

Introduction to Type Theory  
February 2008  
Alpha Lernet Summer School  
Piriapolis, Uruguay

Herman Geuvers  
Nijmegen & Eindhoven, NL

Lecture 3: Polymorphic Type Theory: Full polymorphism and ML style  
polymorphism

## Why Polymorphic $\lambda$ -calculus?

---

- Simple type theory  $\lambda \rightarrow$  is not very expressive
- In simple type theory, we can not 'reuse' a function.  
E.g.  $\lambda x:\alpha.x : \alpha \rightarrow \alpha$  and  $\lambda x:\beta.x : \beta \rightarrow \beta$ .

We want to define functions that can treat types **polymorphically**: add types  $\forall \alpha.\sigma$ :

Examples

- $\forall \alpha.\alpha \rightarrow \alpha$   
If  $M : \forall \alpha.\alpha \rightarrow \alpha$ , then  $M$  can map any type to itself.
- $\forall \alpha.\forall \beta.\alpha \rightarrow \beta \rightarrow \alpha$   
If  $M : \forall \alpha.\forall \beta.\alpha \rightarrow \beta \rightarrow \alpha$ , then  $M$  can take two inputs (of arbitrary types) and return a value of the first input type.

## Derivation rules for Weak (ML-style) polymorphism

---

Typ : add  $\forall\alpha_1\dots\forall\alpha_n.\sigma$  for  $\sigma$  a  $\lambda\rightarrow$ -type.

1. Curry style:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall\alpha.\sigma} \quad \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]} \quad \text{for } \tau \text{ a } \lambda\rightarrow\text{-type}$$

2. Church style:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.\sigma} \quad \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]} \quad \text{for } \tau \text{ a } \lambda\rightarrow\text{-type}$$

- $\forall$  only occurs on the outside and is therefore usually left out: “all type variables are **implicitly universally quantified**”
- With weak polymorphism, type checking is still **decidable**: the **principal types algorithm** still works.

## Derivation rules for Weak (ML-style) polymorphism

---

NB! Also the **abstraction rule** is restricted to  $\lambda\rightarrow$ -types:

1. Curry style: 
$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \rightarrow \sigma} \quad \tau \text{ a } \lambda\rightarrow\text{-type}$$

2. Church style: 
$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x:\tau.M : \tau \rightarrow \sigma} \quad \tau \text{ a } \lambda\rightarrow\text{-type}$$

## Examples

---

- $\lambda 2$  à la Curry:  $\lambda x. \lambda y. x : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha$ .
- $\lambda 2$  à la Church:  $\lambda \alpha. \lambda \beta. \lambda x: \alpha. \lambda y: \beta. x : \forall \alpha. \forall \beta. \alpha \rightarrow \beta \rightarrow \alpha$ .
- $\lambda 2$  à la Curry:  $z : \forall \alpha. \alpha \rightarrow \alpha \vdash z z : \forall \alpha. \alpha \rightarrow \alpha$ .
- $\lambda 2$  à la Church:  $z : \forall \alpha. \alpha \rightarrow \alpha \vdash \lambda \alpha. z (\alpha \rightarrow \alpha) (z \alpha) : \forall \alpha. \alpha \rightarrow \alpha$ .
- But NOT  $\vdash \lambda z. z z : \dots$

## Derivation rules of $\lambda 2$ with full (system F-style) polymorphism

---

$\text{Typ} := \text{TVar} \mid (\text{Typ} \rightarrow \text{Typ}) \mid \forall \alpha. \text{Typ}$

1. Curry style:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha. \sigma} \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha. \sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]} \text{ for } \tau \text{ any } \lambda 2\text{-type}$$

2. Church style:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda \alpha. M : \forall \alpha. \sigma} \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha. \sigma}{\Gamma \vdash M \tau : \sigma[\alpha := \tau]} \text{ for } \tau \text{ any } \lambda 2\text{-type}$$

- $\forall$  can also occur deeper in a type.
- With full polymorphism, type checking becomes undecidable! [Wells 1993]

## Derivation rules of $\lambda 2$ with full (system F-style) polymorphism

---

$$\text{Typ} := \text{TVar} \mid (\text{Typ} \rightarrow \text{Typ}) \mid \forall \alpha. \text{Typ}$$

NB: In the **abstraction rule** all types are  **$\lambda 2$ -types**:

1. Curry style: 
$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x. M : \tau \rightarrow \sigma} \quad \sigma, \tau \text{ } \lambda 2\text{-types}$$

2. Church style: 
$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau. M : \tau \rightarrow \sigma} \quad \sigma, \tau \text{ } \lambda 2\text{-types}$$

## Erasure from $\lambda 2$ à la Church to $\lambda 2$ à la Curry

---

$$\begin{array}{lcl} |x| & := & x \\ |\lambda x:\sigma.M| & := & |\lambda x.M| \quad |\lambda \alpha.M| := |M| \\ |MN| & := & |M| |N| \quad |M\sigma| := |M| \end{array}$$

**Theorem** If  $\Gamma \vdash M : \sigma$  in  $\lambda 2$  à la Church, then  $\Gamma \vdash |M| : \sigma$  in  $\lambda 2$  à la Curry.

