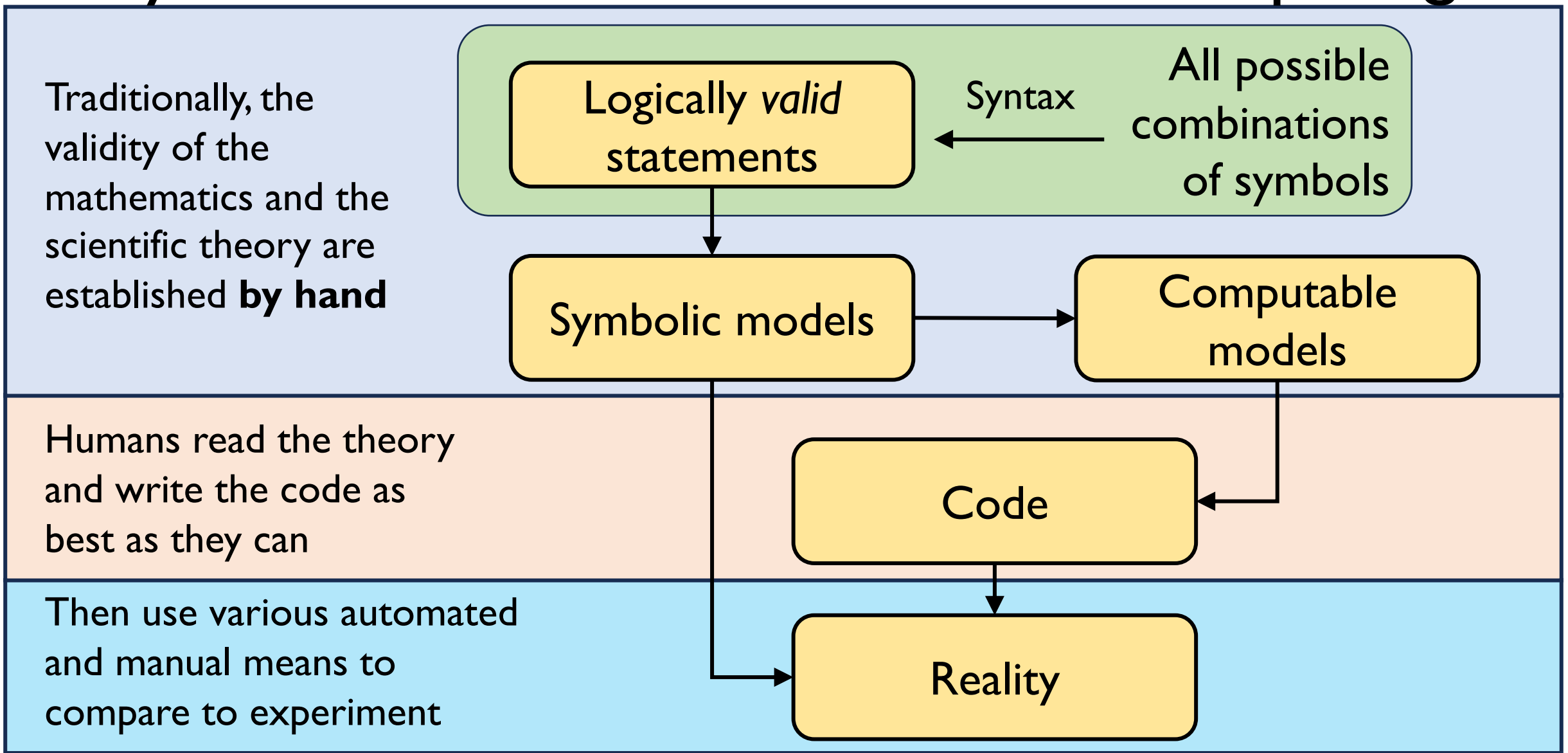
Slide from Lecture 3

Syntax and semantics in scientific computing



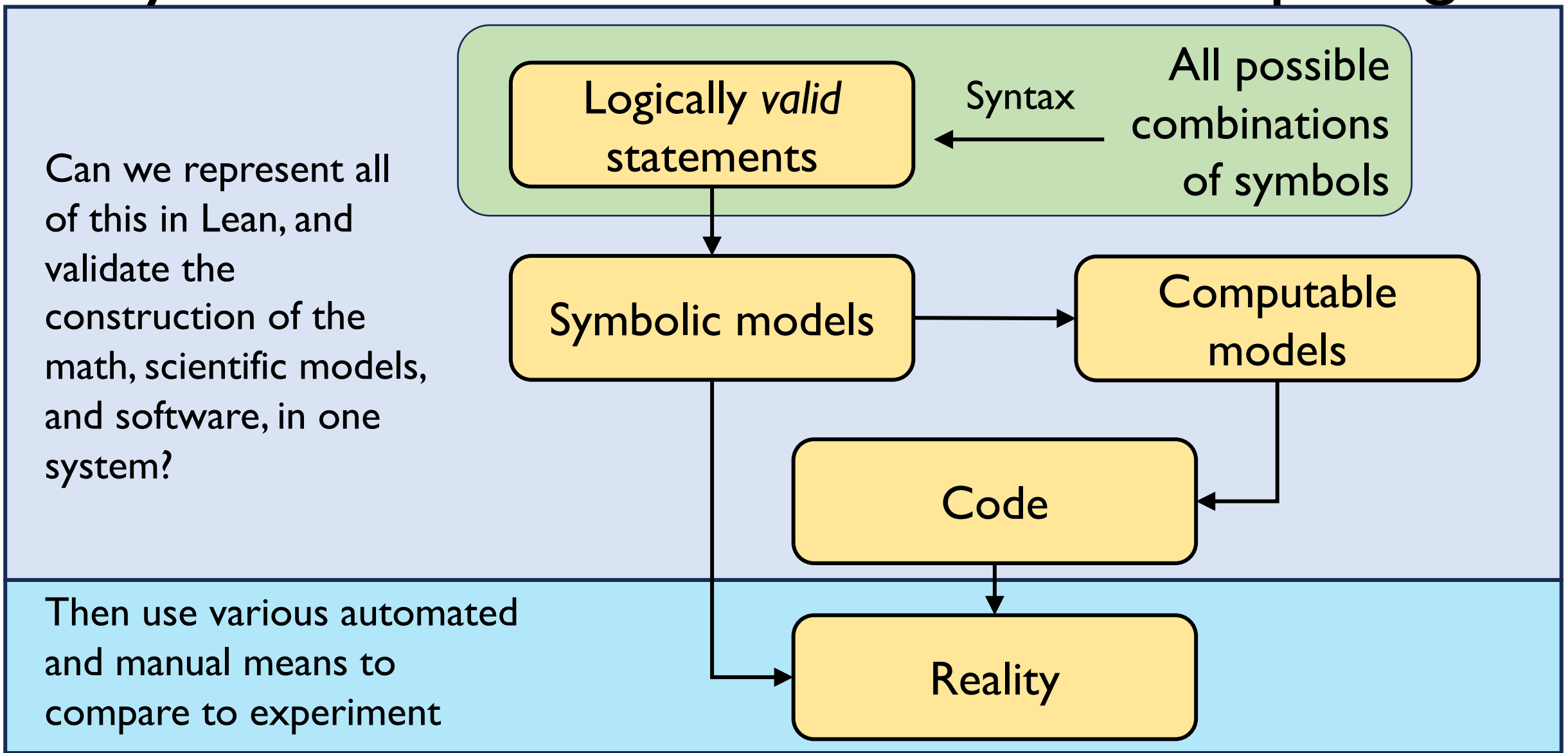
Slide from Lecture 3

Syntax and semantics in scientific computing



Slide from Lecture 3

Syntax and semantics in scientific computing



Polymorphic functions to bridge floats and reals

Adsorption data → Filter data to focus on "BET regime" → Linearize the raw data → Perform linear regression → Fitted coefficients

