Polymorphism

In the simply typed lambda calculus, a term can have many types.

But a variable or parameter has only one type.

Example:

$$(\lambda x.xx)(\lambda y.y)$$

is untypable. But if we substitute actual parameter for formal, we obtain

$$(\lambda y.y)(\lambda y.y) : a \to a$$

Functions which can be applied to arguments of many types are called *polymorphic*.

Polymorphism in Programming

Polymorphism is essential for many program patterns.

Example: map

```
def map f xs =
    if (isEmpty (xs)) nil
    else cons (f (head xs)) (map (f, tail xs))
...
names: List[String]
nums : List[Int]
...
map toUpperCase names
map increment nums
```

Without a polymorphic type for map one of the last two lines is always illegal!

Forms of Polymorphism

Polymorphism means "having many forms".

Polymorphism also comes in several forms.

- *Universal polymorphism*, sometimes also called *generic types*: The ability to instantiate type variables.
- *Inclusion polymorphism*, sometimes also called *subtyping*: The ability to treat a value of a subtype as a value of one of its supertypes.
- *Ad-hoc polymorphism*, sometimes also called *overloading*: The ability to define several versions of the same function name, with different types.

We first concentrate on universal polymorphism.

Two basic approaches: *explicit* or *implicit*.

Explicit Polymorphism

We introduce a polymorphic type $\forall a.T$, which can be used just as any other type.

We then need to make introduction and elimination of $\forall$'s explicit. Typing rules:

$$(\forall \mathrm{E})\ \frac{\Gamma \vdash E : \forall a.T}{\Gamma \vdash E[U] : [U/a]T} \qquad (\forall \mathrm{I})\ \frac{\Gamma \vdash E : T}{\Gamma \vdash \Lambda a.E : \forall a.T}$$

We also need to give all parameter types, so programs become verbose.

Example:

```
def map [a][b] (f: a → b) (xs: List[a])  =
    if (isEmpty [a] (xs)) nil [a]
    else cons [b] (f (head [a] xs)) (map [a][b] (f, tail [a] xs))
...
names: List[String]
nums : List[Int]
...
map [String] [String] toUpperCase names
map [Int] [Int] increment nums
```

Implicit Polymorphism

Implicit polymorphism does not require annotations for parameter types or type instantations.

Idea: In addition to types (as in simply typed lambda calculus), we have a new syntactic category of *type schemes*. Syntax:

$$\text{Type Scheme} \quad S \quad ::= \quad T \mid \forall a.S$$

Type schemes are not fully general types; they are used only to type named values, introduced by a **val** construct.

The resulting type system is called the *Hindley/Milner system*, after its inventors. (The original treatment uses let ... in ... rather than **val** ... ; ...).

Hindley/Milner Typing rules

$$(\textsc{Var})\quad \Gamma, x : S, \Gamma' \vdash x : S \qquad (x \notin \mathrm{dom}(\Gamma'))$$

$$(\forall\mathrm{E})\ \frac{\Gamma \vdash E : \forall a.T}{\Gamma \vdash E : [U/a]T} \qquad (\forall\mathrm{I})\ \frac{\Gamma \vdash E : T \qquad a \notin \mathrm{tv}(\Gamma)}{\Gamma \vdash E : \forall a.T}$$

$$(\textsc{Val})\ \frac{\Gamma \vdash E : S \qquad \Gamma, x : S \vdash E' : T}{\Gamma \vdash \mathbf{val}\ x = E\ ;\ E' : T}$$

The other two rules are as in simply typed lambda calculus:

$$(\to\mathrm{I})\ \frac{\Gamma, x : T \vdash E : U}{\Gamma \vdash \lambda x.E : T \to U} \quad (\to\mathrm{E})\ \frac{\Gamma \vdash M : T \to U \qquad \Gamma \vdash N : T}{\Gamma \vdash M\ N : U}$$

Hindley/Milner in Programming Languages

Here is a formulation of the map example in the Hindley/Milner system.

```
val map = λf.λxs.
    if (isEmpty (xs)) nil
    else cons (f (head xs)) (map (f, tail xs))
...
// names: List[String]
// nums : List[Int]
// map  : ∀a.∀b.(a → b) → List[a] → List[b]
...
map toUpperCase names
map increment nums
```

