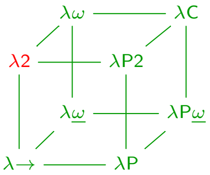
second order logic & polymorphism

Freek Wiedijk

Type Theory & Coq
2024–2025
Radboud University Nijmegen

October 11, 2024



the two remaining chapters

propositional logic	$\lambda{\rightarrow}$	simple types
predicate logic	λP	dependent types
second order logic	$\lambda 2$	polymorphic types
the Coq logic	CIC	inductive types

recap: dependent types

$$(\lambda x : A.\, M) : A \to B$$
$$(\lambda x : A.\, M) : (\Pi x : A.\, B)$$

```
(fun x : A => M) : A -> B
(fun x : A => M) : (forall x : A, B)
```

recap: inductive types

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.

Fixpoint add (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (add n' m)
  end.

Lemma add_n_O (n : nat) :
  add n O = n.
induction n as [|n' IH].
- reflexivity.
- simpl. rewrite IH.
  reflexivity.
Qed.
```

$E[\Gamma] \vdash \texttt{nat\_ind} : \ldots$

today

- second order propositional logic
- polymorphism

- inversion
- some tactics
- program extraction

- λP as a logical framework
- a classical paradox in $\lambda 2$

second order propositional logic

higher order logic

logic:	quantifies over:
first order predicate logic	objects
second order predicate logic	objects predicates
third order predicate logic	objects predicates predicates of predicates
first order propositional logic	~~objects~~
second order propositional logic	~~objects~~ ~~predicates~~ propositions

equivalent: quantification over $\{\text{sets}, \text{functions}, \text{predicates}\}$

minimal second order propositional logic

syntax

$$A, B ::= a \mid A \to B \mid \forall a.\, A$$

rules

$$\dfrac{\begin{matrix}[A^x] \\ \vdots \\ B\end{matrix}}{A \to B}\, I[x]{\to} \qquad \dfrac{\begin{matrix}\vdots \\ A \to B\end{matrix} \qquad \begin{matrix}\vdots \\ A\end{matrix}}{B}\, E{\to}$$

$$\dfrac{\begin{matrix}\vdots \\ A\end{matrix}}{\forall a.\, A}\, I\forall \qquad \dfrac{\begin{matrix}\vdots \\ \forall a.\, A\end{matrix}}{A[a := B]}\, E\forall$$

variable condition of $I\forall$: a not free in available assumptions

example proof

```
Lemma seven
    (a : Prop) :
  (forall b : Prop,
    b) -> a.
intros x.
apply x.
Qed.
```

$$\dfrac{\dfrac{\dfrac{[\forall b.\, b^x]}{a}\,E\forall}{(\forall b.\, b) \to a}\,I[x]{\to}}{\forall a.\, (\forall b.\, b) \to a}\,I\forall$$

$$\begin{gathered}\lambda a : *.\, \lambda x : (\Pi b : *.\, b).\, xa \\ : \\ \Pi a : *.\, (\Pi b : *.\, b) \to a\end{gathered}$$

```
fun (a : Prop) (x : (forall b : Prop, b)) => x a
                        :
   forall a : Prop, (forall b : Prop, b) -> a
```

constructive second order propositional logic

syntax

$$A, B ::= a \mid A \to B \mid A \wedge B \mid A \vee B \mid \neg A \mid \top \mid \bot \mid \forall a.\, A \mid \exists a.\, A$$

rules

$$\frac{\begin{array}{c}\vdots\\ A\end{array}}{\forall a.\, A}\; I\forall \qquad \frac{\begin{array}{c}\vdots\\ \forall a.\, A\end{array}}{A[a := B]}\; E\forall$$

$$\frac{\begin{array}{c}\vdots\\ A[a := B]\end{array}}{\exists a.\, A}\; I\exists \qquad \frac{\begin{array}{c}\vdots\\ \exists a.\, A\end{array} \qquad \begin{array}{c}\vdots\\ \forall a.\, A \to C\end{array}}{C}\; E\exists$$

variable condition of $E\exists$: a not free in C

example proof with existential quantifier

$$
\dfrac{
  \dfrac{
    \dfrac{
      [\exists b.\, a^x] \qquad
      \dfrac{
        \dfrac{[a^y]}{a \to a}\; I[y]{\to}
      }{\forall b.\, a \to a}\; I\forall
    }{a}\; E\exists
  }{(\exists b.\, a) \to a}\; I[x]{\to}
  \qquad
  \dfrac{
    \dfrac{[a^z]}{\exists b.\, a}\; I\exists
  }{a \to (\exists b.\, a)}\; I[z]{\to}
}{
  \dfrac{(\exists b.\, a) \leftrightarrow a}{\forall a.\, (\exists b.\, a) \leftrightarrow a}\; I\forall
}\; I\wedge
$$

