Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Parametric polymorphism

Lecture 14 Thursday, March 13, 2014

1 Parametric polymorphism

Polymorph means "many forms". *Polymorphism* is the ability of code to be used on values of different types. For example, a polymorphic function is one that can be invoked with arguments of different types. A polymorphic datatype is one that can contain elements of different types.

Several kinds of polymorphism are commonly used in modern languages.

- *Subtype polymorphism* gives a single term many types using the subsumption rule. For example, a function with argument τ can operate on any value with a type that is a subtype of τ.
- *Ad-hoc polymorphism* usually refers to code that appears to be polymorphic to the programmer, but the actual implementation is not. A typical example is *overloading*: using the same function name for functions with different kinds of parameters. Although it looks like a polymorphic function to the code that uses it, there are actually multiple function implementations (none being polymorphic) and the compiler invokes the appropriate one. Ad-hoc polymorphism is a dispatch mechanism: the type of the arguments is used to determine (either at compile time or run time) which code to invoke.
- *Parametric polymorphism* refers to code that is written without knowledge of the actual type of the arguments; the code is parametric in the type of the parameters. Examples include polymorphic functions in ML, or generics in Java 5.

We consider parametric polymorphism in more detail. Suppose we are working in the simply-typed lambda calculus, and consider a "doubling" function for integers that takes a function f, and an integer x, applies f to x, and then applies f to the result.

$$\mathit{doubleInt} \triangleq \lambda f\!:\!\textbf{int} \to \textbf{int}.\, \lambda x\!:\!\textbf{int}.\, f\ (f\ x)$$

We could also write a double function for booleans. Or for functions over integers. Or for any other type...

$$\begin{aligned}\mathit{doubleBool} &\triangleq \lambda f\!:\!\textbf{bool} \to \textbf{bool}.\, \lambda x\!:\!\textbf{bool}.\, f\ (f\ x)\\ \mathit{doubleFn} &\triangleq \lambda f\!:\!(\textbf{int} \to \textbf{int}) \to (\textbf{int} \to \textbf{int}).\, \lambda x\!:\!\textbf{int} \to \textbf{int}.\, f\ (f\ x)\\ &\vdots\end{aligned}$$

In the simply typed lambda calculus, if we want to apply the doubling operation to different types of arguments in the same program, we need to write a new function for each type. This violates the *abstraction principle* of software engineering:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. When similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

In the doubling functions above, the varying parts are the types. We need a way to abstract out the type of the doubling operation, and later instantiate this abstract type with different concrete types.

We extend the simply-typed lambda calculus with abstraction over types, giving the *polymorphic lambda calculus*, also called *System F*.

A *type abstraction* is a new expression, written $\Lambda X.\, e$, where Λ is the upper-case form of the Greek letter lambda, and X is a *type variable*. We also introduce a new form of application, called *type application*, or *instantiation*, written $e_1\ [\tau]$.

When a type abstraction meets a type application during evaluation, we substitute the free occurrences of the type variable with the type. Note that instantiation does not require the program to keep run-time type information, or to perform type checks at run-time; it is just used as a way to statically check type safety in the presence of polymorphism.

1.1 Syntax and operational semantics

The new syntax of the language is given by the following grammar.

$$\begin{aligned} e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid \Lambda X.\, e \mid e\ [\tau] \\ v &::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda X.\, e \end{aligned}$$

The evaluation rules for the polymorphic lambda calculus are the same as for the simply-typed lambda calculus, augmented with new rules for evaluating type application.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E\ [\tau]$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x\!:\!\tau.\, e)\ v \longrightarrow e\{v/x\}}$$

$$\text{TYPE-REDUCTION}\ \frac{}{(\Lambda X.\, e)\ [\tau] \longrightarrow e\{\tau/X\}}$$

Let's consider an example. In this language, the polymorphic identity function is written as

$$ID \triangleq \Lambda X.\, \lambda x\!:\!X.\, x$$

We can apply the polymorphic identity function to **int**, producing the identity function on integers.

