```
--------- THE LEAN PROOF ASSISTANT ---------

-- PRESENTER --
def Name := (Option String × String)
def presenter : Name :=
  (some "Fynn", "Godau")

-- for aesthetics
def day   { n : Nat } (x : Fin (n+1)) := x - 1
def month { n : Nat } (x : Fin (n+1)) := x - 1
def year : Nat → Nat := id

-- PRESENTATION DATE --
def Date  : Type := (Fin 31 × Fin 12 × Int)
def today : Date :=
  (day     26,
   month ⟨(Nat.succ Nat.zero), by simp⟩,
   year    2026)
```

Lean is a functional programming and proof language.

Lean is a functional programming and proof language.

Lean's logic implements the Calculus of Inductive Construction.
What does this mean?

Existence proof

How would you prove that a function of the following type exists?

$$\alpha \times (\alpha \to \beta) \to \beta$$

Existence proof

How would you prove that a function of the following type exists?

$$\alpha \times (\alpha \to \beta) \to \beta$$

By implementing it!

$$(a, f) \mapsto f(a)$$

The compiler of a typed programming language will tell you whether your proof is correct.

Existence proof

How would you prove that a function of the following type exists?

$$\alpha \times (\alpha \to \beta) \to \beta$$

By implementing it!

$$(a, f) \mapsto f(a)$$

The compiler of a typed programming language will tell you whether your proof is correct.

This is an example of the **Curry-Howard-Isomorphism**.

Curry-Howard-Isomorphism

- utilizes similarities between the worlds of functions and proofs

data type (`Type`)	$\leftrightsquigarrow$	proposition (`Prop`)
data instance	$\leftrightsquigarrow$	proof
function (of type $\alpha \to \beta$)	$\leftrightsquigarrow$	implication (of type `Prop`$\to$`Prop`)
$\times$	$\leftrightsquigarrow$	$\wedge$

- enables program code and proof code to share a single syntax

Lean

What does this have to do with Lean?

Lean

What does this have to do with Lean?

- Curry-Howard-Isomorphism is the idea of the **Calculus of Construction** (CoC)

Lean

What does this have to do with Lean?

- Curry-Howard-Isomorphism is the idea of the **Calculus of Construction** (CoC)
- Add inductive types to get the **Calculus of Inductive Construction** (CIC)

Lean

What does this have to do with Lean?

- Curry-Howard-Isomorphism is the idea of the **Calculus of Construction** (CoC)
- Add inductive types to get the **Calculus of Inductive Construction** (CIC)
- Lean **implements** the Calculus of Inductive Construction

Well-known examples for Inductive Types

```
inductive IntList
| nil : IntList
| cons (h : Int) (t : IntList) : IntList

inductive Nat
| zero : Nat
| succ (n : Nat) : Nat
```

Same as other functional programming languages.

Quiz

Quiz: Which of these propositions have an instance in Lean?

- `True`
- `False`
- `¬(p∧¬p)`
- `¬¬p→p`

Quiz / Answers

See `Quiz.lean`.

```
example : True := True.intro

-- type is empty; has no sound proof
example : False := sorry

example : ¬(p ∧ (¬p)) :=
  fun ⟨ a, b ⟩ => b a
```

(Note that `And` is an inductively defined type.)

Quiz / Classical Logic

See `Quiz.lean`.

```
example : ((p → False) → False) → p
  := fun f => sorry
-- we have f : (p → False) → False,
-- but need to procure an x : p
```

- intuitively, the statement holds
- no constructive proof possible

$\Rightarrow$ we need "classical reasoning", which is not available a priori

Axiom of Choice

For every nonempty type, there is a function that picks an arbitrary element.

```
axiom choice (α : Sort u) : Nonempty α → α
```

Axiom of Choice

For every nonempty type, there is a function that picks an arbitrary element.

```
axiom choice (α : Sort u) : Nonempty α → α
```

Using this axiom introduces classical logic into Lean.

