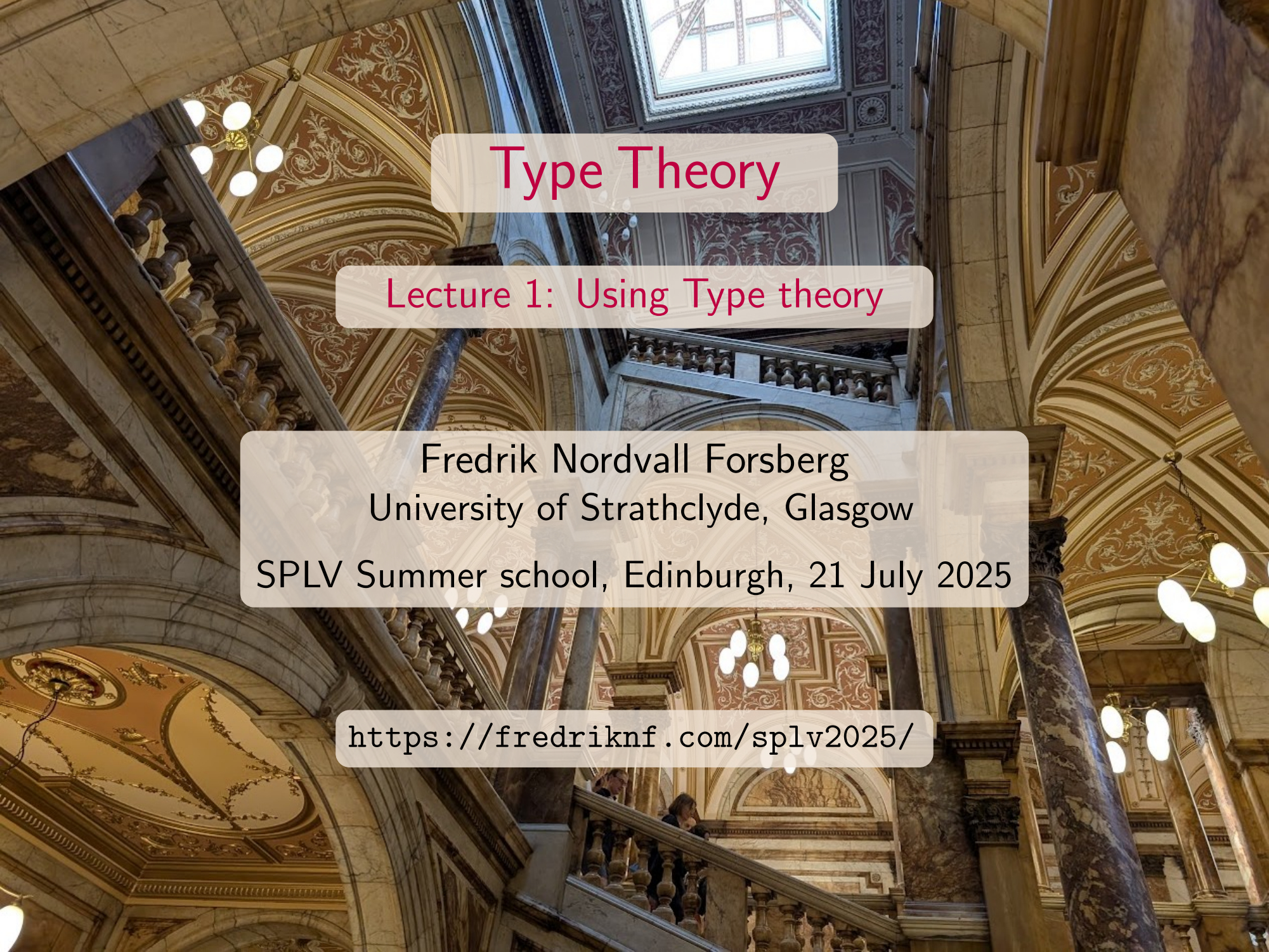


Type Theory

Lecture 1: Using Type theory

Fredrik Nordvall Forsberg
University of Strathclyde, Glasgow

SPLV Summer school, Edinburgh, 21 July 2025

`https://fredriknf.com/splv2025/`

Course plan

- **Today:** Using type theory.
  - Rules
  - Logic and mathematics in type theory
- **Tomorrow:** Semantics of type theory.
- **Thursday:** Implementation and metatheory.

Main source material

- **Principles of Dependent Type Theory**, Carlo Angiuli and Daniel Gratzer. Draft book, 2025.
- **Intuitionistic Type Theory**, Per Martin-Löf, "Bibliopolis Book", 1984.
- **Syntax and semantics of dependent types**, Martin Hofmann, 1997.
- **Slides and exercises:** `https://fredriknf.com/splv2025/`

Why learn type theory?

A few reasons:

Why learn type theory?

A few reasons:

1. Type theories are beautiful mathematical structures with deep connections to logic, category theory, abstract homotopy theory.

Why learn type theory?

A few reasons:

1. Type theories are beautiful mathematical structures with deep connections to logic, category theory, abstract homotopy theory.

2. Type theories are the basis of proof assistants, and you just want to get Reviewer 2 to shut up by formalising all your proofs.

Why learn type theory?

A few reasons:

1. Type theories are beautiful mathematical structures with deep connections to logic, category theory, abstract homotopy theory.

2. Type theories are the basis of proof assistants, and you just want to get Reviewer 2 to shut up by formalising all your proofs.

3. Active research area, with lots of interesting problems to work on.

Why learn type theory?

A few reasons:

1. Type theories are beautiful mathematical structures with deep connections to logic, category theory, abstract homotopy theory.

2. Type theories are the basis of proof assistants, and you just want to get Reviewer 2 to shut up by formalising all your proofs.