```
Lemma eight (a : Prop) :
  (exists b : Prop, a) <-> a.
split.
- intros [b y]. apply y.
- intros z. exists True. apply z.
Qed.
```

defining the logical operators in minimal logic

$$
\begin{aligned}
\bot &:= \forall c.\, c \\
\top &:= \forall c.\, c \to c \\[1ex]
\neg A &:= \forall c.\, A \to c \\[1ex]
A \wedge B &:= \forall c.\, (A \to B \to c) \to c \\
A \vee B &:= \forall c.\, (A \to c) \to (B \to c) \to c \\[1ex]
\exists a.\, A &:= \forall c.\, (\forall a.\, A \to c) \to c
\end{aligned}
$$

admissibility of the proof rules

$A \wedge B := \forall c. (A \to B \to c) \to c$

$$\dfrac{\begin{matrix} \vdots \\ A \wedge B \end{matrix}}{A}\, El\wedge$$

$$\dfrac{\dfrac{\begin{matrix} \vdots \\ \forall c. (A \to B \to c) \to c \end{matrix}}{(A \to B \to A) \to A}\, E\forall \qquad \dfrac{\dfrac{[A^x]}{B \to A}\, I[y]{\to}}{A \to B \to A}\, I[x]{\to}}{A}\, E{\to}$$

$$\begin{aligned} M &: \forall c. (A \to B \to c) \to c \\ M\,A &: (A \to B \to A) \to A \\ (\lambda x : A.\, \lambda y : B.\, x) &: A \to B \to A \\ M\,A\,(\lambda x : A.\, \lambda y : B.\, x) &: A \end{aligned}$$

the polymorphic identity

$$(\lambda x : \mathsf{nat}.\, x) : \mathsf{nat} \to \mathsf{nat}$$
$$(\lambda x : \mathsf{bool}.\, x) : \mathsf{bool} \to \mathsf{bool}$$
$$\lambda a : *.\, (\lambda x : a.\, x) : \Pi a : *.\, a \to a$$

$$\mathsf{id} := \lambda a : *.\, \lambda x : a.\, x$$
$$\mathsf{id}\ \mathsf{nat} \to_\beta \lambda x : \mathsf{nat}.\, x$$
$$\mathsf{id}\ \mathsf{bool} \to_\beta \lambda x : \mathsf{bool}.\, x$$

$$\Lambda a.\, M := \lambda a : *.\, M$$
$$\forall a.\, A := \Pi a : *.\, A$$

$$\Lambda a.\, \lambda x : a.\, x : \forall a.\, a \to a$$

PTSs:

$$M, N, A, B := x \mid M N \mid \lambda x : A.\, M \mid \Pi x : A.\, B \mid * \mid \Box$$

$\lambda{\to}$:

$$\begin{aligned} A, B &:= a \mid A \to B \\ M, N &:= x \mid M N \mid \lambda x : A.\, M \end{aligned}$$

$\lambda 2$:

$$\begin{aligned} A, B &:= a \mid A \to B \mid \forall a.\, A \\ M, N &:= x \mid M N \mid \lambda x : A.\, M \mid M A \mid \Lambda a.\, M \end{aligned}$$

computing the type of a term

$$\begin{aligned}
\mathsf{type}_\Gamma(\lambda x : A.\, M) &= \Pi x : A.\, \mathsf{type}_{\Gamma, x:A}(M) \\
\mathsf{type}_\Gamma(\Pi x : A.\, B) &= \mathsf{type}_{\Gamma, x:A}(B) \\
\mathsf{type}_\Gamma(A \to B) &= \mathsf{type}_\Gamma(B) \\
\mathsf{type}_\Gamma(*) &= \Box \\
\mathsf{type}_\Gamma(x) &= \Gamma(x) \\