$$(\Lambda X.\, \lambda x\!:\!X.\, x)\ [\mathbf{int}] \longrightarrow \lambda x\!:\!\mathbf{int}.\, x$$

We can apply ID to other types as easily:

$$(\Lambda X.\, \lambda x\!:\!X.\, x)\ [\mathbf{int} \to \mathbf{int}] \longrightarrow \lambda x\!:\!\mathbf{int} \to \mathbf{int}.\, x$$

1.2 Type system

We also need to provide a type for the new type abstraction. The type of $\Lambda X.\, e$ is $\forall X.\, \tau$, where τ is the type of e, and may contain the type variable X. Intuitively, we use this notation because we can instantiate the type expression with any type for X: for any type X, expression e can have the type τ (which may mention X).

$$\tau ::= \mathbf{int} \mid \tau_1 \to \tau_2 \mid X \mid \forall X.\, \tau$$

Type checking expressions is slightly different than before. Besides the type environment Γ (which maps variables to types), we also need to keep track of the set of type variables Δ. This is to ensure that a type variable X is only used in the scope of an enclosing type abstraction $\Lambda X.\, e$. Thus, typing judgments are now of the form $\Delta, \Gamma \vdash e : \tau$, where Δ is a set of type variables, and Γ is a typing context. We also use an additional judgment $\Delta \vdash \tau$ ok to ensure that type τ uses only type variables from the set Δ.

$$\frac{}{\Delta, \Gamma \vdash n\!:\!\mathbf{int}} \qquad \frac{\Delta \vdash \tau\ \mathsf{ok}}{\Delta, \Gamma \vdash x\!:\!\tau}\ \Gamma(x) = \tau \qquad \frac{\Delta, \Gamma, x\!:\!\tau \vdash e\!:\!\tau' \quad \Delta \vdash \tau\ \mathsf{ok}}{\Delta, \Gamma \vdash \lambda x\!:\!\tau.\, e\!:\!\tau \to \tau'}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Delta \cup \{X\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \Lambda X.\, e : \forall X.\, \tau} \qquad \frac{\Delta, \Gamma \vdash e : \forall X.\, \tau' \quad \Delta \vdash \tau \ \mathsf{ok}}{\Delta, \Gamma \vdash e\ [\tau] : \tau'\{\tau/X\}}$$

$$\frac{}{\Delta \vdash X \ \mathsf{ok}}\ X \in \Delta \qquad \frac{}{\Delta \vdash \mathbf{int}\ \mathsf{ok}} \qquad \frac{\Delta \vdash \tau_1 \ \mathsf{ok} \quad \Delta \vdash \tau_2 \ \mathsf{ok}}{\Delta \vdash \tau_1 \to \tau_2 \ \mathsf{ok}} \qquad \frac{\Delta \cup \{X\} \vdash \tau \ \mathsf{ok}}{\Delta \vdash \forall X.\, \tau \ \mathsf{ok}}$$

1.3 Examples

Let's consider the doubling operation again. We can write a polymorphic doubling operation as

$$double \triangleq \Lambda X.\, \lambda f : X \to X.\, \lambda x : X.\, f\ (f\ x).$$

The type of this expression is

$$\forall X.\, (X \to X) \to X \to X$$

We can instantiate this on a type, and provide arguments. For example,

$$\begin{aligned} double\ [\mathbf{int}]\ (\lambda n : \mathbf{int}.\, n + 1)\ 7 &\longrightarrow (\lambda f : \mathbf{int} \to \mathbf{int}.\, \lambda x : \mathbf{int}.\, f\ (f\ x))\ (\lambda n : \mathbf{int}.\, n + 1)\ 7 \\ &\longrightarrow^* 9 \end{aligned}$$

