Four Lectures on Standard ML

The following notes give an overview of Standard ML with emphasis placed on the Modules part of the language.

The notes are, to the best of my knowledge, faithful to "The Definition of Standard ML, Version 2"[1], as regards syntax, semantics and terminology. They have been written so as to be independent of any particular implementation. The exercises in the first 3 lectures can be tackled without the use of a machine, although having access to an implementation will no doubt be beneficial. The project in Lecture 4 presupposes access to an implementation of the full language, including modules. (At present, the Edinburgh compiler does not fall into this category; the author used the New Jersey Standard ML compiler.)

Lecture 1 gives an introduction to ML aimed at the reader who is familiar with some programming language but does not know ML. Both the Core Language and the Modules are covered by way of example.

Lecture 2 discusses the use of ML modules in the development of large programs. A useful methodology for programming with functors, signatures and structures is presented.

Lecture 3 gives a fairly detailed account of the static semantics of ML modules, for those who really want to understand the crucial notions of sharing and signature matching.

Lecture 4 presents a one day project intended to give the student an opportunity of modifying a non-trivial piece of software using functors, signatures and structures.

1 ML at a Glance

Suppose we were to draw a map of the landscape of programming languages. Where would ML fit in? COBOL and ML could safely be put down far apart. The input/output facilities in COBOL operate on specific kinds of input/output devices, for instance allowing the programmer to declare index sequential files. ML just has the notion of STREAMS, a stream being a sequence of characters, much like streams in UNIX or text files in PASCAL. On the other hand, ML is extremely concise compared to the verbose COBOL and ML is much better suited for structuring data and algorithms than COBOL is.

ML is closer related to PASCAL. Like PASCAL, ML has data types and there is a type checker which checks the validity of programs before they are run. Both PASCAL and ML follow the tradition of ALGOL in that variables can have local scope which is determined statically from the source program. However, PASCAL and ML are radically different in how algorithms are expressed. In PASCAL, as in many other languages, a variable can be updated (using :=). Algorithms are often expressed as iterated sequences of statements (using while loops, for instance), where the effect of executing one statement is to change the underlying store. In ML, statements are replaced by EXPRESSIONS; the effect of evaluating an expression is to produce a value. Moreover, variables cannot be updated; REFERENCES are special values that can be updated, and as all other values they can be bound to identifiers, but only rarely are the values one binds to variables references. Iteration is expressed using recursive functions instead of loops. In ML, functions are values which can be passed as arguments to functions and returned as results from functions, and ML programmers do this all the time. ML is an example of a FUNCTIONAL language; PASCAL is an example of a PROCEDURAL language.

LISP is also sometimes referred to as a functional language. In LISP, programs can be treated as data, so that LISP programs directly can decompose and transform LISP programs. This is harder in ML. On the other hand, the type discipline of ML is extremely helpful in detecting many of the mistakes that pass unnoticed in a LISP program.

Like ADA, ML has language constructs for writing large programs. Roughly speaking, a STRUCTURE in ML corresponds to a PACKAGE in ADA; a SIGNATURE corresponds to a PACKAGE INTERFACE and a FUNCTOR in ML corresponds to a GENERIC PACKAGE in ADA. However, ML admits structures (not just types) as parameters to functors.

1.1 An ML session

An ML session is an interactive dialogue between the ML system and the user. You type a PROGRAM in the form of one or more DECLARATIONS (terminated by semicolon) and the system responds either by accepting the declarations or, in case the program is ill-formed, by printing an error message.

To give a concrete idea about what ML programs look like, we shall work through the following example. Consider the problem of implementing heaps. A HEAD is a binary tree of ITEMS, for example:

```
  7
 / \
11 9
 / \
17 15
```
For a binary tree to be a heap, it must satisfy that for every item \( i \) in the tree, \( i \) is less than or equal to all items occurring below \( i \). In the above picture items are integers and the relation "less than or equal" is the normal \( \leq \) on integers. The advantage of a heap is that it always gives fast access to a minimal item and that it is easy to insert and delete items from a heap. This has made the heap a popular data structure in a number of very different applications. It was originally conceived under the name "priority queue" as a means of scheduling processes in an operating system; in that case the items are processes and the partial ordering is that process \( p \) is less than or equal to process \( q \), if \( p \) should be executed no later than \( q \). Heaps are also used in the HEAP SORT algorithm, which is based on the observation that one can sort a list of items by first inserting the items one by one in a heap and then removing them one by one.

1.2 Types and Values

In the following figures we present the ML declarations the author provided in this particular session. The responses from the ML compiler are not shown. For clarity, the actual input has been edited using typewriter font for the reserved words and italics for identifiers, regardless of whether these identifiers are pervasives (e.g. \( \text{int} \)) or declared by the user (e.g. \( \text{item} \)).

| type \( \text{item} = \text{int} \); |
| fun \( \text{leq}(p : \text{item}, q : \text{item}) : \text{bool} = p \leq q \); |
| infix \( \text{leq} \); |
| fun \( \text{max}(p, q) = \text{if} \ p \ \text{leq} \ q \ \text{then} \ q \ \text{else} \ p \) |
| and \( \text{min}(p, q) = \text{if} \ p \ \text{leq} \ q \ \text{then} \ p \ \text{else} \ q \) |
| datatype \( \text{tree} = \text{L} \text{of} \ \text{item} |
| \ | \ N \text{of} \ \text{item} \ast \text{tree} \ast \text{tree}; |
| val \( t = \text{N}(7, \ \text{L} \ 11, \ \text{N}(9, \ \text{L} \ 17, \ \text{L} \ 15)) \); |
| fun \( \text{top}(\text{L} \ i) = i \ |
| \ | \ \text{top}(\text{N}(i, \_ , \_)) = i \); |

We start out by considering integer heaps only; therefore we first declare the type \( \text{item} \) to be an abbreviation for \( \text{int} \). Then we declare a function \( \text{leq} \) to be the pervasive \( \leq \) on integers. We then declare that \( \text{leq} \) is to be used as an infix operator, as illustrated in the declaration of the two functions \( \text{max} \) and \( \text{min} \).

Every binary tree is either a leaf containing an item or it is a node containing an item and two trees (the subtrees). This is expressed by the \textbf{datatype} declaration. \textbf{datatype} declarations are automatically recursive, i.e. data types can be declared in terms of themselves. This is illustrated by the declaration of \( \text{tree} \). This data type has two \textbf{constructors}, \( \text{L} \) and \( \text{N} \). Note that for example 7 is an item, but \( \text{L} \ \text{APPLIED TO} \ 7 \), written \( \text{L}(7) \), or just \( \text{L} \ 7 \), is of type \( \text{tree} \). Then the heap from the earlier picture is bound to the value variable \( t \).

**Exercise 1** Declare a heap \( t' \) of the same depth as \( t \) containing the integers 78, 34, 5,
12, 15, 28, and 9.

To define a function on trees it will suffice to define its value in the case the argument tree is a node and in the case the tree is a node. The declaration of the function \textit{top} illustrates this. (\textit{top} applied to a tree returns the item at the top of the tree). (\textit{L i}) and (\textit{N(i, \_ , \_ )}) are examples of \texttt{PATTERNS}. Applying a function (here \textit{top}) to an argument (e.g. \textit{t}) is done by matching the argument against the patterns till a matching pattern is found. For example \textit{top t} evaluates to 7.

\subsection{Recursive Functions}

\begin{verbatim}
fun depth(L \_) = 1 |
  depth(N(i, l, r)) = 1 + max(depth l, depth r);
depth t;

fun isHeap(L \_) : bool = true |
  isHeap(N(i, l, r)) = 
    i leq top l andalso 
    i leq top r andalso 
    isHeap l andalso 
    isHeap r
\end{verbatim}

The function \textit{depth} maps trees to integers; for instance \textit{depth t} evaluates to 3. As spelled out in the declaration of \textit{depth}, the depth of a leaf is 1 and the depth of any other tree is 1 plus the maximum of the depths of the left and right subtrees. The function \textit{depth} is \texttt{RECURSIVE}, i.e. defined in terms of itself. Another example of a recursive function is the function \textit{isHeap} which when applied to a tree returns the value \texttt{true} if the tree is a heap and \texttt{false} otherwise.

\textbf{Exercise 2} Write a function \textit{size} which when applied to a tree returns the total number of items in the tree.

\textbf{Exercise 3} The function \textit{top} returns a minimal item of a heap. Write a recursive function \textit{maxItem} which returns a maximal item.

\subsection{Raising Exceptions}

One often wants to define a function that cannot return a result for some of its argument values. Suppose, for example, that we wish to define a function \textit{initHeap} which for given integer \textit{n} returns a heap of depth \textit{n}. This only makes sense for \textit{n} \geq 1. This can be expressed in ML by raising an \texttt{EXCEPTION} in the case \textit{n} < 1. The effect of evaluating the expression \texttt{raise e}, where \texttt{e} is an exception, is to discontinue the current evaluation. Often, the exception will be \texttt{HANDLED} by a \texttt{handle} expression (not illustrated by our examples); if no handler catches the exception, it propagates to the top-level where it will be reported as an uncaught exception.

\begin{verbatim}
val initial = 0
exception InitHeap
fun initHeap n = 
  if n<1 then raise InitHeap
  else if n = 1 then L(initial)
  else let val t = initHeap(n - 1)
      in N(initial, t, t)
      end
\end{verbatim}

Notice the \texttt{let} \texttt{dec in exp end} expression. To evaluate it, one first evaluates \texttt{initHeap(n - 1)} and binds the resulting value to \texttt{t}. Then one evaluates the body, \texttt{N(initial, t, t)} using this value for \texttt{t}. Notice that the scope of the declaration of \texttt{t} is the expression
\(N(\text{initial}, t, \ell)\); in particular the two occurrences of \(t\) in that expression do not refer to \(N(7, L\ 11, N(9, L\ 17, L\ 15))\).