Floating point numbers

Polymorphic functions

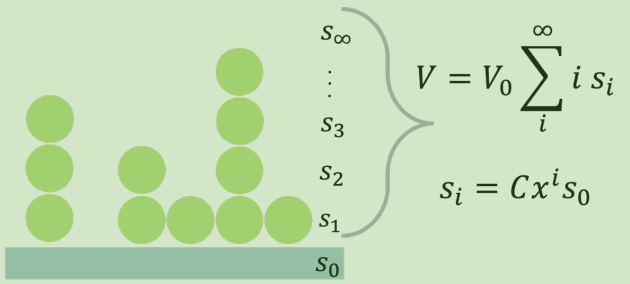
Real numbers

$\mathbb{R}$

Formal proof of BET Theory

$$q = \frac{v_m c p}{(p_0 - p)(1 + (c-1)(p/p_0))}$$

follows from a body of assumptions about



BET Adsorption

Proof that algebra for linearization is correct

Proof that linear regression minimizes least squares error

Proof that output corresponds to meaningful parameters

Slide from Lecture 4

Definitions

- Allow us to reuse terms outside of individual examples / theorems
- Facilitates modular code and verification of different parts of code
- Propositions
- Functions

Slide from Lecture 5

Functions: Programming vs. Math

Programming perspective

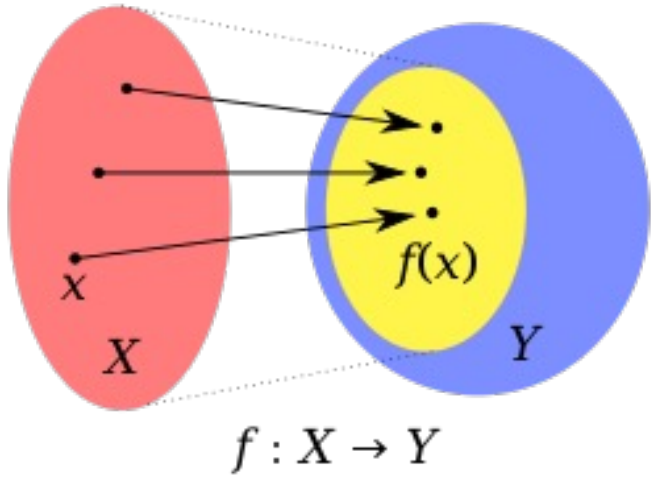
A function takes arguments, performs calculations, and produces an output

