Dependent types

Samuel Mimram

2025

École polytechnique

The logical setting

We work in *Martin-Löf type theory* with

- a hierarchy of universes: $\mathcal{U}_i$
- Π- and Σ-types: $\Pi(x : A).P(x)$, $\Sigma(x : A).P(x)$
- natural numbers: $\mathbb{N}$
- identity types: $x = y$

This implies that we also have

- functions types: $A \to B$
- products and coproducts: $A \times B$, $A \sqcup B$

When needed, we also require

- higher inductive types

which subsume them all.

The logical setting

Here,

- we work in a semi-formal way (no sequent calculus)
- we need to introduce other constructions so that we have to know how it works
- we (informally) introduce the semantics of type constructions

Types

In type theory, everything has a **type**:

$$t : A$$

For instance:

$$3 : \mathbb{N} \qquad\qquad \lambda n.(n+1, \mathsf{true}) : \mathbb{N} \to \mathbb{N} \times \mathsf{Bool}$$

In particular, we write $\mathcal{U}$ for the type of all types

$$\mathbb{N} : \mathcal{U} \qquad\qquad \mathsf{Bool} \to \mathbb{N} \times \mathbb{N} : \mathcal{U}$$

The type $\mathcal{U}$ can also be understood as the type of **propositions**

$$\mathsf{isEven} : \mathbb{N} \to \mathcal{U}$$

Semantically, the elements of $\mathcal{U}$ are **spaces**.

Contexts

$\Gamma \hat{=} x_A : A_1, x_2 : A_2, \ldots, x_n : A_n$

TODO: $\Gamma \vdash A$

$$\dfrac{\dfrac{\dfrac{}{A, B \vdash A}}{A \vdash B \to A}}{\vdash A \to B \to A}$$

context is handled implicitly unless we really need to mention those

Equalities

There are two notions of equality in our type theory

- definitional equality: $t \mathrel{\hat{=}} u$
- propositional equality: $t = u$

Definitional equality is the identification performed implicitly by the proof assistant, e.g.

$$2 + 2 \mathrel{\hat{=}} 4 \qquad\qquad (\lambda n.n + 2)\,5 \mathrel{\hat{=}} 7$$

Propositional equality is a proposition, for which we have explicit proof terms, e.g.

$$\Pi(m : \mathbb{N}).\Pi(n : \mathbb{N}).(m + n = n + m)$$

It will play a central role here.

Functions

We write

$$A \to B$$

for the type of functions from A to B.

Such a function can be defined by a λ-abstraction:

$$f : \mathbb{N} \to \mathbb{N} \qquad\qquad f \mathrel{\hat{=}} \lambda n.(n + 2)$$

and we can apply $f : A \to B$ to $a : A$, written $f\, a$.

Moreover, those satisfy the rules of β-reduction and η-expansion

$$(\lambda x.t)u \mathrel{\hat{=}} t[u/x] \qquad\qquad t \mathrel{\hat{=}} \lambda x.t\, x$$

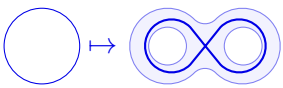
e.g.

$$(\lambda n.(n + 2))\, 3 \mathrel{\hat{=}} 3 + 2 \qquad\qquad \sin \mathrel{\hat{=}} \lambda x.\sin x$$

Functions

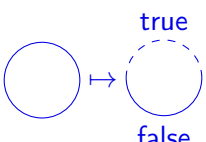
Semantically, $f : A \to B$ should be understood as a *continuous* function, e.g.

$$f : \mathsf{S}^1 \to \mathsf{S}^1 \vee \mathsf{S}^1$$



but not

$$f : \mathsf{S}^1 \to \mathsf{Bool}$$



Functions

A function

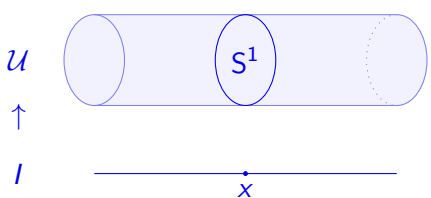
$$f : A \to \mathcal{U}$$

is called a **type family**.

It can also be seen as a continuous family of spaces. For instance,

$$\begin{aligned} f : I &\to \mathcal{U} \\ x &\mapsto \mathsf{S}^1 \end{aligned}$$

can be pictured as



Dependent functions

Given a type A and a type family $B : A \to \mathcal{U}$, we write

$$\Pi(x : A).B(x) \qquad \text{or} \qquad (x : A) \to B(x)$$

for the type of **dependent functions**.

