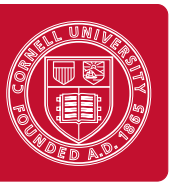
CS 4110 – Programming Languages and Logics
Lecture #19: More Polymorphism



Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

1 Polymorphism in ML and Java

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 she 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`.

The polymorphism in ML is not, however, 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);
```

2 Bounded polymorphism

Polymorphism and subtyping can be combined. That is, we can change type abstraction to include an upper bound on the types that can be used to instantiate the type variable: $\forall X \leq \tau_1.\ \tau_2$. This means that X can only be instantiated with a subtype of τ_1.

Note that if there is a "top type" $\top$ (i.e., every type τ is a subtype of $\top$), then $\forall X.\ \tau$ is equivalent to $\forall X \leq \top.\ \tau$.

Subtyping and parametric polymorphism are largely orthogonal. We need to change the type variable context Δ so that it contains the bounds on type variables, so Δ is now a sequence of elements of the form $X \leq \tau$.

The syntax of expressions, values and types is now as follows.

$$
\begin{aligned}
e &::= n \mid x \mid \lambda x{:}\tau.\ e \mid e_1\ e_2 \mid \Lambda X \leq \tau.\ e \mid e\ [\tau] \\
v &::= n \mid \lambda x{:}\tau.\ e \mid \Lambda X \leq \tau.\ e \\
\tau &::= \textbf{int} \mid \tau_1 \to \tau_2 \mid X \mid \forall X.\ \tau
\end{aligned}
$$

The operational semantics are the same (modulo the minor changes to syntax). Most of the typing rules remain the same. We present the rules for type abstraction, type application, and subsumption. Note that type application requires that the instantiating type is a subtype of the declared bound on the type variable. Note also that the subtyping relation now uses the type variable context Δ.

$$
\frac{\Delta, X \leq \tau_1, \Gamma \vdash e{:}\tau_2}{\Delta, \Gamma \vdash \Lambda X \leq \tau_1.\ e{:}\forall X \leq \tau_1.\ \tau_2}
\qquad
\frac{\Delta, \Gamma \vdash e{:}\forall X \leq \tau_1.\ \tau_2 \quad \Delta \vdash \tau \leq \tau_1}{\Delta, \Gamma \vdash e\ [\tau]{:}\tau_2\{\tau/X\}}
$$

$$
\frac{\Delta, \Gamma \vdash e{:}\tau \quad \Delta \vdash \tau \leq \tau'}{\Delta, \Gamma \vdash e{:}\tau'}
$$

The rule for subtyping a type variable simply uses the type variable context Δ.

$$
\frac{}{\Delta \vdash X \leq \tau}\ X \leq \tau \in \Delta
$$

The subtyping rule for polymorphic types is the most interesting. Going back to the subtyping principle, $\forall X \leq \tau_1.\ \tau_2$ is a subtype of $\forall X \leq \tau_1'.\ \tau_2'$ if whenever a value of type $\forall X \leq \tau_1'.\ \tau_2'$ is expected, a value of type $\forall X \leq \tau_1.\ \tau_2$ can be used instead. Intuitively, an expression of polymorphic type $\forall X \leq \tau_1'.\ \tau_2'$ can be thought of as a function from types to terms. That is, it takes a type as an argument, and returns an expression. (By contrast, functions (also known as term abstractions) that we have seen previously are functions from terms to terms.)

A term of type $\forall X \leq \tau_1'.\ \tau_2'$ may be instantiated with any type that is a subtype of τ_1'; for $\forall X \leq \tau_1'.\ \tau_2'$ to be a subtype, it must also be able to be instantiated with any type that is a subtype of τ_1', so we want τ_1' to be a subtype of τ_1. That is, we are contravariant in the bound on the type variable. Also, when instantiated on any type X that satisfies both constraints (i.e., $X \leq \tau_1$ and $X \leq \tau_1'$), a value produced by an expression of type τ_2 should be usable at type τ_2'. So we want $\Delta, X \leq \tau_1' \vdash \tau_2 \leq \tau_2'$. The inference rule for subtyping on polymorphic types is thus the following.

$$
\frac{\Delta \vdash \tau_1' \leq \tau_1 \quad \Delta, X \leq \tau_1' \vdash \tau_2 \leq \tau_2'}{\Delta \vdash \forall X \leq \tau_1.\ \tau_2 \leq \forall X \leq \tau_1'.\ \tau_2'}
$$