Lean basics

Patrick Massot

Université Paris-Saclay at Orsay

July 14, 2020

A word about foundations

Key point: we don't want to see foundations

A word about foundations

Key point: we don't want to see foundations

But the proof assistant needs foundations

A word about foundations

Key point: we don't want to see foundations

But the proof assistant needs foundations

Standard lie: maths is founded on 1st order logic + ZFC set theory
In such foundations, *everything* is a set: $\mathbb{N}$, $\exp$,
a group structure on a set is a set...

A word about foundations

Key point: we don't want to see foundations

But the proof assistant needs foundations

Standard lie: maths is founded on 1st order logic + ZFC set theory
In such foundations, *everything* is a set: $\mathbb{N}$, $\exp$,
a group structure on a set is a set...

Let's define the set $\mathbb{N}$: $0 := \varnothing$, $1 := 0 \cup \{0\} = \{\varnothing\}$,
$2 := 1 \cup \{1\} = \{\varnothing, \{\varnothing\}\}$, $3 := 2 \cup \{2\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$.

A word about foundations

Key point: we don't want to see foundations

But the proof assistant needs foundations

Standard lie: maths is founded on 1st order logic + ZFC set theory
In such foundations, *everything* is a set: $\mathbb{N}$, $\exp$,
a group structure on a set is a set...

Let's define the set $\mathbb{N}$: $0 := \varnothing$, $1 := 0 \cup \{0\} = \{\varnothing\}$,
$2 := 1 \cup \{1\} = \{\varnothing, \{\varnothing\}\}$, $3 := 2 \cup \{2\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$.

Exercise: 3 is a topology on 2. Note also how $2 \cap 3 = 2$ and $2 \in 3$.

Avoiding those non-sensical statements rely on a gentleman agreement.

Type theory

Every meaningful piece of math has a type:

- $2 : \mathbb{N}$,
- $\exp : \mathbb{R} \to \mathbb{R}$,
- $x \mapsto x \exp(x) : \mathbb{R} \to \mathbb{R}$
- $1 + 1 = 2 : \mathrm{Prop}$

Things like $2 \cap 3$ or $2 \in 3$ do *not* "type-check" (they have no type).

Type theory

Every meaningful piece of math has a type:

- $2 : \mathbb{N}$,
- $\exp : \mathbb{R} \to \mathbb{R}$,
- $x \mapsto x \exp(x) : \mathbb{R} \to \mathbb{R}$
- $1 + 1 = 2 : \mathrm{Prop}$

Things like $2 \cap 3$ or $2 \in 3$ do *not* "type-check" (they have no type).

Pieces of math are called *terms*.
You can ask Lean about the type of a term using **#check**.

Type theory

Every meaningful piece of math has a type:

- $2 : \mathbb{N}$,
- $\exp : \mathbb{R} \to \mathbb{R}$,
- $x \mapsto x \exp(x) : \mathbb{R} \to \mathbb{R}$
- $1 + 1 = 2 : \mathrm{Prop}$

Things like $2 \cap 3$ or $2 \in 3$ do *not* “type-check” (they have no type).

Pieces of math are called *terms*.
You can ask Lean about the type of a term using **#check**.

$x \mapsto x \exp(x) : \mathbb{R} \to \mathbb{R}$ is written as $\lambda\, x, x \exp(x) : \mathbb{R} \to \mathbb{R}$

Type theory

Every meaningful piece of math has a type:

- $2 : \mathbb{N}$,
- $\exp : \mathbb{R} \to \mathbb{R}$,
- $x \mapsto x \exp(x) : \mathbb{R} \to \mathbb{R}$
- $1 + 1 = 2 : \mathrm{Prop}$

Things like $2 \cap 3$ or $2 \in 3$ do *not* "type-check" (they have no type).

Pieces of math are called *terms*.
You can ask Lean about the type of a term using **#check**.

$x \mapsto x \exp(x) : \mathbb{R} \to \mathbb{R}$ is written as $\lambda\, x, x \exp(x) : \mathbb{R} \to \mathbb{R}$

Typing rules are things like: given $f : X \to Y$ and $x : X$, deduce $f\ x : Y$.

Meta-theory vs theory

☡ The assertion $x : t$ that a term x has type t, and typing rules, are *not* something you can prove or disprove inside the theory. They live one level up, in the meta-theoretical world, just as you don't prove the properties of logical operators while working inside ZFC.

Meta-theory vs theory

☡ The assertion $x : t$ that a term x has type t, and typing rules, are *not* something you can prove or disprove inside the theory. They live one level up, in the meta-theoretical world, just as you don't prove the properties of logical operators while working inside ZFC.

☡ computer scientists and logicians use the prefix "meta" whenever something is unusual. It can be used three times in the same sentence with three different meanings.

