Univalent Foundations

I. Type theory

Paige Randall North (slides from Benedikt Ahrens)

What is a foundation of mathematics?

A foundation of mathematics is specified by three things:

1. Syntax for mathematical objects
2. Notion of proposition and proof
3. Interpretation of the syntax into the world of mathematical objects

In this course, we discuss several foundations:

- Martin-Löf type theory with an interpretation in sets
- Martin-Löf type theory with an interpretation in propositions
- Univalent type theory with an interpretation in simplicial sets (Univalent Foundations)

Outline

1. The syntax of type theory and an interpretation in sets
2. An interpretation of type theory in propositions

Outline

Type theory

Type theory is. . .

- A (functional programming) language of types and terms, similar to functional programming languages
- with the infrastructure for writing mathematical statements and proofs

Important features of Martin-Löf type theory

- **Dependent types and functions**, e.g., type $\mathsf{Vect}(n)$ of vectors of length n:

$$\mathsf{concatenate} : \prod_{m,n:\mathsf{Nat}} \mathsf{Vect}(m) \to \mathsf{Vect}(n) \to \mathsf{Vect}(m+n)$$

$$\mathsf{tail} : \prod_{n:\mathsf{Nat}} \mathsf{Vect}(1+n) \to \mathsf{Vect}(n)$$

- All functions are total

Our goal

Our main goal: to write well-typed programs

In type theory, both the activities of

- implementing an algorithm
- proving a mathematical statement

are done by writing well-typed programs.

We hence need to understand the **typing rules** of type theory.
These rules are expressed in a logical language consisting of "judgements" and "inference rules".

Syntax of type theory

Fundamental: **judgment**

$$\text{context} \;\vdash\; \text{conclusion}$$

Contexts & judgments

Γ	sequence of variable declarations
	$(x_1 : A_1), (x_2 : A_2(x_1)), \ldots, (x_n : A_n(\vec{x}_i))$
$\Gamma \;\vdash\; A$	A is well–formed **type** in context Γ
$\Gamma \;\vdash\; a : A$	**term** a is well-formed and of type A
$\Gamma \;\vdash\; A \equiv B$	types A and B are **convertible**
$\Gamma \;\vdash\; a \equiv b : A$	a is convertible to b in type A

$$(x : \mathsf{Nat}), (f : \mathsf{Nat} \to \mathsf{Bool}) \;\vdash\; f(x) : \mathsf{Bool}$$

An example

Suppose you want to write a function isZero? of type Nat → Bool. You start out with

$$\begin{aligned} &\mathsf{isZero?} : \mathsf{Nat} \to \mathsf{Bool} \\ &\mathsf{isZero?}(n) :\equiv \ ?? \end{aligned}$$

At this point, you need to write a term $b(n)$ such that

$$(n : Nat) \ \vdash \ b(n) : \mathsf{Bool}$$

Inference rules and derivations

- An **inference rule** is an implication of judgments,

$$\frac{J_1 \qquad J_2 \qquad \ldots}{J}$$

e.g.,

$$\frac{\Gamma \ \vdash \ f : \mathsf{Nat} \to \mathsf{Bool} \qquad \Gamma \ \vdash \ n : \mathsf{Nat}}{\Gamma \ \vdash \ f@n : \mathsf{Bool}} \qquad \frac{\Gamma \ \vdash \ a \equiv b : A}{\Gamma \ \vdash \ b \equiv a : A}$$

- A **derivation of a judgment** is a tree of inference rules.
e.g., writing Γ for the context $(f : \mathsf{Nat} \to \mathsf{Bool}), (n : \mathsf{Nat})$

$$\frac{\overline{\Gamma \ \vdash \ f : \mathsf{Nat} \to \mathsf{Bool}} \qquad \overline{\Gamma \ \vdash \ n : \mathsf{Nat}}}{\Gamma \ \vdash \ f(n) : \mathsf{Bool}}$$

- We sometimes omit the context when writing judgments.

- We abbreviate the above to, e.g., “ If $a \equiv b$, then $b \equiv a$”.

Interpreting types as sets?

