Recap

Last week:

- Propositional and predicate (higher-order) logic
  Terms and formulae are represented by (typed) $\lambda$-terms

- Proofs are terms too!
  The inhabitants of a type are its proofs
  Non-provable formulae are empty types

- Implication $\Rightarrow$ arrow type
  (Curry-Howard isomorphism)
Overview

1. Dependent types
2. Calculus of Constructions
3. Universes
4. Polymorphism
Representing first-order quantifiers

\( \forall x. P(x) \) is a formula, hence a type.

**What is a proof of \( \forall x. P(x) \)?**
Representing first-order quantifiers

\( \forall x. P(x) \) is a formula, hence a type.

⇒ What is a proof of \( \forall x. P(x) \)?

Examples:

- For the universal quantification, a proof \( p : \forall \tau. x = x \) is a function such that \( p \ u : u = u \) and \( p \ v : v = v \) for \( u, v : \tau \).
  The type of \( p \) is not of the form \( \tau \to \tau' \).

- For the existential quantification, a proof \( p : \exists \mathbb{Z}. x^2 = 4 \) could be a pair \((2, q)\) with \( q : 2^2 = 4 \), or a pair \((-2, q')\) with \( q' : (-2)^2 = 4 \).
  Again, the type of \( p \) is not of the form \( \mathbb{Z} \times \tau' \).
Dependent product

\[ \frac{\Gamma; (x : A) \vdash M : B}{\Gamma \vdash \lambda x : A. M : \Pi x : A. B} \]

This forms a formalism called \( \lambda \Pi \). It is at the basis of many formalisms of Type Theory:

- ELF, Dedukti (\( \lambda \Pi M \))
- Martin Löf’s Type Theory

Note: dependent product is a generalization of arrow types

\[ A \to B = \Pi x : A. B \]
Dependent sum ($\Sigma$-types)

\[ \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[t/x]}{\Gamma \vdash (t, u) : \Sigma x : A.B} \]

Elimination: projections

\[ \frac{\Gamma \vdash \rho : \Sigma x : A.B}{\Gamma \vdash \pi_1(\rho) : A} \quad \frac{\Gamma \vdash \rho : \Sigma x : A.B}{\Gamma \vdash \pi_2(\rho) : B[\pi_1(\rho)/x]} \]

Alternative (more general) elimination:

\[ \frac{\Gamma \vdash \rho : \Sigma x : A.B \quad \Gamma; (x : A); (y : B) \vdash f : P(x, y)}{\Gamma \vdash \text{Elim} (\rho, \lambda x\ y.f) : P \rho} \]

Note: dependent sum is a generalization of cartesian product

\[ A \times B = \Sigma x : A.B \]
Dependent types: back to the examples

\[ \vdash p : \forall x : \tau. x = x \quad \vdash u : \tau \]
\[ \vdash pu : u = u \]

\((x = x[u/x] \text{ is } u = u)\)

\[ \vdash 2 : \mathbb{Z} \quad \vdash q : 2^2 = 4 \]
\[ \vdash (2, q) : \exists x : \mathbb{Z}. x^2 = 4 \]

\((x^2 = 4[2/x] \text{ is } 2^2 = 4)\)
Distinction term/types in a higher-order logic

- Both terms and types are typed $\lambda$-terms: shall we distinguish them?
- Type judgments both relate a proposition to its proofs and ensure that types are well-formed

2 approaches:
- Martin-Löf: 2 judgments $\Gamma \vdash \tau \text{ type}$ and $\Gamma \vdash t : \tau$
- Automath, Coq: 1 judgment $\Gamma \vdash t : t'$ and a special constant (called sort or kind) which inhabitants are types (e.g. HOL’s $o$)
Martin-Löf’s Type Theory

Judgments:
- $\Gamma \vdash A \text{ type}$ (types have a specific judgment)
- $\Gamma \vdash M : A$
- $\Gamma \vdash A = B$ (equality only on well-typed terms)
- $\Gamma \vdash M = N : A$

Organized as:
- formation rules (rule for $\Pi$)
  \[
  \frac{\Gamma \vdash A \text{ Type} \quad \Gamma;\,(x:A) \vdash B \text{ Type}}{\Gamma \vdash \Pi x:A.B \text{ Type}}
  \]
- introduction rules (rule for $\lambda$)
- elimination rules (rule for application)
- computation rules ($\beta$-reduction)
Coquand and Huet (85)

Combines ideas from:

- Martin Löf’s Type Theory
- Automath
- System F (polymorphism)
Calculus of Constructions (CC)

