

second order logic & polymorphism

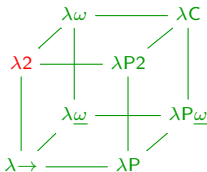
Freek Wiedijk

Type Theory & Coq

2023–2024

Radboud University Nijmegen

October 6, 2023



introduction

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

$(\lambda x : A. M) : A \rightarrow B$

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

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

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

recap: inductive types

```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat.
```

```
Fixpoint add (n m : nat) : nat :=  
  match n with  
  | 0 => m  
  | S n' => S (add n' m)  
  end.
```

```
Lemma add_n_0 (n : nat) :  
  add n 0 = n.  
induction n as [|n' IH].  
- reflexivity.  
- simpl. rewrite IH.  
  reflexivity.  
Qed.
```

$E[\Gamma] \vdash \text{nat_ind} : \dots$

today

- ▶ second order propositional logic
- ▶ polymorphism

- ▶ detours
- ▶ inversion
- ▶ some tactics
- ▶ program extraction

- ▶ λ P as a logical framework
- ▶ a classical paradox in $\lambda 2$

second order propositional logic

higher order logic

logic:

first order predicate logic

second order predicate logic

third order predicate logic

first order propositional logic

second order propositional logic

quantifies over:

objects

objects
predicates

objects
predicates
predicates of predicates

objects

objects
predicates
propositions

equivalent: quantification over {sets, functions, predicates}

minimal second order propositional logic

syntax

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

rules

$$\frac{\begin{array}{c} [A^x] \\ \vdots \\ B \end{array}}{A \rightarrow B} I[x] \rightarrow \qquad \frac{\begin{array}{c} \vdots \\ A \rightarrow B \end{array} \quad \begin{array}{c} \vdots \\ A \end{array}}{B} E \rightarrow$$
$$\frac{\begin{array}{c} \vdots \\ A \end{array}}{\forall a. A} I\forall \qquad \frac{\begin{array}{c} \vdots \\ \forall a. A \end{array}}{A[a := B]} E\forall$$

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.
```

$$\frac{[\forall b. b^x]}{a} E\forall$$
$$\frac{\frac{a}{(\forall b. b) \rightarrow a} I[x] \rightarrow}{\forall a. (\forall b. b) \rightarrow a} I\forall$$

$$\lambda a : *. \lambda x : (\Pi b : *. b). xa$$
$$:$$
$$\Pi a : *. (\Pi b : *. b) \rightarrow a$$

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

$$:$$

