CS 411 Lecture 22 | Parametric Polymorphism, Refs | 10 Nov 2003
Lecturer: Greg Morrisett

Parametric Polymorphism

Languages such as ML and Haskell include a notion of parametric polymorphism as opposed to subtyping. They are both based on the polymorphic lambda calculus (sometimes called System-F) which is an extension to our simply-typed functional language. The abstract syntax for the polymorphic lambda calculus looks like this:

$$\begin{array}{rl}
\text{(types)} & \tau ::= b \mid \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha.\tau \\
\text{(exprs)} & e ::= c \mid x \mid \lambda x : \tau.\, a \mid e_1\ e_2 \mid \Lambda\alpha.\, e \mid e\ [\tau] \\
\text{(values)} & v ::= c \mid \lambda x : \tau.\, a \mid \Lambda\alpha.\, e
\end{array}$$

At the expression level, we have added two new terms. The first "$\Lambda\alpha.\, e$" represents a function which takes in a type as an argument (α) and then executes the function body e. Here, α, which is a type variable, is a formal parameter. Note that like value functions, a type function is considered a value. The second new expression term is "$e\ [\tau]$" which represents a call to a type function. We are passing the type τ as an actual argument to the type function e.

As a simple example, the identity function can be written as:

$$\mathsf{id} = \Lambda\alpha.\, (\lambda x : \alpha.\, x)$$

This function takes in a type α, and returns a function which when given an α value, returns that value. If we apply id to a type, such as int:

$$\mathsf{id}\ [\mathsf{int}] = (\Lambda\alpha.\, (\lambda x : \alpha.\, x))\ [\mathsf{int}]$$

then this reduces by substituting the actual type parameter (int) for the formal type parameter (α):

$$(\Lambda\alpha.\, (\lambda x : \alpha.\, x))\ [\mathsf{int}] \to \lambda x : \mathsf{int}.\, x$$

Note that we can apply id to any type we want. For instance:

$$\mathsf{id}\ [\mathsf{int} \to \mathsf{int}] \to \lambda x : \mathsf{int} \to \mathsf{int}.\, x$$

Another example of a polymorphic function is apply:

$$\mathsf{apply} : \Lambda\alpha.\, \Lambda\beta.\, (\lambda f : \alpha \to \beta.\, (\lambda x : \alpha.\, (f\ x)))$$

Apply takes in two types (α and β), a function f of type $\alpha \to \beta$, and an argument x of type α. It then applies the function f. So, for instance, we can write:

$$(\mathsf{apply}\ [\mathsf{int}])\ [\mathsf{bool}]$$

and this reduces to:

$$\lambda f : \mathsf{int} \to \mathsf{bool}.\, (\lambda x : \mathsf{int}.\, (f\ x))$$

Again, we can call apply with any two types that we want.

The call-by-name, small-step operational semantics for the polymorphic lambda calculus is given by the following rules:

$$
\begin{array}{ll}
[\mathsf{app}] & (\lambda x:\tau.\, e_1)\ e_2 \to e_1[e_2/x] \\[1em]
[\mathsf{tapp}] & (\Lambda\alpha.\, e)\ [\tau] \to e[\tau/\alpha] \\[1em]
[\mathsf{rator}] & \dfrac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \\[1em]
[\mathsf{trator}] & \dfrac{e_1 \to e_1'}{e_1\ [\tau] \to e_1'\ [\tau]}
\end{array}
$$

When type-checking a polymorphic program, we must keep track of the value variables in scope and their types (as before), but also the type variables that are in scope. So, our contexts will be of the form "$\Delta;\Gamma$" where Δ is the set of type variables in scope, and Γ is a finite map from value variables to types:

$$
\begin{array}{rcl}
\Delta & \in & 2^{\mathsf{TypeVars}} \\
\Gamma & \in & \mathsf{Var} \rightharpoonup \mathsf{Type}
\end{array}
$$

Our typing relation is of the form "$\Delta;\Gamma \vdash e:\tau$" which says that e has type τ under the assumptions of Δ and Γ. The rules are the same as for the simply-typed world except for type functions and type applications:

