Introduction to Lean theorem prover

Yoh Tanimoto

University of Rome "Tor Vergata"

Kyoto operator algebras seminar
15 May 2026

(Disclaimer)

I am not an expert of Lean or logic, rather a user of Lean with a bit of experience.

- In 2020 I did some tutorials.
- Then I had a break.
- In late 2023 I resumed tutorials.
- In January 2024 I organized a workshop in Rome and met the experts.
- In early 2024 I started to contribute to `mathlib`, the main library of Lean.

There are very few people (~ 5) doing things on operator algebras for `mathlib`. In other words, you can get to the frontline very quickly.

What is Lean?

Lean is a **proof assistant**, or **interactive theorem prover**. You write the statements and the proofs in the computer language. Computer gives suggestions and **verifies the proofs**.

As of May 2026,

- You can ask AI to write Lean proofs of **some** problems ("automatic theorem proving"), or to formalise **some** textbooks ("autoformalisation").
- "Mathlib intentionally has very high standards (on generality, integration with the remaining library and maintainability, including code style). As of mid-2026, code written by an AI without the supervision of a Lean subject expert fails to meet that bar by a large margin." link

(terminology)

- Lean : the formal computer language
- `mathlib` : the mathematical library in Lean

Why Lean, an interactive theorem prover?

In mathematics, we usually write (informal) proofs. Formal proofs are sequences of statements that can be derived from the axioms and deduction rules.

Hilbert: "One must be able to say at all times–instead of points, straight lines, and planes–tables, chairs, and beer mugs" (link)

Mathematical statements can be written in the symbolic language and processed and verified by computer.

Some current research results are formalised in real time.

In Lean we can talk about (currently very basic) stuff about operator algebras.

(Why I am interested)

I studied physics in my bachelor's study. It was OK until Quantum Mechanics (operator theory).

I dropped out when I studies (interacting) Quantum Field Theory, which is explained in a very *ad hoc* way in textbooks, and switched to mathematic(al physic)s.

In mathematical physics, we value **rigor**. We are supposed to define physical models and prove theorems about them. E.g. conformal nets, vertex operator algebras...

I gradually learnt that, in the most advanced research, this high standard is not always kept. Cf. the UV stability of the Yang-Mills model in 4d.

(my collaborator challenged me: if we wrote a complete proof of the UV stability of more than 1000 pages, who would read it?)

Lean, as a proof assistant

- Definition of C^*-algebras
- Gelfand duality
- Definition of von Neumann algebras
- Riesz-Markov-Kakutani theorem

You write the definitions, statements and proofs, the computer verifies the proofs.

Why interacting theorem prover (in the past)?

Mathematicans had formalised some part of mathematics (Mizar, Metamath, HOL, Isabelle, Coq...), but the developments in most cases had not reached the actual research level until recently.

Theorem provers have been mainly developed by computer scientists, with real applications to verification of hardware and software. "Mistakes can be very costly, examples are the destruction of the Ariane 5 rocket (caused by a simple integer overflow problem that could have been detected by a formal verification procedure) and the error in the floating point unit of the Pentium II processor." (Bridge, 2010. link)

"The failure resulted in a loss of more than US$370 million." (Wikipedia entry about Ariane flight V88)

Lean is developed by people including engineers at Microsoft. Now it is being used in a (long-term) project to verify the messaging app Signal.

Formal system

A **proof** in a formal system is a sequence of formulas that follow the **inference rules**.
Examples (if we already know the propositions above the line, then we can also infer the proposition below the line):

- And-introduction $\dfrac{P \qquad Q}{P \wedge Q}$
- And-elimination, left $\dfrac{P \wedge Q}{P}$, right $\dfrac{P \wedge Q}{Q}$
- $\to$-elimination (modus-ponens) $\dfrac{P \qquad P \to Q}{Q}$
- $\forall$-elimination (specialization) $\dfrac{\forall x, P(x)}{P(x_0)}$
- Introduction rules for $\to, \forall$, rules for $\wedge, \vee, \neg, \exists$

The domain of $\forall x$ depends on the system.

Formal system

cf. Peano axioms in second order logic (not used in Lean)

If x is a term, $S(x)$ is another term.

1. $\forall x, 0 \neq S(x)$
2. $\forall x, y(S(x) = S(y) \implies x = y)$
3. $\forall x(x + 0 = x)$
4. $\forall x, y(x + S(y) = S(x + y))$
5. $\forall x(x \cdot 0 = 0)$
6. $\forall x, y(x \cdot S(y) = x \cdot y + x)$
7. (Induction, quantifycation over predicates "Px")
   $\forall P(P0 \wedge \forall x(Px \to P(S(x))) \to \forall Px)$

(3)–(6) should be seen as inductive definitions of $+, \cdot$. Denote $1 := S(0)$.

The domain of $\forall x$ is considered the "natural numbers", terms that can be written as $0, S(0), S(S(0)), \cdots$.

