Wellesley College ◊ CS251 Programming Languages ◊ Spring, 2000

TYPES

1. Static Properties of Programs

Programs have both dynamic and static properties. A dynamic property is one that can be determined in general only at run-time by executing the program. In contrast, static properties are those that can be determined without executing the program. Such properties are often determined at compile time by a compiler.

For instance, consider the following Scheme expression:

```
(let ((n (read))   ; Scheme's READ reads a value from user
      (sq (lambda (x) (* x x)))
  (if (integer? n)
      (+ (sq (- n 1)) (sq (+ n 1)))
      0))
```

The value of this expression is a dynamic property, because it cannot be known until run-time what input will be entered by the user. However, there are numerous static properties of this program that can be determined at compile-time:

- The free variables of the expresson are + and *.
- The result of the expression is an a non-negative even integer.
- If the user enters an input, the program is guaranteed to terminate.

A property is only static if it is possible to compute it at run-time. In general, most interesting program properties are uncomputable (e.g., does the program halt? is a particular variable guaranteed to be intialized?). There are two ways to address the problem of uncomputability:

1. Make a conservative approximation to the desired property. E.g., for the halting problem answer either "yes, it halts" or "it may not halt".

2. Restrict the language to the point where it is possible to determine the property unequivocally. Such restrictions reduce the expressiveness of the language.

2. Types

Intuitively, types are sets of values. For instance, Java’s integer type stands for the set of all integers, while the boolean type stands for the set of values true and false. In general, finer grained distinctions might be helpful (e.g. even integers, positive integers), but we will stick with the notion of disjoint types supported by most programming languages.

In Scheme (as well as all the toy languages we have studied thus far this semester), every value carries with it a dynamic type that is only checked by primitive operations. In most other modern programming languages, the type of an expression is a static property, not a dynamic one. Proponents of static types give the following reasons for including them in programming languages:

1. Static types help programmers catch errors in their programs before running them.

2. Static types provide information to the compiler that can eliminate the time associated with run-time type checks and the space required to store run-time types.

3. Static types provide documentation about the program that can facilitate reasoning about the program, both by humans and by other programs (e.g. compilers).

3. Monomorphic Typed Languages

In a monomorphically typed language, each expression can be assigned a single type. Here we consider monomorphic type systems in the context of two languages:

(1) HOFLEMT: HOFL with Explicit Monomorphic Types
(2) HOFLIMT: HOFL with Implicit Monomorphic Types

3.1 HOFLEMT Syntax

Figure 1 presents the grammar for HOFLEMT, a statically-typed version of HOFL with explicit types. The grammar is similar to that for the dynamically-typed HOFL except for a few additions and changes.

The major addition is the introduction of type phrases via the non-terminals `B` and `T`. According to the grammar, a type may be of three different forms:

- ***Base types*** `B` are names that designate the types of HOFL literals:
  3. unit – the type of the one-point set {#u};
  4. bool – the type of the two-point set {#t, #f};
  5. int – the type of integers;
  6. string – the type of strings;

- A ***list type*** of the form `(listof T)` designate lists all of whose elements have type `T`. In HOFLEMT, only ***homogeneous lists*** are supported -- that is, lists in which all elements must be of the same type. For example `(listof int)` designates lists of integers, and `(listof bool)` designates lists of booleans, but it is not possible to have a list that contains both integers and booleans.

- A ***function type*** of the form `(-> (`T_1 `...` T_n`)` T_0`)` designates functions whose *n* arguments, in order, have types T_1 `...` T_n, and whose result has type T_0. For example, an incrementing function on integers would have type `(-> (int) int)`, an addition function on integers would have type `(-> (int int) int)`, and a less-than function on integers would have type `(-> (int int) bool)`.