**Exercise 4** Define functions \(\text{leftSub}\) and \(\text{rightSub}\) which when applied to a tree returns the left and the right subtree, respectively.

Finally, we shall write a function \(\text{replace}\) which when applied to a pair \((i, h)\), where \(i\) is an item and \(h\) is a heap, returns a pair \((i', h')\), where \(i'\) is the item at the top of the heap \(h\) and \(h'\) is a heap obtained from \(h\) by inserting \(i\) in place of the top of \(h\). We must make sure that the resulting tree really is a heap. Therefore, in the case that \(i\) is to be inserted in a node above a subtree with a smaller item, \(i\) swops place with the smaller item.

If one types an expression followed by a semicolon (such as \(t;\) in the above program) the ML system evaluates the expression and prints the result. In the above example, it will turn out that even after we have “replaced” \(7\) by 10, \(t\) is bound to the original heap. Indeed, this “replacement” in no way affects the value bound to \(t\); it simply results in a new value, which subsequently is bound to \(t\).

**Exercise 6** After the last declaration, what values are bound to \textit{out2} and \(t\)?

```ml
fun replace(i, h) = (top h, insert(i, h))
and insert(i, L_.) = L(i)
  | insert(i, N(_, L, r))=
    if i leq min(top l, top r)
    then N(i, l, r)
    else if (top l) leq (top r) then
      N(top l, insert(i, l), r)
    else (* top r < min(i, top l) *)
      N(top r, l, insert(i, r));

val (out1, t1) = replace(10, t);

val (out2, t2) = replace(20, t1)
```

### 1.5 Structures

The above declarations of heaps and operations on heaps belong together. In ML there is a program unit called a \textsc{structure} which encapsulates a sequence of declarations. The following declaration declares a structure \textit{Heap} containing all the declarations (copied from above) encapsulated by \texttt{struct} and \texttt{end}.

The special parenthesis (* and *) delimit comments.

**Exercise 5** In the case where one recursively inserts \(i\) in the left subtree, how can one be sure that it is valid to put \(top\ \ell\) above \(r\) in the tree?
structure Heap =
struct
  type item = int;
  fun leq(p; item: q: item): bool = p <= q;
  fun max(p, q) = ...
  and min(p, q) = ...
  datatype tree = L of item
                 | N of item * tree * tree;
  val t = ...
  fun top(L i) = ...
  fun depth(L _) = ...
  fun isHeap(L _): bool = ...
  val initial = 0
  exception InitHeap
  fun initHeap n = ...
  fun replace(i, h) = ...
  and insert(i, _ _) = ...
end; (* Heap *)

val smallHeap = Heap.initHeap(1);
Heap.replace(20, smallHeap);

signature HEAP =
sig
  type item
  val leq: item * item -> bool
  val max: item * item -> item
  val min: item * item -> item
  datatype tree = L of item
                 | N of item * tree * tree
  val t: item
  val top: tree -> item
  val depth: tree -> int
  val isHeap: tree -> bool
  val initial: item
  exception InitHeap
  val initHeap: int -> tree
  val replace: item * tree -> item * tree
  val insert: item * tree -> tree
end; (* HEAP *)

As one can check, the structure Heap matches signature HEAP in the following sense: for every type specified in HEAP, there is a corresponding type in Heap; for every exception specified in HEAP, there is a corresponding exception in Heap, and for every value specified in HEAP there is a corresponding value in Heap which has the specified type.

1.6 Signatures

The “type” of a structure is called a Signature. A signature can specify types, without necessarily saying what the types are. Moreover, one can specify values (in particular functions) by specifying a type for each variable, without saying how such a specification can be met by an actual declaration.

1.7 Coercive Signature Matching

However, the signature HEAP reflects details of the implementation in Heap which heap users should not have to worry about. (Obviously, the value t is completely unnecessary, and there is no reason why users should have access to the constructors L and N given that we have already given the user initHeap and replace.) By pruning the signature we obtain
the following shorter declaration of \textit{HEAP}.

\begin{verbatim}
signature HEAP =
  sig
    type item
    val leq: item * item -> bool
  type tree
  val top: tree -> item
  exception InitHeap
  val initHeap: int -> tree
  val replace: item * tree -> item * tree
end; (* HEAP *)
\end{verbatim}

This is a much cleaner interface, so whenever we refer to \textit{HEAP} in the following, we mean this version.

In practice, one should write down a signature \textit{before} one attempts to write down a structure which matches it. In this way one can decide what types and operations are needed without having to think about algorithms at the same time. So let us assume that we started out by declaring the \textit{HEAP} signature. We then imprint the view provided by \textit{HEAP} on the declaration of the structure \textit{Heap} by a \textsc{signature constraint}:

\begin{verbatim}
structure Heap: HEAP =
  struct
    type item = int;
    ...
  end; (* Heap *)
\end{verbatim}

? Heap.t
Heap.replace(7, Heap.initHeap 3);

After this declaration of \textit{Heap}, we cannot write \textit{Heap.t}, since \texttt{t} is not mentioned in \textit{HEAP}. However, we can write \textit{Heap.replace} as \textit{replace} is specified. Moreover, although \textit{HEAP} does not specify that \textit{item} should be \textit{int}, the ML system discovers that \textit{item} is in fact \textit{int} in \textit{Heap} and that is why 7 will be accepted as an \textit{item} in the application \textit{Heap.replace}(7, \textit{Heap.initHeap} 3). Thus a signature constraint may hide components of a structure, but it does not hide the true identity of the types declared in the structure, except that one can hide the constructors of a \texttt{datatype} by specifying it as a \texttt{type}.

\subsection{Functor Declaration}

Almost all of what we did for heaps containing integer items would work for a heap whose items are of a different type. More precisely, given any type \textit{item}, any binary function \textit{leq} on items and any \textit{initial} item, the signature \textit{HEAP} is satisfied by the declarations we have already written. Let us specify the general requirements of a \textit{Heap} structure.

\begin{verbatim}
signature ITEM =
  sig
    type item
    val leq: item * item -> bool
    val initial: item
  end;
\end{verbatim}

What we are after is a structure which is parameterised on any structure, \textit{Item}, say, which matches \textit{ITEM}. In ML, a parameterised structure is called a \textsc{functor}. The following table contains the complete functor declaration; the new bits are in bold face.
functor Heap(Item: ITEM): HEAP =
  struct
    type item = Item.item
    fun leq(p: item, q: item): bool =
      Item.leq(p,q)
    fun intmax(i: int, j) =
      if i <= j then i else j
    infix leq;
    fun max(p, q) = if p leq q then q else p
      and min(p, q) = if p leq q then p else q
    datatype tree = L of item
      | N of item * tree * tree;
    fun top(L i) = i
      | top(N(i, _, _)) = i;
    fun depth(L _) = 1
      | depth(N(i, l, r)) =
        1 + intmax(depth l, depth r);
    fun isHeap(L _): bool = true
      | isHeap(N(i, l, r)) =
        i leq top l andalso
        i leq top r andalso
        isHeap l andalso
        isHeap r
  exception InitHeap
  fun initHeap n =
    if n<1 then raise InitHeap
    else if n = 1 then L(Item.initial)
    else let val t = initHeap(n - 1)
      in N(Item.initial, t, t)
      end
    fun replace(i, h) = (top h, insert(i, h))
    and insert(i, L _)= L(i)
      | insert(i, N(_, l, r))=
        if i leq min(top l, top r)
          then N(i, l, r)
        else if (top l) leq (top r) then
          N(top l, insert(i, l, r))
        else (* top r < min(i, top l) *)
          N(top r, l, insert(i, r));
    end; (* Heap *)

In the first line, Item is the parameter structure of the functor and HEAP is the result signature of the functor. The body of the functor is everything after the = in the first line.

Notice that we included declarations of item and leq in the body of the functor, since the result signature specifies them, they must be provided. If you read the body carefully, you will see that it makes sense for any structure which matches ITEM.

**Exercise 7** Declare a functor Pair which takes as a parameter a structure matching the simple signature

    sig type coord end

and has the following result signature:

    sig
type point
val mkPoint: coord * coord -> point
val x coord: point -> coord
val y coord: point -> coord
end

You do not have to name these signatures (by the use of signature declarations); they can be written down directly where you need them, if you prefer.

**Exercise 8** When the author first tried to write the Heap functor, he simply copied the original depth function which used max, not intmax. However, the type checker did not let him get away with that. Why?

### 1.9 Functor Application

We can now get various heaps (indeed heaps of heaps) by applying the Heap functor to different argument structures. Of course, we can only apply it to structures that match ITEM; this will be checked by the compiler.

Here is how one can get a string heap:
structure StringItem =
struct
    type item = string
    fun leq(i:item, j) =
        ord(i) <= ord(j)
    val initial = " "
end;

structure StringHeap = Heap(StringItem)

val (out1, tl) =
    StringHeap.replace("abe",
    StringHeap.initHeap(1));
val (out2, t2) =
    StringHeap.replace("man", tl);

The pervasive ord function applied to a string s returns the ASCII ordinal value of the first character in s, and raises exception Ord when s is empty.

