CS251 Programming Languages **Handout # 30**
Prof. Lyn Turbak **Mar. 26, 2002**
Wellesley College

Type Checking

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.
- A **static property** is one that can be determined without executing the program. Static 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 that uncomputability is handled in practice:

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, but in return give precise static information. The ML language is an example of this approach; in order to provide static type information, it forbids many programs that would not give run-time type errors.

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:

- *Safety:* Static types guarantee that type checked programs cannot encounter certain errors at run-time.
- *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.
- *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.
- *Program Development:* Static types help programmers catch errors in their programs before running them and help programmers make representation changes.

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 and interpretation, the typed toy languages will give us insight into ML, type checking, and type reconstruction.

3.1 HOFLEMT Syntax

Fig. 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:

1. **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;
    - `sym` - the type of symbols;

2. 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.

3. A **function type** of the form `(-> (`$T_1 \ldots T_n$`)` T_0`)` designates functions whose n arguments, in order, have types $T_1 \ldots T_n$, 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:

$P \in \mathit{Program}$

$P \rightarrow$ `(program (`$I_{formal_1} \ldots I_{formal_n}$`)` E_{body}`)`	Program

$E \in \mathit{Expression}$

Kernel Expressions:

$E \rightarrow L$	Literal
$E \rightarrow I$	Variable Reference
$E \rightarrow$ `(`O_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`	Primop Application
$E \rightarrow$ `(empty` T`)`	Empty List Primapp
$E \rightarrow$ `(if` E_{test} E_{then} E_{else}`)`	Conditional
$E \rightarrow$ `(bindpar ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Parallel Binding
$E \rightarrow$ `(bindrec ((`I_{name_1} T_1 E_{defn_1}`) ... (`I_{name_n} T_n E_{defn_n}`))` E_{body}`)`	Local Recursion
$E \rightarrow$ `(abs ((`I_1 T_1`) ... (`I_n T_n`))` E_{body}`)`	Abstraction
$E \rightarrow$ `(`E_{rator} $E_{rand_1} \ldots E_{rand_n}$`)`	Function Application

Sugar Expressions:

$E \rightarrow$ `(scand` E_1 E_2`)`	Short-Circuit And
$E \rightarrow$ `(scor` E_1 E_2`)`	Short-Circuit Or
$E \rightarrow$ `(cond (`E_{test_1} E_{body_1}`) ... (`E_{test_n} E_{body_n}`) (else` $E_{default}$`))`	Multi-branch Conditional
$E \rightarrow$ `(bind` I_{name} E_{defn} E_{body}`)`	Local Binding
$E \rightarrow$ `(bindseq ((`I_{name_1} E_{defn_1}`) ... (`I_{name_n} E_{defn_n}`))` E_{body}`)`	Sequential Binding

$L \in \mathit{Literal}$

$L \rightarrow N$	Numeric Literal
$L \rightarrow B$	Boolean Literal
$L \rightarrow Y$	Symbolic Literal

$O \in$ *Primitive Operator*: e.g., `+`, `<=`, `band`, `not`, `prepend`
$F \in$ *Function Name*: e.g., `f`, `sqr`, `+-and-*`
$I \in$ *Identifier*: e.g., `a`, `captain`, `fib_n-2`
$N \in$ *Integer*: e.g., `3`, `-17`
$B \in$ *Boolean*: `true` and `false`
$Y \in$ *Symbol*: e.g., `(symbol a)`, `(symbol captain)`, `(symbol fib_n-2)`

$B \in \mathit{BaseType}$

$B \rightarrow$ `unit`	Unit Type (one-point set)
$B \rightarrow$ `bool`	Boolean Type (two-point set)
$B \rightarrow$ `int`	Integer Type
$B \rightarrow$ `string`	String Type
$B \rightarrow$ `sym`	Symbol Type

$T \in \mathit{Type}$

$T \rightarrow B$	Base Type
$T \rightarrow$ `(listof` T`)`	List Type (with components of Type T)
$T \rightarrow$ `(-> (`$T_1 \ldots T_n$`)` T_0`)`	Function (Arrow) Type

Figure 1: Grammar for the monomorphically typed HOFLEMT language.

