Type theory in Lean - 4

Riccardo Brasca

Université Paris Cité

November 4th 2023

The natural numbers

We have seen several inductive types and inductive propositions.

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 have seen several inductive types and inductive propositions.

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

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.

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.

Constructors

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

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

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 $\mathrm{succ}\ n$:

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

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

Eliminator

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

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.

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.

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*.

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

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

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

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

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*).

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?

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

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

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.

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.

Computation rules

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

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.

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

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

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.

There is no uniqueness principle.

Let's have a look at a special case of the eliminator, the non-dependent version.

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

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

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

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.

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.

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 $\mathrm{Sort}\ 0 = \mathrm{Prop}$.

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

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

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 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

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

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

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

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.

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.

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 $\mathrm{succ}\ n$.

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 $\operatorname{succ} n$. In this part, one is allowed to use $f\ n$.

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.

An example: the double function

Let's define the double function $(\mathrm{double} : \mathbb{N} \to \mathbb{N})$.

An example: the double function

Let's define the double function ($\mathrm{double} : \mathbb{N} \to \mathbb{N}$).

We need to specify:

- The image of 0, that is $z = 0$.

An example: the double function

Let's define the double function ($\mathrm{double} : \mathbb{N} \to \mathbb{N}$).

We need to specify:

- The image of 0, that is $z = 0$.
- The image of $\mathrm{succ}\ n$, given x, the image of n.

An example: the double function

Let's define the double function ($\mathrm{double} : \mathbb{N} \to \mathbb{N}$).

We need to specify:

- The image of 0, that is $z = 0$.
- The image of $\mathrm{succ}\ n$, given x, the image of n. We want the image of $\mathrm{succ}\ n$ to be

$$\mathrm{succ}\ \mathrm{succ}\ x,$$

An example: the double function

Let's define the double function $(\mathrm{double} : \mathbb{N} \to \mathbb{N})$.

We need to specify:

- The image of 0, that is $z = 0$.
- The image of $\mathrm{succ}\ n$, given x, the image of n. We want the image of $\mathrm{succ}\ n$ to be

  $$\mathrm{succ}\ \mathrm{succ}\ x,$$

  so the function $(s : \mathbb{N} \to \mathbb{N} \to \mathbb{N})$ is given by

  $$\mathrm{fun}\ a\ b \mapsto \mathrm{succ}\ (\mathrm{succ}\ b)$$

Addition

We want to define the addition of two natural numbers.

Addition

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

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

Addition

We want to define the addition of two natural numbers. Mathematically this is a function $\mathbb{N} \times \mathbb{N}$, 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 $\mathrm{add}\ 0 : \mathbb{N} \to \mathbb{N}$.

Addition

We want to define the addition of two natural numbers. Mathematically this is a function $\mathbb{N} \times \mathbb{N}$, 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 $\mathrm{add}\ n$

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

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

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

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

One can of course use rec directly, but using pattern matching will be much simpler

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

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

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

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

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

The computation rules say that

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

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

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

The computation rules say that

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

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

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

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

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

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

Using lambda abstraction, we have terms

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

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

The results

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

for all $(a \ b : \mathbb{N})$ are true, but not definitionally.

Using lambda abstraction, we have terms

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

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

The results

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

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.

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.

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

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

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

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

The eliminator wants two things:

- A term of type $0 + 0 = 0$.

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

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

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

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

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

- 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.

- 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.

- 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 for $=$.

The pred function

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

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.

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 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

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

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 $0 \neq 1$?

$0 \neq 1$

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

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

$0 \neq 1$

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.

$0 \neq 1$

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 $A \neq B$.

$0 \neq 1$

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 $f \colon \mathbb{N} \to T$ defined, via the eliminator, by

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

$0 \neq 1$

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 $f \colon \mathbb{N} \to T$ 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

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

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).

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 $A = f\ 0 = f\ 1 = B$

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 $0 = 1$.

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 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

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

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.

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!

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).

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.

$\mathrm{succ}\ n \neq 0$

Similarly, we can prove that $\mathrm{succ}\ n \neq 0$ for any given $(n : \mathbb{N})$.

$\mathrm{succ}\ n \neq 0$

Similarly, we can prove that $\mathrm{succ}\ n \neq 0$ for any given $(n : \mathbb{N})$.

Suppose $\mathrm{succ}\ n = 0$ and let's consider the function $f : \mathbb{N} \to \mathrm{Prop}$ given by

$$f\ 0 = \mathrm{False} \text{ and } f\ (\mathrm{succ}\ a) = \mathrm{True}$$

as above.

succ $n \neq 0$

Similarly, we can prove that $\mathrm{succ}\ n \neq 0$ for any given $(n : \mathbb{N})$.

Suppose $\mathrm{succ}\ n = 0$ and let's consider the function $f : \mathbb{N} \to \mathrm{Prop}$ given by

$$f\ 0 = \mathrm{False} \text{ and } f\ (\mathrm{succ}\ a) = \mathrm{True}$$

as above.

We have $\mathrm{False} = f\ 0 = f\ (\mathrm{succ}\ n) = \mathrm{True}$, so we are done as before.