Type theory in Lean

Riccardo Brasca

Université Paris Cité

July 5, 2025

Introduction

One of the main goals of a proof assistant is to verify exactness of mathematical proofs, starting from the axioms.

One has to choose an axiomatic system.

Usually, mathematicians pick Zermelo-Fraenkel set theory plus the axiom of choice.

Type theory

Type theory is an alternative to set theory as a foundation of mathematics.

It has been introduced by Bertrand Russel around 1910.

Type theory it is its own deductive system (it does not depend on first order logic).

Sets and $\in$ play no special role, they are mathematical notions with a "normal" definition.

Terms, types and universes

In type theory there are three layer of mathematical objects:

- terms;
- types;
- universes.

(Almost) everything is a term.

The natural number 0 is a term. So is the the function $\sin : \mathbb{R} \to \mathbb{R}$ and the "set" $\mathbb{Z}$.

$\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ are types (but also terms!).

Slogan

Every term x has its own type T, written

$$(x : T).$$

In a sense, the type of x is its own nature. The fact that the type of x is T is not a (true/false) mathematical statement. We cannot prove or disprove it, we can only *check* the type of a term.

$\mathbb{N}$ is a type, but also a term. Its type is Type.

$$(\mathbb{N} : \mathrm{Type}) \quad (\mathbb{R} : \mathrm{Type}).$$

Type is also a term! It has type Type 1.

$$(\mathrm{Type} : \mathrm{Type}\ 1).$$

Type and $\mathrm{Type}\ 1$ are *universes*.
Lean has a countable non-cumulative hierarchy of universes.

$$\mathrm{Type}\ 0 = \mathrm{Type}$$

$$\mathrm{Type}\ 1$$

$$\mathrm{Type}\ 2$$

And so on. In general

$$\mathrm{Type}\ n : \mathrm{Type}\ n + 1$$

At the very bottom there is a special universe: $\mathrm{Prop} : \mathrm{Type}$.

Definitional equality

In Lean there are several notion of equality:

- Syntactic equality: two literally equal expressions ("written using the same keys on the keyboard").
- Mathematical equality (more on this later).
- Definitional equality, denoted today $\equiv$.

Two terms are definitionally equal if they are equal "by definition".

$$x \equiv y$$

is *not* a mathematical statement, but it can be checked in practice.

There is a precise set of rules to check whether $x \equiv y$. For example:

- syntactic equality implies definitional equality;
- the functions $x \mapsto \sin(x)$ and $y \mapsto \sin(y)$ are definitionally equal;
- if f is the function $x \mapsto \sin(x)$, then $f(\pi) \equiv \sin(\pi)$;
- if $(x : T)$ and $(y : S)$ are definitionally equal, then $T \equiv S$;
- ...

In principle definitional equality is algorithmically decidable.

Slogan

Two terms x and y are definitionally equal if the following proof works.

```
example : x = y := rfl
```

This is *not* an equivalence relation, but the "true" definitional equality is.

How to build new types

How to build new types out of old ones?

We fix a universe u. For today's lecture, almost all of the types will be of type $\mathrm{Type}\ u$.

One can be more general, allowing any sort (in particular also Prop).

We will study in details two constructions:

- Dependent functions.
- Dependent pairs.

General rules

For each construction will will follow the same pattern.

- *Formation rules*: when we can form a new type using the construction.
- *Constructors* or *formation rules*: how to build terms of the new type. In particular how to build functions into the new type.
- *Eliminators* or *elimination rules*: how to use terms of the new type. In particular how to build functions out of the new type.

- *Computation rules*: definitional equalities about the application of the eliminators to the constructors.

- An optional *uniqueness principle*: roughly speaking it is the fact that the only terms of the new type are those obtained using the constructors. It is optional, and often it can be proved using the previous rules. In this case it is a design choice to make it a definitional equality (one cannot prove definitional equalities!).

Functions types

The most basic construction we will consider is the type of functions between two given types.

- Formation rule: if A and B are two types, we have another type $A \to B$, whose terms are called *functions* from A to B.
- Constructors: there is only one constructor, called *lambda abstraction*. If E is any expression containing a variable x such that $(E : B)$ if $(x : A)$, then

$$(\text{fun } x \mapsto E : A \to B)$$

is of type $A \to B$.

- Eliminators: there is only one eliminator. If

$$(f : A \to B) \text{ and } (a : A),$$

then we have a well defined term $(f\ a : B)$.

- Computation rules: there is only one computation rule, saying that, if E is as above and $(a : A)$, then

$$(\text{fun } x \mapsto E)\, a \equiv E[x := a].$$

Here $E[x := a]$ is the expression E with each x replaced by a (syntactically).

- Uniqueness principle: if $(f : A \to B)$, then

$$f \equiv \text{fun } x \mapsto f\ x,$$

so all functions can be obtained via lambda abstraction.

Note that, contrary to ZFC, $A \to B$ is a primitive notion.

At the moment there is no *functional extensionality principle*: if $f\ x = g\ x$ for all x, we cannot prove that $f = g$. The uniqueness principle implies that if $f\ x \equiv g\ x$ for all x, then $f \equiv g$, but this is difficult to state in Lean.

Dependent functions

Consider the assignment

$$n \mapsto 0 \in \mathbb{R}^n,$$

where n is a natural number. It looks like a function with domain $\mathbb{N}$. What is its codomain?

We want $(f\ n : \mathbb{R}^n)$ for all $(n : \mathbb{N})$, but $(f : \mathbb{N} \to ?)$.

f is *not* a function as above. The type of $f\ n$ depends on n.

The type of f will be a $\prod$-type, and f will be a *dependent function*.

The idea is that f is like an ordinary function, but the type of $f\ x$ can depend on x.

If $(f\ x : B)$ for all $(x : A)$ (i.e. the type of $f\ x$ is constant), then f is an ordinary function.

The real primitive construction is the Π-type, with functions as a special case.

Example

If $f : A \times B \to C$ is a function, let's define

$$\text{swap } f : B \times A \to C$$
$$(b, a) \mapsto f(a, b)$$

If we think to f as a term $(f : A \to B \to C)$, then $(\text{swap } f : B \to A \to C)$ and

$$\text{swap} : (A : \text{Type } u) \to (B : \text{Type } u) \to$$
$$(A \to B \to C) \to (B \to A \to C)$$

Cartesian product

We now move on to the Cartesian product.

- Formation rule: if A and B are two types, we have another type, denoted $A \times B$, whose terms are called *pairs* of elements of A and B.

- Constructors: there is only one constructor. If we have two terms $(a : A)$ and $(b : B)$, then we have a term, denoted (a, b) or $\langle a, b \rangle$, of type $A \times B$.

$$((a, b) : A \times B)$$

- Eliminators: there is only one eliminator. Given $(x : A \times B)$ and a function $f : A \to B \to C$, we have a well defined term $(\mathrm{rec}_{A \times B}\ f\ x : C)$. This gives a function $(\mathrm{rec}_{A \times B}\ f : A \times B \to C)$.

- Computation rules: there is only one computation rule. If we have terms $(a : A)$ and $(b : B)$ and a function $(f : A \to B \to C)$, then

$$\mathrm{rec}_{A \times B}\ f\ (a, b) \equiv f\ a\ b.$$

  Note that $((a, b) : A \times B)$ is the term given by the constructor.

The name $\mathrm{rec}_{A \times B}$ comes from the theory of inductive types.

Definition

We let $\pi_1 : A \times B \to A$ be the function given by the eliminator via

$$\pi_1 = \mathrm{rec}_{A \times B}\left((\mathrm{fun}\ a\ b \mapsto a) : A \to B \to A\right)$$

The computation rule says that

$$\pi_1\ (a, b) \equiv a.$$

In Lean we write $x.1$ for $\pi_1\ x$. The function π_1 is also called fst.
We similarly have a function $\pi_2 \colon A \times B \to B$ also called snd.

