CS251 Programming Languages **Handout #29**
Prof. Lyn Turbak **April 6, 2001**
Wellesley College

TYPE CHECKING

3.1 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***:

> 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 a type error 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+{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 judgement*** 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*.

3.2 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

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

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 identifiers
- *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*.

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:

T_{IB} = `(-> (int) bool)`
T_{IBB} = `(-> (`T_{IB}`) bool)`
E_{absf} = `(abs ((f `T_{IB}`)) (f 5))`
E_{absx} = `(abs ((x int)) (> x 0))`
E_{bind} = `(bind app5 `E_{absf}` (app5 `E_{absx}`))`
A_1 = {`f:` T_{IB} }
A_2 = {`app5:` T_{IBB} }
A_3 = {`app5:` T_{IBB}, `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.

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

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

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

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

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

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

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

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

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

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

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

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).

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

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

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

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

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

Figure 1: Typing rules for HOFLEMT

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) 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.

Vertical format makes it easier to draw type derivations for more complex expressions using fewer abbreviations without running out of space. For example, Figure 2 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: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
```

Figure 2: 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.

3.3 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.

Figures 3 and 4 present an SML implementation of a type checker for HOFLEMT. The core of the type checker is the `checkExp` function defined in Figure 3, whose SML type is:

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

The `checkExp` function encodes all the typing rules from Figure 1 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.

```
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
```

Figure 3: definitions of type checking functions `checkExp` and `checkExpList`.

```
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 4: SML definition of HOFLEMT type checker.

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. Figure 5 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 Figure 4, 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.

```
(* 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 5: Sample primitive descriptors from `PrimopEnv`.