

Proving with Computer Assistance
Lecture 10

Higher Order Logic and the Calculus of Constructions

Herman Geuvers

For the slides, thanks to: Freek Wiedijk

The Barendregt cube

Barendregt cube: 8 typed λ -calculi, defined in one coherent way.

Generalization: Berardi & Terlouw: Pure Type Systems

framework for defining and studying typed λ -calculi

PTS = pure type system

the PTS rules are basically the λP rules as presented before.

variations on the product rule

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A. B : s_2}$$

$$\lambda P \quad s_1 = *, s_2 \in \{*, \square\}$$

$$(s_1, s_2) \in \{(*, *), (*, \square)\}$$

$$\lambda \rightarrow \quad (s_1, s_2) \in \{(*, *)\}$$

$$\lambda 2 \quad (s_1, s_2) \in \{(*, *), (\square, *)\}$$

$$\lambda C \quad (s_1, s_2) \in \{(*, *), (*, \square), (\square, *), (\square, \square)\}$$

(**axiom**) $\vdash * : \square$

(**var**)
$$\frac{\Gamma \vdash A : s}{\Gamma, x:A \vdash x : A}$$

(**weak**)
$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x:A \vdash M : C}$$

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

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

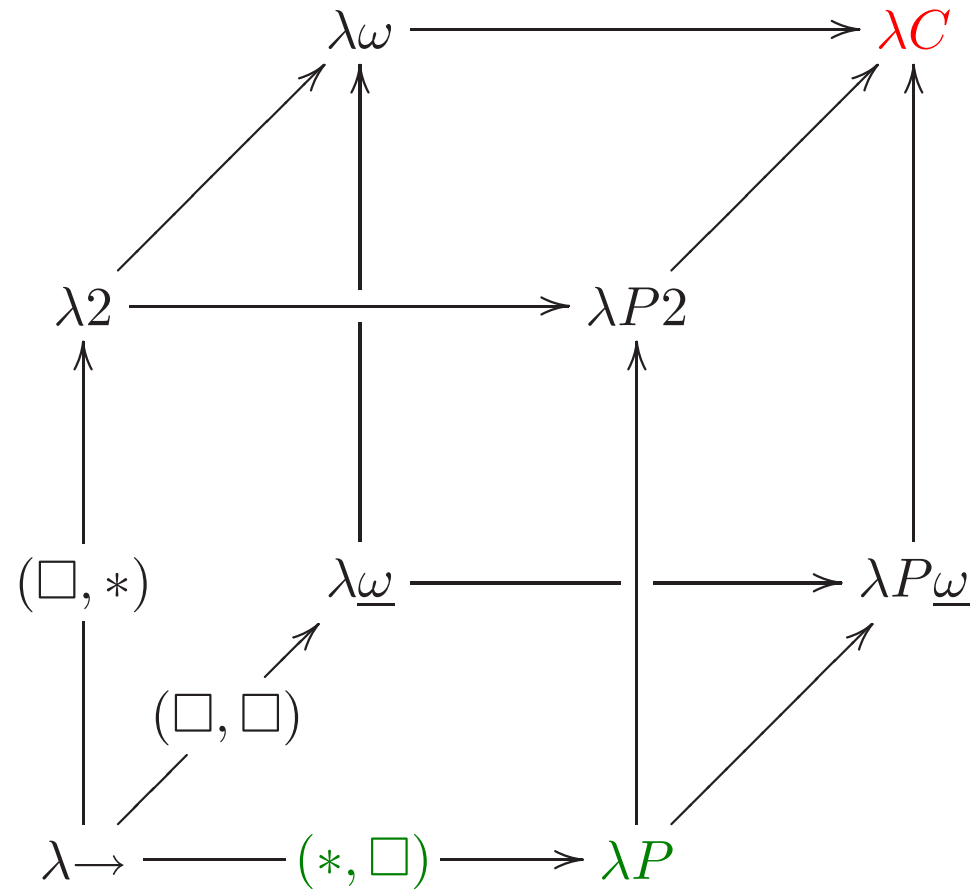
$$(\text{app}) \quad \frac{\Gamma \vdash M : \Pi x:A. B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

$$(\text{conv}) \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \text{ if } A =_{\beta} B$$