"set α is not empty" $\Rightarrow$ "there exists an $a \in \alpha$"

$$\neg\neg p \Rightarrow p$$

Tactic mode

See `Tactics.lean`.

```
theorem linearEq :
  (a b c : Int) → (a - b < c) → (c + b > a) :=
  sorry

-- double negation and excluded middle are equivalent
theorem doubleNegationEqEm :
  (¬¬p ↔ p) ↔ (p ∨ ¬p) :=
  sorry
```

Tactic mode / Answers

```
theorem linearEq :
  (a b c : Int) → (a - b < c) → (c + b > a) :=
  by omega -- ILP solver

-- double negation and excluded middle are equivalent
theorem doubleNegationEqEm :
  (¬¬p ↔ p) ↔ (p ∨ ¬p) :=
  by grind -- find proof automatically
```

`InsertionSort` comparison

See `SortingList.lean` and `SortingArray.lean`.

List	Array
automatic termination proof	manual termination proof
—	in-place*
correctness proofs straight-forward	correctness proofs tricky

* Lean automatically optimizes the `Array.swap` operation to run in-place if possible, but the in-place property cannot be proven in the Lean language.

Comparison to Rocq

Lean and Rocq both...

- are concrete implementations of the same theory (Calculus of Inductive Construction).
- hold similar axioms.

The differences are in the "implementation details".

Comparison to Rocq

Lean and Rocq both...

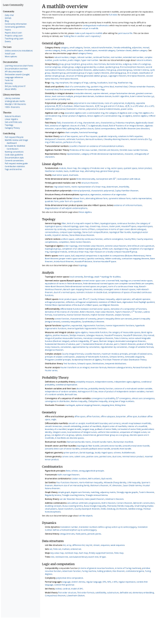
- are concrete implementations of the same theory (Calculus of Inductive Construction).
- hold similar axioms.

The differences are in the "implementation details".

(Over which a real type theorist would fight culture wars.)

mathlib

- designed for research
- aims to be *the* coherent mathematics library for Lean
- still work in progress; some areas of math have not yet seen much progress


Lean Community

A mathlib overview

General algebra

Linear algebra

Topology

Analysis

Probability Theory

Geometry

Combinatorics

Dynamics

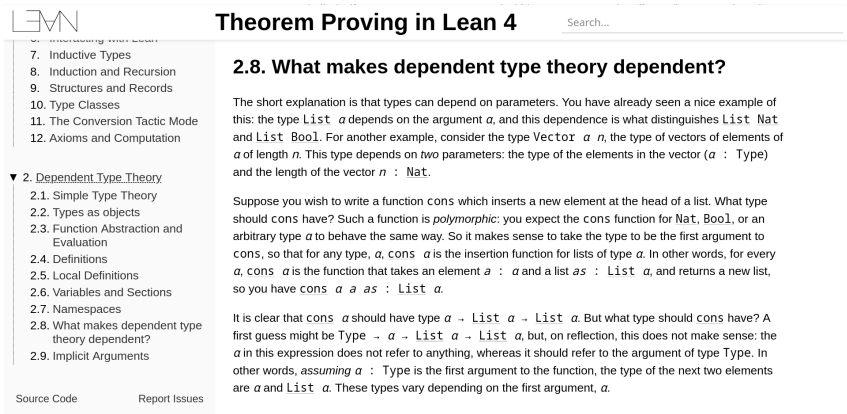
Data structures

Logic and computation

Example:
"Continuum Hypothesis is independent of ZFC" was *first* verified in Lean, using mathlib.