3. Active research area, with lots of interesting problems to work on.

4. It might give you a new perspective on other "more mainstream" programming languages you might be working in/on.

Brief and incomplete history leading to type theory

Brief and incomplete history leading to type theory

- BHK interpretation: informal explanation of what a constructive proof is (Heyting 1934, Kolmogorov 1932)

L.E.J. Brouwer

Arend Heyting

Andrey Kolmogorov

Brief and incomplete history leading to type theory

- BHK interpretation: informal explanation of what a constructive proof is (Heyting 1934, Kolmogorov 1932)
- Curry: the simply typed lambda calculus/typed combinatory logic corresponds precisely to implicational fragment of constructive propositional logic (Curry 1958)

L.E.J. Brouwer

Arend Heyting

Andrey Kolmogorov

Haskell Curry

Brief and incomplete history leading to type theory

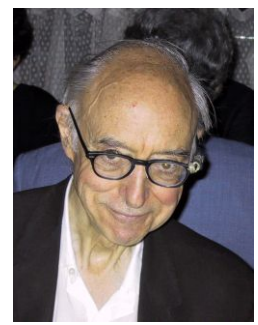
- BHK interpretation: informal explanation of what a constructive proof is (Heyting 1934, Kolmogorov 1932)
- Curry: the simply typed lambda calculus/typed combinatory logic corresponds precisely to implicational fragment of constructive propositional logic (Curry 1958)
- Howard: Curry's correspondance extends also to $\wedge$, $\vee$, $\forall$ and $\exists$ — latter two correspond to dependent types (Howard 1969)

L.E.J. Brouwer

Arend Heyting

Andrey Kolmogorov

Haskell Curry



William Howard

Brief and incomplete history leading to type theory

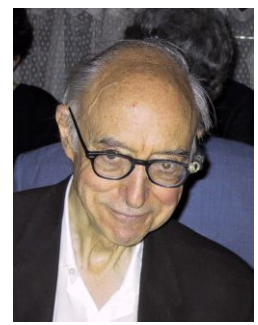
- BHK interpretation: informal explanation of what a constructive proof is (Heyting 1934, Kolmogorov 1932)
- Curry: the simply typed lambda calculus/typed combinatory logic corresponds precisely to implicational fragment of constructive propositional logic (Curry 1958)
- Howard: Curry's correspondance extends also to $\wedge$, $\vee$, $\forall$ and $\exists$ — latter two correspond to dependent types (Howard 1969)
- Constructive Type Theory extends correspondence to predicate logic by introducing the identity type (Martin-Löf 1972)

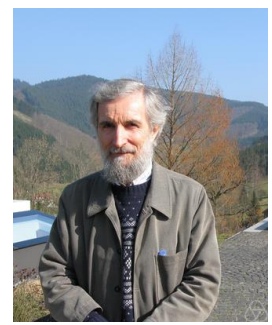

L.E.J. Brouwer


Arend Heyting


Andrey Kolmogorov


Haskell Curry


William Howard


Per Martin-Löf

Informal vs formal

Constructive type theory is meant to be a foundational system for constructive mathematics.

Informal vs formal

Constructive type theory is meant to be a foundational system for constructive mathematics.

As such, it is a formal system presented by rules.

When working in type theory, arguments can be presented informally (cf. "working in ZFC").

Informal vs formal

Constructive type theory is meant to be a foundational system for constructive mathematics.

As such, it is a formal system presented by rules.

When working in type theory, arguments can be presented informally (cf. "working in ZFC").

But when working on type theory, the rules of course need to be precise.

Judgements

Fundamental underlying concept: judgements.

$$\text{“context”} \vdash \text{“statement”}$$

Judgements

Fundamental underlying concept: judgements.

$$\text{“context”} \vdash \text{“statement”}$$

$$x : \mathbb{N} \vdash x + 5 : \mathbb{N}$$

Judgements

Fundamental underlying concept: judgements.

$$\text{“context”} \vdash \text{“statement”}$$

$$x : \mathbb{N} \vdash x + 5 : \mathbb{N}$$

For simple (non-dependent) types, the judgements are also simple:

Γ valid	Γ is a well formed context
A type	A is a well formed type
$\Gamma \vdash a : A$	a is of type A
$\Gamma \vdash a \equiv a' : A$	a and a' are the same term (in type A)

Simple types

Simple types are built up inductively:

$$\frac{}{\mathbb{N}\ \text{type}} \qquad \frac{}{\text{Bool type}} \qquad \frac{A\ \text{type} \qquad B\ \text{type}}{A \to B\ \text{type}} \qquad \frac{A\ \text{type} \qquad B\ \text{type}}{A \times B\ \text{type}}$$

Simple types

Simple types are built up inductively:

$$\frac{}{\mathbb{N}\ \text{type}} \qquad \frac{}{\text{Bool type}} \qquad \frac{A\ \text{type} \qquad B\ \text{type}}{A \to B\ \text{type}} \qquad \frac{A\ \text{type} \qquad B\ \text{type}}{A \times B\ \text{type}}$$

Contexts are lists of variables and their types:

$$\frac{}{\cdot\ \text{valid}} \qquad \frac{\Gamma\ \text{valid} \qquad A\ \text{type}}{\Gamma, x : A\ \text{valid}}$$

Simple types

Simple types are built up inductively:

$$\frac{}{\mathbb{N}\ \text{type}} \qquad \frac{}{\text{Bool type}} \qquad \frac{A\ \text{type} \qquad B\ \text{type}}{A \to B\ \text{type}} \qquad \frac{A\ \text{type} \qquad B\ \text{type}}{A \times B\ \text{type}}$$

Contexts are lists of variables and their types:

$$\frac{}{\cdot\ \text{valid}} \qquad \frac{\Gamma\ \text{valid} \qquad A\ \text{type}}{\Gamma, x : A\ \text{valid}}$$

And terms then make sense in context, e.g.

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : A \to B} \qquad \cdots$$

The need for dependent types

If we want more expressiveness, we need so-called dependent types.

For example, with simple types we can form the types

$$\mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}$$

but in general, we can only form such Bool^k for fixed, external k.

The need for dependent types

If we want more expressiveness, we need so-called dependent types.

For example, with simple types we can form the types

$$\mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}$$

but in general, we can only form such Bool^k for fixed, external k.

This fact is responsible for a lot of complexity and boilerplate in languages such as Haskell and C++.

The need for dependent types

If we want more expressiveness, we need so-called dependent types.

For example, with simple types we can form the types

$$\mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}$$

but in general, we can only form such Bool^k for fixed, external k.

This fact is responsible for a lot of complexity and boilerplate in languages such as Haskell and C++.

In dependent type theory, we can have types depending on terms instead:

$$n : \mathbb{N} \vdash \mathsf{Bool}^n \text{ type}$$

The need for dependent types

If we want more expressiveness, we need so-called dependent types.

