Standard ML Type Inference

8

Many language implementations, like C++ and Java, check the types of values and operations to be sure each operation is supported for the types of its operands. An important feature of Standard ML is the type inference system which is somewhat like the type checkers of C++ and Java, but a bit more powerful. A type checker checks the types written by the programmer to be sure each type declaration is consistent with the operations being performed, values being passed to functions, and the values being returned. Compilers for Java and C++ even infer the types of some expressions when polymorphic operators are used. For instance, the addition operator has multiple result types depending on the types of its operands.

The type inference system of Standard ML distinguishes itself from other type inference systems by *inferring* almost all the types of an SML program, rather than requiring the programmer to declare the types of variables. The SML type inference system *infers* the types of values in its programs by using type information about constant values and the types supported by its built-in operators or functions. Many of the functions in Standard ML are polymorphic allowing more than one type of argument to be passed to them. The type inference system of Standard ML is able to handle this polymorphism. Robin Milner, Roger Hindley, and Luis Damas all contributed to this powerful polymorphic type inference system.

This chapter develops a polymorphic type inference system for the Small language using Prolog as the implementation language. A typical way to describe type inference is with type inference rules. Each of the type inference rules associated with the Small language is presented along with some of the type inference rule implementations. Not all code is provided since some problems are left as exercises for the reader, but the Prolog examples in this chapter come from a working type inference system for the Small subset of Standard ML.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_8

8.1 Why Static Type Inference?

To motivate our discussion, consider the program found in Fig. 8.1. This is a valid Small program and when compiled to JCoCo and run it prints 1 to the screen. Contrast that to the program in Fig. 8.2, which is not a valid Small program or Standard ML program. It is missing the dereference operator in the expressions referring to x. This program should not execute. Executing such a program would, at best, have unpredictable results. With the target language as the JCoCo virtual machine the program actually does run and produces 1 as its output, which is even worse than it not running at all. Any change in the compiler could end up breaking this program when at one time it seemed to compile and run successfully.

8.1.1 Exception Program

Here is another example. This question was posted on stackoverflow.com. The question posed was,

When executing the code [from Fig. 8.3] in SML I get:

```
stdIn:216.8-216.12 Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
    z 3
```

That's fine - I understand that the line z(3); causes an error, since z throws int instead of real. But my problem is with the line x(3.0); why doesn't it cause an error?

```
let val x = ref 0
in
  x := !x + 1;
  println (!x)
end
```

Fig. 8.1 test8.sml

```
let val x = ref 0
in
  x := x + 1;
  println (x)
end
```

Fig. 8.2 test13.sml

```
exception E of int;
fun g(y) = raise E(y);
fun f(x) =
    let
      exception E of real;
      fun z(y)= raise E(y);
    in
      x(3.0);
      z(3)
    end;
f(g);
```

Fig. 8.3 Exception program

The answer is that the program in Fig. 8.3 never executes in Standard ML because it is not correctly typed. Since it is not correctly typed, the type inference system finds the type error, not with the first sequentially evaluated expression, but with the function application of z to 3. Without static type checking before the program runs, the Small language that we developed in Chap. 6 would try to execute this program and would encounter an error when evaluating *x(3.0)*. We need type inference to prevent this from happening. Preventing an incorrectly typed program from running catches many unintended errors that might only be caught at run-time otherwise. The type checker helps us find errors that might otherwise go undetected until the code path gets executed.

8.1.2 A Bad Function Call

One more example helps to illustrate the need for type inference. Consider the program in Fig. 8.4. This program is incorrect because it is missing a semicolon between the two println expressions. However, in the absence of type inference it starts run-

```
let val x = 6
in
  println x
  println "Done"
end
```

Fig. 8.4 A bad function call

ning and produces a run-time error stating that *None* is not a callable object. The first call to *println* looks like a curried function call of *println x println "Done"*. The result of *println x* is *None*. That appears to Small to be a function that should be passed the next argument, *println*. Hence we get the *"None is not callable"* run-time error message from the JCoCo virtual machine when the correct error message should come from type inference on this program to say that the println function application does not match its signature.

It would be much better to report to the programmer that the programs in Sect. 8.1 are invalid and do not pass the type inference system. It is dangerous for a program to execute that has undefined results because while an implementation detail like the JCoCo Virtual Machine's use of cell variables may allow a program to execute with the correct output, the implementation of the virtual machine or even a completely different target architecture could then cause a once working program to suddenly stop working. As programmers we rely on the tools we use to produce correct code and to guarantee that once debugged the behavior of a program won't suddenly change due to external factors like a compiler change.

8.2 Type Inference Rules

A type inference system is defined in terms of *type inference rules*. The collection of these rules define a type inference system. Each type inference system defines its own set of rules. Type inference rules follow a pattern of necessary conditions, or premises, and a logical conclusion. The rules are written in this form.

RuleName

$$\frac{Premise_1,\ \ Premise_2,\ \ ...,\ \ Premise_n}{Conclusion}$$

The way to read this is to say that if each of the premises hold in some model, then the conclusion holds as well in that model. An inference system contains a collection of inference rules. Normally each rule in an inference system is given a name so it can be referred to in proofs. The collection of inference rules can be used in constructing a proof. In this case a proof of an expression's type.

All the type inference rules for the Small language are provided in the sections in this chapter. Some of the type inference rules will contain braces surrounding sytantic elements of the language (i.e. { and }). These braces are used to indicate zero or more occurrences of syntactic elements.

Much of the Prolog implementation of this type inference system is provided as well, although some pieces of it are left as exercises for the reader.

8.3 Using Prolog

The Small language and grammar is sufficiently complex that writing a top-down parser for it would be difficult. Since Prolog's grammar support creates a top-down parser from a grammar, it is not powerful enough to parse programs in the Small language. So, the program is not parsed by Prolog. Instead, the *mlcomp* compiler writes a file called *a.term* which is a Prolog term representing the abstract syntax of the source program. This AST is read by the Prolog type inference system. Consider the program in Fig. 8.1. The AST Prolog term for this program is shown in Fig. 8.5. In most cases, even if the compiler has not been extended to generate the correct code for a program, the compiler will still write a correct Prolog term. If compiling a new extension to the language the *writeTerm* function in *mlcomp.sml* may have to be extended to support the new extension.

The code in Fig. 8.6 starts the type checker. The *run* predicate reads the abstract syntax tree for the program from the file called *a.term*. The *print* prints it back to

```
letdec(
  bindval(idpat('x'),apply(id('ref'),int('0'))),
  [apply(id(':='),tuple([id('x'),
   apply(id('+'),tuple([apply(id('!'),id('x')),
         int('1')]))])),
   apply(id('println'),apply(id('!'),id('x')))
  ]).
```

Fig. 8.5 test8.sml AST

```
finalStatus(typeerror) :- print('The program failed to pass the typechecker.'), nl, !.
finalStatus(_) :- print('The program passed the typechecker.'), nl, !.

warning([],_) :- !.
warning(_,fn(_,_)) :- !.
warning([_|_],_) :-
  print('Warning: type vars not instantiated in result type initialized to dummy types!'),
  nl, nl, !.

errorOut(error(E)) :-
     nl, nl, print('Error: Typechecking failed. Message was : '), nl,
     print(E), nl, nl, halt(0).
errorOut(typeerror(E)) :-
     nl, nl, print('Error: Typechecking failed due to type error. Message was : '), nl,
     print(E), nl, nl, halt(0).
errorOut(E) :-
     nl, nl, print('Error: Typechecking failed for unknown reason : '), nl,
     print(E), nl, nl, halt(0).
run :- print('Typechecking is commencing...'), nl,
       readAST(AST), print('Here is the AST'), nl, print(AST), nl, nl, nl,
       catch(typecheckProgram(AST,Type),E,errorOut(E)),
       nl, nl, print('val it : '), printType(Type,TypeVars), nl, nl,
       warning(TypeVars,Type), finalStatus(Type).
runNonInteractive :- run, halt(0).
```

Fig. 8.6 The type checker run predicate

the screen just for visual confirmation. The *catch* is a Prolog predicate that provides exception handling. The first argument to *catch* is a predicate to satisfy. If an exception occurs while attempting to satisfy the predicate the error is unified with *E* and the *errorOut* predicate is called which prints one of three messages depending on the error.