How to learn Lean

"Theorem Proving in Lean 4"
https://lean-lang.org/theorem_proving_in_lean4/



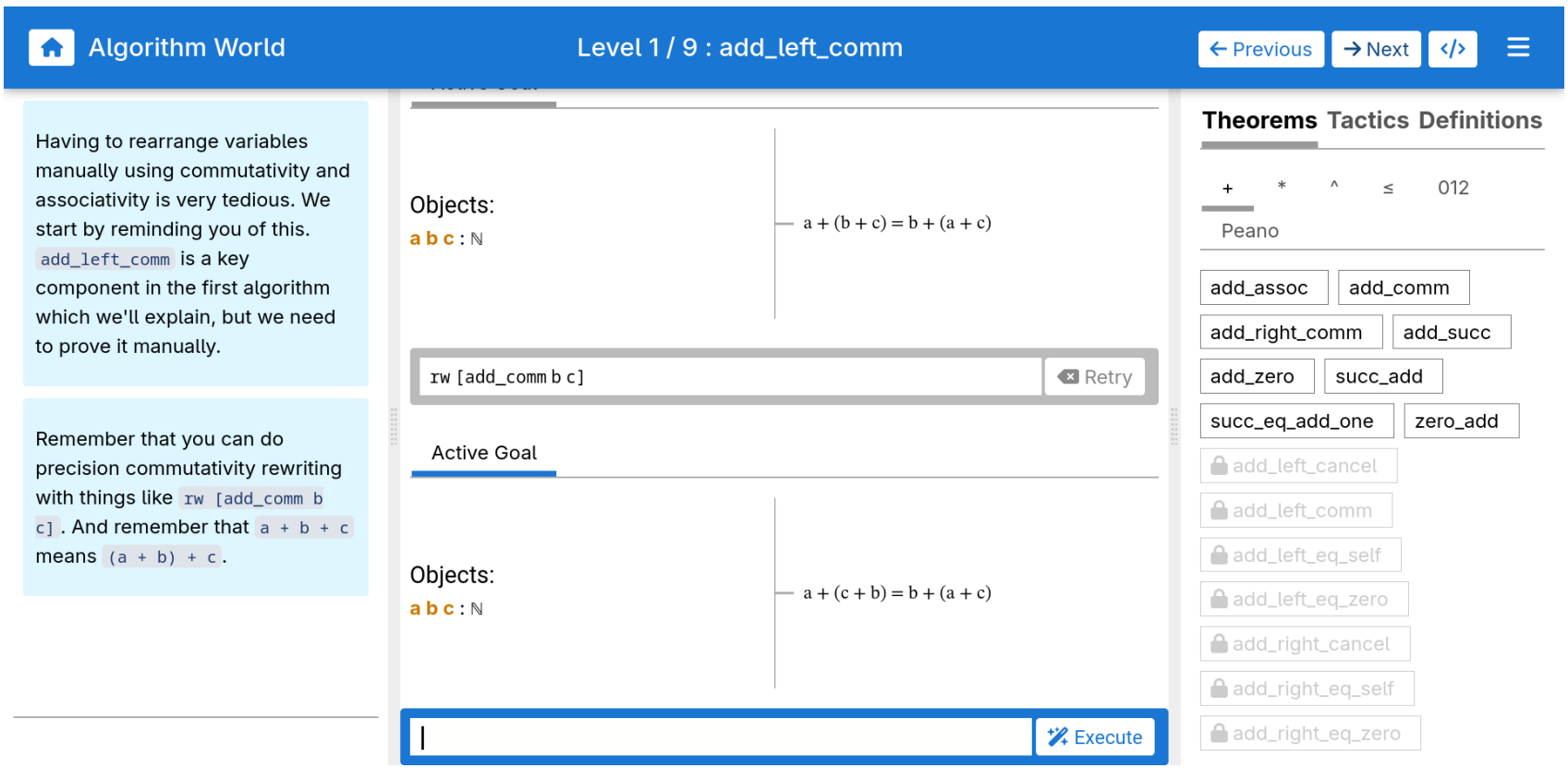
- my personal starting point, until I felt confident enough to implement and prove `insertionSort`
- to be read at a slow pace
- good iterative explanations of Lean's core concepts
- much additional type-theoretic understanding needed to *really get it*

How to learn Lean

Lean Game Server

https://adam.math.hhu.de/

- interactive tutorials teaching tactic mode
- learned about it too late, but seems like an easy way to start



Is it worth it?

It depends on your starting point.