For example, with simple types we can form the types

$$\mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \quad \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}$$

but in general, we can only form such Bool^k for fixed, external k.

This fact is responsible for a lot of complexity and boilerplate in languages such as Haskell and C++.

In dependent type theory, we can have types depending on terms instead:

$$n : \mathbb{N}, m : \mathbb{N} \vdash \mathsf{Bool}^{n+m} \text{ type}$$

Judgements for dependent types

The price to pay is that our judgements are more complicated, and more intertwined.

Γ valid	Γ is a well formed context
$\Gamma \vdash A$ type	A is a well formed type (in context Γ)
$\Gamma \vdash a : A$	a is of type A
$\Gamma \vdash A \equiv B$	A and B are the same type
$\Gamma \vdash a \equiv a' : A$	a and a' are the same term (in type A)

Judgements for dependent types

The price to pay is that our judgements are more complicated, and more intertwined.

Γ valid	Γ is a well formed context
$\Gamma \vdash A$ type	A is a well formed type (in context Γ)
$\Gamma \vdash a : A$	a is of type A
$\Gamma \vdash A \equiv B$	A and B are the same type
$\Gamma \vdash a \equiv a' : A$	a and a' are the same term (in type A)

Convention: We normally suppress mentioning Γ, and only show context extensions $x : A \vdash \ldots$.

Inference rules

Formally type theory is given by a collection of inference rules

$$\frac{\mathcal{J}_1 \quad \dots \quad \mathcal{J}_n}{\mathcal{J}}$$

Inference rules

Formally type theory is given by a collection of inference rules

$$\frac{\mathcal{J}_1 \quad \dots \quad \mathcal{J}_n}{\mathcal{J}}$$

A judgement $\mathcal{J}$ is derivable if we can construct a derivation tree with conclusion $\mathcal{J}$ using the inference rules. For example:

$$\dfrac{\dfrac{}{n : \mathbb{N}, m : \mathbb{N} \vdash \mathsf{Bool}\ \mathsf{type}} \quad \dfrac{\dfrac{}{n : \mathbb{N}, m : \mathbb{N} \vdash n : \mathbb{N}} \quad \dfrac{}{n : \mathbb{N}, m : \mathbb{N} \vdash m : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash n + m : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash \mathsf{Bool}^{n+m}\ \mathsf{type}}$$

Inference rules

Formally type theory is given by a collection of inference rules

$$\frac{\mathcal{J}_1 \quad \dots \quad \mathcal{J}_n}{\mathcal{J}}$$

A judgement $\mathcal{J}$ is derivable if we can construct a derivation tree with conclusion $\mathcal{J}$ using the inference rules. For example:

$$\dfrac{\dfrac{}{n : \mathbb{N}, m : \mathbb{N} \vdash \mathsf{Bool}\ \mathsf{type}} \quad \dfrac{\dfrac{}{n : \mathbb{N}, m : \mathbb{N} \vdash n : \mathbb{N}} \quad \dfrac{}{n : \mathbb{N}, m : \mathbb{N} \vdash m : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash n + m : \mathbb{N}}}{n : \mathbb{N}, m : \mathbb{N} \vdash \mathsf{Bool}^{n+m}\ \mathsf{type}}$$

Of course, when working in type theory, we never explicitly construct derivation trees!

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory).

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even.

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even. ✓ Yes, would expect "$y : \mathbb{N} \vdash \mathrm{Even}(y)$ type".

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even. ✓ Yes, would expect "$y : \mathbb{N} \vdash \mathrm{Even}(y)$ type".
- `"hello"` is even.

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even. ✓ Yes, would expect "$y : \mathbb{N} \vdash \mathrm{Even}(y)$ type".
- `"hello"` is even. ✗ No, statement does not type-check, since `"hello"` is not a natural number.

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even. ✓ Yes, would expect "$y : \mathbb{N} \vdash \mathrm{Even}(y)$ type".
- `"hello"` is even. ✗ No, statement does not type-check, since `"hello"` is not a natural number.
- $\pi \in \cos$ (in set theory).

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even. ✓ Yes, would expect "$y : \mathbb{N} \vdash \mathrm{Even}(y)$ type".
- `"hello"` is even. ✗ No, statement does not type-check, since `"hello"` is not a natural number.
- $\pi \in \cos$ (in set theory). ✓ Yes.

Caveats

Judgements are "external" — they cannot be proven inside the language.

In contrast, "$a \in A$" in set theory is an "internal" statement.

Quiz: what makes sense to prove or disprove?

- 5 is a natural number (in type theory). ✗ No, "$5 : \mathbb{N}$" is a judgement, not a statement.
- 5 is even. ✓ Yes, would expect "$y : \mathbb{N} \vdash \mathrm{Even}(y)$ type".
- `"hello"` is even. ✗ No, statement does not type-check, since `"hello"` is not a natural number.
- $\pi \in \cos$ (in set theory). ✓ Yes.

Note that $A \equiv B$ and $a \equiv a' : A$ also are "external" statements; we will see an internal version that can be (dis)proven later.

Basic rules

Variables:

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}$$

Conversion:

$$\frac{t : A \qquad A \equiv B}{t : B}$$

Judgemental equality:

$$\frac{t : A}{t \equiv t : A} \qquad \frac{t \equiv s : A}{s \equiv t : A} \qquad \frac{s \equiv t : A \qquad t \equiv u : A}{s \equiv u : A}$$

Congruence rules: for example

$$\frac{A \equiv A' \qquad B \equiv B'}{(A \to B) \equiv (A' \to B')}$$

(many more!)

A pattern for introducing types

A type is usually given by four (five) groups of rules:

Formation What is needed to construct the type?

Introduction What is needed to construct canonical elements of the type?

Elimination How can elements of the type be used?

Computation What happens when you eliminate canonical elements? ("β-rules")

Uniqueness (sometimes) How are functions into or out of the type determined? ("η-rules")

Pair types

Formation

$$\frac{A \text{ type} \qquad B \text{ type}}{A \times B \text{ type}}$$

Introduction

$$\frac{a : A \qquad b : B}{(a, b) : A \times B}$$

Elimination

$$\frac{p : A \times B}{\mathsf{fst}\, p : A} \qquad \frac{p : A \times B}{\mathsf{snd}\, p : B}$$

Computation $\mathsf{fst}\,(a, b) \equiv a : A$ and $\mathsf{snd}\,(a, b) \equiv b : B$.

