Introduction to

Type Theory

Vincent Moreau
for the catsup working group

based on the HoTT book

Friday the 20th of October 2023

Plan

1. A few words on types
   - History : Russell, Church, Martin-Löf
   - Curry-Howard : $A \wedge B$ $\quad A \times B$
     propositions as types
   - Equality and homotopy
   - Many other directions : linear logic
2. First concepts of type theory
   - Every term is typed
   - Two equalities : $\equiv$ and $=$
   - Contexts and open terms
3. Introduction to the syntax
   - $\to$, $\mathcal{U}$, Π (preliminaries)
   - $\times$, Σ and the axiom of choice
   - $\mathbb{N}$, $=$

① A few words on types

- History:
  - Russell proposed type theory for a foundations of mathematics
  - Church: λ-calculus, a theory of functions & algorithm, and simple types like

$$A \to (A \to A) \quad \text{with} \quad \begin{array}{l} f(a)(a') = a \\ g(a)(a') = a' \end{array}$$

λ-calculus $\neq$ type theory

  - Martin-Löf: intuitionistic type theory, with dependent types

$$\prod_{n \in \mathbb{N}} \big((n+1 = 0) \to \bot \big)$$

  - Curry-Howard, propositions as types.

proofs = inhabitants (terms)

$$\begin{array}{ccccc} A \wedge B & A \vee B & A \Rightarrow B & \forall x.\, B(x) & \exists x.\, B(x) \\ \rightsquigarrow & \rightsquigarrow & \rightsquigarrow & \rightsquigarrow & \rightsquigarrow \\ A \times B & A + B & A \to B & \prod_{x:A} B(x) & \sum_{x:A} B(x) \end{array}$$

- Equality : for a, b of type A, there is a type $a = b$ which represents identifications between a and b.

First intuition :

- if $a = b$ is empty, no proof that a and b are the same
- if $a = b$ is non-empty, a and b are the same, but can be identified in multiple ways.

We represent $p : a = b$ by : $a \overset{p}{\text{———}} b$

We can then iterate, for $p, q : a = b$,

$r : p = q$ represents

$$\begin{tikzcd} a \arrow[r, bend left, "p"] \arrow[r, bend right, "q"'] \arrow[r, phantom, "\Downarrow r"] & b \end{tikzcd}$$

Under specified :

- ETT, $p = q$ collapses to a point
- HoTT, we accept the higher order equalities and actively use them.

also linear logic ...

② First concepts of type theory

- Judgments: if A is a type, then

$$a : A \quad \text{means} \quad a \text{ has type } A$$

which also means, under the propositions as types point of view, that

$$a \text{ is a proof of } A.$$

Terms are never in isolation, they are intrisically typed. This differs from set theory, where a formula

$$\forall n \in \mathbb{N}, \; P(n)$$

closer to actual mathematical practice

is an abbreviation for

$$\forall n, \; n \in \mathbb{N} \Rightarrow P(n)$$

which quantifies on all the entities n, regardless of there nature. In contrast, we have in type theory

$$\prod_{n : \mathbb{N}} P(n)$$

mandatory!

- The point where type theory differs the most from set theory is in its treatment of equality.

We said that, given $a, b : A$, we have a <u>type</u>

$$a = b \qquad \left(\begin{array}{c}\text{propositional}\\ \text{equality}\end{array}\right)$$

of identifications of a with b.

Type theory is also a programming language, and hence has a notion of computation. For instance,

$$(x \mapsto x+1)(2) \quad \text{computes to} \quad 3 \qquad (*)$$

In general, we have a judgment

$$a \equiv b : A$$

for $a, b : A$, which holds when a and b are the same modulo computations like $(*)$.

- Type theory comes with a notion of context, which contains all the free variables of an expression. Such a context is usually written as Γ before a turnstile, as in

$$\Gamma \vdash a : A \quad \text{and} \quad \Gamma \vdash a \equiv b : A$$

open: $m : \mathbb{N} \vdash m+1 : \mathbb{N}$

closed: $\vdash m \mapsto m+1 : \mathbb{N} \to \mathbb{N}$

For example:

- $m : \mathbb{N},\ n : \mathbb{N} \vdash m + n : \mathbb{N}$
- $a : A,\ b : A,\ p : a = b \vdash p^{-1} : b = a$

Under the propositions as types point of view, the context can be thought of as the assumptions or the hypotheses used to derive a proof.

In the following, we follow the HoTT book and choose not to write contexts, "Informal type theory".

(3) Introduction to the syntax

We now give the definitions of multiple types. The pattern is fairly general and contains the following elements:

- formation rules to build the new type

  e.g. A, B types $\rightsquigarrow$ $A \times B$ type

e.g. negative non-dependent product

- constructors, to build terms of the new type

  e.g. $a : A,\ b : B \rightsquigarrow \langle a, b \rangle : A \times B$

- destructors, or induction principles, to use the terms of the new type

  e.g. $p : A \times B \rightsquigarrow p.1 : A$

  $p : A \times B \rightsquigarrow p.2 : B$

- conversion rules to relate constructors & destructors

  e.g. $\langle a, b \rangle .1 \equiv a$, $\langle a, b \rangle .2 \equiv b$

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

Function type

If A and B are two types, then $A \to B$ is a type.

