COMP3161/COMP9161 Supplementary Lecture Notes

Parametric Polymorphism

Liam O'Connor-Davis
Gabriele Keller

October 24, 2014

Polymorphism is a prominent part of most modern programming languages. It allows some form of *generic* programming, where values of *different types* can be manipulated by the *same function*.

Parametric polymorphism, sometimes called *generics* in OO languages, is the simplest form of polymorphism[1], where a function can declared to operate over *any type at all*. For example, suppose we had a swap function:

$$\texttt{swap}\ \ x, y = (y, x)$$

What would the type of this function be? In a *monomorphic* language like MinHS, we couldn't write this function generically. We would have to have a variety of functions, $\texttt{swapBI} : \texttt{Bool} \times \texttt{Int} \to \texttt{Int} \times \texttt{Bool}$, $\texttt{swapIB} : \texttt{Int} \times \texttt{Bool} \to \texttt{Bool} \times \texttt{Int}$, and so on - a total of T^2 functions where T is the number of types in the language[2]. This is obviously highly impractical, seeing as all these functions have the same implementation. What we want is to express a type $\texttt{swap} : \alpha \times \beta \to \beta \times \alpha$ for *all* types α and β. That is what parametric polymorphism gives us.

1 Type Parameters

Currently, all functions in MinHS take some values of a concrete type, and return a value of a concrete type. Sometimes in literature, these functions from *values* to *values* are represented as $\lambda(x : \tau).\ y$, where τ is the type of the argument. In MinHs, we write $\texttt{letfun}\ (f : \tau \to \tau')\ x = y$, annotating the function name with a type. This has the advantage that both the argument type τ and the result type τ' are visible.

A function that constructs a pair of two integers could be written with this notation as follows:

$$\texttt{mkIntIntPair} = (\lambda(x : \texttt{Int}).\ (\lambda(y : \texttt{Int}).\ \texttt{Pair}\ x\ y))$$

This function takes an argument x of type Int, and returns a *function*, which, given a y of type Int, will produce a pair of x and y. This nesting of functions is how we achieve n-ary functions in Haskell and similar languages, and is called *currying*.

In order to get parametric polymorphism, we extend functions slightly. In addition to having functions from *values* to *values*, like above, we include functions from *types* to *values*, usually written like $\Lambda\tau.\ v$. These uppercase-Λ binders introduce *type variables*, usually written with greek letters, which can be used in type signatures for values wherever it is in scope. For example, a generic mkPair function could be written like this:

$$\texttt{mkPair} = \Lambda\alpha.\ \Lambda\beta.\ \lambda(x : \alpha).\ \lambda(y : \beta).\ \texttt{Pair}\ x\ y$$

Applying a type to one of these generic functions is called *specialising*, and is written a variety of ways in the literature, including mkPair@Int@Int, mkPair [Int] [Int] or mkPair {Int} {Int}. As a result of the application to the type we get a monomorphic function: here, a pair-function which only works on integers.

Universal Quantification

To give a type to our new $\Lambda\tau.\ e$ form, and thus to our mkPair function, we need to *reflect* the type variables introduced by the Λ on to the type level, where they are introduced by the *universal quantifier*, $\forall$:

[1] And yet, it remains one of the worst-implemented features of all time in C++, and it simply doesn't exist in Go

[2] Since we have products and sums, $T = \infty$

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda\tau.\ e : \forall\alpha.\ \tau}$$

This *generalisation rule* allows us to provide a type to our `mkPair` function.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : \alpha; y : \beta \vdash x : \alpha \quad x : \alpha; y : \beta \vdash y : \beta}{x : \alpha; y : \beta \vdash \texttt{Pair}\ x\ y : \alpha \times \beta}}{x : \alpha \vdash \lambda(y : \beta).\ \texttt{Pair}\ x\ y : \beta \to \alpha \times \beta}}{\vdash \lambda(x : \alpha).\ \lambda(y : \beta).\ \texttt{Pair}\ x\ y : \alpha \to \beta \to \alpha \times \beta}}{\vdash \Lambda\beta.\ \lambda(x : \alpha).\ \lambda(y : \beta).\ \texttt{Pair}\ x\ y : \forall\beta.\ \alpha \to \beta \to \alpha \times \beta}}{\vdash \Lambda\alpha.\ \Lambda\beta.\ \lambda(x : \alpha).\ \lambda(y : \beta).\ \texttt{Pair}\ x\ y : \forall\alpha.\ \forall\beta.\ \alpha \to \beta \to \alpha \times \beta}$$

