School on Univalent Mathematics

(Cortona 2022)

I. Type theory

Gialuca Amato

slides mostly stolen from Benedikt Ahrens' ones
errors definitively added by me

Foundation of Mathematics

By the name *foundations of mathematics* we mean the study of *formal systems* that allows us to formalize much if not all of mathematics.

There are several approaches to the foundations of mathematics, which we may mostly divide in two big families:

- *set theories*;
- *type theories*.

Set theories

- everything is a set;
- *naive* set-theory is the de-facto standard for most mathematicians not interested in the foundations of mathematics;
- Example:
  a function from A to B is a subset of $A \times B$ such that ...

Type theories

- everything is a type or a term (program) of a given type;
- Example: a function from A to B is a type, denoted by $A \to B$;
- Example: the costant function which maps each element of A to the constant b of type B is the term $\lambda(x : A).b$ of type $A \to B$;
- all type theories contains λ-calculus at their core (a functional programming language) with the infrastructure for writing mathematical proofs;
- in some type theories, to each proposition P corresponds a type P, and proofs of P are terms of type P (*propositions as types*).

Martin-Löf type theory

In this course we will work in the type theory introduced by Per Martin-Löf. Its main characteristics:

- **propositions as types**;
- **dependent types and functions**: a type may depend on a element (term) of an other type:
  - 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 and computable;

In the following we use the term “type theory” to denote the Martin-Löf type theory.

Multiple interpretations of type theory

There are two basic interpretation of types and terms which help intuition.

Set based	a type A is a set; a term a of type A is an element of the set A.
Logic based	a type A is a proposition (or a predicate); a term a of type A is a proof of A.

More complex interpretations (such as types as **simplicial sets**) are at the basis of the Univalence Foundations of mathematics.

We will not discuss these interpretations in our lecture.

Outline

Outline

Our goal

Our main goal: to write well-typed terms

In type theory, both the activities of

- defining a mathematical object;
- proving a mathematical statement;

are done by writing well-typed terms.

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 $\mathsf{isZero?}$ of type $\mathsf{Nat} \to \mathsf{Bool}$. You start out with

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

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

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

Inference rules and derivations (1)

Inference rules allow to derive correct judgments from already proved judgments.

- An inference rule is an implication of judgments,

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

  e.g.,

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

$$\frac{\Gamma \ \vdash \ a \equiv b : A}{\Gamma \ \vdash \ b \equiv a : A}$$

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

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

Inference rules and derivations (2)

We will be more informal in this presentation:

- We sometimes omit the context when writing judgments.
- We will use english for writing inference rules.
  e.g., by writing

  "If $a \equiv b$, then $b \equiv a$"

  instead of

$$\frac{\Gamma \;\vdash\; a \equiv b : A}{\Gamma \;\vdash\; b \equiv a : A}$$

Important facts about judgments

- term a does not exist independently of its type A
- If $x : A$ and $A \equiv B$ then $x : B$;
- a well-formed term a has exactly one type up to $\equiv$

  (whereas an element a can be member of many different sets)
- $\equiv$ is a congruence, e.g., if $a \equiv a'$ and $f \equiv f'$, 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	ways to use a term of the new type to construct terms
Conversion	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

(sets: set of functions from A to B)

(logics: A implies B)

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

(b may conain some occurrences of x)

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

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

substitution $b[a/x]$ is built-in and not part of the language of terms

Conversion and computation

The judgment

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

(and others we will see later) may be given a computational meaning by orienting the equivalence from left to right:

$$(\lambda(x : A).b)(a) \Longrightarrow b[a/x]$$

Rewriting terms according to $\Longrightarrow$ gives us an algorithm that

- always terminates;
- transforms every term to a **normal form**;
- may be used to decide whether two terms are convertible.

The singleton type

Formation $\mathbf{1}$ is a type

(sets: a one-element set $\{\mathsf{t}\}$)

(logic: the true proposition $\top$)

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

(sets: the only element o $\mathbf{1}$)

(logic: the trivial proof that $\top$ is true)

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

($\mathsf{rec}_{\mathbf{1}}$ is called a **recursor**)

($\mathsf{rec}_{\mathbf{1}}$ is not very useful until we introuce dependent types)

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

Booleans

Exercise: Define the type of boolean values, with two elements.

