CMSC 336: Type Systems for Programming Languages

Lecture 10: Polymorphism

Acar & Ahmed **19 February 2008**

Contents

1 Polymorphism

In the simply-typed λ-calculus, every expression must have a unique type. In particular, every function has uniquely determined domain and range types. Consider identity function—we must write a distinct identity function for each type:

$$id_\tau = \lambda x : \tau.x$$

Consider the composition function: f o g
If f has type $\tau_2 \rightarrow \tau_3$ and g has type $\tau_1 \rightarrow \tau_2$ then fog has type $\tau_1 \rightarrow \tau_3$.
$o_{\tau_1,\tau_2,\tau_3} = \lambda f : \tau_2 \rightarrow \tau_3.\lambda g : \tau_1 \rightarrow \tau_2.\lambda x : \tau_1.f(g(x)$

Now in the type systems that we have seen thus far you have to write a distinct composition function for each triple of types. Rather than writing the "same" function over and over again, with the only difference being the types, can we do something smarter? What is needed is a way to write code that is generic or parametric in the types involved. This is called polymorphism.

2 Polymorphic λ-Calculus: Syntax

Polymorphic Typed λ- Calculus/ System F/ Second-order λ calculus.

$$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau$$

$$e ::= x \mid \lambda x : \tau_1.e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e[\tau]$$

3 Static Semantics

Typing judgments. Before we give rules for typing expressions of the polymorphic λ-calculus, we need rules for type formation. We need a type variable context Δ.

$$\Delta = \alpha_1\ \texttt{type}, \alpha_2\ \texttt{type}, \alpha_3\ \texttt{type}, \ldots$$

Delta tells us what type variables are good.

Type formation. The judgment $\Delta \vdash \tau\ \texttt{type}$ says that τ is a well-formed type under context Δ. We adopt the convention that whenever we whenever we have the context $\Delta, \alpha\ \texttt{type}$ that means that $\alpha \notin \Delta$. The type formation rules are as follows:

$$\frac{}{\Delta, \alpha\ \texttt{type} \vdash \alpha\ \texttt{type}}$$

$$\frac{\Delta \vdash \tau_1\ \texttt{type} \qquad \Delta \vdash \tau_2\ \texttt{type}}{\Delta \vdash \tau_1\ \rightarrow\ \tau_2\ \texttt{type}}$$

$$\frac{\Delta, \alpha\ \texttt{type} \vdash \tau\ \texttt{type}}{\Delta \vdash \forall\alpha.\tau}$$

Notice that the last rule above (the rule for $\forall\alpha.\tau$), we do not have the premise $\alpha \notin \Delta$. While it would be perfectly fine to add that premise, we do not need it since we adopted the convention above that whenever we have the context $\Delta, \alpha\ \texttt{type}$, this means that $\alpha \notin \Delta$.

Typing expressions. The judgment $\Delta; \Gamma \vdash e : \tau$ says that the expression e has type τ under contexts Δ and Γ (where Δ ranges over finite sets of type variable formation hypotheses $\alpha\ \texttt{type}$, and Γ ranges over finite sets of expression variable typing hypotheses $x : \tau$). Again, we adopt the convention that when we write $\Delta, \alpha\ \texttt{type}$, this means $\alpha \notin \Delta$. Similarly, whenever we have the context $\Gamma, x : \tau$ this means $x \notin \mathrm{dom}(\Gamma)$.

The rules for typing expressions are as follows:

$$\frac{\forall x \in \mathrm{dom}(\Gamma).\ \Delta \vdash \Gamma(x)\ \texttt{type}}{\Delta; \Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.e : \tau_1\ \rightarrow\ \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_2\ \rightarrow\ \tau \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Delta, \alpha\ \texttt{type}; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}$$

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau' \quad \Delta \vdash \tau\ \texttt{type}}{\Delta; \Gamma \vdash e[\tau] : [\tau/\alpha]\ \tau'}$$

Notice that we want it to be the case that whenever we have a judgment $\Delta; \Gamma \vdash e : \tau$, all free type variables that appear in Γ, e or τ, must appear in Δ. The above typing rules ensure that this is the case—in particular, notice the premise of the typing rules for variables. In most presentations, however, the variable rule will simply be written as follows.

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau}$$

The above rule is justified by adopting the convention that we will only be concerned with well-formed judgments. A judgment $\Delta; \Gamma \vdash e : \tau$ is well-formed if all free type variables in Γ appear in Δ (i.e., if $\texttt{FTV}(\Gamma) \subseteq \Delta$, or equivalently, if $\forall x \in \text{dom}(\Gamma).\ \Delta \vdash \Gamma(x)\ \texttt{type}$).

An Example:

$$\frac{\dfrac{\alpha\ \texttt{type} \vdash \alpha\ \texttt{type} \qquad \alpha\ \texttt{type} \vdash \alpha\ \texttt{type}}{\alpha\ \texttt{type} \vdash \alpha\ \rightarrow\ \alpha\ \texttt{type}}}{\bullet \vdash \forall\alpha.\alpha\ \rightarrow\ \alpha\ \texttt{type}}$$

Why is "alpha" renaming important here. Consider the example:

$$\Lambda\alpha.\Lambda\alpha.\lambda x : \alpha.x$$

Which alpha is it being used. For example if we instantiate the alpha with `int`and `bool`in that order which alpha is the inner one?

$$\vdash \Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha\ \rightarrow\ \alpha.$$

Let's define the polymorphic identity function:

$$id = \Lambda\alpha.\lambda x : \alpha.x$$

We can instantiate id with `bool` to serve as the boolean identity function: $id[bool] \equiv \lambda x : bool.x$

Notice that we have types in our terms. This was why before adding polymorphism, we could not write a function that was polymorphic in the type.