Conversion rules

At the meta-theory level also live the "conversion rules" that assert some term are so-called *definitionaly* equal.
For instance:

- $\lambda x : \mathbb{N}, x + 2 \equiv \lambda y : \mathbb{N}, y + 2$ by the α-conversion rule.

Conversion rules

At the meta-theory level also live the “conversion rules” that assert some term are so-called *definitionaly* equal.
For instance:

- $\lambda x : \mathbb{N}, x + 2 \equiv \lambda y : \mathbb{N}, y + 2$ by the α-conversion rule.
- $(\lambda x : \mathbb{N}, x + 2)\, 3 \equiv 3 + 2$ by the β-conversion rule.

Conversion rules

At the meta-theory level also live the "conversion rules" that assert some term are so-called *definitionaly* equal.
For instance:

- $\lambda x : \mathbb{N}, x + 2 \equiv \lambda y : \mathbb{N}, y + 2$ by the α-conversion rule.
- $(\lambda x : \mathbb{N}, x + 2)\ 3 \equiv 3 + 2$ by the β-conversion rule.
- By repeated δ-conversion:

$$\begin{aligned} 3 + 2 &\equiv S(S(S(0))) + S(S(0)) \\ &\equiv S(S(S(S(0))) + S(0)) \\ &\equiv S(S(S(S(S(0))) + 0) \\ &\equiv S(S(S(S(S(0))))) \\ &\equiv 5 \end{aligned}$$

Proofs in type theory

Given $P : \mathrm{Prop}$, a term $h : P$ is a *proof* of P.

Proofs in type theory

Given $P : \mathrm{Prop}$, a term $h : P$ is a *proof* of P.

The statement $P \implies Q$ is the function type $P \to Q : Prop$.
By the function typing rule, if $h : P \to Q$ and $h_P : P$ then $h\ h_P : Q$.

Proofs in type theory

Given $P : \mathrm{Prop}$, a term $h : P$ is a *proof* of P.

The statement $P \implies Q$ is the function type $P \to Q : Prop$. By the function typing rule, if $h : P \to Q$ and $h_P : P$ then $h\ h_P : Q$.

Given $P : X \to \mathrm{Prop}$, we get $\forall x, P\ x : \mathrm{Prop}$.

Proofs in type theory

Given $P : \mathrm{Prop}$, a term $h : P$ is a *proof* of P.

The statement $P \implies Q$ is the function type $P \to Q : Prop$.
By the function typing rule, if $h : P \to Q$ and $h_P : P$ then $h\ h_P : Q$.

Given $P : X \to \mathrm{Prop}$, we get $\forall x, P\ x : \mathrm{Prop}$.

Given $h : \forall x, P\ x$ and $x_0 : X$, we get $h\ x_0 : P\ x_0$.

So h behaves like a kind of function, but its target type depends on the value of its input. We have a *dependent function type*.

Proofs in type theory

Given $P : \mathrm{Prop}$, a term $h : P$ is a *proof* of P.

The statement $P \implies Q$ is the function type $P \to Q : Prop$. By the function typing rule, if $h : P \to Q$ and $h_P : P$ then $h\ h_P : Q$.

Given $P : X \to \mathrm{Prop}$, we get $\forall x, P\ x : \mathrm{Prop}$.

Given $h : \forall x, P\ x$ and $x_0 : X$, we get $h\ x_0 : P\ x_0$.

So h behaves like a kind of function, but its target type depends on the value of its input. We have a *dependent function type*.

Verifying a proof is a special case of type-checking a term.

Inductive types

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

```
inductive or (a b : Prop) : Prop
| inl (h : a) : or
| inr (h : b) : or
```

```
inductive Exists {α : Sort u} (p : α → Prop) : Prop
| intro (w : α) (h : p w) : Exists
```

Elaboration

```
theorem infinitude_of_primes : ∀ N, ∃ p ≥ N, nat.prime p
```

Lean needs the types of N and p and an order relation.

Elaboration

```
theorem infinitude_of_primes : ∀ N, ∃ p ≥ N, nat.prime p
```

Lean needs the types of N and p and an order relation.

It goes from left to right, inserting holes (meta-variables) when needed.

- $N : ?m_1$
- $p : ?m_2$
- see $p \geq N$, deduce $?m_1 = ?m_2$, take note we'll need an order relation on $?m_1$
- see `nat.prime p` which makes sense only if $p : \mathbb{N}$
- look up a database of order relation to get one for $\mathbb{N}$

Coercion

During elaboration, when cornered, Lean will try to find a *coercion*.

For instance, in `∀ x : ℝ, ∀ ε > 0, ∃ n : ℕ, x ≤ n*ε`

Coercion

During elaboration, when cornered, Lean will try to find a *coercion*.

For instance, in `∀ x : ℝ, ∀ ε > 0, ∃ n : ℕ, x ≤ n*ε`

One can encourage Lean to insert coercion by writing *type ascriptions*, as in `1/(n+1 : ℝ)`