CS251 Programming Languages **Handout # 33**
Prof. Lyn Turbak **March 29, 2002**
Wellesley College

Polymorphic Types

1 HOFLEPT: A Language with Explicit Polymorphic Types

In a typed language with **parametric polymorphism**, 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 **E**xplicit **P**olymorphic **T**ypes. 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.

1.1 The Problem With Monomorphic Types

As noted in the context of HOFLEMT, monomorphic types can be constraining. Consider a `map` function written in HOFLEMT. 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)))
              (if (empty? lst)
                  (empty int)
                  (prepend (f (head lst)) (map f (tail lst)))))))
  body of bindrec)
```

But we also want to perform 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)))
              (if (empty? lst)
                  (empty bool)
                  (prepend (f (head lst)) (map f (tail lst)))))))
  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)))
              (if (empty? lst)
                  (empty bool)
                  (prepend (f (head lst)) (map f (tail lst)))))))
  body of bindrec)
```

In all three cases, the definition of the mapping function is the same except for the explict type annotations. So we essentially have several different copies of a mapping function that differ only

in their types. 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.

1.2 Universal Types via `forall`

We introduce a new type of the form `(forall (`$J_1 \ldots J_n$`)` T`)` that serves as a template for types that have a similar form. Here, J ranges over a collection of **type identifiers**. For example, all of the mapping types above can be captured by a single `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.

By convention, we will write type identifiers with uppercase letters to distinguish them from other identifiers, which we will write in lowercase letters. However, the HOFLEPT language does not enforce this convention.

`Forall`s can be nested just like `abs`s, and the type variables they introduce obey similar scoping conventions to those for `abs`-bound variables. For instance, consider the type

```
(forall (A B) (-> (A) (forall (C) (-> (C) B))))
```

The `c` introduce by the inner `forall` can be renamed to `a` without changing the meaning of the type; the new inner `a` would not conflict with the outer `a`. However, the `c` could not be renamed to `b`, because then the `b` originally appearing in `(-> (c) b)` would be captured.

A type of the form `(forall (`$J_1 \ldots J_n$`)` T`)` is said to be a **universal** type because the type parameters $I_1 \ldots I_n$ range over the universe of types. Does the universe of types over which $I_1 \ldots I_n$ ranges include universal types themselves? If the answer is "yes", the type system is said to be **impredicative**, while if the answer is "no", the type system is said to be **predicative**. In a predicative type system, there is a hierarchy of types: there is a universe U_0 of "regular" types that do not include `forall` types, a universe U_1 that includes `forall` types quantified over the regular types in U_0, a universe U_2 that includes `forall` types quantified over the types in U_1, and so on.

1.3 Instantiating Polymorphic Values Via `papp`

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 type application special form, which has the form

(papp E $T_1 \ldots T_n$).

Intuitively, `papp` specifies that an expression E with `forall` type should in this particular case have its formal type parameters instantiated with the actual types $T_1 \ldots T_n$. 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))
```

The process of instantiating the type variables in a polymorphic value with particular types via `papp` is sometimes called **projection**. For instance, `(papp map int bool)` can be pronounced "the result of projecting the the polymorphic `map` value on `int` and `bool`".

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

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

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

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

hoflept> ((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 true (empty bool)))
```

is not well typed because the operator should have a function (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 HOFLEPTlanguage (formally described later). Assume that `make-sub` is a function with type `(-> (int) (-> (int) int))` that, when given an integer n, returns a "subtract-n" function.

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

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

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

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

Although we shall not handle them this way 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)))
```

1.4 Creating Polymorphic Values Via `pabs`

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 (`$J_1 \ldots J_n$`)` E`)` whose only purpose is to convert the value of E into a polymorphic value. The `pabs` introduces formal type parameters $I_1 \ldots I_n$ 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 (as seen in Sec. 1.3).

Note in the above definition that the recursive call to `map` in the body of `map` is wrapped in a call to `papp`. This is because `map` is defined as a polymorphic value. Alternatively, we can move the recursive definition of the mapping function "underneath" the `pabs` to avoid this projection:

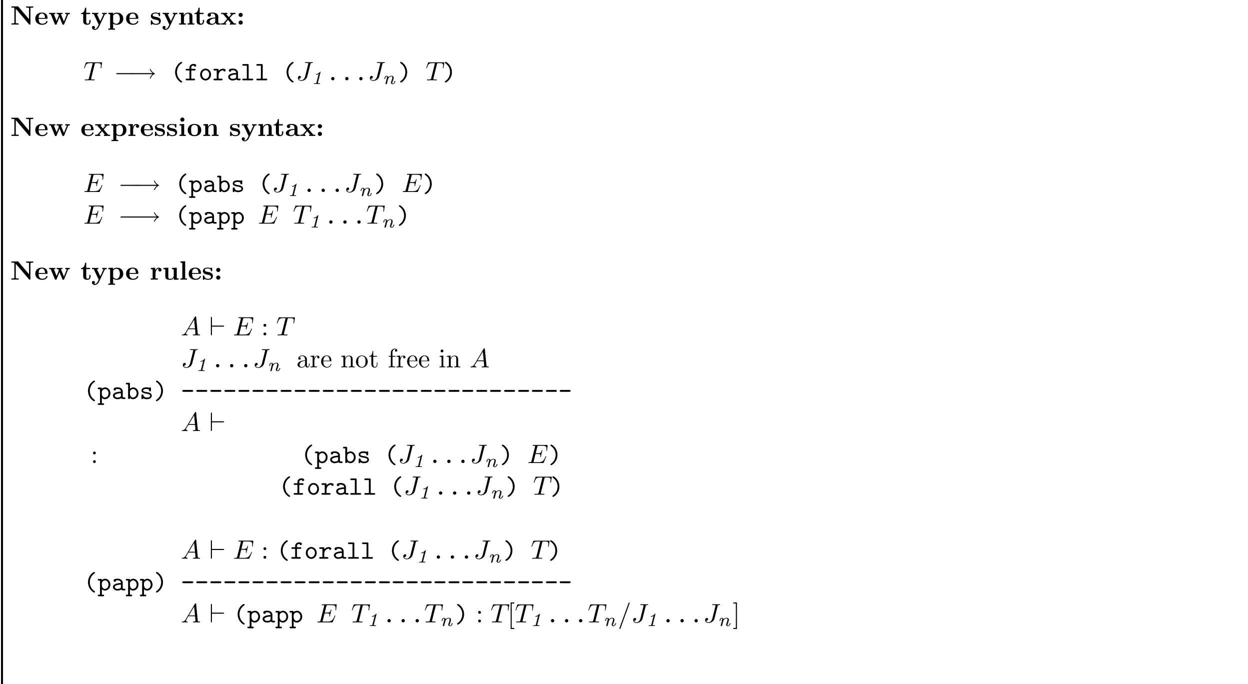
```
(bind map (pabs (A B)
             (bindrec ((inner-map (-> ((-> (A) B) (listof A)) (listof B))
                          (abs ((f (-> (A) B))
                            (lst (listof A)))
                              (if (empty? lst)
                                  (empty B)
                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

