Type theory in Lean - 3

Riccardo Brasca

Université Paris Cité

October 28th 2023

The universe Prop

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

The universe Prop

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

Even mathematical statement.

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

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.

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.

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.

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...

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?

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!

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?

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.

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.

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 (more on this next week).*

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 (more on this next week).*

If u *is a universe and* $(T : \mathrm{Type\ u})$ *then it is safe to think to* T *as a set.*

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 (more on this next week).*

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 $(p : 2 + 2 = 4)$ is *a proof* that $2 + 2 = 4$.

A term p of type $(p : 2 + 2 = 5)$ would be a proof that $2 + 2 = 5$.

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 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.

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 $\mathbb{R}$

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*.

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.

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`, 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.

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

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 $x^n + y^n = z^n$.

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 $(P : \mathrm{Prop})$?

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.

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...

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 constructions of last week.

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 constructions of last week.

A proposition P obtained by (a generalization of) the constructions of last week will have constructors

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 constructions of last week.

A proposition P obtained by (a generalization of) the constructions of last week will have constructors, that allows us to build terms p of type $(t : P)$

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 constructions of last week.

A proposition P obtained by (a generalization of) the constructions of last week 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 $(A \to B : \mathrm{Type}\ u)$

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

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

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 $\mathrm{Type}\ u$).

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.

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.

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 $(a : A)$.

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

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 $P\ a$.

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

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

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* $(a : A)$.

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 $(\prod_{(a:A)} P\ a : \mathrm{Prop})$.

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 $v \neq 0$.

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

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

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 $(p : P)$

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.

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.

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.

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 $P \to Q$!

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 $P \to Q$.

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 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 last week, giving the introduction rule, constructors, eliminators...

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 last week, 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.

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.

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

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

- 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

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

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

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

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

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

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.

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.

For example we have the dependent version of the eliminator and so on.

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.

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.*

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.*

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.*

Double implication

We move on to the *if and only if* operator.

Double implication

We move on to the *if and only if* operator.

Definition

Given two proposition P and Q, we the proposition P *if and only if* Q, denoted $P \leftrightarrow Q$, as

$$(P \to Q) \land (Q \to P)$$

Double implication

We move on to the *if and only if* operator.

Definition

Given two proposition P and Q, we the proposition P *if and only if* Q, denoted $P \leftrightarrow Q$, as

$$(P \to Q) \wedge (Q \to P)$$

We can also define $P \leftrightarrow Q$ as an inductive proposition, giving the introduction rule and so on.

Disjunction

Let's define the *or* operator.

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.

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

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

- 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 $(q : Q)$.

- 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 $P \lor Q$ we can prove P or we can prove Q, using the two constructors.

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.

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 $P \vee Q \to R$.

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 $Q \to R$

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 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.*

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.*

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.)*

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.)*

Remark

Note that the eliminator for $\vee$ *only allows to build function to a proposition* R *(so to prove it).*

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.)*

Remark

Note that the eliminator for $\vee$ only allows to build function to a proposition R (so to prove it). For example, it doesn't allow to build a function $P \vee Q \to \mathbb{N}$

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.)*

Remark

Note that the eliminator for $\vee$ *only allows to build function to a proposition* R *(so to prove it). For example, it doesn't allow to build a function* $P \vee Q \to \mathbb{N}$, *while the eliminator for* $\wedge$ *does it.*

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.)*

Remark

Note that the eliminator for $\vee$ only allows to build function to a proposition R (so to prove it). For example, it doesn't allow to build a function $P \vee Q \to \mathbb{N}$, while the eliminator for $\wedge$ does it. This is an example of a subtle problem, called subsingleton elimination (more on this later).

The True proposition

We now define $(\mathrm{True} : \mathrm{Prop})$, a special proposition that models the True truth value.

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

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

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.

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.

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, 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.

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.

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.

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.

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.

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

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

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.

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 $(A : \mathrm{Sort}\ u)$.

- 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.

- 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.

- 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.

- 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.

- 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

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

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.

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.

Negation

Definition

Given a proposition (P : 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.

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.

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*

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

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 see this next week).*

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

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

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

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

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

- 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 $P\ a$.

- 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.

- 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*

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).*

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

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

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).*

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).

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.