If no error occurs, the variable *Type* will hold the type returned by the Small program. The *printType* predicate prints the type in Standard ML format and returns a list of any type variables it finds. The *warning* predicate warns of any uninstantiated type variables found in the type.

The *cut operator* (i.e. !) stops Prolog from backtracking. Normally, if a point is reached where Prolog cannot satisfy a predicate, it will undo the last unification and look for another way to satisfy the original query. The type inference system has side-effects, like printing error messages, and the type inference is deterministic in its choices. There is only one way to satisfy predicates in the type inference system: by finding the type of the program. To prevent backtracking the cut operator can be used. Technically, the cut operator is not needed because different cases of a predicate should all be logically mutually exclusive. However, it is sometimes more convenient to use the cut. When Prolog comes across a cut operator, the search space is pruned. The predicate in which the cut is found may not be satisfied by any other choices in that predicate. In the *warning* predicate, once one of the patterns matches (from the top down), the warning predicate cannot be satisfied by any other *warning* definition. As a Standard ML programmer this is appealing because it leads to the same kind of pattern matching used in Standard ML programs.

So, a term like the one in Fig. 8.5 is read as the *AST* by the type checker and passed to the predicate called *typeCheckProgram* that does the type inference of the Small program. The AST description is given in Standard ML form in Fig. 8.7. Prolog does not require datatypes be declared so there is no explicit declaration of the AST datatype in the typechecker. Nevertheless, the datatype is coded into the expected values of AST nodes in the type checker predicates. The Prolog AST format is nearly an exact copy from the Standard ML AST definition except that *boolval* in the Standard ML implementation is called *bool* in the Prolog version, the *infixexp* in the Standard ML AST is replaced with an *apply* in the type checker, and the *raise* AST node is replaced with an apply. See the *writeExp* function for infix expressions in *mlcomp.sml* for the details of the conversion from *infixexp* to *apply* and *raise* to *apply*. The implementation of the type checker follows from the definition of the abstract syntax.

The Standard ML types used in the Small language include the types in Fig. 8.8. These types include the usual boolean, integer, and string types. The *exn* type is the type of exceptions. The *tuple* type is a tuple of some aggregation of other types. Lists must be homogeneous meaning they are a list of some one type of value. The type *fn(A,B)* is the type of all functions. Every function takes one argument, which may be a tuple, and returns one value. The *ref* types are the reference types and are defined by the type of value to which they point. Type variables are denoted by the *typevar* type. The string in a typevar is the name of the type variable. The type checker assigns variable names as a, b, c, d, etc. The type checker is strict in *typeerror*, meaning once

```
datatype
  exp = int of string
      | ch of string
      | str of string
      | bool of string
      | id of string
      | listcon of exp list
      | tuplecon of exp list
      | apply of exp * exp
      | expsequence of exp list
      | letdec of dec * (exp list)
      | handlexp of exp * match list
      | ifthen of exp * exp * exp
      | whiledo of exp * exp
      | func of string * match list
and
  match = match of pat * exp
and
  pat = intpat of string
      | chpat of string
      | strpat of string
      | boolpat of string
      | idpat of string
      | wildcardpat
      | infixpat of string * pat * pat
      | tuplepat of pat list
      | listpat of pat list
      | aspat of string * pat
and
  dec = bindval of pat * exp
      | bindvalrec of pat * exp
      | funmatch of string * match list
      | funmatches of (string * match list) list
```

Fig. 8.7 AST description

an expression results in a type error all other expressions that interact with it also result in *typeerror*.

The job of the type checker is to map a program in the syntax of Fig. 8.7 into its type as defined in Fig. 8.8. Type inference rules provide the mapping instructions. The rest of this chapter explores type inference for the simplest nodes first, working up to more complex language constructs.

```
type = bool
     | int
     | str
     | exn
     | tuple of type list
     | listOf of type
     | fn of type * type
     | ref of type
     | typevar of string
     | typeerror
```

Fig. 8.8 Small types

8.4 The Type Environment

Functions in Standard ML are typed by their signature as seen in Chap. 5. For instance, the *Int.fromString* function has a signature of

$$fn : str \rightarrow int$$

The *environment* of the type checker provides information about the signature of built-in functions and operators in the language. The environment is referred to as epsilon (i.e. ε), the *type environment*, or just the *environment*. More generally, the *environment* provides a mapping of identifiers to types which can be consulted during type checking as needed.

Some functions are polymorphic and therefore type variables are necessary to describe their type. For instance, the *print* function has a type of

$$fn : \alpha \rightarrow ()$$

The α represents a type variable in the signature of the *print* function. The existence of type variables makes it possible for functions in the Standard ML type inference system to be polymorphic.

In Prolog the environment is created by the *typecheckProgram* predicate which passes it to the *typecheckExp* predicate. Figure 8.9 provides the type environment given to the *typecheckExp* predicate.

There are a number of functions and operators provided in the environment. The function type begins with *fn*. All type variables are named *typevar* and the *unit* type is denoted as *tuple([ ])* in the type checker. The environment is represented as follows in the type inference rules.

$$\varepsilon = [Exception \mapsto \alpha \rightarrow exn,\ raise \mapsto exn \rightarrow \alpha,\ andalso \mapsto bool \times bool \rightarrow bool, ...]$$

```
typecheckProgram(Expression,Type) :-
  typecheckExp([('Exception',fn(typevar(a),exn)),
                ('raise',fn(exn,typevar(a))),
                ('andalso',fn(tuple([bool,bool]),bool)),
                ('orelse',fn(tuple([bool,bool]),bool)),
                (':=',fn(tuple([ref(typevar(a)),typevar(a)]),tuple([]))),
                ('!',fn(ref(typevar(a)),typevar(a))),
                ('ref',fn(typevar(a),ref(typevar(a)))),
                ('::',fn(tuple([typevar(a),listOf(typevar(a))]),listOf(typevar(a)))),
                ('>', fn(tuple([typevar(a),typevar(a)]),bool)),
                ('<', fn(tuple([typevar(a),typevar(a)]),bool)),
                (@,fn(tuple([listOf(typevar(a)),listOf(typevar(a))]),listOf(typevar(a)))),
                ('Int.fromString',fn(str,int)),
                ('input',fn(str,str)),
                ('explode',fn(str,listOf(str))),
                ('implode',fn(listOf(str),str)),
                ('println',fn(typevar(a),tuple([]))),
                ('print',fn(typevar(a),tuple([]))),
                ('cprint',fn(typevar(a),cprint)),
                ('type',fn(typevar(a), str)),
                (+,fn(tuple([int,int]),int)),
                (-,fn(tuple([int,int]),int)),
                (*,fn(tuple([int,int]),int)),
                ('div',fn(tuple([int,int]),int))],
          Expression,Type).
```

Fig. 8.9 The type environment

The environment is a list of bindings of identifiers to types. The environment is always searched from left to right to find a binding as needed by type inference rules. The symbol $\mapsto$ is pronounced *maps to*. For instance, Exception maps to a polymorphic type from *alpha* to *exn*.