- Can interpret types and terms as sets
- $a : A$ is interpreted as $\lfloor a \rfloor \in \lfloor A \rfloor$

Differences between $a : A$ and $a \in A$

- the judgment $a : A$ is **not** a statement that can be proved or disproved within type theory
- term a does not exist independently of its type A
- a well-formed term a has exactly one type up to $\equiv$, whereas a set a can be member of many different sets

Important facts about convertibility

- If $x : A$ and $A \equiv B$ then $x : B$
- $\equiv$ is a congruence, e.g., if $a \equiv a'$ then $f@a \equiv f@a'$

Declaring types & terms

Any type and its terms are declared by giving 4 (groups of) rules:

Formation: a way to construct a new type

Introduction: way(s) to construct **canonical terms** of that type

Elimination: how to use a term of the new type to construct terms of other types

Computation: what happens when one does Introduction followed by Elimination

The type of functions $A \to B$

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

Introduction If $(x : A) \vdash b : B$, then $\vdash \lambda(x : A).b(x) : A \to B$

Elimination If $f : A \to B$ and $a : A$, then $f@a : B$

Computation $(\lambda(x : A).b)@a \equiv b[a/x]$

- **Substitution** $b[a/x]$ is built-in
- Notational convention: write $f(a)$ for $f@a$ — beware of potential confusion
- Interpretation in sets: Set of functions from A to B

The singleton type

Formation $\mathbf{1}$ is a type

Introduction $\mathsf{t} : \mathbf{1}$

Elimination If $x : \mathbf{1}$ and C is a type and $c : C$, then $\mathsf{rec}_{\mathbf{1}}(C, c, x) : C$

Computation $\mathsf{rec}_{\mathbf{1}}(C, c, \mathsf{t}) \equiv c$

- Interpretation in sets: a one-element set, $\mathsf{t} \in \mathbf{1}$

Booleans

Formation

Introduction

Elimination

Computation

Booleans

Formation Bool is a type

Introduction true : Bool, false : Bool

Elimination If x : Bool and C is a type and $c, c' : C$, then
$\mathsf{rec}_{\mathsf{Bool}}(C, c, c', x) : C$

Computation $\mathsf{rec}_{\mathsf{Bool}}(C, c, c', \mathsf{true}) \equiv c$
$\mathsf{rec}_{\mathsf{Bool}}(C, c, c', \mathsf{false}) \equiv c'$

- Interpretation in sets a two-element set

The empty type

Formation 0 is a type

Introduction

Elimination If $x : 0$ and C is a type, then $\mathrm{rec}_0(C, x) : C$

Computation

Exercise

Define a function of type $0 \to \mathrm{Bool}$.

- Interpretation in sets: the empty set

The type of natural numbers

Formation Nat is a type

Introduction $0 : \mathsf{Nat}$
If $n : \mathsf{Nat}$, then $S(n) : \mathsf{Nat}$

Elimination If C is a type and $c_0 : C$ and $c_s : C \to C$ and $x : \mathsf{Nat}$
then $\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, x) : C$

Computation $\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, 0) \equiv c_0$
$\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, S(n)) \equiv c_s @ (\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, n))$

- Interpretation in sets: the set of natural numbers

Exercise

Define a function $\mathsf{isZero?} : \mathsf{Nat} \to \mathsf{Bool}$

Pattern matching

Exercise

Define a function $\mathsf{isZero?} : \mathsf{Nat} \to \mathsf{Bool}$

Solution

$\mathsf{isZero?} :\equiv \lambda(x : \mathsf{Nat}).\mathsf{rec}_{\mathsf{Nat}}(\mathsf{Bool}, \mathsf{true}, \lambda(x : \mathsf{Bool}).\mathsf{false}, x)$

Pattern matching

Exercise

Define a function isZero? : $\mathsf{Nat} \to \mathsf{Bool}$

Solution

$\mathsf{isZero?} :\equiv \lambda(x : \mathsf{Nat}).\mathsf{rec}_{\mathsf{Nat}}(\mathsf{Bool}, \mathsf{true}, \lambda(x : \mathsf{Bool}).\mathsf{false}, x)$

- Programming in terms of the eliminators rec is cumbersome.
- Equivalently, we can specify functions by **pattern matching**: A function $A \to C$ is specified completely if it is specified on the **canoncial elements of** A.

$$\mathsf{isZero?} : \mathsf{Nat} \to \mathsf{Bool}$$
$$\mathsf{isZero?}(0) :\equiv \mathsf{true}$$
$$\mathsf{isZero?}(S(n)) :\equiv \mathsf{false}$$

- The "specifying equations" correspond to the computation rules.

Pattern matching for 0, 1

Exercise

Define $f : 0 \to A$.

Solution

Nothing to do.

Pattern matching for 0, 1

Exercise

Define $f : 0 \to A$.

Solution

Nothing to do.

Exercise

Define $f : 1 \to A$.

Solution

$f(\mathsf{t}) :\equiv \ ?? : A$

Pattern matching for Bool

Exercise

Define $f : \mathsf{Bool} \to A$.