```
forall a : Prop, (forall b : Prop, b) -> a
```


constructive second order propositional logic

syntax

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

rules

$$\frac{\vdots}{A} \frac{A}{\forall a. A} I\forall$$

$$\frac{\vdots}{\forall a. A} \frac{\forall a. A}{A[a := B]} E\forall$$

$$\frac{\vdots}{A[a := B]} \frac{A[a := B]}{\exists a. A} I\exists$$

$$\frac{\vdots}{\exists a. A} \frac{\vdots}{\forall a. A \rightarrow C} \frac{\exists a. A \quad \forall a. A \rightarrow C}{C} E\exists$$

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

example proof with existential quantifier

$$\frac{\frac{\frac{[a^y]}{a \rightarrow a} I[y] \rightarrow}{\forall b. a \rightarrow a} I\forall}{[\exists b. a^x]} E\exists \quad \frac{\frac{[a^z]}{\exists b. a} I\exists}{a \rightarrow (\exists b. a)} I\wedge}{\frac{a}{(\exists b. a) \rightarrow a} I[x] \rightarrow \quad \frac{(\exists b. a) \leftrightarrow a}{\forall a. (\exists b. a) \leftrightarrow a} I\forall} I\wedge$$

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

$$\perp := \forall c. c$$

$$\top := \forall c. c \rightarrow c$$

$$\neg A := \forall c. A \rightarrow c$$

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

$$A \vee B := \forall c. (A \rightarrow c) \rightarrow (B \rightarrow c) \rightarrow c$$

$$\exists a. A := \forall c. (\forall a. A \rightarrow c) \rightarrow c$$

admissibility of the proof rules

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

$$\begin{array}{c}
 \vdots \\
 \frac{A \wedge B}{A} \text{El}\wedge \\
 \\
 \frac{\forall c. (A \rightarrow B \rightarrow c) \rightarrow c}{(A \rightarrow B \rightarrow A) \rightarrow A} \text{E}\forall \quad \frac{\frac{[A^x]}{B \rightarrow A} I[y] \rightarrow}{A \rightarrow B \rightarrow A} I[x] \rightarrow}{A} \text{E}\rightarrow
 \end{array}$$

$M : \forall c. (A \rightarrow B \rightarrow c) \rightarrow c$

$MA : (A \rightarrow B \rightarrow A) \rightarrow A$

$(\lambda x : A. \lambda y : B. x) : A \rightarrow B \rightarrow A$

$MA(\lambda x : A. \lambda y : B. x) : A$

polymorphism

the polymorphic identity

$$(\lambda x : \text{nat}. x) : \text{nat} \rightarrow \text{nat}$$
$$(\lambda x : \text{bool}. x) : \text{bool} \rightarrow \text{bool}$$
$$\lambda a : *. (\lambda x : a. x) : \Pi a : *. a \rightarrow a$$
$$\text{id} := \lambda a : *. \lambda x : a. x$$
$$\text{id nat} \rightarrow_{\beta} \lambda x : \text{nat}. x$$
$$\text{id bool} \rightarrow_{\beta} \lambda x : \text{bool}. x$$
$$\Lambda a. M := \lambda a : *. M$$
$$\forall a. A := \Pi a : *. A$$
$$\Lambda a. \lambda x : a. x : \forall a. a \rightarrow a$$

stratified syntax

PTSs:

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

$\lambda \rightarrow$:

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

$\lambda 2$:

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

computing the type of a term

$$\text{type}_\Gamma(\lambda x : A. M) = \Pi x : A. \text{type}_{\Gamma, x:A}(M)$$

$$\text{type}_\Gamma(\Pi x : A. B) = \text{type}_{\Gamma, x:A}(B)$$

$$\text{type}_\Gamma(A \rightarrow B) = \text{type}_\Gamma(B)$$

$$\text{type}_\Gamma(*) = \square$$

$$\text{type}_\Gamma(x) = \Gamma(x)$$

$$\text{type}_\Gamma(FM) = A[x := M] \quad \text{if } \text{type}_\Gamma(F) =_\beta \Pi x : \text{type}_\Gamma(M). A$$

$$\text{type}(\lambda a : *. \lambda x : a. x) = \Pi a : *. \text{type}_{a:*}(\lambda x : a. x)$$

$$= \Pi a : *. \Pi x : a. \text{type}_{a:*, x:a}(x)$$

$$= \Pi a : *. \Pi x : a. a$$

$$= \Pi a : *. a \rightarrow a$$

$$\text{type}(\Pi a : *. a \rightarrow a) = \text{type}_{a:*}(a \rightarrow a)$$

$$= \text{type}_{a:*}(a)$$

$$= *$$

the rules of λ_2

the seven PTS rules:

axiom rule, variable rule, weakening rule, application rule,
abstraction rule, **product rule**, conversion rule

λ_P :

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

λ_2 :

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

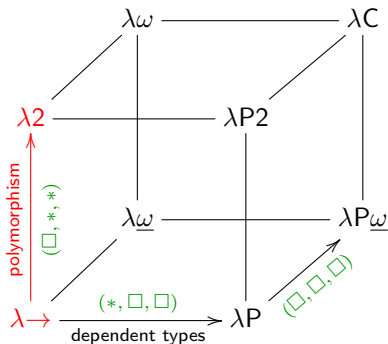
lambda cube:

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

the lambda cube

$\mathcal{R} =$

$\lambda \rightarrow$	$\{(*, *, *)\}$
$\lambda 2$	$\{(*, *, *), (\square, *, *)\}$
λP	$\{(*, *, *), (*, \square, \square)\}$
λC	$\{(*, *, *), (\square, *, *), (*, \square, \square), (\square, \square, \square)\}$



examples of types in different systems

$\lambda \rightarrow$:

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

λP :

$$\text{vec} : \underbrace{\text{nat}}_{:*} \rightarrow \underbrace{*}_{:\square}$$

$\lambda 2$:

$$\text{id} = (\lambda a : *. \lambda x : \text{nat}. x) : \underbrace{\Pi a : *.}_{:\square} \underbrace{a \rightarrow a}_{:*}$$

impredicative encoding of datatypes

using the non-dependent recursor as the definition

$$A \times_2 B := \Pi a : *. (A \rightarrow B \rightarrow a) \rightarrow a$$

$$A +_2 B := \Pi a : *. (A \rightarrow a) \rightarrow (B \rightarrow a) \rightarrow a$$

$$\text{bool}_2 := \Pi a : *. a \rightarrow a \rightarrow a$$

$$\text{nat}_2 := \Pi a : *. (a \rightarrow a) \rightarrow a \rightarrow a$$

...

the type of polymorphic Church numerals:

$$\mathbb{3} := \lambda a : *. \lambda f : a \rightarrow a. \lambda x : a. f(f(fx))$$

:

$$\text{nat}_2 = \Pi a : *. (a \rightarrow a) \rightarrow a \rightarrow a$$

detours

proof normalization

detour =

introduction rule directly followed by
elimination rule for the same connective

cut = corresponding notion in sequent calculus

can be eliminated: detour elimination, cut elimination

proof \longleftrightarrow proof term

detour \longleftrightarrow redex

proof in normal form \longleftrightarrow term in normal form

detour for implication

'prove a lemma A and then prove B using this lemma'

detour elimination is inlining the proof₁ of the lemma everywhere

$$\frac{\frac{[A^H] \quad \vdots_2}{B} \quad \frac{A \rightarrow B}{A \rightarrow B} \quad I[H] \rightarrow \quad \frac{\vdots_1 \quad A}{A} \quad E \rightarrow}{B} \quad \rightarrow_{\beta} \quad \frac{\vdots_1 \quad A \quad \vdots_2}{B}$$

$$(\lambda H : A. P_2)P_1 \rightarrow_{\beta} P_2[H := P_1]$$

detour for the universal quantifier

generalize the statement $A[x := M]$ to A with arbitrary x

elimination is specializing the proof₁ to M

$$\frac{\frac{\vdots_1}{A} \text{IV}}{\forall x. A} \text{EV} \quad \longrightarrow_{\beta} \quad \frac{\vdots_1[x := M]}{A[x := M]}$$

$$(\lambda x : D. P_1)M \rightarrow_{\beta} P_1[x := M]$$

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)).
```