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

System	\mathcal{R}
$\lambda \rightarrow$	$(*, *)$
$\lambda 2$ (system F)	$(*, *) \quad (\square, *)$
λP (LF)	$(*, *) \quad (*, \square)$
$\lambda \bar{\omega}$	$(*, *) \quad (\square, \square)$
$\lambda P 2$	$(*, *) \quad (\square, *) \quad (*, \square)$
$\lambda \omega$ (system Fω)	$(*, *) \quad (\square, *) \quad (\square, \square)$
$\lambda P \bar{\omega}$	$(*, *) \quad (*, \square) \quad (\square, \square)$
$\lambda P \omega$ (CC)	$(*, *) \quad (\square, *) \quad (*, \square) \quad (\square, \square)$

the Barendregt cube



Calculus of Constructions

$\lambda \rightarrow$ in this presentation is equivalent to $\lambda \rightarrow$ as presented before.
Similarly for $\lambda 2$, λP , ... This **cube** also gives a **fine structure** for the

Calculus of Constructions, CC (Coquand and Huet)

- Polymorphic **data types** on the $*$ -level,
e.g. $\Pi \alpha : * . \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha : *$.
- **Predicate domains** on the \square -level,
e.g. $N \rightarrow N \rightarrow * : \square$
- **Logic** on the $*$ -level,
e.g. $\varphi \wedge \psi := \Pi \alpha : * . (\varphi \rightarrow \psi \rightarrow \alpha) \rightarrow \alpha : *$.
- **Universal quantification** (first and higher order),
e.g. $\Pi P : N \rightarrow * . \Pi x : N . P x \rightarrow P x : *$.

Examples

- Induction

$$\forall P:N \rightarrow * ((P\ 0) \rightarrow (\forall x:N.(P\ x \rightarrow P(S\ x))) \rightarrow \forall x:N.P\ x)$$

- Higher order predicates/functions: transitive closure of a relation R

$$\lambda R : A \rightarrow A \rightarrow * . \lambda x, y : A .$$

$$(\forall Q : A \rightarrow A \rightarrow * . (\text{trans}(Q) \rightarrow (R \subseteq Q) \rightarrow Q\ x\ y))$$

of type

$$(A \rightarrow A \rightarrow *) \rightarrow (A \rightarrow A \rightarrow *)$$

Higher order logic HOL

In higher order logic (originally due to Church[1940]) we have:

- higher order domains: D , $D \rightarrow \text{Prop}$, $(D \rightarrow \text{Prop}) \rightarrow \text{Prop}$, etc (sets of predicates over predicates over ...).
- higher order function domains: $(D \rightarrow D) \rightarrow D$, $((D \rightarrow D) \rightarrow D) \rightarrow D$, etc.
- \forall -quantification over all domains

Full Church higher order logic [1940]

Church has additional things that we will not consider now:

- **Negation** connective with rules
- Classical logic

$$\frac{\Delta \vdash \neg\neg\varphi}{\Delta \vdash \varphi}$$

- Define other connectives in terms of $\rightarrow, \forall, \neg$ (classically).
- **Choice** operator $\iota_\sigma : (\sigma \rightarrow *) \rightarrow \sigma$
- Rule for ι :

$$\frac{\Delta \vdash \exists!x:\sigma.P x}{\Delta \vdash P(\iota_\sigma P)}$$

Full Church higher order logic [1940]

Church' original higher order logic is basically the logic of the theorem prover HOL (Gordon, Melham, Harrison) and of Isabelle-HOL (Paulson, Nipkow).

The need for a **conversion** rule

$$\frac{\Delta \vdash \forall P:N \rightarrow * . (\dots Pc \dots)}{\Delta \vdash (\dots (\lambda y:N. y > 0)c \dots)} \forall\text{-elim}$$
$$\frac{\Delta \vdash (\dots (\lambda y:N. y > 0)c \dots)}{\Delta \vdash (\dots c > 0 \dots)} \text{conv}$$

Definability of other connectives (constructively)

$$\begin{aligned}\perp & := \forall \alpha : * . \alpha \\ \varphi \wedge \psi & := \forall \alpha : * . (\varphi \rightarrow \psi \rightarrow \alpha) \rightarrow \alpha \\ \varphi \vee \psi & := \forall \alpha : * . (\varphi \rightarrow \alpha) \rightarrow (\psi \rightarrow \alpha) \rightarrow \alpha \\ \exists x : \sigma . \varphi & := \forall \alpha : * . (\forall x : \sigma . \varphi \rightarrow \alpha) \rightarrow \alpha\end{aligned}$$

Idea:

The definition of a connective is an encoding of the **elimination** rule.