Exercise 9 Declare a structure IntItem using the declarations we originally used for integer heaps. Then obtain a structure IntHeap by functor application.

Exercise 10 How does one get an integer heap whose top is always maximal?

Exercise 11 Declare a structure IntHeapHeap whose items themselves are integer heaps. (You can use the top function to define a leq function on integer heaps.)

1.10 Summary

ML consists of a core language and a modules language. The core language has values (functions are values), data types, type abbreviations and exceptions. The modules language has structures, signatures and functors. There is no actual language construct
2 Programming with ML Modules

2.1 Introduction

This lecture gives a more thorough introduction to the modules part of ML and describes a methodology for programming with its main constructs: structures, signatures and functors.

The core language is interactive: you type a declaration, get a reply, type another declaration and so on, thus gradually adding more and more bindings to the top-level environment. If we could think strictly bottom-up, declaring one value or type in terms of the preceding values and types, without ever making unfortunate implementation decisions or losing the perspective of the entire project, then this gradual expansion of the top-level environment would be quite sufficient. Unfortunately, we cannot, indeed a program which is written as one long list of core language declarations can easily end up looking rather like a long shopping list where items have been added in the order they came to mind.

Regardless of whether a programming language is interactive or not, one needs the ability to divide large programs into relatively independent units which can be written, read, compiled and changed in relative isolation from each other.

One approach, taken by some, is to provide more or less language independent software packages that help programmers organise collections of programs typically by allowing (or forcing) them to document their programs in specific ways. The crucial problem with this approach is of course to ensure consistency between the documentation and the programs, in particular to ensure that the information held by the tool really is sufficient to ensure that the constituent units can be put together in a consistent manner.

Another approach, taken in several programming languages (e.g. Ada and ML), is to provide module facilities in the programming language itself. Many of the operations one needs when programming with modules are similar to operations one needs when programming in the small, so many ideas from usual programming languages apply to programming in the large as well. For instance, just as it is a type error (in the small) to add \textit{true} and 7, say, so it is a type error (in the large) to write a module \texttt{M2}, say, assuming the existence of a module \texttt{M1} which provides a function \texttt{f}, and then combine \texttt{M2} with an actual module \texttt{M1} which either does not provide any \texttt{f} or provides an \texttt{f} of the wrong type. The idea is that such mistakes should be detected by a type checker at the modules level.

This leads to the exciting idea of having just one language with constructs that work uniformly for “small” as well as for “large” programs. One such language is Pebble by Burstall and Lampson. In Pebble records can contain types, so a module consisting of a collection of types and values is now itself a value, which for example can be passed as an argument to a function. There are some trade-offs, however. The ML type checker is based on a strict separation of run-time and compile-time. In designing the modules language it has been necessary to restrict the operations on types in comparison with the operations on values in order to maintain the static type checking. This has led to a stratified language, in which the modules language contains phrases from the core language, but not the other way around.

I shall use the term “module” rather vaguely to mean “a relatively independent program unit”. In particular languages they have been called “packages”, “clusters”,

9
"modules" and in ML we use the word "structure".

Likewise, there is no standard terminology for "the type of a module", which has acquired names such as "package description", "interface" and the ML term "signature".

As we shall see, the real power of a modules system comes from the ability to parameterise modules. Ada has "generic packages". ML has "functors".

In ML, a structure is a collection of data types, types, values, exceptions and even other structures. A signature specifies types and data types and gives the types of values and exceptions. A functor is essentially a function from structures to structures. Functors cannot take functors as arguments, nor can they produce functors as results. The purpose of this lecture is to convince you that even this apparently simple notion of functor constitutes a powerful extension of the core language. As will be demonstrated, one can write an entire system using just signatures and functors and then build the system using functor applications.

Imagine we want to write a parser for a programming language. In order to build the parser top-down, we might start by sketching the parser itself. However, as programming normally is a complex process involving both the odd low-level implementation idea and more high-level considerations about overall structure, let us start at an intermediate level, the problem of writing a symbol table. (A symbol table is simply a facility which allows one to store and retrieve information about symbols.)

2.2 Signatures

The way one in ML sketches a structure is to write down a signature. Here is a first sketch of a symbol table signature, called \textit{OTable} as it is opaque in the sense that it does not reveal many implementation details.

\begin{verbatim}
signature OTable =
sig
  type table
  exception Lookup
  val lookup: table * Sym.sym -> Val.value
  val update: table * Sym.sym * Val.value -> table
end
\end{verbatim}

At this early stage, we cannot know exactly what symbols are going to be; nor can we know what kind of values we are going to store with the symbols. Therefore we imagine structures \textit{Sym} and \textit{Val} which declare the types \textit{sym} and \textit{value}, respectively. \textit{Sym.sym} is an example of a \textsc{long identifier}, in this case a \textsc{long type constructor}. The two structure identifiers \textit{Sym} and \textit{Val} are \textsc{free} in \textit{OTable}.

There are many different ways of implementing a symbol table which matches this signature. One possibility is to use an association list, i.e. a list of pairs of symbols and values. Since the symbol table is going to be used extensively, we will probably want something more efficient. We cannot use an array, for arrays map integers (rather than symbols) to values. (Actually, the ML language definition does not include arrays, but they are provided in most implementations). But we can implement the symbol table as a hash table: we can require that the \textit{Sym} structure provides a hash function from symbols to integers and then assume the existence of another structure, \textit{Int.Map}, which implements maps on the integers. Since the hash function may map different symbols to the same integer, we take an \textit{Int.Map} which maps integers
to lists of pairs of symbols and values:

signature TTTable =
  sig
  datatype table = TBL of
    (Sym.sym * Val.value) list IntMap.map
  exception Lookup
  val lookup: table * Sym.sym -> Val.value
  val update: table * Sym.sym * Val.value
                -> table
end

2.3 Structures

Here is a structure which implements a symbol table.

structure SymTbl =
  struct
  datatype table = TBL of
    (Sym.sym * Val.value) list IntMap.map
  exception Lookup

  fun find(sym, [ ]) = raise Lookup
  | find(sym, (sym', v)::rest) =
    if sym = sym' then v
    else find(sym, rest)

  fun lookup(TBL map, s) =
    let val n = Sym.hash(s)
    val l = IntMap.apply(map, n)
    in find(s,l)
    end handle IntMap.NotFound =>
      raise Lookup
  ...
end

When binding a structure to a structure identifier one can impose a signature constraint on the structure.

structure SymTbl : OTable =
  struct
  ...
  end

As a result, all identifiers of the structure that are not mentioned in the signature are hidden. In the above example we hide the constructor TBL and the function find. Besides update, the ... in SymTbl may declare extra values, exceptions and types, but as a result of the signature constraint, none of these extra components will be visible from outside the structure.

It is often the case that there is not a single signature which “best” constrains a given structure because different parts of the program should see different degrees of details of the structure. (The parser should be written using a opaque signature for the symbol table; by contrast, a structure which prints out the symbol table (for testing, for example) will need to know more details).

During the design of ML it was decided that it is vital to admit different views of the same structure. One way of achieving this is to bind the structure to more that one structure identifier, each time using a different signature constraint.

structure SymTbl : TTTable =
  struct
  datatype table = TBL of
    (Sym.sym * Val.value) list IntMap.map
  exception Lookup

fun find(sym, [ ]) = raise Lookup
| find(sym, (sym', v)::rest) =
| if sym = sym' then v
| else find(sym, rest)

fun lookup(TBL map, s) =
  let val n = Sym.hash(s)
  val l = IntMap.apply(map, n)
  in find(s, l)
  end handle IntMap.NotFound =>
    raise Lookup
  end

structure SmallTbl: OTable = SymTbl

The dynamic evaluation of the
struct...end yields an environment just as
if we had typed the constituent declarations
at top-level. Dynamically, there is just one
lookup function, for example, and as a re-
result of the above declarations, this function
is shared between SymTbl and SmallTbl.

Statically, however, the elaboration of the
above declarations yields two different views
of this environment. Since there is just
one lookup function, we should of course
be free to refer to it as SymTbl.lookup or
SmallTbl.lookup, whichever we prefer. This
requires that these two long identifiers have
the same type; so the static semantics must
be such that the types SymTbl.table and
SmallTbl.table are considered shared.

2.4 Functors

Unfortunately, neither the declaration of the
signature OTable nor the declaration of the
structure SymTbl makes sense on its own.
The reason is that they both contain free
identifiers. OTable relies on structures Val
and Sym and SymTbl in addition relies on
IntMap. As a consequence, we can compile
neither OTable nor SymTbl in the initial top-
level environment.

What we need to achieve this is clearly the
ability to abstract both OTable and SymTbl
on their free identifiers. Such an abstraction
is called a FUNCTOR in ML:

functor SymTblFct(
  structure IntMap: IntMapSig
  structure Val: ValSig
  structure Sym: SymSig):

  sig
  type table
  exception Lookup
  val lookup: table * Sym.sym => Val.value
  val update: table * Sym.sym * Val.value
                  => table
  end =

  struct

    datatype table = TBL of
      (Sym.sym * Val.value) list IntMap.map
    exception Lookup

    fun find(sym, [ ]) = raise Lookup
    | find(sym, (sym', v)::rest) =
      if sym = sym' then v
      else find(sym, rest)

    fun lookup(TBL map, s) =
      let val n = Sym.hash(s)
      val l = IntMap.apply(map, n)
      in find(s, l)
      end handle IntMap.NotFound =>
        raise Lookup
      end

end
Now the Sym, Val and IntMap that occur in
the result signature and in the functor body
are bound as formal parameters of the functor.
Of course for this functor declaration to
make sense, we must first declare the three
signatures IntMapSig, ValSig and SymSig,
but that can be done without worrying about
how we get the corresponding structures.
Indeed, declaring these signatures is healthy ex-
ercise, as it makes us summarise what a sym-
bol table needs to know about symbols, val-
ues and intmaps.

