Type theory in Lean - 2

Riccardo Brasca

Université Paris Cité

October 21st 2023

Recall from last week:

Recall from last week:

- Three layers of objects: terms, types and universes.

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.
- Every term has its own type.

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.
- Every term has its own type.
- Every type lives in a universe.

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.
- Every term has its own type.
- Every type lives in a universe.

A countable non-cumulative hierarchy of universes:

$$\mathrm{Type}\ n$$

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.
- Every term has its own type.
- Every type lives in a universe.

A countable non-cumulative hierarchy of universes:

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

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.
- Every term has its own type.
- Every type lives in a universe.

A countable non-cumulative hierarchy of universes:

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

Moreover

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

Recall from last week:

- Three layers of objects: terms, types and universes.
- Everything is a term.
- Every term has its own type.
- Every type lives in a universe.

A countable non-cumulative hierarchy of universes:

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

Moreover

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

In general

$$\mathrm{Type}\ n = \mathrm{Sort}\ n+1 \text{ and } \mathrm{Prop} = \mathrm{Sort}\ 0$$

The type of a term can only be checked.

The type of a term can only be checked. Lean has an algorithm to do so.

The type of a term can only be checked. Lean has an algorithm to do so.

Lean also has an algorithm to decide that an expression is a well formed type.

The type of a term can only be checked. Lean has an algorithm to do so.

Lean also has an algorithm to decide that an expression is a well formed type.

There is a notion of *definitional equality* of two terms (of the same type).

The type of a term can only be checked. Lean has an algorithm to do so.

Lean also has an algorithm to decide that an expression is a well formed type.

There is a notion of *definitional equality* of two terms (of the same type).

Definitionally equality is not a mathematical property.

The type of a term can only be checked. Lean has an algorithm to do so.

Lean also has an algorithm to decide that an expression is a well formed type.

There is a notion of *definitional equality* of two terms (of the same type).

Definitionally equality is not a mathematical property. It can be checked by Lean.

The type of a term can only be checked. Lean has an algorithm to do so.

Lean also has an algorithm to decide that an expression is a well formed type.

There is a notion of *definitional equality* of two terms (of the same type).

Definitionally equality is not a mathematical property. It can be checked by Lean.

In practice, if $x \equiv y$ then one can replace x by y everywhere.

How to build new types

Goal of today: how to build new types

How to build new types

Goal of today: how to build new types out of old ones.

How to build new types

Goal of today: how to build new types out of old ones.

We fix a universe u.

How to build new types

Goal of today: how to build new types out of old ones.

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

How to build new types

Goal of today: how to build new types out of old ones.

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

One can be more general

How to build new types

Goal of today: how to build new types out of old ones.

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

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

How to build new types

Goal of today: how to build new types out of old ones.

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

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

We will study in details two constructions:

- Dependent functions.
- Dependent pairs.

For each construction will will follow the same pattern.

For each construction will will follow the same pattern.

- *Formation rules*: when we can form a new type using the construction.

For each construction will will follow the same pattern.

- *Formation rules*: when we can form a new type using the construction.
- *Constructors* or *formation rules*: how to build terms of the new type.

For each construction will will follow the same pattern.

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

For each construction will will follow the same pattern.

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

For each construction will will follow the same pattern.

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

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

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

- An optional *uniqueness principle*: roughly speaking it is the fact that the only terms of the new type are those obtained using the constructors.

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

- An optional *uniqueness principle*: roughly speaking it is the fact that the only terms of the new type are those obtained using the constructors. It is optional, and often it can be proved using the previous rules.

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

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

Functions types

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

Functions types

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

- Formation rule: if A and B are two types, we have another type $A \to B$, whose terms are called *functions* from A to B.

Functions types

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

- Formation rule: if A and B are two types, we have another type $A \to B$, whose terms are called *functions* from A to B.
- Constructors: there is only one constructor, called *lambda abstraction*.

Functions types

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

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

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

  is of type $A \to B$.