Examples in Python

```
def squared(x):
    y = x*x
    return y
```

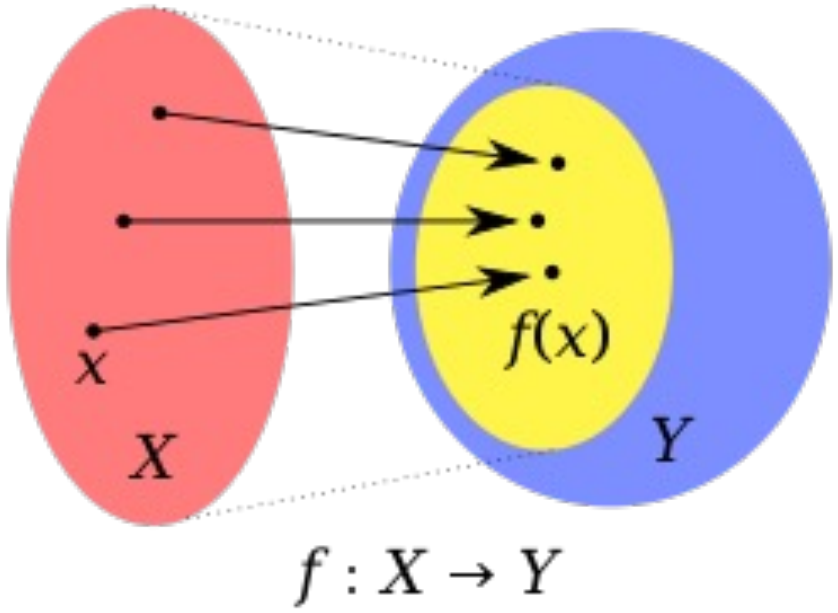
Math perspective

A function maps values from a domain to a co-domain



Slide from Lecture 5

Functions: Programming vs. Math




```
def squareroot(x):
    y = x**(1/2)
    return y
```

$$f(x) = \sqrt{x}$$

Not always a function!
With type $\mathbb{Z} \to \mathbb{Z}$ or $\mathbb{R} \to \mathbb{R}$, there is no mapping from the x < 0 part of the domain

With type $\mathbb{N} \to \mathbb{R}$ or $\mathbb{R} \to \mathbb{C}$, it *is* a function; every part of the domain maps to a value in the co-domain

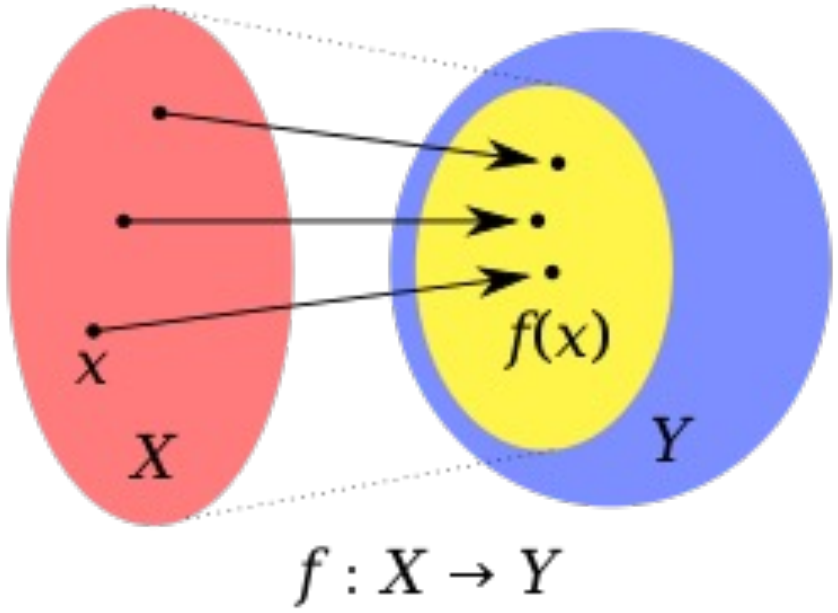
Domain

Co-domain

Image

Slide from Lecture 6

Functions: Programming vs. Math



Domain

Co-domain

Image

```
def division(x,y):
    z = x/y
    return z
```

$$f(x, y) = \frac{x}{y}$$

Not a function!

Type is $\mathbb{R} \to \mathbb{R} \to \mathbb{R}$

Everyplace in the domain maps to an place in the codomain, *except* for y = 0

So… what do you do?

$$y \neq 0 \qquad f(x, y) = \frac{x}{y}$$

$$y = 0 \qquad f(x, y) = 0$$

https://xenaproject.wordpress.com/2020/07/05/division-by-zero-in-type-theory-a-faq/

Slide from Lecture 5

Glossary

- Equation
  - Proposition about equality statement
- Formula
  - Proposition about expressions, includes equalities, inequalities, as well as logical operators
- Expression
  - Like the "right hand side" of an equation
  - Type depends on the types and operations of things inside
- Function (aka pure function)
  - An expression that maps from domain to co-domain
- Partial function
  - An expression that maps from *part* of domain to co-domain

Slide from Lecture 6

Functions in Lean

- Further discussion in Lecture 7
- No parentheses needed – just a space will do
  - f(x) is written as f x
- We can *prove* things about pure functions; it's much harder with partial functions
- Lean requires you to label "noncomputable" functions
  - Noncomputable means "incapable of being computed by any algorithm in a finite amount of time"
  - Real.pi is noncomputable

Slide from Lecture 1

A guide to number systems

$\mathbb{N}$ - Natural numbers (0, 1, 2, 3, 4, …)

$\mathbb{Z}$ - Integers (… -3, -2, -1, 0, 1, 2, …)

$\mathbb{Q}$ - Rational numbers (1/2, 3/4, 5/9, etc.)