For instance,

$$\begin{aligned} \text{zeroes} : \Pi(n : \mathbb{N}).\ &\text{Vec}\ \mathbb{R}\ n \\ n \mapsto\ &[0., 0., \ldots, 0.] \end{aligned}$$

Those generalize arrow types, e.g.

$$\mathbb{N} \to \mathbb{N} \qquad \hat{=} \qquad \Pi(x : \mathbb{N}).\mathbb{N}$$

Type constructors

In order to specify a construction on types we need to specify

1. a **type former**: a formal operation on types
2. **constructors**: to create terms of the new type
3. **eliminators**: to use terms of the type
4. **computation** rules: how eliminator behave on constructors
5. **uniqueness** rules: how to express any term of the new type from constructors

(also congruence rules, which will not be mentioned here)

Note: we already did this for Π-types.

Products

Type former: $- \times - : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$.

Constructor:

$$(-,-) : A \to B \to A \times B$$

Eliminators:

$$\mathsf{fst} : A \times B \to A \qquad \mathsf{snd} : A \times B \to B$$

Computation rules (β-conversion):

$$\mathsf{fst}\,(a,b) \mathrel{\hat{=}} a \qquad \mathsf{snd}\,(a,b) \mathrel{\hat{=}} b$$

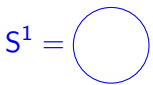
Uniqueness rules (η-conversion): for $x : A \times B$,

$$x \mathrel{\hat{=}} (\mathsf{fst}\ x, \mathsf{snd}\ x)$$

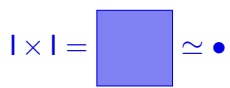
Products: semantics

Semantically, the space interpreting $A \times B$ is the cartesian product of A and B.

Writing



we have



Coproducts

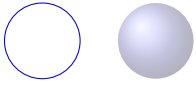
The **coproduct** or **sum** of two types is noted

$$A \sqcup B$$

It corresponds to the disjoint union.

For instance,

$$S^1 \sqcup S^2 =$$



Coproducts: rules

Type former: $\sqcup : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$.

Constructors:

$$\mathsf{inl} : A \to A \sqcup B \qquad\qquad \mathsf{inr} : B \to A \sqcup B$$

Eliminator:

$$\mathsf{elim} : (C : A \sqcup B \to \mathcal{U}) \to ((a : A) \to C\,(\mathsf{inl}\,a)) \to ((b : B) \to C\,(\mathsf{inr}\,b)) \to (x : A \sqcup B) \to C\,x$$

which corresponds to the inductive definition

$$h(\mathsf{inl}\ a) \hat{=} f\ a$$
$$h(\mathsf{inr}\ b) \hat{=} g\ b$$

Computation rules:

$$\mathsf{elim}\ C\ f\ g\,(\mathsf{inl}\ a) \hat{=} f\ a \qquad\qquad \mathsf{elim}\ C\ f\ g\,(\mathsf{inr}\ b) \hat{=} g\ b$$

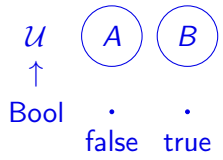
Dependent sums

We would like to generalize coproducts to countable (or more) types:

$$A_1 \sqcup A_2 \sqcup \dots$$

The two types can be encoded as a family

$$\begin{aligned} F : \mathsf{Bool} &\to \mathcal{U} \\ \mathsf{false} &\mapsto A \\ \mathsf{true} &\mapsto B \end{aligned}$$

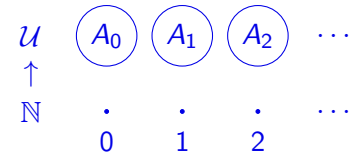


and their sum will be $\Sigma(x : \mathsf{Bool}).F\,x$.

The countable case is similarly encoded by

$$F : \mathbb{N} \to \mathcal{U}$$

and the sum is $\Sigma(n : \mathbb{N}).F\,n$.



Dependent sums

Given a type A and a type family $B : A \to \mathcal{U}$, we have a **Σ-type**

$$\Sigma(x : A).B(x) \qquad \text{or} \qquad (x : A) \times B(x)$$

which can be understood as

- the coproduct indexed by A,
- the type of pairs (x, y) with $x : A$ and $y : B(x)$,
- the proofs that there exists an element of A satisfying B.