8.5 Integers, Strings, and Boolean Constants

The types of integer, string, and boolean constant values are determined by the scanner when read in *mlcomp*. Determining their types then is just a matter of matching their scanned type to a type in the type checker. So we write the following statements about the types of simple constant values. In each case, there are no premises that must be satisfied. When we see a boolean constant we can immediately determine its type.

BoolCon

$$\frac{}{\varepsilon \vdash bool(v) : bool}$$

IntCon

$$\frac{}{\varepsilon \vdash int(v) : int}$$

StringCon

$$\frac{}{\varepsilon \vdash str(v) : str}$$

```
typecheckExp(_,int(_),int).
typecheckExp(_,bool(_),bool).
typecheckExp(_,str(_),str).
```

Fig. 8.10 Constant type inference

To keep things simpler in the type inference algorithm we'll limit our discussion to integers for all numbers. Each type inference rule will be named in bold and its definition will be indented underneath it as seen here. In Prolog constant types are given a type by the typecheckExp predicate as shown in Fig. 8.10. The environment is the first argument to the typecheckExp predicate and is a don't care value in this case since the environment is not needed to determine the type of a constant. The AST argument is the second argument to the predicate. The third argument is the type of the expression.

Consider the expression *5*. This is mapped into the term *int(5)* by the mlcomp compiler. Passing *int(5)* to the type checker matches the predicate in Fig. 8.10 and returns *int* for its type. The type is printed by the type checker. Output from the type checker looks like this.

```
Typechecking is commencing...
Here is the AST
int(5)
val it : int
The program passed the typechecker.
```

8.6 List and Tuple Constants

The type of a list is derived from its constituent type. Lists are homogeneous in Small as they are in Standard ML, meaning that all elements must have the same type. The type of a tuple is derived from its constituent types. For example consider this list and tuple.

$$[6, 5, 4] : int\ \ list$$
$$("hi", true, 6) : str * bool * int$$

In the abstract syntax, list and tuple constants are written as lists of values. For instance, written in Prolog syntax, to typecheck the two values above, *typecheckExp* is implemented as follows.

```
typecheckExp(Env,listcon(L),listOf(T)) :- typecheckList(Env,L,T).
typecheckExp(Env,tuple(L),tuple(T)) :- typecheckTuple(Env,L,T).
```

Typechecking the list and tuple constants above returns these type values.

```
listOf(int)
tuple([str,bool,int])
```

Note the type value of *listOf* here. *list* is a built-in predicate in Prolog and should not be used. Here is the type inference rule that describes the type of lists in Small.

ListCon

$$\frac{\forall i \;\; 1 \leq i \leq n, n \geq 0 \quad \varepsilon \vdash e_i : \alpha}{\varepsilon \vdash [e_1, e_2, ..., e_n] : \alpha \;\; list}$$

The *List* type inference rule can be read as follows: If in the type environment the types of all elements of a list are found to be α, then the type of the list constant of these values is $\alpha \; list$ in the same type environment. In the vacuous condition, where $n = 0$, there are no premises with the type of the list being polymorphically $\alpha \; list$.

For tuples the type inference rule is somewhat similar. The $\times$ in the rule below is the cross product symbol and is the symbol that corresponds to $*$ printed by the Standard ML type checker. The writing of this cross product forms the type for tuples of n elements.

TupleCon

$$\frac{\forall \;\; 1 \leq i \leq n, n \geq 0 \quad \varepsilon \vdash e_i : \alpha_i}{\varepsilon \vdash (e_1, e_2, ..., e_n) : \times_{i=1}^{n} \alpha_i}$$

In the vacuous condition of $n = 0$ in the *TupleCon* rule the type is the empty Cartesian product which is denoted as the *unit* type in Standard ML. In other words, the empty tuple has type *unit* in Standard ML.

Consider type checking the expression *[1,2,3,4]*. The type checker provides output as shown below. Typechecking the list constant calls *typecheckList* as shown earlier in this section. The *typecheckList* predicate proceeds through the list of elements making sure all the types match, resulting in the type you see below.

```
Typechecking is commencing...
Here is the AST
listcon([int(1),int(2),int(3),int(4)])
val it : int list
The program passed the typechecker.
```

8.7 Identifiers

When a program uses an identifier the type of the identifier must be looked up in the type environment. Lookup in the environment is denoted as $\varepsilon[id \mapsto \alpha]$ which says that in the type environment find *id* and its associated type *alpha*. The rule

```
exists(Env,Name) :-
    member((Name,_),Env), !.
find(Env,Name,Type) :-
    member((Name,Type),Env), !.
find(Env,Name,Type) :-
    writeMsg(['Failed to find ',
    Name,' with type ',Type,
    ' in environment : ']), print(Env), nl,
    throw(typeerror('unbound identifier')).
typecheckExp(Env,id(Name),Type) :-
    find(Env,Name,Type).
```

Fig. 8.11 Environment lookup predicates

below indicates the type of an identifier is its type in the environment. In the Prolog implementation a *find* predicate is written to look up an indentifier in an environment to find its type. Here is the identifier type inference rule.

Identifier

$$\frac{}{\varepsilon[id \mapsto \alpha] \vdash id : \alpha}$$

The code in Fig. 8.11 provides the details of the *find* predicate implementation in Prolog. There is also an *exists* predicate that is satisfied if an environment contains a binding. The *member* predicate is a built-in predicate in Prolog. Normally in a proof this lookup will be implied when an identifier is looked up in the bindings and this step will be omitted. Consider the expression containing just the name of a function, as in *println*. Type checking this expression will reveal the type of *println*, which is not a Standard ML function but is in the Small language.

```
1 Typechecking is commencing...
2 Here is the AST
3 id(println)
4 val it : 'a -> unit
5 The program passed the typechecker.
```

The type checker sees the identifier and looks in the environment, finding the *println* identifier and yielding its type.

8.8 Function Application

Function application in Small and Standard ML occurs when two expressions are written next to each other as in the expression

```
1 println 6
```

for instance. In the Prolog AST this appears as *apply(id('println'),int('6'))*. Function application is the act of calling a function. The type of *println* is $\alpha \rightarrow unit$. The println function is being applied to an integer. We need a type inference rule that formally defines a legal function application.

Before the function application type inference rule can be written one more operator is needed which may be a bit difficult to understand at first. Small and Standard ML support polymorphic type checking. When a type contains type variables the type variables place restrictions on the kinds of values to which the type may be instantiated. For instance, the *println* function has type $\alpha \rightarrow unit$ which says that the function *println* is polymorphic taking arguments of any type. The type is defined with the type variable α, but just when is *println* polymorphic? The answer is every time *println* is called. One application of *println* can be given an integer, while the next application could be given a tuple of an integer and a boolean value. In each case the α type variable is instantiated to a type, an integer in the first case and a tuple in the second. Type inference rules need a way of creating instances of polymorphic types. In this way, one instance of the polymorphic type $\alpha \rightarrow unit$ can be instantiated as $int \rightarrow unit$ while the next can be instantiated as $int \times bool \rightarrow unit$.

In type inference rules this instantiation operator is written as *inst*. It is given a type and returns an instance of that type where all type variables are replaced by fresh, unbound instances of variables. In the type inference rule below the result type of function application is the specialization of the instantiated result type given an instance of the type of the argument passed to the function.