$$\frac{P(0) \quad \forall n. \text{even}(n) \rightarrow P(n) \rightarrow P(n+2)}{\forall n. \text{even}(n) \rightarrow P(n)}$$

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 0)))

=====

False

even(3) $\rightarrow \perp$

even(M) $\rightarrow A$

$\forall n. n = M \rightarrow \text{even}(n) \rightarrow A$

$\forall n. \text{even}(n) \rightarrow n = M \rightarrow A$

$P(0) \quad \forall n. \text{even}(n) \rightarrow P(n) \rightarrow P(n + 2)$

$\forall n. \text{even}(n) \rightarrow P(n)$

$P(n) := (n = M \rightarrow A)$

applying the induction principle

$$\frac{P(0) \quad \forall n. \text{even}(n) \rightarrow P(n) \rightarrow P(n+2)}{\forall n. \text{even}(n) \rightarrow P(n)}$$

$$P(n) := (n = 3 \rightarrow \perp)$$

$$\frac{0 = 3 \rightarrow \perp \quad \forall n. \text{even}(n) \rightarrow (n = 3 \rightarrow \perp) \rightarrow n + 2 = 3 \rightarrow \perp}{\forall n. \text{even}(n) \rightarrow n = 3 \rightarrow \perp}$$

cleaning up the goals

discriminate H .

$$\begin{aligned} & S n \neq 0 \\ H : S n = 0 \end{aligned}$$

injection H .

$$\begin{aligned} & S n = S m \rightarrow n = m \\ H : S n = S m \end{aligned}$$

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_0 n :  
  ~(S n = 0) :=  
  eq_ind (S n) is_S I 0.
```

```
Definition S_inv n :=  
  match n with S m => m | _ => 0 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  $n_1 \dots n_k$ ; 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
```

elimination tactics

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

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

```
induction n.
```

```
                                n : nat  
                                IHn : P n  
                                =====  
=====                          P (S n)  
P 0
```

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 0)
  (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | S n0 => f0 n0 (F n0)
end
: forall P : nat -> Prop,
  P 0 ->
  (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:

$$\begin{array}{c} \{m : \text{nat} \mid n = S m\} \\ \parallel \\ \text{@sig nat (fun m : nat => n = S m)} \end{array}$$

the Coq term for predecessor with specification