Exercise 12 Declare the signatures
IntMapSig, ValSig and SymSig. Also com-
plete the functor body by declaring update,
extending your signatures, if needed.

When, in due course, we have defined
actual structures FastIntMap, Data and
Identifier, say, corresponding to the formal
structures IntMap, Val and Sym, respec-
tively, we can obtain a particular symbol ta-
ble by applying the symbol table functor.

```plaintext
structure MyTbl =
    SymTblFct(structure IntMap = FastIntMap
        structure Val = Data
        structure Sym = Identifier)
```

Dynamically, the functor body is not eval-
uated when the functor is declared, but once
for each time the functor is applied. (In this
respect, functors behave a functions in the
core language.)

As part of the functor application, the com-
piler will check to see whether the actual argu-
ment structures match the specified signa-
tures. If that is not the case (for instance,
if Identifier does not contain a type sym or
FastIntMap, apply takes three instead of two
arguments) then an error will be reported and
hence we are prevented from putting together
the inconsistent structures.

What is the signature of MyTbl? It is
not simply the result signature of SymTblFct,
for that signature refers to the formal func-
tor parameters Val and Sym. Clearly, if
Identifier.sym is string and Data.value is
real, then we should be able to write for in-
stance

```
sqrt(MyTbl.lookup(t, “pi”))
```

The signature of MyTbl, therefore, is ob-
tained by substituting the types of the actual
arguments for the types in the formal result
signature of SymTblFct.

```plaintext
sig
type table
exception Lookup
val lookup: table * Identifier.sym
    => Data.value
val update: table * Identifier.sym
    * Data.value => table
end
```

### 2.5 Substructures

As we saw above, the signature of the result
of a functor application depends on the ac-
tual arguments to the functor. So apparently
there is no single signature which describes
all the symbol tables which can be created
by applying SymTblFct. But how, then, are
we going to declare a functor ParseFct, say,
which we can apply to any symbol table cre-
at by SymTblFct?
The solution to this problem is to make explicit in the symbol table signature that any symbol table depends on a `Val` and a `Sym` structure.

| signature SymTblSig =  
| sig  
| structure Val: ValSig  
| structure Sym: SymSig  
| type table  
| val lookup: table * Sym.sym -> Val.value  
| val update: table * Sym.sym * Val.value -> table  
| end |

The specifications of `Val` and `Sym` no longer refer to particular structures outside the signature, i.e., `Val` and `Sym` are now considered bound in the signature. (Of course the signature identifiers `SymSig` and `ValSig` are still free, but those you have already declared in the exercise.)

The idea is that a structures can contain not just values, exceptions and types as components, but even other structures. These are called the SUBSTRUCTURES of the structure. To make the result of `SymTblFct` match `SymTblSig`, we have to declare structures `Val` and `Sym` in the body. But that is easily done; we simply bind them to the formal parameters.

| functor SymTblFct(  
|   structure IntMap: IntMapSig  
|   structure Val: ValSig  
|   structure Sym: SymSig): SymTblSig =  
| struct  
|   structure Val = Val  
|   structure Sym = Sym  
| end |

datatype table = TBL of (Sym.sym * Val.value)list IntMap.map
exception Lookup

fun find(sym,[ ]) = raise Lookup
|   find(sym,(sym',v)::rest) =  
|     if sym = sym' then v  
|     else find(sym,rest) |

fun lookup(TBL map, s) =  
|   let val n = Sym.hash(s)  
|     val l = IntMap.apply(map,n)  
|     in find(s,l)  
|     end handle IntMap.NotFound =>  
|       raise Lookup  
|     end  

| ... |

### 2.6 Sharing

A signature for lexical analysers might be as follows (a lexical analyser reads individual characters from an input file and assembles them into symbols — in the case of a pro-gramming language typically reserved words and identifiers):

| signature LexSig =  
| sig  
| structure Sym : SymSig  
| val getsym : unit -> Sym.sym  
| end |

We have included the specification of a substructure `Sym` because a lexical analyser needs to know about symbols. (Indeed, if we want to declare `LexSig` before defining any particular `Sym` structure, we are forced to include the substructure specification.)
Here is a first attempt at defining ParseFct.

```ml
functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig) =
  struct
    ...
    let val next = Lex.getsym()
    in SymTbl.update(table, next, "declared")
    end
end
```

However, the let expression in the body is not type correct! Since the type of `getsym()` is `Lex.Sym.sym`, `next` has type `Lex.Sym.sym`. However, by the specification of `update`, its second argument must be of type `SymTbl.Sym.sym`. The problem is that although we have specified that `SymTbl` depends on a `Sym` structure and `Lex` depends on a `Sym` structure, nowhere have we specified that they depend on the same `Sym` structure. The type checker will not make an attempt to identify these two types, for the idea is that the functor should be applicable to any arguments that satisfy the formal parameter specification (not just those that satisfy the specification and in addition have extra sharing). Therefore one is allowed to specify needed sharing as well as needed components by a so-called SHARING SPECIFICATION. Grammatically, a sharing specification can occur anywhere amongst structure, type, value and exception specifications.

```ml
functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig
  sharing SymTbl.Sym = Lex.Sym
  and type SymTbl.Val.value = string) =
  struct
    ...
    let val next = Lex.getsym()
    in SymTbl.update(table, next, "declared")
    end
end
```

One can specify sharing of structures and of types (but not of values or exceptions). In our example, we have to add yet a sharing specification, this time a type sharing specification.

```ml
functor ParseFct(
  structure SymTbl: SymTblSig
  structure Lex: LexSig
  sharing SymTbl.Sym = Lex.Sym
  and sharing SymTbl.Val.value = string) =
  struct
    ...
    let val next = Lex.getsym()
    in SymTbl.update(table, next, "declared")
    end
end
```

### 2.7 Building the System

Notice that we have now written the code of the parser solely by declaring signatures and functors. We have not had to write a single top-level structure declaration. Having finished declaring the parser functor, we can return to the basics and declare functors that implement `Sym` and `Val`. These functors can be NULLARY, i.e., have an empty specification of formal parameters.

**Exercise 13** Write nullary functors `ValFct`, `SymFct` and `IntMapFct` whose result match your signatures from the previous exercise.
We can now build the entire system by functor applications and top-level structure declarations.

```plaintext
structure Val = ValFct()
structure Sym = SymFct()
structure TTable =
  SymTblFct(structure IntMap=IntMapFct() )
  structure Val = Val
  structure Sym = Sym
structure Lex = LexFct(Sym)
structure Parser =
  ParseFct(structure SymTbl = TTable
                        structure Lex = Lex)
```

The compiler will check that the sharing specified in the declaration of ParseFct really is met by the actual arguments.

**Exercise 14** What is wrong with the following attempt to build the parser?

```plaintext
structure Val = ValFct()
structure TTable =
  SymTblFct(structure IntMap=IntMapFct() )
  structure Val = Val
  structure Sym = SymFct()
structure Lex = LexFct(SymFct())
structure Parser =
  ParseFct(structure SymTbl = TTable
                        structure Lex = Lex)
```

### 2.8 Separate Compilation

Some ML implementations have facilities that allow you to compile declarations, for instance functor declarations, in such a way that the compiled code can persist between sessions. However, even without such a facility, using signatures and functors in the manner described above gives the valuable ability to separately compile modules consisting of signature and functor declarations, although the result of the compilation will not outlive the session.

Most ML systems have a `use` function which allows the ML source to be read from a file rather than from the terminal. One can then keep signatures in suitably named files and use these files at the beginning of each module.

```plaintext
use "symb.sig";
use "val.sig";
use "symtbl.sig";
use "lex.sig";
use "parse.sig";

functor ParseFct(  
  structure SymTbl: SymTblSig  
  structure Lex: LexSig  
  sharing SymTbl, Sym = Lex.Sym  
  and type SymTbl, Val, value = string): ParseSig =
  struct
    ...
  end
```

In this way one avoids repeating the same signature declaration in many files (and thus also the problem of updating all copies if the signature is changed).
2.9 Good Style

It is good practice to keep signatures as small as possible. If one programs using functors and signatures as described above then writing the body of a functor will reveal which components of its formal parameters that particular functor needs to know about.

Different functors will need different details. Rather than gradually extending a single signature till it gets very large, one can use the include specification to enrich an existing signature.

```plaintext
signature SmallTbl = sig... end

signature BigTbl =
sig
  include SmallTbl
datatype DebugInfo = ...
  val printInfo : unit->unit
end
```

2.10 Bad Style

Signature declarations can contain free structure and type identifiers.

Structure declarations can contain free identifiers of any kind.