Recall that in the simply-typed lambda calculus, we had no way of typing the expression $\lambda x.\, x\ x$. In the polymorphic lambda calculus, however, we can type this expression if we give it a polymorphic type and instantiate it appropriately.

$$\vdash \quad \lambda x : \forall X.\, X \to X.\, x\ [\forall X.\, X \to X]\ x \quad : \quad (\forall X.\, X \to X) \to (\forall X.\, X \to X)$$

1.4 Erasure

The semantics of System F presented above explicitly passes type. In an implementation, one often wants to eliminate types for efficiency. The following translation "erases" the types from a System F expression.

$$\begin{aligned} erase(x) &= x \\ erase(n) &= n \\ erase(\lambda x : \tau.\, e) &= \lambda x.\, erase(e) \\ erase(e_1\ e_2) &= erase(e_1)\ erase(e_2) \\ erase(\Lambda X.\, e) &= \lambda z.\, erase(e) && \text{where } z \notin FV(e) \\ erase(e\ [\tau]) &= erase(e)(\lambda x.\, x) \end{aligned}$$

The following theorem states that the translation is adequate.

Theorem 1 (Adequacy). *For all expressions e and e', we have $e \longrightarrow^* e'$ iff $erase(e) \longrightarrow^* erase(e')$.*

The type reconstruction problem asks whether, for a given untyped λ-calculus expression e' there exists a well-typed System F expression e such that $erase(e) = e'$. It was shown to be undecidable by Wells in 1994, by showing that type checking is undecidable for a variant of untyped λ-calculus without annotations. See Pierce Chapter 23 for further discussion, and restrictions of System F for which type reconstruction is decidable.

In our language, we have explicit annotations for type abstraction ($\Lambda X.\, e$) and type application ($e\ [\tau]$). Given these annotations, then type checking is decidable, and an adaptation of our constraint generating type inference system would work.

1.5 Polymorphism in ML and Java

But in real languages such as ML, programmers don't have to annotate their programs with $\forall X.\ \tau$ or $e\ [\tau]$. Both are automatically inferred by the compiler (although the programmer can specify the former if he wishes). For example, we can write `let double f x = f (f x);;` and Ocaml will figure out that the type is `('a -> 'a) -> 'a -> 'a` (which is roughly equivalent to $\forall A.\ (A \to A) \to A \to A$).

We can also write `double (fun x -> x+1) 7;;`, and Ocaml will infer that the polymorphic function `double` is instantiated on the type `int`.

So what's going on? How can type inference in these languages work?

The polymorphism in ML is not exactly like the polymorphism in System F. ML restricts what types a type variable may be instantiated with. Specifically, type variables can not be instantiated with polymorphic types. Also, polymorphic types are not allowed to appear on the left-hand side of arrows (i.e., a polymorphic type cannot be the type of a function argument). This form of polymorphism is known as *let-polymorphism* (due to the special role played by let in ML), or *prenex polymorphism*. These restrictions ensure that *type inference* is possible.

An example of a term that is typable in System F but not typable in ML is the self-application expression $\lambda x.\, x\, x$. (Try typing `fun x -> x x;;` in the top-level loop of Ocaml, and see what happens...)

Java, as of version 1.5, provides *generics*, a form of parametric polymorphism. For instance, we can write a class that is parameterized on an unknown reference type `T`:

```
class Pair⟨T⟩ {
  T x, y;

  Pair(T x, T y) {
    this.x = x;
    this.y = y;
  }

  T fst(Pair⟨T⟩ p) {
    this.x = p.x;
    return p.x;
  }
}
```

This is a class that contains a pair of two elements of some unknown type T. The parameterization $\forall T$ is implicit around the class declaration. Since Java does not support type inference, type instantiations are required. Type instantiations are done by writing the actual type in angle brackets:

```
Pair⟨Boolean⟩ p;
p = new Pair⟨Boolean⟩(Boolean.TRUE, Boolean.FALSE);
Boolean x = p.fst(p);
```