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

- 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)
                  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 HOFLEMT program in Fig. 2 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)))))))))
```

Figure 2: Example of HOFLEMT program.

As we shall see below, 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 integers. 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

Fig. 3 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 3: Standard ML data types for HOFLEMT.

4 Type Checking

4.1 Well-Typedness

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 HOFLEMT satisfies a type soundness theorem:

Theorem 4.1 (Type Soundness). *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.* □

The type soundness theorem means that it is impossible to encounter dynamic type errors when evaluating a well-typed expression at run-time. The type soundness theorem is often summed up by the motto "Well-typed programs do not go wrong". This motto is somewhat deceptive – well-typed programs can encounter errors at run-time, but those errors cannot be type errors. Other errors that can still be encountered are errors that depend on particular values (e.g. divide-by-zero, attempt to take the head of an empty list, accessing an array at an out-of-bounds index) as well as logical errors in the program (it gives the wrong answer).

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

```
():unit
true:bool
5:int
"foo":string
(symbol cs251):sym
(prepend 42 (prepend -17 (empty int))):(listof int)
(abs ((a int) (b int)) (div (+ a b) 2)):(-> (int int) bool)
```

Type environments are environments that associate value variable names with types. We will write type environments as sets of bindings of the form $E{:}T$. 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 an identifier, and T is a type, we use the notation $A(I)$ to denote the type bound to I in type environment A, and $A+\{I_1{:}T_1, \ldots, I_n{:}T_n\}$ to stand for the environment A extended with bindings between $I_1 \ldots I_n$ and T_1n, respectively.

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

4.2 Proving Expressions Well-Typed

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

```
          Hypothesis_1; ... ; Hypothesis_n
(rulename)---------------------------------
                   Conclusion
```

where each of the hypotheses and conclusions is a typing judgement. Such a rule is pronounced as follows: "If the hypotheses `Hypothesis_1` ... `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 Fig. 4. These typing rules 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 $A \vdash E : T$ 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.

For example, consider the expression

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

Suppose that we want to show that this expression 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:

```
TIB = (-> (int) bool)
TIBB = (-> (TIB) bool)
Eabsf = (abs ((f TIB)) (f 5))
Eabsx = (abs ((x int)) (> x 0))
Ebind = (bind app5 Eabsf (app5 Eabsx))
A1 = f: TIB
A2 = app5: TIBB
A3 = app5: TIBB, x:int
```

Below is a typing derivation for the expression 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.

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.

Figure 4: Type rules for the HOFLEMT language.

Figure 4: Type rules for the HOFLEMT language.

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 forma**t for displaying the above type derivation that makes much better use of horizontal space:

```
      + (var) A1 |- f : TIB
      + (int) A1 |- 5 : int
    + (app) A1 |- (f 5) : bool
  + (abs)  |- (abs ((f TIB)) (f 5)): TIBB
  | + (var) A2 |- app5 : TIBB
  | |   + (var) A3 |- x : int
  | |   + (int) A3 |- 0 : int
  | | + (gt) A3 |- (> x 0) : bool
  | + (abs) A2 |- (abs ((x int)) (> x 0)) : TIB
  +  (app) A2 |- (app5 Eabsx) : bool
  (bind)  |- (bind app5 Eabsf (app5 Eabsx)) : 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.

The vertical format makes it easier to draw type derivations for more complex expressions using fewer abbreviations without running out of space. For example, Fig. 5 shows 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:

```
  TII  = (-> (int) int)
  A1 = app5_1: (-> (TII) int),
       app5_2: (-> ((-> (int) TII)) TII),
       make-sub: (-> (int) TII)
```

Note that the above derivation contains two separate copies of the `app5` 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!

```
    + (var) f:TII |- f : TII
    + (int) f:TII |- 5 : int
  + (app) f:TII |- (f 5) : int
+ (abs)  |- (abs ((f TII) (f 5)) : (-> (TII) int)
|   + (var) f:(-> (int) TII) |- f : (-> (int) TII)
|   + (int) f:(-> (int) TII) |- 5 : int
| + (app) f:(-> (int) TII) |- (f 5) : TII
+ (abs)  |- (abs ((f (-> (int) TII))) (f 5)) : (-> ((-> (int) TII)) TII)
|     + (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_1 : (-> (TII) int)
| | + (var) A1 |- make-sub : (-> (int) TII)
| | |   + (var) A1 |- app5_2: (-> ((-> (int) TII)) TII)
| | |   + (var) A1 |- make-sub: (-> (int) TII)
| | | + (app) A1 |- (app5 make-sub): TII
| | | + (int) A1 |- 3: int
| | + (app) A1 |- ((app5 make-sub) 3) : int
| + (app) A1 |- (make-sub ((app5 make-sub) 3)) : TII
+ (app) A1 |- (app5 (make-sub ((app5 make-sub) 3))) : int
(bindpar)  |- (bindpar ((app5_1 (abs ((f TII)) (f 5))
                          (app5_2 (abs ((f (-> (int) TII))) (f 5))
                          (make-sub (abs ((n int))
                                      (abs ((x int))
                                        (- x n)))))
                  (app5_1 (make-sub ((app5_2 make-sub) 3))) : int
```

Figure 5: Example type derivation using the vertical format.

