Type theory in Lean - 5

Riccardo Brasca

Université Paris Cité

November 13th 2023

Proof irrelevance

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

$$(p : P)$$

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

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

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

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.

$$p \equiv p'$$

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

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.

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 Coq) 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.

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.

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.

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
```

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`.

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

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

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

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

- Computation rule: with the above notations we have

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

- 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 $\mathbb{N}$.

- 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 $(\mathrm{default}\ x : A)$.

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).

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.

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!

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`.

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`.

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.

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

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

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`.

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

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

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

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

such that

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

Now, to build terms of type $\mathrm{Nonempty}\ \mathbb{N}$ one can use any term of type $\mathbb{N}$.

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

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

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

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

and in particular

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

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

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

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`.

Small elimination

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

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

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

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.

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.*

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*.*

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.*

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*.*

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.*

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.*

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.

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.*

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 $(Q : \mathrm{Prop})$.

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*

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*.*

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

For example, given $(t : \mathrm{False})$ we can produce a term of any type

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

For example, given $(t : \mathrm{False})$ we can produce a term of any type, so False, despite being an inductive proposition, supports large elimination.

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

For example, given $(t : \mathrm{False})$ we can produce a term of any type, so False, despite being an inductive proposition, supports large elimination.

Going back to the eliminator for $P \wedge Q$, one see that it also supports large elimination.

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

For example, given $(t : \mathrm{False})$ we can produce a term of any type, so False, despite being an inductive proposition, supports large elimination.

Going back to the eliminator for $P \wedge Q$, one see that it also supports large elimination. Recall that the eliminator in this case takes a function $f : P \to Q \to A$ and gives a function $g : P \wedge Q \to A$.

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

For example, given $(t : \mathrm{False})$ we can produce a term of any type, so False, despite being an inductive proposition, supports large elimination.

Going back to the eliminator for $P \wedge Q$, one see that it also supports large elimination. Recall that the eliminator in this case takes a function $f : P \to Q \to A$ and gives a function $g : P \wedge Q \to A$. The idea is that there is only one way of proving $P \wedge Q$: to do so one has to give $(p : P)$ and $(q : Q)$.

Subsingleton elimination

There is an exception to the previous slogan, called *subsingleton elimination*.

For example, given $(t : \mathrm{False})$ we can produce a term of any type, so False, despite being an inductive proposition, supports large elimination.

Going back to the eliminator for $P \wedge Q$, one see that it also supports large elimination. Recall that the eliminator in this case takes a function $f : P \to Q \to A$ and gives a function $g : P \wedge Q \to A$. The idea is that there is only one way of proving $P \wedge Q$: to do so one has to give $(p : P)$ and $(q : Q)$. Now it's clear that we can set

$$g\ \langle p, q \rangle = f\ p\ q$$

and this is indeed the computation rule.

Slogan

If P is an inductive proposition such that "there is at most one way to canonically produce terms of type P" then P supports large elimination

Slogan

If P is an inductive proposition such that "there is at most one way to canonically produce terms of type P" then P supports large elimination, meaning that the eliminator allows to produce functions

$$P \to A$$

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

Slogan

If P is an inductive proposition such that "there is at most one way to canonically produce terms of type P" then P supports large elimination, meaning that the eliminator allows to produce functions

$$P \to A$$

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

We say in this case that P is a *syntactic subsingleton*.

Slogan

If P is an inductive proposition such that "there is at most one way to canonically produce terms of type P" then P supports large elimination, meaning that the eliminator allows to produce functions

$$P \to A$$

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

We say in this case that P is a *syntactic subsingleton*.

The precise definition is technical, but essentially it is the following.

Definition

A syntactic subsingleton is an inductive proposition with at most one constructor whose arguments are either Prop or appear as immediate arguments in the output type.

Example

- False has no constructors, so it is a syntactic subsingleton.

Example

- False has no constructors, so it is a syntactic subsingleton.
- $P \land Q$ has only one constructor $P \to Q \to P \land Q$ whose arguments are propositions, so it is a syntactic subsingleton.

Example

- False has no constructors, so it is a syntactic subsingleton.
- $P \wedge Q$ has only one constructor $P \to Q \to P \wedge Q$ whose arguments are propositions, so it is a syntactic subsingleton.
- We will see that $=$ is a syntactic subsingleton.

Example

- False has no constructors, so it is a syntactic subsingleton.
- $P \wedge Q$ has only one constructor $P \to Q \to P \wedge Q$ whose arguments are propositions, so it is a syntactic subsingleton.
- We will see that $=$ is a syntactic subsingleton.
- $P \vee Q$ is *not* a syntactic subsingleton

Example

- False has no constructors, so it is a syntactic subsingleton.
- $P \wedge Q$ has only one constructor $P \to Q \to P \wedge Q$ whose arguments are propositions, so it is a syntactic subsingleton.
- We will see that $=$ is a syntactic subsingleton.
- $P \vee Q$ is *not* a syntactic subsingleton because it has two constructors.

Example

- False has no constructors, so it is a syntactic subsingleton.
- $P \wedge Q$ has only one constructor $P \to Q \to P \wedge Q$ whose arguments are propositions, so it is a syntactic subsingleton.
- We will see that $=$ is a syntactic subsingleton.
- $P \vee Q$ is *not* a syntactic subsingleton because it has two constructors. The idea is that if we have $f : P \to A$ and $g : Q \to A$, we cannot build a function $P \vee Q \to A$, because, given $(x : P \vee Q)$ we don't know whether x has been proved by proving P (and then we want to use f), or has been proved by proving Q (and then we want to use g).

Example

- False has no constructors, so it is a syntactic subsingleton.
- $P \wedge Q$ has only one constructor $P \to Q \to P \wedge Q$ whose arguments are propositions, so it is a syntactic subsingleton.
- We will see that $=$ is a syntactic subsingleton.
- $P \vee Q$ is *not* a syntactic subsingleton because it has two constructors. The idea is that if we have $f : P \to A$ and $g : Q \to A$, we cannot build a function $P \vee Q \to A$, because, given $(x : P \vee Q)$ we don't know whether x has been proved by proving P (and then we want to use f), or has been proved by proving Q (and then we want to use g). On the other hand, if A is a proposition, this does not matter because of proof irrelevance.

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.

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.

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.

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.

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.

- Formation rule: it says what we need to build the inductive type. It can be “nothing” (for example in the case of True and $\mathbb{N}$)

- 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).

- 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.

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

- 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 $A \to B \to A \times B$.

- 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 $(n : \mathbb{N})$.

- 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.

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.

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.

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

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.

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 $P \to Q \to A$

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).

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.

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.

- 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.

- 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.

- 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.

- 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 property

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

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

For example

- They're always injective.

The constructors automatically satisfy a lot of property (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.

The constructors automatically satisfy a lot of property (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 constructors automatically satisfy a lot of property (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 $0 \neq 1$.

The constructors automatically satisfy a lot of property (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

The constructors automatically satisfy a lot of property (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

The constructors automatically satisfy a lot of property (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.

The constructors automatically satisfy a lot of property (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.*

Inductive families

Lean also supports a slight generalization of inductive types, *inductive families*.

Inductive families

Lean also supports a slight generalization of inductive types, *inductive families*.