- Uniqueness principle: if $(x : A \times B)$, then

$$x \equiv (\pi_1\ x, \pi_2\ x),$$

so all terms of $A \times B$ are given by pair of elements. The above equality is provable using the computation rule, but we assume it as a definitional equality. This implies that

$$\mathrm{rec}_{A \times B}\ f \equiv \mathrm{fun}\ (x : A \times B) \mapsto f\ x.1\ x.2$$

for all $(f : A \to B \to C)$.

The Cartesian product is not a primitive notion in Lean's type theory: it is a special case of an inductive type.

The uniqueness principle implies the following, for all functions $(f : A \times B \to C)$.

$$f \equiv \text{fun } (x : A \times B) \mapsto f\ (x.1, x.2)$$

So every function $f : A \times B \to C$ can be defined using π_1 and π_2.

In practice we always use this observation to build a function $A \times B \to C$.

Universal constructions

The eliminator allows to construct a function $A \times B \to C$ given a function $A \to B \to C$.

In practice it is given by a (dependent!) function $\mathrm{rec}_{A \times B}$ of type

$$\left(\mathrm{rec}_{A \times B} : \prod_{(C : \mathrm{Type}\ u)} (A \to B \to C) \to A \times B \to C \right)$$

The computation rule says

$$\mathrm{rec}_{A \times B}\ f\ (a, b) \equiv f\ a\ b$$

We have

$$\pi_1 \equiv \mathrm{rec}_{A \times B}\ (\mathrm{fun}\ a\ b \mapsto a)$$

Dependent pair types

Dependent pair types (also called $\sum$-types) generalize Cartesian product in the same way as dependent functions generalize function types: we allow the type of the second component to depend on the first one.

- Formation rule: if A is a type and $(B : A \to \mathrm{Type}\ u)$ is a function, we have another type, denoted

$$\sum_{(a:A)} B\ a = (a : A) \times B\ a$$

whose terms are called *dependent pairs*.

The universe Prop

Recall that *everything* is a term and has its own type.

Even mathematical statement.

$$(2 + 2 = 4 : \mathrm{Prop}) \text{ and } (1 < 0 : \mathrm{Prop})$$

Prop is a type, like $\mathbb{N}$:

$$(\mathrm{Prop} \ : \mathrm{Type})$$

Remark

Defining a proposition doesn't mean to prove it:

$$(\forall\ (n\,x\,y\,z : \mathbb{N}),\ n > 2 \Rightarrow x^n + y^n = z^n \Rightarrow xyz = 0\ : \mathrm{Prop})$$

is the statement of Fermat's last theorem. Constructing such a term is trivial. Indeed, a proposition can be true or false mathematically, for Lean they are just terms with type Prop.

Everything is a term... what about proofs? They are also terms! Of which type?

If $(P : \mathrm{Prop})$ is a mathematical statement, then a proof p of P is a term of type P:

$$(p : P)$$

In particular, if $(P : \mathrm{Prop})$, then P is also a well formed type.

Remark

Usually we think to types as sets. This is pretty accurate for $\mathbb{N}$ *or* $\mathbb{R}$*, but it is completely misleading for a proposition* P*.*

Prop *behaves like a set (the set of mathematical statements). On the other hand, the type given by one single statement, even if it is a type, does not behave like a set.*

If u *is a universe and* $(T : \mathrm{Type\ u})$ *then it is safe to think to* T *as a set. All types* T *are terms of type* $\mathrm{Type\ u}$ *for some* u*, except mathematical statements, that have type* $\mathrm{Prop} = \mathrm{Sort}\ 0$*.*
(Remember that in general $\mathrm{Type}\ u = \mathrm{Sort}\ u + 1$*.)*

A term p of type $(p : 2 + 2 = 4)$ is *a proof* that $2 + 2 = 4$.

A term p of type $(p : 2 + 2 = 5)$ would be a proof that $2 + 2 = 5$. Hopefully it is impossible to construct such a term.

A proposition P is provable if there is a term $(p : P)$ of type P, so if there is a proof of P.

When we write

```
theorem easy : 1 + 1 = 2 := by ...
```

Lean checks that the type of the term `easy` (defined after the $:=$) is $1 + 1 = 2$, so that `easy` is *a proof* that $1 + 1 = 2$.

To Lean, this is the same as checking that π has type $\mathbb{R}$, but we think to easy as a *witness that* $1 + 1 = 2$ *holds*.

In the statement of Fermat's last theorem

```
theorem FLT (n x y z : ℕ) (hn : n > 2)
  (H : x ^ n + y ^ n = z ^ n) : x * y * z = 0
:= by ...
```

n, x, y, z are of type $\mathbb{N}$, where `hn` and `H` are of type $n > 2$ and $x^n + y^n = z^n$ respectively. They are proofs of two propositions.

The term `FLT n x y z hn H` would be of type `x * y * z = 0`, so it would be a proof that $xyz = 0$ given n, x, y, z, hn, H. Since $n > 2$ and $x^n + y^n = z^n$ are propositions, we can think to hn and H as proofs of those propositions, that we suppose to have at our disposal, and in practice we can think that n, x, y, z satisfy $n > 2$ and $x^n + y^n = z^n$.

At the end, we need to construct a proof of $xyz = 0$ being given four natural numbers n, x, y, z that satisfy $n > 2$ and $x^n + y^n = z^n$, so we need to prove Fermat's last theorem in the usual sense.

How to build proofs?

How can we build a proof of $(P : \mathrm{Prop})$?

First of all we have to build a well defined proposition.

Remember that any $(P : \mathrm{Prop})$ is itself a type... to build P we can generalize the construction above.

A proposition P obtained by (a generalization of) the construction above will have constructors, that allows us to build terms p of type $(t : P)$ i.e. proofs of P.

Back to functions types

Remember that if $(A\ B : \mathrm{Type}\ u)$ for some universe u, then we have the type $(A \to B : \mathrm{Type}\ u)$

More generally, if $(A : \mathrm{Type}\ u)$ and $(B : \mathrm{Type}\ v)$, then we have

$$(A \to B : \mathrm{Type}\ \max u\ v)$$

Even more generally, if $(A : \mathrm{Type}\ u)$ and $(B : A \to \mathrm{Type}\ v)$, then we have

$$\left(\prod_{(a:A)} B\ a : \mathrm{Type}\ \max u\ v\right)$$

We want to generalize this to arbitrary sorts (i.e. $\mathrm{Sort}\ u$ instead of $\mathrm{Type}\ u$). The only remaining case is that of a proposition.

Let $(A : \mathrm{Type}\ u)$ be a type and let $(P : A \to \mathrm{Prop})$ be a function. A dependent function

$$\left(f : \prod_{(a:A)} P\ a \right)$$

is the datum of a term $(f\ a : P\ a)$ for all $(a : A)$. But $(P\ a : \mathrm{Prop})$ is a proposition, so $f\ a$ is a proof of $P\ a$. In practice, giving f is the same as giving a proof of $P\ a$ for all $(a : A)$!

Example

Proving that $\forall (n : \mathbb{N}), n + 0 = n$ means constructing a term of type

$$\prod_{(n:\mathbb{N})} n + 0 = n$$

In lean, the $\forall$ symbol if *defined* as a synonym of a $\prod$-type.

Slogan

Constructing a term $(f : \prod_{(a:A)} P\ a)$ *is the same as proving* $P\ a$ *for all* $(a : A)$.

In particular, it is reasonable to have $(\prod_{(a:A)} P\ a : \mathrm{Prop})$.

The general rule is as follows. Let $(A : \mathrm{Sort}\ u)$ and $(B : A \to \mathrm{Sort}\ v)$. Then

$$\left(\prod_{(a:A)} B\ a : \mathrm{Sort}\ \mathrm{imax}\ u\ v \right) \text{ where}$$

$\mathrm{imax}\ u\ 0 = 0$ and $\mathrm{imax}\ u\ v = \max u\ v$ if $v \neq 0$.

We say that Prop is *impredicative*.