This allows you to write for example

```plaintext
structure Parser =
  struct
    structure Lex = Lex
    structure MyPervasives = MyPervasives
    structure ErrorReports = ErrorReports
    structure PrintFcs = PrintFcs
    structure Table = Table
    structure BigTable = BigTable
    structure Aux = Aux
  fun f(...) = ... Table.lookup ...
end
```

Here, the programmer has apparently made some effort to show that the parser depends on the structures listed at the beginning. However, if he has missed out a couple of structures from his list, it will have no effect on the declarations that follow, and so one does not as a reader feel confident that the list is exhaustive.

Moreover, when the reader wants to find out what the type of the lookup function is, he has to look in the declaration of the Table structure. In case Table is constrained by a signature, the search continues in the declaration of the signature. Otherwise, one will have to look at the code for lookup.

Most serious of all, when encountering the call of lookup one has no idea whether lookup has side effects that are important to other structures. In that case, the value of structuring code into structures and substructures is purely cosmetic. The only reliable help it gives you is a pointer to the structure in which the identifier is declared.

One particular horror is the misuse of open. Available both in the core language and in the modules language, open S is a declaration which has the effect of adding all the bindings
of the structure $S$ to the current environment. This is helpful, if one has a single structure `MyPervasives`, say, which is used everywhere in the project. But look at this:

```plaintext
structure Parser =
struct
  structure Lex = Lex
  open MyPervasives ErrorReports PrintFns
    Table BigTable Aux

  fun f(...) = ... lookup ... end
```

Now finding `lookup` is reduced to pure guesswork!

**Exercise 15** For each of the above points of criticism, consider to what extent it applies if one programs with signatures and functors only.
3 The Static Semantics of Modules

The purpose of this lecture is to explain the static semantics of modules. In particular, we shall look into the details of the crucial concepts SIGNATURE MATCHING and SHARING.

3.1 Elaboration

Consider the two following signatures, the first of which stem from the *MyTbl* example of Lecture 1.

```
sig
type table
exception Lookup
val lookup: table * Identifier.sym
    -> Data.value
val update: table * Identifier.sym
    * Data.value -> table
end
```

```
sig
type table
exception Lookup
val lookup: table * string -> real
val update: table * string * real -> table
end
```

In one sense, these signatures are very different; the meaning of the first one depends on the free structures *Data* and *Identifier*, whereas the second depends on the pervasives only. However, if *Identifier.sym* happens to be *string* and *Data.value* happens to be *real* then the two expression are just different ways of expressing the same meaning. In that sense, the two signatures turn out to be equal.

To avoid such confusion concerning equality, it is often helpful to distinguish between a SIGNATURE EXPRESSION (the syntactic object) and a SIGNATURE (its meaning). The transition from signature expressions to signatures is called ELABORATION. We use the word elaboration instead of evaluation, because, unlike evaluation, all elaboration can be done statically, by a compiler. The result of elaborating a signature expression depends on the meaning of the identifiers occurring free in the expression. In any given context, there are infinitely many signature expressions that elaborate to the same signature. It is even the case that in every context, every signature expression elaborates to infinitely many signatures, if it elaborates to any at all. However, among these there will always be some that are PRINCIPAL which means that, in a certain technical sense, all the others are instances of them, and one always takes a principal signature as the meaning of a signature declared at top-level.

Elaboration applies to STRUCTURE EXPRESSIONS and FUNCTOR DECLARATIONS as well, yielding STRUCTURES and FUNCTOR SIGNATURES, respectively.

Essentially, the modules part of ML is a language for computing these (abstract) signatures, structures and functor signatures. The purpose of this lecture is to explain the principles that govern elaboration.

We shall not introduce a separate notation for structures, signatures and functor signatures. In many cases these are very similar to the expressions from which they were obtained, so we make do with the device of "decorating" expressions with so-called names. Names are semantic objects, completely distinct from identifiers; in the above examples, *string* and *Identifier.sym* are both identifiers which elaborate to the same type name.
3.2 Names

structure Stack =
  struct
    type elt = int
  datatype stack = ST of elt list ref
  val initStack = ST(ref[ ]) end

structure StackUser1 =
  struct
    structure Stack1 = Stack
    ...
  end

structure StackUser2 =
  struct
    structure Stack2 = Stack
    ...
    datatype stack = ST of elt list ref
  end

All the following sharing equations hold: StackUser1.Stack1 = StackUser2.Stack2, elt = int, Stack.stack = StackUser1.Stack1.stack. None of the following sharing equations hold: StackUser1 = StackUser2, Stack.stack = StackUser2.stack.

Sharing equations can be decided by decorating programs with NAMES. There are two kinds:

structure names: n1, n2, ..., m1, m2, ...

type names: t1, t2, ..., s1, s2, ..., unit, int, bool, →

Two structures SHARE if they are decorated by the same structure name; two types SHARE if they are decorated by the same type name.

3.3 Decorating Structures

Each elaboration of a structure expression of the form

  struct ... end

yields a fresh structure, i.e., a structure decorated by a new name. Therefore such expressions are called GENERATIVE STRUCTURE EXPRESSIONS.

Each elaboration of a data type declaration (datatype ...) yields a fresh type i.e., a type decorated by a new name.

structure Stack_n1 =
  struct
    type elt_int = int
  datatype stack_t1 = ST of elt list ref
  val initStack_t1 = ST(ref[ ]) end

structure StackUser1_n2 =
  struct
    structure Stack1_n1 = Stack
    ...
  end

structure StackUser2_n3 =
  struct
    structure Stack2_n1 = Stack
    ...
    datatype stack_t23 = ST of elt list ref
  end

To be complete, one would have to decorate each structure not merely by a name but also with its decorated components and subcomponents. (A structure expression in the form of a functor application does not in itself reveal the components of the resulting structure.) However, to keep decorations to a minimum, we shall usually not spell out the decorated subcomponents.
3.4 Decorating Signatures

signature $\text{StackSig}_{\text{m1}, \text{s1}, \text{s2}}$ =
    sig $\text{m1}$
    type $\text{clt}_{\text{s1}}$
    type $\text{stack}_{\text{s2}}$
    val $\text{new}_{\text{unit} \rightarrow \text{stack}}$
end

signature $\text{TranspSig}_{\text{m1}, \text{s1}}$ =
    sig $\text{m1}$
    type $\text{clt}_{\text{s1}}$
    type $\text{stack}_{\text{t1}}$
    sharing type $\text{stack}_{\text{t1}} = \text{Stack} \cdot \text{stack}_{\text{t1}}$
    val $\text{new}_{\text{unit} \rightarrow \text{t1}} : \text{unit} \rightarrow \text{stack}$
end

Bound names are collected at the signature identifier. They are listed between parenthesis to indicate that they are merely place holders.

The bound names of $\text{StackSig}$ are $\text{m1}$, $\text{s1}$ and $\rightarrow$. The free names of $\text{StackSig}$ are $\text{unit}$ and $\rightarrow$. The bound names of $\text{TranspSig}$ are $\text{m1}$ and $\text{s1}$. The free names of $\text{TranspSig}$ are $\text{t1}$, $\text{unit}$ and $\rightarrow$.

Exercise 16  Consider

signature $\text{Symbol}$ =
    sig
    type $\text{symbol}$
    type $\text{value}$
    sharing type $\text{value} = \text{int}$
end

Decorate this signature declaration with type and structure names. How many bound names are there? How many free?

If two structures are found to share by the static analysis then they really are the same at run-time. Therefore, when decorating signatures one must make sure that if two structures are made to share (by being given the same name) then any type or structure which is visible in both structures must be made to share as well.

Exercise 17  Consider the following signatures most of which you have already seen in Lecture 1.

signature $\text{ValSig}$ =
    sig
    type $\text{value}$
end

signature $\text{SymSig}$ =
    sig
    $\text{eqtype}$ $\text{sym}$
    val $\text{hash} : \text{sym} \rightarrow \text{int}$
end

signature $\text{LexSig}$ =
    sig
    structure $\text{Sym} : \text{SymSig}$
    val $\text{getsym} : \text{unit} \rightarrow \text{Sym} \cdot \text{sym}$
end

signature $\text{SymTblSig}$ =
    sig
    structure $\text{Val} : \text{ValSig}$
    structure $\text{Sym} : \text{SymSig}$
    type $\text{table}$
    val $\text{lookup}$:
        $\text{table} \cdot \text{Sym} \cdot \text{sym} \rightarrow \text{Val} \cdot \text{value}$
end

signature $\text{ParseSig}$ =
    sig
    structure $\text{Lex} : \text{LexSig}$
    structure $\text{Tbl} : \text{SymTblSig}$
    sharing $\text{Lex} \cdot \text{Sym} = \text{Tbl} \cdot \text{Sym}$
type abssyn  
val parse : unit -> abssyn  
end

Decorate these signatures. When one signature refers to another (for instance LexSig refers to SymSig) you should put a full decoration on the structure identifier (Sym), i.e. a decoration which shows both a name and the subcomponents of the structure. Full decorations can be drawn as trees; in the example at hand you can decorate Sym by

\[
\begin{array}{c}
\text{sym} \\
\downarrow \text{m1} \\
\downarrow \text{hash} \\
\text{s1} \quad \text{s1} \rightarrow \text{int}
\end{array}
\]

Make sure that you decorate shared substructures (for instance Sym in ParseSig) consistently so as to represent that sharing of two structures implies sharing of their substructures.