Limitations of Hindley/Milner

Hindley/Milner still does not parameter types to be polymorphic. I.e.

$$(\lambda x.xx)(\lambda y.y)$$

is still ill-typed, even though the following is well-typed:

$$\textbf{val}\ id = \lambda y.y\ ;\ id\ id$$

With explicit polymorphism the expression could be completed to a well-typed term:

$$(\Lambda a.\lambda x : (\forall a : a \to a).x[a \to a](x[a]))(\Lambda b.\lambda y.y)$$

The Essence of val

We regard

$$\textbf{val}\ x = E\ ;\ E'$$

as a shorthand for

$$[E/x]E'$$

We use this equivalence to get a revised Hindley/Milner system.

Definition: Let HM' be the type system that results if we replace rule (Val) from the Hindley/Milner system HM by:

$$(\text{Val'})\ \frac{\Gamma\ \vdash\ E : T \qquad \Gamma\ \vdash\ [E/x]E' : U}{\Gamma\ \vdash\ \textbf{val}\ x = E\ \ ;\ E' : U}$$

Theorem: $\Gamma\ \vdash_{HM}\ E : S$ iff $\Gamma\ \vdash_{HM'}\ E : S$

The theorem establishes the following connection between the Hindley/Milner system and the simply typed lambda calculus F_1:

Corollary: Let E^* be the result of expanding all **val**'s in E according to the rule

$$\mathbf{val}\ x = E\ ;\ E' \quad \rightarrow \quad [E/x]E'$$

Then

$$\Gamma \vdash_{HM} E : T \quad \Rightarrow \quad \Gamma \vdash_{F_1} E^* : T$$

Furthermore, if every **val**-bound name is used at least once, we also have the reverse:

$$\Gamma \vdash_{F_1} E^* : T \quad \Rightarrow \quad \Gamma \vdash_{HM} E : T$$

Principal Types

Definition: A type T is a *generic instance* of a type scheme $S = \forall a_1 \ldots \forall a_n.T'$ if there is a substitution s on $a_1, \ldots, a_n$ such that $T = sT'$. We write in this case $S \leq T$.

Definition: A type scheme S' is a generic instance of a type scheme S iff for all types T

$$S' \leq T \quad \Rightarrow \quad S \leq T$$

We write in this case $S \leq S'$.

Definition: A type scheme S is *principal* (or: *most general*) for Γ and E iff

- $\Gamma \vdash E : S$
- $\Gamma \vdash E : S'$ implies $S \leq S'$

Definition: A type system TS has the *principal typing property* iff, whenever $\Gamma \vdash_{TS} E : S$ then there exists a principal type scheme for Γ and E.

Theorem:

1. HM' without **val** has the p.t.p.
2. HM' with **val** has the p.t.p.
3. HM has the p.t.p.

Proof sketch: (1.): Use type reconstruction result for the simply typed lambda calculus. (2.): Expand all **val**'s and apply (1.). (3.): Use equivalence between HM and HM'.

These observations could be used to come up with a type reconstruction algorithm for HM. But in practice one takes a more direct approach.

Type Reconstruction for Hindley/Milner

Type reconstruction for the Hindley/Milner system works as for simply typed lambda calculus. We only have to add a clause for **val** expressions:

$$TP : \mathit{Judgement} \rightarrow \mathit{Subst} \rightarrow \mathit{Subst}$$

$$\begin{array}{l} TP(\Gamma \vdash E : T)\ s = \\ \quad \textbf{case}\ E\ \textbf{of} \\ \qquad ... \\ \qquad \textbf{val}\ x = E_1\ ;\ E_2 \quad : \quad \textbf{let}\ a, b\ \text{fresh}\ \textbf{in} \\ \qquad \qquad \qquad \qquad \qquad \textbf{let}\ s_1 = TP\ (\Gamma \vdash E_1 : a)\ \textbf{in} \\ \qquad \qquad \qquad \qquad \qquad TP\ (\Gamma, x : \textbf{gen}(s_1\ \Gamma, s_1\ a) \vdash E_2 : b)\ s_1 \end{array}$$

where $\textbf{gen}(\Gamma, T) \ = \ \forall \mathrm{tv}(T) \backslash \mathrm{tv}(\Gamma).T$.