Implication

It follows that if $(A : \mathrm{Sort}\ u)$ and $(P : \mathrm{Prop})$, then $(A \to P : \mathrm{Prop})$. In particular, if $(Q : \mathrm{Prop})$ is also a proposition, then

$$(P \to Q : \mathrm{Prop})$$

To specify a term of type $(f : P \to Q)$ we have to specify a term $(f\ p : Q)$ for all $(p : P)$, so we need to prove Q given a proof of P.

Slogan

Constructing a term of type $P \to Q$ is the same as proving that P implies Q.

Modus pones

Modus pones is the rule of inference that says that if P holds and P implies Q, then Q holds.

This means that if we have a proof of P, so a term $(p : P)$, and a term $(h : P \to Q)$, then we have a term of type Q. This is simply given by $h\ p$ using the eliminator of $P \to Q$!

```
example (p : P) (h : P → Q) : Q := by
  exact h p
example (p : P) (h : P → Q) : Q := h p
```

The logic operators

We have introduced the first logic operator, implication $P \to Q$.

We now move on to the operators *and*, *if and only if*, *or* and *negation*.

The first three are special cases of an inductive type (an inductive proposition in this case), so we will follow the same pattern as above, giving the introduction rule, constructors, eliminators...

To define negation we will define two particular propositions, $\mathrm{True} : \mathrm{Prop}$ and $\mathrm{False} : \mathrm{Prop}$, again as inductive propositions.

Conjunction

We start with the *and* operator.

- Formation rule: if P and Q are two propositions, we have another proposition

$$(P \wedge Q : \mathrm{Prop}),$$

  called the *conjunction* of P and Q.
- Constructors: there is only one constructor. If $(p : P)$ and $(q : Q)$, then

$$(\langle p, q \rangle : P \wedge Q)$$

- Eliminators (non-dependent version): there is only one eliminator. Given a function $(f : P \to Q \to A)$, where $(A : \mathrm{Sort}\ u)$, we have a function

$$(\mathrm{And.elim}\ f : P \wedge Q \to A)$$

- Computation rules: there is only one computation rule, saying that, if A is as above, then

$$\mathrm{And.elim}\ f \langle p, q \rangle \equiv f\ p\ q$$

for all $(p : P)$ and $(q : Q)$.

As in the case of the Cartesian product, if $(t : P \wedge Q)$, we have $(t.1 : P)$ and $(t.2 : Q)$.

- Uniqueness principle: for all $(t : P \wedge Q)$ we have

$$t \equiv \langle t.1, t.2 \rangle$$

We will see later that the uniqueness is in reality useless.

All the remarks we made for the Cartesian product hold true.

In practice, to prove $P \wedge Q$ we need to prove P and to prove Q, and if we have a proof t of $P \wedge Q$ we have a proof $t.1$ of P and a proof $t.2$ of Q.

Remark

In Lean, to build a term of an inductive type with only one constructor, we can use the `constructor` *tactic. For example, if the goal is $P \wedge Q$, after* `constructor`*, we will have two goals, one of type P and one of type Q. This includes all the constructions we saw so far, except functions types, that are not inductive types.*

Disjunction

Let's define the *or* operator.

- Formation rule: if P and Q are two propositions, we have another proposition

$$(P \vee Q : \mathrm{Prop}),$$

  called the *disjunction* of P and Q.
- Constructors: there are two constructors. If $(p : P)$, then

$$(\mathrm{Or.intro\_left}\ Q\ p : P \vee Q)$$

  and, if $(q : Q)$, then

$$(\mathrm{Or.intro\_right}\ P\ q : P \vee Q)$$

- Eliminators: there is only one eliminator. Given two functions $(f : P \to R)$ and $(g : Q \to R)$, where $(R : \mathrm{Prop})$, and a term $(t : P \vee Q)$, we have a term

$$(\mathrm{Or.elim}\ t\ f\ g : R)$$

- Computation rules and uniqueness principle: one can guess that there are two computations rules, saying that, if R is as above, then

$$\mathrm{Or.elim}\ (\mathrm{Or.intro\_left}\ Q\ p)\ f\ g \equiv f\ p$$
$$\mathrm{Or.elim}\ (\mathrm{Or.intro\_right}\ P\ q)\ f\ g \equiv g\ q$$

for all $(p : P)$ and $(q : Q)$. This is true (definitionally!), but in reality these rules are useless (more on this later).

In practice, to prove $P \vee Q$ we can prove P or we can prove Q, using the two constructors. In Lean we can use the `left` and the `right` tactics respectively.

Suppose we have a proof $(t : P \vee Q)$ and wants to prove $(R : \mathrm{Prop})$, so we have to build a function $P \vee Q \to R$. Using the eliminator, we have to build two functions $P \to R$ and $Q \to R$, so in practice we have to prove that P implies R and that Q implies R. In other words, we have to prove that R holds in both cases where P or Q holds, as expected.

Remark

In Lean, to use a term of an inductive type with several constructors, we can use the `cases` *or the* `rcases` *tactics. For example, if the assumption is* $(t : P \vee Q)$*, after* `cases t`*, we will have two goals, one assuming* P *holds and one assuming* Q *holds. (We will see the precise syntax in the examples.)*

The True proposition

We now define $(\mathrm{True} : \mathrm{Prop})$, a special proposition that models the True truth value. It is an inductive proposition as those defined above.

- Formation rule: we simply have a proposition

$$(\mathrm{True} : \mathrm{Prop})$$

- Constructors: there is only one constructor. we can produce a term

$$(p : \mathrm{True})$$

for free. In Lean, if we need to produce a term of type True we can use the `trivial` tactic.

In practice, to prove True, we have nothing to do.

- Eliminators: there is only one eliminator. Given a term $(a : A)$, where $(A : \mathrm{Sort}\ u)$ and $(p : \mathrm{True})$, we have a term $(a : A)$. In practice, the datum $(p : \mathrm{True})$ (i.e. the fact that True holds) doesn't help.
- Computation rules: there is only one computation rule, that says that the element constructed using $(a : A)$ is definitionally equal to itself.
- There is no uniqueness principle. To be precise, there is no need to assume any uniqueness principle.

The False proposition

We now define $(\mathrm{False} : \mathrm{Prop})$, a special proposition that models the False truth value. It is again an inductive proposition.

- Formation rule: we simply have a proposition

$$(\mathrm{False} : \mathrm{Prop})$$

- Constructors: there are no constructor. In practice, it is impossible to prove False.

- Eliminators: there is only one eliminator. Given $(p : \mathrm{False})$, we have a term

$$(\mathrm{False.elim}\ p : A)$$

for any sort $(A : \mathrm{Sort}\ u)$. In practice, the datum $(p : \mathrm{False})$ (i.e. the fact that False holds) allows to produce a term of any given type. In Lean, we can use the `exfalso` tactic to turn any goal into a proof of False.

Remark

The idea is that, given $(p : \mathrm{False})$*, to build* $(a : A)$*, we have to do so in all the cases p can be obtained, via the constructors. But there are no constructors, so there is nothing to do. In Lean,* `cases p` *closes any goal.*

Negation

Definition

Given a proposition $(P : \mathrm{Prop})$, we define its *negation*, denoted $\neg P$ as

$$P \to \mathrm{False}$$

Remark

This means that $\neg P$ is an implication. To prove it, we assume that P holds and the we need do to prove False. In practice we start with `intro p` *and we get a goal of type False. This is not a proof by contradiction.*

The existential quantifier

The last logic operator we need to introduce is the existential quantifier.

Remark

A natural approach to define the statement $\exists\,(a : A),\ P\ a$, *where* $(P : A \to \mathrm{Prop})$, *is to define it as a term of type*

$$\left(t : \sum_{(a:A)} P\ a \right)$$

While is it possible to generalize the dependent pair construction to work with a function to Prop, *it is not possible to make it taking value itself in* Prop.

We will introduce it as an inductive proposition, giving the usual rules.