In HOFLEMT, the syntax of abstractions, recursions, and the empty list primitive application have been extended to include type information:

- In an abstraction `(abs ((`I_1 T_1`) ... (`I_n T_n`))` E`)`, each formal parameter name is paired with the type of that parameter. For example, here is a function that takes an integer and a boolean; it increments the integer if the boolean is true, but doubles it if the boolean is false:

  ```
  (abs ((n int) (b bool))
    (if b (+ n 1) (* n 2)))
  ```

  As another example, consider a function that composes a string to integer function with an integer to boolean function:

  ```
  (abs ((f (-> (int) bool)) (g (-> (string) int)))
    (abs ((x string))
      (f (g x))))
  ```

HOFLEMT Program Grammar

P Program

```
P → (program (I₁ ... Iₙ) E)
```

HOFLEMT Expression Grammar

L Literals

```
L → #u                       ; Unit Literal
L → {#t,#f}                  ; Boolean Literal
L → N                        ; Integer Literal
L → R                        ; String Literal
```

O Primop = {+, -, *, div, mod,

```
            <, <=, =, !=, >=, >,
            band, bor, not,
            prepend, head, tail, empty  ; Note: empty is special
            }
```

E Expressions

```
E → L                                              ; Literal
E → I                                              ; Variable Reference
E → (O E1 ... En)                                  ; Primop Application
E → (empty T)                                      ; Empty List Primapp
E → (abs ((I1 T1) ... (In Tn)) E)                  ; Abstraction
E → (E0 E1 ... En)                                 ; Function Application
E → (if E1 E2 E3)                                  ; Conditional
E → (bindpar ((I1 E1) ... (In En)) E)              ; Parallel bindings
E → (bindrec ((I1 T1 E1) ... (In Tn En)) E) ; Local recursion
```

HOFLEMT Type Grammar

B **BaseType**

```
B → unit                      ; Unit Type (one-point set)
B → bool                      ; Boolean Type (two-point set)
B → int                       ; Integer Type
B → string                    ; String Type
```

T **Type**

```
T → B                         ; Base Type
T → (listof T)                ; List Type (with components of type T)
T → (-> (T1 ... Tn) T0)       ; Function (Arrow) Type
```

Figure 1: Grammar for HOFLEMT

- In a local recursion (`bindrec ((I1 T1 E1) ... (In Tn En)) E`), each binding is annotated with the type of that binding. For example:

```
   (bindrec ((even? (-> (int) bool)
              (abs ((n int))
                (if (= n 0)
                    #t
                    (odd? (- n 1)))))
             (odd? (-> (int) bool)
               (lambda ((n int))
                 (if (= n 0)
                     #f
                     (even? (- n 1))))))
(even? 5))
```

- The empty list primitive application has a type annotation that indicates what type of empty list is being created. For example, `(empty bool)` creates an empty list of booleans, `(empty (-> (int) bool)` creates an empty list of integer predicates, and `(empty (listof int))` creates an empty list of integer lists.

The following HOFLEMT program illustrates all three different kinds of type annotations. The type annotations have been highlighted in bold for emphasis. Make sure you can justify to yourself why all the type annotations are the way they are.

```
      (program (a b m n)
        (bindrec ((from-to (-> (int int) (listof int))
                     (abs ((lo int) (hi int))
                       (if (> lo hi)
                           (empty int)
                           (prepend lo (from-to (+ lo 1) hi)))))
                  (map (-> ((-> (int) (listof bool)) (listof int))
                           (listof (listof bool)))
                    (abs ((f (-> (int) (listof bool)))
                          (lst (listof int)))
                      (if (empty? lst)
                          (empty (listof bool))
                          (prepend (f (head lst))
                                   (map f (tail lst)))))))
      (bind ints (from-to m n)
            (bindpar ((bools1 (map (abs ((n int))
                                     (prepend (> n a)
                                              (prepend (< n b)
                                                       (empty bool))))
                                   ints))
                      (bools2 (map (abs ((n int))
                                     (prepend (> n 0)
                                              (empty bool)))
                                   ints)))
              (prepend bools1 (prepend bools2 (empty (listof (listof bool))))))))
```

As we shall see later, the explicit type annotations in HOFLEMT are designed to support automatic type checking. It turns out that the HOFLEMT annotations are the minimal set of annotations that allow the expression to be type checked via a simple type "evaluator" that "evaluates" each expression to its type. Program parameters do not need to be annotated since they are assumed to be ints. Unlike `bindrec`, the `bind` and `bindpar` constructs do not require the type of the named definition(s) to be given an explicit type. This is because in these constructs the, the type checker can determine the type of the name(s) from the type of the definition(s). In contrast, the recursive scope of `bindrec` makes it generally necessary to know the type of a recursively bound name(s) as part of calculating the type of the associated definition(s).