To type the application of our `mkPair` function, we need a type for the *specialisation* form, $e@\tau$

$$\frac{e : \forall\alpha.\tau'}{e@\tau : \tau'[\alpha := \tau]}$$

This states that we can substitute the type variable α in the type for e with the type after the @ and get a well-typed result. Now we can type a term like `mkPair@Int@Bool 3 True`:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \vdash \texttt{mkPair} : \forall\alpha\beta.\ \alpha \to \beta \to \alpha \times \beta}}{\cdots \vdash \texttt{mkPair@Int} : \forall\beta.\ \texttt{Int} \to \beta \to \texttt{Int} \times \beta}}{\cdots \vdash \texttt{mkPair@Int@Bool} : \texttt{Int} \to \texttt{Bool} \to \texttt{Int} \times \texttt{Bool}} \quad \dfrac{}{\cdots \vdash 3 : \texttt{Int}}}{\cdots \vdash \texttt{mkPair@Int@Bool 3} : \texttt{Bool} \to \texttt{Int} \times \texttt{Bool}} \quad \dfrac{}{\cdots \vdash \texttt{True} : \texttt{Bool}}}{\cdots \vdash \texttt{mkPair@Int@Bool 3 True} : \texttt{Int} \times \texttt{Bool}}$$

This lets us define functions that are *generic* over their arguments, as required, so now let us examine what extensions we need to add to MinHS to make parametric polymorphism possible in MinHS programs.

2 Applying to MinHS

2.1 New Syntax

We introduce two new forms of expression syntax: `type` α `in` e for type abstraction, which corresponds to the $\Lambda\alpha.\ e$ notation from earlier, and `Inst` $e\ \tau$ for the type instantiation of polymorphic functions, which is analogous to type application $e@\tau$.

We also extend type syntax with the universal quantifier `Forall a.` τ and type variables `a`, `b`, etc. This means our static semantics must ensure that types are well formed – that all type variables have an accompanying quantifier. We achieve by keeping track of the variables bound in a quantifier in the set Δ. The predicate OkP defines a superset of Ok which includes types with $\forall$-quantifiers.

$$\frac{t \in \Delta}{\Delta \vdash t\ Ok} \qquad \frac{}{\Delta \vdash \texttt{Int}\ Ok} \qquad \frac{}{\Delta \vdash \texttt{Bool}\ Ok} \qquad \frac{\Delta \vdash \alpha\ Ok \quad \Delta \vdash \beta\ Ok}{\Delta \vdash \alpha \to \beta\ Ok}$$

$$\frac{\Delta \cup \{\texttt{a}\} \vdash \tau\ OkP}{\Delta \vdash \texttt{Forall a.}\ \tau\ OkP} \qquad \frac{\Delta \vdash \tau\ Ok}{\Delta \vdash \tau\ OkP}$$

For example, the type `Forall a. a` $\to$ `b` would not be a valid polymorphic type, since it contains the free type variable b, but `Forall a. Forall b. a` $\to$ `b` is fine. Also, the type `Forall a.` $\to$ `Forall a.` a is not in OkP, even though it doesn't have any free variables, since it has internal quantifiers.

2.2 Typing Rules

The typing rules for `Type` α `in` e are the same as the typing rules for $\Lambda\alpha.\ e$:

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \texttt{Type}\ \tau\ \texttt{in}\ e : \forall\alpha.\ \tau}$$

Similarly for `Inst` $e\ \tau$ and $e@\tau$ respectively:

$$\frac{e : \forall\alpha.\tau'}{\texttt{Inst}\ e\ \tau : \tau'[\alpha := \tau]}$$

2.3 Prenex Restriction

Note that the well-formedness rules for types have been split into two judgements, *Ok* and *OkP*, where polymorphic types are always in *OkP*. This means that τ *Ok* impleis that τ is a monotype. Furthermore, due to the rules defining *OkP*, we have restricted the `Forall` form to the outermost part of a type expression. This means that polymorphic functions are not first class – it is impossible to, for example, type a binding like this:

$$
\begin{gathered}
f :: (\forall \alpha \beta . \alpha \to \beta) \to \mathtt{Int} \to \mathtt{Bool}) \\
f\ x\ i = (\mathtt{Inst}\ (\mathtt{Inst}\ x\ \mathtt{Int})\ \mathtt{Bool})\ i
\end{gathered}
$$

Even though a valid typing derivation can be produced for it. This is because our type-wellformedness rules preclude the possibility of *higher-rank* polymorphism, where quantifiers like $\forall$ can be nested inside other types.

There is no reason for this in the *explicitly*-typed MinHS we have described above, however incorporating higher-rank polymorphism into an *implicitly* typed language with *type inference* becomes very difficult.