- Formation rule: if $(A : \mathrm{Sort}\ u)$ is any sort and $P : A \to \mathrm{Prop}$ is a function, we have another proposition

$$(\exists\ (a : A),\ P\ a : \mathrm{Prop})$$

- Constructors: there is only one constructor. If $(a : A)$ and $(h : P\ a)$ (so a is a term such that $P\ a$ holds), then

$$(\langle a, h\rangle : \exists\ (a : A),\ P\ a)$$

- Eliminators: there is only one eliminator. Given $(h_1 : \exists\ (a : A),\ P\ a)$ and $(h_2 : \forall\ (a : A), P\ a \to Q)$, where $(Q : \mathrm{Prop})$ we have a term

$$(\mathrm{Exists.elim}\ h_1\ h_2 : Q)$$

  In practice, the datum $(h_2 : \forall\ (a : A), P\ a \to Q)$ means that $P\ a$ implies Q (that is a fixed proposition, not depending on a) for all a, while $(h_1 : \exists\ (a : A),\ P\ a)$ means that there is a term $(a : A)$ such that $P\ a$. We conclude that Q holds.

There is no need for computation rules or uniqueness principle.

Remark

Note that $\forall\ (a : A), P\ a \to Q$ *is the same as (by definition of* $\forall$*!)* $\prod_{(a:A)} P\ a \to Q$*, so the eliminator is really the analogue of the non-dependent eliminator for the dependent pair type, but note that* Q *is restricted to be a* Prop *(while for the dependent pair type it can be any sort). Technically there is also a dependent version, but it is uninteresting.*

Remark

In Lean, to prove $\exists\ (a : A),\ P\ a$*, we can use the* `use` *tactic. It will produce two goals, the first one will be a term* a *of type* A*, and the second one will be that* $P\ a$ *holds.*

Remark

In Lean, while proving any proposition, if we have $(h_1 : \exists\, (a : A),\ P\ a)$ *we can use the* `obtain` *tactic to get* $(a : A)$ *and the hypothesis that* $P\ a$ *holds (we will see the precise syntax in the examples). This is a direct application of the eliminator above, so it can used only in while proving a proposition (and not while, say, constructing a natural number).*

The idea is that h_1 *only "knows" the existence of some* a*, not the precise value of such an* a*. In particular we can not use this knowledge to define a natural number, since the definition could depend on which* a *we use.*

The natural numbers

We have seen several inductive types and inductive propositions.

We move on to the main example of an inductive type: the natural numbers.

We follow the usual pattern.

- Formation rule: there is a well formed type $\mathbb{N}$.

$$(\mathbb{N} : \mathrm{Type})$$

  Its terms are called *natural numbers*.

Constructors

There are two constructors. First of all we have a natural number called *zero* and denoted 0:

$$(0 : \mathbb{N})$$

Moreover, if $(n : \mathbb{N})$ is a natural number, we have another natural number called *the successor of n* and denoted $\mathrm{succ}\ n$:

$$(\mathrm{succ}\ n : \mathbb{N})$$

In particular, we have a function

$$\mathrm{succ} : \mathbb{N} \to \mathbb{N}$$

The fact that succ takes a natural number and gives another natural number is what makes $\mathbb{N}$ an *inductive* type.

Eliminator

Let's start with the most general version, we will deduce various special cases later on.

Recall that the eliminator allows to define dependent functions out of the given type.

Let u be a universe and let $(M : \mathbb{N} \to \mathrm{Sort}\ u)$ be a function. M will give the codomain of the dependent function we want to define, in Lean it is usually called the *motive*.

We want to define a term

$$\left(f : \prod_{(n:\mathbb{N})} M\ n \right)$$

Given a term $(z : M\ 0)$ and a dependent function

$$\left(s : \prod_{(n:\mathbb{N})} (M\ n \to M\ (\mathrm{succ}\ n)) \right),$$

we get a function

$$\left(\mathrm{rec}\ z\ s : \prod_{(n:\mathbb{N})} M\ n \right)$$

Note that there is no need to tell to rec what M is, Lean will guess it from the type of s (we say that M is an *implicit variable*).

In particular, if $(n : \mathbb{N})$, then

$$(\mathrm{rec}\ z\ s\ n : M\ n)$$

Universal construction

What is the type of rec?

Even if one does not write M explicitly as an argument, the variable is still there, so the type of rec is

$$\prod_{(M:\mathbb{N}\to\mathrm{Sort}\ u)} M\ 0 \to \left(\prod_{(n:\mathbb{N})} M\ n \to M\ (\mathrm{succ}\ n)\right) \to \prod_{(n:\mathbb{N})} M\ n$$

To be precise, this is the type of $\mathrm{rec}.\{u\}$, the eliminator for the universe u. Since universes are not terms, we cannot take a further product over universes, and there is no universe big enough to contain all the $\mathrm{Sort}\ u$'s, so this is unavoidable.

Computation rules

There are two computation rules. As usual they say what happens when we apply the eliminator to terms obtained via the constructors.

Let M, z and s be as above. We have

$$\mathrm{rec}\ z\ s\ 0 \equiv z$$

and, if $(n : \mathbb{N})$,

$$\mathrm{rec}\ z\ s\ (\mathrm{succ}\ n) \equiv s\ n\ (\mathrm{rec}\ z\ s\ n)$$

There is no uniqueness principle.

Let's have a look at a special case of the eliminator, the non-dependent version. Let $(A : \mathrm{Type}\ u)$ be fixed. To specify a term

$$(f : \mathbb{N} \to A)$$

we need to fix a term $(z : A)$ and a (non-dependent) function $(s : \mathbb{N} \to A \to A)$. We get

$$(\mathrm{rec}\ z\ s : \mathbb{N} \to A)$$

such that

$$\mathrm{rec}\ z\ s\ 0 \equiv z \text{ and } \mathrm{rec}\ z\ s\ (\mathrm{succ}\ n) \equiv s\ n\ (\mathrm{rec}\ z\ s\ n)$$

for all $(n : \mathbb{N})$.

We see that in practice we need to specify the image of 0, and the image of $\mathrm{succ}\ n$ given the image of n. Indeed, the image of 0 is given by z, and the image of $\mathrm{succ}\ n$ is given by $s\ n\ x$, where x is the image of n.

Slogan

Using rec*, one can define functions*

$$(f : \mathbb{N} \to A)$$

by recursion in the usual way.

Let's go back to the dependent version of the eliminator, but in the special case where the motive $(M : \mathbb{N} \to \mathrm{Prop})$ takes values in $\mathrm{Sort}\ 0 = \mathrm{Prop}$.

Recall that in this case we have

$$\left(\prod_{(n:\mathbb{N})} M\ n : \mathrm{Prop} \right)$$

and in particular any $\left(p : \prod_{(n:\mathbb{N})} M\ n\right)$ is a proof of the proposition

$$\forall (n : \mathbb{N}), M\ n.$$

Let's construct such a p.

Using the eliminator, we need a term

$$(z : M\ 0)$$

and a function

$$\left(s : \prod_{(n:\mathbb{N})} M\ n \to M\ (\mathrm{succ}\ n) \right)$$

We have that $(M\ 0 : \mathrm{Prop})$, so z is now a proof that $M\ 0$ holds. On the other hand, we also have

$$\left(\prod_{(n:\mathbb{N})} M\ n \to M\ (\mathrm{succ}\ n) : \mathrm{Prop} \right)$$

So s corresponds to a proof of the proposition

$$\forall (n : \mathbb{N}), M\ n \to M\ (\mathrm{succ}\ n)$$

that is, $M\ n$ implies $M\ (\mathrm{succ}\ n)$ for all $(n : \mathbb{N})$.

In practice, to prove that $M\ n$ holds for all $(n : \mathbb{N})$, we need to prove that $M\ 0$ holds and that $M\ n$ implies $M\ (\mathrm{succ}\ n)$ for all $(n : \mathbb{N})$.

Slogan

Using rec*, one can prove propositions on* $\mathbb{N}$ *by induction in the usual way.*

Pattern matching

Using the eliminator explicitly is often impractical.

Lean allows a much more convenient notation, called *pattern matching*, where to specify a function f with domain $\mathbb{N}$ (in particular to prove a theorem about natural numbers) one has to:

- Specify the image of 0.
- Specify the image of $\mathrm{succ}\ n$. In this part, one is allowed to use $f\ n$.