$$
\begin{array}{lll}
(\mathsf{const}) & \Delta;\Gamma \vdash c : b & \text{(constants have base types)} \\[1em]
(\mathsf{var}) & \Delta;\Gamma \vdash x : \Gamma(x) & \text{(lookup variables in symbol table)} \\[1em]
(\mathsf{abs}) & \dfrac{\Delta;\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \qquad \mathsf{FTV}(\tau_1) \subseteq \Delta}{\Delta;\Gamma \vdash \lambda x:\tau.\, e : \tau_1 \to \tau_2} & (x \notin \Gamma) \\[1em]
(\mathsf{app}) & \dfrac{\Delta;\Gamma \vdash e_1 : \tau' \to \tau \qquad \Delta;\Gamma \vdash e_2 : \tau'}{\Delta;\Gamma \vdash e_1\ e_2 : \tau} & \\[1em]
(\mathsf{tabs}) & \dfrac{\Delta[\alpha];\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda\alpha.\, e : \forall\alpha.\tau} & (\alpha \notin \Delta) \\[1em]
(\mathsf{tapp}) & \dfrac{\Delta;\Gamma \vdash e : \forall\alpha.\tau_1 \qquad \mathsf{FTV}(\tau_2) \subseteq \Delta}{\Delta;\Gamma \vdash e\ [\tau_2] : \tau_1[\tau_2/\alpha]} &
\end{array}
$$

The const, var, abs, and app rules are largely the same as for the simply-typed language. One minor difference is that we need to check function argument types to make sure that they only mention type variables that are in scope. The notation $\mathsf{FTV}(\tau)$ means the set of type variables that occur free in the type τ. The condition "$\mathsf{FTV}(\tau_1) \subseteq \Delta$" thus means that all of the type variables in τ_1 are assumed to be in scope since they are recorded in the assumptions Δ.

We can define the set of free type variables for a type as follows:

$$
\begin{array}{rcl}
\mathsf{FTV}(c) & = & \emptyset \\
\mathsf{FTV}(\tau_1 \to \tau_2) & = & \mathsf{FTV}(\tau_1) \cup \mathsf{FTV}(\tau_2) \\
\mathsf{FTV}(\alpha) & = & \{\alpha\} \\
\mathsf{FTV}(\forall\alpha.\tau) & = & \mathsf{FTV}(\tau) \setminus \{\alpha\}
\end{array}
$$

The tabs rule allows us to assign a type function $\Lambda\alpha.\, e$ a polymorphic type $\forall\alpha.\tau$ if, when we extend the set of type variables in scope to include α, we determine that e has type τ. So, for instance, here's how we might prove that the apply function has type

$$\forall\alpha.(\forall\beta.(\alpha \to \beta) \to (\alpha \to \beta))$$

$$
\text{(tabs)}\,\cfrac{\text{(tabs)}\,\cfrac{\text{(abs)}\,\cfrac{\text{(abs)}\,\cfrac{\text{(app)}\,\cfrac{\{\alpha,\beta\};\{f:\alpha\to\beta,x:\alpha\}\vdash f:\alpha\to\beta \qquad \{\alpha,\beta\};\{f:\alpha\to\beta,x:\alpha\}\vdash x:\alpha}{\{\alpha,\beta\};\{f:\alpha\to\beta,x:\alpha\}\vdash f\ x:\beta}}{\{\alpha,\beta\};\{f:\alpha\to\beta\}\vdash \lambda x:\alpha.\,(f\ x):\alpha\to\beta}}{\{\alpha,\beta\};\emptyset\vdash \lambda f:\alpha\to\beta.\,\lambda x:\alpha.\,(f\ x):(\alpha\to\beta)\to\alpha\to\beta}}{\{\alpha\};\emptyset\vdash \Lambda\beta.\,\lambda f:\alpha\to\beta.\,\lambda x:\alpha.\,(f\ x):\forall\beta.(\alpha\to\beta)\to\alpha\to\beta}}{\emptyset;\emptyset\vdash \Lambda\alpha.\,\Lambda\beta.\,\lambda f:\alpha\to\beta.\,\lambda x:\alpha.\,(f\ x):\forall\alpha.\forall\beta.(\alpha\to\beta)\to\alpha\to\beta}
$$

Working bottom to top, we used the tabs rule twice, then the abs rule twice, then the app rule, and finally the var rule. Notice that when we used the abs rule, the types of the variables mentioned α and β, but that's okay since both α and β are in scope.

The tapp rule allows us to specialize the type of a polymorphic function. In particular, if we have a polymorphic function of type $\forall\alpha.\tau_1$ and we apply that function to the type τ_2, then the resulting expression has type $\tau_1[\tau_2/\alpha]$. For example, if we pass the identity function, which has type $\forall\alpha.\alpha \to \alpha$ the type int, then we get an expression of type $\mathsf{int} \to \mathsf{int}$.

Notice that when we use the tapp rule, we must make sure that we only pass types whose free variables are in scope.

Notes on the Polymorphic Lambda Calculus

It's pretty straightforward to extend our proof of type soundness from the simply-typed lambda calculus to the polymorphic lambda calculus. One important lemma that we need is that substituting a (well-formed) type for a type variable doesn't change the type-ability of an expression:

Lemma 1 (Type Substitution). *If* $\Delta[\alpha];\Gamma \vdash e : \tau_1$ *and* $\mathsf{FTV}(t_2) \subseteq \Delta$ *then* $\Delta;\Gamma[\tau_2/\alpha] \vdash e[\tau_2/\alpha] : \tau_1[\tau_2/\alpha]$.

