1 ML Polymorphism

The type reconstruction algorithm of the previous lecture doesn’t fully reconstruct the type $\lambda x. x$. Instead it generates a type $T_f \rightarrow T_f$ where $T_f$ is some fresh type variable. This is actually an opportunity to obtain a more expressive type system: $\lambda x. x$ can be assigned a type that is a schema: $\forall T_f. T_f \rightarrow T_f$. $\lambda x. x$ is a polymorphic term. That is it can be used at many different types, by instantiating it. The type $\text{int} \Rightarrow \text{int}$ is an instantiation of this schema.

Consider the following code that is valid in ML:

```
let id = $\lambda x. x$ in
let a = $\lambda f. \text{int} \rightarrow \text{int} \cdot \lambda g. \text{bool} \rightarrow \text{bool}$ in
(a id) id
```

We can define a language that captures what is going on in ML:

\[
e ::= x \mid b \mid \lambda x.e \mid e_0.e_1 \mid \text{let } x = e \text{ in } e'
\]

\[
\sigma ::= \forall X_1, \ldots, X_n. \tau (n \geq 0)
\]

\[
\tau ::= B \mid \tau_1 \rightarrow \tau_2 \mid X
\]

\[
\Delta ::= \emptyset \mid \Delta, X
\]

\[
\Gamma ::= \emptyset \mid \Gamma, x : \sigma
\]

\[
x \in \text{Var}
\]

\[
X \in \text{Typevar}
\]

In the above rules $\sigma$ is a type schema. We allow $n \geq 0$ so we can have a schema without any $X$’s; we denote that by $\tau$ instead of $\forall \tau$.

We introduce a new piece of our typing context called $\Delta$, which keeps track of the legal type names. We will treat it as a set of names for now.

Our typing context is now $\Delta; \Gamma$, so $\Delta; \Gamma \vdash e : \tau$ is our standard type assertion. We also have the assertion $\Delta \vdash \tau$ which tells us that $\tau$ is a well-formed type in $\Delta$.

1.1 Well-Formedness Rules for Types

\[
\begin{array}{c}
\Delta \vdash B \\
\Delta, X \vdash X \\
\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2 \\
\Delta \vdash \tau_1 \rightarrow \tau_2
\end{array}
\]
1.2 Typing Rules

\[
\Delta; \Gamma \vdash b : B
\]

\[\forall i \leq n. \Delta \vdash \tau_i\]

\[\Delta; \Gamma, x : \forall X_1, \ldots, X_n. \tau \vdash x : \tau \{ \tau_1 / X_1, \ldots, \tau_n / X_n \}\]

\[
\Delta; \Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Delta; \Gamma \vdash e_1 : \tau
\]

\[\Delta; \Gamma \vdash e_0 \ e_1 : \tau'\]

\[\Delta; \Gamma \vdash \lambda x. e : \tau \rightarrow \tau'\]

\[
\Delta, X_1, \ldots, X_n; \Gamma \vdash e : \tau
\]

\[
\Delta; \Gamma \vdash \forall X_1, \ldots, X_n. \tau \vdash e' : \tau' \quad \{ X_1, \ldots, X_n \} \cap \Delta = \emptyset
\]

Here’s an example of how we can type a program containing the polymorphic term \(\lambda x. x\):

\[
\emptyset \vdash \text{int}
\]

\[
X; \emptyset \vdash x : X \quad X \vdash X
\]

\[\emptyset; \emptyset \vdash \lambda x. x : X \rightarrow X\]

\[\emptyset; \emptyset \vdash \text{id} \ : \int \ \text{in} \ \text{id} \ 2 : \int\]

Clearly we have gained expressive power in this type system.

1.3 Type Reconstruction

\[
\mathcal{R}(x, \Gamma, S) = ?
\]

let \(\forall X_1, \ldots, X_n. \tau = \Gamma(x)\)

in \(\langle \tau \{ T_1 f / X_1, \ldots, T_n f / X_n \}, s \rangle\)

\[
\mathcal{R}((\text{id} \ 2), \Gamma, S) =
\]

let \(\langle \tau_1, S_1 \rangle = \mathcal{R}(\text{id}, \Gamma, S)\)

in \(\langle \tau_1 \ 2, S_1 \rangle\)

\[
\{ X_1, \ldots, X_n \} = \text{FTV}(S \tau_1) - \text{FTV}(S \Gamma)\]

where \(\text{FTV}(\tau)\) reports the type variables in a type \(\tau\) and \(\text{FTV}(\Gamma)\) reports the free type variables in a typing context \(\Gamma\).

2 Parametric Polymorphism

ML-style (“let”) polymorphism, which we have just seen, is an example of \emph{parametric polymorphism}. In the type schema \(\forall X_1, \ldots, X_n. \tau\), the \(X_i\) are \emph{type parameters}. We can think of the type schema as being a function that can be applied to (instantiated on) real types \(\tau_i\) to obtain a type. Because type parameters can be instantiated only on ordinary types \(\tau\), this is \emph{predicative} polymorphism.