FunApp

$$\frac{\varepsilon \vdash e_1 : \alpha \rightarrow \beta,\ \ \alpha' \rightarrow \beta' : inst(\alpha \rightarrow \beta),\ \ \varepsilon \vdash e_2 : \alpha_{e2}, \alpha' : inst(\alpha_{e2})}{\varepsilon \vdash e_1 e_2 : \beta'}$$

The Prolog implementation of instantiation will shed some light on instantiation. In Prolog, all type variables are written as *typevar(id)* where *id* is typically some letter from *a* to *z*, but could be any identifier. This corresponds to the way type variables appear in Standard ML's type inference system when types like

fn:'a->'a

are printed. In the Prolog implementation of the typechecker the function type *fn:'a ->'a* is represented as *fn(typevar(a),typevar(a))*. Making an instance of a type like this creates a type that can be unified with other types in Prolog. An instance of this type would be written as *fn(A,A)*. In this Prolog term the variable *A* is unbound since it is not unified with any other term. The Prolog term *fn(A,A)* is an instance of the type *fn(typevar(a),typevar(a))*. Instantiation is performed by the *inst* predicate shown in Fig. 8.12.

On line 17 of Fig. 8.12 the *inst* operator calls the *instanceOf* predicate with an empty environment. The *instanceOf* predicate recursively traverses the type, changing all occurrences of type variables to Prolog variables. The environment keeps track

```
1  instanceOfList(Env,[],[],Env).
2  instanceOfList(Env,[H|T],[G|S],NewEnv) :-
3      instanceOf(Env,H,G,Env1), instanceOfList(Env1,T,S,NewEnv).
4  instanceOf(Env,A,A,Env) :- var(A), !.
5  instanceOf(Env,A,A,Env) :- simple(A), !.
6  instanceOf(Env,fn(A,B), fn(AInst,BInst),Env2) :-
7      instanceOf(Env,A,AInst,Env1), instanceOf(Env1,B,BInst,Env2), !.
8  instanceOf(Env,listOf(A),listOf(B),NewEnv) :- instanceOf(Env,A,B,NewEnv), !.
9  instanceOf(Env,ref(A),ref(B),NewEnv) :- instanceOf(Env,A,B,NewEnv), !.
10 instanceOf(Env,tuple(L),tuple(M),NewEnv) :- instanceOfList(Env,L,M,NewEnv), !.
11 instanceOf(Env,typevar(A),B,Env) :- exists(Env,A), find(Env,A,B), !.
12 instanceOf(Env,typevar(A),B,[(A,B)|Env]) :- !.
13 instanceOf(_,A,B,_) :-
14     print('Type Error: Type '), printType(B,_),
15     print(' is not an instance of '), printType(A,_), nl,
16     throw(typeerror('type mismatch')), !.
17 inst(X,Y) :- instanceOf([],X,Y,_).
```

Fig. 8.12 The instantiation operator

```
typecheckExp(Env,apply(Exp1,Exp2),ITT) :-
    typecheckExp(Env,Exp1,fn(FT,TT)),
    typecheckExp(Env,Exp2,Exp2Type),
    inst(Exp2Type,Exp2TypeInst),
    catch(inst(fn(FT,TT), fn(Exp2TypeInst,ITT)),_,
        printApplicationErrorMessage(Exp1,
            fn(FT,TT),Exp2,Exp2Type,ITT)), !.
```

Fig. 8.13 Function application type inference

of the mapping of type variables to Prolog variables so if a type variable appears more than once in a type it is replaced by the same Prolog variable as is evident with the example of the polymorphic type of function *f* in the preceeding paragraph. Line 11 insures the same Prolog variable is used when the type variable is found in the environment. Line 12 creates a new Prolog variable when the type variable is not found in the environment.

Line 4 of Fig. 8.12 uses the *var* predicate which returns true if *A* is an unbound Prolog variable. This clause is important because if *instanceOf* is called with an uninstantiated variable already, then it will unify with anything it is matched to, like the function type in line 6 for instance. Line 4 insures that an unbound variable stays unbound. Line 5 uses the *simple* predicate which just means that *A* is a simple term like *int*, or *bool*. It is not complex, meaning there are no subterms that are a part of this term. A complex term would be a type like *tuple([typevar(a),typevar(a)])*. Line 5 handles all the simple types by just returning them. Simple types are not polymorphic.

Type inference for function application in Prolog utilizes Prolog exception handling as shown in Fig. 8.13. If a function call is not correct due to a type error, the

instantiation predicate in Fig. 8.12 will throw a type error exception. In that case it would be nice to know there was an error with a function call. The error is caught in this code and a message is printed.

8.8.1 Instantiation

When an instance of a type is created with free variables, the Prolog variables only stay free as long as the instantiated type is not unified with any other types. Once that instance of a type is unified some or all of the free variables will be bound. In this way, when an instance of a type is created, it moves towards being a type with no free variables as type inference proceeds. If unification is not possible due to a type error, then that condition is recognized and the resulting type is the special type *typeerror* which is handled in the Prolog implementation by throwing an exception.

Several of the rules in the next section use instantiation so that unification of types is possible. When an instance of a type is the result of a type inference rule, all free variables have been unified with bound values producing a valid type except in the cases of type errors in the original program. Consider the invocation of *println 6* and how we would arrive at a type. The following instance of the *FunApp* rules shows how it is proved to be a valid function application.

$$\frac{\varepsilon \vdash println : \alpha \rightarrow unit, \quad int \rightarrow unit : inst(\alpha \rightarrow unit), \quad \varepsilon \vdash 6 : int}{\varepsilon \vdash println\ 6 : unit}$$

8.9 Let Expressions

Binding identifiers to values is the job of *let* expressions in Standard ML and Small. Let expressions create bindings between identifiers and values through the use of patterns. Identifiers can be bound to one or more function definitions in a *let* expression because functions are values too in Small and Standard ML. A little new notation must be introduced to write type inference rules for *let* expressions.

Let expression build new environments. To properly define type inference for the newly created environment, environments must be considered values in the type checker. A declaration produces an environment mapping one or more identifiers to their types. To combine two environments a new overlay operator is defined. One environment can then be used to partially overlay another environment. Consider two environments ε_1 and ε_2. To combine the first with the second environment the overlay $\oplus$ operator is defined as demonstrated here.

$$\begin{aligned}
\varepsilon_1 &= [x \mapsto \alpha \to \beta,\ y \mapsto int,\ z \mapsto \alpha \times \beta] \\
\varepsilon_2 &= [u \mapsto \alpha \times \beta \to \beta,\ y \mapsto\ bool] \\
\varepsilon_2 \oplus \varepsilon_1 &= [u \mapsto \alpha \times \beta \to \beta,\ y \mapsto\ bool] \oplus [x \mapsto \alpha \to \beta,\ y \mapsto int,\ z \mapsto \alpha \times \beta] \\
&= [u \mapsto \alpha \times \beta \to \beta,\ y \mapsto\ bool, x \mapsto \alpha \to \beta,\ y \mapsto int,\ z \mapsto \alpha \times \beta]
\end{aligned}$$

Since environments are always searched from left to right, the result of the overlay operator is the concatenation of the two environments. In this example the result is that y is mapped to *bool* in the new environment $\varepsilon_2 \oplus \varepsilon_1$. In Prolog, environments are represented as lists of bindings just as described here. The overlay operator is simply the *append* predicate in Prolog. Recalling that the *find* predicate searches an environment from left to right the result of appending two lists is the overlay of the bindings in the second list. One more bit of notation is needed. When a declaration creates a new environment it will be written using a double right arrow as follows.

$$\varepsilon \vdash dec \Rightarrow \varepsilon_{dec}$$

This indicates that the declaration builds a new environment ε_{dec} that will be used later in the type inference rule. Now we are ready to define the *let* expression type inference rule.