\mathsf{type}_\Gamma(F M) &= A[x := M] \quad \text{if } \mathsf{type}_\Gamma(F) =_\beta \Pi x : \mathsf{type}_\Gamma(M).\, A
\end{aligned}$$

$$\begin{aligned}
\mathsf{type}(\lambda a : *.\, \lambda x : a.\, x) &= \Pi a : *.\, \mathsf{type}_{a:*}(\lambda x : a.\, x) \\
&= \Pi a : *.\, \Pi x : a.\, \mathsf{type}_{a:*, x:a}(x) \\
&= \Pi a : *.\, \Pi x : a.\, a \\
&= \Pi a : *.\, a \to a
\end{aligned}$$

$$\begin{aligned}
\mathsf{type}(\Pi a : *.\, a \to a) &= \mathsf{type}_{a:*}(a \to a) \\
&= \mathsf{type}_{a:*}(a) \\
&= *
\end{aligned}$$

the rules of $\lambda 2$

the seven PTS rules:
axiom rule, variable rule, weakening rule, application rule,
abstraction rule, product rule, conversion rule

λP:

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : s}{\Gamma \vdash (\Pi x : A.\, B) : s}$$

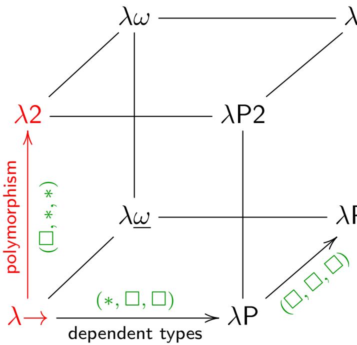
$\lambda 2$:

$$\frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash (\Pi x : A.\, B) : *}$$

lambda cube:

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi x : A.\, B) : s_3} \; (s_1, s_2, s_3) \in \mathcal{R}$$

$$
\begin{array}{ll}
 & \mathcal{R} = \\
\lambda{\rightarrow} & \{(*,*,*)\} \\
\lambda 2 & \{(*,*,*), (\Box,*,*)\} \\
\lambda \mathsf{P} & \{(*,*,*), \qquad\qquad\quad (*,\Box,\Box)\} \\
\lambda \mathsf{C} & \{(*,*,*), (\Box,*,*), (*,\Box,\Box), (\Box,\Box,\Box)\}
\end{array}
$$



examples of types in different systems

$\lambda{\rightarrow}$:

$$(\lambda x : \mathsf{nat}.\, x) : \underbrace{\mathsf{nat}}_{:\,*} \rightarrow \underbrace{\mathsf{nat}}_{:\,*}$$

$\lambda\mathsf{P}$:

$$\mathsf{vec} : \underbrace{\mathsf{nat}}_{:\,*} \rightarrow \underbrace{*}_{:\,\Box}$$

$\lambda 2$:

$$\mathsf{id} = (\lambda a : *.\, \lambda x : \mathsf{nat}.\, x) : \underbrace{\Pi a : *.}_{:\,\Box}\, \underbrace{a \rightarrow a}_{:\,*}$$

impredicative encoding of datatypes

using the non-dependent recursor as the definition

$$
\begin{aligned}
A \times_2 B &:= \Pi a : *.\,(A \to B \to a) \to a \\
A +_2 B &:= \Pi a : *.\,(A \to a) \to (B \to a) \to a \\
\mathsf{bool}_2 &:= \Pi a : *.\, a \to a \to a \\
\mathsf{nat}_2 &:= \Pi a : *.\,(a \to a) \to a \to a \\
&\ldots
\end{aligned}
$$

the type of polymorphic Church numerals:

$$
\begin{gathered}
3 := \lambda a : *.\, \lambda f : a \to a.\, \lambda x : a.\, f(f(fx)) \\
: \\
\mathsf{nat}_2 = \Pi a : *.\,(a \to a) \to a \to a
\end{gathered}
$$

inversion

when to use inversion

