Scribe: Misha Zatsman Lecturer: Andrew Myers

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 $T_f \to T_f$ 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: $\forall T_f.T_f \to T_f$. $\lambda x.\, x$ is a polymorphic term. That is it can be used at many different types, by instantiating it. The type $\mathsf{int} \Rightarrow \mathsf{int}$ is an instantiation of this schema.

Consider the following code that is valid in ML:

let id $= \lambda x.\, x$ in
  let $a = \lambda f.\mathsf{int} \to \mathsf{int}.\lambda g.\mathsf{bool} \to \mathsf{bool}$
    in
      (a id) id

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

$$\begin{array}{rcl} e & ::= & x \mid b \mid \lambda x.e \mid e_0\ e_1 \mid \mathsf{let}\ x = e\ \mathsf{in}\ e' \\ \sigma & ::= & \forall X_1, \ldots, X_n.\tau^{(n \geq 0)} \\ \tau & ::= & B \mid \tau_1 \to \tau_2 \mid X \\ \Delta & ::= & \emptyset \mid \Delta, X \\ \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma \\ x & \in & \mathbf{Var} \\ X & \in & \mathbf{Typevar} \end{array}$$

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

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

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

1.1 Well-Formedness Rules for Types

$$\frac{}{\Delta \vdash B} \qquad \frac{}{\Delta, X \vdash X} \qquad \frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}$$

1.2 Typing Rules

$$\frac{}{\Delta;\Gamma \vdash b : B}$$

$$\frac{\forall i \leq n. \Delta \vdash \tau_i}{\Delta;\Gamma, x\!:\!\forall X_1,\ldots,X_n.\tau \vdash x : \tau\{\tau_1/X_1,\ldots\tau_n/X_n\}}$$

$$\frac{\Delta;\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Delta;\Gamma \vdash e_1 : \tau}{\Delta;\Gamma \vdash e_0\ e_1 : \tau'}$$

$$\frac{\Delta;\Gamma, x\!:\!\tau \vdash e : \tau' \quad \Delta \vdash \tau}{\Delta;\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

$$\frac{\Delta, X_1,\ldots,X_n;\Gamma \vdash e : \tau \quad \Delta;\Gamma, x\!:\!\forall X_1,\ldots,X_n.\tau \vdash e' : \tau' \quad \{X_1,\ldots,X_n\} \cap \Delta = \emptyset}{\Delta;\Gamma \vdash \ \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'}$$

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

$$\frac{\dfrac{X;x\!:\!X \vdash x : X \quad X \vdash X}{X;\emptyset \vdash \lambda x.x : X \rightarrow X} \quad \dfrac{\dfrac{\emptyset \vdash \mathsf{int}}{\emptyset;\mathsf{id}.\forall X.X \rightarrow X \vdash \mathsf{id} : \mathsf{int} \rightarrow \mathsf{int}}}{\emptyset;\mathsf{id}.\forall X.X \rightarrow X \vdash \mathsf{id}\ 2 : \mathsf{int}}}{\emptyset;\emptyset \vdash \mathsf{let}\ \mathsf{id} = \lambda x.x\ \mathsf{in}\ (\mathsf{id}\ 2) : \mathsf{int}}$$

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
$\langle\tau\{T_{1f}/X_1,\ldots,T_{nf}/X_n\}, s\rangle$

$\mathcal{R}(\text{let } x = e_1 \ \mathsf{in}\ e_2,\Gamma,S) =$
let $\langle\tau_1,S_1\rangle = \mathcal{R}(e_1,\Gamma,S)$ in
let $\sigma = \forall X_1,\ldots,X_n.\tau_1$
in $\mathcal{R}(e',\Gamma[x \mapsto \sigma],S_1)$

$\{X_1,\ldots,X_n\} = \ FTV(S\tau_1) - \ FTV(S\Gamma)$ where FTV(τ) reports the type variables in a type τ and FTV(Γ) reports the free type variables in a typing context Γ.

2 Parametric Polymorphism

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

$$\begin{array}{rcl} \textit{application} & : & \textit{term} \times \textit{term} \rightarrow \textit{term} \\ \textit{parametric polymorphism} & : & \textit{term} \times \textit{type} \rightarrow \textit{term} \\ \textit{higher order polymorphism} & : & \textit{type} \times \textit{type} \rightarrow \textit{type} \\ \textit{dependent types} & : & \textit{type} \times \textit{term} \rightarrow \textit{type} \end{array}$$

Consider the template declaration `template` $\langle$`class` $X\rangle$ 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: array[n] of int, n: 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.

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\, e \mid e_0\, e_1 \mid \Lambda X.e \mid e[\tau] \mid \lambda x\!:\!\sigma.\, e \mid \lambda x\!:\!\tau.\, e \\ \sigma & ::= & \tau \mid \forall X.\sigma \mid \sigma_1 \to \sigma_2 \end{array}$$

The first new rule for e is type abstraction and the second is type application. The last two replace our old rule for λ.

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 Λ'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:

$$\frac{}{\Delta; \Gamma, x\!:\!\sigma \vdash x : \sigma} \qquad \frac{\Delta; \Gamma, x\!:\!\sigma \vdash e : \sigma' \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash \lambda x\!:\!\sigma.\, e : \sigma \to \sigma'}$$

$$\frac{\Delta, X; \Gamma \vdash e : \sigma \quad X \notin \Delta}{\Delta; \Gamma \vdash \Lambda X.e : \forall X.\sigma} \qquad \frac{\Delta; \Gamma \vdash e : \forall X.\sigma \quad \Delta \vdash \tau}{\Delta; \Gamma \vdash e[\tau] : \sigma\{\tau/X\}}$$

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,

$$\textbf{true} = \Lambda X.\lambda x\!:\!X.\lambda y\!:\!X.x$$

However, with predicative polymorphism we still can't type $SA = \lambda x(xx)$

3 Impredicative Polymorphism

If we fold together the two kinds of types τ and σ, we arrive at the *polymorphic λ 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 σ, not just an ordinary type, and thus a type variable X can refer to an arbitrary type schema σ.

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 λ calculus is strongly normalizing. This is particularly surprising becasue the language is quite expressive; for example, we can compute all primitive recursive functions in the polymorphic λ 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.