```
inductive foo : ... → Sort u where
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

Inductive families

Lean also supports a slight generalization of inductive types, *inductive families*.

```
inductive foo : ... → Sort u where
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

Here `foo` is a function that takes a list of arguments and produces an inductive type

Inductive families

Lean also supports a slight generalization of inductive types, *inductive families*.

```
inductive foo : ... → Sort u where
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

Here `foo` is a function that takes a list of arguments and produces an inductive type, in this sense it is a family of inductive types.

Inductive families

Lean also supports a slight generalization of inductive types, *inductive families*.

```
inductive foo : ... → Sort u where
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

Here `foo` is a function that takes a list of arguments and produces an inductive type, in this sense it is a family of inductive types. Each constructor then constructs an element of some member of the family

Inductive families

Lean also supports a slight generalization of inductive types, *inductive families*.

```
inductive foo : ... → Sort u where
| constructor₁ : ... → foo ...
| constructor₂ : ... → foo ...
...
| constructorₙ : ... → foo ...
```

Here `foo` is a function that takes a list of arguments and produces an inductive type, in this sense it is a family of inductive types. Each constructor then constructs an element of some member of the family, using terms of other elements of the family.

Lean's type theory

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

Lean's type theory

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

- a non-cumulative hierarchy of universes

Lean's type theory

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

- a non-cumulative hierarchy of universes
- dependent types

Lean's type theory

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

- a non-cumulative hierarchy of universes
- dependent types
- inductive types

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

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

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.

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.

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?

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.

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.

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).

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.

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.