Solution

$$f(\mathsf{true}) :\equiv \ ?? : A$$
$$f(\mathsf{false}) :\equiv \ ?? : A$$

The type of pairs $A \times B$

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

Introduction If $a : A$ and $b : B$, then $\mathsf{pair}(a, b) : A \times B$

Elimination If C is a type, and $p : A \to (B \to C)$ and $t : A \times B$, then $\mathsf{rec}_\times(A, B, C, p, t) : C$

Computation $\mathsf{rec}_\times(A, B, C, p, \mathsf{pair}(a, b)) \equiv p@a@b$

- Interpretation in sets: Cartesian product of sets A and B
- Notational convention: write (a, b) instead of $\mathsf{pair}(a, b)$

Exercises

Exercise

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

Exercise

Compute $\quad \mathsf{fst}(\mathsf{pair}(a, b)) \quad$ and $\quad \mathsf{snd}(\mathsf{pair}(a, b))$

Exercises

Exercise

Given types A and B, write a function swap of type $A \times B \to B \times A$.

Exercise

What is the type of swap@pair(t, false)? Compute the result.

Associativity of cartesian product

Exercise

Write a function assoc of type $(A \times B) \times C \to A \times (B \times C)$.

Type dependency

In particular: dependent type B over A

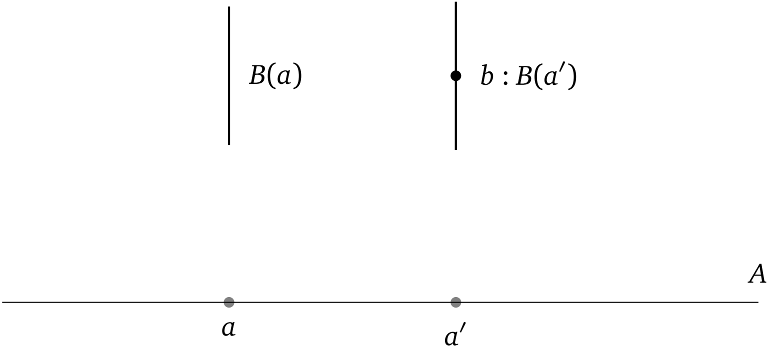
$$x : A \;\vdash\; B(x)$$

"family B of types indexed by A"

- A type can depend on several variables
- Example: type of vectors (with entries from, e.g., Nat) of length n

$$n : \mathsf{Nat} \;\vdash\; \mathsf{Vect}(n)$$

Dependent types in pictures



Universes

Universes

- There is also a type Type. Its elements are types, e.g. $A : \mathsf{Type}$.
- The dependent type $x : A \vdash B$ can be considered as a function

$$\lambda x.B : A \to \mathsf{Type}$$

What is the type of Type?

- Actually, hierarchy $(\mathsf{Type}_i)_{i \in I}$ to avoid paradoxes.
- But we ignore this for the most part, and only write Type.

$$(n : \mathsf{Nat}), (A : \mathsf{Type}) \vdash \mathsf{Vect}(A, n) : \mathsf{Type}$$

The type of dependent functions $\prod_{x:A} B$

Formation If $x : A \vdash B$, then $\prod_{x:A} B(x)$ is a type

Introduction If $(x : A) \vdash b : B$, then $\lambda(x : A).b : \prod_{x:A} B$

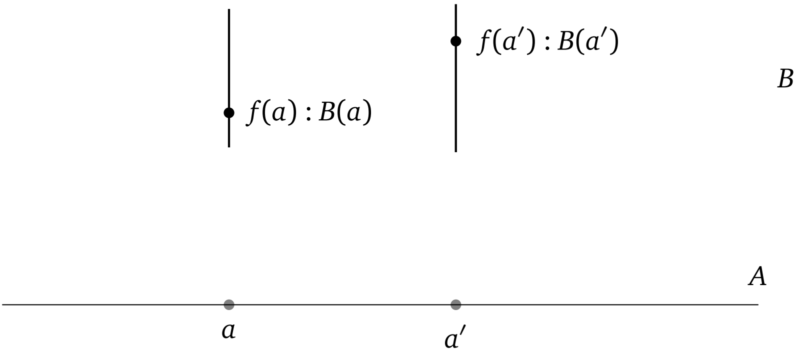
Elimination If $f : \prod_{x:A} B$ and $a : A$, then $f(a) : B[x := a]$

Computation $(\lambda(x : A).b)(a) \equiv b[x := a]$

- The case $A \to B$ is a special case, where B does not depend on $x : A$
- Interpretation in sets: The product $\prod_{x:A} B$

A dependent function in pictures

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



Pattern matching for 0, 1

Exercise

Specify a dependent function $f : \prod_{x:0} A(x)$.