Let

$$\frac{\varepsilon \vdash dec \Rightarrow \varepsilon_{dec},\ \ \varepsilon_{dec} \oplus \varepsilon \vdash e_{sequence} : \beta}{\varepsilon \vdash let\ dec\ in\ e_{sequence}\ end : \beta}$$

The *dec* declaration in the rule above can be one of two types of declarations in Small: either a *val* declaration or a series of *fun* declarations. The type inference for these two types of declarations is provided in the rules below. The expression *e* in the rule above is a sequence of expressions. The type inference rule for sequential execution is provided in a later section of this chapter.

ValDec

$$\frac{pat : \alpha \Rightarrow \varepsilon_{pat},\ \ \varepsilon \vdash e : close(\alpha)}{\varepsilon \vdash val\ pat = e \Rightarrow \varepsilon_{pat}}$$

In the *ValDec* rule there are pattern declarations. The type inference rules for pattern declarations are provided in the next section of the chapter. Each pattern declaration provides an environment mapping identifiers in the pattern to their associated types. The next section provides the type inference rules for pattern matching along with the environments yielded by each type of pattern.

ValRecDec

$$\frac{[id : \alpha] \oplus \varepsilon \vdash e : \alpha}{\varepsilon \vdash val\ rec\ id = e \Rightarrow [id : close(\alpha)]}$$

A *ValRecDec* is used when an identifier is bound to an anonymous function that calls itself recursively. Anonymous functions don't normally call themselves. In this one instance, the anonymous function can through the use of a recursive binding. The binding in this case binds the identifier to the type of the function in the body of the function.

FunDecs

$$\frac{\forall i\ 1 \leq i \leq n, \forall j\ 1 < j \leq n,\ \ n \geq 1, \quad [id_1 \mapsto \alpha_1 \to \beta_1\ \{,\ id_j \mapsto \alpha_j \to \beta_j\}] \oplus \varepsilon \vdash id_i\ matches_i : \alpha_i \to \beta_i}{\varepsilon \vdash fun\ id_1\ matches_1\ \{and\ id_j\ matches_j\} \Rightarrow [id_1 \mapsto close(\alpha_1 \to \beta_1)\ \{,\ id_j \mapsto close(\alpha_j \to \beta_j)\}]}$$

In the rule above the braces (i.e. { and }) are EBNF and represent zero or more occurrences as necessary. Since *j* must be greater than 1, if *n=1* then no occurrences of the parts written inside braces are necessary. This rule introduces *matches*. The type inference for matches appears right after the section on patterns.

A *FunDecs* is a series of mutually recursive function definitions. See *mlcomp.sml* for examples where the keyword *and* is used between function definitions. The rule above starts with the premise that each function in the *FunDecs* has a type $\alpha \to \beta$. The rule makes an instance of the function type and places it in the environment given the *matches*. The *matches* are the list of pattern matches for one function definition. This is done because all recursive function calls to functions in the *FunDecs* must have consistent types. As the type inference rules are satisfied the instance of the type is bound to type values. If these premises are met, the conclusion produces a new environment with each function bound to its type.

The newly built environment that results from the *FunDecs* rule contains a type function called *close*. This type function is important. Closing a type means that any free type variables (i.e. Prolog type variables) are instantiated to *typevar* type variables. This is needed because otherwise the first application of a function with free type variables would instantiate them to the types of that particular function application. This would not be a problem if functions were not polymorphic. However, functions in Standard ML often have polymorphic types. The *close* type function is needed to support polymorphic type inference. The *close* function is the inverse of the *inst* type function.

8.10 Patterns

Patterns are used in *ValDec* declarations and in *matches* which are discussed in the next section. When a pattern is used, it produces bindings of one or more identifiers to types. Constant values can be used as patterns as in the *IntPat*, *BoolPat*, *StrPat*, *NilPat*, and *UnitPat* rules. Patterns like this don't produce any bindings because identifiers are not part of these patterns.

IntPat

$$\frac{}{integer\_constant : int \Rightarrow [\,]}$$

BoolPat

$$\frac{}{true : bool \Rightarrow [\,]}$$

$$\frac{}{false : bool \Rightarrow [\,]}$$

StrPat

$$\frac{}{string\_constant : str \Rightarrow [\,]}$$

NilPat

$$\frac{}{nil : \alpha\ list \Rightarrow [\,]}$$

ConsPat

$$\frac{pat_1 : \alpha \Rightarrow \varepsilon_{pat_1},\quad pat_2 : \alpha\ list \Rightarrow \varepsilon_{pat_2}}{pat_1 :: pat_2 : \alpha\ list \Rightarrow \varepsilon_{pat_1} + \varepsilon_{pat_2}}$$

TuplePat

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, n \ge 0 \\ pat_i : \alpha_i \Rightarrow \varepsilon_{pat_i}\end{array}}{(pat_1, pat_2, ..., pat_n) : \times_{i=1}^{n} \alpha_i \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

ListPat

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, n \ge 0 \\ pat_i : \alpha \Rightarrow \varepsilon_{pat_i}\end{array}}{[pat_1, pat_2, ..., pat_n] : \alpha\ list \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

The *ConsPat*, *TuplePat*, and *ListPat* rules may contain other patterns. Each of them employ the disjoint union operator to build new environments from their sub-environments. Disjoint union is used because duplicate identifiers are not allowed in patterns. The $+$ and $\sum$ symbols are used to denote the disjoint union of sets of patterns.

The *TuplePat* rule forms the cross product type of all its constituent types and forms the environment that results from all the sub-pattern environments being overlayed

```
let
  val (x,y)::L = [(1,2),(3,4)]
in
  println x
end
```

Fig. 8.14 Pattern matching

on one another. In the vacuous case, when $n = 0$, the *TuplePat* rule derives the *unit* pattern (i.e. the empty tuple) and yields an empty environment.

The vacuous case of the *ListPat* rule, when $n = 0$, provides an alternative form of specifying the empty list. Both *nil* and *[]* represent the empty list in Standard ML with polymorphic type $\alpha\ list$.

IdPat

$$\frac{}{id : \alpha \Rightarrow [id \mapsto \alpha]}$$

Most patterns boil down to creating bindings of identifiers to values. The *IdPat* type inference rule yields a new binding environment, binding the identifier to its type. Consider the program in Fig. 8.14. Typechecking this program results in the following output.

```
1 letdec(
2   bindval(infixpat({:}{:},tuplepat([idpat(x),idpat(y)]),idpat(L)),
3     listcon([tuple([int(1),int(2)]),tuple([int(3),int(4)])])),
4   [apply(id(println),id(x))])
5 val (x,y){:}{:}L : (int * int) list
6 val it : unit
7 The program passed the typechecker.
```

The type inference rules specify how the type checker works. To see this in action a proof is possible using the type inference rules. Each step in the proof is justified by a type inference rule written to the right side of the rule's use. To reach the conclusion (1) of the type checker, premises (2) and (3) must hold.

$$\frac{(2)\varepsilon \vdash val\ (x, y)::L = [(1, 2), (3, 4)] \Rightarrow \varepsilon_{dec} \quad (3)\varepsilon_{dec} \oplus \varepsilon \vdash println\ x : unit}{(1)\varepsilon \vdash let\ val\ (x, y)::L = [(1, 2), (3, 4)]\ in\ println\ x\ end : unit}(Let)$$
$$\varepsilon_{dec} = [x \mapsto int, y \mapsto int, L \mapsto int * int\ list]$$