3.2 HOFLEMT Typing Rules

A HOFLEMT expression `E` is said to be ***well-typed*** if it is possible to prove that it has a type `T` using a set of typing rules. It turns out that for any well-typed HOFLEMT expression `E` that has a type `T`, the run-type value of `E` is guaranteed to be a member of the set of values denoted by `T`. It is impossible to encounter a type error when evaluating a well-typed expression at run-time.

We use the notation `E:T` to indicate that `E` is a well-typed expression with type `T`. For example:

```
#u : unit
#t : bool
5 : int
(prepend 42 (prepend –17 (empty int))) : (listof int)
(abs ((x int)) (> x 0)) : (-> (int) bool)
```

Type environments are environments that associate value variable names with types. For example, the type environment `{a:int, b:bool,f:(-> (int) int)}` associates the name `a` with the type `int`, the name `b` with the type `bool`, and the name `f` with the type `(-> (int) int)`. If A is a type environment, I is a term variable, and T is a type, we use the notation A(I) to denote the type bound to I in type environment A, and A+{I1 : T1,… , In : Tn} to stand for the environment A extended with bindings between I1…In and T1…Tn, respectively.

Just as expressions can be evaluated relative to a value environment, expressions can be typed relative to a type environment. A ***type judgment*** of the form A |- `E : T` is pronounced "Given the type environment A, `E` has type `T`", or, more succinctly, "A proves that `E` has type `T`.

The well-typedness of expressions can be formalized in terms of a set of ***typing rules***. A typing rule has the form

$$(\textit{rulename})\ \frac{\textit{Hypothesis}_1;...;\ \textit{Hypothesis}_n}{\textit{Conclusion}}$$

where each of the hypotheses and conclusions is a typing judgement. Such a rule is pronounced as follows: "If the hypotheses $\textit{Hypothesis}_1$ ... $\textit{Hypothesis}_n$ are all true, then the conclusion *Conclusion* is true." The name *rulename* is just a handy way to refer to a particular rule.

The typing rules for HOFLEMT appear in Figure 2. These rules use the following metavariable conventions:

- A ranges over type environments
- N ranges over numeric literals
- E ranges over expressions
- I ranges over term variables
- T ranges over types

The typing rules in Figure 2 can be used to prove that a given HOFLEMT expression is well-typed. A proof that expression E is well-typed with respect to a type environment A is a tree of type judgements where:

- The root of the tree is A |- E : T for some type T;
- Each judgement J appearing in the tree is justified by instantiating one of the typing rules such that J is the conclusion of the instantiated rule and the children judgements of J are the hypotheses of the instantiated rule.

Such a tree of judgements whose root is the judgement J is said to be a **type derivation** (or **typing**) for J.

$$(\texttt{int})\quad \frac{}{A \vdash N : \texttt{int}}$$ (Other literal rules are similar)

$$(\texttt{var})\quad \frac{}{A \vdash I : A(I)}$$

$$(\texttt{if})\quad \frac{A \vdash E_1 : \texttt{bool};\ A \vdash E_2 : T;\ A \vdash E_3 : T}{A \vdash (\texttt{if}\ E_1\ E_2\ E_3) : T}$$

$$(\texttt{abs})\quad \frac{A + \{I_1 : T_1,\ \ldots,\ I_n : T_n\} \vdash E : T}{A \vdash (\texttt{abs}\ ((I_1\ T_1)\ \ldots\ (I_n\ T_n))\ E) : (\texttt{->}\ (T_1 \ldots T_n)\ T)}$$

$$(\texttt{app})\quad \frac{\begin{array}{l} A \vdash E_0 : (\texttt{->}\ (T_1\ \ldots\ T_n)\ T) \\ A \vdash E_1 : T_1;\ \ldots\ ;\ A \vdash E_n : T_n \end{array}}{A \vdash (E_0\ E_1\ \ldots\ E_n) : T}$$

$$(\texttt{bindpar})\quad \frac{\begin{array}{c} A \vdash E_1 : T_1;\ \ldots;\ A \vdash E_n : T_n \\ A + \{I_1 : T_1,\ \ldots,\ I_n : T_n\} \vdash E : T \end{array}}{A \vdash (\texttt{bindpar}\ ((I_1\ E_1)\ \ldots\ (I_1\ E_n))\ E) : T}$$

The `bind` construct is treated like `bindpar` with a single binding.