3.5 Signature Instantiation

structure Stack_{n1} =
  struct
  type elt_{int} = int
  datatype stack_{elt} = ST of elt list ref
  fun new_{unit->elt}() = ST(ref [ ])  
end

signature StackSigA_{(m1,s1,s2)} =
  sig_{m1}
  type elt_{s1}
  datatype stack_{elt} = ST of elt list ref
  val new_{unit->elt} : unit -> stack
end

Note that if we substitute n1 for m1, int for s1 and t1 for s2 in the decoration of StackSigA then we get the decoration of Stack. We say that Stack is an instance of StackSig. More generally, we say that a structure is an instance of a signature if the decoration of the former is obtained from the decoration of the latter by performing a substitution of names for the bound names of the signature (the free names of the signature must be left unchanged). The process of substituting names for bound names is called REALISATION.
structure \(\text{Stack}_{n1} =\)
struct
  type \(\text{elt}_{int} = \text{int}\)
datatype \(\text{stack}_{t1} = \text{ST of elt list ref}\)
fun \(\text{new}_{\text{unit} \rightarrow \text{t1}}O = \text{ST}(\text{ref}[])\)
end

signature \(\text{StackSigB}_{(m1,s1)} =\)
sig
  type \(\text{elt}_{s1}\)
datatype \(\text{stack}_{t1} = \text{ST of elt list ref}\)
sharing type \(\text{stack}_{t1} = \text{Stack} \cdot \text{stack}_{t1}\)
val \(\text{new}_{\text{unit} \rightarrow \text{t1}} : \text{unit} \rightarrow \text{stack}\)
end

\(\text{Stack}\) is an instance of \(\text{StackSigB}\) via the realisation \(\{m1 \mapsto n1, s1 \mapsto \text{int}\}\).

structure \(\text{OddStr}_{n1} =\)
struct
  type \(\text{elt}_{int} = \text{int}\)
val \(\text{test}_{\text{bool}} = \text{false}\)
end

signature \(\text{WrongSig}_{(m1,s1)} =\)
sig
  type \(\text{elt}_{s1}\)
val \(\text{test}_{s1} : \text{elt}\)
end

\(\text{OddStr}\) is not an instance of \(\text{WrongSig}\), for \(s1\) would have to be realised by \(\text{int}\) (because of \(\text{elt}\)) but then \(\text{test}\) is decorated by \(\text{int}\) in the signature and by \(\text{bool}\) in the structure.

3.6 Signature Matching

Matching of a structure against a signature is a combination of two operations. The first, signature instantiation (described above), is concerned with instantiating the bound names of the signature to the “real” names of the structure. The second is concerned with ignoring information in the structure which is not required by the signature.

structure \(\text{Tree}_{n1} =\)
struct
  datatype \(\text{a tree}_{t1} = \text{LEAF of } \text{a}\)
    \| \text{NODE of } \text{a tree} * \text{a tree}\n  type \(\text{intTree}_{\text{int}t1} = \text{int tree}\)
fun \(\text{max}(a : \text{int}, b : \text{int}) =\)
  if \(a > b\) then \(a\) else \(b\)
fun \(\text{depth}_{\text{a tree} \rightarrow \text{int}}(\text{LEAF } \_ ) = 1\)
  \| \(\text{depth}(\text{NODE}(\text{left}, \text{right})) = \\text{max}(\text{depth \ left}, \text{depth \ right})\)
end

signature \(\text{TreeSig}_{(m1,s1,s2)} =\)
sig
  type \(\text{a tree}_{s1}\)
type \(\text{intTree}_{s2}\)
fun \(\text{depth}_{\text{int} \rightarrow \text{int}} : \text{int tree} \rightarrow \text{int}\)
end

A structure MATCHES a signature if the structure can be cut down to an instance of the signature by

1. forgetting components;
2. forgetting polymorphism of variables.

\(\text{Tree}\) matches \(\text{TreeSig}\). First perform the realisation \(\{m1 \mapsto n1, s1 \mapsto t1, s2 \mapsto \text{int} t1\}\) on the signature. The resulting decoration can be obtained from the decoration of \(\text{Tree}\) by

1. forgetting the constructors \(\text{LEAF}\) and \(\text{NODE}\)
2. instantiating \(\text{a} \ t1 \rightarrow \text{int}\) to \(\text{int} t1 \rightarrow \text{int}\)
   (i.e. the realisation of \(s2 \mapsto \text{int}\)).
Exercise 18  Let mytype be a type which is declared in a structure and specified in a signature. In which of the following cases can the structure match the signature?

1. mytype is declared as a datatype and specified as a datatype.
2. mytype is declared as a datatype and specified as a type.
3. mytype is declared as a type and specified as a type.
4. mytype is declared as a type and specified as a datatype.

3.7 Signature Constraints

signature SymSig =
  sig
    type sym
    type code
    sharing type code = int
  val hash : sym->int
  val mksym : string->sym
  val nameof : sym->string
end

structure Sym : SymSig =
  struct
    datatype sym = SYM of string * int
    type code = int
    fun convert(s: string): code = ...
    fun hash(SYM(s, n)) = n
    fun mksym(s) = SYM(s, convert s)
    fun nameof(SYM(s, _)) = s
  end

Exercise 19  Complete the declaration of convert.

Exercise 20  Declare a different structure NewSym, also constrained by SymSig, such that NewSym.sym shares with string. Which of the following expressions are valid?

1. "a" ^ NewSym.mksym "d"
2. "a" ^ Sym.mksym "d"
3.8 Decorating Functors

Dynamically, the body of a functor is not evaluated when the functor is declared but it is evaluated once for each time the functor is applied.

functor StackFct() =
  struct
    datatype stack = ST of int list ref
    val data = ST(ref[ ])
  ...
  end

structure Stack1 = StackFct()
structure Stack2 = StackFct()

Since the two applications of StackFct create two distinct references, Stack1 and Stack2 are different and must not be seen to share.

Now let us consider the problem of decorating StackFct with names. We start out by decorating the body in the usual way. However, each time we need a fresh name, we record it at the = in the first line. The names that hence are accumulated are called the GENERATIVE NAMES of the functor. The generative names are bound in the sense that they stand as place holders for fresh names which we choose when we eventually apply the functor. (Like the bound names in signatures, we write generative names between parenthesis; unlike the bound names of signatures, generative names are written on the right — because they concern the right side only.)

In the case of nullary functors, i.e. functors that take an empty argument, the structure resulting from a functor application is decorated by taking the decoration of the functor body with each generative name replaced by a fresh name.

functor StackFct() =
  struct
    datatype stack = ST of int list ref
    val data = ST(ref[ ])
  ...
  end

structure Stack1 = StackFct()
structure Stack2 = StackFct()

Exercise 21 The decorations of Stack1 and Stack2 show the top-most structure name only. Complete the decorations.

Notice that Stack1 and Stack2 do not share; not even the types Stack1.stack and Stack2.stack share. Consequently, the variables Stack1.data and Stack2.data have different types and so the type checker prevents one from mistaking the one for the other.

3.9 External Sharing

Within the body of a functor one may refer to identifiers (of any kind) declared in the context of the functor. Such identifiers are said to occur FREE in the functor. This results in EXTERNAL SHARING, i.e. a decoration in which some of the names stem from outside the functor.
3.10 Functors with Arguments

signature $SymSig_{(m1,s1)} = \\[\]
sig_{n1}
  eqtype sym_{s1}
end

functor $SymDir(Sym: SymSig) =_{(m2,s2)}$
  struct
    datatype dir_{s2} = DIR of
      Sym, sym -> int
  end
fun update ...
end

When decorating the body of a functor which has an argument, we assume that we have a structure (by the name of the formal parameter) which \emph{precisely} matches the parameter signature. We assume neither more components nor more sharing than is specified in the signature, for we want the functor to be applicable to all actual argument structures that match the formal parameter signature.

In the above example we simply assume that the name of $Sym$ is $m1$ and that the name of $Sym.sym$ is $s1$ (as those names are not used of free structures elsewhere; in general, one might have to rename some of the bound names. Having used $m1$ and $s1$ we simply start generating names from $m2$ and $s2$ in the body.

structure $Actual_{n1} =$
  struct type sym_{string} = string
end

structure $Result_{n2} = SymDir(Actual)$

Notice that external names are not generative; they are left unchanged when the functor is applied.

Exercise 22 Which of the following sharing equations hold?
$Stack1' = Stack2'$;
$Stack1'.MyPer = Stack2'.MyPer$;
type $Stack1'.stack = Stack2'.stack$. 

structure $MyPervasives =$
  struct
    datatype num_{t1} = NUM of int
    ...
  end

functor $StackFcl'() =_{(m2)}$
  struct
    structure $MyPer_{n1} = MyPervasives$
    type $stack_{t1 list ref} =$
      $MyPer . num list ref$
    val $data_{t1 list ref}: stack = ref [ ]$
  end

structure $Stack1_{n8} = StackFcl'()$

structure $Stack2_{n9} = StackFcl'()$
Result receives a fresh structure name and Result, dir a fresh type name. Note that Actual matches SymSig.

### 3.11 Sharing Between Argument and Result

```
signature SymSig(s1) =
  sig
  eqtype sym, s1
end

functor SymDir(Sym: SymSig) =
  struct
    type dir, int = Sym.sym->int
    fun update ...
  end
```

The type name s1 is shared between the argument and the body. When the functor is applied, this sharing must be translated into sharing between the actual argument and the actual result.

```
structure Actual =
  struct
    type sym = string
  end

structure Result = SymDir(Actual)
```

### 3.12 Explicit Result Signatures

When a functor declaration contains a result signature, the decoration of the functor declaration proceeds as follows:

1. decorate the functor without the result signature;

2. decorate the result signature. If one can get an instance of the result signature by removing components and polymorphism from the decorated body, then this instance is used as a formal result instead of the decorated body; otherwise the declaration is rejected.