**Theorem** If  $\Gamma \vdash P : \sigma$  in  $\lambda 2$  à la Curry, then there is an  $M$  such that  $|M| \equiv P$  and  $\Gamma \vdash M : \sigma$  in  $\lambda 2$  à la Church.



## Derivation rules of $\lambda 2$ with full (system F-style) polymorphism

---

$\text{Typ} := \text{TVar} \mid (\text{Typ} \rightarrow \text{Typ}) \mid \forall \alpha. \text{Typ}$

1. Curry style:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha. \sigma} \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha. \sigma}{\Gamma \vdash M : \sigma[\alpha := \tau]} \text{ for } \tau \text{ any } \lambda 2\text{-type}$$

2. Church style:

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda \alpha. M : \forall \alpha. \sigma} \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha. \sigma}{\Gamma \vdash M \tau : \sigma[\alpha := \tau]} \text{ for } \tau \text{ any } \lambda 2\text{-type}$$

Examples valid only with full polymorphism:

- $\lambda 2$  à la Curry:  $\lambda x. \lambda y. x : (\forall \alpha. \alpha) \rightarrow \sigma \rightarrow \tau$ .
- $\lambda 2$  à la Church:  $\lambda x. (\forall \alpha. \alpha). \lambda y. \sigma. x \tau : (\forall \alpha. \alpha) \rightarrow \sigma \rightarrow \tau$ .

## Let polymorphism in ML

---

To regain some of the “full polymorphism”, ML has **let polymorphism**

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \mathbf{let} \ x = M \ \mathbf{in} \ N : \tau} \text{ for } \tau \text{ a } \lambda\text{-type}, \sigma \text{ a } \lambda^2\text{-type}$$

This allows the formation of a  $\beta$ -redex

$$(\lambda x:\sigma.N)M$$

for  $\sigma$  a polymorphic type.

But **not**  $\lambda x:\sigma.N : \sigma \rightarrow \tau$

## Recall: Important Properties

---

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

$\Gamma \vdash M : ?$  TSP

$\vdash ? : \sigma$  TIP

## Properties of polymorphic $\lambda$ -calculus

- TIP is **undecidable**, TCP and TSP are equivalent & decidable.

TCP	à la Church	à la Curry
• ML-style	decidable	<b>decidable</b>
System F-style	<b>decidable</b>	<b>undecidable</b>

With **full polymorphism** (system F), **untyped terms** contain **too little information** to compute the type.

## Some examples of typing in $\lambda 2$

---

Abbreviate  $\perp := \forall\alpha.\alpha$ ,  $\top := \forall\alpha.\alpha \rightarrow \alpha$ .

- Curry  $\lambda 2$ :  $\lambda x.xx : \perp \rightarrow \perp$
- Church  $\lambda 2$ :  $\lambda x:\perp.x(\perp \rightarrow \perp)x : \perp \rightarrow \perp$ .
- Church  $\lambda 2$ :  $\lambda x:\perp.\lambda\alpha.x(\alpha \rightarrow \alpha)(x\alpha) : \perp \rightarrow \perp$ .

Exercises:

- Verify that in Church  $\lambda 2$ :  $\lambda x:\top.x\top x : \top \rightarrow \top$ .
- Verify that in Curry  $\lambda 2$ :  $\lambda x.xx : \top \rightarrow \top$
- Find a type in Curry  $\lambda 2$  for  $\lambda x.x x x$
- Find a type in Curry  $\lambda 2$  for  $\lambda x.(x x)(x x)$

## Formulas-as-types for $\lambda 2$

---

There is a **formulas-as-types** isomorphism between  $\lambda 2$  and **second order proposition logic**, PROP2

**Derivation rules** of PROP2:

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \forall \alpha. \sigma} \quad \alpha \notin \text{FV}(\Gamma) \qquad \frac{\Gamma \vdash \forall \alpha. \sigma}{\Gamma \vdash \sigma[\alpha := \tau]}$$

**NB** This is **constructive** second order proposition logic:

$$\forall \alpha. \forall \beta. ((\alpha \rightarrow \beta) \rightarrow \alpha) \rightarrow \alpha \text{ Peirce's law}$$

is **not derivable**.