Formal system

Some Peano axioms

- $\forall x(x + 0 = x)$
- $\forall x, y(x + S(y) = S(x + y))$
- (Induction) $\forall P(P0 \wedge \forall x(Px \to P(S(x))) \to \forall Px)$

Denote $1 := S(0)$.

Theorem : $\forall x(x + 0 = 0 + x)$

Proof: Put $Px := x + 0 = 0 + x$.
$\checkmark P0 : 0 + 0 = 0 + 0$
Assume, for a fixed x, Px. Then
$S(x) + 0 = S(x) = S(x + 0) = S(0 + x) = 0 + S(x)$.
$\checkmark Px \to P(S(x))$
By induction, $\forall x, Px$ follows.

(in principle we can make a diagram of inferences...)

Formal system

A **proof** in a formal system is a sequence of formulas that follow the **inference rules**.

Example in the *natural deduction* (taken from Logic and proof)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x\ A(x)}{A(y)}\,1 \qquad \dfrac{\forall x\ B(x)}{B(y)}\,2}{A(y) \wedge B(y)}}{\forall y\ (A(y) \wedge B(y))}}{\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y))}\,2}{\forall x\ A(x) \to (\forall x\ B(x) \to \forall y\ (A(y) \wedge B(y)))}\,1$$

Dependent type theory

Lean uses **dependent type theory** as its basis. Everything has a **type**. "`x : X`" means "`x` has type `X`".

- `Nat` ($= \mathbb{N}$) (we will see the definition in a moment)
- `zero : Nat` ($= 0$), `succ zero : Nat` ($= 1$).

From one type, we can create more types.

- $\mathbb{N} \times \mathbb{N}$ (ordered pairs)
- $\mathbb{Z}$ (defined as a type containing images of $\mathbb{N}$ by embedding and $\mathbb{N}$ by $n \mapsto -(n+1)$)
- $\mathbb{Q}$ (defined as a type of $\mathbb{N} \times \mathbb{Z}$ with conditions (non zero deniminator, reduced))
- $\mathbb{N} \to \mathbb{Q}$ (sequences in $\mathbb{Q}$)
- $\mathbb{R}$ (defined as the completion of a quotient of the type of Cauchy sequences)

Dependent type theory / Calculus of constructions

In Lean, proposions have also a taylored type.

- `Prop` (propositions)
- $\mathbb{R} \to$ `Prop` (predicates that depends on a real number)

When `p : Prop`, then `p` itself *is a type*.
`hp : p` is interpreted as *a proof of* `p`.

Having a proof of `p : Prop` means having an object `hp : p`.

Propositional logic in Lean / Calculus of constructions

- variable declarations
- `p : Prop` ("`p` is a proposition")
- `hp : p` ("`hp` is a proof of `p`")

```
variable (p :  Prop) (hp :  p)
#check p
#check hp
```

link

Propositional logic in Lean / Calculus of constructions

- If `p q : Prop`, then `p ∧ q : Prop`.

```
variable (p q : Prop)
#check p ∧ q
```

- If `hp : p` and `hq : q`, then we have a proof of `p ∧ q`.
- Indeed, Lean allows one to create an object of type `p ∧ q` if we already have objects `hp : p, hq : q`.

```
variable (p q : Prop) (hp : p) (hq : q)
#check And.intro hp hq
```

link

Propositional logic in Lean / Calculus of constructions

- If `p q : Prop`, then `p → q : Prop`.

```
variable (p q : Prop)
#check p → q
```

- Assume that we have a proof `hp : p`.
- How can we get a proof of `p → q`??

```
variable (p q : Prop) (hp : p)
#check p → q
theorem not_a_good_theorem : p → q := sorry
```

link
(if we have `hq : q`, we have a trivial proof of `p → q`)

Corresponding to the inference rules, there are operations to construct objects of the type.

Inductive type

In Lean, `Nat` ($= \mathbb{N}$) is defined as follows (link)

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

This means that

- `Nat` is a type
- `zero` has type `Nat`
- `succ zero` has type `Nat`
- `succ (succ zero)` has type `Nat`
- ...
- Nothing else has type `Nat`
- Because `n :  Nat` is either `zero` or `succ m` with another `m :  Nat`, if we prove a predicate `P n` for both cases, Lean accepts it as a proof of `∀ n, P n` (induction)

Inductive type

We can define

```
def MyAdd (m n :  Nat) :  Nat :=
match n with
| zero => m
| succ n => succ (MyAdd m n)
```

We are defining `MyAdd`, say $\underset{\mathtt{My}}{+}$ inductively:

- $m \underset{\mathtt{My}}{+} 0 := m$
- $m \underset{\mathtt{My}}{+} (n+1) := (m \underset{\mathtt{My}}{+} n) \underset{\mathtt{My}}{+} 1$

Can we prove $0 \underset{\mathtt{My}}{+} n = n \underset{\mathtt{My}}{+} 0$?

