Lambda-Calculus and Type Theory

ISR 2024
Obergurgl, Austria

Herman Geuvers & Niels van der Weide

Radboud University Nijmegen, The Netherlands

Lecture 6

Polymorphic Types

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 $\forall\alpha_1.\ldots.\forall\alpha_n.\sigma$ for σ a $\lambda{\to}$-type.

1. Curry style:

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

2. Church style:

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

- $\forall$ 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 $\lambda{\to}$-types:

1. Curry style:

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

2. Church style:

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

Examples

- $\lambda 2$ à la Curry: $\lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha.$
- $\lambda 2$ à la Church: $\lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha\to\beta\to\alpha.$
- $\lambda 2$ à la Curry: $z : \forall\alpha.\alpha\to\alpha \vdash z\,z : \forall\alpha.\alpha\to\alpha.$
- $\lambda 2$ à la Church: $z : \forall\alpha.\alpha\to\alpha \vdash \lambda\alpha.z\,(\alpha\to\alpha)\,(z\,\alpha) : \forall\alpha.\alpha\to\alpha.$
- But NOT $\vdash \lambda z.z\,z : \ldots$

Examples with flag style derivations

$\lambda x.\lambda y.x : \forall\alpha.\forall\beta.\alpha{\to}\beta{\to}\alpha.$

More examples with flag style derivations

Self application?

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

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ} \to \mathsf{Typ}) \mid \forall \alpha.\mathsf{Typ}$$

1. Curry style:

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

2. Church style:

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

- $\forall$ 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

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ} \to \mathsf{Typ}) \mid \forall \alpha.\mathsf{Typ}$$

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

1. Curry style: $\dfrac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \tau \to \sigma}\ \sigma, \tau\ \ \lambda 2\text{-types}$
2. Church style: $\dfrac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x{:}\tau.M : \tau \to \sigma}\ \sigma, \tau\ \ \lambda 2\text{-types}$

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

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

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

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ} \to \mathsf{Typ}) \mid \forall \alpha.\mathsf{Typ}$$

1. Curry style:

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

2. Church style:

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

Examples valid only with full polymorphism:

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

Some examples of typing in $\lambda 2$

Abbreviate $\bot := \forall\alpha.\alpha$, $\top := \forall\alpha.\alpha{\to}\alpha$.

- Curry $\lambda 2$: $\lambda x.xx : \bot{\to}\bot$
- Church $\lambda 2$: $\lambda x{:}\bot.x(\bot{\to}\bot)x : \bot{\to}\bot$.
- Church $\lambda 2$: $\lambda x{:}\bot.\lambda\alpha.x(\alpha{\to}\alpha)(x\alpha) : \bot{\to}\bot$.

Exercises:

- Verify that in Church $\lambda 2$: $\lambda x{:}\top.x\top x : \top{\to}\top$.
- Verify that in Curry $\lambda 2$: $\lambda x.xx : \top{\to}\top$
- 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

$\lambda x.\lambda y.x : \bot \to \sigma \to \tau.$

More examples with flag style derivations

$\lambda x.x\,x : \bot \to \bot$.

Let polymorphism in ML

To regain some of the "full polymorphism", ML has let polymorphism

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

This allows the formation of a β-redex

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

for σ a polymorphic type.
But not $\lambda x{:}\sigma.N : \sigma{\to}\tau$

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 $\lambda 2$

$$\mathrm{Nat} := \forall\alpha.\alpha{\to}(\alpha{\to}\alpha){\to}\alpha$$

This type uses the encoding of natural numbers as Church numerals

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

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

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

  We have $\mathrm{It}\,c\,g\,n = g(\ldots(g\,c))$ (n times g), i.e.

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

Examples

- Addition

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

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

- Multiplication

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

- Predecessor is difficult!
  This requires defining primitive recursion in terms of iteration.
  As a consequence:

$$\mathrm{Pred}(n+1) \twoheadrightarrow_\beta n$$

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

Iteration, the more general picture

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

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

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

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

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

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

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

Examples of Nat-iteration

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

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

We derive Plus and Mult using Nat-iteration.

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

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

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

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

Data types in $\lambda 2$ ctd.

$$\mathrm{List}_A := \forall\alpha.\alpha{\to}(A{\to}\alpha{\to}\alpha){\to}\alpha$$

The type of lists over A uses the following encoding:

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

- $\mathrm{Nil} := \lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha.x$
- $\mathrm{Cons} := \lambda a{:}A.\lambda\ell{:}\mathrm{List}_A.\lambda\alpha.\lambda x{:}\alpha.\lambda f{:}A{\to}\alpha{\to}\alpha.f\ a\,(\ell\,\alpha\,x\,f)$
- Iteration: if $c : \sigma$ and $g : A{\to}\sigma{\to}\sigma$, then $\mathrm{It}\,c\,g : \mathrm{List}_A{\to}\sigma$ is defined as

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

  We have $\mathrm{It}\,c\,g\,[a_1, \ldots, a_n] = g\ a_1\,(\ldots(g\ a_n\,c))$ (n times g), that is:

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

The List-iteration scheme

List_A has constructors $\mathrm{Nil} : \mathrm{List}_A$ and $\mathrm{Cons} : A \to \mathrm{List}_A \to \mathrm{List}_A$.
This implies a List-iteration scheme for defining functions $f : \mathrm{List}_A \to D$ (for any type D).

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

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

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

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

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

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

Example: Map

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

$$\mathrm{Map}\, h : \mathrm{List}_\sigma \to \mathrm{List}_\tau$$

which applies h to all elements in the input-list.
We can specify it via these equations:

$$\begin{array}{rcl} \mathrm{Map}\, h\, \mathrm{Nil} & = & \mathrm{Nil} \\ \mathrm{Map}\, h\, (\mathrm{Cons}\, a\, k) & = & \mathrm{Cons}\, (h\, a)(\mathrm{Map}\, h\, k) \end{array}$$

So we can define it using List-iteration:

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

Or in full detail

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

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

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

Exercise:

- Define $\mathrm{inl} : \sigma \to \sigma + \tau$
- Define the first projection: $\pi_1 : \sigma \times \tau \to \sigma$
- Define $\mathrm{leaf} : B \to \mathrm{Tree}_{A,B}$ and
  $\mathrm{join} : \mathrm{Tree}_{A,B} \to \mathrm{Tree}_{A,B} \to A \to \mathrm{Tree}_{A,B}$
- Give the Tree-iteration scheme for $\mathrm{Tree}_{A,B}$ and define
  $h : \mathrm{Tree}_{A,B} \to \mathrm{Nat}$ 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.