2 sorts: **Prop** and **Type** (literature: **Type/Kind** or */□*)

\[
\begin{align*}
\Gamma \vdash T : s & \quad \Gamma \vdash (x : T) \in \Gamma & \quad \Gamma \vdash \text{Prop} : \text{Type} \\
\Gamma ; x : T \vdash & \quad \Gamma \vdash x : T & \\
\Gamma \vdash A : s_1 & \quad \Gamma ; x : A \vdash B : s_2 & \quad \Gamma \vdash \Pi x : A. B : s_2 \\
\Gamma \vdash \Pi x : A. B : s_2 & \quad \Gamma \vdash \lambda x : T. M : \Pi x : A. B & \\
\Gamma \vdash M : \Pi x : A. B & \quad \Gamma \vdash N : A & \quad \Gamma \vdash M : T \quad T =^\beta T' \quad \Gamma \vdash T' : s \\
\Gamma \vdash M N : B[N/x] & \quad \Gamma \vdash M : T' \\
\end{align*}
\]

Conversion rule (\(=^\beta\) includes \(\beta\)-reduction/expansion + congruence rules): 2 convertible types have the same inhabitants/proofs. Necessary for good metatheoretical properties.
Why is $\forall P : \text{Prop}, P$ well-typed?
Type of types

Why is \( \forall P : \text{Prop}, \ P \) well-typed?

Because \( \text{Prop} : \text{Type} \) (and \( P : \text{Type} \))
Why is $\forall P:\text{Prop}, P$ well-typed?

Because $\text{Prop}:\text{Type}$ (and $P:\text{Type}$)

We may want to accept $\forall P:\text{Type}, P \rightarrow P$ ($\forall \alpha. \alpha \rightarrow \alpha$)
Why is `forall P : Prop, P` well-typed?

Because `Prop : Type` (and `P : Type`)

We may want to accept `forall P : Type, P -> P` \((\forall \alpha. \alpha \rightarrow \alpha)\)

What is the type of `Type`?

Coq < Check Type.
Type
  : Type.
Why is `\forall P:Prop, P` well-typed?

Because `Prop:Type` (and `P:Type`)

We may want to accept `\forall P:Type, P \rightarrow P` (`\forall \alpha. \alpha \rightarrow \alpha`)

What is the type of `Type`?

Coq < Check Type.
`Type` : `Type`.

Really?
Universes

Type:Type

- Proposed by Martin-Löf (71)
  Natural idea, used by many programming languages
- Girard (72) showed that any type could be inhabited

Girard’s paradox:
- A variant of Burali-Forti’s paradox:
  ordinals do not form a set
- Simplified by Hurkens

Fix:
- hierarchy of universes
  small type, large types, very large types, etc.
- restricted quantification: predicativity
  \( \prod x : A. B \) lives in universes that contain both \( A \) and \( B \)
Calculus of Constructions with Universes ($\text{CC}_\omega$)

A hierarchy of predicative universes is added (Coquand, 1986).

\[
\text{Prop} : \text{Type}_1 : \text{Type}_2 : \text{Type}_3 \ldots
\]

Consistency proved by Luo

Set Printing Universes.
Check Type.
Type@{Top.29}
  : Type@{Top.29+1}
Polymorphism

System F (J.-Y. Girard (72), Reynolds (74)) extends the simply typed λ-calculus with a new type former (polymorphism):

\[ \forall \alpha. \tau \]

Inhabitants of this type are terms that have type \( \tau \) for all possible substitution of a type for \( \alpha \).

In Coq, explicit version (abstraction over types):

```coq
Definition id : forall X:Type, X -> X :=
  fun (X:Type) (a:X) => a.
Check id (Prop->Prop) : (Prop->Prop)->Prop->Prop.
```

- Polymorphism allows to define many datatypes, in particular arithmetic
Polymorphism allows to define a type by quantification over all types, including itself.

Check \((\forall P:\text{Prop}, P \to P) : \text{Prop}\).

⇒ Impredicativity

Allows for self-application!

Check \((\text{id} (\forall P:\text{Prop}, P \to P) \text{id}) : \forall P:\text{Prop}, P \to P\).
Polymorphism allows to define a type by quantification over all types, including itself.

Check \((\forall P:\text{Prop}, P \to P) : \text{Prop}\).

⇒ Impredicativity

Allows for self-application!

Check \((\text{id} (\forall P:\text{Prop}, P \to P) \text{id}) : \forall P:\text{Prop}, P \to P\).

But no paradox!
Polymorphism allows to define a type by quantification over all types, including itself.

Check \((\forall P : \text{Prop}, P \to P) : \text{Prop}\).

\[\Rightarrow\text{Impredicativity}\]

Allows for self-application!

Check \((\text{id} (\forall P : \text{Prop}, P \to P) \text{id}) : \forall P : \text{Prop}, P \to P\).

But no paradox!

Impredicativity rejected by Martin-Löf
Next week...

- Inductive types
Next week...

- Inductive types
- Inductive types
Next week...

- Inductive types
- Inductive types
- Inductive types