To <u>construct</u> terms of type $A \to B$, we give an equation $f(x) :\equiv \phi$ where $x : A$ is assumed and $\phi : B$ given that it may contain x.

Ex: given a type $\mathbb{N}$ of natural numbers,

$$f(x) :\equiv x+1 \quad \text{with} \quad f : \mathbb{N} \to \mathbb{N}$$

The term f is usually written $\lambda(x:A).\phi$

To <u>use</u> terms of type $A \to B$, we can apply them to terms of type A to get terms of type B. Ex: $(\lambda x.x+1)(2) : \mathbb{N}$.

It <u>computes</u> using substitution. Ex: $(\lambda x.x+1)(2)$

$$\begin{array}{c} \equiv \\ 2+1 \end{array}$$

Universes

We want to use types as terms of the calculus, i.e. as first-class elements which can be inputs, outputs, variables, etc.

For this, we introduce universes, which are types whose terms are also types.

We have an infinite hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \cdots\cdots$$

Universes are complex, and we choose here to focus on other aspects, so we will simply write

$$A : \mathcal{U} \quad \text{to mean that} \quad A \text{ is a type}$$

Universes are handy to denote families of types. For example, if $\mathbb{N}$ is a type of natural numbers:

predicates!

- $\mathrm{Fin} : \mathbb{N} \longrightarrow \mathcal{U}$
- $\mathrm{Vec} : \mathbb{N} \longrightarrow \mathcal{U}$ for some $A : \mathcal{U}$

Dependent functions

This is like functions, but whose codomain may vary.

For any type $A : \mathcal{U}$ and type family $B : A \to \mathcal{U}$, we have the type

$$\prod_{x:A} B(x)$$

which may be identified to $A \to B$ when B does not depend on A.

<u>Constructor</u> : $\lambda(x:A).\,\phi$, where ϕ may contain x

Ex : $\lambda(n:\mathbb{N}).\ \iota(n) \ : \ \prod_{n:\mathbb{N}} \mathrm{Fin}(n)$

<u>Destructor</u> : application to $a : A$, which yields a term of $B(a)$. Computation is substitution.

When $B : \mathcal{U} \to \mathcal{U}$, we have a function taking a type as argument $\to$ polymorphism

Product types

If A and B are types, then $A \times B$ is a type.

<u>Constructor</u>: if $a : A$ and $b : B$, then

$$\langle a, b \rangle : A \times B$$

For the induction principle, consider the idea that a function out of $A \times B$ is defined. The pairs $\langle a, b \rangle$ are "canonical" terms.

This idea yields the following <u>recursor</u>

$$\mathrm{rec}_{A \times B} : \prod_{C : \mathcal{U}} (A \to B \to C) \to A \times B \to C$$

together with the computation rule:

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

We can define the projections as

$$\mathrm{pr}_1 = \mathrm{rec}_{A \times B}(A, \lambda a \lambda b . a)$$

$$\mathrm{pr}_2 = \mathrm{rec}_{A \times B}(B, \lambda a \lambda b . b).$$

However, this does not use all the power of dependent types!

Given $A, B : \mathcal{U}$, we have

$$\mathrm{ind}_{A\times B} : \prod_{C : A\times B \to \mathcal{U}} \left(\prod_{(a:A)} \prod_{(b:B)} C(\langle a,b\rangle) \right) \to \prod_{p : A\times B} C(p)$$

with the defining equation that for $a:A$, $b:B$,

$$\mathrm{ind}_{A\times B}(C, g, \langle a,b\rangle) \equiv g(a)(b)$$

This is the full <u>destructor</u> of the product type.

We can use it to prove that, for $p : A\times B$,

$$\langle \mathrm{pr}_1(p), \mathrm{pr}_2(p)\rangle = p$$

is inhabited. The family $C : A\times B \to \mathcal{U}$ is

$$C(p) :\equiv \underbrace{\langle \mathrm{pr}_1(p), \mathrm{pr}_2(p)\rangle = p}_{\text{the equality type}}$$

Suppose that we always have a term $\mathrm{refl}_x : x = x$.

We get : $\mathrm{ind}_{A\times B}(C, \lambda a\, \lambda b.\, \mathrm{refl}_{\langle a,b\rangle})$

Dependent pairs

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

$$\sum_{x:A} B(x) \quad : \quad \mathcal{U}$$

<u>Construction</u> : $\langle a, b \rangle : \sum_{x:A} B(x)$.