```
Inductive even : nat -> Prop :=
| even_0 : even 0
| even_SS n : even n -> even (S (S n)).
```

```
  H : even (S (S (S 0)))
  ============================
  False
```

```
inversion H.
```

```
H : even (S (S (S 0)))
n : nat
H1 : even (S 0)
H0 : n = S 0
============================
False
```

the induction principle of even

```
Inductive even : nat -> Prop :=
| even_0 : even 0
| even_SS n : even n -> even (S (S n)).
```

$$\frac{P(0) \qquad \forall n.\, \mathsf{even}(n) \to P(n) \to P(n+2)}{\forall n.\, \mathsf{even}(n) \to P(n)}$$

```
even_ind
     : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, even n -> P n ->
         P (S (S n))) ->
       forall n : nat, even n -> P n
```

proving that three is not even

```
1 subgoal (ID 17)

  H : even (S (S (S O)))
  ============================
  False
```

$$\mathsf{even}(3) \to \bot$$

$$\mathsf{even}(M) \to A$$

$$\forall n.\, n = M \to \mathsf{even}(n) \to A$$

$$\forall n.\, \mathsf{even}(n) \to n = M \to A$$

$$\frac{P(0) \qquad \forall n.\, \mathsf{even}(n) \to P(n) \to P(n+2)}{\forall n.\, \mathsf{even}(n) \to P(n)}$$

$$P(n) := \big(n = M \to A\big)$$

applying the induction principle

$$\frac{P(0) \qquad \forall n.\, \mathsf{even}(n) \to P(n) \to P(n+2)}{\forall n.\, \mathsf{even}(n) \to P(n)}$$

$$P(n) := \big(n = 3 \to \bot\big)$$

$$\frac{0 = 3 \to \bot \qquad \forall n.\, \mathsf{even}(n) \to (n = 3 \to \bot) \to n + 2 = 3 \to \bot}{\forall n.\, \mathsf{even}(n) \to n = 3 \to \bot}$$

cleaning up the goals

`discriminate H.`

$$\mathsf{S}\,n \neq \mathsf{O}$$
$$\mathsf{H} : \mathsf{S}\,n = \mathsf{O}$$

`injection H.`

$$\mathsf{S}\,n = \mathsf{S}\,m \to n = m$$
$$\mathsf{H} : \mathsf{S}\,n = \mathsf{S}\,m$$

works with all constructors of all inductive types

how do discriminate and injection work?

```
Definition is_S n :=
  match n with S _ => True | _ => False end.

Definition discriminate_S_O n :
    ~(S n = O) :=
  eq_ind (S n) is_S I O.


Definition S_inv n :=
  match n with S m => m | _ => O end.

Definition injection_S n m :
    S n = S m -> n = m :=
  eq_ind (S n) (fun z => n = S_inv z)
    (eq_refl n) (S m).
```

some tactics

rewriting tactics

`unfold` c.
`unfold` c `in` H.
`unfold` c `in *`.

`simpl.`
`simpl in` H.
`simpl in *.`

`rewrite` M.
`rewrite <-` M.
`rewrite` M `in` H.
`rewrite` M `in *`.

`pattern` N `at` n_1 $\ldots n_k$`;` `rewrite` M.

`subst.`

the pattern tactic

```
  x, y : A
  H : x = y
  ============================
  p x x x
```

```
pattern x at 1 3.
```

```
  x, y : A
  H : x = y
  ============================
  (fun a : A => p a x a) x
```

```
rewrite H.
```

```
  x, y : A
  H : x = y
  ============================
  p y x y
```

```
elim M.
destruct M.
induction x.
```

```
  n : nat
  ============================
  P n
```

```
induction n.
```

```
                                n : nat
                                IHn : P n
  ============================  ============================
  P O                           P (S n)
```

```
Show Proof.
```

```
(fun n : nat =>
 nat_ind (fun n0 : nat => P n0)
   ?Goal
   (fun (n0 : nat) (IHn : P n0) =>
    ?Goal0@n:=n0) n)
```

induction principle versus fix / match

```
nat_ind =
fun (P : nat -> Prop) (f : P O)
  (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | O => f
  | S n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, P n -> P (S n)) ->
       forall n : nat, P n
```

program extraction

predecessor with specification

```
Require Import Arith.