We will see the precise syntax in the examples.

Addition

We want to define the addition of two natural numbers. Mathematically this is a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$, but we will use currying and we will define

$$\mathrm{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

First of all we need the image of 0, that is $\mathrm{add}\ 0 : \mathbb{N} \to \mathbb{N}$. This will of course be the identity function, so

$$\mathrm{add}\ 0\ n \equiv n$$

for all $(n : \mathbb{N})$.

Then we need to define $\mathrm{add}\ (\mathrm{succ}\ n)$ using $\mathrm{add}\ n$. This will be the function

$$\mathrm{fun}\ a \mapsto \mathrm{succ}\ (\mathrm{add}\ n\ a)$$

One can of course use rec directly, but using pattern matching will be much simpler, since one has not to write explicitly the function

$$(s : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}))$$

that says how to specify $\mathrm{add}\ (\mathrm{succ}\ n)$ given $\mathrm{add}\ n$.

The computation rules say that

$$0 + a = a$$
$$\text{succ } a + b = \text{succ } (a + b)$$

hold definitionally for all $(a\ b : \mathbb{N})$. In particular we get terms

$$(\text{zero\_add } a : 0 + a = a)$$
$$(\text{succ\_add } a\ b : \text{succ } a + b = \text{succ } (a + b))$$

for all $(a\ b : \mathbb{N})$.

We will explain in the following lectures why definitional equality implies equality, a notion that at the moment we have not defined.

Using lambda abstraction, we have terms

$$\left(\text{zero\_add} \ : \prod_{(a:\mathbb{N})} 0 + a = a\right)$$

$$\left(\text{succ\_add} \ : \prod_{(a\ b:\mathbb{N})} \text{succ}\ a + b = \text{succ}\ (a + b)\right)$$

The results

$$a + 0 = a \text{ and } a + \text{succ}\ b = \text{succ}\ (a + b)$$

for all $(a\ b : \mathbb{N})$ are true, but not definitionally. One can prove such results seeing them as dependent functions and using the eliminator explicitly, but Lean has a much nicer syntax, using the `induction` tactic. Under the hood, one has to use the eliminator.

Let's have a look at how to prove the first equality using the eliminator explicitly. We want to construct a term

$$\left(\mathrm{add\_zero} : \prod_{(a:\mathbb{N})} a + 0 = a \right)$$

The eliminator wants two things:

- A term of type $0 + 0 = 0$. This is given by

  $$(\mathrm{zero\_add}\ 0 : 0 + 0 = 0)$$

  This is proved using that two definitionally equal terms are equal.

- Next we need a dependent function

$$\left(f : \prod_{(a:\mathbb{N})} (a + 0 = a) \to (\mathrm{succ}\ a + 0 = \mathrm{succ}\ a) \right)$$

Using lambda abstraction again, it's enough to give a function

$$f\ a : (a + 0 = a) \to (\mathrm{succ}\ a + 0 = \mathrm{succ}\ a)$$

where $(a : \mathbb{N})$. In other words we need to prove that $a + 0 = a$ implies that $\mathrm{succ}\ a + 0 = \mathrm{succ}\ a$ as expected. Since we need to construct a function, we can use lambda abstraction again. In Lean this is easily done using the `intro` and `rw` tactics, but we will see that $a + 0 = a$ is an inductive proposition, so to construct such a function one can use the constructor and the eliminator for $=$.

The pred function

Using the recursor we can easily define the pred function and prove that it is a right inverse of succ.

- We define $\mathrm{pred}\ 0$ to be 0.
- We define $\mathrm{pred}\ (\mathrm{succ}\ n)$ to be n.

The computation rules say that

$$\mathrm{pred}\ 0 \equiv 0 \text{ and } \mathrm{pred}\ (\mathrm{succ}\ n) \equiv n$$

In particular one can prove that succ is injective.

$0 \neq 1$

How to prove that $0 \neq 1$? Remember that by definition this is the implication

$$0 = 1 \to \mathrm{False}$$

So we suppose that $0 = 1$ and we need to prove False.

The idea is the following: suppose we have two terms A and B, of any type T, such that we know that $A \neq B$. We consider the function $f : \mathbb{N} \to T$ defined, via the eliminator, by

$$f\ 0 = A \text{ and } f\ (\mathrm{succ}\ n) = B$$

In particular

$$f\ 0 \equiv A \text{ and } f\ 1 \equiv B$$

hold definitionally.

We need to prove False, but $A \neq B$ means

$$A = B \to \mathrm{False}$$

so we can prove $A = B$ (here we use the eliminator of the function type).

This is clear since $A = f\ 0 = f\ 1 = B$, where $f\ 0 = f\ 1$ is a consequence of our assumption that $0 = 1$.

It remains to find two terms that we are able to prove they are different. We now show that $\mathrm{True} \neq \mathrm{False}$.

We need to prove that $\mathrm{True} \neq \mathrm{False}$, that is the implication

$$\mathrm{True} = \mathrm{False} \to \mathrm{False}$$

In practice, we assume $\mathrm{True} = \mathrm{False}$ and we need to prove False.

By our very assumption ($\mathrm{True} = \mathrm{False}$), to prove False we can prove True! But this is trivial (by definition of True).

Remark

We didn't reason by contradiction.

Proof irrelevance

Remember that if $(P : \mathrm{Prop})$ is a proposition, then terms

$$(p : P)$$

are *proofs of* P or *witnesses that* P *holds*.

It looks like P is the "set" of its proof, but this is really a misleading analogy, because of *proof irrelevance*.

Slogan

If $(p\ p' : P)$ are two proofs of a proposition P, then p and p' are definitionally equal.

$$p \equiv p'$$

This property (called proof irrelevance) is a feature of Lean's type theory, and it is built-in in the kernel.

Other proof assistants (for example Rocq) use a *proof relevant* type theory, where proofs are not always definitionally equal.

Remark

Since proof irrelevance is part of the kernel and it is not stated as an axiom, it is not possible to study proof relevant type theory in Lean.

The idea is that a proof $(p : P)$ does not carry any information about P besides the fact that P holds. In practice, P is empty (meaning that we are not able to construct a term of type P) or it is a singleton. In the former case P is unprovable, in the latter case P holds.

Let's consider the following inductive constructions.

```
inductive Inhabited (A : Type) : Type
| intro (val : A) : Inhabited A
```

and

```
inductive Nonempty (A : Type) : Prop
| intro (val : A) : Nonempty A
```

Let's have a quick look at the rules for `Inhabited`.

- Formation rule: if A is a type, we have a well defined type `Inhabited A`.
- Constructors: there is only one constructor. If $(a : A)$, then

$$(\langle a \rangle : \text{Inhabited } A)$$

- Eliminator: if $(B : \text{Sort } u)$ and $(f : A \to B)$, then we have a function

$$\text{rec } f : \text{Inhabited } A \to B$$

- Computation rule: with the above notations we have

$$\mathrm{rec}\ f\ \langle a \rangle \equiv f\ a$$

Note that the eliminator allows to construct functions to any $(B : \mathrm{Sort}\ u)$, for example to $\mathbb{N}$.

In particular we can take $B = A$ and $f = \mathrm{id}$, getting a function

$$\mathrm{default} : \mathrm{Inhabited}\ A \to A$$

such that

$$\mathrm{default}\ \langle a \rangle \equiv a$$

The idea is that fixing $(x : \mathrm{Inhabited}\ A)$ correspond to fix a term $(\mathrm{default}\ x : A)$. We say that $\mathrm{Inhabited}\ A$ *contains data*, because any $(x : \mathrm{Inhabited}\ A)$ allows to reconstruct the term a it is build from (x "knows" a).

Mathematically, giving a term of type $\mathrm{Inhabited}\ A$ is the same as giving a term of type A.

This is not the same as proving that A is not empty! Knowing that A is not empty should not give a well defined term of type A.

Let's now have a quick look at the rules for `Nonempty`.