To prove (2):

$$\frac{(4)(x, y)::L : int \times int\ list \Rightarrow \varepsilon_{dec} \quad (5)\varepsilon \vdash [(1, 2), (3, 4)] : int \times int\ list}{(2)\varepsilon \vdash val\ (x, y)::L = [(1, 2), (3, 4)] \Rightarrow \varepsilon_{dec}}(ValDec)$$

To prove (4):

$$\frac{(6)(x, y) : int \times int \Rightarrow [x \mapsto int, y \mapsto int] \quad (7)L : int \times int\ list \Rightarrow [L \mapsto int \times int\ list]}{(4)(x, y)::L : int \times int\ list \Rightarrow \varepsilon_{dec}}(ConsPat)$$

To prove (6):

$$\frac{(8)x : int \Rightarrow [x \mapsto int] \quad (9)y : int \Rightarrow [y \mapsto int]}{(6)(x, y) : int \times int \Rightarrow [x \mapsto int, y \mapsto int]}(TuplePat)$$

Premises (7), (8), and (9) are true by virtue of the *IdPat* inference rule. Considering (5):

$$\frac{(10)\varepsilon \vdash (1, 2) : int \times int \quad (11)\varepsilon \vdash (3, 4) : int \times int}{(5)\varepsilon \vdash [(1, 2), (3, 4)] : int \times int\ list}(ListCon)$$

Considering (10) and a similar argument for (11):

$$\frac{(12)\varepsilon \vdash 1 : int \quad (13)\varepsilon \vdash 2 : int}{(10)\varepsilon \vdash (1, 2) : int \times int}(TupleCon)$$

Both (12) and (13) are true by the *IntCon* rule. A similar argument holds for (11). The proof nears completion by proving (3):

$$\frac{(14)\varepsilon_{dec} \oplus \varepsilon \vdash println : \alpha \rightarrow unit \quad int \rightarrow unit : inst(\alpha \rightarrow unit) \quad (15)\varepsilon_{dec} \oplus \varepsilon \vdash x : inst(int)}{(3)\varepsilon_{dec} \oplus \varepsilon \vdash println\ x : unit}(FunApp)$$

Both (14) and (15) are true by the *Identifier* rule concluding the proof of the type correctness of this program. The sequence rule was glossed over in this proof. Sequence type checking appears later in the chapter.

Practice 8.1 Prove that the program in Fig. 8.15 is correctly typed. The abstract syntax for this program is provided here.

```
letdec(bindval(idpat('x'),int('5')),
  [letdec(bindval(idpat('y'),int('6')),
    [apply(id('println'),apply(id('+'),tuple([id('x'),id('y')])))
        ])
  ]).
```

You can check your answer(s) in Section 8.19.1.

Practice 8.2 Minimally, what must the type environment contain to correctly type check the program in Fig. 8.15.
You can check your answer(s) in Section 8.19.2.

```
let val x = 5
    val y = 6
in
  println (x + y)
end
```

Fig. 8.15 test10.sml

8.11 Matches

Matches

There are two alternatives to the *Matches* rule differing only in the syntax of the match.

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ n \ge 1 \\ \varepsilon \vdash id : \alpha \to \beta,\ \ pat_i : \alpha \Rightarrow \varepsilon_{pat_i},\ \ \varepsilon_{pat_i} \oplus \varepsilon \vdash e_i : \beta\end{array}}{\varepsilon \vdash id\ pat_1 = e_1 \{\mid id\ pat_j = e_j\} : \alpha \to \beta}$$

or

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ n \ge 1 \\ \varepsilon \vdash id : \alpha \to \beta,\ \ pat_i : \alpha \Rightarrow \varepsilon_{pat_i},\ \ \varepsilon_{pat_i} \oplus \varepsilon \vdash e_i : \beta\end{array}}{\varepsilon \vdash id\ pat_1 => e_1 \{\mid pat_j => e_j\} : \alpha \to \beta}$$

The *Matches* type inference rule handles one or more matches in a function definition or other matches occurrence. A match has an identifier (i.e. the name of the function), a pattern, and an expression. Each match takes an argument and returns a value. The argument and pattern must be of type α and the type of the expression must be of type β. In addition, the bindings created by the pattern are part of the environment when the type of the expression is inferred.

```
let fun f(0,y) = y
      | f(x,y) = g(x,x*y)
    and g(x,y) = f(x-1,y)
in
  println (f(10,5))
end
```

Fig. 8.16 test11.sml

Consider the program in Fig. 8.16. This is an example of a program with multiple function declarations separated by the keyword *and*, thus allowing them to be mutually exclusive, which they are. The first function, f has two matches, which the *Matches* rule handles. The abstract syntax for this program includes two *funmatches*, one for each function f and g.

```
letdec(
  funmatches(
    [funmatch(f,
       [match(tuplepat([intpat(0),idpat(y)]),id(y)),
        match(tuplepat([idpat(x),idpat(y)]),apply(id(g),
            tuple([id(x),apply(id(*),tuple([id(x),id(y)]))]))))]),
     funmatch(g,
       [match(tuplepat([idpat(x),idpat(y)]),
          apply(id(f),tuple([apply(id(-),tuple([id(x),int(1)])),id(y)
              ]))))]),
  [apply(id(println),apply(id(f),tuple([int(10),int(5)])))])
```

Consulting the AST for the program the two matches for f each include a pattern and the expression after the equals sign. The first expression is the y that is returned for the first match of f. The second match of f returns $g(x, x * y)$.

8.12 Anonymous Functions

AnonFun

$$\frac{[id \mapsto \alpha \to \beta] \oplus \varepsilon \vdash id\ matches : \alpha \to \beta}{\varepsilon \vdash fn\ id\ matches : \alpha \to \beta}$$

An anonymous function is given a name by the parser before a Prolog term is created. Names are needed for code generation. The type checker uses the name only to provide consistency in the way the *Matches* type inference rule is satisfied. However, the identifier is not used by the type inference rule because an anonymous function never calls itself recursively except in the case of a *val rec* binding, where a different identifier is present to be bound to the function. Consider the anonymous function defined in Fig. 8.17. The abstract syntax for this program is as shown here.

```
func(anon@0,[match(idpat(x),apply(id(+),tuple([id(x),int(1)])))])
```

Notice that the compiler has assigned a name to this function. The name *anon@0* is needed by the code generator and also by the *Matches* rule above (only to syntactically match the rule though), but is not used during type inference. Applying this program to the *AnonFun* rule we get this instance.

```
(fn x => x+1)
```

Fig. 8.17 Anonymous function

$$\frac{[anon@0 \mapsto int \to int] \oplus \varepsilon \vdash anon@0\ x => x + 1 : int \to int}{\varepsilon \vdash fn\ anon@0\ x => x + 1 : int \to int}$$

In this instance it doesn't appear much has changed. The *fn* has dropped in the premise. The premise is now an instance of the *Matches* rule which can then be applied to further reduce the proof.

Practice 8.3 Provide a complete proof that the program in Fig. 8.17 is correctly typed.
You can check your answer(s) in Section 8.19.3.

8.13 Sequential Execution

Sequence

$$\frac{\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ n \ge 1 \quad \varepsilon \vdash e_i : \alpha_i}{\varepsilon \vdash e_1\{; e_j\} : \alpha_n}$$

Sequential execution of expressions results in the last value of the sequence. All other values are discarded. So, the type of a sequence is the type of the last expression evaluated. In the degenerative case, where $n = 1$, the type of the sequence is the type of the only expression in the sequence.