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.

4.3 Type Checking

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.

Figs. 6–7 present an SML implementation of a type checker for HOFLEMT. The core of the type checker is the `checkExp` function defined in Fig. 6, whose SML type is:

```
val checkExp : AST.Exp -> Type Ident.Env.env -> Type
```

The `checkExp` function encodes all the typing rules from Fig. 4 except for the rules that handled primitives. It calculates the type of an expression from the types of its subexpressions. If the subexpression types do not match the typing rules, `checkExp` raises a `TypeCheckError` exception indicating that the expression is not well typed.

The type checking of primitive applications is specified in the `PrimopEnv` structure (not shown in the figures). This module has the following signature:

```
signature PRIMOP_ENV = sig

    exception PrimTypeCheckError of string
    exception PrimEvalError of string

    datatype PrimDesc =
      PDesc of   Primitive.Primop              (* name of primitive *)
               * (Type.Ty list -> Type.Ty)     (* type checker *)
               * (Value.Val list -> Value.Val) (* meaning of primop *)

    val lookup : Primitive.Primop -> PrimDesc

end
```

The `PrimDesc` datatype is used to encode the type checking rules and evaluation rules of primitive operators. Fig. 8 shows a few representative examples of the primitive descriptors for HOFLEMT. The `typeMismatch` function (not shown) raises a `PrimTypeErrorException` with an appropriate explanation of the mismatch.

The type of a program is found by the `checkProg` function in Fig. 7, whose SML type is:

```
val checkProg : AST.Program -> Type
```