4 Dynamic Semantics

We define a call-by-value, small-step operational semantics.

The values in the polymorphic λ-calculus are as follows:

$$v ::= \lambda x : \tau_1.e \mid \Lambda\alpha.e$$

The rules are as follows:

$$\frac{e_1 \ \rightarrow \ e_1'}{e_1 \ e_2 \ \rightarrow \ e_1' \ e_2}$$

$$\frac{e_2 \ \rightarrow \ e_2'}{v_1 \ e_2 \ \rightarrow \ v_1 \ e_2'}$$

$$\frac{}{(\lambda x : \tau.e)\, v \ \rightarrow \ e[v/x]}$$

$$\frac{e \ \rightarrow \ e'}{e[\tau] \ \rightarrow \ e'[\tau]}$$

$$\frac{}{(\Lambda\alpha.e)[\tau] \ \rightarrow \ e[\tau/\alpha]}$$

5 Substitution

Substitution Lemma:

if $\Delta; \Gamma, x : \tau_1 \vdash e : \tau$ and $\Gamma \vdash e_1 : \tau_1$
then $\Gamma \vdash [e_1/x]\ e : \tau$

If Δ, α `type` $\vdash \tau$ `type` and $\Delta \vdash \tau_1$ `type`
then $\Delta \vdash [\tau_1/\alpha]\ \tau$ `type`.

If Δ, α `type`$; \Gamma \vdash e : \tau$ and $\Delta \vdash \tau_1$ `type`
then $\Delta; [\tau_1/\alpha]\ \Gamma \vdash [\tau_1/\alpha]\ e : [\tau_1/\alpha]\ \tau$.

Recall: When we write $\Delta; \Gamma \vdash e : \tau$, we assume that Γ is a well-formed context with respect to Δ (written $\Delta \vdash \Gamma$) and that *tau* is a well-formed type wrt Δ ($\Delta \vdash \tau$ `type`). Here's how we define what it means for Γ to be a well-formed given some Δ.

$$\frac{\forall (x : \tau) \in \Gamma. \Delta \vdash \tau \ \texttt{type}}{\Delta \vdash \Gamma}$$

6 Type Safety

Prove progress and preservation as for the simply typed lambda calculus, appealing to substitution and inversion lemmas as appropriate.

Inversion Lemma:
If $\Delta; \Gamma \vdash \Lambda\alpha.e : \tau$
then $\tau = \forall\alpha.\tau'$ and $\Delta, \alpha\ \texttt{type}; \Gamma \vdash e : \tau'$

If $\Delta; \Gamma \vdash e[\tau_1] : \tau$
then $\tau = [\tau_1/\alpha]\ \tau_2$, $\Delta; \Gamma \vdash e : \forall\alpha.\tau_2$, and $\Delta \vdash \tau_1\ \texttt{type}$.

Some history: John Reynolds and Jean-Yves Girard independently worked on a logic with variable types. Girard developed his logic for second-order arithmetic and called it *System F*. Reynolds developed his logic for polymorphic programming and called it the *polymorphic typed lambda calculus* or the *second order lambda calculus*.

7 Church Encodings: Defining Primitive Types in System F

void is the empty type. It has no terms in it.

unit is the type that has only one element in it.

How can we encode unit? How about mapping the type of unit to $\forall\alpha.\alpha \to \alpha$.

$$\begin{aligned} \textit{unit} &\equiv \forall\alpha.\alpha \to \alpha \\ &\equiv \Lambda\alpha.\forall x : \alpha.x \end{aligned}$$

(no elim)

$$\begin{aligned} \texttt{bool} &\equiv \forall\alpha.\alpha \to \alpha \to \alpha \\ \texttt{true} &\equiv \Lambda\alpha.\forall x : \alpha.\forall y : \alpha.x \\ \texttt{false} &\equiv \Lambda\alpha.\forall x : \alpha.\forall y : \alpha.y \\ \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 &\equiv e_0[\tau]e_1e_2 \end{aligned}$$

$$\begin{aligned} \tau_1 + \tau_2 &\equiv \forall\alpha.(\tau_1 \to \alpha) \to (\tau_2 \to \alpha) \to \alpha \\ \texttt{inl}_e &\equiv \Lambda\alpha.\lambda f_l : \tau_1 \to \alpha.\lambda f_r : \tau_2 \to \alpha.f_l(e) \\ \texttt{inr}_e &\equiv \Lambda\alpha.\lambda f_l : \tau_1 \to \alpha.\lambda f_r : \tau_2 \to \alpha.f_r(e) \\ \texttt{case}\ e\ \texttt{of}\ \texttt{inl}_{x:\tau_1} \Rightarrow e_1 : \tau \mid \texttt{inr}_{y:\tau_2} \Rightarrow e_2 : \tau &\equiv e[\tau](\lambda x : \tau_1.e_1)(\lambda y : \tau_2.e_2) \end{aligned}$$

$$\begin{aligned} \tau_1 \times \tau_2 &\equiv \forall\alpha.(\tau_1 \to \tau_2 \to \alpha) \to \alpha \\ < e_1, e_2 > &\equiv \Lambda\alpha.\lambda p : \tau_1 \to \tau_2 \to \alpha.p\ e_1\ e_2 \\ fst(e) &\equiv e[\tau_1]\ (\lambda x : \tau_1 : \lambda y : \tau_2.x) \\ snd(e) &\equiv e[\tau_2]\ (\lambda x : \tau_1 : \lambda y : \tau_2.y) \end{aligned}$$

Bob Harper's book (PFPL, chapter 24) shows how primitive recursion can be encoded in this calculus (i.e., an encoding of the type `nat`, with intro forms `z` and `succ`(e), and elim form `rec`$(e, e_0, x : \tau.e_1)$).