- Formation rule: if A is a type, we have a well defined proposition `Nonempty A`. This is almost the same as for `Inhabited A`, but `Nonempty A` is a proposition, while `Inhabited A` is in Type.
- Constructors: there is only one constructor. If $(a : A)$, then

$$(\langle a \rangle : \mathrm{Nonempty}\ A)$$

  This is exactly the same as for `Inhabited A`.
- Eliminator and computation rule: we now show that *it is not possible* for `Nonempty A` to have the same eliminator and computation rule as `Inhabited A`.

If this were the case, following the same reasoning as above, we would be able to construct a function

$$\text{default} : \text{Nonempty } A \to A$$

such that

$$\text{default } \langle a \rangle \equiv a$$

Now, to build terms of type $\text{Nonempty } \mathbb{N}$ one can use any term of type $\mathbb{N}$. In particular we have

$$(\langle 0 \rangle : \text{Nonempty } \mathbb{N}) \text{ and } (\langle 1 \rangle : \text{Nonempty } \mathbb{N})$$

But $(\mathrm{Nonempty}\ \mathbb{N} : \mathrm{Prop})$ is a proposition, so by proof irrelevance

$$\langle 0 \rangle \equiv \langle 1 \rangle$$

and in particular

$$\mathrm{default}\ \langle 0 \rangle \equiv \mathrm{default}\ \langle 1 \rangle$$

that implies

$$0 \equiv \mathrm{default}\ \langle 0 \rangle \equiv \mathrm{default}\ \langle 1 \rangle \equiv 1$$

In particular, 0 and 1 would be definitionally equal, and this is not the case (we even know that they are not propositionally equal).

Small elimination

In particular `Nonempty` cannot have the same eliminator as `Inhabited`.

Without an eliminator, `Nonempty` would be completely useless.

- Eliminator: if $(P : \mathrm{Prop})$ and $(f : A \to P)$, then we have a function

$$\mathrm{rec}\ f : \mathrm{Nonempty}\ A \to P$$

  It looks like the eliminator for `Inhabited`, but it is restricted to take value in a proposition.

It is called a *small eliminator*.

Remark

- *If* $(x : \mathrm{Nonempty}\ A)$ *and* f *is as above, then*

$$(\mathrm{rec}\ f\ x : P)$$

so $\mathrm{rec}\ f\ x$ *is a proof of* P *and* P *holds. The data of* f *means that we are able to prove* P *given any term of type* A*. The eliminator says that if we know that* $\mathrm{Nonempty}\ A$ *and moreover we can prove* P *given any* $(a : A)$*, then* P *holds. There is no need to fix a term of type* A*.*

- *There is no need for a computation rule, as any two terms of type* P *are always definitionally equal.*
- *Similarly, the above argument to show that the eliminator does not exist does not work, since it would prove that any two terms of type* P *are definitionally equal.*

Slogan

- *Inductive types i.e. inductively defined terms T of type $(T : \mathrm{Type}\ u)$ for some universe u, support large eliminators. This means that the eliminators allows to define functions*

$$(f : T \to A)$$

for any $(A : \mathrm{Sort}\ v)$, for any universe v.

- *Inductive propositions i.e. inductively defined terms P of type $(P : \mathrm{Prop})$ only support small eliminators. This means that the eliminator allows to define functions*

$$(f : P \to Q)$$

for any $(Q : \mathrm{Prop})$. Note that in this case $P \to Q$ is itself a proposition, the fact that P implies Q.

Inductive types

Every concrete type other than the universes and every type constructor other than $\prod$ is given by the construction of an inductive type.

The result of this construction (the inductive type) can live in any universe that is higher than than the universes used in the input.

The precise rules to form a new inductive type are the most complex aspect of Lean's type theory. We will only give an idea.

Let's review the rules to build an inductive type, taking into account all the various examples and trying to be as general as possible.

- Formation rule: it says what we need to build the inductive type. It can be "nothing" (for example in the case of True and $\mathbb{N}$) or a finite list of arguments (for example $\wedge$ takes two propositions).

- Constructors: they are a finite list of rules to produce terms of the new type. Each constructor can take:
  - No arguments. It corresponds to a fixed term of the new type (like $(0 : \mathbb{N})$).
  - Arguments given by the formation rule. For example the formation rule for $\times$ wants two type A and B, and the constructor is a function $A \to B \to A \times B$.
  - Arguments given by the inductive type itself. For example the constructor succ of $\mathbb{N}$ allows to construct a new natural number $\mathrm{succ}\ n$ given $(n : \mathbb{N})$. These are the constructors that really make the type *inductive*.

There are precise rules that say precisely how can an inductive type T appear in the constructors of T itself. We will not be precise here, but we will see an example of a forbidden constructor.

- Eliminators. After a new inductive type T has been declared, Lean add a new constant $T.\mathrm{rec}$ (or to be precise one for any universe) that allows to build dependent functions out of T. Besides the motive, that specifies the type of the (dependent) function we are going to build, $T.\mathrm{rec}$ takes several arguments.

  - First of all it needs to know where to send any constructor without arguments (for example where to send $(0 : \mathbb{N})$).
  - It also needs to know where to send the terms obtained by the constructors with arguments, in a way given by form of the constructors. For example, for $P \wedge Q$ the eliminator takes a function $P \to Q \to A$, and for $P \vee Q$ it takes two functions $P \to R$ and $Q \to R$ (here R needs to be a proposition).
  - Finally, for the constructors that use terms t of type T itself, one is allowed to use t to specify the image of the constructor. Again, the precise formulation is complicated, but think about what happens for $\mathbb{N}$.

- Computations rules. The computation rules say that evaluating the eliminator at a term given by a constructor the result is *definitionally equal* to the argument given to eliminator. For example, for the constructors without arguments, the result is precisely the term we gave to the eliminator. For the other constructors the situation is similar, but more complicated.

Remark

The computation rules are not added as new axioms, since we can not state that two terms are definitionally equal. They are added to the list of rules that Lean uses internally to check definitionally equality.

The constructors automatically satisfy a lot of properties (thanks to the existence of the eliminator and the computation rules).

For example

- They're always injective. The idea is to construct a right inverse as in the case of the Pred function.
- They have disjoint images. The proof is similar to the proof that $0 \neq 1$.

Remark

These properties are not axioms, they're proved using the eliminator. Lean automatically generates a lot of such results (and their proofs) when declaring a new inductive type. one can see what is generated using the `whatsnew in` *command.*

Lean's type theory

Putting all together, here is the axiomatic framework of Lean:

- a non-cumulative hierarchy of universes
- dependent types
- inductive types
- Prop is impredicative
- proof irrelevance
- subsingleton elimination

Is this consistent?

There are a lot of rules in Lean's type theory. For example, they imply the existence of an infinite type (like $\mathbb{N}$), while the existence of an infinite set is one of the axioms of ZF. In Mathlib there are even three additional axioms. Are we adding too many rules?

Theorem (Carneiro)

Mathlib's type theory is equivalent to ZFC plus the existence of countably many inaccessible cardinals.

Remark

In mathlib there is a construction of a model of ZFC with countably many inaccessible cardinals. The other way is a pen and paper proof.

The Curry–Howard correspondence

One can ask whether our definition of, say,

$$\exists x, P\ x$$

as an inductive proposition really models the "standard" notion of an existential (used in classical logic).

Slogan (Curry–Howard correspondence)

Doing mathematics using classical logic is "the same" as doing mathematics in type theory.

In particular, there is a correspondence between classical proofs and Lean's proofs.

Equality

Let's analyze one last example of an inductive proposition, the equality $a = b$.

In type theory this is a precisely defined notion, without anything special. Here is the definition

```
inductive Eq {A : Sort u} : A → A → Prop
| refl (a : A) : Eq a a

infix:50 " = "  => Eq
```

In particular, $a = b$ is just notation for the proposition $\mathrm{Eq}\ a\ b$. There is not need to specify A here, it is guessed looking at the type of a and b, that must be of the same type.

Remark

- *It is an inductive family of propositions rather than a single proposition.*
- *$a = b$ only makes sense if a and b are terms of the same type.*

We call the relation $a = b$ *propositional equality*. Let's have a look at the usual rules.