Uniqueness

$$\frac{p : A \times B}{p \equiv (\mathsf{fst}(p), \mathsf{snd}(p)) : A \times B}$$

Pair types: alternative elimination and computation rules

Elimination

$$\frac{p : A \times B}{\mathsf{fst}\, p : A} \qquad \frac{p : A \times B}{\mathsf{snd}\, p : B}$$

Computation

$\mathsf{fst}\,(a, b) \equiv a : A$ and $\mathsf{snd}\,(a, b) \equiv b : B$.

Uniqueness

$$\frac{p : A \times B}{p \equiv (\mathsf{fst}(p), \mathsf{snd}(p)) : A \times B}$$

Pair types: alternative elimination and computation rules

Elimination

$$\frac{p : A \times B}{\mathsf{fst}\, p : A} \qquad \frac{p : A \times B}{\mathsf{snd}\, p : B}$$

Computation

$\mathsf{fst}\,(a, b) \equiv a : A$ and $\mathsf{snd}\,(a, b) \equiv b : B$.

Uniqueness

$$\frac{p : A \times B}{p \equiv (\mathsf{fst}(p), \mathsf{snd}(p)) : A \times B}$$

Alternatively:

Elimination'

$$\frac{z : A \times B \vdash C \text{ type} \quad x : A, y : B \vdash c : C[z \mapsto (x, y)] \quad p : A \times B}{\mathsf{elim}_\times(C, c, p) : C[z \mapsto p]}$$

Computation'

$$\mathsf{elim}_\times(C, c, (a, b)) \equiv c[x \mapsto a, y \mapsto b] : C[z \mapsto (a, b)].$$

Pair types: alternative elimination and computation rules

Elimination

$$\frac{p : A \times B}{\mathsf{fst}\, p : A} \qquad \frac{p : A \times B}{\mathsf{snd}\, p : B}$$

Computation $\mathsf{fst}\,(a, b) \equiv a : A$ and $\mathsf{snd}\,(a, b) \equiv b : B$.

Uniqueness

$$\frac{p : A \times B}{p \equiv (\mathsf{fst}(p), \mathsf{snd}(p)) : A \times B}$$

Alternatively:

Elimination'

$$\frac{z : A \times B \vdash C \text{ type} \quad x : A, y : B \vdash c : C[z \mapsto (x, y)] \quad p : A \times B}{\mathsf{elim}_\times(C, c, p) : C[z \mapsto p]}$$

Computation'

$$\mathsf{elim}_\times(C, c, (a, b)) \equiv c[x \mapsto a, y \mapsto b] : C[z \mapsto (a, b)].$$

Exercise

Show that E + C follows from E' + C', and that E' + C' follows from E + C + Uniqueness. Does Uniqueness follow from E' + C'?

Function types

Formation

$$\frac{A \text{ type} \qquad B \text{ type}}{A \to B \text{ type}}$$

Introduction

$$\frac{x : A \vdash t : B}{\lambda(x : A).t : A \to B}$$

Elimination

$$\frac{f : A \to B \qquad a : A}{f\ a : B}$$

Computation $(\lambda(x : A).t)\, a \equiv t[x \mapsto a] : B$

Uniqueness

$$\frac{f : A \to B}{f \equiv (\lambda(x : A).f\ x) : A \to B}$$

Example: swap function

Given types A and B, let us write $\mathsf{swap} : A \times B \to B \times A$.

$$\mathsf{swap} : A \times B \to B \times A$$

$$\mathsf{swap} = ?_0 : A \times B \to B \times A$$

Example: swap function

Given types A and B, let us write $\mathsf{swap} : A \times B \to B \times A$.

$$\mathsf{swap} : A \times B \to B \times A$$

$$\mathsf{swap} = \lambda(x : A \times B).\ ?_1 : B \times A$$

Example: swap function

Given types A and B, let us write $\mathsf{swap} : A \times B \to B \times A$.

$$\mathsf{swap} : A \times B \to B \times A$$

$$\mathsf{swap} = \lambda(x : A \times B).(\ ?_2 : B\ ,\ ?_3 : A\ )$$

Example: swap function

Given types A and B, let us write $\mathsf{swap} : A \times B \to B \times A$.

$$\mathsf{swap} : A \times B \to B \times A$$

$$\mathsf{swap} = \lambda(x : A \times B).(\mathsf{snd}\, x, \; ?_3 : A \;)$$

Example: swap function

Given types A and B, let us write $\mathsf{swap} : A \times B \to B \times A$.

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

Example: swap function

Given types A and B, let us write $\mathsf{swap} : A \times B \to B \times A$.

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

$$\dfrac{\dfrac{\dfrac{}{x : A \times B \vdash x : A \times B}}{x : A \times B \vdash \mathsf{snd}\, x : B} \qquad \dfrac{\dfrac{}{x : A \times B \vdash x : A \times B}}{x : A \times B \vdash \mathsf{fst}\, x : A}}{\dfrac{x : A \times B \vdash (\mathsf{snd}\, x, \mathsf{fst}\, x) : B \times A}{\vdash \lambda(x : A \times B).(\mathsf{snd}\, x, \mathsf{fst}\, x) : A \times B \to B \times A}}$$

The empty type

Formation

$$\frac{}{\mathbf{0}\ \mathsf{type}}$$

Introduction (none)

Elimination

$$\frac{C\ \mathsf{type} \qquad p : \mathbf{0}}{\mathsf{elim}_{\mathbf{0}}(C, p) : C}$$

Computation (none)

Exercise

Prove a dependent elimination rule from the non-dependent one:

$$\frac{z : \mathbf{0} \vdash C\ \mathsf{type} \qquad p : \mathbf{0}}{\mathsf{elim}_{\mathbf{0}}(C, p) : C[z \mapsto p]}$$

The unit type

Formation

$$\frac{}{\mathbf{1}\ \mathsf{type}}$$

Introduction $\star : \mathbf{1}$.

Elimination (none)

Computation (none)

Uniqueness

$$\frac{u : \mathbf{1}}{u \equiv \star : \mathbf{1}}$$

Exercise

Formulate and prove elimination and computation rules.

