CS251 Programming Languages **Handout #32**
Prof. Lyn Turbak **April 11, 2001**
Wellesley College

POLYMORPHIC TYPES

1. Polymorphic Typed Languages

1.1 Polymorphic Types

In a typed language with **parametric polymorpism**, each expression can potentially be assigned multiple types. Here we develop the notion of polymorphic types and study them in the context of the toy language HOFLEPT: HOFL with Explicit Polymorphic Types. In this section, we informally introduce the explicitly typed HOFLEPT language via a series of examples. In the next section, we formalize the typing rules for HOFLEPT.

As noted in the context of HOFLEMT, monomorphic types can be constraining. Consider a `map` function written in HOFLEPT. To write such a function, we have to fix the element type of the input list and of the output list. For example, we might say that `map` maps an integer list to an integer list.

```
(bindrec ((map (-> ((-> (int) int) (listof int)) (listof int))
            (abs ((f (-> (int) int)) (lst (listof int)))
              body of map)))
  body of bindrec)
```

But sometimes we want to use mapping in the context of other types. For example,
if we want to map a list of integers to a list of booleans we need to define a second mapping function

```
(bindrec ((map-int-bool (-> ((-> (int) bool) (listof int)) (listof bool))
            (abs ((f (-> (int) bool)) (lst (listof int)))
              body of map)))
  body of bindrec)
```

and if we want one that maps boolean lists to boolean lists we need to define a third:

```
(bindrec ((map-bool-bool (-> ((-> (bool) bool) (listof bool)) (listof bool))
            (abs ((f (-> (int) bool)) (lst (listof bool)))
              body of map)))
  body of bindrec)
```

In all three cases, the code labeled *`body of map`* is *exactly* the same. So we essentially have several different copies of a mapping function that differ only in their type annotations. In general, it is bad software engineering practice to make multiple copies of an abstraction. Copying is a tedious process that can introduce bugs. More important, the existence of multiple copies makes it more difficult to change the implementation of the abstraction, since any such change must be made consistently across multiple copies. In practice, such changes are often either made to only some of the copies, or they are simply not made at all.

Software engineering principles suggest an alternative approach: develop some sort of template that captures the commonalities between the different versions of map and an associated mechanism for instantiating the template. We will explore one such approach here, although there are others.

We introduce a new type of the form `(forall (`I_1 `...` I_n`)` T`)` that serves as a template for types that have a similar form. For example, all of the mapping types above can be captured by the following `forall` type:

```
map : (forall (A B) (-> ((-> (A) B) (listof A)) (listof B)))
```

Here, the `A` and `B` are formal type parameters that stand in the place of actual types that will be supplied later. The formal type parameters of a `forall` type serve the same role as the formal parameters of an `abs`, the only difference being that `abs`-bound names stand for values whereas `forall`-bound names stand for types. `Foralls` can be nested just like `abss`, and the type variables they introduce obey the same scoping conventions as those for `abs`-bound variables.

An expression that has a `forall` type is said to designate a **polymorphic value**. The value is polymorphic in the sense that it can have different types in different contexts. In a language with polymorphic values, it is unnecessary to make multiple copies of the `map` function with different types. Instead, there is a mechanism for specifying how a single polymorphic `map` function should be instantiated with particular types.

Let's assume for the moment that there's some way to define a `map` function with the above `forall` type (we'll see how to do this later). Then we need some way to supply actual types for the formal type parameters in the `forall`. This is accomplished by a **polymorphic projection** special form, which has the form (`papp` *E* T_1 ... T_n). Intuitively, `papp` is a declaration that an expression E with `forall` type should in this particular case have its formal type parameters instantiated with the actual types T_1 ... T_n. For example, here are the types of some `papp` expressions involving `map`:

```
(papp map int int) : (-> ((-> (int) int) (listof int)) (listof int))
(papp map int bool) : (-> ((-> (int) bool) (listof int)) (listof bool))
(papp map bool int) : (-> ((-> (bool) int) (listof bool)) (listof int))
(papp map bool bool) : (-> ((-> (bool) bool) (listof bool)) (listof bool))
```

Once we have projected the polymorphic `map` value onto particular types, we can use it as a regular function value. E.g.:

```
((papp map int int) (abs ((x int)) x)
                    (prepend 2 (prepend -3 (prepend 5 (empty int))))
=> (list 4 9 25)

((papp map int bool) (abs ((x int)) (> x 0))
                     (prepend 2 (prepend -3 (prepend 5 (empty int))))
=> (list true false true)

((papp map int bool) (abs ((b bool)) (if b 1 0))
                     (prepend true (prepend false (empty bool)))
=> (list 1 0)

((papp map bool bool) (abs ((b bool)) (not b))
                      (prepend true (prepend false (empty bool)))
=> (list false true)
```

It is an error to attempt to use `map` as a function without first projecting it. For example,

```
(map (abs ((b bool)) (not b)) (prepend #t (empty bool)))
```

is not well typed because the operator should have an arrow type, while `map` has a `forall` type.