$$(\texttt{bindrec})\quad \frac{\begin{array}{c} A_{rec} \vdash E_1 : T_1;\ \ldots;\ A \vdash E_n : T_n \\ A_{rec} \vdash E : T \end{array}}{A \vdash (\texttt{bindrec}\ ((I_1\ T_1\ E_1)\ \ldots\ (I_n\ T_n\ E_n))\ E) : T}$$

where Arec = A + $\{I_1 : T_1, \ldots,, I_n : T_n]$

$$(\texttt{add})\quad \frac{A \vdash E_1 : \texttt{int};\ A \vdash E_2 : \texttt{int}}{A \vdash (+\ E_1\ E_2) : \texttt{int}}$$

There are analogous rules for the other primitive applications. To allow list operations to be polymorphic, we need special rules for list primitives (shown below).

$$(\texttt{prepend})\quad \frac{A \vdash E_1 : T;\ A \vdash E_2 : (\texttt{listof}\ T)}{A \vdash (\texttt{prepend}\ E_1\ E_2) : T}$$

$$(\texttt{head})\quad \frac{A \vdash E : (\texttt{listof}\ T)}{A \vdash (\texttt{head}\ E) : T}$$

$$(\texttt{tail})\quad \frac{A \vdash E : (\texttt{listof}\ T)}{A \vdash (\texttt{tail}\ E) : (\texttt{listof}\ T)}$$

$$(\texttt{empty})\quad \frac{}{A \vdash (\texttt{empty}\ T) : (\texttt{listof}\ T)}$$

$$(\texttt{empty?})\quad \frac{A \vdash E : (\texttt{listof}\ T)}{A \vdash (\texttt{empty?}\ E) : \texttt{bool}}$$

Figure 2: Typing rules for HOFLEMT

For example, consider the term

```
(bind app5 (abs ((f (-> (int) bool)))
             (f 5))
  (app5 (abs ((x int))
          (> x 0)))
```

Suppose that we want to show that this term is well-typed with respect to the empty environment. Because the typing derivation will be a rather wide tree, we will introduce the following abbreviations to make it narrower:

T_{IB} = `(-> (int) bool)`
T_{IBB} = `(-> (`T_{IB}`) bool)`
A_1 = {`f`: T_{IB} }
A_2 = {`app5`: T_{IBB} }
A_3 = {`app5`: T_{IBB}, x:`int`}

Below is a typing derivation for the term that proves that it has type `bool`. Each horizontal line is labeled with the name of the instantiated rule. Note that the leaves of the typing derivation are judgements involving literals or variables; these have no hypotheses. Also note that the "shape" of the derivation is an "upside down" abstract syntax tree for the expression at the root. That is, a judgement for an expression E follows from the judgements of its direct subexpressions.

$$\text{(bind)}\frac{\text{(abs)}\dfrac{\text{(app)}\dfrac{\text{(var)}\dfrac{}{A_1 \mid\!- f : T_{IB}}\;;\;\text{(int)}\dfrac{}{A_1 \mid\!- 5 : \text{int}}\;;}{A_1 \mid\!- (f\ 5) : \text{bool}}\;;}{\{\}\mid\!- (\text{abs}\ ((f\ T_{IB}))\ (f\ 5)) : T_{IBB};}\quad \text{(app)}\dfrac{\text{(var)}\dfrac{}{A_2 \mid\!- \text{app5}: T_{IBB}}\;;\;\text{(abs)}\dfrac{\text{(gt)}\dfrac{\text{(var)}\dfrac{}{A_3 \mid\!- x: \text{int}}\;;\;\text{(int)}\dfrac{}{A_3 \mid\!- 0: \text{int}}}{A_3 \mid\!- (>\ x\ 0))\ :\text{bool}}}{A_2 \mid\!- (\text{abs}\ ((x\ \text{int}))\ (>\ x\ 0)): T_{IB}}}{A_2 \mid\!- (\text{app5}\ (\text{abs}\ ((x\ \text{int}))\ (>\ x\ 0)):\text{bool}}}{\{\}\mid\!- (\text{bind app5}\ (\text{abs}\ ((f\ T_{IB})\ (f\ 5))\ (\text{app5}\ (\text{abs}\ ((x\ \text{int}))\ (>\ x\ 0)))\ :\text{bool}}$$

As shown above, type derivations can be drawn as trees in which all hypotheses for a rule are on the same line above the horizontal bar and the conclusion of a rule is below the horizontal bar. We shall call this the **horizontal format** for a type derivation.

