System F

Reference: Chapter 23 in Pierce's *TAPL*

Recall the Simply-Typed Lambda-Calculus (STLC)

- Syntax

$$
\begin{array}{rrcl}
\text{(Terms)} & M & ::= & x \mid \lambda x : \tau.\, M \mid M\, M \\
\text{(Types)} & \tau & ::= & \mathtt{T} \mid \tau \to \tau \\
\text{(Values)} & v & ::= & \lambda x : \tau.\, M \\
\text{(Contexts)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau
\end{array}
$$

- Reduction

$$\frac{}{(\lambda x : \tau.\, M_1)\, M_2 \longrightarrow M_1[M_2/x]}\ \text{(E-AppAbs)}$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\, M_2 \longrightarrow M_1'\, M_2}\ \text{(E-App1)} \qquad \frac{M_2 \longrightarrow M_2'}{M_1\, M_2 \longrightarrow M_1'\, M_2'}\ \text{(E-App2)}$$

$$\frac{M \longrightarrow M'}{\lambda x : \tau.\, M \longrightarrow \lambda x : \tau.\, M'}\ \text{(E-Abs)}$$

Recall the Simply-Typed Lambda-Calculus (STLC)

- Typing

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{T-VAR}) \qquad \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash (\lambda x : \tau_1 .\ M) : \tau_1 \to \tau_2}\ (\text{T-ABS})$$

$$\frac{\Gamma \vdash M_1 : \tau \to \tau' \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1\ M_2 : \tau'}\ (\text{T-APP})$$

- Soundness

Theorem (Preservation)

For all M, M' and τ, if $\bullet \vdash M : \tau$ and $M \longrightarrow M'$, then $\bullet \vdash M' : \tau$.

Theorem (Progress)

For all M and τ, if $\bullet \vdash M : \tau$, then either $M \in$ Values or $\exists M'.\ M \longrightarrow M'$.

Recall the Simply-Typed Lambda-Calculus (STLC)

We can write an infinite number of "doubling" functions in STLC:

$$\begin{aligned}
\mathsf{doubleNat} &\stackrel{\text{def}}{=} \lambda f : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda x : \mathsf{Nat}.\ f\,(f\,x) \\
\mathsf{doubleBool} &\stackrel{\text{def}}{=} \lambda f : \mathsf{Bool} \to \mathsf{Bool}.\ \lambda x : \mathsf{Bool}.\ f\,(f\,x) \\
\mathsf{doubleFun} &\stackrel{\text{def}}{=} \lambda f : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat}).\ \lambda x : \mathsf{Nat} \to \mathsf{Nat}.\ f\,(f\,x)
\end{aligned}$$

Different types of arguments, but the same function body.

Recall the Simply-Typed Lambda-Calculus (STLC)

We can write an infinite number of “doubling” functions in STLC:

$$\begin{aligned}
\mathsf{doubleNat} &\stackrel{\text{def}}{=} \lambda f : \mathsf{Nat} \to \mathsf{Nat}.\ \lambda x : \mathsf{Nat}.\ f\,(f\,x) \\
\mathsf{doubleBool} &\stackrel{\text{def}}{=} \lambda f : \mathsf{Bool} \to \mathsf{Bool}.\ \lambda x : \mathsf{Bool}.\ f\,(f\,x) \\
\mathsf{doubleFun} &\stackrel{\text{def}}{=} \lambda f : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat}).\ \lambda x : \mathsf{Nat} \to \mathsf{Nat}.\ f\,(f\,x)
\end{aligned}$$

Different types of arguments, but the same function body.

Can we abstract out the types?

Polymorphism

poly = many, *morph* = form
Allow a single piece of code to be used with multiple types.

Our focus: *parametric polymorphism*.

- Code is typed “generically”, using variables in place of actual types, and then instantiated with particular types as needed.
- *Uniform*: all of their instances behave the same.
- By contrast, *ad-hoc polymorphism* (e.g. overloading) allows the code to exhibit different behaviors at different types.

System F

System F was first discovered by Jean-Yves Girard (1972), in the context of proof theory in logic.

John Reynolds (1974) independently developed a type system with the same power, called *the polymorphic lambda-calculus*.

It is also sometimes called *the second-order lambda-calculus*, because it corresponds, via the Curry-Howard correspondence, to second-order intuitionistic logic, which allows quantification not only over individuals [terms], but also over predicates [types].