8.14 If-Then and While-Do

If-Then expressions and While-Do expressions have type restrictions on the types of values they can process. The type inference rules provided here describe those restrictions. The IfThen type inference rule was first presented in Chap. 5.

IfThen

$$\frac{\varepsilon \vdash e_1 : bool,\ \ \varepsilon \vdash e_2 : \alpha,\ \ \varepsilon \vdash e_3 : \alpha}{\varepsilon \vdash if\ e_1\ then\ e_2\ else\ e_3 : \alpha}$$

WhileDo

$$\frac{\varepsilon \vdash e_1 : bool,\ \ \varepsilon \vdash e_2 : \alpha}{\varepsilon \vdash while\ e_1\ do\ e_2 : \alpha}$$

```
typecheckExp(Env,ifthen(Exp1,Exp2,Exp3), RT) :-
  typecheckExp(Env,Exp1,bool), typecheckExp(Env,Exp2,RT),
  typecheckExp(Env,Exp3,RT), !.
typecheckExp(Env,ifthen(Exp1,Exp2,Exp3), _) :-
  typecheckExp(Env,Exp1,bool), typecheckExp(Env,Exp2,ThenType),
  typecheckExp(Env,Exp3,ElseType),
  print('Error: Result types of then and else expressions must match.'), nl,
  print('Then Expression type is: '), printType(ThenType,_), nl,
  print('Else Expression type is: '), printType(ElseType,_), nl,
  throw(typeerror('result type mismatch in if-then-else expression')).
typecheckExp(Env,ifthen(Exp1,_,_), _) :-
  typecheckExp(Env,Exp1,Exp1Type), Exp1Type \PYGZbs{=} bool,
  print('Error: Condition of if then expression must have bool type.'), nl,
  print('Condition Expression type was: '), printType(Exp1Type,_), nl,
  throw(typeerror('type not bool in if-then-else expression condition')).
```

Fig. 8.18 If-Then type inference

While reporting *yes* it type checked correctly and here is your type, or *no* it did not type check correctly is what Prolog would do by default, that isn't really enough information to determine where in a program the type checker failed. As the type checker proceeds, certain error messages can be printed. For instance, consider the code for type checking If-Then expressions in Prolog.

The first rule in Fig. 8.18 is the Prolog implementation of the If-Then type inference rule. If the first rule works the cut operator insures that no backtracking will occur to match it another way. If the first rule is not satisfied, then an error message is printed and an exception is thrown to terminate the type checker.

Strictly speaking, an exception does not need to be thrown in the code of Fig. 8.18. The result of the If-Then failure could be the special type *typeerror*. The type inference algorithm is said to by *strict* in *typeerror* which means that once a type results in *typeerror* all types in which it takes part must also result in *typeerror*. However, this still leads to the whole program failing type inference and throwing an exception is a quick and dirty way to terminate the type inference algorithm.

8.15 Exception Handling

Handler

$$\frac{\varepsilon \vdash e : \alpha,\ \ [handle@ \mapsto\ exn \to \alpha] \oplus \varepsilon \vdash handle@\ matches : exn \to \alpha}{\varepsilon \vdash e\ handle\ matches : \alpha}$$

An exception handler is a polymorphic function as far as the type inference system is concerned, mapping from type *exn* to the type of the expression. Both the expression and its exception handler must have the same result type according to this definition. To implement the handler like a function the identifier *handle@* is bound to the type of the handler.

8.16 Chapter Summary

This is a shorter but denser chapter than some in the text. Type inference is difficult at best to demonstrate on paper. Section 8.10 carries out a complete proof of type correctness as one example from beginning to end of type inference. The type inference system implemented here relies heavily on the unification of Prolog variables to terms. Perhaps the best way to understand this code is to extend it. Implementing type inference rules demands an understanding of how Prolog works. Examining already written type inference rules can help as well.

In spite of it being a challenging topic, inference and unification are two very powerful techniques available to computer programmers through the use of Prolog. Unification provides the means to work both backwards and forwards or anywhere in between as was pointed out with the *append* predicate in the last chapter. In terms of type inference, one important aspect is being able to assign a type to an expression before you know what its type is. By assigning a Prolog variable that will be unified to an actual type later, the type inference can be written very declaratively, like the inference rules themselves, without regard to exactly the order that information is known. That's the power of Prolog. The unification algorithm makes declarative programming in Prolog possible.

Type checking, without type inference, is effective and simpler to implement but costs the programmer more in having to explicitly declare types of each variable. Being explicit about types is not always a bad thing. Even the SML compiler needs a little help sometimes by declaring the type of a function parameter. Regardless of the language, every type checker engages in some type inference. Standard ML's type inference system differs from other language implementations by the extent to which types are inferred.

8.17 Review Questions

1. What appears above and below the line in a type inference rule?
2. Why don't infix operators appear in the abstract syntax of programs handled by the type checker?
3. What does *typevar* represent in Fig. 8.8?
4. What does *typeerror* represent in Fig. 8.8?
5. What does the *type* of the list [("hello",1,true)] look like as a Prolog term?
6. What is the type environment?
7. Give an example of the use of the overlay operator.
8. What pattern(s) are used in this let expression?

```
- let val (x,y,z) = (1+s+s2{h}ellop{,}1,true) in println x end
```

   What is the pattern as a Prolog term?
9. Give an example where the *Sequence* rule might be used to infer a type.
10. Give a short example of where the *Handler* rule might be used to infer a type.

8.18 Exercises

1. The following program does not compile correctly or typecheck correctly using the mlcomp compiler and type inference system. However, it is a valid Standard ML program. Modify both the mlcomp compiler and type checker to correctly compile and infer its type. This program is included in the compiler project as test20.sml.

```
let val [(x,y,z)] = [(l+s+s2{h}ellop{,}1,true)] in println x end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(bindval(listpat([tuplepat([idpat(x),idpat(y),idpat(z)])]),
       listcon([tuple([str("hello"),int(1),bool(true)])])),
       [apply(id(println),id(x))])
val [(x,y,z)] : (str * int * bool) list
val it : unit
The program passed the typechecker.
```

2. Implement the Prolog type predicates to get the following program to type check successfully. This program is test14.sml in the mlcomp compiler project. This will involve writing type checking predicates for matching, boolean patterns, integer patterns, and sequential execution.

```
let fun f(true,x) = (println(x); g(x-1))
      | f(false,x) = g(x-1)
    and g 0 = ()
      | g x = f(true,x)
in
       g(10)
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(funmatches([funmatch(f,[match(tuplepat([boolpat
    (true),idpat(x)]),
        expsequence([apply(id(println),id(x)),apply(id(
            g),apply(id(-),
        tuple([id(x),int(1)]))])]),match(tuplepat([
            boolpat(false),idpat(x)]),
        apply(id(g),apply(id(-),tuple([id(x),int(1)])))
            )]),funmatch(g,[match(intpat(0),
        tuple([])),match(idpat(x),apply(id(f),tuple([
            bool(true),id(x)]))])])]),
        [apply(id(g),int(10))])
val f = fn : bool * int -> unit
val g = fn : int -> unit
val it : unit
The program passed the typechecker.
```

