

Proving with Computer Assistance

Lecture Polymorphic Type Theory: Full polymorphism and ML style polymorphism

Herman Geuvers

Overview

- ▶ Weak (ML style) polymorphism, à la Church and à la Curry.
- ▶ Full (system F) polymorphism, à la Church and à la Curry.
- ▶ Data types and functions defined by iteration in full polymorphic λ -calculus
- ▶ Properties of the systems TCP, TSP, TIP, Subject Reduction, Uniqueness of Types, ...

Why Polymorphic λ -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 σ a $\lambda\rightarrow$ -type.

1. Curry style:

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

2. Church style:

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

- ▶ \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:

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \rightarrow \sigma} \text{ for } \sigma, \tau \lambda\rightarrow\text{-types}$$

2. Church style:

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x:\tau.M : \tau \rightarrow \sigma} \text{ for } \sigma, \tau \lambda\rightarrow\text{-types}$$

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 $\vdash \lambda z. z z : \dots$

Examples with flag style derivations

$\lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha\rightarrow\beta\rightarrow\alpha.$

More examples with flag style derivations

Self application?

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

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

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:

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

2. Church style:

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

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: $\lambda x : (\forall \alpha. \alpha). \lambda y : \sigma. x \tau : (\forall \alpha. \alpha) \rightarrow \sigma \rightarrow \tau$.

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 $\lambda x.(x x)(x x)$

Examples with flag style derivations

$\lambda x.\lambda y.x : \perp \rightarrow \sigma \rightarrow \tau.$

More examples with flag style derivations

$\lambda x.x x : \perp \rightarrow \perp$.

Let polymorphism in ML

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

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

This allows the formation of a β -redex

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

for σ 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 λ -calculus

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

TCP	à la Church	à la Curry
▶ ML-style	decidable	decidable
System F-style	decidable	undecidable

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

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**

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

- ▶ **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

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

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

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

Examples

- ▶ Addition

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

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

- ▶ Multiplication

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

- ▶ Predecessor is **difficult!**

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

As a consequence:

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

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

Iteration, the more general picture

Nat is a data-type with two constructors $0 : \text{Nat}$ and $S : \text{Nat} \rightarrow \text{Nat}$. This implies the **Nat-iteration scheme** for defining functions $f : \text{Nat} \rightarrow D$ (for any type D).

LEMMA[**Nat-iteration**] If $d : D$ and $g : D \rightarrow D$, then there is a function $f : \text{Nat} \rightarrow D$ satisfying

$$\begin{aligned}f 0 &= d \\ f (S x) &= g (f x)\end{aligned}$$

In $\lambda 2$, this f , also called **It $d g$** , can be defined as $\lambda n:\text{Nat}.n D d g$.

The other way around: if I want to have a function $h : \text{Nat} \rightarrow D$ that I can **specify by case distinction**, satisfying these equations:

$$\begin{aligned}h 0 &= d \\ h (S x) &= g (h x)\end{aligned}$$

Then I have it, because I can take $h := \lambda n:\text{Nat}.n D d g$.

Examples of Nat-iteration

Nat-ITERATION in $\lambda 2$: if $d : D$ and $g : D \rightarrow D$, then $f := \lambda n:\text{Nat}.n D d g$ satisfies

$$\begin{aligned}f 0 &= d \\f (S x) &= g (f x)\end{aligned}$$

We derive Plus and Mult using Nat-iteration.

$$\begin{aligned}\text{Plus } m 0 &= m \\ \text{Plus } m (S x) &= S (\text{Plus } m x)\end{aligned}$$

So we can take $\text{Plus } m := \lambda n:\text{Nat}.n \text{Nat } m S$ and we define **Plus** $:= \lambda m, n:\text{Nat}.n \text{Nat } m S$

$$\begin{aligned}\text{Mult } m 0 &= 0 \\ \text{Mult } m (S x) &= \text{Plus } m (\text{Mult } m x)\end{aligned}$$

So we define **Mult** $:= \lambda m, n:\text{Nat}.n \text{Nat } 0 (\lambda y:\text{Nat}.\text{Plus } m y)$

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

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

We have **It $c g$** $[a_1, \dots, a_n] = g a_1 (\dots (g a_n c))$ (n times g), that is:

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

The List-iteration scheme

List_A has constructors $\text{Nil} : \text{List}_A$ and $\text{Cons} : A \rightarrow \text{List}_A \rightarrow \text{List}_A$.