Syntax

$$
\begin{array}{rccl}
\text{(Terms)} & M & ::= & x \mid \lambda x:\tau.\, M \mid M\,M \mid \Lambda\alpha.\, M \mid M\,\langle\tau\rangle \\
\text{(Types)} & \tau & ::= & \alpha \mid \mathtt{T} \mid \tau \to \tau \mid \forall\alpha.\,\tau \\
\text{(Values)} & v & ::= & \lambda x:\tau.\, M \mid \Lambda\alpha.\, M
\end{array}
$$

- Type variable α
- Type abstraction $\Lambda\alpha.\, M$
- Type application $M\,\langle\tau\rangle$
- Universal type $\forall\alpha.\,\tau$

Reduction

$$\frac{}{(\lambda x : \tau.\ M_1)\ M_2 \longrightarrow M_1[M_2/x]}\ \text{(E-AppAbs)}$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\ M_2 \longrightarrow M_1'\ M_2}\ \text{(E-App1)} \qquad \frac{M_2 \longrightarrow M_2'}{M_1\ M_2 \longrightarrow M_1'\ M_2'}\ \text{(E-App2)}$$

$$\frac{M \longrightarrow M'}{\lambda x : \tau.\ M \longrightarrow \lambda x : \tau.\ M'}\ \text{(E-Abs)}$$

$$\frac{}{(\Lambda\alpha.\ M_1)\ \langle\tau_2\rangle \longrightarrow M_1[\tau_2/\alpha]}\ \text{(E-TAppTAbs)}$$

$$\frac{M_1 \longrightarrow M_1'}{M_1\ \langle\tau_2\rangle \longrightarrow M_1'\ \langle\tau_2\rangle}\ \text{(E-TApp)} \qquad \frac{M \longrightarrow M'}{\Lambda\alpha.\ M \longrightarrow \Lambda\alpha.\ M'}\ \text{(E-TAbs)}$$

Statics

$$
\begin{array}{rcll}
\text{(Terms)} & M & ::= & x \mid \lambda x : \tau.\, M \mid M\, M \mid \Lambda\alpha.\, M \mid M\,\langle\tau\rangle \\
\text{(Types)} & \tau & ::= & \alpha \mid \mathtt{T} \mid \tau \to \tau \mid \forall\alpha.\, \tau \\
\text{(Values)} & v & ::= & \lambda x : \tau.\, M \mid \Lambda\alpha.\, M \\
 & & & \\
\text{(Contexts)} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\text{(TypeVarContexts)} & \Delta & ::= & \bullet \mid \Delta, \alpha
\end{array}
$$

Type well-formedness: $\Delta \vdash \tau$

Typing judgment: $\Delta; \Gamma \vdash M : \tau$

Type Well-Formedness

$$\frac{}{\Delta, \alpha \vdash \alpha} \qquad \frac{}{\Delta \vdash \mathtt{T}} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \rightarrow \tau_2} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha.\, \tau}$$

An alternative formulation :

$$\frac{fv(\tau) \subseteq \Delta}{\Delta \vdash \tau}$$

$$fv(\alpha) \stackrel{\text{def}}{=} \{\alpha\} \qquad fv(\mathtt{T}) \stackrel{\text{def}}{=} \emptyset \qquad fv(\tau_1 \rightarrow \tau_2) \stackrel{\text{def}}{=} fv(\tau_1) \cup fv(\tau_2)$$

$$fv(\forall \alpha.\, \tau) \stackrel{\text{def}}{=} fv(\tau) - \{\alpha\}$$

Typing

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau} \text{ (T-Var)}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta; \Gamma, x : \tau_1 \vdash M : \tau_2}{\Delta; \Gamma \vdash (\lambda x : \tau_1 .\, M) : \tau_1 \to \tau_2} \text{ (T-Abs)}$$

$$\frac{\Delta; \Gamma \vdash M_1 : \tau \to \tau' \qquad \Delta; \Gamma \vdash M_2 : \tau}{\Delta; \Gamma \vdash M_1\, M_2 : \tau'} \text{ (T-App)}$$

$$\frac{\Delta, \alpha; \Gamma \vdash M : \tau}{\Delta; \Gamma \vdash (\Lambda\alpha .\, M) : \forall \alpha . \tau} \text{ (T-TAbs)}$$

$$\frac{\Delta; \Gamma \vdash M_1 : \forall \alpha . \tau \qquad \Delta \vdash \tau_2}{\Delta; \Gamma \vdash M_1 \langle \tau_2 \rangle : \tau[\tau_2 / \alpha]} \text{ (T-TApp)}$$