- Eliminators: there is only one eliminator. If

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

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

- Eliminators: there is only one eliminator. If

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

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

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

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

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

- Eliminators: there is only one eliminator. If

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

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

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

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

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

Not 100% precise: consider the case where E is the expression $x + (\mathrm{fun}\ x \mapsto \sin(x))\ 0$.

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

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

so all functions can be obtained via lambda abstraction.

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

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

so all functions can be obtained via lambda abstraction.

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

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

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

so all functions can be obtained via lambda abstraction.

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

At the moment there is no *functional extensionality principle*: if $f\ x = g\ x$ for all x, we cannot prove that $f = g$.

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

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

so all functions can be obtained via lambda abstraction.

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

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

Functions of several variables

If A, B and C are types, the type

$$A \to B \to C = A \to (B \to C)$$

can be thought as the type of functions $A \times B \to C$.

Functions of several variables

If A, B and C are types, the type

$$A \to B \to C = A \to (B \to C)$$

can be thought as the type of functions $A \times B \to C$.

If $(a : A)$, $(b : B)$ and $(f : A \to B \to C)$, then $(f\ a\ b : C)$.

Functions of several variables

If A, B and C are types, the type

$$A \to B \to C = A \to (B \to C)$$

can be thought as the type of functions $A \times B \to C$.

If $(a : A)$, $(b : B)$ and $(f : A \to B \to C)$, then $(f\ a\ b : C)$.

This process is called *currying*.

Dependent functions

Consider the assignment

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

where n is a natural number.

Dependent functions

Consider the assignment

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

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

Dependent functions

Consider the assignment

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

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

Dependent functions

Consider the assignment

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

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

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

Dependent functions

Consider the assignment

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

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

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

f is *not* a function as above.

Dependent functions

Consider the assignment

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

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

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

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

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

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

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

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

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

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

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

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

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

The real primitive construction is the Π-type

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

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

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

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

Rules

Here are the various rules for dependent functions.

Rules

Here are the various rules for dependent functions. They generalize the corresponding rules for functions.

Rules

Here are the various rules for dependent functions. They generalize the corresponding rules for functions.

- Formation rule: if A is a type and, for all $(a : A)$ we have an expression $B(a)$ that is a well formed type (one can think to B as a function $(B : A \to \mathrm{Type}\ u)$).

Rules

Here are the various rules for dependent functions. They generalize the corresponding rules for functions.

- Formation rule: if A is a type and, for all $(a : A)$ we have an expression $B(a)$ that is a well formed type (one can think to B as a function $(B : A \to \mathrm{Type}\ u)$). We have another type, denoted
$$\prod_{(a:A)} B(a) = (a : A) \to B(a)$$
called Π-type. Its terms are called *dependent functions*.

- Constructors: there is only one constructor, called *lambda abstraction*.

- Constructors: there is only one constructor, called *lambda abstraction*. If E is any expression containing a variable x such that $(E : B(x))$ if $(x : A)$, then

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

is of type $\prod_{(a:A)} B(a) = (a : A) \to B(a)$.

- Eliminators: there is only one eliminator. If

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

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

- Eliminators: there is only one eliminator. If

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

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

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

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

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

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

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

so all functions can be obtained via lambda abstraction.

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

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

so all functions can be obtained via lambda abstraction.

Example

The identity function can be considered as a dependent function.

$$\text{id} : (A : \text{Type } u) \to (A \to A)$$

Example

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

$$\begin{aligned} \mathrm{swap}\ f &: B \times A \to C \\ (b, a) &\mapsto f(a, b) \end{aligned}$$

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

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

Cartesian product

We now move on to the Cartesian product.

Cartesian product

We now move on to the Cartesian product.

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

Cartesian product

We now move on to the Cartesian product.

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

- Constructors: there is only one constructor.

Cartesian product

We now move on to the Cartesian product.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Definition

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

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

The computation rule says that

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

Definition

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

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

The computation rule says that

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

In Lean we write $x.1$ for $\pi_1\ x$.