Dependent sums: rules

Type former: $\Sigma : (A : \mathcal{U}) \to (B : A \to \mathcal{U}) \to \mathcal{U}$.

Constructor:

$$(-,-) : (x : A) \to B\,x \to \Sigma(x : A).B(x)$$

Eliminators:

$$\mathsf{fst} : \Sigma(x : A).B(x) \to A \qquad \mathsf{snd} : (p : \Sigma(x : A).B(x)) \to B(\mathsf{fst}\ p)$$

Computation rules:

$$\mathsf{fst}\,(a, b) \mathrel{\hat{=}} a \qquad \mathsf{snd}\,(a, b) \mathrel{\hat{=}} b$$

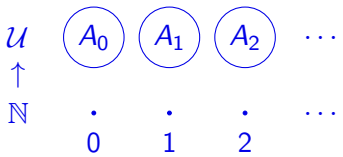
Uniqueness rules: for $x : \Sigma(x : A).B(x)$,

$$x \mathrel{\hat{=}} (\mathsf{fst}\ x, \mathsf{snd}\ x)$$

Dependent sums: semantics

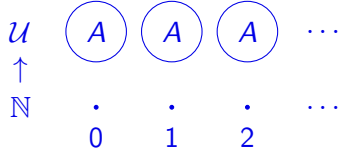
We have seen that the semantics of $\Sigma\mathbb{N}.F$ for a family

$$F : \mathbb{N} \to \mathcal{U}$$



is the disjoint union of the family (and the first projection is the vertical one).

In particular, for $A_i = A$, we obtain $\mathbb{N} \times A$:

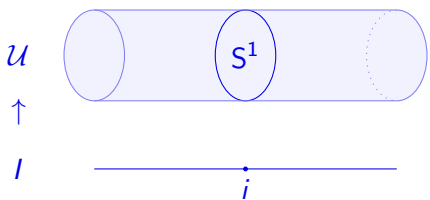


More generally, for $A, B : \mathcal{U}$, $\Sigma(x : A).(\lambda x.B) = A \times B$.

Dependent sums: semantics

In the continuous case, $\Sigma A.B$ is the **total space** of B.

For instance, with $B : \mathsf{I} \to \mathcal{U}$ such that $B(i) = \mathsf{S}^1$



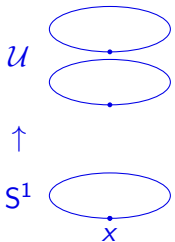
we have $\Sigma\,\mathsf{I}\,.B$ is the torus $\mathsf{I} \times \mathsf{S}^1$, i.e. S^1.

Dependent sums: semantics

Can we do more than trivial families of types? No and yes :)

Consider $B : A \to \mathcal{U}$. Given A which is connected all the fibers $B\,x$ have to be the same.

For instance, we have a family $B : \mathsf{S}^1 \to \mathcal{U}$ with Bool as fibers:



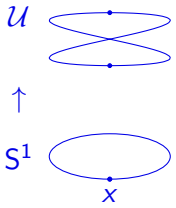
We have

$$\Sigma(x : \mathsf{S}^1).B\,x \quad = \quad \mathsf{Bool} \times \mathsf{S}^1 = \mathsf{S}^1 \sqcup \mathsf{S}^1$$

Dependent sums: semantics

However, the family can still be (globally) non-trivial.

We have another family $B : \mathsf{S}^1 \to \mathcal{U}$ with fibers Bool:



The total space is

$$\Sigma(x : \mathsf{S}^1).B\,x \quad = \quad \mathsf{S}^1$$

and the projection map $\mathsf{fst} : \mathsf{S}^1 \to \mathsf{S}^1$ is the "double speed" map.

real Hopf fibration $\mathsf{S}^0 \to \mathsf{S}^1 \to \mathsf{S}^1$

Falsity

We have a type **false** noted

$$\bot \qquad \text{or} \qquad 0$$

whose semantics is the empty space.

There is no introduction rule!

The elimination rule is

$$\text{rec} : \bot \to A$$

Truth

We have a type **true** noted

$$\top \qquad \text{or} \qquad 1$$

whose semantics is the space with one point.

The introduction rule is

$$\star : \top$$

The elimination rule is

$$\text{rec} : (\top \to A) \to A$$

(this is quite useless).

