COMP3161/COMP9164 Supplementary Lecture Notes

Parametric Polymorphism

Liam O'Connor

November 7, 2019

Polymorphism is a prominent ... of *generic* programming, w... *same function*.

Parametric polymorp... t 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, $\texttt{swapBI} : \texttt{Bool} \times \texttt{Int} \rightarrow \texttt{Int} \times \texttt{Bool}$, $\texttt{swapIB} : \texttt{Int} \times \texttt{Bool} \rightarrow \texttt{Bool} \times \texttt{Int}$, 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 sa... $\texttt{swap} : a \times b \rightarrow b \times a$ for *all* types a an...

1 Type Parameters

Currently, all functions in MinHS take some values of a concrete type, a... concrete type. The literature on typed λ-calculus den... *values* to *values* with the notation $\lambda(x : \tau).\ y$, which is analogous to $\texttt{recfun}\ (f : \tau \rightarrow \tau')\ x = y$ in MinHS. A function that constructs a pair of two integers could be written with this notation as follows:

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

This function takes an argument x of type $\texttt{Int}$, and returns a *function*, which, given a y of type $\texttt{Int}$, 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 $\Lambda t.\ v$. These Λ binders introduce *type variables* which can be used in type signatures for values wherever it is in scope. For example, a generic $\texttt{mkPair}$ function could be written like this:

$$\texttt{mkPair} = \Lambda a.\ \Lambda b.\ \lambda(x : a).\ \lambda(y : b).\ (x, y)$$

Applying a type to one of these generic functions is called *type application* or *specialisation*, and is written a variety of ways in the literature, including $\texttt{mkPair@Int@Int}$, $\texttt{mkPair}\ [\texttt{Int}]\ [\texttt{Int}]$ or $\texttt{mkPair}\ \{\texttt{Int}\}\ \{\texttt{Int}\}$.

[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, $T = \infty$

Universal Quantification

To give a type to our new $\Lambda t.\ e$ 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*, $\forall$:

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

Note that we add a requirement that the introduced variable t is not in the set of free type variables in the environment Γ. This ensures that we don't have clashing type variable names.

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

$$\frac{\dfrac{x : a; y : b \vdash x : a \quad x : a; y : b \vdash y : b}{x : a; y : b \vdash \texttt{pair}(x, y) : a \times b}}{}$$

$\vdash \Lambda$

$\vdash \Lambda a.\ \Lambda$

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

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

This states that we can substitute the type variable a 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`

$\cdots \vdash$

$\cdots \vdash$ `mk`

$\cdots \vdash$ `mkPair`

$\cdots \vdash \quad\quad \cdots \vdash$ `True : Bool`

$\cdots \vdash$ `mkPair@Int@Bool 3`

This lets us define functions that are *generic* over th... et us examine what extensions we need to add to MinHS to make parame... MinHS programs.

2 Applying to MinHS

2.1 New Syntax

We introduce two new forms of expression syntax: **type** $a.\ e$, which is analogous to the $\Lambda a.\ e$ notation from earlier, and $e@\tau$ for type application.

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:

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

$$\frac{\Delta, a\ \textbf{bound} \vdash \tau\ \mathit{Ok}}{\Delta \vdash \texttt{forall a.}\ \tau\ \mathit{Ok}}$$

2.2 Typing Rules

The typing rules for `type` $a.\ e$ are the same as the typing rules for $\Lambda \alpha.\ e$, only using our explicit wellformedness checking for types (which necessitates an additional context of type variables):

$$\frac{a\ \textbf{bound}, \Delta; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \textbf{type}\ a.\ e : \forall a.\ \tau}$$

Similarly for type application:

$$\frac{\Delta; \Gamma \vdash e : \texttt{forall}\ a.\ \tau \quad \Delta \vdash \rho\ \mathbf{ok}}{\Delta; \Gamma \vdash e@\rho : \tau[a := \rho]}$$

2.3 Dynamic Semantics

Firstly, we evaluate the expression in the left-hand side as much as possible:

$$\frac{e \quad \mapsto_M \quad e'}{e@\tau \quad \mapsto_M \quad e'@\tau}$$

Once it has fully evaluated, we expect to see a **type**-abstraction remaining. Then we apply a substitution, similarly to nor

3 Implementation Techniques

Our simple dynamic semantics belies a complex implementation headache.

While we can easily define functions that operate uniformly on multiple types, when this is compiled to machine code the results may differ depending on the *size* of the type in question on the machine stack. For example, a function that takes a `int` parameter will allocate a different amount of stack space than a function that takes a `long` parameter.