Using the horizontal format, it is very easy to run out of horizontal space when drawing a type derivation. Below, we illustrate an alternative **vertical format** for displaying the above type derivation that makes much better use of horizontal space:

```
    + (var) A_1 |- f : T_IB
    + (int) A_1 |- 5 : int
  + (app) A_1 |- (f 5) : bool
+ (abs) {} |- (abs ((f T_IB)) (f 5)): T_IBB
| + (var) A_2 |- app5 : T_IBB
| |   + (var) A_3 |- x : int
| |   + (int) A_3 |- 0 : int
| | + (gt) A_3 |- (> x 0) : bool
| + (abs) A_2 |- (abs ((x int)) (> x 0)) : T_IB
+  (app) A_2 |- (app5 (abs ((x int)) (> x 0))) : bool
(bind) {} |- (bind app5 (abs ((f T_IB)) (f 5))
                (app5 (abs ((x int)) (> x 0))) : bool
```

In this alternative representation, each conclusion of a rule is labeled with the name of the rule used to derive it, and the hypotheses of the rule are those judgements on the lines labelled "+" directly above the leftmost character of the rule name. Vertical lines are used to connect the hypotheses of the same rule.

Vertical format makes it easier to draw type derivations for more complex expressions using fewer abbreviations without running out of space. For example, below is a type derivation for the following expression:

```
(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)))
```

The type derivation uses the following abbreviations:

```
T_II = (-> (int) int)
A_1 = {app5_1: (-> (T_II) int),
     app5_2: (-> ((-> (int) T_II)) T_II),
     make-sub: (-> (int) T_II)}


    + (var) {f:T_II} |- f : T_II
    + (int) {f:T_II} |- 5 : int
  + (app) {f:T_II} |- (f 5) : int
+ (abs) {} |- (abs ((f T_II) (f 5)) : (-> (T_II) int)
|   + (var) {f:(-> (int) T_II)} |- f : (-> (int) T_II)
|   + (int) {f:(-> (int) T_II)} |- 5 : int
| + (app) {f:(-> (int) T_II)} |- (f 5) : T_II
+ (abs) {} |- (abs ((f (-> (int) T_II))) (f 5)) : (-> ((-> (int) T_II)) T_II)
|     + (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_1 : (-> (T_II) int)
| | + (var) A_1 |- make-sub : (-> (int) T_II)
| | |   + (var) A_1 |- app5_2: (-> ((-> (int) T_II)) T_II)
| | |   + (var) A_1 |- make-sub: (-> (int) T_II)
| | | + (app) A_1 |- (app5 make-sub): T_II
| | | + (int) A_1 |- 3: int
| | + (app) A_1 |- ((app5 make-sub) 3) : int
| + (app) A_1 |- (make-sub ((app5 make-sub) 3)) : T_II
+ (app) A_1 |- (app5 (make-sub ((app5 make-sub) 3))) : int
(bindpar) {} |- (bindpar ((app5_1 (abs ((f T_II)) (f 5))
                          (app5_2 (abs ((f (-> (int) T_II))) (f 5))
                          (make-sub (abs ((n int))
                                      (abs ((x int))
                                        (- x n)))))
                  (app5_1 (make-sub ((app5_2 make-sub) 3))) : int
```

Note that the above derivation contains two separate copies of the apply-to-5 function: one that assumes the argument f has type `(-> (int) int)` and the other that assumes that the argument f has type `(-> (int) (-> (int) int))`. Two separate copies of this function are needed in HOFLEMT because it is a **monomorphic** language: every expression has exactly one type. Since the function is applied at two different argument types, it is necessary to have one copy of the function per argument type.

Examples of real-life monomorphic languages include C, Pascal, and Fortran. As suggested by the above example, in monomorphic languages it may be necessary to create many copies of the same function that differ only in their type. For example, in monomorphic languages, it is necessary to write separate sorting routines for arrays of integers and arrays of floating point numbers because these two arrays have different types! Even worse, in Pascal, the size of the array is part of the array type, so one must write a different sorting function to sort arrays of 10 integers and arrays of 11 integers!

Above we only considered showing that HOFLEMT expressions are well-typed. It is also possible to show that HOFLEMT programs are well-typed. This can be done by showing that the body of the program is well-typed with respect to a type environment where each program parameter is bound to the `int` type.