3. Implement enough of the type checker to get test12.sml to type check correctly. This will mean writing the *WhileDo* inference rule as a Prolog predicate, implementing the *Match* rule's predicate called *typecheckMatch*, and the type inference predicate for sequential execution named *typecheckSequence* as defined in the *Sequence* rule. The code for test12.sml is given here for reference.

```
let val zero = 0
    fun fib n =
    let val i = ref zero
        val current = ref 0
        val next = ref 1
        val tmp = ref 0
    in
      while !i < n do (
        tmp := !next + !current;
        current := !next;
        next := !tmp;
        i := !i + 1
      );
      !current
    end
    val x = Int.fromString(input(l+s+s2{"lease enter an integer:"))
    val r = fib(x)
in
  print l+s+s2{F}ib(p{;}
  print x;
  print l+s+s2{()} is p{;}
  println r
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(bindval(idpat(zero),int(0)),[letdec(funmatches
    ([funmatch(fib,
    [match(idpat(n),letdec(bindval(idpat(i),apply(id(
        ref),id(zero))),
    [letdec(bindval(idpat(current),apply(id(ref),int(0)
        )),
    [letdec(bindval(idpat(next),apply(id(ref),int(1))),
    [letdec(bindval(idpat(tmp),apply(id(ref),int(0))),
    [whiledo(apply(id(<),tuple([apply(id(!),id(i)),id(n
        )])),
    expsequence([apply(id(:=),tuple([id(tmp),apply(id
        (+),tuple([apply(id(!),id(next)),
    apply(id(!),id(current))])])),apply(id(:=),tuple([
        id(current),apply(id(!),
    id(next))])),apply(id(:=),tuple([id(next),apply(id
        (!),id(tmp))])),apply(id(:=),
    tuple([id(i),apply(id(+),tuple([apply(id(!),id(i)),
        int(1)])])])])),apply(id(!),
    id(current))])])])])])]),[letdec(bindval(idpat(x),
        apply(id(Int.fromString),
    apply(id(input),str("Please enter an integer:")))),
```

```
    [letdec(bindval(idpat(r),apply(id(fib),id(x))),[
       apply(id(print),str("Fib(")),
    apply(id(print),id(x)),apply(id(print),str(") is"))
       ,apply(id(println),id(r))])])])
val zero : int
val i : int ref
val current : int ref
val next : int ref
val tmp : int ref
val fib = fn : int -> int
val x : int
val r : int
val it : unit
The program passed the typechecker.
```

4. Add support to the type checker to correctly infer the types of *case* expressions in Small. The following program should type check correctly once this project is completed. This test is in test15.sml in the mlcomp compiler project. This will involve writing code to correctly type check matches according to the *Match* rule. If case statements are not yet implemented in the compiler, support must be added to the compiler to parse *case* expressions, build an AST for them, and write their AST to the *a.term* file.

```
let val x = 4
in
  println
    (case x of
      1 => "hello"
    | 2 => "how"
    | 3 => "are"
    | 4 => "you")
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(bindval(idpat(x),int(6)),[apply(id(println),caseof(id(x),
       [match(intpat(1),str("hello")),match(intpat(2),str("how")),
       match(intpat(3),str("are")),match(intpat(4),str("you"))]))])
val x : int
val it : unit
The program passed the typechecker.
```

5. Add support to the type checker to correctly infer the types for test7.sml. The code is provided below for reference. Support will need to be added to infer the types of anonymous functions defined in the rule *AnonFun*, matching defined in the rule *Matches*, and the *ConsPat* rule.

```
let fun append nil L = L
      | append (h{:}{:}t) L = h {:}{:} (append t L)
    fun appendOne x = (fn nil => (fn L => L)
                        | h{:}{:}t => (fn L => h {:}{:} (appendOne t L
                            ))) x
in
  println(append [1,2,3] [4]);
  println(appendOne [1,2,3] [4])
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(funmatches([funmatch(append,[match(idpat(v0),
    func(anon@3,
[match(idpat(v1),apply(func(anon@2,[match(tuplepat([
    idpat(nil),idpat(L)]),id(L)),
match(tuplepat([infixpat({:}{:},idpat(h),idpat(t)),
    idpat(L)]),apply(id({:}{:}),
tuple([id(h),apply(apply(id(append),id(t)),id(L))])))
    ]),
tuple([id(v0),id(v1)]))])])])]),[letdec(funmatches([
    funmatch(appendOne,
[match(idpat(x),apply(func(anon@6,[match(idpat(nil),
    func(anon@4,
[match(idpat(L),id(L))])),match(infixpat({:}{:},idpat(
    h),idpat(t)),
func(anon@5,[match(idpat(L),apply(id({:}{:}),tuple([id
    (h),apply(apply(id(appendOne),id(t)),
id(L))])))])])]),id(x))])]),[apply(id(println),apply(
    apply(id(append),
listcon([int(1),int(2),int(3)])),listcon([int(4)]))),
    apply(id(println),
apply(apply(id(appendOne),listcon([int(1),int(2),int
    (3)])),listcon([int(4)]))])])
val append = fn : 'a list -> 'a list -> 'a list
val appendOne = fn : 'a list -> 'a list -> 'a list
val it : unit
The program passed the typechecker.
```

6. Add support for type inference for recursive bindings. The following program, saved as test19.sml in the Small compiler project, is a valid program with a recursive binding. It will type check correctly if the *ValRecDec* type inference rule is implemented. Write the code to get this program to pass the type checker as a valid program.

```
let val rec f = (fn 0 => 1
                  | x => x * (f (x-1)))
in
   println(f 5)
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
    Here is the AST
    letdec(bindvalrec(idpat(f),func(anon@0,[match(
        intpat(0),int(1)),match(idpat(x),
        apply(id(*),tuple([id(x),apply(id(f),apply(id
            (-),tuple([id(x),int(1)]))))]))))]),
        [apply(id(println),apply(id(f),int(5)))])
    val f = fn : int -> int
    val it : unit
    The program passed the typechecker.
```

7. Currently the type checker allows duplicate identifiers in compound patterns like listPat and tuplePat. Standard ML does not allow duplicate identifiers in patterns. The type checker uses the *append* predicate to combine pattern binding environments. This is not good enough. Find the locations in the type checker where pattern environments are incorrectly appended and rewrite this code to enforce that all identifiers within a pattern must be unique. If not, you should print an error message like *"Error: duplicate variable in pattern(s): x"* to indicate the problem and typechecking should end with an error.
8. Currently, the abstract syntax and parser of *Small* includes support for the wildcard pattern in pattern matching, but the type checker does not support it. Add support for wildcard patterns, write a test program, and test the compiler and type checker. Be sure to write a type inference rule for wildcard patterns first.
9. Currently, the abstract syntax and parser of *Small* includes support for the *as* pattern in pattern matching, but the type checker does not support it. Add support for *as* patterns, write a test program, and test the compiler and type checker. The *as* pattern comes up when you write a pattern like *L as h::t* which assigns *L* as a pattern that represents the same value as the compound pattern of *h::t*. Be sure to write a type inference rule for *as* patterns first.

8.19 Solutions to Practice Problems

8.19.1 Solution to Practice Problem 8.1

Proving this requires a proof like was done in the chapter. Rules involved include *Let*, *ValDec*, *IdPat*, *TupleCon*, and *FunApp*. Technically, the *Sequence* rule is also required, but only in the degenerative case (i.e. when $n = 1$).

8.19.2 Solution to Practice Problem 8.2

Minimally the environment must contain *println* bound to a function type of $\alpha \rightarrow unit$ and the $+$ function bound to a function type of $int \times int \rightarrow int$.

8.19.3 Solution to Practice Problem 8.3

The *AnonFun* rule is applied first which requires the *Matches* rule be applied. The *Matches* rule requires the use of the *IdPat* rule and the *FunApp* rule. Finally, the *IntCon* rule is needed to complete the proof.