Existential quantifier

$$\exists x:\sigma.\varphi := \forall\alpha:*. (\forall x:\sigma.\varphi \rightarrow \alpha) \rightarrow \alpha$$

Derivation of the elimination rule in HOL.

$$\frac{\begin{array}{c} [\varphi] \\ \vdots \\ \exists x:\sigma.\varphi \quad C \end{array}}{C} \quad x \notin \text{FV}(C, \text{ass.}) \quad \frac{\frac{\exists x:\sigma.\varphi}{(\forall x:\sigma.\varphi \rightarrow C) \rightarrow C} \quad \frac{\begin{array}{c} [\varphi] \\ \vdots \\ C \end{array}}{\forall x:\sigma.\varphi \rightarrow C}}{C}$$

Existential quantifier

$$\exists x:\sigma.\varphi := \forall\alpha:*. (\forall x:\sigma.\varphi \rightarrow \alpha) \rightarrow \alpha$$

Derivation of the introduction rule in HOL.

$$\frac{\frac{\frac{\varphi[t/x]}{\exists x:\sigma.\varphi}}{\varphi[t/x]} \quad \frac{\frac{[\forall x:\sigma.\varphi \rightarrow \alpha]}{\varphi[t/x] \rightarrow \alpha}}{\alpha}}{(\forall x:\sigma.\varphi \rightarrow \alpha) \rightarrow \alpha}}{\exists x:\sigma.\varphi}$$

Equality

Equality is **definable** in higher order logic:

t and q terms are equal if they share the same properties
(**Leibniz** equality)

Definition in HOL (for $t, q : A$):

$$t =_A q := \forall P:A \rightarrow *. (Pt \rightarrow Pq)$$

- This equality is **reflexive** and **transitive** (easy)
- It is also **symmetric**(!) Trick: find a “smart” predicate P

Exercise: Prove reflexivity, transitivity and symmetry of $=_A$.

Exercise

The **transitive closure** of a binary relation R on A has been defined as follows.

$$\text{trclos } R \quad := \quad \lambda x, y: A. \\ (\forall Q: A \rightarrow A \rightarrow * . (\text{trans}(Q) \rightarrow (R \subseteq Q) \rightarrow (Q \ x \ y))).$$

1. Prove that the **transitive closure** is **transitive**.
2. Prove that the **transitive closure of R** contains R .

Derivation in HOL

$$\frac{\frac{\frac{\forall x^A y^A R x y \rightarrow R y x \rightarrow \perp}{\forall y^A R x y \rightarrow R y x \rightarrow \perp}}{R x x \rightarrow R x x \rightarrow \perp} [R x x]}{R x x \rightarrow \perp} [R x x]}{\perp}}{R x x \rightarrow \perp}}{\forall x^A . R x x \rightarrow \perp}$$

Derivation in HOL, with terms

$$\begin{array}{c}
 z : \forall x^A y^A R x y \rightarrow R y x \rightarrow \perp \\
 \hline
 zx : \forall y^A R x y \rightarrow R y x \rightarrow \perp \\
 \hline
 zxx : R x x \rightarrow R x x \rightarrow \perp \quad [q : R x x] \\
 \hline
 zxxq : R x x \rightarrow \perp \quad [q : R x x] \\
 \hline
 zxxqq : \perp \\
 \hline
 \lambda q : (R x x). zxxqq : R x x \rightarrow \perp \\
 \hline
 \lambda x : A. \lambda q : (R x x). zxxqq : \forall x^A. R x x \rightarrow \perp
 \end{array}$$

Typing judgement in CC:

$$\begin{array}{l}
 A : *, R : A \rightarrow A \rightarrow *, \quad z : \Pi x, y : A. (R x y \rightarrow R y x \rightarrow \perp) \vdash \\
 \lambda x : A \lambda q : (R x x). z x x q q : (\Pi x : A. R x x \rightarrow \perp)
 \end{array}$$

CC versus HOL

Question: is the type theory CC really isomorphic with HOL?

No: only if we **disambiguate** $*$ into Set and Prop (or $*_s$ and $*_p$).
This is the type theory of Coq.

Properties of CC

- **Uniqueness of types**
If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_{\beta} B$.
- **Subject Reduction**
If $\Gamma \vdash M : A$ and $M \longrightarrow_{\beta} N$, then $\Gamma \vdash N : A$.
- **Strong Normalization**
If $\Gamma \vdash M : A$, then all β -reductions from M terminate.