$\mathbb{R}$ - Real numbers (-1, 3.6, π, $\sqrt{2}$)

$\mathbb{C}$ - Complex numbers (-1, 5 + 2i, $\sqrt{2}$ + 5i, etc.)

Programming Paradigms

Imperative

- Emphasizes *how* to solve
- **State and Mutation**: Variables can be changed after they are set
- **Procedural Style**: Follows a sequence of steps to achieve a result
- **Control Flow**: Uses loops, conditionals, and other control structures
- **Side Effects**: Functions or methods can modify global state or have other side effects
- **Examples**: Python, Java, most languages

Functional

- Emphasizes *what* to solve
- **Immutability**: Variables, once assigned, cannot be changed
- **Declarative Style**: Focuses on defining and declaring what things are
- **First-Class Functions**: Functions can be passed as arguments, returned from other functions, and assigned to variables
- **Pure Functions**: No side effects, given the same input, always produces the same output
- **Examples**: Haskell, Lean 4!

It's possible to write functional-style code in languages like Python
Lean 4 is *purely functional*; it doesn't let you use imperative techniques

Mutability and Immutability in Python

Lists in Python are *mutable*:

```
a = [10, 20, 30]
a[1] = 80
print(a)
[10, 80, 30]
```

Tuples in Python are *immutable*:

```
b = (10, 20, 30)
b[1] = 80
TypeError: 'tuple' object does not
support item assignment
```

Something *like* mutation is possible by creating a new variable:

```
b = (10, 20, 30)
c = (b[0], 80, b[2])
print(c)
(10, 80, 30)
```

Mutability – not allowed in Lean

- Declarations must be unique

```
def a : Float := 50.0

def a : Float := a + 10
Error: 'a' has already been declared
```

- Sometimes, a Lean tactic might generate a variable with the same name as one already declared. When this happens, Lean renames the new one, adding a dagger †

Why is mutability so popular?

Efficiency

0.61	0.13	0.03
0.27	0.68	0.22
0.22	0.83	0.98
0.15	0.99	0.14
0.24	0.38	0.62
0.46	0.92	0.88
0.41	0.28	0.69
0.58	0.29	0.36
0.68	0.89	0.02
0.89	0.15	0.94

Multiply one element by 2 →

0.61	0.13	0.03
0.27	0.68	0.22
0.22	0.83	0.98
0.15	0.99	0.14
0.24	0.76	0.62
0.46	0.92	0.88
0.41	0.28	0.69
0.58	0.29	0.36
0.68	0.89	0.02
0.89	0.15	0.94

If this matrix is immutable, you need to re-copy the rest of the matrix!

In this case, 2x the memory and 30x the computational cost

Functional programming languages use various tricks to manage cost

Lean 4 introduced the "functional but in-place" paradigm

(see de Moura and Ullrich, CADE 2021 for more details)

Recursive functions

- Functions can call other functions
- A function is recursive when *it calls itself*
- Python example: factorial function, n!

Imperative style

```
def factorial_loop(n):
    result = 1
    for i in range(1,n+1):
        result = result*i
    return result
```

Functional style

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

Factorial function – for loop

Imperative style

```
def factorial_loop(n):
    result = 1
    for i in range(1,n+1):
        result = result*i
    return result
```

Potential for "side effects"

"result" holds a value in memory, which is constantly being overwritten
If "result" is a global variable, other parts of the program might modify it

```
factorial_loop(5)


n       i       result
5               1
5       1       1
5       2       2
5       3       6
5       4       24
5       5       120


return 120
```

Factorial function – recursive

Functional style

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

Notice how the "stack" of calculations keeps increasing.
At scale, this creates memory issues.