```
pred =
fun (n : nat) (H : lt 0 n) =>
match n as n0 return (lt 0 n0 -> m : nat | n0 = S m) with
| 0 =>
    fun H0 : lt 0 0 =>
      False_rec m : nat | 0 = S m (lt_irrefl 0 H0)
| S m =>
    fun _ : lt 0 (S m) =>
      exist (fun m0 : nat => S m = S m0) m eq_refl
end H
: forall n : nat, lt 0 n -> m : nat | n = S m
```

extracting predecessor

Recursive Extraction pred.

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

type nat =
| 0
| S of nat

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

let pred = function
| 0 -> 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

 Twelf

 Automath

 Dedukti

  Isabelle

 MetaPRL

 Metamath

two kinds of Curry-Howard

► **propositions are types**

single logic for each type theory

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

► **propositions are objects**

different logics possible

$$\text{form} : *$$
$$\text{True} : \text{form} \rightarrow *$$
$$A : \text{form}$$
$$M : \text{True } A$$

a λP context for a small logic

$$\frac{\begin{array}{c} \vdots \\ A \end{array} \quad \begin{array}{c} \vdots \\ B \end{array}}{A \wedge B} I\wedge \quad \frac{\begin{array}{c} \vdots \\ A \wedge B \end{array}}{A} El\wedge \quad \frac{\begin{array}{c} \vdots \\ A \wedge B \end{array}}{B} Er\wedge$$

form : *

\wedge : form \rightarrow form \rightarrow form

True : form \rightarrow *

$I\wedge$: $\Pi A : \text{form}. \Pi B : \text{form}. \text{True } A \rightarrow \text{True } B \rightarrow \text{True } (\wedge A B)$

$El\wedge$: $\Pi A : \text{form}. \Pi B : \text{form}. \text{True } (\wedge A B) \rightarrow \text{True } A$

$Er\wedge$: $\Pi A : \text{form}. \Pi B : \text{form}. \text{True } (\wedge A B) \rightarrow \text{True } B$

impredicativity

a $\lambda 2$ typing judgment

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

$$\text{type}_{\Gamma}(\lambda x : A. M) = \Pi x : A. \text{type}_{\Gamma, x:A}(M)$$

$$\text{type}_{\Gamma}(\Pi x : A. B) = \text{type}_{\Gamma, x:A}(B)$$

$$\text{type}_{\Gamma}(A \rightarrow B) = \text{type}_{\Gamma}(B)$$

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

the problem with a set theoretic interpretation

$$\text{bool} : * \vdash (\prod a : *. a \rightarrow \text{bool}) : *$$

the power set of a

the power set of a

$$X := \prod_{a \in \text{Set}} \mathcal{P}(a) \in \text{Set}$$

$$X = \mathcal{P}(X) \times \prod_{\substack{a \in \text{Set} \\ a \neq X}} \mathcal{P}(a)$$

non-empty

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

$$X := (\Pi a : *. a \rightarrow \text{bool}) : *$$
$$f := \lambda a : *. \lambda x : a. (\text{if } a = X \text{ then } \neg(xXx) \text{ else true})$$
$$f : X$$
$$f : \Pi a : *. a \rightarrow \text{bool}$$
$$fX : X \rightarrow \text{bool}$$
$$fXf : \text{bool}$$
$$fXf = \neg(fXf)$$
$$\text{true} = \text{false}$$

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
```

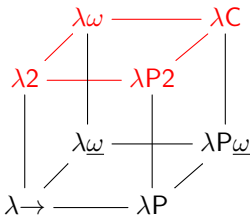
$$b : * \vdash (\prod a : *. a \rightarrow b) : \square$$

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

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

$$b : * \vdash (\prod a : *. a \rightarrow b) : *$$

the impredicative plane of the lambda cube



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 the first part of the course

- ▶ untyped lambda calculus

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

$$(\lambda x. M)N \rightarrow_{\beta} M[x := N]$$

- ▶ 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	$\lambda 2$	polymorphic types

summary of the first part of the course (continued)

- ▶ CIC = the type theory of Coq
inductive types

induction principles = recursion principles

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

- ▶ 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?

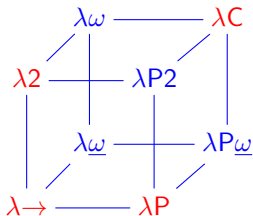


table of contents

contents

introduction

second order propositional logic

polymorphism

detours

inversion

some tactics

program extraction

logical frameworks

impredicativity

conclusion

table of contents