The `checkProg` function returns the type of the body of a program under the assumption that all the arguments of the program are integers. Like `checkExp`, it raises a `TypeCheckError` exception if the program is not well-typed.

```
fun checkExp (Lit(UnitLit)) env = UnitTy
  | checkExp (Lit(IntLit(_))) env = IntTy
  | checkExp (Lit(BoolLit(_))) env = BoolTy
  | checkExp (Lit(StringLit(_))) env = StringTy
  | checkExp (Lit(SymLit(_))) env = SymTy

  | checkExp (VarRef(name)) env =
      (case TEnv.lookup(name, env) of
         NONE => raise TypeCheckError
                       ("Unbound variable: " ^ (Ident.toString(name)))
       | SOME(ty) => ty)
  | checkExp (exp as If(test,thenExp,elseExp)) env =
      let val testTy = checkExp test env
          val thenTy = checkExp thenExp env
          val elseTy = checkExp elseExp env
       in if not(Type.equal(testTy,BoolTy)) then
             raise TypeCheckError("if: non-boolean test expression")
          else if not(Type.equal(thenTy,elseTy)) then
            raise TypeCheckError("if: branch types don't match:\n"
                                 ^ "Then type: " ^ (Type.toString(thenTy))
                                 ^ "\nElse type: " ^ (Type.toString(elseTy)))
          else
             thenTy
      end

  | checkExp (Abs(formals,types,body)) env =
      ArrowTy(types, checkExp body (TEnv.extend(formals,types,env)))

  | checkExp (FunApp(rator, rands)) env =
      typeApply (checkExp rator env) (checkExpList rands env)


  | checkExp (PrimEmpty(ty)) env = ListTy(ty) (* special prim in HOFLEMT *)
  | checkExp (exp as (PrimApp(primop,rands))) env =
      let val PrimopEnv.PDesc(_,primCheck,_) = PrimopEnv.lookup(primop)
       in primCheck (checkExpList rands env)
          handle PrimopEnv.PrimTypeCheckError(msg) =>
                   raise TypeCheckError(msg)
        end

  | checkExp(BindPar(names,defns,body)) env =
      checkExp body (TEnv.extend(names, checkExpList defns env, env))

  | checkExp (BindRec(names,tys,defns,body)) env =
      let val recEnv = TEnv.extend(names,tys,env)
          val defnTys = checkExpList defns recEnv
       in case ListOps.some3 (fn(name,ty,defnTy) =>
                                not (Type.equal(ty,defnTy)))
                            names tys defnTys of
            NONE => checkExp body recEnv
          | SOME(name,ty,defnTy) =>
            raise TypeCheckError
                  ("bindrec: binding type doesn't match definition type:\n"
                   ^ "binding name: " ^ (Ident.toString name)
                   ^ "\nbinding type: " ^ (Type.toString ty)
                   ^ "\ndefinition type: " ^ (Type.toString defnTy))
      end

and checkExpList exps env = map (fn exp => checkExp exp env) exps
```

```
signature TYPE_CHECK = sig

  exception TypeCheckError of string
    (* Exception raised when type checking error encountered *)

  val checkProg : AST.Program -> Type
    (* Returns the type of a well-typed program. Raises TypeCheckError
       if the program is not well-typed. *)

  val checkExp : AST.Exp -> Type Ident.Env.env -> Type
    (* Returns the type of a well-typed expression relative to the
       given type environment. Raises TypeCheckError if the expression
       is not well-typed relative to the type environment *)
end

structure TypeCheck : TYPE_CHECK = struct

  local open AST Type in

    exception TypeCheckError of string

    structure TEnv = Ident.Env (* abbreviation *)

    fun checkProg(Prog(formals,body)) =
          checkExp body (TEnv.extend(formals,
                                     List.map (fn _ => IntTy) formals,
                                     TEnv.empty))

    and checkExp ... (* definition given in Figure 3 *)

    and checkExpList ... (* definition given in Figure 3 *)

    and typeApply (ratorTy as (ArrowTy(formalTys,resultTy))) actualTys =
          if not (List.length(formalTys) = List.length(actualTys)) then
            raise TypeCheckError
                ("funapp: mismatch between number of formals ("
                 ^ (Int.toString (List.length(formalTys)))
                 ^ ") and number of actuals ("
                 ^ (Int.toString (List.length(actualTys)))
                 ^ ")")
          else (case ListOps.some2 (fn(fty,aty) => not(Type.equal(fty,aty)))
                                   formalTys
                                    actualTys of
                 NONE => resultTy
                | SOME(fty,aty) =>
                    raise TypeCheckError
                      ("funapp: formal type doesn't match actual type.\n"
                       ^ "Expected: " ^ (Type.toString fty)
                       ^ "\nActual: " ^ (Type.toString aty))
                )
      | typeApply ratorTy _ =
          raise TypeCheckError
                ("funapp: attempt to apply non function --\n"
                 ^ "Rator type: " ^ (Type.toString ratorTy))
   end (* local *)
end (* struct *)
```

Figure 7: SML definition of HOFLEMT type checker.

```
    (* Primitive descriptor for + *)
  PDesc(Add,
            fn [IntTy,IntTy] => IntTy
           | tys => typeMismatch(Add, [IntTy,IntTy], tys),
            fn [IntVal(i1), IntVal(i2)] => IntVal(i1 + i2)
        )

    (* Primitive descriptor for < *)
    PDesc(LT,
            fn [IntTy,IntTy] => BoolTy
           | tys => typeMismatch(LT, [IntTy,IntTy], tys),
            fn [IntVal(i1), IntVal(i2)] => BoolVal(i1 < i2)
        )

    (* Primitive descriptor for prepend *)
  PDesc(Prepend,
            fn [ty1,ListTy(ty2)] =>
             if Type.equal(ty1,ty2) then
               ListTy(ty2)
             else
               raise PrimTypeCheckError
                   ("prepend: type of prepended element does not\n"
                    ^ "match component type of list\n"
                    ^ "Prepended element type: " ^ (Type.toString ty1)
                    ^ "\nList component type: " ^ (Type.toString ty2)
                    ^ "\n")
           | tys => raise PrimTypeCheckError
                     ("prepend: wrong argument types "
                      ^ (typeListToString(tys))),
          fn [x,ListVal(xs)] => ListVal(x::xs)
        )

    (* Primitive descriptor for head *)
    PDesc(Head,
            fn [ListTy(ty)] => ty
         | tys => raise PrimTypeCheckError
                     ("head: wrong argument types "
                      ^ (typeListToString(tys))),
          fn [ListVal([])] => raise PrimEvalError
                               ("attempt to take head of empty list")
         | [ListVal(x::xs)] => x
        )
```

Figure 8: Sample primitive descriptors from PrimopEnv.