This has the effect that upon application of the functor, sharing is propagated as before, but only the components and polymorphism of the result signature are visible in the actual result.

**Exercise 23** Complete the decoration of Result.

**Exercise 24** (1) Using the latest definition of SymDir, is the following expression legal?

```
fn (d: Result, dir) => d "abc"
```

**Exercise 25** Why is it not always the case that obtaining R by
functor $F(S : SIG) : SIG'$
struct...end

structure $R = F(...)$

is equivalent to obtaining $R$ by

functor $F(S : SIG) =$
struct...end

structure $R : SIG' = F(...)$

Give a condition on $SIG'$ under which this
difference disappears.
4 Implementing an Interpreter in ML

The purpose of this lecture is to show a worked example of program development using ML modules. We shall tackle the problem of implementing a small ML system. The system is of course going to considerable simplified compared to a real ML implementation.

We implement only a few of the language constructs found in real ML. The user of our system will not get the ability to declare new types and data types; however, there will be arithmetic on the build-in integers, if...then...else expressions, and indeed lists, higher order functions and recursion, so it is far from a trivial language. We shall refer to this language as Mini ML.

Moreover, the system will be an interpreter rather than a compiler. It still has a type-checker, indeed we shall see how one can implement a restricted form of polymorphism.

The system is actually running and you can modify and extend it provided you have access to an implementation and to the files listed in Appendix B. To make life easier for you, we provide a parse functor which can parse a string (the Mini ML source expression) into an ABSTRACT SYNTAX TREE, the shape of which will be defined below. The rest of the interpreter works on abstract syntax trees.

The interpreter uses a TYPECHECKER to check the validity of input expressions and an EVALUATOR to evaluate them. Initially, the typechecker and evaluator handle only a tiny subset of Mini ML. In this lecture I shall show how one in successive steps can extend the typechecker to handle polymorphic lists, variables and let expressions. In the practical sessions you can extend the evaluator in the same manner (it is easier than extending the typechecker).

The typechecker and the evaluator can be developed independently as long as you do not change the signatures we provide. This will allow you to take the typechecker functors I have written and plug into your own system as you improve the power of your evaluator. Alternatively, you might want to modify or extend my typechecker functors, and take over evaluator functors that other people write.

The source of the bare interpreter is in Appendix A. An overview of how to run the systems is provided in Appendix B.

The development of the typechecker and the evaluator need not be in step. You can disable either by assigning false to one of the variables tc and eval.
signature INTERPRETER =
   sig
     val interpret : string -> string
     val eval : bool ref
     and tc : bool ref
   end;

The syntax of the language is as follows

\[
\begin{align*}
exp &::= exp + exp \\
&\quad \quad exp * exp \\
&\quad \quad true \\
&\quad \quad false \\
&\quad \quad exp = exp \\
&\quad \quad if exp then exp else exp \\
&\quad \quad exp :: exp \\
&\quad \quad [exp_1, \ldots, exp_n] \ (n \geq 0) \\
&\quad \quad let x = exp in exp \\
&\quad \quad let rec x = exp in exp \\
&\quad \quad x \\
&\quad \quad fn x => exp \\
&\quad \quad exp (exp) \quad \text{(function application)} \\
&\quad \quad n \quad \text{(natural numbers)} \\
&\quad \quad (exp) \\
\end{align*}
\]

The abstract syntax of Mini ML is defined as a datatype in the signature `EXPRESSION`.

**Exercise 1** Find this signature. What is the constructor corresponding to `let` expressions?

We program with signatures and functors only. After the signatures, which we shall not yet study, the first functor is the interpreter itself.

**Exercise 2** Find this functor. Find the application of `Ty.prType`. Find it’s type. What do you think `Ty.prType` is supposed to do? What is the type of `absSyn`? What do you think the evaluator is supposed to do when asked to evaluate something which has not yet been implemented?

We shall now describe Version 1, the bare typechecker, and then proceed to the extensions.
4.1 VERSION 1: The bare Typechecker (Appendix A)

The first version is just able to type check integer constants and +. As signature TYPE reveals, the type Type of types is abstract, but there are functions we can use to build basic types and decompose them. unTypeInt is one of the latter; it is supposed to raise Type if applied to any Mini ML type different from the int (however the type int is represented). This is a common way of hiding implementation details, and it might be helpful to look at how functor Type produces a structure which matches the signature Type.

As revealed by signature TYPECHECKER, the typechecker is going to depend on the abstract syntax and a Type structure. However, as you can see from the declaration of functor TypeChecker, all the typechecker knows about the implementation of types is what is specified by the signature TYPE. This allows us to experiment with the implementation of types to obtain greater efficiency without changing the typechecker, as we shall see in the later stages. As you see from functor TypeChecker, all the typechecker is capable of handling is integer constants and +.

Exercise 3  Modify the typechecker to handle true, false, and multiplication of integers.

Given the signature and functor declarations in Appendix A, one can build the system. First we import the parser

```
use "parser.sml";
```

and then we build the system by the following declarations (which can be read from file build1.sml).

```sml
structure Expression= Expression();

structure Parser= Parser(Expression);

structure Value = Value();

structure Evaluator=
    Evaluator(structure Expression= Expression
        structure Value = Value);

structure Ty = Type();

structure TyCh=
    TypeChecker(structure Ex = Expression
        structure Ty = Ty);

structure Interpreter=
    Interpreter(structure Ty= Ty
```
structure Value = Value
structure Parser = Parser
structure TyCh = TyCh
structure Evaluator = Evaluator);

open Interpreter;

4.2 VERSION 2: Adding lists and polymorphism

The first extension is to implement the type checking of lists. In Version 1 the type of an expression could be inferred either directly (as in the case of \texttt{true} and \texttt{false}, or from the type of the subexpressions (as in the case of the arithmetic operations). When we introduce list, this is no longer the case. Consider for example the expression

\[
\text{if } ([ ] = [9]) \text{ then 5 else 7}
\]

Suppose we want to type check \([ ] = [9]\) by first type checking the left subexpression \([ ]\), then the right subexpression \([9]\) and finally checking that the left and right-hand sides are of the same type before returning the type \texttt{bool}. The problem now is that when we try to type check \([ ]\) we cannot know that this empty list is supposed to be an integer list. The typechecker therefore just ascribes the type \('a\ list\) to \([ ]\), where \('a\) is a \texttt{TYPE VARIABLE}. The \([9]\) of course turns out to be an \texttt{int list}. The typechecker now “compares” the two types \('a\ list\) and \texttt{int list} and discovers that they can be made the same by applying the substitution that maps \('a\) to \texttt{int}. Hence the type of the expression \([ ]\) depends not just on the expression itself, but also on the context of the expression. The context can force the type inferred for the expression to become more specific.

This “comparison” of types performed by the typechecker is called \texttt{UNIFICATION} and is an algebraic operation of great importance in symbolic computing. Indeed, whole programming languages have evolved around the idea of unification (PROLOG, for example). Here is a couple of examples to illustrate how unifications works in the special case of interest, that of unifying types.

\[
[ [ ] , [[5]] ]
\]

This expression is well-typed! The point is that the \([ ]\) can be regarded as an \texttt{int list list}. Let us see how the typechecker manages to infer the type \texttt{int list list list} for (1). The typechecker first rewrites the expression to the equivalent:

\[
[ ] ::= (((5 :: [ ]) :: [ ]) :: [ ])
\]

Checking the first argument of the topmost :: yields:

\[
[ ] :: \ 'a\ list
\]
To check \(((5 : {} : {}) : {} : {}))\), we first check the left-hand \((5 : {} : {})\). To check this, we first check the left-hand \((5 : {})\). To check this, we first check the left-hand 5, for which the typechecker wisely infer the type `int`. Continuing to the right-hand part of \((5 : {})\), {} gets the type `a2 list`. To check the {} of \((5 : {} : {}))\), we now unify `int list` and `a2 list`, which results in the substitution

\[ S_1(a2) = \text{int}. \]

Thus the type of \((5 : {})\) is `int list`.

Returning to \(((5 : {}) : {} : {}))\), the right-hand {} first gets type `a3 list` which by unification with `int list` yields the substitution

\[ S_2(a3) = \text{int list}. \]

Thus the type of \(((5 : {}) : {} : {}))\) is `int list list`.

Returning to \(((5 : {}) : {} : {})) : {}\), the right-hand {} gets the type `a4 list` which by unification with `int list list` yields the substitution

\[ S_3(a4) = \text{int list list}. \]

Thus the type of \(((5 : {}) : {} : {})) : {}\) is `int list list list`.

Finally, returning to (2) and (3), we get to unify `a1 list` with `int list list list`, yielding the substitution

\[ S_4(a1) = \text{int list list}. \]

The type of (2), and therefore the type of (1), is thus found to be `int list list list`.

Note that

\[ [ [4], [[5]] ] \]

is NOT well-typed. In an attempt to compute \(S_4\), we would now be unifying `int list list` and `int list list list` and that gives a unification error.