Examples

- id $\stackrel{\text{def}}{=}$ $\Lambda\alpha.\, \lambda x : \alpha.\, x$
  - id : $\forall\alpha.\, \alpha \to \alpha$
  - id $\langle$Nat$\rangle$: Nat $\to$ Nat
  - id $\langle$Nat $\to$ Nat$\rangle$: (Nat $\to$ Nat) $\to$ (Nat $\to$ Nat)

Examples

- id $\stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda x : \alpha.\ x$
  - id : $\forall\alpha.\ \alpha \to \alpha$
  - id $\langle$Nat$\rangle$: Nat $\to$ Nat
  - id $\langle$Nat $\to$ Nat$\rangle$: (Nat $\to$ Nat) $\to$ (Nat $\to$ Nat)
- double $\stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda f : \alpha \to \alpha.\ \lambda x : \alpha.\ f\,(f\,x)$
  - double : $\forall\alpha.\ (\alpha \to \alpha) \to \alpha \to \alpha$
  - double $\langle$Nat$\rangle$: (Nat $\to$ Nat) $\to$ Nat $\to$ Nat

Examples

- $\mathsf{id} \stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda x : \alpha.\, x$
  - $\mathsf{id} : \forall\alpha.\, \alpha \to \alpha$
  - $\mathsf{id}\,\langle \mathsf{Nat} \rangle : \mathsf{Nat} \to \mathsf{Nat}$
  - $\mathsf{id}\,\langle \mathsf{Nat} \to \mathsf{Nat} \rangle : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$
- $\mathsf{double} \stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\,(f\,x)$
  - $\mathsf{double} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$
  - $\mathsf{double}\,\langle \mathsf{Nat} \rangle : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$
- $\mathsf{quadruple} \stackrel{\text{def}}{=} \Lambda\alpha.\, \mathsf{double}\,\langle \alpha \to \alpha \rangle\, (\mathsf{double}\,\langle \alpha \rangle)$
  - $\mathsf{quadruple} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$

Examples

- $\mathsf{id} \stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda x : \alpha.\, x$
  - $\mathsf{id} : \forall\alpha.\, \alpha \to \alpha$
  - $\mathsf{id}\,\langle \mathsf{Nat} \rangle : \mathsf{Nat} \to \mathsf{Nat}$
  - $\mathsf{id}\,\langle \mathsf{Nat} \to \mathsf{Nat} \rangle : (\mathsf{Nat} \to \mathsf{Nat}) \to (\mathsf{Nat} \to \mathsf{Nat})$
- $\mathsf{double} \stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\,(f\,x)$
  - $\mathsf{double} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$
  - $\mathsf{double}\,\langle \mathsf{Nat} \rangle : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Nat}$
- $\mathsf{quadruple} \stackrel{\text{def}}{=} \Lambda\alpha.\, \mathsf{double}\,\langle \alpha \to \alpha \rangle\,(\mathsf{double}\,\langle \alpha \rangle)$
  - $\mathsf{quadruple} : \forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$
- $\mathsf{selfApp} \stackrel{\text{def}}{=} \lambda x : (\forall\alpha.\, \alpha \to \alpha).\;\; x\,\langle \forall\alpha.\, \alpha \to \alpha \rangle\, x$
  - $\mathsf{selfApp} : (\forall\alpha.\, \alpha \to \alpha) \to (\forall\alpha.\, \alpha \to \alpha)$
  - Recall in STLC there's no way to type $\lambda x.\, x\, x$.

Properties

Theorem (Preservation)

For all M, M' and τ, if $\bullet; \bullet \vdash M : \tau$ and $M \longrightarrow M'$, then $\bullet; \bullet \vdash M' : \tau$.

Theorem (Progress)

For all M and τ, if $\bullet; \bullet \vdash M : \tau$, then either $M \in \text{Values}$ or $\exists M'.\ M \longrightarrow M'$.

Strong normalization: Every reduction path starting from a well-typed System F term is guaranteed to terminate.

Church Encodings

Recall in the untyped λ-calculus, we can encode boolean values:

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, x \\ \text{False} &\stackrel{\text{def}}{=} \lambda x.\, \lambda y.\, y \end{aligned}$$

In System F:

Church Encodings

Recall in the untyped λ-calculus, we can encode boolean values:

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ x \\ \text{False} &\stackrel{\text{def}}{=} \lambda x.\ \lambda y.\ y \end{aligned}$$

In System F:

$$\begin{aligned} \text{True} &\stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ x \\ \text{False} &\stackrel{\text{def}}{=} \Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ y \end{aligned}$$

Their type: $\text{Bool} \stackrel{\text{def}}{=} \forall\alpha.\ \alpha \to \alpha \to \alpha$.

$$\text{not} \stackrel{\text{def}}{=} \lambda b : \text{Bool}.\ \Lambda\alpha.\ \lambda x : \alpha.\ \lambda y : \alpha.\ b\,\langle\alpha\rangle\, y\, x$$

Its type: $\text{Bool} \to \text{Bool}$.