As another example, consider the `app5` function in HOFL:

```
(bind app5 (abs (f) (f 5))
  body of bind)
```

In dynamically typed HOFL, the function `f` supplied to `app5` must accept an integer, but it can return any type of value. In HOFLEMT, app5 can return only a single type of value. In a polymorphic version of HOFL, we can define a version of `app5` that has the following type:

```
app5 : (forall (T) (-> ((-> (int) T)) T))
```

Below are some sample uses of the polymorphic `app5` function in the explicitly typed HOFLEPT language (formally described later). (We shall assume that `make-sub` is the function used in the previous section with type `(-> (int) (-> (int) int))).`)

```
((papp app5 int) (abs ((x int)) (* x x))) => 25

((papp app5 bool) (abs ((x int)) (> x 0))) => true

((papp apply int) (make-sub 3)) => 2

(((papp app5 (-> (int) int)) make-sub) 3) -> -2
```

Although we shall not do so in HOFLEPT, the built-in list operations could naturally be characterized with `forall` types:

```
prepend : (forall (T) (-> (T (listof T)) (listof T)))
head : (forall (T) (-> ((listof T)) T))
tail : (forall (T) (-> ((listof T)) (listof T)))
empty? : (forall (T) (-> ((listof T)) bool))
empty : (forall (T) (-> () (listof T)))
```

The only thing we are missing is a way to create polymorphic values, i.e., values of `forall` type. We introduce a new expression construct `(pabs (`I_1 `...` I_n`)` *E*`)` whose only purpose is to convert the value of *E* into a polymorphic value. The `pabs` introduce formal type parameters I_1 `...` I_n that may be referenced within the body expression *E*. For example, here is the definition of `app5`:

```
(bind app5 (pabs (T)
             (abs ((f (-> (int) T)))
               (f 5)))
  body of bind)
```

The type of `app5` in the above expression would be:

```
(forall (T) (-> ((-> (int) T)) T))
```

Note how the parameter `T` introduced by `pabs` can be used within the type expression for `f`. The names introduced by `pabs` are in a different namespace from those introduced by `pabs`: `pabs`-bound names designate types whereas `abs`-bound names designate values. The two namespaces do not interfere with each other. For example, consider the following `test` function:

```
(bind test (abs ((t int))
             (pabs (t)
               (abs ((x t)) t)))
  body of bind)
```

In the expression `(abs ((x t)) t)`, the first `t` refers to the `pabs`-bound variable while the second `t` refers the the `abs`-bound variable.

Now we're ready to see the definition of the polymorphic map function in HOFLEPT:

```
(bindrec ((map (forall (A B) (-> ((-> (A) B) (listof A)) (listof B)))
            (pabs (A B)
              (abs ((f (-> (A) B))
                    (lst (listof A)))
                (if (empty? lst)
                    (empty B)
                    (prepend (f (head lst))
                             ((papp map A B) f (tail lst))))))))
  body of bindrec)
```

The plethora of type annotations make the definition rather difficult to read, but once the polymorphic map function is defined, we can use papp to instantiate map to whatever type we desire.

4.2 Formal Description of HOFLEPT

The formal details of the polymorphic features of the HOFLEPT language are summarized in Figure 3. HOFLEPT has the same type grammar as HOFLEMT except that there is one new type construct (forall). It has the same expression grammar as HOFLEMT except that there are two new expression constructs (pabs and papp). It has the same typing rules as HOFLEMT except that there are two new rules, (pabs) and (papp), which indicate the interplay between the new constructs. The pabs expression is the only form that can produce values of forall type, while the papp expression is the only form that can consume values of forall type. In this respect, pabs, papp and forall share a similar relationship to abs, application, and -> types: abs is the only construct creating values of -> type, and application is the only construct that consumes values of -> type.