Definition pred (n : nat) : lt 0 n -> {m : nat | n = S m}.
intro H. destruct n as [|m].
- elim (lt_irrefl 0 H).
- exists m. reflexivity.
Defined.
```

two inputs: a `nat` and a proof of `lt 0 n`
two outputs: a `nat` and a proof of `n = S m`

the output type is a Sigma type of dependent pairs:

```
{m : nat | n = S m}
        ||
@sig nat (fun m : nat => n = S m)
```

the Coq term for predecessor with specification

```
pred =
fun (n : nat) (H : lt O n) =>
match n as n0 return (lt O n0 -> {m : nat | n0 = S m}) with
| O =>
    fun H0 : lt O O =>
    False_rec {m : nat | O = S m} (lt_irrefl O H0)
| S m =>
    fun _ : lt O (S m) =>
    exist (fun m0 : nat => S m = S m0) m eq_refl
end H
     : forall n : nat, lt O n -> {m : nat | n = S m}
```

extracting predecessor

```
Recursive Extraction pred.


type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)

type nat =
| O
| S of nat

(** val pred : nat -> nat **)

let pred = function
| O -> assert false (* absurd case *)
| S m -> m
```

removes all objects of which the type is in Prop
removes all the dependencies from the types

logical frameworks

systems for multiple logics

- **most proof assistants**

  logic: fixed
  theory: defined

- **logical frameworks**

  logic: defined
  theory: defined

  Automath
  Dedukti
  Isabelle
  MetaPRL
  Metamath
  Twelf

two kinds of Curry-Howard

- **propositions are types**

  single logic for each type theory

$$A : *$$
$$M : A$$

- **propositions are objects**

  different logics possible

$$\mathsf{form} : *$$
$$\mathsf{True} : \mathsf{form} \to *$$

$$A : \mathsf{form}$$
$$M : \mathsf{True}\ A$$

a λP context for a small logic

$$\dfrac{\begin{matrix}\vdots & \quad & \vdots\\ A & & B\end{matrix}}{A \wedge B}\,I\wedge \qquad \dfrac{\begin{matrix}\vdots\\ A \wedge B\end{matrix}}{A}\,El\wedge \qquad \dfrac{\begin{matrix}\vdots\\ A \wedge B\end{matrix}}{B}\,Er\wedge$$

$$\begin{aligned}
\mathsf{form} &: *\\
\wedge &: \mathsf{form} \to \mathsf{form} \to \mathsf{form}\\[1em]
\mathsf{True} &: \mathsf{form} \to *\\
I\wedge &: \Pi A : \mathsf{form}.\, \Pi B : \mathsf{form}.\, \mathsf{True}\ A \to \mathsf{True}\ B \to \mathsf{True}\,(\wedge\, A\, B)\\
El\wedge &: \Pi A : \mathsf{form}.\, \Pi B : \mathsf{form}.\, \mathsf{True}\,(\wedge\, A\, B) \to \mathsf{True}\ A\\
El\wedge &: \Pi A : \mathsf{form}.\, \Pi B : \mathsf{form}.\, \mathsf{True}\,(\wedge\, A\, B) \to \mathsf{True}\ B
\end{aligned}$$

impredicativity

a $\lambda 2$ typing judgment

$$b : * \vdash (\Pi a : *.\, a \to b) : *$$

$$\begin{aligned}
\mathsf{type}_\Gamma(\lambda x : A.\, M) &= \Pi x : A.\, \mathsf{type}_{\Gamma, x:A}(M) \\
\mathsf{type}_\Gamma(\Pi x : A.\, B) &= \mathsf{type}_{\Gamma, x:A}(B) \\
\mathsf{type}_\Gamma(A \to B) &= \mathsf{type}_\Gamma(B)
\end{aligned}$$

$$\mathsf{type}_{b:*}(\Pi a : *.\, a \to b) = \mathsf{type}_{b:*,\, a:*}(a \to b) = \mathsf{type}_{b:*,\, a:*}(b) = *$$

the problem with a set theoretic interpretation

$$\mathsf{bool} : * \vdash (\Pi a : *.\, \underbrace{a \to \mathsf{bool}}_{\text{the power set of } a}) : *$$

$$X := \prod_{a \in \mathsf{Set}} \overbrace{\mathcal{P}(a)}^{\text{the power set of } a} \in \mathsf{Set}$$

$$X = \mathcal{P}(X) \times \underbrace{\prod_{\substack{a \in \mathsf{Set} \\ a \neq X}} \mathcal{P}(a)}_{\text{non-empty}}$$

but: X has a smaller cardinality than $\mathcal{P}(X)$!

diagonalisation

$$X := (\Pi a : *.\, a \to \mathsf{bool}) : *$$

$$f := \lambda a : *.\, \lambda x : a.\, (\mathsf{if}\ a = X\ \mathsf{then}\ \neg(xXx)\ \mathsf{else}\ \mathsf{true})$$

$$\begin{aligned} f &: X \\ f &: \Pi a : *.\, a \to \mathsf{bool} \\ fX &: X \to \mathsf{bool} \\ fXf &: \mathsf{bool} \end{aligned}$$

$$\begin{aligned} fXf &= \neg(fXf) \\ \mathsf{true} &= \mathsf{false} \end{aligned}$$

needs if-then-else on equality of types
and proof irrelevance for equality

impredicativity in Coq

Set is predicative
Prop is impredicative