This means this is not "tail recursive."

```
factorial(5)


factorial(5)
5*factorial(5-1)
5*factorial(4)
5*4*factorial(3)
5*4*3*factorial(2)
5*4*3*2*factorial(1)
5*4*3*2*1*factorial(0)
5*4*3*2*1*1


return 120
```

Factorial function – tail-recursive

Functional style

```
def factorial_tail(n, acc=1):
    if n == 0:
        return acc
    else:
        return factorial_tail(n-1, n*acc)
```

This tail-recursive function manages the "stack" so it doesn't blow up.

Almost always, tail-recursive functions perform better

```
factorial(5)


factorial(5,1)
factorial(4,5*1)
factorial(4,5)
factorial(3,5*4)
factorial(3,20)
factorial(2,20*3)
factorial(2,60)
factorial(1,60*2)
factorial(1,120)
factorial(0,120)


return 120
```

The halting problem

- Consider an arbitrary program in a programming language
- Will it stop running (halt) or will it run forever?

```
i = 0
while i < 10:
    print i
    i = i + 1
```

Halts

```
i = 0
while i < 10:
    print i
    i = i - 1
```

Runs forever

```
i = 20
while i < 10:
    print i
    i = i - 1
```

Halts

- You can prove a program halts by running and seeing it halt
- But, it's hard to tell the difference between "runs a long time" and "runs forever"

The halting problem

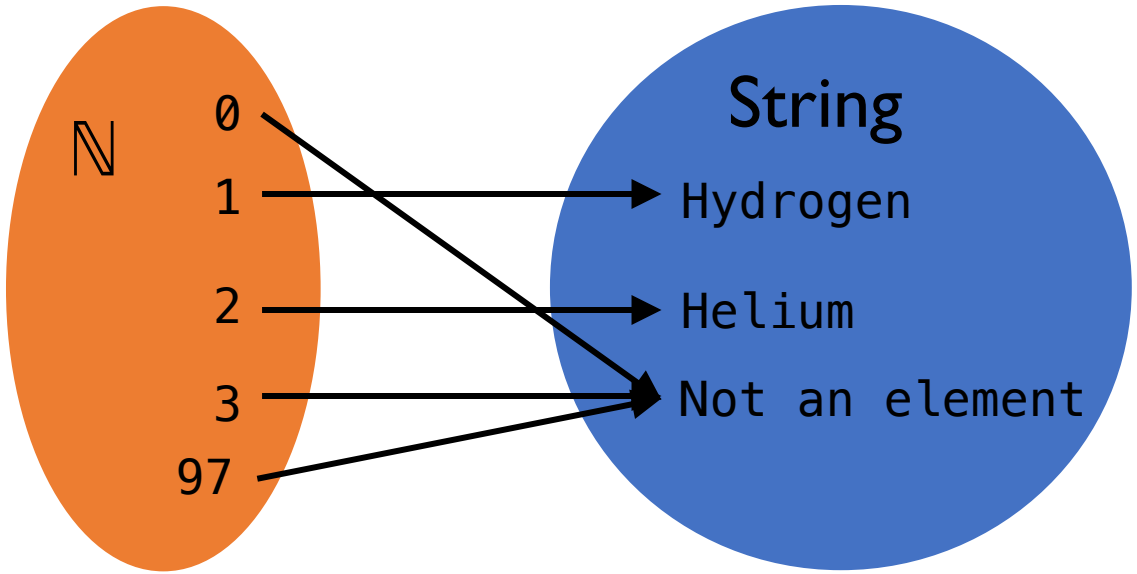
- Let's consider recursive functions
- Does factorial(5) halt?
- How about factorial(20)?
- factorial(1523482)?
- What about factorial(-3)?
- factorial(-60)?

```
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
```

You don't need to finish running the program every time
You're using logic to figure this out!