Inductive type

```
#check Nat.succ.injEq

theorem Nat.MyAdd_comm (m :  Nat) :
    MyAdd m zero = MyAdd zero m :=
match m with
| zero => rfl
| succ n => by
  rw [MyAdd, MyAdd]
  rw [← Nat.MyAdd_comm n]
  rw [MyAdd]
```

link

Can we prove $m \underset{\texttt{My}}{+} n = n \underset{\texttt{My}}{+} m$?

```
theorem MyAdd_comm (m n :  Nat) :
    MyAdd m n = MyAdd n m := sorry
```

Which mathematics can we write in Lean?

- Lean : the formal computer language
- `mathlib` : the mathematical library in Lean

The "core" of Lean contains only very basic definitions such as `Nat`, `Int`, `Rat`...

`mathlib` contains algebraic and analytic structures on `Nat`, `Int`, `Rat`..., the definition of `Real`, `Complex`.
More abstract structures as well such as `Magma`, `Monoid`, `Semigroup`, `Group`, `Module`, `Algebra`, topological properties such as `TopologicalSpace`, `MetricSpace`, `CompleteSpace`, analytic structures such as `integral`, `fderiv`, `InnerProductSpace`, `CStarAlgebra`, `ContinuousLinearMap`...

What is a formal proof good for?

- correctness
- encouraging complete proofs
- searchability
- collaboration
- education

Encouraging complete proofs

To write a formal proof, one needs to know a very detailed informal proof. If it becomes more common to write formal proofs, it will urge people to give details.

Terence Tao, in an attempt to formalise his proof, he noticed that he had done a division by zero.
(link)

Gouëzel–Shchur found a reversed inequality in a paper, corrected it and check the proof (in Isabelle/HOL).
(link)

Tooby–Smith found that a claim about stability of the potential in a theoretical physics paper was not really sufficient for stability.
(link)

Searchability

`mathlib` is accompanied with various search engines.

- leansearch
- loogle

One can find statements that contain a certain combination of keywords (link)

Collaboration

There are some theorems that require very different fields of mathematics.
Anyone can help by filling in auxilliary results needed in a bigger project.
(Fermat Last Theorem blueprint, Chapter 10 about Haar measure)

Can AI write proofs (in Lean) of interesting problems?

- Erdős problems database
- Sphere packing (announcement by Math. inc) (report by human formalisers)
- Textbook autoformalisation
- Aristotle API

Can AI write proofs (in Lean) of interesting problems?

Classical theorems

- ✓ The double commutant theorem (Solution by Aristotle, supervised by Luccioli (10 May 2026) : $\sim$ 600 lines of Lean code)
- Kaplansky density theorem (probably?)
- Classification of factors (the definition of types not yet available)
- The Tomita-Takesaki theory (the definition of the graph of antilinear operators not yet available)

Open/recently solved problems

- Connes embedding conjecture/refutation of it (the statement not yet available)
- The free group factor problem (the definition of the free group factors not yet available)
- The invariant subspace problem (the statement **available** since August 2025)
- Formal-conjectures repo

Can AI write definitions?

Do we trust autoformalisation?

Bad examples:

- C^∞ vs C^ω
- Osterwalder-Schrader axioms for test functions supported in the full space
- reflection positivity without reflection

Or perhaps they are not wrong, but how do we know it?

- Lean checks **proofs**
- No one except you tells that the Lean **definitions** and **statements** are those that you are interested in.

How can AI help us do mathematics?

- You don't want to read AI-generated proofs.
- You don't want to read AI-generated definitions either.
- If you don't read the definitions and statements, you don't know what is proven.

- You need to write (or verify) **definitions and statements**.
- Perhaps AI can write proofs of some lemmas and/or help organizing long proofs.
- You need these lemmas to make **more definitions**.

- If you don't care about `mathlib`, maybe it is ok.
- `mathlib AI policy`

  "Getting code to mathlib's standards requires understanding and writing Lean code by hand."

What have been formalised about operator algebras?

Latest developments:

- weak operator topology
- continuous functional calculus
- Riesz-Markov-Kakutani theorem
- GNS representation
- Integration against vector-valued measures

(Personal) outlook

Spectral theorem for bounded self-adjoint operators on a Hilber space.
What is done:

- Hilbert spaces
- Bounded operators, adjoint
- Definition of C^*-algebras
- Continuous functional calculus in a C^*-algebra
- Weak and strong operator topologies
- Measure theory
- Riesz-Markov-Kakutani representation theorem

Todo:

- Definition of resolution of unity (projection-valued measure)
- Continuity results of functional calculus
- Riesz-Markov-Kakutani theorem for bounded complex functionals
- (Correspondence between sesquilinear forms and operators)

Using Lean

Lean portal : Lean Programming Language

Try: Lean playground

Set up: Installation instructions

Textbooks:

- Natural Number Game
- Mechanics of proof
- Mathematics in Lean
- Theorem proving in Lean

Browse: `mathlib` documentation

Search: LeanSearch

Ask: Zulip chat