Formation

Introduction

Elimination

Conversion

Booleans

Formation Bool is a type

(sets: a two element set {true, false})

Introduction $\mathsf{true} : \mathsf{Bool}$, $\mathsf{false} : \mathsf{Bool}$

Elimination If $x : \mathsf{Bool}$ and C is a type and $c, c' : C$,

then $\mathsf{rec}_{\mathsf{Bool}}(C, c, c', x) : C$

(interpretation: if x = true then c else c')

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

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

The empty type

Formation $\mathbf{0}$ is a type

(sets: the empty set)

(logic: the false proposition)

Introduction

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

(logic: from falsehood, anything)

Conversion

- Exercise: Define a function of type $\mathbf{0} \to \mathsf{Bool}$.

The type of natural numbers

Formation Nat is a type

(sets: the set of natural numbers)

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 $\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, x) : C$

$$\left(\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, x) = \begin{cases} c_0 & \text{if } x = 0; \\ c_s(\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, y)) & \text{if } x = S(y) \end{cases} \right)$$

Conversion $\mathsf{rec}_{\mathsf{Nat}}(C, c_0, c_s, 0) \equiv c_0$

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

Using the nat recursor

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

Using the nat recursor

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

Solution:

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

whose meaning is

$$\begin{aligned}\mathsf{isZero?} := \lambda(x : \mathsf{Nat}).\text{if } x = 0 \text{ then } \mathsf{true} \\ \text{else } (\lambda(x : \mathsf{Bool}).\mathsf{false})(\mathsf{isZero?}(x - 1))\end{aligned}$$

Pattern matching

- Programming in terms of the recursors 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 **canonical elements of** A.

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

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

Pattern matching for $\mathbf{0}$, $\mathbf{1}$, Bool

How to define a map

- $\mathbf{0} \to A$

  Nothing to do

- $\mathbf{1} \to A$

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

- $f : \mathsf{Bool} \to A$

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

The type of pairs $A \times B$

Formation	If A and B are types, then $A \times B$ is a type (sets: Cartesian product of sets A and B) (logic: $A \wedge B$)
Introduction	If $a : A$ and $b : B$, then $\langle a, b \rangle : A \times B$ (logic: given proofs a, b of A and B, we get a proof of $A \wedge B$)
Elimination	If C is a type, and $p : A \to (B \to C)$ and $t : A \times B$, then $\mathsf{rec}_\times(A, B, C, p, t) : C$
Conversion	$\mathsf{rec}_\times(A, B, C, p, \langle a, b \rangle) \equiv p(a)(b)$

Exercises

- Define $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$
  - using the eliminator
  - by pattern matching
- Compute $\mathsf{fst}(\langle a, b \rangle)$ and $\mathsf{snd}(\langle a, b \rangle)$

Exercises

- Define $\mathsf{fst} : A \times B \to A$ and $\mathsf{snd} : A \times B \to B$
  - using the eliminator

$$\mathsf{fst} := \lambda(t : A \times B).\mathsf{rec}_\times(A, B, A, \lambda(x : A).\lambda(y : B).x, t)$$

  - by pattern matching

$$\mathsf{fst}(\langle a, b\rangle) := a$$

- Compute $\mathsf{fst}(\langle a, b\rangle)$ and $\mathsf{snd}(\langle a, b\rangle)$

Exercises

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

- What is the type of $\mathsf{swap}(\langle \mathsf{t}, \mathsf{false} \rangle)$? Compute the result.

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

Solution

$$\mathsf{swap} := \lambda(x : A \times B).\,(\mathsf{snd}(x), \mathsf{fst}(x))$$

- What is the type of $\mathsf{swap}(\langle \mathsf{t}, \mathsf{false} \rangle)$? Compute the result.

Solution

$\mathsf{swap}((\mathsf{t}, \mathsf{false})) : \mathsf{Bool} \times \boldsymbol{1}$

$$\begin{aligned} \mathsf{swap}(\langle \mathsf{t}, \mathsf{false} \rangle) &\equiv \Big\langle \mathsf{snd}(\langle \mathsf{t}, \mathsf{false} \rangle), \mathsf{fst}(\langle \mathsf{t}, \mathsf{false} \rangle) \Big\rangle \\ &\equiv \langle \mathsf{false}, \mathsf{t} \rangle \end{aligned}$$