And, as before, we need to establish a value substitution lemma, a canonical forms lemma, a progress lemma, and a preservation lemma (see the notes on the Simply-Typed Lambda Calculus).

An interesting fact is that every well-typed program terminates, just like in the simply-typed world. This is not at all easy to see. Consider, for instance:

$$\mathsf{id} = \Lambda\alpha.\,\lambda x:\alpha.\,x$$

The identity function has type $\forall\alpha.\alpha \to \alpha$. We can apply the identity function to its type:

$$\mathsf{id}\ [\forall\alpha.\alpha \to \alpha]$$

The resulting program has type: $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$. Using this, we can pass the identity function to itself:

$$(\mathsf{id}\ [\forall\alpha.\alpha \to \alpha])\ \mathsf{id}$$

Of course, this program terminates, but it demonstrates that the polymorphic lambda calculus does admit self-application which seems to be a pre-cursor for building up some way of looping (e.g., fix). However, it turns out that you can't build general recursion using only the pure language.

Surprisingly, you can encode a lot of things directly including booleans, natural numbers, integers, pairs, sums, lists, and trees. You can also code up existential types which are the duals of $\forall$ types. Existential types are useful for modelling object-oriented languages because they give us a way to build up "first-class abstract data types". Although we talked about this a little bit in class, we don't have time to go into much detail about this. If you're interested in this sort of thing, then I can point you to some references.

The Polymorphic Lambda Calculus in the Real World

Languages like ML differ somewhat from the pure polymorphic lambda calculus. First, they don't require you to put in the "∀s" nor do they require that you put in explicit type abstractions (Λ) or type applications. Instead, the compiler figures those out for you through the process of type inference.

That makes the language much more convenient. Wouldn't you hate to write "nil [int]" or worse, "cons [int] 3 x" every time you wanted to build an integer list?

However, this convenience comes at a price. The ML type checker will only infer types of the form:

$$\forall\alpha_1.\ldots\forall\alpha_n.\tau$$

where τ contains no quantifiers. That is, ML forces a prenex quantification restriction on programs that causes all of the ∀'s to be pulled out front. That means, for instance, there is no way to pass a non-instantiated polymorphic function to another function, or to return a non-instantiated polymorphic function, or to put a non- instantiated polymorphic function into a data structure. Another way to say this is that polymorphic functions in ML are second-class.

Usually, we don't notice this because we can do something like pass the identity function to itself. But that's just because what we're really passing is the identity function *instantiated* to some particular type.

New languages, including Java 1.5 and C# include a more powerful form of parametric polymorphism. Instead of requiring that a polymorphic function work for *all* types, in these languages, we can restrict the set of types. For instance, in Java 1.5, it's possible to write a sort method with a type roughly of the form:

$$\forall\alpha \leq \mathsf{Comparable}.\, \mathsf{list}(\alpha) \rightarrow \mathsf{list}(\alpha)$$

This type says that sort can take a list of any objects α that happen to implement the Comparable interface. This make sense because we need to be able to compare the list elements in order to sort them. In ML, you'd have to pass in the comparison function as an extra argument:

$$\forall\alpha.(\alpha \rightarrow \alpha \rightarrow \mathsf{bool}) \rightarrow \mathsf{list}(\alpha) \rightarrow \mathsf{list}(\alpha)$$

The style of polymorphism in Java 1.5 and C# is called **bounded polymorphism** and it combines the best features of subtyping and parametric polymorphism.

References

Adding mutable references to our little functional language is pretty straightforward. We need to combine aspects of our IMP language with our functional language. In particular, we need to use a store to keep track of the value that a ref-cell contains.

Consider the little subset of an ML-like language below:

(types) $\tau ::= \mathsf{int} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2 \mid \mathsf{ref}(\tau)$
(exprs) $e ::= i \mid () \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid l$

We have added a new type, $\mathsf{ref}(\tau)$ to the language which represents reference cells. The C/C++ analog is a τ* (i.e., pointer to a τ). The operation $!e$ reads the contents of the ref cell e. The operation $e_1 := e_2$ changes the contents of e_1's cell to hold the value of e_2.

To model this language, we need to have some way of representing pointers. We could use machine addresses, but to keep things abstract, we'll use a new class of variables that we'll call locations. I'll use l to range over locations:

(locations) $\quad l \in \mathsf{Loc}$

We'll also need to use a store to model the memory of the abstract machine. The store will be a partial function from locations to values:

(values) $\quad v ::= i \mid () \mid \lambda x : \tau.\, e \mid l$
(stores) $\quad s \in \mathsf{Loc} \rightharpoonup \mathsf{Value}$

Here, stores are a lot like the stores in IMP, except that they map locations to values instead of mapping variables to integers. In a language like ML, the variables never change value. So, we can safely substitute their value for the variables. Only the contents of variables can change values.