```
Inductive bool : Set := true : bool | false : bool.
Check (forall a : Set, a -> bool).
```

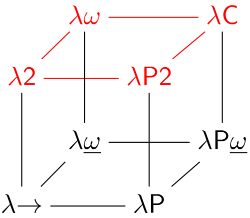
```
forall a : Set, a -> bool
     : Type
```

$$b : * \vdash (\Pi a : *.\ a \to b) : \square$$

```
Inductive pbool : Prop := ptrue : pbool | pfalse : pbool.
Check (forall a : Prop, a -> pbool).
```

```
forall a : Prop, a -> pbool
     : Prop
```

$$b : * \vdash (\Pi a : *.\ a \to b) : *$$



impredicativity = defining something by quantifying over a domain that contains the thing you are defining

generally not inconsistent with classical mathematics

you often define the smallest set closed under some operations as the intersection of all such sets

conclusion

summary of Femke's part of the course

- untyped lambda calculus

$$M, N ::= x \mid MN \mid \lambda x.\, M$$

$$(\lambda x.\, M)N \to_\beta M[x := N]$$

- typed lambda calculus = type theories

$$\Gamma \vdash M : A$$

- Curry-Howard correspondence
  proof terms

propositional logic	STT	simple types
predicate logic	λP	dependent types
second order logic	λ2	polymorphic types

summary of the first part of the course (continued)

- CIC = the type theory of Coq
  inductive types

  induction principles = recursion principles

  $$M \to_{\beta\delta\iota\zeta\eta} N$$

- Coq as a functional programming language

  `Inductive`
  `Fixpoint match`

- Coq as a proof language

  `Lemma Definition Qed Defined`
  `intros apply`
  `reflexivity split left right exists`
  `elim destruct induction inversion`
  `unfold simpl compute rewrite pattern clear subst`
  `Check Print Show Eval`

thank you

thanks for listening!

questions?

$$
\begin{tikzcd}
 & \lambda\omega \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash] & & \lambda C \arrow[dl, dash] \arrow[dd, dash] \\
\lambda 2 \arrow[rr, dash] \arrow[dd, dash] & & \lambda P2 \arrow[dd, dash] & \\
 & \lambda\underline{\omega} \arrow[rr, dash] \arrow[dl, dash] & & \lambda P\underline{\omega} \arrow[dl, dash] \\
\lambda{\to} \arrow[rr, dash] & & \lambda P &
\end{tikzcd}
$$

table of contents

contents