Definition

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

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

The computation rule says that

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

In Lean we write $x.1$ for $\pi_1\ x$. The function π_1 is also called fst.

Definition

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

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

The computation rule says that

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

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

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

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

so all terms of $A \times B$ are given by pair of elements.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The eliminator and the computation rule we gave are for non-dependent functions from $A \times B$: we also need a dependent version.

- Eliminators (dependent version). Let C be a function $(C : A \times B \to \mathrm{Type}\ u)$. Given a dependent function $(f : \prod_{(a:A)} \prod_{(b:B)} C\ (a, b))$, we have a well defined term $(\mathrm{rec}_{A \times B}\ f\ x : C\ x)$ for all $(x : A \times B)$.

- Eliminators (dependent version). Let C be a function $(C : A \times B \to \mathrm{Type}\ u)$. Given a dependent function $(f : \prod_{(a:A)} \prod_{(b:B)} C\ (a, b))$, we have a well defined term $(\mathrm{rec}_{A \times B}\ f\ x : C\ x)$ for all $(x : A \times B)$. This gives a dependent function
$$\left(\mathrm{rec}_{A \times B}\ f : \prod_{(x:A \times B)} C\ x\right)$$

- Eliminators (dependent version). Let C be a function ($C : A \times B \to \mathrm{Type}\ u$). Given a dependent function ($f : \prod_{(a:A)} \prod_{(b:B)} C\ (a, b)$), we have a well defined term ($\mathrm{rec}_{A\times B}\ f\ x : C\ x$) for all ($x : A \times B$). This gives a dependent function
$$\left(\mathrm{rec}_{A\times B}\ f : \prod_{(x:A\times B)} C\ x\right)$$
- Computation rules (dependent version). If we have terms ($a : A$) and ($b : B$) and a dependent function f as above, then
$$\mathrm{rec}_{A\times B}\ f\ (a, b) \equiv f\ a\ b.$$

- Eliminators (dependent version). Let C be a function $(C : A \times B \to \mathrm{Type}\ u)$. Given a dependent function $(f : \prod_{(a:A)} \prod_{(b:B)} C\ (a, b))$, we have a well defined term $(\mathrm{rec}_{A \times B}\ f\ x : C\ x)$ for all $(x : A \times B)$. This gives a dependent function
$$\left(\mathrm{rec}_{A \times B}\ f : \prod_{(x:A \times B)} C\ x\right)$$
- Computation rules (dependent version). If we have terms $(a : A)$ and $(b : B)$ and a dependent function f as above, then
$$\mathrm{rec}_{A \times B}\ f\ (a, b) \equiv f\ a\ b.$$

We have the same remarks as before.

Universal constructions (non-dependent version)

The (non-dependent) eliminator allows to construct a function $A \times B \to C$ given a function $A \to B \to C$.

Universal constructions (non-dependent version)

The (non-dependent) eliminator allows to construct a function $A \times B \to C$ given a function $A \to B \to C$.
In practice it is given by a (dependent!) function $\mathrm{rec}_{A \times B}$ of type

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

Universal constructions (non-dependent version)

The (non-dependent) eliminator allows to construct a function $A \times B \to C$ given a function $A \to B \to C$.
In practice it is given by a (dependent!) function $\mathrm{rec}_{A\times B}$ of type

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

The computation rule says

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

Universal constructions (non-dependent version)

The (non-dependent) eliminator allows to construct a function $A \times B \to C$ given a function $A \to B \to C$.
In practice it is given by a (dependent!) function $\mathrm{rec}_{A \times B}$ of type

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

The computation rule says

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

Note that in Lean the variable C is implicit.