## Definability of the other connectives

---

$$\begin{aligned}\perp & := \forall\alpha.\alpha \\ \sigma \wedge \tau & := \forall\alpha.(\sigma \rightarrow \tau \rightarrow \alpha) \rightarrow \alpha \\ \sigma \vee \tau & := \forall\alpha.(\sigma \rightarrow \alpha) \rightarrow (\tau \rightarrow \alpha) \rightarrow \alpha \\ \exists\alpha.\sigma & := \forall\beta.(\forall\alpha.\sigma \rightarrow \beta) \rightarrow \beta\end{aligned}$$

and all the standard constructive derivation rules are derivable.

Example ( $\wedge$ -elimination):

$$\frac{\frac{\forall\alpha.(\sigma \rightarrow \tau \rightarrow \alpha) \rightarrow \alpha}{(\sigma \rightarrow \tau \rightarrow \sigma) \rightarrow \sigma} \quad \frac{\frac{[\sigma]^1}{\tau \rightarrow \sigma}}{\sigma \rightarrow \tau \rightarrow \sigma} 1}{\sigma}}$$

## Definability of connectives and derivation rules

---

$$\begin{aligned} \perp & := \forall \alpha. \alpha \\ \sigma \wedge \tau & := \forall \alpha. (\sigma \rightarrow \tau \rightarrow \alpha) \rightarrow \alpha \\ \sigma \vee \tau & := \forall \alpha. (\sigma \rightarrow \alpha) \rightarrow (\tau \rightarrow \alpha) \rightarrow \alpha \\ \exists \alpha. \sigma & := \forall \beta. (\forall \alpha. \sigma \rightarrow \beta) \rightarrow \beta \end{aligned}$$

Example ( $\wedge$ -elimination) with  $\lambda$ -terms:

$$\frac{\frac{M : \forall \alpha. (\sigma \rightarrow \tau \rightarrow \alpha) \rightarrow \alpha}{M\sigma : (\sigma \rightarrow \tau \rightarrow \sigma) \rightarrow \sigma} \quad \frac{\frac{[x : \sigma]^1}{\lambda y : \tau. x : \tau \rightarrow \sigma}}{\lambda x : \sigma. \lambda y : \tau. x : \sigma \rightarrow \tau \rightarrow \sigma} 1}{M\sigma(\lambda x : \sigma. \lambda y : \tau. x) : \sigma}$$

So the following term is a ‘witness’ for the  $\wedge$ -elimination.

$$\lambda z : \sigma \wedge \tau. z \sigma (\lambda x : \sigma. \lambda y : \tau. x) : (\sigma \wedge \tau) \rightarrow \sigma$$

## Data types in $\lambda 2$

---

$$\text{Nat} := \forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$$

This type uses the encoding of **natural numbers** as **Church numerals**

$$n \mapsto c_n := \lambda x. \lambda f. f(\dots(fx)) \quad n\text{-times } f$$

- **0** :=  $\lambda \alpha. \lambda x:\alpha. \lambda f:\alpha \rightarrow \alpha. x$
- **S** :=  $\lambda n:\text{Nat}. \lambda \alpha. \lambda x:\alpha. \lambda f:\alpha \rightarrow \alpha. f(n \alpha x f)$
- **Iteration**: if  $c : \sigma$  and  $g : \sigma \rightarrow \sigma$ , then **It c g** :  $\text{Nat} \rightarrow \sigma$  is defined as

$$\lambda n:\text{Nat}. n \sigma c g$$

Then **It c g n** =  $g(\dots(g c))$  ( $n$  times  $g$ ), i.e.

$$\text{It } c g 0 = c \quad \text{and} \quad \text{It } c g (S x) = g(\text{It } c g x)$$



## Why is this a good/useful type for the natural numbers?

---

- It's the straightforward type for the **Church numerals**.
- It represents the **type of proofs that a number is inductive** in second order predicate logic:

$$0 : D, S : D \rightarrow D$$

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

$N(x)$  iff  $x$  is in the smallest 'set' containing 0 and closed under  $S$ .

E.g.  $N(0), (N(S 0), \dots, N(S^p(0)))$ .

Stripping all **first order** information (moving from PRED2 to PROP2):

$$N := \forall P. P \rightarrow (P \rightarrow P) \rightarrow P$$

The normal proof of  $N(S^p(0))$  is the Church numeral  $c_n$  under a suitable Curry-Howard embedding.

