CS251 Programming Languages **Handout #28**
Prof. Lyn Turbak **April 4, 2001**
Wellesley College

An Introduction to 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 expression 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 compile-time. In general, most interesting program properties are uncomputable (e.g., does the program halt? is a particular variable guaranteed to be initialized?). 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 `int` type stands for the set of all integers (actually, the set of all integers representable using 32 bits), 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 dynamic type information that is only checked when the value is examined during evaluation. For example:

- Evaluating a primitive application checks that the number and types of the operands are appropriate for the primitive operator.

- Evaluating an `if` expression checks that the test subexpression has boolean type.

- Evaluating a function application checks that the operator is a closure and that the number of actual arguments matches the number of formal parameters expected by the closure.

These sorts of run-time checks are the essence of ***dynamic type checking***.

In most 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. *Safety* : Static types guarantee that type checked programs cannot encounter certain errors at run-time.

2. *Program Development*: Static types help programmers catch errors in their programs before running them and help programmers make representation changes.

3. *Efficiency* : 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.

4. *Documentation*: Static types provide documentation about the program that can facilitate reasoning about the program, both by humans and by other programs (e.g. compilers). Such information is especically valuable in large programs.

3. HOFLEMT: A Language with Monomorphic Types

In a language with ***monomorphic types***, each expression can be assigned a single type. Here we consider monomorphic type systems in the context of the toy language HOFLEMT, a language that extends HOFL with explicit monomorphic types. HOFLEMT is the first of a series of typed toy languages we will study. Just as the toy languages FOFL, FOBS, and HOFL gave us insight into Scheme, the typed toy languages will give us insight into ML.

3.1 HOFLEMT Syntax

Figure 1 presents the grammar for HOFLEMT, a statically-typed version of HOFL with explicit monomorphic 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:
  - `unit` – the type of the one-point set {()};
  - `bool` – the type of the two-point set {true, false};
  - `int` – the type of integers;
  - `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.

HOFLEMT Program Grammar

P **Program**

```
P → (program (I_1 ... I_n) E)
```

HOFLEMT Expression Grammar

L **Literal**

```
L → ()                      ; Unit Literal
L → {true,false}            ; Boolean Literal
L → N                       ; Integer Literal
L → R                       ; String Literal
L → Y                       ; Symbol Literal
```

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

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

E **Expression**

```
E → L                                         ; Literal
E → I                                         ; Variable Reference
E → (O E_1 ... E_n)                           ; Primop Application
E → (empty T)                                 ; Empty List Primapp
E → (abs ((I_1 T_1) ... (I_n T_n)) E)         ; Abstraction
E → (E_0 E_1 ... E_n)                         ; Function Application
E → (if E_1 E_2 E_3)                          ; Conditional
E → (bindpar ((I_1 E_1) ... (I_n E_n)) E)     ; Parallel bindings
E → (bindrec ((I_1 T_1 E_1) ... (I_n T_n E_n)) 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
B → sym                       ; Symbol Type
```

T **Type**

```
T → B                         ; Base Type
T → (listof T)                ; List Type (with components of type T)
T → (-> (T_1 ... T_n) T_0)    ; Function (Arrow) Type
```

Figure 1: Grammar for HOFLEMT

- 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))))
  ```

- In a local recursion `(bindrec ((`I_1 T_1 E_1`) ... (`I_n T_n E_n`)) `E`)`, each binding is annotated with the type of that binding. For example:

  ```
  (bindrec ((even? (-> (int) bool)
              (abs ((n int))
                (if (= n 0)
                    true
                    (odd? (- n 1)))))
           (odd? (-> (int) bool)
             (abs ((n int))
               (if (= n 0)
                   false
                   (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.

3.2 Example

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).

3.3 Representing HOFLEMT Programs in SML

Figure 2 shows how the grammar of HOFLEMT can be expressed using Standard ML data types. Except for the addition of types, the data types are almost exactly the same as those we studied for the dynamically typed HOFL language.

```
 datatype Lit =
   UnitLit
 | IntLit of int
 | BoolLit of bool
 | StringLit of string
 | SymLit of string

 datatype Primop =
   Add | Sub | Mul | Div | Mod      (* arithmetic ops *)
 | LT | LEQ | EQ | NEQ | GT | GEQ   (* relational ops *)
 | Band | Bor | Not                 (* logical ops *)
 | SymEq                            (* symbol ops *)
 | IsEmpty | Prepend | Head | Tail  (* list ops *)

 datatype Type =
   UnitTy | IntTy | BoolTy | StringTy | SymTy (* base types *)
 | ListTy of Type                             (* list types *)
 | ArrowTy of Type list * Type                (* function types *)

 datatype Exp =
   Lit of Lit
 | VarRef of Id
 | PrimApp of Primop * Exp list          (* rator, rands *)
 | PrimEmpty of Type                     (* special exp for typed empty list *)
 | If of Exp * Exp * Exp                 (* test, then, else *)
 | Abs of Id list * Type list * Exp      (* formals, formalTypes, body *)
 | FunApp of Exp * Exp list              (* names, defns, body *)
 | BindPar of Id list * Exp list * Exp   (* names, defns, body *)
 | BindRec of Id list * Type list
            * Exp list * Exp             (* names, types, defns, body *)

datatype Program = Prog of Id list * Exp (* formals, body *)
```

Figure 2: Standard ML data types for HOFLEMT