This implies a **List-iteration scheme** for defining functions

$f : \text{List}_A \rightarrow D$ (for any type D).

LEMMA[List-iteration] If $d : D$ and $g : A \rightarrow D \rightarrow D$, then there is a function $f : \text{List}_A \rightarrow D$ satisfying

$$\begin{aligned}f \text{ Nil} &= d \\f (\text{Cons } a \ x) &= g \ a \ (f \ x)\end{aligned}$$

In $\lambda 2$, this f can be defined as $\lambda \ell : \text{List}_A . \ell \ D \ d \ g$ (which is also written as $\text{It } d \ g$).

Example: the length of a list, $\text{len} : \text{List}_A \rightarrow \text{Nat}$. It satisfies

$$\begin{aligned}\text{len Nil} &= 0 \\ \text{len} (\text{Cons } a \ k) &= \text{S} (\text{len } k)\end{aligned}$$

So we can define $\text{len} := \text{It } 0 \ (\lambda a : A . \lambda n : \text{Nat} . \text{S } n)$, or in full detail $\text{len} := \lambda \ell : \text{List}_A . \ell \ \text{Nat } 0 \ (\lambda a : A . \lambda n : \text{Nat} . \text{S } n)$.

Example: Map

Map is one of the standard functional programs over List. Given $h : \sigma \rightarrow \tau$, we have

$$\text{Map } h : \text{List}_\sigma \rightarrow \text{List}_\tau$$

which applies h to all elements in the input-list.

We can specify it via these equations:

$$\text{Map } h \text{ Nil} = \text{Nil}$$

$$\text{Map } h (\text{Cons } a \ k) = \text{Cons } (h \ a) (\text{Map } h \ k)$$

So we can define it using List-iteration:

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

Or in full detail

$$\text{Map} := \lambda h : \sigma \rightarrow \tau. \lambda \ell : \text{List}_\sigma. \ell \ \text{List}_\tau \ \text{Nil} (\lambda x : \sigma. \lambda k : \text{List}_\tau. \text{Cons } (h \ x) \ k).$$

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**: **Unit** := $\forall \alpha. \alpha \rightarrow \alpha$
- ▶ **Binary trees** with **nodes in A** and **leaves in B** :
 $\text{Tree}_{A,B} := \forall \alpha. (B \rightarrow \alpha) \rightarrow (A \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$
 $\text{Tree}_{A,B}$ has two constructors, $\text{leaf} : B \rightarrow \text{Tree}_{A,B}$ and
 $\text{join} : \text{Tree}_{A,B} \rightarrow \text{Tree}_{A,B} \rightarrow A \rightarrow \text{Tree}_{A,B}$

Exercise:

- ▶ Define $\text{inl} : \sigma \rightarrow \sigma + \tau$
- ▶ Define the first projection: $\pi_1 : \sigma \times \tau \rightarrow \sigma$
- ▶ Define $\text{leaf} : B \rightarrow \text{Tree}_{A,B}$ and
 $\text{join} : \text{Tree}_{A,B} \rightarrow \text{Tree}_{A,B} \rightarrow A \rightarrow \text{Tree}_{A,B}$
- ▶ Give the Tree-iteration scheme for $\text{Tree}_{A,B}$ and define
 $h : \text{Tree}_{A,B} \rightarrow \text{Nat}$ that counts the number of leaves of a tree.

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.