Solution

Nothing to do.

Exercise

Specify a dependent function $f : \prod_{x:1} A(x)$.

Solution

$$f(\mathsf{t}) :\equiv\ ?? : A(\mathsf{t})$$

Pattern matching for Bool

Exercise

Specify a dependent function $f : \prod_{x:\mathsf{Bool}} A(x)$.

Solution

$$f(\mathsf{true}) :\equiv \ ?? : A(\mathsf{true})$$
$$f(\mathsf{false}) :\equiv \ ?? : A(\mathsf{false})$$

The type of dependent pairs $\sum_{x:A} B$

Formation If $x : A \vdash B$, then $\sum_{x:A} B(x)$ is a type

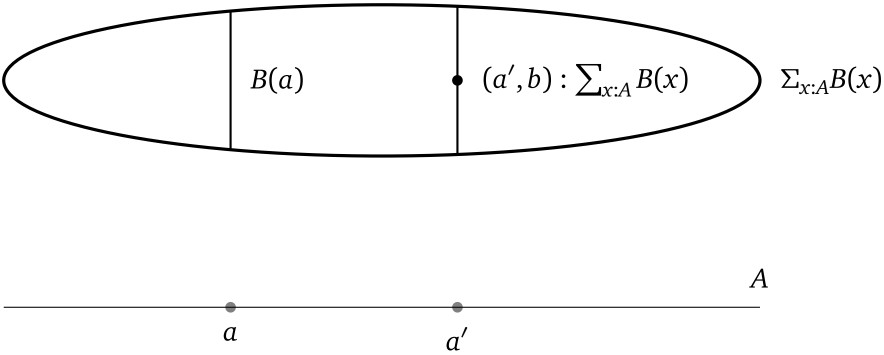
Introduction If $a : A$ and $b : B(a)$, then $\mathsf{pair}(a, b) : \sum_{x:A} B(x)$

Elimination ...

Computation ...

- The case $A \times B$ is a special case, where B does not depend on $x : A$
- Interpretation in sets: The disjoint union $\coprod_{x:A} B$

Σ-type in pictures



The identity type

Formation If $a : A$ and $b : A$, then $\mathsf{Id}_A(a, b)$ is a type

Introduction If $a : A$, then $\mathsf{refl}(a) : \mathsf{Id}_A(a, a)$

Elimination If

$(x,y : A), (p : \mathsf{Id}_A(x,y)) \ \vdash \ C(x,y,p)$

and

$(x : A) \ \vdash \ t(x) : C(x,x,\mathsf{refl}(x))$

then

$(x,y : A), (p : \mathsf{Id}_A(x,y) \ \vdash \ \mathsf{ind}_{\mathsf{Id}}(t; x,y,p) : C(x,y,p)$

Computation . . .

Interpretation in sets

Equality $a = b$

Exercise

Exercise

Write a term of type $\mathsf{Id}_A(\mathsf{snd}(\mathsf{t}, \mathsf{false}), \mathsf{false})$. (Hint: remember the important facts about $\equiv$.)

The elimination principle for Id_A

- By pattern matching, to specify a map on a family of identities $\mathsf{Id}_A(x,y)$, it suffices to specify its image on $\mathsf{refl}(x)$ for some x.
- For instance, to define

$$\mathsf{sym} : \prod_{x,y:A} \mathsf{Id}(x,y) \;\to\; \mathsf{Id}(y,x)$$

it suffices to specify its image on $(x,x,\mathsf{refl}(x))$

$$\mathsf{sym}(x,x,\mathsf{refl}(x)) \equiv$$

The elimination principle for Id_A

- By pattern matching, to specify a map on a family of identities $\mathsf{Id}_A(x,y)$, it suffices to specify its image on $\mathsf{refl}(x)$ for some x.
- For instance, to define

$$\mathsf{sym} : \prod_{x,y:A} \mathsf{Id}(x,y) \;\to\; \mathsf{Id}(y,x)$$

it suffices to specify its image on $(x, x, \mathsf{refl}(x))$

$$\mathsf{sym}(x, x, \mathsf{refl}(x)) \equiv \mathsf{refl}(x)$$

More about identities

Exercise

Exercise: Using pattern matching, construct a term trans of type

$$\prod_{x,y:A} \mathsf{Id}(x,y) \to \prod_{z:A} \mathsf{Id}(y,z) \to \mathsf{Id}(x,z)$$