- Formation rule: if $(A : \mathrm{Sort}\ u)$ and $(a\ b : A)$, we have a proposition $\mathrm{Eq}\ a\ b$, that we denote $a = b$.
- Constructors: there is only one constructor, called refl. Given $(a : A)$,

$$(\mathrm{refl}\ a\ : a = a)$$

  In particular $\mathrm{refl}\ a$ is a proof that $a = a$.

Remark

If $a \equiv b$, then $(\mathrm{refl}\ a\ : a = b)$ is accepted by Lean, since there is no difference between a and b. In particular, definitional equality implies propositional equality.

- Eliminator. It is the most subtle aspect of the notion of equality. Here is the most general form.

```
#check @Eq.rec

{A : Sort u} → {a : A} →
{motive : (b : A) → Eq a b → Sort v} →
motive a (_ : Eq a a) → {b : A} →
(t : Eq a b) → motive b t
```

To better understand it, let's first of all fix $(A : \mathrm{Sort}\ u)$ and $(a : A)$ and let's consider `@Eq.rec A a`.

```
#check @Eq.rec A a

{motive : (b : A) → Eq a b → Sort v} →
motive a (_ : Eq a a) → {b : A} →
(t : Eq a b) → motive b t
```

The next thing we want to simplify is the motive. It is a dependent function

$$(b : A) \to \mathrm{Eq}\ a\ b \to \mathrm{Sort}\ v$$

Let's first of all consider the case of propositions, so $v = 0$ and $\mathrm{Sort}\ v = \mathrm{Prop}$.

To specify the motive we need a proposition $P\ b$ (depending on b), defined given h, a proof that $a = b$. For example, we can have a function

$$(P : A \to \mathrm{Prop}),$$

so a proposition $P\ b$ for all $(b : A)$ (regardless we can prove $a = b$). We can take for the motive the function

$$\mathrm{fun}\ b\ h \mapsto f\ b$$

We now have

```
#check @Eq.rec A a (fun b h ↦ f b)

P a → ∀ {b : A}, a = b → P b
```

So the eliminator just say that if we can prove $P\ a$ and we have $(b : A)$ such that $a = b$, then we have a proof of $P\ b$.

This special case of the eliminator is called the *substitution principle*.

Remark

Note that we don't need $P\ b$ to be defined for all $(b : A)$, but only if we already know that $a = b$. This can look like we only have one single proposition, $P\ a$, but remember that we are defining $a = b$. For example, replace $a = b$ by "a and b are friends". Then the eliminator says the following. Suppose we have a proposition $P\ b$ for all b friends of a. If $P\ a$ holds and b is a friend of a, then $P\ b$ holds.

Reflexivity

We now prove that equality is an equivalence relation.

Let's start with reflexivity. Let $(A : \mathrm{Sort}\ u)$ and let $(a : A)$. Then

$$(\mathrm{refl}\ a\ : a = a)$$

so $=$ is a reflexive relation.

Another way of thinking about $=$ is that it is the *smallest* reflexive relation (we will prove this in the examples).

Symmetry

To prove that $=$ is a symmetric relation, let $(a\ b : A)$ and let $(h : a = b)$. We need to prove $b = a$.

We want to use the eliminator, so first of all we need to find the motive. Let's consider the function $(P : A \to \mathrm{Prop})$ given by

$$\mathrm{fun}\ (x : A) \mapsto x = a$$

Note that, as above, this function is defined on all $(x : A)$, not only when $a = x$ (the fact that $=$ appears also in the definition of P is irrelevant).

We need to prove $P\ b$. The eliminator says exactly that if $P\ a$ holds and $a = b$, then $P\ b$ holds. But $P\ a$ holds by reflexivity and $a = b$ is our assumption, so we are done.

Transitivity

To prove that $=$ is a transitive relation, let $(a\ b\ c : A)$ and let $(hab : a = b)$ and $(hbc : b = c)$. We need to prove $a = c$.

As above first of all we need the motive, to apply the eliminator. Let's consider the function $(P : A \to \mathrm{Prop})$ given by

$$\mathrm{fun}\ (x : A) \mapsto a = x$$

We need to prove $P\ c$. The eliminator says that it is enough to prove $P\ b$ and that $b = c$.

We have $(hab : P\ b)$ and $(hbc : b = c)$, so we are done.

Substitution

Let $(B : \mathrm{Sort}\ v)$ be a type and let $(f : A \to B)$ be a function. If $(a\ b : A)$ with $a = b$, then

$$f\ a = f\ b$$

This result is called the *substitution principle*.

To prove it we proceed as above. Let's consider the function $(P : A \to \mathrm{Prop})$ given by

$$\mathrm{fun}\ (x : A) \mapsto f\ a = f\ x$$

We need to prove $P\ b$. The eliminator says that it is enough to prove $P\ a$ and that $a = b$. But $P\ a$ holds by reflexivity (applied to the term $f\ a$), and $a = b$ is our assumption.

Corollary

In particular, if $(f : A \to \mathrm{Prop})$*, we have that, if* $a = b$*, then* $f\ a \iff f\ b$ *as expected.*

Remark

Let $(B : A \to \mathrm{Sort}\ v)$ *be a function and let*

$$\left(f : \prod_{(a:A)} B\ a \right)$$

be a dependent function. If $(a\ b : A)$ *are terms such that* $a = b$*, can we deduce that* $f\ a = f\ b$*? This does not even make sense! Indeed* $(f\ a : B\ a)$ *and* $(f\ b : B\ b)$ *have not the same type. There exists a generalization of equality to take into account this situation (called heterogenous equality), but it is less interesting.*

Remark

- *Note that $=$ is a syntactic subsingleton, so the eliminator allows to define functions to any type.*
- *When using that $a = b$, the idea is that if we need to prove/construct something for b it is enough to do it for a. But beware that this "something" must be well defined. More precisely, finding the motive is often the main point of the proof. It should be well defined for all x such that $a = x$.*

Mathlib's axioms

Mathlib is Lean's standard mathematical library.

It consists of more than 1.5 millions lines of code of mathematics, from undergraduate to research level.

It adds three axioms to Lean's type theory:

- Propositional extensionality.
- Quotient types.
- The axiom of choice.

Propositional extensionality

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

Propositional extensionality says that if two propositions are equivalent then they are equal.

In particular if P is provable, meaning we can construct $(p : P)$, then $P = \mathrm{True}$. Indeed, any proposition implies True (since we can prove True for free), and if we have $(p : P)$ then $\mathrm{True} \to P$. Similarly, if $\neg P$ holds, then $P = \mathrm{False}$.

Thinking about proposition as "sets" that are empty when false and singletons when provable, then propositional extensionality says that if we have functions

$$P \to Q \text{ and } Q \to P$$

then $P = Q$.

This is reasonable since the existence of the two functions (that is the existence of the two implications) forces P and Q to be both empty or both singletons.

Quotient types

In set theory, if $\sim$ is an equivalence relation on a set X, one can explicitly build the quotient set $X/\sim$ using equivalence classes:

$$X/\sim \subseteq \mathcal{P}(X),$$

where $\mathcal{P}(X)$ is the power set of X. The existence of the sets $\mathcal{P}(X)$ and $X/\sim$ are immediate consequences of the axioms of set theory.

Then one defines the canonical map $X \to X/\sim$ and one can prove the universal property.

How to build $X/\sim$ in type theory?

We only have two ways of building types:

- Dependent functions.
- Inductive types.

Neither of these two constructions allows to build $X/\sim$.

Lean's type theory adds a new axiom (a function in this case) that allows to build the quotient type.

```
axiom Quot :
  {X : Sort u} → (X → X → Prop) → Sort u
```

In particular, Quot allows to build a new type $\mathrm{Quot}\ R$ given any relation $R : X \to X \to \mathrm{Prop}$ (we don't even need to assume that R is an equivalence relation).

We also need the canonical map $X \to \mathrm{Quot}\ R$:

```
axiom Quot.mk :
  {X : Sort u} → (R : X → X → Prop) → X →
    Quot R
```

At this point this is just any function, we don't know anything about it.