Interlude: Pattern matching

```
def element : Nat → String
  | 0 => "Not an element"
  | 1 => "Hydrogen"
  | 2 => "Helium"
  | _ => "Not an element"
```



Like “match … case” in Python

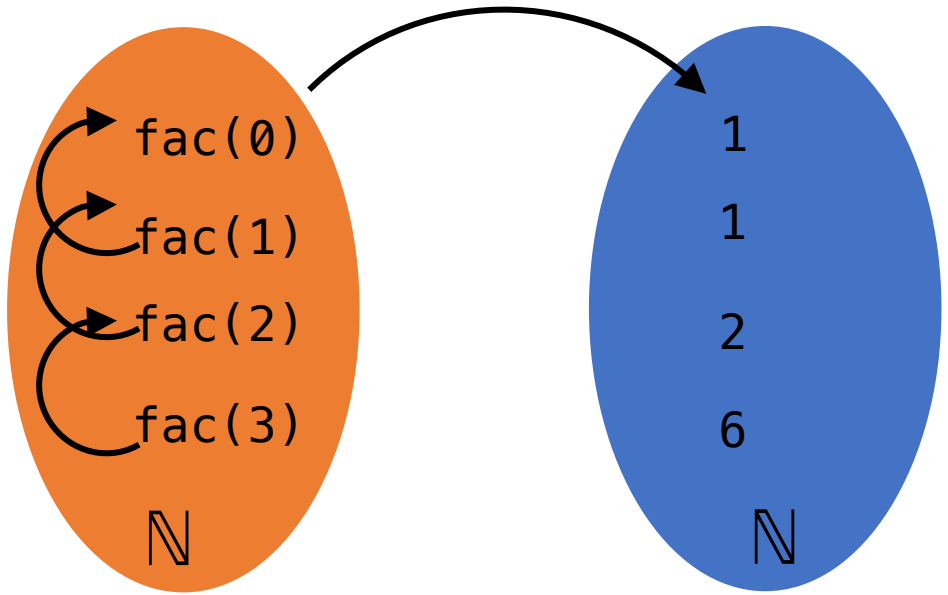
Every entry in the domain must be covered!

Learn more in FPIL 1.5; TPIL 8

Recursion in Lean

This function works

```
def factorial : ℕ → ℕ
  | 0 => 1
  | n + 1 => (n + 1) * factorial n
```



This function is broken

```
def not_factorial : ℕ → ℕ
  | 0 => 1
  | n + 1 => (n + 1) * not_factorial (n+1)
```

Check out the error message on not_factorial:

```
fail to show termination for not_factorial
with errors
structural recursion cannot be used:
```

In factorial, Lean automatically proves termination via structural recursion, so this function is okay.

Lean-like recursion in Python

```
def factorial : ℕ → ℕ
  | 0 => 1
  | n + 1 => (n + 1) * factorial n
```

```
def factorial(n: int) -> int:
    match n:
        case 0:
            return 1
        case _:
            return n * factorial(n - 1)
```

Structures in Lean

- If you've used C++ before, structures in Lean are like structures in C++
- Something defined as a structure must have all fields defined
- Example 1: points on an x-y plane
  - Functional programming in Lean, 1.4
- Example 2: atoms and molecules!
- Example 3: gas law thermodynamics

Structures in Lean

What if your “thing” has multiple parts? How can you define its type?