(Aside: In both Java and C, variables can change value. So, we need to model these languages using a store to map variables to values. Furthermore, in C, every variable maps to a location and you cannot change the variable's location. Rather, you can only change the contents of the location to which a variable refers. The distinctions are subtle, but have a lot of ramifications. In C, an integer variable x and an integer reference y can actually point to the same location in memory (e.g., via $y = \&x$). So, a change to the contents of y's location has the side effect of changing the contents of x's location. In contrast, in Java, an integer reference can never point to the same "location" as an integer variable. That is, there's no way to get the "location" associated with a variable. That's because there need not be such a location! Rather, the integer value is associated with the variable. For instance, we may register-allocate a Java or ML variable. In C, we first have to prove that the variable cannot be aliased before we can register-allocate it.)

Here is a small-step, call-by-value operational semantics for our little language with references. Configurations are of the form (s, e) where s is a store, and e is an expression to be evaluated (much like in IMP):

$$(s, (\lambda x : \tau.\, e)v) \to (s, e[v/x])$$

$$(s, \mathsf{ref}\ v) \to (s[l \mapsto v], l) \qquad (l \notin \mathsf{Dom}(s))$$

$$(s, !l) \to (s, s(l))$$

$$(s, l := v) \to (s[l \mapsto v], ())$$

$$\frac{(s_1, e_1) \to (s_2, e_1')}{\begin{array}{ccc}(s_1, e_1\ e_2) \to (s_2, e_1'\ e_2) & (s_1, e_1 := e_2) \to (s_2, e_1' := e_2) & (s_1, \mathsf{ref}\ e_1) \to (s_2, \mathsf{ref}\ e_1') \\ (s_1, !e_1) \to (s_2, !e_2) & (s_1, v\ e_1) \to (s_2, v\ e_1') & (s_1, v := e_1) \to (s_2, v := e_1')\end{array}}$$

The rules at the bottom force a left-to-right, innermost-to-outermost evaluation order. The application rule at the top is as usual. So the only interesting rules are the ones that manpulate refs. The second rule creates a fresh location, initializes it in the store with the given value, and returns the location as the result. The third rule looks up the value associated with a location in the store. The fourth rule updates the store so that the given location is assigned the new value (and it returns unit as its result.)

Homework

Due Wednesday, 19 November 2003 by 10:00am.

1. Write an interpreter for the little ref language above (extended with a few more constructs.) Use the following definitions:

   ```
   type var = string
   type loc = string

   datatype opn = Plus | Times | Minus | Lte

   datatype value = Bool of bool | Int of int | Unit | Fn of var * exp |
                    Loc of loc

   and exp =      Val of value | Var of var | App of exp * exp
                | Ref of exp | Deref of exp | Assign of exp * exp
                | If of exp * exp * exp | Opn of exp * opn * exp

   type store = loc -> value
   ```

   Your interpreter should have the following signature:

   ```
   val eval : (store * exp) -> (store * value)
   ```

   You may assume that we will only pass well-typed programs to the interpreter.

2. Now consider a little imperative programming language based on a subset of C:

   ```
   datatype iexp = Bool_i of bool | Int_i of int | Opn_i of iexp * opn * iexp |
                   Var_i of string | Addr_i of var | Deref_i of iexp

   datatype icom = Skip_i | Seq_i of icom * icom | If_i of iexp * icom * icom |
                   Assign_i of iexp * iexp

   type iprog = { vars: (var*iexp) list, main: icom }
   ```

   The intention is that variables in this language behave like C. In particular, Addr_i(x) is the notation we have for getting the address of a variable x (in C we would write &x) and Deref_i(e) corresponds to *e. Assign_i(e_1, e_2) corresponds to $e_1 = e_2$ in C. An iprog declares some variables (initializing them to the expressions given) and then executes a command.

   Your task is to write a compiler that translates iprogs to our little ref language above. In particular, you should write a function with the signature:

   ```
   cc : iprog -> exp
   ```

   You should be able to then run the resulting expression using your interpreter from Part 1.

You may assume that we will only manipulate iprogs that are well-typed. So, for instance, initializers for variables will only refer to previously declared variables, and expressions and commands will use variables in a type-consistent manner.

The translation needs to be careful in its treatment of variables. As an example, consider the little program below:

```
int x = 3;
int *y = &x;
int **z = &y;

*y = *y + 1;
**z = **z + 1;
```

At the end of execution, the contents of `x` should be 5.

Make sure to explain your translation using good comments. Write clean and careful code. I will count off for ugly, half thought- through gunk.

Extra Credit: add while loops.