2 Formal Description of HOFLEPT

The formal details of the polymorphic features of the HOFLEPT language are summarized in Fig. 1. In words:

- HOFLEPT has the same type grammar as HOFLEMT except that there is one new type construct: `forall`.
- HOFLEPT has the same expression grammar as HOFLEMT except that there are two new expression constructs (`pabs` and `papp`).

- HOFLEMT 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 that produces values of `->` type, while application is the only construct that consumes values of `->` type.



New type syntax:

$T \longrightarrow$ `(forall (`$J_1 \ldots J_n$`)` T`)`

New expression syntax:

$E \longrightarrow$ `(pabs (`$J_1 \ldots J_n$`)` E`)`
$E \longrightarrow$ `(papp` E $T_1 \ldots T_n$`)`

New type rules:

```
         A ⊢ E : T
         J1 ... Jn  are not free in A
(pabs)  ----------------------------
         A ⊢
:                  (pabs (J1 ... Jn) E)
                 (forall (J1 ... Jn) T)

         A ⊢ E : (forall (J1 ... Jn) T)
(papp)  ----------------------------
         A ⊢ (papp E T1 ... Tn) : T[T1 ... Tn/J1 ... Jn]
```

Figure 1: 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 of evaluation, 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.

3 Example

As a simple illustration of the power of polymorphism, we revisit an example from Handout #30 that required two copies of an apply-to-5 function in the monomorphic HOFLEMT language:

```
(bindpar ((app5_1 (abs ((f (-> (int) int))) (f 5))
          (app5_2 (abs ((f (-> (int) (-> (int) int)))) (f 5))
          (make-sub (abs ((n int)) (abs ((x int)) (- x n)))))
  (app5_1 (make-sub ((app5_2 make-sub) 3)))
```

In the polymorphic HOFLEPT language, only one copy of the apply-to-5 function is necessary. Here is the HOFLEPT expression; the type derivation appears in Fig. 2:

```
(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)))
```

Below is a type derivation that uses the following abbreviations:

```
TIA = (-> (int) A)
TII = (-> (int) int)
Tapp5 = (forall (A) (-> (TIA) A))
A1 = {app5: Tapp5, make-sub: (-> (int) TII)}

      + (var) {f:TIA} |- f : TIA
      + (int) {f:TIA} |- 5 : int
    + (app) {f:TIA} |- (f 5) : A
  + (abs) {} |- (abs ((f TIA)) (f 5)) : (-> (TIA) A)
+ (pabs) {} |- (pabs (A) (abs ((f TIA)) (f 5))) : Tapp5
|     + (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)) : TII
+ (abs) {} |- (abs ((n int)) (abs ((x int)) (- x n))) : (-> (int) TII)
|   + (var) A1 |- app5 : Tapp5
| + (papp) A1 |- (papp app5 int): (-> (TII) int)
| | + (var) A1 |- make-sub : (-> (int) TII)
| | |     + (var) A1 |- app5: Tapp5
| | |   + (papp) A1 |- (papp app5 (-> (int) int)): (-> ((-> (int) TII)) TII)
| | |   + (var) A1 |- make-sub: (-> (int) TII)
| | | + (app) A1 |- ((papp app5 (-> (int) int)) make-sub): TII
| | | + (int) A1 |- 3: int
| | + (app) A1 |- (((papp app5 (-> (int) int)) make-sub) 3) : int
| + (app) A1 |- (make-sub (((papp app5 (-> (int) int)) make-sub) 3)) : TII
+ (app) A1 |- ((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
```

Figure 2: Example of polymorphic app5 example in HOFLEPT.