Booleans

We have the type of booleans:

$$\mathsf{Bool}$$

The introduction rules are

$$\mathsf{false} : \mathsf{Bool} \qquad\qquad \mathsf{true} : \mathsf{Bool}$$

The (dependent) eliminator is

$$\mathsf{elim} : (A : \mathsf{Bool} \to \mathcal{U}) \to A\,\mathsf{false} \to A\,\mathsf{true} \to (b : \mathsf{Bool}) \to A\,b$$

Computation rules are

$$\mathsf{elim}\ A\,t\,u\,\mathsf{false} \mathrel{\hat{=}} t \qquad\qquad \mathsf{elim}\ A\,t\,u\,\mathsf{true} \mathrel{\hat{=}} u$$

Uniqueness rules are

$$....TODO...$$

Natural numbers

We have a type of natural numbers

$$\mathbb{N}$$

The introduction rules are

$$0 : \mathbb{N} \qquad\qquad \mathsf{suc} : \mathbb{N} \to \mathbb{N}$$

The eliminator is

$$\mathsf{elim}(A : \mathbb{N} \to \mathcal{U}) \to A\,0 \to ((n : \mathbb{N}) \to A\,n \to A\,(\mathsf{suc}\,n)) \to (n : \mathbb{N}) \to A\,n$$

This is the usual recurrence!

Finite types

We have a type family

$$\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$$

which to $n : \mathbb{N}$ associates the type

$$\mathsf{Fin}\ n$$

with n elements $(0, 1, \ldots, n-1)$.

In particular,

$$\bot = \mathsf{Fin}\ 0 \qquad \top = \mathsf{Fin}\ 1 \qquad \mathsf{Bool} = \mathsf{Fin}\ 2$$

Universes

The **universe** is written

$$\mathcal{U} \qquad \text{or} \qquad \mathsf{Type}$$

its elements are **types**.

We have a problem however: if we set $\mathcal{U} : \mathcal{U}$ we have an inconsistency (intuitively, $\mathcal{U}$ is too big to be a type).

In order to avoid this, we have a type $\mathcal{U}_1$ of "big types" and set $\mathcal{U} : \mathcal{U}_1$.

We have to continue like this

$$\mathcal{U} \hat{=} \mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots$$

Universes

All type formers are available on all universe levels ℓ, e.g.

$$\begin{array}{llll} - \times - : \mathcal{U}_\ell & \to \mathcal{U}_\ell & \to \mathcal{U}_\ell \\ A & B & \mapsto A \times B \end{array}$$

Moreover, it is sometimes useful to allow for heterogeneous levels:

$$- \times - : \mathcal{U}_\ell \to \mathcal{U}_{\ell'} \to \mathcal{U}_{\ell \vee \ell'}$$

Identity types

Finally, the type which will be of main interest for us:
identity types / propositional equality.

For any type A and $x, y : A$, we write

$$x =^A y \qquad \text{or} \qquad x = y$$

for the type of proofs of **identities/equalities/paths** between x and y.

Identity types: rules

Type former:

$$- = - : (A : \mathcal{U}) \to A \to A \to \mathcal{U}$$

Constructor:

$$\mathsf{refl} : (x : A) \to x = x$$

Eliminator:

$$\begin{aligned} \mathsf{J} : (A : \mathcal{U}) \to (x : A) \to (P : (y : A) \to x = y \to \mathcal{U}) \to \\ P\, x\, (\mathsf{refl}\ x) \to \\ (y : A) \to (p : x = y) \to P\, y\, p \end{aligned}$$

Computation:

$$\mathsf{J}\ A\, x\, P\, r\, x\, (\mathsf{refl}\ x) \mathrel{\hat{=}} r$$

Uniqueness: ???

Identity types: semantics

Given a type A and $x, y : A$, the type $x = y$ is the type of **paths** form x to y in A.

This means continuous maps

$$p : \mathsf{I} \to A$$

with $\mathsf{I} = [0, 1]$ such that $p(0) = x$ and $p(1) = y$.

For this reason, J is sometimes called **path induction**: in order to prove a property on all paths, it is enough to prove it for refl.

Inductive types

Modern languages feature inductive types.

All the previous constructions can be implemented as particular inductive types, e.g.

```
data ℕ : Type where
  zero : ℕ
  suc  : ℕ → ℕ
```

or

```
data _⊔_ (A : Type) (B : Type) : Type where
  inl : A → A ⊔ B
  inr : B → A ⊔ B
```

Inductive types

Even the identity type can be defined as an inductive one:

```
data _≡_ {A : Type} (x : A) : (y : A) → Type where
  refl : x ≡ x
```