Universal constructions (non-dependent version)

The (non-dependent) eliminator allows to construct a function $A \times B \to C$ given a function $A \to B \to C$.
In practice it is given by a (dependent!) function $\mathrm{rec}_{A\times B}$ of type

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

The computation rule says

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

Note that in Lean the variable C is implicit.
We have

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

Universal constructions (dependent version)

Similarly, the (dependent) eliminator is given by a function $\mathrm{rec}_{A\times B}$ of type

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

Universal constructions (dependent version)

Similarly, the (dependent) eliminator is given by a function $\mathrm{rec}_{A \times B}$ of type

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

The computation rule says that

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

Universal constructions (dependent version)

Similarly, the (dependent) eliminator is given by a function $\mathrm{rec}_{A\times B}$ of type

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

The computation rule says that

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

We will see that $\mathrm{rec}_{A\times B}$ is a special case of the *recursor* of an inductive type.

Dependent pair types

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

Dependent pair types

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

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

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

whose terms are called *dependent pairs*.

Definition

In “standard” mathematics a *magma* is a set M equipped with a binary operation $\cdot \colon M \times M \to M$.

Definition

In "standard" mathematics a *magma* is a set M equipped with a binary operation $\cdot \colon M \times M \to M$. More formally, we say that it is a pair $(M, \cdot \colon M \times M \to M)$.

Definition

In "standard" mathematics a *magma* is a set M equipped with a binary operation $\cdot \colon M \times M \to M$. More formally, we say that it is a pair $(M, \cdot \colon M \times M \to M)$.

In Lean, a magma is a type equipped with a binary operation

Definition

In "standard" mathematics a *magma* is a set M equipped with a binary operation $\cdot \colon M \times M \to M$. More formally, we say that it is a pair $(M, \cdot \colon M \times M \to M)$.

In Lean, a magma is a type equipped with a binary operation, so it is a pair such that the type of the second component, that is $M \times M \to M$, depends on the first one.

Definition

In "standard" mathematics a *magma* is a set M equipped with a binary operation $\cdot \colon M \times M \to M$. More formally, we say that it is a pair $(M, \cdot \colon M \times M \to M)$.

In Lean, a magma is a type equipped with a binary operation, so it is a pair such that the type of the second component, that is $M \times M \to M$, depends on the first one.

Definition

A *magma* is a term of type

$$\sum_{(M:\mathrm{Type}\ u)} (M \times M \to M)$$

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

$$(\langle a, b \rangle : \sum_{(a:A)} B\ a)$$

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

$$(\langle a, b\rangle : \sum_{(a:A)} B\ a)$$

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

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

$$(\langle a, b \rangle : \sum_{(a:A)} B\ a)$$

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

- Computation rules (non-dependent version): there is only one computation rule. If we have terms $(a : A)$ and $(b : B\ a)$, where notation is as above, then

$$\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ \langle a, b\rangle \equiv f\ a\ b.$$

- Computation rules (non-dependent version): there is only one computation rule. If we have terms $(a : A)$ and $(b : B\ a)$, where notation is as above, then

$$\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ \langle a, b \rangle \equiv f\ a\ b.$$

We can now define a function $(\pi_1 : \sum_{(a:A)} B\ a \to A)$ as above. It satisfies

$$\pi_1\ \langle a, b \rangle \equiv a$$

- Computation rules (non-dependent version): there is only one computation rule. If we have terms $(a : A)$ and $(b : B\ a)$, where notation is as above, then

$$\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ \langle a, b\rangle \equiv f\ a\ b.$$

We can now define a function $(\pi_1 \colon \sum_{(a:A)} B\ a \to A)$ as above. It satisfies

$$\pi_1\ \langle a, b\rangle \equiv a$$

Note that for π_2 we need the dependent version of the eliminator and of the computation rule.

- Eliminators (dependent version). Let C be a function $(C : \sum_{(a:A)} B\ a \to \mathrm{Type}\ u)$. Given a dependent function $(f : \prod_{(a:A)} \prod_{(b:B\ a)} C\ \langle a, b\rangle)$, we have a well defined term $(\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ x : C\ x)$ for all $(x : \sum_{(a:A)} B\ a)$.