Is it worth it?

It depends on your starting point.

- functional programming
  $\Rightarrow$ a few hours of confusion, then the concepts become natural

Is it worth it?

It depends on your starting point.

- functional programming
  $\Rightarrow$ a few hours of confusion, then the concepts become natural
- imperative programming verification
  $\Rightarrow$ pure joy not having to think about the heap

Is it worth it?

It depends on your starting point.

- functional programming
  $\Rightarrow$ a few hours of confusion, then the concepts become natural
- imperative programming verification
  $\Rightarrow$ pure joy not having to think about the heap
- Rocq
  $\Rightarrow$ not much will be different

~ Appendix ~

Dependent types

What is p in the last quiz example?

```
example : ((p → False) → False) → p
  := fun f => sorry
```

Dependent types

What is p in the last quiz example?

```
example : ((p → False) → False) → p
  := fun f => sorry
```

Its use is actually short for:

```
example : (p : Prop) → ((p → False) → False) → p
  := fun p => fun f => sorry
```

In the type declaration, the variable p is assigned and then used in later parts of the definition.

Dependent types

What is p in the last quiz example?

```
example : ((p → False) → False) → p
  := fun f => sorry
```

Its use is actually short for:

```
example : (p : Prop) → ((p → False) → False) → p
  := fun p => fun f => sorry
```

In the type declaration, the variable p is assigned and then used in later parts of the definition.

This is also possible for functions on data, for polymorphic functions:

```
def insertionSort :
  (α : Type) → List α → List α := sorry
```

Axioms in Lean

- Propositional extensionality: the concept of "proof irrelevance".

```
axiom propext (a b : Prop) : (a ↔ b) → a = b
```

Axioms in Lean

- Propositional extensionality: the concept of "proof irrelevance".

```
axiom propext (a b : Prop) : (a ↔ b) → a = b
```

Why is this desirable?

Axioms in Lean

- Propositional extensionality: the concept of "proof irrelevance".

```
axiom propext (a b : Prop) : (a ↔ b) → a = b
```

Consider a data type that contains a proof:

```
inductive Z₁₀
| intro (i : Nat) (proof : i < 10)

def a : Z₁₀ := Z₁₀.intro 2 (by omega)
def b : Z₁₀ := Z₁₀.intro 2 sorry

example : a = b := rfl
```

Practical motivation: proofs are erased at compile-time.

Axioms in Lean

- Quotient construction soundness:
  In essence, we can group every set (i. e. type) by any relation, think equivalence classes.

Axioms in Lean

- Quotient construction soundness:
  In essence, we can group every set (i. e. type) by any relation, think equivalence classes.

- Axiom of choice: for every nonempty type, there is a function that picks an arbitrary element.

```
axiom choice (α : Sort u) : Nonempty α → α
```

Axioms in Lean

- Quotient construction soundness:
  In essence, we can group every set (i. e. type) by any relation, think equivalence classes.

- Axiom of choice: for every nonempty type, there is a function that picks an arbitrary element.

```
axiom choice (α : Sort u) : Nonempty α → α
```

  Using this axiom introduces classical logic into Lean.

$$\text{“set } \alpha \text{ is not empty”} \Rightarrow \text{“there exists an } a \in \alpha\text{”}$$

$$\neg\neg p \Rightarrow p$$

Axioms in Lean and Rocq

- Lean's axioms imply Rocq's axioms
- Rocq's axioms do not imply definitional proof irrelevance

Computability of functions

`noncomputable` functions:

- for reasoning
- can use axioms
- are dropped at runtime

computable functions:

- for evaluation (executing)
- can be compiled as standalone programs

Lean tracks each function's status.

`partial` functions

- Lean requires a termination proof by default
- no termination proof necessary for `partial` or `partial_fixpoint` functions
- `partial` functions are opaque – cannot be unfolded
- `partial_fixpoint` functions are not opaque, but require recursion to be in tail position
- a proof that return type is inhabited (`Nonempty`) is always required, due to soundness