Introduzione alla formalizzazione della matematica in Lean
Lesson 2/10 Dependent type theory

Yoh Tanimoto

Corso di dottorato, University of Rome “Tor Vergata”

11 Dicembre 2025

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".

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. For Peano's axioms, they are terms that can be written as $0, S(0), S(S(0)), \cdots$.

Formal system

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

Example (taken from Logic and proof)

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

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 $Px := \forall x(x + 0 = 0 + x)$.
$\checkmark P0 : 0 + 0 = 0 + 0$
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, $\forall Px$ follows.

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

Formal system

Theorem : $\forall x(x+1=1+x)$

Proof: Put $Px := \forall x(x+1=1+x)$.
$\checkmark P0 : 0+1=0+S(0)=S(0+0)=S(0)=1=1+0$
Assume, for a fixed x, Px. Then $S(x)+1=S(x)+S(0)=S(S(x)+0)=S(S(x))=S(S(x+0))=S(x+S(0))=S(x+1)=S(1+x)=1+S(x)$.
$\checkmark Px \to P(S(x))$
By induction, $\forall Px$ follows.

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

By a double induction...

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` and `hp : p`, this means that `hp` is *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
```

Propositional logic in Lean / Calculus of constructions

"and" $\wedge$

Let `p q : Prop`. If we have proofs of `p, q`, then we also have a proof of `p ∧ q`:

$$\frac{\texttt{p} \quad \texttt{q}}{\texttt{p} \wedge \texttt{q}}$$

"if... then" $\rightarrow$

Let `p q : Prop`. If we have proofs of `p` and `p → q`, then we also have a proof of `q`:

$$\frac{\texttt{p} \quad \texttt{p} \rightarrow \texttt{q}}{\texttt{q}}$$

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
```

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

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

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

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

link

Propositional logic in Lean / Calculus of constructions

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

link

In Lean, the application of a function `f` to an object `x` is denoted by `f x`, instead of `f(x)`.
`hpq : p → q` behaves as if it is a function from `p` to `q`. That is, an object `hpq : p → q` produces a proof of `q` from `hp : p`.

Propositional logic in Lean / Calculus of constructions

```
variable (p q r :  Prop) (hp :  p) (hpq :  p → q) (hqr :
q → r)
#check hqr (hpq hp)
```

link

```
variable (p q :  Prop) (hpq :  p ∧ q)
#check hqp.left
#check hqp.right
```

Writing theorems and proofs in Lean

We can write it as a theorem

```
theorem modus_ponens_comp (p q r : Prop) (hp : p)
    (hpq : p → q) (hqr : q → r) : r := hqr (hpq hp)
```

Read:

- Assume that `p, q, r` are propositions
- Assume that `p, p → q, q → r` are true, their proofs are called `hp, hpq, hqr`
- Then `r` is true
- The proof is given by `hqr (hpq hp)`

link

One can omit the theorem name and write "example".

```
example (p q r : Prop) (hp : p) (hpq : p → q)
    (hqr : q → r) : r := hqr (hpq hp)
```

link

Writing theorems and proofs in Lean

The previous examples are simple enough that we could write a proof in one line. In order to write and maintain longer proofs, Lean allows one to write proofs in the **tactic mode**.

```
example (p q r : Prop) (hp : p) (hpq : p → q)
    (hqr : q → r) : r := by
  apply hqr
  apply hpq
  exact hp
```

link

Read: to prove `r`, as we assume `q → r`, it is enough to prove `q` (the first "apply").

To prove `q`, as we assume `p → q`, it is enough to prove `p` (the second "apply").

To prove `p`, as we assume `p`, it is exactly the assumption (the "exact").

Tactic

- exact
- apply
- rw
- intro
- ...

Many many more tactics, and people are adding more powerful tactics.
This makes Lean an "interacting" theorem prover:

- You write the statement
- Lean tells you what to prove.
- You propose using some tactics.
- Lean gives you the remaining "goals".

In this course, we will see some of the most useful ones.

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  (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 $+_{\texttt{My}}$ inductively:

- $m +_{\texttt{My}} 0 := m$
- $m +_{\texttt{My}} (n + 1) := (m +_{\texttt{My}} n) +_{\texttt{My}} 1$

Can we prove $0 +_{\texttt{My}} n = n +_{\texttt{My}} 0$?

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
```

Dependent type theory

In Lean, functions from a type α to another β have the type $\alpha \to \beta$.

- `succ : Nat → Nat` (mathematically, $n \mapsto n + 1$).
- `add : Nat → (Nat → Nat)` (mathematically, $(m, n) \mapsto m + n$).
- `Funcion.comp succ succ : Nat → Nat`
  $(n \mapsto (n + 1) + 1 = n + 2)$.

Here, `Function.comp` takes two parameters `f : β → δ, g : α → β` and gives `g ∘ f : α → δ`.
The resulting type of `Function.comp f g` **depends on the types** of `f` and `g`. Allowing this makes Lean *dependent* type theory.

Dependent type theory

One can define a function explicitly using `fun`:

- `fun (n : Nat) => succ (succ n)` defines the function $n \mapsto n + 1 + 1$.
- `fun (n : Nat) => MyAdd n n` defines the function $n \mapsto n + n$.

One can give name to a function using `def`. One can either specify the type, or let Lean guess the type, or write the argument as a parameter.

```
def MyAddTwo : Nat → Nat :=
    fun (n : Nat) => succ (succ n)
#check MyAddTwo

def MyAddTwo' := fun (n : Nat) => succ (succ n)
#check MyAddTwo'

def MyAddTwo'' (n : Nat) := succ (succ n)
#check MyAddTwo''
```

Sets in Lean

Let `X` be a type.

- In Lean, the type `Set X` is defined as `X → Prop`.
- This means that, if `A : Set X`, then for each `x : X`, one has `A x : Prop`.
- The idea:
  - if `A x` is true, then this should correspond to `x ∈ A`
  - if `A x` is false, then this should correspond to `x ∉ A`
- `x ∈ A : Prop` is *defined to be* `A x`

Sets in Lean

- Subset. `A ⊂ B : Prop` is defined as `∀ x : X, x ∈ A → x ∈ B`.
- Union. `A ∪ B : Set X` is defined as `fun x => A x ∨ B x`.
- Intersection. `A ∩ B : Set X` is defined as `fun x => A x ∧ B x`.
- Complement. `Aᶜ : Set X` is defined as `fun x => ¬ A x`.

Practice

- Set theory game