Structures are useful for this

Consider a “Point” on an xy plane with two values (FPIL 1.4)

Gas law thermodynamics with structures

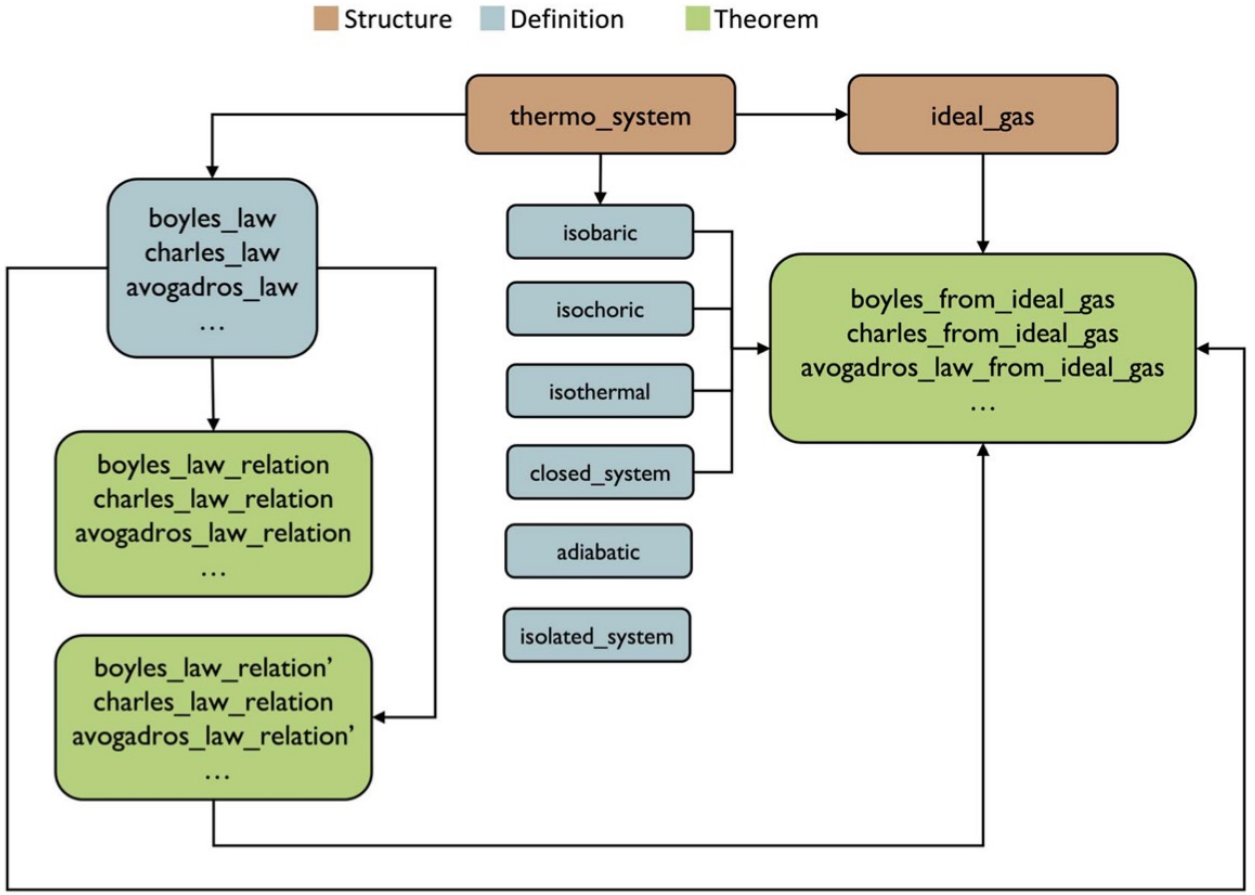


Fig. 3 Thermodynamic system in Lean. Here the *thermo_system* and *ideal_gas* are Lean *structures* that describe different kinds of thermodynamic systems like *isobaric*, *isochoric*, *isothermal etc.* using Lean *definitions* to proof *theorems* relating to the *gas laws*.

https://github.com/ATOMSLab/LeanChemicalTheories/blob/00841fedb86c0af282f515eda7e83a525a42e048/src/thermodynamics/basic.lean - sorry, only in Lean 3 today