Transport

Exercise

Given $x : A \ \vdash \ B$, define a function of type

$$\mathsf{transport}^B : \prod_{x,y:A} \mathsf{Id}(x,y) \to B(x) \to B(y)$$

Exercise: swap is involutive

Exercise

Given types A and B, write a function of type

$$\prod_{t:A\times B} \mathsf{Id}(\mathsf{swap}(\mathsf{swap}(t)), t)$$

The disjoint sum $A + B$

Formation If A and B are types, then $A + B$ is a type

Introduction If $a : A$, then $\mathsf{inl}(a) : A + B$
If $b : B$, then $\mathsf{inr}(b) : A + B$

Elimination If $f : A \to C$ and $g : B \to C$, then
$\mathsf{rec}_+(C, f, g) : A + B \to C$

Computation $\mathsf{rec}_+(C, f, g)(\mathsf{inl}(a)) \equiv f(a)$
$\mathsf{rec}_+(C, f, g)(\mathsf{inr}(b)) \equiv g(b)$

- Interpretation in sets: disjoint union
- What is the pattern matching principle for $A + B$?
- Can be seen as a special case of $\sum$

Interpreting types as sets

Syntax	Set interpretation
A	set A
$a : A$	$a \in A$
$A \times B$	cartesian product
$A \to B$	set of functions $A \to B$
$A + B$	disjoint union $A \amalg B$
$x : A \ \vdash \ B(x)$	family B of sets indexed by A
$\sum_{x:A} B(x)$	disjoint union $\coprod_{x:A} B(x)$
$\prod_{x:A} B(x)$	dependent function
$\mathsf{Id}_A(a, b)$	equality $a = b$

Outline

1. The syntax of type theory and an interpretation in sets

2. An interpretation of type theory in propositions

Interpreting types as propositions

Syntax	Logic
A	proposition A
$a : A$	a is a proof of A
1	$\top$
0	$\bot$
$A \times B$	$A \wedge B$
$A \to B$	$A \Rightarrow B$
$A + B$	$A \vee B$
$x : A \ \vdash \ B(x)$	predicate B on A
$\sum_{x:A} B(x)$	$\exists x \in A, B(x)$
$\prod_{x:A} B(x)$	$\forall x \in A, B(x)$
$\mathsf{Id}_A(a, b)$	equality $a = b$

- The connectives $\vee$ and $\exists$ thus obtained behave constructively.
- Known as the **Curry-Howard correspondence**.

Negation

Definition

$$\neg A \quad :\equiv \quad A \to 0$$

Exercise

1. Construct a term of type $A \to \neg\neg A$
2. Try to construct a term of type $\neg\neg A \to A$

Summary: Logic in type theory

Curry-Howard correspondence resp. Brouwer-Heyting-Kolmogorov interpretation:

- propositions are types
- proofs of P are terms of type P

Hence

- In principle, all types could be called propositions.
- To prove a proposition P means to construct a term of type P.
- In UF, only some types are called ‘propositions’ (and only some types are called ‘sets’), cf. later.

Convention

For type X, we also say “**Show** X” or “**Prove** X” for “**Construct a term of type** X”.

Exercise

Construct a term of type $\neg(\mathsf{Id}(\mathsf{true}, \mathsf{false}))$.

Hint: use $\mathsf{transport}^B$ with a suitable $B : \mathsf{Bool} \to \mathsf{Type}$

Solution

Set $B :\equiv \mathsf{rec}_{\mathsf{Bool}}(\mathsf{Type}, 1, 0) : \mathsf{Bool} \to \mathsf{Type}$. *Then* $B(\mathsf{true}) \equiv 1$ *and* $B(\mathsf{false}) \equiv 0$.

Hence

$$\lambda p : \mathsf{Id}(\mathsf{true}, \mathsf{false}).\mathsf{transport}^B(p, \mathsf{t}) \quad : \quad \mathsf{Id}(\mathsf{true}, \mathsf{false}) \to 0$$

Dependent elimination for 0, 1, Bool

- 0 If $x : 0 \;\vdash\; C(x)$ is a type family and $x : 0$, then $\mathsf{ind}_1(C, x) : C(x)$
- 1 If $x : 1 \;\vdash\; C(x)$ is a type family and $c_{\mathsf{t}} : C(\mathsf{t})$ and $x : 1$, then $\mathsf{ind}_1(C, c, x) : C(x)$
- Bool If $x : \mathsf{Bool} \;\vdash\; C(x)$ is a type family and $c_{\mathsf{true}} : C(\mathsf{true})$ and $c_{\mathsf{false}} : C(\mathit{false})$ and $x : \mathsf{Bool}$, then $\mathsf{ind}_{\mathsf{Bool}}(C, c, c', x) : C(x)$