It is possible to check the well-typedness of a HOFLEMT expression or program via an automatic **type checker**. A type checker is very much like an evaluator, except that rather than finding the type of an expression relative to a value environment, it determines the type of an expression relative to a type environment.

3.3 HOFLIMT

The HOFLIMT language is an implicitly typed version of HOFLEMT. Each HOFLIMT program or expression is the result of erasing the type annotations of a corresponding HOFLEMT program or expression. The typing rules of HOFLIMT are those of HOFLEMT in which the type annotations have been erased from every expression. For example, below is a HOFLIMT type derivation for the following expression:

```
(bind app5 (abs (f) (f 5)) (app5 (abs (x) (> x 0))))

+ (var) A_1 |- f : T_IB
    + (int) A_1 |- 5 : int
  + (app) A_1 |- (f 5) : bool
+ (abs) {} |- (abs (f) (f 5)): T_IBB
| + (var) A_2 |- app5 : T_IBB
| |   + (var) A_3 |- x : int
| |   + (int) A_3 |- 0 : int
| | + (gt) A_3 |- (> x 0) : bool
| + (abs) A_2 |- (abs (x) (> x 0)) : T_IB
+  (app) A_2 |- (app5 (abs (x) (> x 0))) : bool
(bind) {} |- (bind app5 (abs (f) (f 5))
               (app5 (abs (x) (> x 0))) : bool
```

4. Polymorphic Typed Languages

In a polymorphic typed language, each expression can potentially be assigned multiple types. Here we develop the notion of polymorphic types and study them in the context of two toy languages:

(1) HOFLEPT: HOFL with Explicit Polymorphic Types
(2) HOFLIPT: HOFL with Implicit Polymorphic Types

4.1 Polymorphic Types

Here we informally introduce the explicitly typed HOFLEPT language via a series of examples.

As noted above, 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))))
=> (4 9 25)

((papp map int bool) (abs ((x int)) (> x 0))
                     (prepend 2 (prepend -3 (prepend 5 (empty int))))
=> (#t #f #t)

((papp map int bool) (abs ((b bool)) (if b 1 0))
                     (prepend #t (prepend #f (prepend #t (empty bool))))
=> (1 0 1)

((papp map bool bool) (abs ((b bool)) (not b))
                      (prepend #t (prepend #f (prepend #t (empty bool))))
=> (#f #t #f)
```

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))) -> #t

((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:

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

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

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 simultaneouse 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 ... 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 ... J_n in T with a `forall` only if the type variables J_1 ... 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
```

There is an implicitly typed HOFLIPT language that has the same relationship to HOFLEPT as HOFLIMT has to HOFLEMT. The expression of HOFLIPT are the type erasures of those in HOFLEPT, and the typing rules of HOFLIPT are those of HOFLEPT in which all type annotations have been erase from all expressions.

5. Other Typing Issues

In our study of typed versions of HOFL, we have only scratched the surface of type systems, which are currently a hot area of programming language research. Here are a few notes indicating other dimensions of type systems:

- We considered types that described only primitive data, functions, and lists. In general, type systems must handle additional constructs like arrays, records, variants, and exception handling constructs.

- In the above system, the criterion for type compatibility was very narrow. A function with formal parameter type S could only be applied to an argument of type T if S is exactly equal to T. However, in many situations there is a natural notion of **subtyping** whereby it is OK if T is a subtype of S. For instance if a function extracts the field `phone` from a record argument, then all that matters is that the argument is a record with a phone field. Whether it has additional fields is irrelevant. In this case, any record type with a `phone` field would be a subtype of the record type with just a `phone` field. Subtyping plays an important role in typed object-oriented languages. (There are numerous issues of typing in object-oriented languages that have not been resolved.)

- As illustrated by the `map` example from above, explicit types can get very messy. A number of systems have been developed to reduce or eliminate the type annotations specified by the user. Some systems support **implicit projection**, in which the the `papp` construct is not necessary because it can be inferred by the type checker. In **type inference** or **type reconstruction** systems, the programmer writes very few type annotations at all; most are automatically inferred. ML and Haskell are two modern languages that make heavy use of type inference technology. A drawback of systems with type inference is that certain programs cannot be written (because their types cannot be inferred). An alternative approach are **soft typing** systems, which attempt to infer the type of a program but do not prevent the program from being run if a type cannot be found.