There are two mai

Template Speci rphic copies of each polymor f we defined our polymorphic swap function:

$$\begin{aligned} swap = \mathbf{type}\ a\ \mathbf{type} \\ \mathbf{recfun}\ swap \\ p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{aligned}$$

Then a type application like *swap*@`Int`@`Bool` would be replaced *statically* by the compiler with the monomorphic version:

$$\begin{aligned} swap_{\texttt{IB}} = \mathbf{recfun}\ swap :: (\texttt{Int} \times \texttt{Bool}) \rightarrow (\texttt{Bool} \times \texttt{Int}) \\ p = (\mathsf{snd}\ p, \mathsf{fst}\ p) \end{aligned}$$

A new copy is made for each unique type application.

This approach is not too difficult to implement, produces code with minimal run-time overhead, and is relatively easy for a programmer to understand and debug. It also allows the compiler to make optimisations based on the specific types used for a polymorphic function. For example, a list of booleans could be represented more efficiently as a bit vector.

However, as a copy is made for each type application, the binary sizes end up rather large, which can slow compilation times. More importantly, it imposes a severe restriction on the kinds of polymorphism that can be allowed — we must be able to determine all of the possible type instantiations *statically*, which rules out *polymorphic recursion* and other situations where the type instantiations is dependent on run-time information.

Boxing An alternative to the copy-paste-heavy template instantiation approach is to make *all types* represented on the stack in the same way. Thus, a polymorphic function only requires one function in the generated code.

Typically this is done by *boxing* each type. That is, all data types are represented as a *pointer* to a data structure on the heap. If everything is a pointer, then all values use exactly 32 (or 64) bits of stack space.

The extra indirection has a run-time penalty, and it can make garbage collection more necessary, but it results in smaller binaries and unrestricted polymorphism.

4 Generality and Parametricity

If we need a function of type `Int` $\rightarrow$ `Int`, a polymorphic function of type `forall` $a.\ a \rightarrow a$ will do just fine, we can just instantiate the type variable to `Int`. But the reverse is not true. This gives rise to an ordering.

We say that a type τ is *more general* than a type ρ, often written ρ τ, if type variables in τ can be instantiated to give the ty *ost general* type is $\forall a.\ a$, as a could be insta types.

$$\texttt{Int} \rightarrow \texttt{Int} \quad \sqsubseteq \quad \forall z.\ z \rightarrow z \quad \sqsubseteq \quad \forall x\ y.\ x \rightarrow y \quad \sqsubseteq \quad \forall a.\ a$$

As we get more and more general, the number of possible total, terminating implementations of the type gets smaller and smaller:

Type	Size
`Int` $\rightarrow$ `Int`	$(2^{32})^{2^{32}}$

This means that the more general we make our type, the more constrained we are in implementation. This allows us to conclude some things about a function ju signature.

For example, consider this list function *foo*:

$$foo :: \forall a.\ [a] \rightarrow [a]$$

This function has a type that immediately lets me conclude that the output list consists only of elements in the input list. This is because *foo* is defined regardless of what the type a is. Therefore, the only values of type a that *foo* knows about are those values that are in the input. Therefore, the output list can only be made from elements in the input.

The formal principle that captures this intuition is called *parametricity*.

Formally, we can say that if we apply any arbitrary function f to the polymorphically-typed inputs of a polymorphic function, then that is equivalent to applying that same function to the polymorphically-typed outputs of the same function. For example, for *foo* above, the parametricity theorem is, for any f and ls:

$$map\ f\ (foo\ ls) = foo\ (map\ f\ ls)$$

The identity function, $id :: \forall a.\ a \rightarrow a$, has an even simpler theorem:

$$id\ (f\ x) = f(id\ x)$$

There are many other examples, like *head*:

$$head :: \forall a.\ [a] \rightarrow a$$

Which has the following theorem:

$$f\ (head\ \ell) = head\ (map\ f\ \ell)$$

The list appending function:

$$(++) :: \forall a.\ [a] \to [a] \to [a]$$

Which has the following theorem:

$$map\ f\ (a ++ b) = map\ f\ a ++ map\ f\ b$$

Concatenating a list of lists:

$$concat :: \forall a.\ [[a]] \to [a]$$

Has the following theorem:

$$map\ f\ (concat\ ls) = concat\ (map\ (map\ f)\ ls)$$

It even works for higher order fun

Which has the following theorem:

$$filter\ p\ (map\ f\ ls) = map\ f\ (filter\ (p \circ f)\ ls)$$