<u>Recursor.</u> If we have a type C and a function $g : \prod_{x:A} B(x) \to C$, then we can build $f : \left(\sum_{x:A} B(x)\right) \to C$

More formally, we have a term

$$\mathrm{rec}_{\Sigma AB} : \prod_{C:\mathcal{U}} \left(\prod_{x:A} B(x) \to C\right) \to \sum_{x:A} B(x) \to C$$

This way, we can construct the first projection, as

$$pr_1(x) :\equiv rec_{\Sigma AB}(A, \lambda a \lambda b.a, x)$$

The second projection, however, is

$$pr_2 : \prod_{x:\Sigma AB} B(pr_1(x))$$

and cannot be defined using the recursor $\rightarrow$ need the dependent <u>induction</u>!

$$ind_{\Sigma AB} : \prod_{C:\Sigma AB \to \mathcal{U}} \left(\prod_{(a:A)} \prod_{(b:B(a))} C(\langle a,b \rangle) \right) \to \prod_{p:\Sigma AB} C(p)$$

with the computation rule:

$$ind_{\Sigma AB}(C, g, \langle a, b \rangle) \equiv g(a)(b)$$

Then, the second projection is

$$ind_{\Sigma AB}(\lambda p. B(pr_1(p)). \lambda a \lambda b. b) .$$

Notice that, under the propositions as types point of view, the type representing the <u>axiom of choice</u> is, for some relation $R : A \to B \to \mathcal{U}$,

$$\left(\prod_{(a:A)} \sum_{(b:B)} R(a)(b) \right) \to \sum_{(f:A\to B)} \prod_{(a:A)} R(a)(f(a))$$

Indeed, we work in constructive mathematics, so a term of type

$$H : \prod_{(a:A)} \sum_{(b:B)} R(a)(b)$$

can be thought of as a function associating to each $a : A$ a pair

$$\langle b, t \rangle$$

where $b : B$ and $t : R(a)(b)$, which is seen as a proof that a and b are related by R. It then suffices to extract the function $a \mapsto b$.

More formally : $\lambda H . \langle \lambda a . pr_1(H(a)) . \lambda a . pr_2(H(a)) \rangle$.

Natural numbers

The ideas of Peano can be used in type theory to define a type $\mathbb{N}$ whose terms behave like natural numbers.

<u>Constructors</u>: $0 : \mathbb{N}$ and, if $n : \mathbb{N}$, we have $s(n) : \mathbb{N}$.

<u>Destructors</u>: the destructor for $\mathbb{N}$ is the usual induction principle:

$$\mathrm{ind}_{\mathbb{N}} : \prod_{C : \mathbb{N} \to \mathcal{U}} C(0) \to \left(\prod_{n : \mathbb{N}} C(n) \to C(n+1) \right) \to \prod_{n : \mathbb{N}} C(n)$$

which justifies the name induction!

We have the <u>computation</u> rules

$$\mathrm{ind}_{\mathbb{N}}(C, t_0, t_s, 0) \equiv t_0$$

$$\mathrm{ind}_{\mathbb{N}}(C, t_0, t_s, s(n)) \equiv t_s(n, \mathrm{ind}_{\mathbb{N}}(C, t_0, t_s, n)).$$

Notice that all functions terminate, so in particular the ones of type $\mathbb{N} \to \mathbb{N}$.

Identity types

The identity types behave very differently from the equality of set theory. As type theory has functions, it has a diagonal

$$\Delta : A \to A \times A \qquad \left(\begin{array}{c}\text{this is}\\ \text{our intuition}\end{array}\right)$$

$$\Delta := \lambda a . \langle a, a \rangle$$

In a way, the identity type is defined as the image of this map.

For every $A : \mathcal{U}$, $a, b : A$, we have a type

$$a = b : \mathcal{U}$$

<u>Constructor</u>: For every $a : A$, we have a term

$$\mathrm{refl}_a : a = a$$

This is what we have used to prove

$$\prod_{p : A \times B} \langle pr_1(p), pr_2(p) \rangle = p$$

Notice that, given $p, q : a = b$, we have a type

$$p = q$$

and ad infinitum...

What is the destructor / induction principle for equality types? To define a function out of $a = b$, it should suffice to have it defined on the canonical elements of $a = b$. These are the reflexivities.

We therefore get the <u>J-rule</u>, which to

(destructor) $$C : \prod_{a,b:A} a = b \longrightarrow \mathcal{U}$$

associates the term

$$J(C) : \left(\prod_{a:A} C(a, a, \mathrm{refl}_a) \right) \longrightarrow \prod_{a,b:A} \prod_{p:a=b} C(a,b,p)$$

This means that to define a function

$$f : \prod_{a,b:A} \prod_{p:a=b} C(a,b,p)$$

it suffices to have a function

$$f_r : \prod_{a:A} C(a, a, \mathrm{refl}_a)$$

Computation rule: $J(C, f_r, a, a, \mathrm{refl}_a) \equiv f_r(a)$

Using the J-rule, we can prove that equality is symmetric. This amounts to building a term of type

$$a = b \longrightarrow b = a .$$

For this, we consider the family

$$C(a, b, p) \equiv b = a$$

which does not depend on $p : a = b$. To use the J-rule, we must provide a term of type

$$\prod_{a:A} C(a, a, \mathrm{refl}_a) \equiv a = a$$

which can be built using reflexivity.

Therefore, we get a term

$$J(\lambda a \lambda b \lambda p.\, b = a,\ \lambda a.\, \mathrm{refl}_a) : \prod_{(a,b:A)} a = b \to b = a$$

In the same way, we also get transitivity, and higher coherences. All together, these equip each type with a structure of <u>∞-groupoid</u>.