Disjoint union type

Formation

$$\frac{A \text{ type} \quad B \text{ type}}{A + B \text{ type}}$$

Introduction

$$\frac{a : A}{\mathsf{inl}\, a : A + B} \qquad \frac{b : B}{\mathsf{inr}\, b : A + B}$$

Elimination

$$\frac{\begin{array}{c} \\ z : A + B \vdash C \text{ type} \end{array} \quad \begin{array}{c} x : A \vdash c : C[z \mapsto \mathsf{inl}\, x] \\ y : B \vdash d : C[z \mapsto \mathsf{inr}\, y] \end{array} \quad \begin{array}{c} \\ s : A + B \end{array}}{\mathsf{elim}_+(C, c, d, s) : C[z \mapsto s]}$$

Computation

$$\mathsf{elim}_+(C, c, d, \mathsf{inl}\, a) \equiv c[x \mapsto a] : C[z \mapsto \mathsf{inl}\, a]$$
$$\mathsf{elim}_+(C, c, d, \mathsf{inr}\, b) \equiv d[y \mapsto b] : C[z \mapsto \mathsf{inr}\, b]$$

Exercise

Define Bool $= \mathbf{1} + \mathbf{1}$, and formulate and prove its rules.

Dependent function types

Formation

$$\frac{A \text{ type} \qquad x : A \vdash B \text{ type}}{(\Pi x : A)B \text{ type}}$$

Introduction

$$\frac{x : A \vdash t : B}{\lambda(x : A).t : (\Pi x : A)B}$$

Elimination

$$\frac{f : (\Pi x : A)B \qquad a : A}{f\ a : B[x \mapsto a]}$$

Computation $(\lambda(x : A).t)\, a \equiv t[x \mapsto a] : B[x \mapsto a]$

Uniqueness

$$\frac{f : (\Pi x : A)B}{f \equiv (\lambda(x : A).f\ x) : (\Pi x : A)B}$$

$A \to B$ is the special case when B does not depend on $x : A$.

Dependent pair types

Formation

$$\frac{A \text{ type} \qquad x : A \vdash B \text{ type}}{(\Sigma x : A)B \text{ type}}$$

Introduction

$$\frac{a : A \qquad b : B[x \mapsto a]}{(a, b) : (\Sigma x : A)B}$$

Elimination

$$\frac{p : (\Sigma x : A)B}{\mathsf{fst}\, p : A} \qquad \frac{p : (\Sigma x : A)B}{\mathsf{snd}\, p : B[x \mapsto \mathsf{fst}\, p]}$$

Computation $\mathsf{fst}\,(a, b) \equiv a : A$ and $\mathsf{snd}\,(a, b) \equiv b : B[x \mapsto a]$.

Uniqueness

$$\frac{p : (\Sigma x : A)B}{p \equiv (\mathsf{fst}(p), \mathsf{snd}(p)) : (\Sigma x : A)B}$$

$A \times B$ is the special case when B does not depend on $x : A$.

Example: the type-theoretic "Theorem of Choice"

Assume A type, B type and $x : A, y : B \vdash R$ type.

$$\mathsf{ac} : \Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

$$\mathsf{ac}\,g = ?_0 : (\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)$$

Example: the type-theoretic "Theorem of Choice"

Assume A type, B type and $x : A, y : B \vdash R$ type.

$$\mathsf{ac} : \Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

$$\mathsf{ac}\,g = (\ ?_1 : A \to B\ ,\ ?_2 : (\Pi x : A)R[x, ?_1\,x]\ )$$

Example: the type-theoretic "Theorem of Choice"

Assume A type, B type and $x : A, y : B \vdash R$ type.

$$\mathsf{ac} : \Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

$$\mathsf{ac}\,g = (\lambda(x : A).\ ?_3 : B\ ,\ ?_2 : (\Pi x : A)R[x, ?_1\,x]\ )$$

Example: the type-theoretic "Theorem of Choice"

Assume A type, B type and $x : A, y : B \vdash R$ type.

$$\mathsf{ac} : \Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

$$\mathsf{ac}\,g = (\lambda(x : A).\mathsf{fst}\,(g\,x),\ ?_2 : (\Pi x : A)R[x, \mathsf{fst}\,(g\,x)]\ )$$

Example: the type-theoretic "Theorem of Choice"

Assume A type, B type and $x : A, y : B \vdash R$ type.

$$\mathsf{ac} : \Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

$$\mathsf{ac}\,g = (\lambda(x : A).\mathsf{fst}\,(g\,x), \lambda(x : A).\ ?_4 : R[x, \mathsf{fst}\,(g\,x)]\ )$$

Example: the type-theoretic "Theorem of Choice"

Assume A type, B type and $x : A, y : B \vdash R$ type.

$$\mathsf{ac} : \Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

$$\mathsf{ac}\,g = (\lambda(x : A).\mathsf{fst}\,(g\,x), \lambda(x : A).\mathsf{snd}\,(g\,x))$$

Universes

A type $\mathcal{U}$ of types.

"À la Russell":

$$\frac{A : \mathcal{U}}{A \text{ type}}$$

"À la Tarski":

$$\frac{A : \mathcal{U}}{T(A) \text{ type}}$$

U contains "codes" for the types we are interested in. Allows computing types from data ("large elimination"), by computing a code in the universe instead.

Proof assistants such as Agda and Rocq often blur the distinction between "A type" and "$A : \mathcal{U}$".

Natural numbers

Formation

$$\frac{}{\mathbb{N}\ \mathsf{type}}$$

Introduction

$$\frac{}{0 : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{suc}\, n : \mathbb{N}}$$

Elimination

$$\frac{z : \mathbb{N} \vdash C\ \mathsf{type} \quad \begin{array}{c} c : C[z \mapsto 0] \\ x : \mathbb{N}, \bar{x} : C[z \mapsto x] \vdash d : C[z \mapsto \mathsf{suc}\, x] \end{array} \quad n : \mathbb{N}}{\mathsf{elim}_{\mathbb{N}}(C, c, d, n) : C[z \mapsto n]}$$

Computation

$$\mathsf{elim}_{\mathbb{N}}(C, c, d, 0) \equiv c : C[z \mapsto 0]$$

$$\mathsf{elim}_{\mathbb{N}}(C, c, d, \mathsf{suc}\, n) \equiv d[x \mapsto n, \bar{x} \mapsto \mathsf{elim}_{\mathbb{N}}(C, c, d, n)] : C[z \mapsto \mathsf{suc}\, n]$$