Associativity of cartesian product

Exercise

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

Associativity of cartesian product

Exercise

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

Solution

$$\mathsf{assoc} := \lambda(x : (A \times B) \times C).\big\langle \mathsf{fst}(\mathsf{fst}(x)), \langle \mathsf{snd}(\mathsf{fst}(x)), \mathsf{snd}(x)\rangle \big\rangle$$

or

$$\mathsf{assoc}((x,y),z) := (x,(y,z))$$

Outline

Type dependency

In particular: dependent type B over A

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

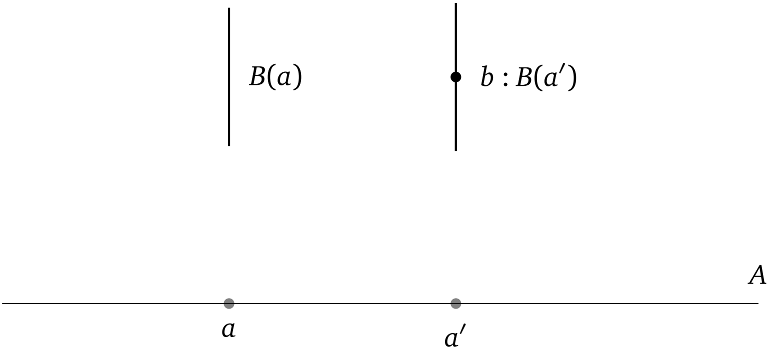
"family B of types indexed by A"

- Example: type of vectors (with entries from, e.g., Bool) of length n

$$n : \mathsf{Nat} \;\vdash\; \mathsf{Vect}(n) \quad (= \mathsf{Bool}^n)$$

- A type can depend on several variables

Dependent types in pictures



Universes

- There is also a type Type. Its elements are types, $A : \mathsf{Type}$;
- The judgment $x : A \ \vdash \ B$ may be viewed as $x : A \ \vdash \ B : \mathsf{Type}$;
- $(n : \mathsf{Nat}), (A : \mathsf{Type}) \ \vdash \ \mathsf{Vect}(A, n) : \mathsf{Type}$.

What is the type of Type?

- Actually, hierarchy $(\mathsf{Type}_i)_{i \in I}$ to avoid paradoxes.

$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \cdots$$

- But we ignore this for the most part, and only write Type.

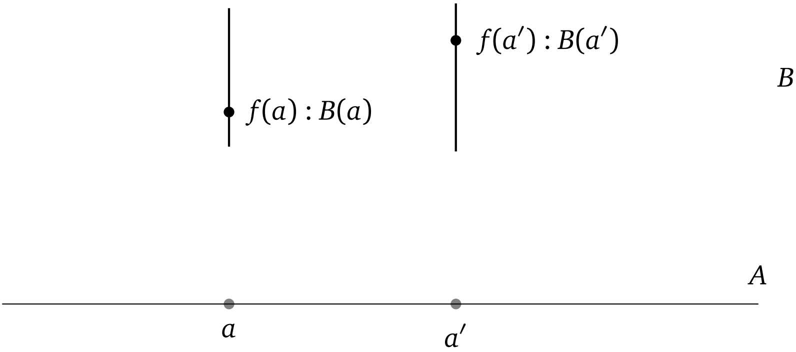
The type of dependent functions $\prod_{x:A} B$

Formation	If $x : A \;\vdash\; B(x)$, then $\prod_{x:A} B(x)$ is a type.
	(sets: mapping each $x \in A$ to the set $B(x)$)
	(logic: $\forall x : A, B(x)$)
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 $f(a) : B[x/a]$
Conversion	$(\lambda(x : A).b)(a) \equiv b[x/a]$

The case $A \to B$ is a special case, where B does not depend on $x : A$

A dependent function in pictures

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



Pattern matching for $\mathbf{0}$, $\mathbf{1}$, Bool

To specify a dependent function

- $f : \prod_{x:\mathbf{0}} A(x)$

Nothing to do

- $f : \prod_{x:\mathbf{1}} A(x)$

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

- $f : \prod_{x:\mathsf{Bool}} A(x)$

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

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

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

(sets: disjoint union $\amalg_{x:a} B(x)$)

(logic: $\exists x : A, B(x)$)

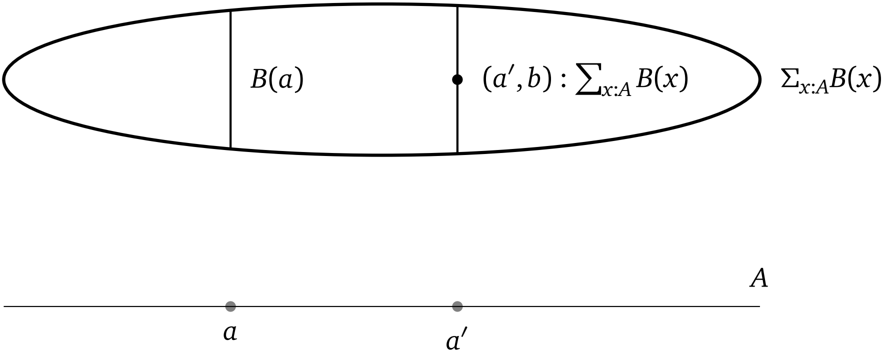
Introduction If $a : A$ and $b : B[x/a]$, then $\langle a, b \rangle : \sum_{x:A} B$

Elimination ...