We also need to know that $\mathrm{Quot.mk}$ is surjective. In analogy with the induction principle for an inductive type, this is stated by saying that to prove a proposition about all $(q : \mathrm{Quot}\ R)$, it is enough to prove it for those terms of the form $\mathrm{Quot.mk}\ R\ a$.

```
axiom Quot.ind :
    ∀ {X : Sort u} {R : X → X → Prop}
    {P : Quot R → Prop},
    (∀ a, P (Quot.mk R a)) →
    ∀ q, P q
```

It implies that $\mathrm{Quot.mk}$ is surjective in the usual sense.

We also need to lift functions that are constant along the equivalence classes.

```
axiom Quot.lift :
  {X : Sort u} → {R : X → X → Prop} →
  {Y : Sort u} → (f : X → Y) →
  (∀ a b, R a b → f a = f b) →
  Quot R → Y
```

Note that, as for inductive types, we can now build terms of type $\mathrm{Quot}\ R$ (using $\mathrm{Quot.mk}$) and define functions (including proving propositions) out of $\mathrm{Quot}\ R$.

We want to relate the function given by $\mathrm{Quot.lift}$ with $\mathrm{Quot.mk}$. If $f : X \to Y$ is such that $(H : \forall a\ b,\ R\ a\ b \to f\ a = f\ b)$, then we want $\mathrm{Quot.lift}\ H(\mathrm{Quot.mk}\ R\ x) = f\ x$ for all $(x : X)$. We add this computation rule as a *definitional equality*

$$\mathrm{Quot.lift}\ H\ (\mathrm{Quot.mk}\ R\ x) \equiv f\ x$$

Remark

These functions are left undefined. In practice we are assuming their existence, but we are not assuming any special property. This existence is not a very strong assumption, for example $\mathrm{Quot}\ R$ *could be* X, $\mathrm{Quot.mk}$ *the identity and* $\mathrm{Quot.lift}\ f\ H = f$. *In this case* $\mathrm{Quot.ind}$ *and the computation rule above trivially hold.*

The axioms above are part of Lean's type theory, but what makes them really a construction of quotient types is the following axioms, added in mathlib.

```
axiom Quot.sound :
  ∀ {X : Type u} {R : X → X → Prop} {a b : X},
    R a b → Quot.mk R a = Quot.mk R b
```

It does not follow from the previous axioms (try to prove it!).

Remark

- *If* $\mathrm{Quot}\ R = X$ *then* $\mathrm{Quot.sound}$ *does not hold. It is a genuine new axiom.*
- *Note that we didn't assume* R *to be an equivalence relation, but this is the situation where the axioms are most useful.*

Functional extensionality

Using quotient types we can now *prove* the following.

Theorem (Functional extensionality)

Let $(A : \mathrm{Sort}\ u)$ *and* $(B : \mathrm{Sort}\ v)$. *Given two functions* $(f\ g : A \to B)$ *such that*

$$\forall (a : A),\ f\ a = g\ a$$

we have $f = g$.

In particular, any theorem that uses functional extensionality will depend on $\mathrm{Quot.sound}$.

The axiom of choice

Remember the definition of $\mathrm{Nonempty}\ A$

```
inductive Nonempty (A : Sort u) : Prop
| intro (val : A) : Nonempty A
```

It is an inductive *proposition* with only one constructor: if $(a : A)$, then

$$(\langle a \rangle : \mathrm{Nonempty}\ A)$$

It is easy to prove that

$$\mathrm{Nonempty}\ A \iff \exists (a : A),\ \mathrm{True}$$

We explained that $\mathrm{Nonempty}\ A$ does not contain data, and that it is impossible to build a function

$$\mathrm{default} : \mathrm{Nonempty}\ A \to A$$

such that

$$\mathrm{default}\ \langle a \rangle = a$$

for all $(a : A)$.

The axiom of choice is the following function

```
axiom choice {A : Sort u} : Nonempty A → A
```

We have that choice magically produces an element of A given the assumption $\mathrm{Nonempty}\ A$. For Lean it is a well defined function (even if it has no definition).

It is important to understand that if, say, $(h : \mathrm{Nonempty}\ \mathbb{N})$ (something easy to prove), the natural number $(\mathrm{choice}\ h : \mathbb{N})$ is *well defined and fixed once and for all*. It is always the same, in all proofs where we use the axiom of choice. Of course it is impossible to say anything about it (except that it is a natural number), but for Lean is perfectly well defined. Moreover, it does not depend on h, since any other proof of $\mathrm{Nonempty}\ \mathbb{N}$ is definitionally equal to h by proof irrelevance.

In practice we have fixed (once and for all) a term $(a : A)$ for all A such that $\mathrm{Nonempty}\ A$.

Remark

- *As we explained it is impossible to have*

$$\text{choice}\ \langle a \rangle = a$$

  for all $(a : A)$.
- *This version of the axiom of choice is slightly stronger than the usual version in set theory (the product of nonempty sets is nonempty) and it is called the axiom of global choice. But remember that Lean's type theory plus mathlib's three axioms is equivalent to ZFC plus the existence of countably many inaccessible cardinals. In particular the usual axiom of choice holds in Lean.*

Excluded middle

Excluded middle is the following property, for any $(P : \mathrm{Prop})$,

$$P \vee \neg P$$

Remember that by definition $\neg P$ is the implication $P \to \mathrm{False}$ Excluded middle says that P holds or P implies false. In classical logic it is one of the basic assumptions.

It is used for example to do proofs by contradiction. To prove P we need to consider the two cases of $P \vee \neg P$:

- If P holds we are done (since we want to prove P! In practice this case is never explicitly considered).
- If $\neg P$ holds we still have to prove P, but of couse our assumption cannot help. But it is (as always) enough to prove False. So we can prove that $\neg P \to \mathrm{False}$, that is $\neg\neg P$.

Excluded middle is not assumed in Lean's type theory and indeed we didn't use it until now (and we never did a proof by contradiction).

Using the three new axioms we can now *prove* excluded middle.

Theorem (Diaconescu)

Let $(P : \mathrm{Prop})$. *Using functional extensionality (hence quotient types and propositional extensionality) and the axiom of (global) choice, we have*

$$P \vee \neg P$$

We will not prove Diaconescu's theorem, but let's first of all see some consequences.

Theorem

Let $(P : \mathrm{Prop})$. *Then* $P = \mathrm{True} \vee P = \mathrm{False}$.

Proof.

Using that $P \vee \neg P$ holds we have to consider two cases. In both we will use propositional extensionality.

- If P holds then it is easy to prove that $P \iff \mathrm{True}$ (since both hold), so $P = \mathrm{True}$.
- If $\neg P$ holds, we prove that $P \iff \mathrm{False}$, hence $P = \mathrm{False}$. To prove $P \Rightarrow \mathrm{False}$ note that this is exactly $\neg P$, that is our assumption. The other implication is $\mathrm{False} \Rightarrow P$, that always holds (for any P).

Theorem

For all $(P : \mathrm{Prop})$ *we have*

$$\neg\neg P \to P$$

Proof.

We consider again the two cases $P \vee \neg P$.

- If P holds there is nothing to prove.
- Suppose that $\neg P$ holds, so that $P \Rightarrow \mathrm{False}$. To prove P it is enough to prove False (using $\mathrm{False.rec}$). Since we are supposing $\neg\neg P$ we can prove $\neg P$ and we are done.

Corollary

Let $(P\ Q : \mathrm{Prop})$. *If both* $P \to Q$ *and* $\neg P \to Q$ *hold then* Q *holds.*

Proof.

It is an immediate consequence that $P \vee \neg P$ holds. □

In particular we can prove theorems by cases, supposing P or $\neg P$ holds: in Lean we can use the `by_cases` tactic.

To reason by contradiction (i.e. to use $\neg\neg P \to P$ to prove P) we can use the `by_contra!` tactic. Indeed, `by_contra! h` will create (and simplify) an assumption $(h : \neg P)$, where P is the current goal, and replace the goal with False.