## Examples

---

- Addition

$$\mathbf{Plus} := \lambda n:\mathbf{Nat}.\lambda m:\mathbf{Nat}.\mathbf{It} \ m \ S \ n$$

or  $\mathbf{Plus} := \lambda n:\mathbf{Nat}.\lambda m:\mathbf{Nat}.n \ \mathbf{Nat} \ m \ S$

- Multiplication

$$\mathbf{Mult} := \lambda n:\mathbf{Nat}.\lambda m:\mathbf{Nat}.\mathbf{It} \ 0 \ (\lambda x:\mathbf{Nat}.\mathbf{Plus} \ m \ x) \ n$$

- Predecessor is **difficult!**

This requires defining **primitive recursion** in terms of **iteration**.

As a consequence:

$$\mathbf{Pred}(n + 1) \rightarrow_{\beta} n$$

in a number of steps of  $O(n)$ .

## Data types in $\lambda 2$ ctd.

---

$$\text{List}_A := \forall \alpha. \alpha \rightarrow (A \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$$

the type of **lists over  $A$** , using the following encoding

$$[a_1, a_2, \dots, a_n] \mapsto \lambda x. \lambda f. f a_1 (f a_2 (\dots (f a_n x))) \quad n\text{-times } f$$

- **Nil** :=  $\lambda \alpha. \lambda x: \alpha. \lambda f: A \rightarrow \alpha \rightarrow \alpha. x$
- **Cons** :=  $\lambda a: A. \lambda l: \text{List}_A. \lambda \alpha. \lambda x: \alpha. \lambda f: A \rightarrow \alpha \rightarrow \alpha. f a (l \alpha x f)$
- **Iteration**: if  $c : \sigma$  and  $g : A \rightarrow \sigma \rightarrow \sigma$ , then **It  $c g$**  :  $\text{List}_A \rightarrow \sigma$  is def. as

$$\lambda l: \text{List}_A. l \sigma c g$$

Then, for  $l = [a_1, \dots, a_n]$ , **It  $c g l$**  =  $g a_1 (\dots (g a_n c))$  ( $n$  times  $g$ ) i.e.

$$\text{It } c g \text{ Nil} = c \quad \text{and} \quad \text{It } c g (\text{Cons } a l) = g a (\text{It } c g l)$$

## Example

---

- Map, given  $f : \sigma \rightarrow \tau$ ,  $\text{Map } f : \text{List}_\sigma \rightarrow \text{List}_\tau$  applies  $f$  to all elements in a list.

$$\text{Map} := \lambda f : \sigma \rightarrow \tau. \text{It Nil}(\lambda x : \sigma. \lambda l : \text{List}_\tau. \text{Cons}(f x) l).$$

Then

$$\begin{aligned} \text{Map } f \text{ Nil} &= \text{Nil} \\ \text{Map } f (\text{Cons } a k) &= \text{It Nil}(\lambda x : \sigma. \lambda l : \text{List}_\tau. \text{Cons}(f x) l) (\text{Cons } a k) \\ &= (\lambda x : \sigma. \lambda l : \text{List}_\tau. \text{Cons}(f x) l) a (\text{Map } f k) \\ &= \text{Cons}(f a) (\text{Map } f k) \end{aligned}$$

## Many data-types can be defined in $\lambda 2$

---

- **Product** of two data-types:  $\sigma \times \tau := \forall \alpha. (\sigma \rightarrow \tau \rightarrow \alpha) \rightarrow \alpha$
- **Sum** of two data-types:  $\sigma + \tau := \forall \alpha. (\sigma \rightarrow \alpha) \rightarrow (\tau \rightarrow \alpha) \rightarrow \alpha$
- **Unit type**:  $\text{Unit} := \forall \alpha. \alpha \rightarrow \alpha$
- **Binary trees** with nodes in  $A$  and leaves in  $B$ :  
 $\text{Tree}_{A,B} := \forall \alpha. (B \rightarrow \alpha) \rightarrow (A \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$

### Exercise:

- Define  $\text{inl} : \sigma \rightarrow \sigma + \tau$
- Define the first projection:  $\pi_1 : \sigma \times \tau \rightarrow \sigma$
- Define  $\text{join} : \text{Tree}_{A,B} \rightarrow \text{Tree}_{A,B} \rightarrow A \rightarrow \text{Tree}_{A,B}$

## Properties of $\lambda_2$

---

- For  $\lambda_2$  à la Church: **Uniqueness of types**  
If  $\Gamma \vdash M : \sigma$  and  $\Gamma \vdash M : \tau$ , then  $\sigma = \tau$ .
- **Subject Reduction**  
If  $\Gamma \vdash M : \sigma$  and  $M \longrightarrow_{\beta\eta} N$ , then  $\Gamma \vdash N : \sigma$ .
- **Strong Normalization**  
If  $\Gamma \vdash M : \sigma$ , then all  $\beta\eta$ -reductions from  $M$  terminate.