Lists

Formation

$$\frac{A \text{ type}}{\text{List } A \text{ type}}$$

Introduction

$$\frac{}{[] : \text{List } A} \qquad \frac{a : A \qquad as : \text{List } A}{(a :: as) : \text{List } A}$$

Elimination

$$\frac{as : \text{List } A \quad z : \text{List } A \vdash C \text{ type} \qquad \begin{array}{c} c : C[z \mapsto []] \\ xs : \text{List } A, \bar{xs} : C[z \mapsto xs] \vdash d : C[z \mapsto x :: xs] \end{array}}{\text{elim}_{\text{List}}(C, c, d, as) : C[z \mapsto as]}$$

Computation

$$\text{elim}_{\text{List}}(C, c, d, []) \equiv c : C[z \mapsto []]$$

$$\text{elim}_{\text{List}}(C, c, d, a :: as) \equiv d[xs \mapsto as, \bar{xs} \mapsto \text{elim}_{\text{List}}(C, c, d, as)] : C[a :: as]$$

Propositions as types

A proof of	**. . . is, according to BHK. . .**
$A \wedge B$	a proof of A and a proof of B
$A \vee B$	a proof of A or a proof of B
$A \to B$	a way to prove B given a proof of A
$\top$	always has a proof
$\bot$	never has a proof
$\neg A$	a proof that A is impossible
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	
$A \to B$	a way to prove B given a proof of A	
$\top$	always has a proof	
$\bot$	never has a proof	
$\neg A$	a proof that A is impossible	
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	
$\top$	always has a proof	
$\bot$	never has a proof	
$\neg A$	a proof that A is impossible	
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	
$\bot$	never has a proof	
$\neg A$	a proof that A is impossible	
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	$\mathbf{1}$
$\bot$	never has a proof	
$\neg A$	a proof that A is impossible	
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	$\mathbf{1}$
$\bot$	never has a proof	$\mathbf{0}$
$\neg A$	a proof that A is impossible	
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	$\mathbf{1}$
$\bot$	never has a proof	$\mathbf{0}$
$\neg A$	a proof that A is impossible	$A \to \mathbf{0}$
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	$\mathbf{1}$
$\bot$	never has a proof	$\mathbf{0}$
$\neg A$	a proof that A is impossible	$A \to \mathbf{0}$
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	$(\Pi x : A)B$
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	$\mathbf{1}$
$\bot$	never has a proof	$\mathbf{0}$
$\neg A$	a proof that A is impossible	$A \to \mathbf{0}$
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	$(\Pi x : A)B$
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	$(\Sigma x : A)B$
$s = t$		

Propositions as types

A proof of	**. . . is, according to BHK. . .**	
$A \wedge B$	a proof of A and a proof of B	$A \times B$
$A \vee B$	a proof of A or a proof of B	$A + B$
$A \to B$	a way to prove B given a proof of A	$A \to B$
$\top$	always has a proof	$\mathbf{1}$
$\bot$	never has a proof	$\mathbf{0}$
$\neg A$	a proof that A is impossible	$A \to \mathbf{0}$
$\forall(x : A).B[x]$	a way to prove $B[a]$ for any $a : A$	$(\Pi x : A)B$
$\exists(x : A).B[x]$	a choice of $a : A$ and a proof of $B[a]$	$(\Sigma x : A)B$
$s = t$		?

The identity type

Formation

$$\frac{A \text{ type} \qquad a : A \qquad a' : A}{a =_A a' \text{ type}}$$

Introduction

$$\frac{a : A}{\text{refl}_a : a =_A a}$$

Elimination

$$\frac{\begin{array}{c} x : A, y : A, z : x =_A y \vdash C \text{ type} \\ x : A \vdash d : C[x \mapsto x, y \mapsto x, z \mapsto \text{refl}_x] \quad p : a =_A a' \end{array}}{\text{elim}_=(C, d, p) : C[x \mapsto a, y \mapsto a', z \mapsto p]}$$

Computation

$$\text{elim}_=(C, d, \text{refl}_a) \equiv d[x \mapsto a] : C[x \mapsto a, y \mapsto a, z \mapsto \text{refl}_a].$$

Exercise

Use $\text{elim}_=$ to show $=$ is symmetric and transitive, and to define $\text{subst} : x =_A y \to P[x] \to P[y]$.

Contentious axioms

Many extensions of type theory relates to the identity type.

Function extensionality:

$$\frac{(\Pi x : A)\ \left(f\ x =_B g\ x\right)}{f =_{(\Pi x:A)B} g}$$

Extensional Type Theory: Adds the equality reflection rule

$$\frac{p : a =_A b}{a \equiv b : A}$$

Uniqueness of Identity Proofs:

$$\frac{p : a =_A b \qquad q : a =_A b}{p =_{a=_A b} q}$$

Univalence: "$(A =_{\mathcal{U}} B) \cong (A \cong B)$"

Contentious axioms

Many extensions of type theory relates to the identity type.

Function extensionality:

$$\frac{(\Pi x : A)\ \left(f\ x =_B g\ x\right)}{f =_{(\Pi x:A)B} g}$$

Extensional Type Theory: Adds the equality reflection rule

$$\frac{p : a =_A b}{a \equiv b : A}$$

Uniqueness of Identity Proofs:

$$\frac{p : a =_A b \qquad q : a =_A b}{p =_{a=_A b} q}$$

Univalence: "$(A =_{\mathcal{U}} B) \cong (A \cong B)$"

Are they independent of other axioms? Consistent?
Computationally well behaved?

Mathematics in type theory

Using $\mathrm{elim}_{\mathbb{N}}$, we can define addition, and multiplication.

We can then define $\mathsf{Even}(y) :\equiv (\Sigma k : \mathbb{N})\ y =_{\mathbb{N}} 2 \cdot k$.

We can construct terms

$$\begin{aligned} \text{0-even} &: \mathsf{Even}(0) \\ \text{ss-even} &: (\Pi n : \mathbb{N})\big(\mathsf{Even}(n) \to \mathsf{Even}(n+2)\big) \\ \text{even-dec} &: (\Pi n : \mathbb{N})\big(\mathsf{Even}(n) + (\mathsf{Even}(n) \to \mathbf{0})\big) \\ &\ldots \end{aligned}$$

Exercise

Use the recursion principle for $\mathbb{N}$ and a universe to show $\neg(0 =_{\mathbb{N}} 1)$. Can you do it without a universe?