Church Encodings

Recall the untyped Church numerals:

$$\begin{aligned} \underline{0} &\stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, x \\ \underline{1} &\stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, f\, x \\ \underline{2} &\stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, f\,(f\, x) \end{aligned}$$

In System F:

$$\begin{aligned} \underline{0} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, x \\ \underline{1} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\, x \\ \underline{2} &\stackrel{\text{def}}{=} \Lambda\alpha.\, \lambda f : \alpha \to \alpha.\, \lambda x : \alpha.\, f\,(f\, x) \end{aligned}$$

Read *TAPL* for the encodings of many other data and operators.

Parametricity

Parametricity: polymorphic terms behave uniformly on their type variables.

- Given a parametrically polymorphic type, we know quite a bit about the behavior of any term of that type.

Parametricity

Parametricity: polymorphic terms behave uniformly on their type variables.

- Given a parametrically polymorphic type, we know quite a bit about the behavior of any term of that type.

Example

Write down all the functions that have type $\forall\alpha.\ \alpha \rightarrow \alpha$.

Parametricity

Parametricity: polymorphic terms behave uniformly on their type variables.

- Given a parametrically polymorphic type, we know quite a bit about the behavior of any term of that type.

Example

Write down all the functions that have type $\forall\alpha.\ \alpha \to \alpha$.

Every term you write behaves identically to $\Lambda\alpha.\ \lambda x : \alpha.\ x$.

Intuition: Because the term with type $\forall\alpha.\ \alpha \to \alpha$ is polymorphic in α, whatever it wants to do needs to work for every possible type α, and the lambda calculus is so *simple* that the only such thing it can do is to *return the argument*.

Parametricity

Parametricity: polymorphic terms behave uniformly on their type variables.

- Given a parametrically polymorphic type, we know quite a bit about the behavior of any term of that type.

Example

Consider the type Bool $\stackrel{\text{def}}{=} \forall\alpha.\, \alpha \to \alpha \to \alpha$.

Only two terms: $\Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, x$ *and* $\Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, y$.

They are exactly the terms True and False.

Parametricity

Parametricity: polymorphic terms behave uniformly on their type variables.

- Given a parametrically polymorphic type, we know quite a bit about the behavior of any term of that type.

Example

Consider the type Bool $\stackrel{\text{def}}{=} \forall\alpha.\, \alpha \to \alpha \to \alpha$.

Only two terms: $\Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, x$ *and* $\Lambda\alpha.\, \lambda x : \alpha.\, \lambda y : \alpha.\, y$.

They are exactly the terms True and False.

Read the paper *Theorems for free!* written by Phil Wadler in 1989. It's a fun paper and a famous application of parametricity.

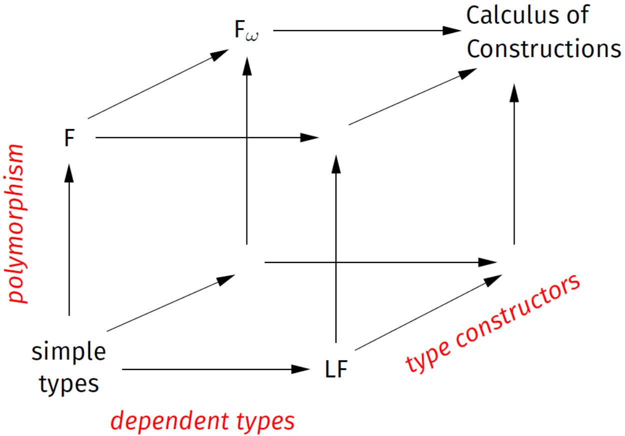
Impredicativity

The polymorphism of System F is often called *impredicative*.

In general, a definition (of a set, a type, etc.) is called *impredicative* if it involves a quantifier whose domain includes the very thing being defined.

For example, in System F, the type variable α in the type $\tau = \forall\alpha.\, \alpha \to \alpha$ ranges over all types, including τ itself (so that, for example, we can instantiate a term of type τ at type τ, yielding a function from τ to τ).

Lambda Cube



Proposed by Henk Barendregt in 1991.
The theoretical basis of Coq: Calculus of Inductive Constructions (CC + inductive definitions).