Conversion ...

The case $A \times B$ is a special case, where B does not depend on $x : A$

Σ-type in pictures



The identity type

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

(logic: the equality predicate $a = b$)

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

(the trivial proof that a is equal to itself)

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

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

Conversion . . .

Exercise

- Write a term of type $\mathsf{Id}_A(\mathsf{snd}(\langle \mathsf{t}, \mathsf{false} \rangle), \mathsf{false})$.

  (Hint: remember the important facts about $\equiv$)

Exercise

- Write a term of type $\mathsf{Id}_A(\mathsf{snd}(\langle \mathsf{t}, \mathsf{false} \rangle), \mathsf{false})$.

  (Hint: remember the important facts about $\equiv$)

Solution

We have

$$\mathsf{snd}(\langle \mathsf{t}, \mathsf{false} \rangle) \equiv \mathsf{false}$$

and hence

$$\mathsf{Id}_A(\mathsf{snd}(\mathsf{t}, \mathsf{false}), \mathsf{false}) \equiv \mathsf{Id}_A(\mathsf{false}, \mathsf{false})$$

Since

$$\mathsf{refl}(\mathsf{false}) : \mathsf{Id}_A(\mathsf{false}, \mathsf{false})$$

we also have

$$\mathsf{refl}(\mathsf{false}) : \mathsf{Id}_A(\mathsf{snd}(\mathsf{t}, \mathsf{false}), \mathsf{false})$$

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

$$\mathsf{sym}(x,x,\mathsf{refl}(x)) :=$$

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

$$\mathsf{sym}(x, x, \mathsf{refl}(x)) := \mathsf{refl}(x)$$

More about identities

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

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

More about identities

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

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

$$\mathsf{trans}(x,x,\mathsf{refl}(x),z,p) :\equiv p$$

Transport

Exercise

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

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

Transport

Exercise

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

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

Solution

$$\mathsf{transport}^B(x,x,\mathsf{refl}(x),b) := b$$

Exercise: swap is involutive

Exercise

Given types A and B, write a function of type

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

Exercise: swap is involutive

Exercise

Given types A and B, write a function of type

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

Solution

$$f(\langle a, b\rangle) :\equiv \mathsf{refl}(\langle a, b\rangle)$$

Why is f a solution?

The disjoint sum $A + B$

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

(sets: disjoint union)

(logic: constructive disjunction $A \vee B$)

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

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

- Exercise: write down the dependent eliminator for $A + B$
- What is the pattern matching principle for $A + B$?

Outline

1. Non-dependent types
2. Dependent types
3. More on propositions as types
4. Problem session

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
$\mathsf{Id}_A(a, b)$	???

Interpreting types as propositions

Syntax	Logic
A	proposition A
$a : A$	a is a proof of A
$\mathbf{1}$	$\top$
$\mathbf{0}$	$\bot$
$A \times B$	$A \wedge B$
$A \to B$	$A \Rightarrow B$
$A + B$	$A \vee B$
$x : A \;\vdash\; B(x)$	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.

Negation

Definition

$$\neg A := A \to \mathbf{o}$$

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', cf later.

Convention

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

true is not false

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 $B := \mathsf{rec}_{\mathsf{Bool}}(\mathsf{Type}, \mathbf{1}, \mathbf{0}) : \mathsf{Bool} \to \mathsf{Type}$.

Then $B(\mathsf{true}) \equiv \mathbf{1}$ *and* $B(\mathsf{false}) \equiv \mathbf{0}$. *Hence*

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

Exercise: Dependent elimination rules

Write down the dependent elimination rule for

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

Outline

Problems

- Solve the exercises from the lecture.
- Define addition $+$ of natural numbers in terms of the eliminator, and via pattern matching.
- Give a proof of $\mathsf{Id}(2+2,4)$. Explain how/why your proof works.
- Given types A, B, and C, define maps between $A \times (B + C)$ and $A \times B + A \times C$. Show that they are pointwise inverses.
- For A, B, and $P : A \to \mathsf{Type}$, give maps between $\sum_{x:A} B \times P(x)$ and $B \times \sum_{x:A} P(x)$. Show that they are pointwise inverses.
- Prove that, for any $x : \mathbf{1}$, $\mathsf{Id}(x, \mathsf{t})$.