## Strong Normalization of $\beta$ for $\lambda 2$

---

- There are two kinds of  $\beta$ -reductions
  - $(\lambda x:\sigma.M)P \longrightarrow_{\beta} M[x := P]$
  - $(\lambda\alpha.M)\tau \longrightarrow_{\beta} M[\alpha := \tau]$
- The second does no harm, so we can just look at  $\lambda 2$  à la Curry

Recall the proof for  $\lambda \rightarrow$ :

- $\llbracket \alpha \rrbracket := \text{SN}$ .
- $\llbracket \sigma \rightarrow \tau \rrbracket := \{M \mid \forall N \in \llbracket \sigma \rrbracket (MN \in \llbracket \tau \rrbracket)\}$ .

Question:

How to define  $\llbracket \forall\alpha.\sigma \rrbracket$  ??

$$\llbracket \forall\alpha.\sigma \rrbracket := \prod_{X \in U} \llbracket \sigma \rrbracket_{\alpha := X} ??$$

## Strong Normalization of $\beta$ for $\lambda_2$

---

Question:

How to define  $\llbracket \forall \alpha. \sigma \rrbracket$  ??

$$\llbracket \forall \alpha. \sigma \rrbracket := \prod_{X \in U} \llbracket \sigma \rrbracket_{\alpha := X} ??$$

- What should be  $U$ ?

The collection of “all possible interpretations” of types (?)

- $\prod_{X \in U} \llbracket \sigma \rrbracket_{\alpha := X}$  gets too big:  $\text{card}(\prod_{X \in U} \llbracket \sigma \rrbracket_{\alpha := X}) > \text{card}(U)$
- Girard:  $\llbracket \forall \alpha. \sigma \rrbracket$  should be small

$$\bigcap_{X \in U} \llbracket \sigma \rrbracket_{\alpha := X}$$

- Girard: Definition of  $U$ .



## Strong Normalization of $\beta$ for $\lambda 2$

---

$U := \text{SAT}$ , the collection of **saturated sets** of (untyped)  $\lambda$ -terms.

$X \subseteq \Lambda$  is **saturated** if

- $xP_1 \dots P_n \in X$  (for all  $x \in \text{Var}$ ,  $P_1, \dots, P_n \in \text{SN}$ )
- $X \subseteq \text{SN}$
- If  $M[x := N]\vec{P} \in X$  and  $N \in \text{SN}$ , then  $(\lambda x.M)N\vec{P} \in X$ .

Let  $\rho : \text{TVar} \rightarrow \text{SAT}$  be a **valuation** of type variables. Define  $[[\sigma]]_\rho$  by:

- $[[\alpha]]_\rho := \rho(\alpha)$
- $[[\sigma \rightarrow \tau]]_\rho := \{M \mid \forall N \in [[\sigma]]_\rho (MN \in [[\tau]]_\rho)\}$
- $[[\forall \alpha. \sigma]]_\rho := \bigcap_{X \in \text{SAT}} [[\sigma]]_{\rho, \alpha := X}$

## Proposition

---

$$x_1 : \tau_1, \dots, x_n : \tau_n \vdash M : \sigma \Rightarrow M[P_1/x_1, \dots, P_n/x_n] \in \llbracket \sigma \rrbracket_\rho$$

for all valuations  $\rho$  and  $P_1 \in \llbracket \tau_1 \rrbracket_\rho, \dots, P_n \in \llbracket \tau_n \rrbracket_\rho$

## Proof

By induction on the derivation of  $\Gamma \vdash M : \sigma$ .

## Corollary $\lambda 2$ is SN

(Proof: take  $P_1$  to be  $x_1, \dots, P_n$  to be  $x_n$ .)

## A little bit on semantics

---

$\lambda 2$  does **not have a set-theoretic model!** [Reynolds]

**Theorem:** If

$$\llbracket \sigma \rightarrow \tau \rrbracket := \llbracket \tau \rrbracket^{\llbracket \sigma \rrbracket} \text{ ( set theoretic function space )}$$

then  $\llbracket \sigma \rrbracket$  is a singleton set for every  $\sigma$ .

So: in a  $\lambda 2$ -model,  $\llbracket \sigma \rightarrow \tau \rrbracket$  must be 'small'.