New type syntax:

$T \rightarrow$ (forall (I_1 ... I_n) T)

New expression syntax:

$E \rightarrow$ (pabs (I_1 ... I_n) E)

$E \rightarrow$ (papp E T_1 ... T_n)

New type rules:

$$\text{(pabs)}\ \frac{A \vdash E : T \qquad J_1, \ldots, J_n \text{ are not free in } A}{A \vdash (\texttt{pabs}\ (J_1\ \ldots\ J_n)\ E) : (\texttt{forall}\ (J_1\ \ldots\ J_n)\ T)}$$

$$\text{(papp)}\ \frac{A \vdash E : (\texttt{forall}\ (J_1\ \ldots\ J_n)\ T)}{A \vdash (\texttt{papp}\ E\ T_1\ \ldots\ T_n) : T[T_1, \ldots, T_n / J_1, \ldots, J_n]}$$

Figure 3: Summary of the extensions to HOFLEMT that yield HOFLEPT

Two features of the typing rules deserve explanation:

1. In the (papp) rule, the notation $T[T_1, \ldots, T_n / J_1, \ldots, J_n]$ is pronounced "the result of simultaneously substituting T_1 for J_1, ... , and T_n for J_n in T. " The simultaneous substitution process is very similar to that which we studied for the substitution model, except that the substitution is being performed on a type abstract syntax tree rather than an expression abstract syntax tree.

2. In the (pabs) rule, the condition "$J_1 \ldots J_n$ are not free in A" is a subtle technical detail. It turns out that it is safe to abstract over the variables $J_1 \ldots J_n$ in T with a `forall` only if the type variables $J_1 \ldots J_n$ do not appear as free variables in the type bindings in the type environment A. To see why the restriction is necessary, consider the following (contrived) example:

```
(bindrec ((polytest (forall (t) (-> (t) (forall (t) t)))
            (pabs (t)
              (abs ((x t))
                (pabs (t) x)))))
  body of bindrec)
```

In the `forall` type given to `polytest`, the type `t` of `x`, which should reference the outer `t`, has been captured by the inner `t`. The restriction in the (pabs) rule outlaws this sort of name capture.

As a simple illustration of the power of polymorphism, we revisit an example from above that required two copies of the apply-to-5 function in a monomorphic system. In the polymorphic system, it requires only one copy of the apply-to-5 function. Here is the HOFLEPT expression; the type derivation appears below:

```
(bindpar ((app5 (pabs (A) (abs ((f (-> (int) A))) (f 5))))
          (make-sub (abs ((n int)) (abs ((x int)) (- x n)))))
  ((papp app5 int)
   (make-sub (((papp app5 (-> (int) int)) make-sub) 3)))
```

The type derivation uses the following abbreviations:

```
T_{IA} = (-> (int) A)
T_{II} = (-> (int) int)
T_{app5} = (forall (A) (-> (T_{IA}) A))
A_1 = {app5: T_{app5}, make-sub: (-> (int) T_{II})}

      + (var) {f:T_{IA}} |- f : T_{IA}
      + (int) {f:T_{IA}} |- 5 : int
    + (app) {f:T_{IA}} |- (f 5) : A
  + (abs) {} |- (abs ((f T_{IA})) (f 5)) : (-> (T_{IA}) A)
+ (pabs) {} |- (pabs (A) (abs ((f T_{IA})) (f 5))) : T_{app5}
|     + (var) {n:int,x:int} |- x : int
|     + (var) {n:int,x:int} |- n : int
|   + (sub) {n:int,x:int} |- (- x n) : int
| + (abs) {n:int} |- (abs ((x int)) (- x n)) : T_{II}
+ (abs) {} |- (abs ((n int)) (abs ((x int)) (- x n))) : (-> (int) T_{II})
|   + (var) A_1 |- app5 : T_{app5}
| + (papp) A_1 |- (papp app5 int): (-> (T_{II}) int)
| | + (var) A_1 |- make-sub : (-> (int) T_{II})
| | |     + (var) A_1 |- app5: T_{app5}
| | |   + (papp) A_1 |- (papp app5 (-> (int) int)): (-> ((-> (int) T_{II})) T_{II})
| | |   + (var) A_1 |- make-sub: (-> (int) T_{II})
| | | + (papp) A_1 |- ((papp app5 (-> (int) int)) make-sub): T_{II}
| | | + (int) A_1 |- 3: int
| | + (app) A_1 |- (((papp app5 (-> (int) int)) make-sub) 3) : int
| + (app) A_1 |- (make-sub (((papp app5 (-> (int) int)) make-sub) 3)) : T_{II}
+ (app) A_1 |- ((papp app5 int)
             (make-sub (((papp app5 (-> (int) int)) make-sub) 3))) : int
(bindpar) {} |- (bindpar ((app5 (pabs (A)
                                  (abs ((f (-> (int) A))) (f 5))
                          (make-sub (abs ((n int))
                                      (abs ((x int))
                                        (- x n)))))
                  ((papp app5 int)
                   (make-sub (((papp app5 (-> (int) int))
                                make-sub)
                              3))) : int
```