Proof of SN is a **really difficult**.

Decidability Questions

$\Gamma \vdash M : \sigma?$ TCP

$\Gamma \vdash M : ?$ TSP

$\Gamma \vdash ? : \sigma$ TIP

For **CC**:

- TIP is **undecidable**

- TCP/TSP: simultaneously.

The type checking algorithm is close to the one for λP . (In λP we had a judgement of **correct** context; this form of judgement could also be introduced for CC)

Variations

Variations on the rules of CC:

- There are many type system with (slightly) different rules
- Many (proofs of) properties are similar
- **Plan**: Study these type systems in one general framework:
 - **Pure Type Systems** (Terlouw, Berardi)

The **Extended Calculus of Constructions** has in addition

- **Cumulativity**: $* \subseteq \square_0 \subseteq \square_1 \subseteq \dots$, so

$$\frac{\Gamma \vdash A : *}{\Gamma \vdash A : \square_0} \quad \frac{\Gamma \vdash A : \square_i}{\Gamma \vdash A : \square_{i+1}}$$

- **Σ -types**:

$$\frac{\Gamma \vdash A : * \quad \Gamma, x:A \vdash B : *}{\Gamma \vdash \Sigma x:A. B : *} \quad \frac{\Gamma \vdash A : \square_i \quad \Gamma, x:A \vdash B : \square_j}{\Gamma \vdash \Sigma x:A. B : \square_{\max(i,j)}}$$

For $\varphi : *$

- We have $\Pi A:\square_i. \varphi : *$, but
- We do **not** have $\Sigma A:\square_i. \varphi : *$.

Note: The type theory of Coq has in addition $\text{Set} : \square$ and rules (Set, Set) , (\square_i, Set) , $(\text{Set}, *)$.

Coq examples using dependent types

natlist

```
Inductive natlist : Set :=  
  nil : natlist  
| cons : nat -> natlist -> natlist.
```

append & reverse

```
Fixpoint append (k l : natlist) {struct k} : natlist :=
  match k with
  | nil => l
  | cons h t => cons h (append t l)
  end.
```

```
Fixpoint reverse (k : natlist) : natlist :=
  match k with
  | nil => nil
  | cons h t => append (reverse t) (cons h nil)
  end.
```

dependent lists

we will define a type for **lists of a given length**

```
a : (natlist_dep 6)
```

this corresponds in normal programming languages to something like

```
int a[6]
```

the type of **a** is called a **dependent** type
it **depends** on the natural number 6

```
natlist_dep : nat -> Set  
natlist_dep 6 : Set
```

natlist_dep

```
Inductive natlist_dep : nat -> Set :=
  nil : natlist_dep 0
| cons : forall n : nat,
  nat -> natlist_dep n -> natlist_dep (S n).
```

3, 1, 4, 1, 5, 9

↓

```
cons 5 3 (cons 4 1 (cons 3 4 (cons 2 1 (cons 1 5 (cons 0 9 nil))))))
                                          : (natlist_dep 6)
```

zeroes for dependent lists

```
Fixpoint zeroes (n : nat) : natlist_dep n :=
  match n return natlist_dep n with
  | 0 => nil
  | S n' => cons n' 0 (zeroes n')
end.
```

the type of dependent zeroes

~~zeroes : nat -> natlist_dep ?~~

zeroes : forall n : nat, natlist_dep n

dependent product

generalizes the notion of function type

append for dependent lists

```
Fixpoint append (n m : nat)
  (k : natlist_dep n) (l : natlist_dep m) {struct k} :
  natlist_dep (plus n m) :=
match k
  in natlist_dep n return natlist_dep (plus n m)
with
  nil => l
| cons n' h t => cons (plus n' m) h (append n' m t l)
end.
```

programming with dependent types: reverse for dependent lists

Fixpoint reverse

```
(n : nat) (k : natlist_dep n) {struct k} :  
  natlist_dep n :=  
match k in natlist_dep n return natlist_dep n with  
  nil => nil  
| cons n' h t =>  
    eq_rec (plus n' 1) (fun n => natlist_dep n)  
      (append n' 1 (reverse n' t) (cons 0 h nil))  
      (S n') (plus_one n')  
end.
```

has type `natlist_dep (plus n' 1)`

but should have type `natlist_dep (S n')`

dependently typed functional programming languages

- **cayenne**

‘dependent haskell’

Lennart Augustsson

- **dependent ML**

Hongwei Xi

- **epigram**

Conor McBride