- Eliminators (dependent version). Let C be a function $(C : \sum_{(a:A)} B\ a \to \mathrm{Type}\ u)$. Given a dependent function $(f : \prod_{(a:A)} \prod_{(b:B\ a)} C\ \langle a, b\rangle)$, we have a well defined term $(\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ x : C\ x)$ for all $(x : \sum_{(a:A)} B\ a)$. This gives a dependent function

$$\left(\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f : \prod_{(x:\sum_{(a:A)} B\ a)} C\ x\right)$$

- Eliminators (dependent version). Let C be a function $(C : \sum_{(a:A)} B\ a \to \mathrm{Type}\ u)$. Given a dependent function $(f : \prod_{(a:A)} \prod_{(b:B\ a)} C\ \langle a, b\rangle)$, we have a well defined term $(\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ x : C\ x)$ for all $(x : \sum_{(a:A)} B\ a)$. This gives a dependent function

$$\left(\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f : \prod_{(x:\sum_{(a:A)} B\ a)} C\ x\right)$$

- Computation rules (dependent version). If we have terms $(a : A)$ and $(b : B\ a)$ and a dependent function f as above, then

$$\mathrm{rec}_{\sum_{(a:A)} B\ a}\ f\ \langle a, b\rangle \equiv f\ a\ b.$$

Using the dependent version of the eliminator, we can now define π_2 as a dependent function

$$\left(\mathrm{snd} : \prod_{(x:\sum_{(a:A)} B\ a)} B\ x.1\right)$$

Using the dependent version of the eliminator, we can now define π_2 as a dependent function

$$\left(\mathrm{snd} : \prod_{(x : \sum_{(a:A)} B\ a)} B\ x.1\right)$$

- Uniqueness principle: if $(x : \sum_{(a:A)} B\ a)$, then

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

so all terms of $\sum_{(a:A)} B\ a$ are given by pair of elements.

Using the dependent version of the eliminator, we can now define π_2 as a dependent function

$$\left(\mathrm{snd} : \prod_{(x : \sum_{(a:A)} B\ a)} B\ x.1\right)$$

- Uniqueness principle: if $(x : \sum_{(a:A)} B\ a)$, then

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

  so all terms of $\sum_{(a:A)} B\ a$ are given by pair of elements. This implies that

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

  for all $(f : \prod_{(a:A)}(B\ a \to C))$.

All the remarks above are still true. For example, the uniqueness principle implies that all functions of type $\prod_{(x:\sum_{(a:A)} B\ a)} C\ x$ can be defined via π_1 and π_2, and this is what we do in practice.

Also the dependent pair type is a special case of an inductive type.

We also the analogues universal constructions. The (dependent) eliminator is given by a function $\mathrm{rec}_{\sum_{(a:A)} B\ a}$ of type

$$\left(\mathrm{rec}_{\sum_{(a:A)} B\ a} : \prod_{(C:\sum_{(a:A)} B\ a \to \mathrm{Type}\ u)} \left(\prod_{(a:A)} \prod_{(b:B\ a)} C\ \langle a, b\rangle\right) \to \prod_{(x:\sum_{(a:A)} B\ a)} C\ x\right)$$

The computation rule says that

$$\mathrm{rec}_{\sum_{(a:A)} B\ a}\ C\ f\ \langle a, b\rangle \equiv f\ a\ b$$

We also the analogues universal constructions. The (dependent) eliminator is given by a function $\mathrm{rec}_{\sum_{(a:A)} B\ a}$ of type

$$\left(\mathrm{rec}_{\sum_{(a:A)} B\ a} : \prod_{(C:\sum_{(a:A)} B\ a \to \mathrm{Type}\ u)} \left(\prod_{(a:A)} \prod_{(b:B\ a)} C\ \langle a, b\rangle\right) \to \prod_{(x:\sum_{(a:A)} B\ a)} C\ x\right)$$

The computation rule says that

$$\mathrm{rec}_{\sum_{(a:A)} B\ a}\ C\ f\ \langle a, b\rangle \equiv f\ a\ b$$