Mathematics in type theory

Using $\mathsf{elim}_{\mathbb{N}}$, we can define addition, and multiplication.

We can then define $\mathsf{Even}(y) :\equiv (\Sigma k : \mathbb{N})\ y =_{\mathbb{N}} 2 \cdot k$.

We can construct terms

$$\begin{aligned} \text{0-even} &: \mathsf{Even}(0) \\ \text{ss-even} &: (\Pi n : \mathbb{N})\big(\mathsf{Even}(n) \to \mathsf{Even}(n+2)\big) \\ \text{even-dec} &: (\Pi n : \mathbb{N})\big(\mathsf{Even}(n) + (\mathsf{Even}(n) \to \mathbf{0})\big) \\ &\ldots \end{aligned}$$

Exercise

Use the recursion principle for $\mathbb{N}$ and a universe to show $\neg(0 =_{\mathbb{N}} 1)$. Can you do it without a universe?

All is well?

Sometimes you want a non-free lunch

All fine as long as we do not want to identify data with different generators, e.g.

$$\frac{1}{2} \stackrel{!}{=} \frac{5}{10}$$

Sometimes you want a non-free lunch

All fine as long as we do not want to identify data with different generators, e.g.

$$\frac{1}{2} \stackrel{!}{=} \frac{5}{10}$$

Similarly, sometimes we do not want to make a distinction between different witnesses of an existential statement.

$$|(2, \ldots)| \stackrel{!}{=} |(7, \ldots)| : (\exists p : \mathbb{N}).\mathsf{Prime}(p)$$

Sometimes you want a non-free lunch

All fine as long as we do not want to identify data with different generators, e.g.

$$\frac{1}{2} \stackrel{!}{=} \frac{5}{10}$$

Similarly, sometimes we do not want to make a distinction between different witnesses of an existential statement.

$$|(2, \ldots)| \stackrel{!}{=} |(7, \ldots)| : (\exists p : \mathbb{N}).\mathsf{Prime}(p)$$

Worse, some statements are simply inconsistent when formulated using Σ as existential quantification, even though they have topos-theoretic models (Escardó and Xu 2015).

Setoids

One approach is to not work with raw types directly, but to manually keep track of the equality they "should" have.

Setoids

One approach is to not work with raw types directly, but to manually keep track of the equality they "should" have.

Definition A setoid is given by a type A and a relation $R_A : A \to A \to \mathcal{U}$, together with terms

$$r : (\Pi a : A).R_A(a, a)$$
$$s : (\Pi a : A)(\Pi b : A).R_A(a, b) \to R_A(b, a)$$
$$t : (\Pi a : A)(\Pi b : A)(\Pi c : A).R_A(a, b) \to R_A(b, c) \to R_A(a, c)$$

Definition A setoid morphism $(A, R_A) \to (B, R_B)$ is given by a function $f : A \to B$ such that if $R_A(a, b)$ then $R_B(f(a), f(b))$.

Setoids

One approach is to not work with raw types directly, but to manually keep track of the equality they "should" have.

Definition A setoid is given by a type A and a relation $R_A : A \to A \to \mathcal{U}$, together with terms

$$r : (\Pi a : A).R_A(a, a)$$
$$s : (\Pi a : A)(\Pi b : A).R_A(a, b) \to R_A(b, a)$$
$$t : (\Pi a : A)(\Pi b : A)(\Pi c : A).R_A(a, b) \to R_A(b, c) \to R_A(a, c)$$

Definition A setoid morphism $(A, R_A) \to (B, R_B)$ is given by a function $f : A \to B$ such that if $R_A(a, b)$ then $R_B(f(a), f(b))$.

Any type A can be turned into a setoid $(A, =_A)$; appropriate for e.g. $\mathbb{N}$ and other "free" types. There is nothing to check for setoid morphisms $(A, =_A) \to (B, R_B)$.

Function setoids and quotient setoids

Given two setoids A and B, we can form the setoid $A \to B$ of setoid morphisms, and with

$$R_{A \to B}(f, g) \coloneqq (\Pi a : A).R_B(f(a), g(a))$$

Function setoids and quotient setoids

Given two setoids A and B, we can form the setoid $A \to B$ of setoid morphisms, and with

$$R_{A \to B}(f, g) \coloneqq (\Pi a : A).R_B(f(a), g(a))$$

If we have a setoid (A, R_A) and an equivalence relation $\sim$ on A (which respects R_A), then we can form the quotient setoid $(A, \sim)$.

Function setoids and quotient setoids

Given two setoids A and B, we can form the setoid $A \to B$ of setoid morphisms, and with

$$R_{A \to B}(f, g) \coloneqq (\Pi a : A).R_B(f(a), g(a))$$

If we have a setoid (A, R_A) and an equivalence relation $\sim$ on A (which respects R_A), then we can form the quotient setoid $(A, \sim)$.

In particular, we can quotient $(\Sigma a : A).P$ by the "chaotic" relation which relates everything to create $(\exists a : A).P$.

Function setoids and quotient setoids

Given two setoids A and B, we can form the setoid $A \to B$ of setoid morphisms, and with

$$R_{A \to B}(f, g) \coloneqq (\Pi a : A).R_B(f(a), g(a))$$

If we have a setoid (A, R_A) and an equivalence relation $\sim$ on A (which respects R_A), then we can form the quotient setoid $(A, \sim)$.

In particular, we can quotient $(\Sigma a : A).P$ by the "chaotic" relation which relates everything to create $(\exists a : A).P$.

Two major downsides:

- Every time we write a function, we need to prove that it is "well-defined";
- We also want type families to respect the equivalence relation, but stating this gets unwieldy.

Quotient types

Homotopy type theory (and Lean) instead has a way to quotient ordinary types.

Formation

$$\frac{A \text{ type} \qquad R : A \to A \to \mathcal{U}}{A/R \text{ type}}$$

Introduction

$$\frac{a : A}{|a| : A/R} \qquad \frac{r : R[a, b]}{\mathrm{eq}\, r : |a| =_{A/R} |b|}$$

Elimination

$$\frac{\begin{array}{c} x : A/R \vdash C \text{ type} \quad a : A \vdash d : C[x \to |a|] \quad y : A/R \\ a : A, b : A, r : R[a, b] \vdash \mathrm{subst}(\mathrm{eq}\, r, d[a]) =_{C[|b|]} d[b] \end{array}}{\mathrm{elim}_{A/R}(C, d, y) : C[x \mapsto y]}$$

Computation

$$\mathrm{elim}_{A/R}(C, d, |a|) \equiv d : C[x \mapsto |a|].$$

Consequences of adding quotient types

Simply adding quotient types to type theory destroys many nice properties of the theory.