Through application we derive a term from two other terms, in parametric polymorphism we derive a term from a term and a type. There are other generalizations of application:

\[
\text{application} : \text{term} \times \text{term} \rightarrow \text{term}
\]

\[
\text{parametric polymorphism} : \text{term} \times \text{type} \rightarrow \text{term}
\]

\[
\text{higher order polymorphism} : \text{type} \times \text{type} \rightarrow \text{type}
\]

\[
\text{dependent types} : \text{type} \times \text{term} \rightarrow \text{type}
\]
Consider the template declaration template (class \( X \)) \( e \); in C++, the type of the declared expression \( e \) is roughly \( \forall X.\tau \). This is an example of parametric polymorphism in an industrial language. (If \( e \) is a class declaration, it is also an example of higher-order polymorphism. Java extensions like GJ and PolyJ also support this feature.)

Dependent types are seen in some varieties of Pascal; for example, \( f(a: \text{array}[n] \text{ of int}, n: \text{int}) \) is a declaration of a function that takes in an array whose type depends on the term \( n \).

### 2.1 Full Predicative Polymorphism

We can generalize the previous type system to obtain the full power of predicative polymorphism at the cost of losing the ability to infer types.

\[
\begin{align*}
e & \ ::= \ x \mid \lambda x.\ e \mid e_0\ e_1 \mid \Lambda X.\ e \mid e[\tau] \mid \lambda x:\sigma.\ e \mid \lambda x:\tau.\ e \\
\sigma & \ ::= \ \tau \mid \forall X.\sigma \mid \sigma_1 \to \sigma_2
\end{align*}
\]

The first new rule for \( e \) is type abstraction and the second is type application. The last two replace our old rule for \( \lambda \).

### 2.2 Well-Formedness Rules

Type judgements now have the form \( \Delta \vdash \sigma \). Type well-formedness needs the following extension:

\[
\frac{\Delta \vdash \sigma_1 \quad \Delta \vdash \sigma_2}{\Delta \vdash \sigma_1 \to \sigma_2}
\]

We also add the following reduction to our operational semantics: the application of a type abstraction. Notice that it has no actual computational content; erasing all the \( \Lambda \)'s, \( [\tau] \)'s, and other type annotations doesn’t affect evaluation of a program in this language.

\[
(\Lambda X.\ e)[\tau] \mapsto e\{\tau/X\}
\]

### 2.3 Typing Rules

We can actually simplify our rule for typing variables, it is replaced by the first rule:

\[
\frac{\Delta;\Gamma, x:\sigma \vdash e: \sigma}{\Delta;\Gamma \vdash \lambda x:\sigma.\ e : \sigma \to \sigma'}
\]

\[
\frac{\Delta;\Gamma \vdash e : \sigma \\ X \notin \Delta}{\Delta;\Gamma \vdash \lambda X.\ e : \forall X.\sigma}
\]

\[
\frac{\Delta;\Gamma \vdash \forall X.\sigma \quad \Delta \vdash \tau}{\Delta;\Gamma \vdash e[\tau] : \sigma\{\tau/X\}}
\]

### 2.4 Example

We can now give types to many of the terms we saw when exploring lambda calculus encodings earlier in the class. For example,

\[
\text{true} = \Lambda X.\lambda x:X.\lambda y:X.\ x
\]

However, with predicative polymorphism we still can’t type \( \text{SA} = \lambda x(xx) \)
3 Impredicative Polymorphism

If we fold together the two kinds of types $\tau$ and $\sigma$, we arrive at the *polymorphic $\lambda$ calculus*, also called "System F". This language provides *impredicative* polymorphism in which a type schema can be instantiated on a type schema:

$$\tau, \sigma ::= B \mid X \mid \sigma_1 \rightarrow \sigma_2 \mid \forall X.\sigma$$

The typing rules and SOS are unchanged. The difference is that now a polymorphic term can be instantiated on a type schema $\sigma$, not just an ordinary type, and thus a type variable $X$ can refer to an arbitrary type schema $\sigma$.

We can now type $SA$:

$$\lambda x : \forall X.X \rightarrow X(x[\forall Y.Y \rightarrow Y] x) : (\forall Y.Y \rightarrow Y) \rightarrow (\forall Y.Y \rightarrow Y)$$

However we still can’t type $\Omega = (SA \ SA)$. In fact, we can’t type any divergent term: the polymorphic $\lambda$ calculus is strongly normalizing. This is particularly surprising because the language is quite expressive; for example, we can compute all primitive recursive functions in the polymorphic $\lambda$ calculus. The proof of strong normalization is achieved using logical relations.

Impredicative polymorphism is harder to implement, and unlike the simply typed lambda calculus, it doesn’t have a set-theoretic model (despite the fact that it has no divergent terms.) The difficulty is that the natural interpretation of a polymorphic type such as $\forall X.X$ is the set of all functions that map a type interpretation (i.e., a set) to another type interpretation. However, this function must map the interpretation of the type $\forall X.X$ itself, which means that the extensional view of the function is not a well-founded set.