However, it is nevertheless quite convenient to work with!

To regain the nice properties, one can move to a different type theory such as Observational Type Theory or Cubical Type Theory.

Quotient types gives function extensionality

Perhaps surprisingly, quotient types also changes the meaning of equality at seemingly unrelated types. (Hofmann 1995)

Theorem If we have the quotient type $\mathrm{Bool}/\sim$ where $\mathrm{false} \sim \mathrm{true}$, then we have

$$\mathrm{funext} : \big((\Pi x : A).f(x) =_B g(x)\big) \to f =_{\Pi(x:A).B} g$$

Quotient types gives function extensionality

Perhaps surprisingly, quotient types also changes the meaning of equality at seemingly unrelated types. (Hofmann 1995)

Theorem If we have the quotient type $\mathrm{Bool}/\sim$ where $\mathrm{false} \sim \mathrm{true}$, then we have

$$\mathrm{funext} : \big((\Pi x : A).f(x) =_B g(x)\big) \to f =_{\Pi(x:A).B} g$$

Assume $h : (\Pi x : A).f(x) =_B g(x)$. For each $x : A$, we can define $h_x : \mathrm{Bool} \to B[x]$ by

$$h_x(\mathrm{false}) = f(x)$$
$$h_x(\mathrm{true}) = g(x)$$

and $h(x)$ is a proof that $h_x(\mathrm{false}) = h_x(\mathrm{true})$.

Hence we get $\mathrm{elim}_{\mathrm{Bool}/\sim}(h_x) : \mathrm{Bool}/\sim \to B[x]$ and we have $f \equiv (\lambda x : A).f(x) \equiv (\lambda x : A).\mathrm{elim}_{\mathrm{Bool}/\sim}(h_x)(|\mathrm{false}|)$ and $g \equiv (\lambda x : A).g(x) \equiv (\lambda x : A).\mathrm{elim}_{\mathrm{Bool}/\sim}(h_x)(|\mathrm{true}|)$, hence $f = g$ since $|\mathrm{false}| = |\mathrm{true}|$.

The Axiom of Choice, again

We can "prop-truncate" any type A into one with at most one inhabitant $\|A\| := A/\sim$, and we define $(\exists x : A).P := \|(\Sigma x : A).P\|$.

The Axiom of Choice, again

We can "prop-truncate" any type A into one with at most one inhabitant $\|A\| := A/\sim$, and we define $(\exists x : A).P := \|(\Sigma x : A).P\|$.

Previously we saw a proof of

$$\Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\, x]\big)\Big)$$

Can we also prove

$$\Big((\Pi x : A)\big((\exists y : B)R[x, y]\big)\Big) \to \Big((\exists f : A \to B)\big((\Pi x : A)R[x, f\, x]\big)\Big)?$$

The Axiom of Choice, again

We can "prop-truncate" any type A into one with at most one inhabitant $\|A\| := A/\!\sim$, and we define $(\exists x : A).P := \|(\Sigma x : A).P\|$.

Previously we saw a proof of

$$\Big((\Pi x : A)\big((\Sigma y : B)R[x,y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)$$

Can we also prove

$$\Big((\Pi x : A)\big((\exists y : B)R[x,y]\big)\Big) \to \Big((\exists f : A \to B)\big((\Pi x : A)R[x, f\,x]\big)\Big)?$$

What happens when we try to do it with setoids? What about countable choice, i.e., for $A = \mathbb{N}$?

The Axiom of Choice, again

We can "prop-truncate" any type A into one with at most one inhabitant $\|A\| := A/\sim$, and we define $(\exists x : A).P := \|(\Sigma x : A).P\|$.

Previously we saw a proof of

$$\Big((\Pi x : A)\big((\Sigma y : B)R[x, y]\big)\Big) \to \Big((\Sigma f : A \to B)\big((\Pi x : A)R[x, f\, x]\big)\Big)$$

Can we also prove

$$\Big((\Pi x : A)\big((\exists y : B)R[x, y]\big)\Big) \to \Big((\exists f : A \to B)\big((\Pi x : A)R[x, f\, x]\big)\Big)?$$

No. What happens when we try to do it with setoids? No luck.
What about countable choice, i.e., for $A = \mathbb{N}$? It works for setoids.

Summary

We have seen rules for Martin-Löf type theory.

Curry-Howard correspondence: can encode logic as types.

In mathematical practice, one also needs some kind of quotient construction.

Tomorrow: Semantics of type theory.

References

Haskell B. Curry and Robert Feys. *Combinatory logic*. Studies in Logic and the Foundations of Mathematics. Elsevier, 1958.

Arend Heyting. *Mathematische Grundlagenforschung. Intuitionismus.Beweistheorie*. Springer, 1934.

Martin Hofmann. "Extensional concepts in intensional type theory". PhD thesis. University of Edinburgh, 1995. URL: https://www.lfcs.inf.ed.ac.uk/reports/95/ECS-LFCS-95-327/.

Martín Hötzel Escardó and Chuangjie Xu. "The Inconsistency of a Brouwerian Continuity Principle with the Curry–Howard Interpretation". In: *TLCA 2015*. Ed. by Thorsten Altenkirch. Vol. 38. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 153–164. DOI: 10.4230/LIPIcs.TLCA.2015.153.

William Howard. "The formulae-as-types notion of construction". Reprinted in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 1980. 1969.

Andrey Kolmogoroff. "Zur Deutung der intuitionistischen Logik". In: *Mathematische Zeitschrift* 35 (1932), pp. 58–65.

Per Martin-Löf. "An intuitionistic theory of types". In: *Twenty-Five Years of Constructive Type Theory*. Ed. by G. Sambin and J. Smith. Reprinted version of an unpublished report from 1972. Oxford University Press, 1998.

Per Martin-Löf. *Intuitionistic type theory*. Vol. 1. Studies in proof theory. Bibliopolis, 1984.