To implement all this, we first extend the `TYPE` signature and introduce a new signature, `UNIFY`:

```haskell
signature TYPE =
  sig
    eqtype tyvar
    val freshTyvar: unit => tyvar
    ...
    val mkTypeTyvar: tyvar => Type
        and unTypeTyvar: Type => tyvar
        ...
    val mkTypeList: Type => Type
        and unTypeList: Type => Type
        ...
```
type subst
val Id: subst
  (* the identify substitution; *)
val mkSubst: tyvar*Type => subst
  (* make singleton substitution; *)
val on : subst * Type => Type
  (* application; *)

val prType: Type=>string (* printing *)
end

signature UNIFY=
sig
  structure Type: TYPE
  exception NotImplemented of string
  exception Unify
  val unify: Type.Type * Type.Type => Type.subst
end;

The nice thing is that we can extend the typechecker without knowing anything about
the inner workings of unification, simply by including a formal parameter of signature
UNIFY in the typechecker functor:

functor TypeChecker
  (...;
    structure Ty: TYPE
    structure Unify: UNIFY
      sharing Unify.Type = Ty
  )=
struct
  infix on
  val (op on) = Ty.on
...

fun tc (exp: Ex.Expression): Ty.Type =
  (case exp of
    ...
    | Ex.LISTexpr [] =>
      let val new = Ty.freshTyvar ()
      in Ty.mkTypeList(Ty.mkTypeTyvar new)
      end
    | Ex.CONSexpr(e1,e2) =>
      ...
let val t1 = tc e1
val t2 = tc e2
val new = Ty.freshTyvar ()
val newt= Ty.mkTypeTyvar new
val t2’ = Ty.mkTypeList newt
val S1 = Unify.unify(t2, t2’)
  handle Unify.Unify=>
  raise TypeError(e2,"expected list type")

val S2 = Unify.unify(S1 on newt,S1 on t1)
  handle Unify.Unify=>
  raise TypeError(exp,
  "element and list have different types")
in S2 on (S1 on t2)
end
| ...

)handle Unify.NotImplemented msg => raise NotImplemented msg
end; (*TypeChecker*)

We also have to extend the Type functor to meet the enriched TYPE signature. The easiest way of doing this is

functor Type():TYPE =
  struct
  type tyvar = int
  val freshTyvar =
    let val r= ref 0 in fn()=>(r:= !r +1; !r) end
  datatype Type = INT
    | BOOL
    | LIST of Type
    | TYVAR of tyvar
...

fun mkTypeTyvar tv = TYVAR tv
and unTypeTyvar(TYVAR tv) = tv
  | unTypeTyvar _ = raise Type

fun mkTypeList(t)=LIST t
and unTypeList(LIST t)= t
  | unTypeList(_)= raise Type
type subst = Type \to Type

fun Id x = x

fun mkSubst(tv,ty)=
  let fun su(TYVAR tv')= if tv=tv' then ty else TYVAR tv'
     | su(INT) = INT
     | su(BOOL)= BOOL
     | su(LIST ty') = LIST (su ty')
  in su
  end

fun on(S,t)= S(t)

fun prType ...
  | prType (LIST ty) = "(" ^ prType ty ^ ")list"
  | prType (TYVAR tv) = "a" ^ makestring tv
end;

Exercise 4 Extend Version 2 to handle equality. All you have to do is to fill in the relevant case in the definition of the function tc. (See appendix B about how you get the source of Version 2).

4.3 VERSION 3: A different implementation of types

Version 3 arises from Version 2 by replacing the Type functor by a different implementation of types. The idea is that instead of having substitutions as functions, we can implement type variables by references (pointers) and then do substitutions directly by assignments.

In case you have not seen the reserved word withtype before, withtype is used to declare a type abbreviation locally within a datatype declaration.

functor ImpType():TYPE =
struct
  datatype 'a option = NONE | SOME of 'a

datatype Type = INT
    | BOOL
    | LIST of Type
    | TYVAR of tyvar

  withtype tyvar = Type option ref
type tyvar = Type option ref

fun freshTyvar() = ref (NONE)

exception Type

fun mkTypeInt() = INT
and unTypeInt(INT)=()

| ...
| unTypeInt(TYVAR(ref (SOME t)))= unTypeInt t
| unTypeInt _ = raise Type

...

type subst = unit

val Id = ();

exception MkSubst;

fun mkSubst(tv,ty)=
  case tv of
    ref(NONE) => tv:= (SOME ty)
  | ref(SOME t) => raise MkSubst

fun on(S,t)= t

fun prType ...
  | prType (TYVAR (ref NONE)) = "a?"
  | prType (TYVAR (ref (SOME t))) = prType t

end;

We can now build two systems at the same time and compare the efficiency of the two implementations. The nice thing is that we do not have to modify the typechecker functor at all, nor do we even have to modify the unification functor; we can just extend the final sequence of structure declarations to use both implementations of types.

Exercise 5 When I did this, I found (to my surprise), that the functional version in some cases was twice as fast, and never slower than the imperative variant. The relative performance of the two vary greatly from expression to expression. Can you find an expression for which the imperative version really is faster? (See Appendix B for how to get hold of the source of Version 3). Be careful with generating very demanding tasks for the ML system; you can make it crash!
ML implementors normally opt for the imperative version. In all fairness, the above comparison ignores that composing substitutions is much easier in the imperative version than it is in the applicative version; in the fragment of Mini ML considered so far, we have not had to compose substitutions.

One should not be too concerned with performance issues at too early a stage. It can be surprisingly difficult to predict where efficiency is most needed, and it is much more important, at first, to get the overall structure of the system right. It was important, for example, that we did NOT make the constructors of the datatype Type visible in the signature TYPE, and that we wrote the unification algorithm in a way which does not use the internal structure of Type. Had we not done this, we would not have been able to switch from one implementation to another that easily, and therefore chances are that we would have chosen the imperative one, assuming that it was the more efficient one, without ever trying the “obvious” applicative implementation.

4.4 VERSION 4: Introducing variables and let

We now extend Version 3 by implementing the type checking of let expressions and of identifiers.

The type checker function tc now has to take TWO arguments,

\[ \text{tc(TE, e)} \]

where e is an expression and TE is a TYPE ENVIRONMENT, which maps variables occurring free in e to TYPE SHMASES. The definition of what a type scheme is will be given below; for now it suffices to know that every type can be regarded as a type scheme.

To take an example, if TE maps x to int and y to int, then tc will deduce the type int for the expression x+y. (However, if TE mapped y to bool, there would be a type error.)

The fact that we can bind variables to expressions whose types have been inferred to contain type variables means that we get type variables in the type environment. For instance, to type check

\[ \text{let } x = [] \text{ in } 4 :: x \text{ end} \]

we first check [] yielding the type 'a1 list, say. Then we bind x to the type scheme \( \forall \text{'a1 } \text{ list} \). Here the binding \( \forall \text{'a1} \) of 'a1 indicates that when we look up the the type of x in the type environment, we return a type obtained from the type scheme \( \forall \text{'a1} \text{'a1 list} \) by instantiating the bound variables (here just 'a1) by fresh type variables. In our example, when we look up x in the type environment during the checking of 4 :: x, we instantiate 'a1 to a fresh type variable 'a2, say, yielding the type 'a2 list for x. Thus we get to unify int list against 'a2 list, yielding the substitution of int for 'a2.

Throughout the body of the let, x will be bound to \( \forall \text{'a1 } \text{ list} \) in the type environment. Since we take a fresh instance of this type scheme each time we look up x, we can use x both as an int list and as an int list list, say:
let x = □ in (4::x)::x end

**Exercise 6** Assuming that you instantiate the bound 'a1 to 'a3 when you meet the last occurrence of x, what two types should be unified, and what is the resulting substitution on 'a3 ?

The variable x is an example of **polymorphism**: after x has been declared, an occurrence of x can potentially be given infinitely many types: int list, bool list, int list list, and so on, all captured by the type scheme \( \forall 'a1. 'a1 list \). In ML, a **type scheme** always takes the form \( \forall \alpha_1 \cdots \alpha_n. \tau \), \( (n \geq 0) \), where \( \alpha_1, \ldots, \alpha_n \) are type variables and \( \tau \) is a type. In the fragment of Mini ML considered so far, it will always be the case that any type variable occurring in \( \tau \) is amongst the \( \alpha_1, \ldots, \alpha_n \), but when one introduces functions and application, this no longer is the case.

Here is how we implement variables and let. We first extend the **type signature**:

```ml
signature TYPE =
  sig
    ...
    type TypeScheme
    val instance: TypeScheme -> Type
    val close: Type -> TypeScheme
  end
```

Version 1 (Appendix A) already contains a signature for environments (find it). It was actually intended for the practical where you need it to extend the evaluator, but we can make use of it to implement type environments. The signature of the typechecker can be left unchanged, but we need to change the functor that builds the typechecker by including the environment management among the formal parameters:

```ml
functor TypeChecker
  (structure Ex: EXPRESSION
   structure Ty: TYPE
   structure Unify: UNIFY
     sharing Unify.Type = Ty
   structure TE: ENVIRONMENT
   )=
  struct
    infix on
    val (op on) = Ty.on
    structure Exp = Ex
    structure Type = Ty
  end
```
exception Not Implemented of string
exception TypeError of Ex.Expression * string

fun tc (TE: Ty.TypeScheme TE.Environment, exp: Ex.Expression): Ty.Type =
(case exp of
  Ex.BOOLexpr b => Ty.mkTypeBool()
| Ex.NUMBERexpr _ => Ty.mkTypeInt()
| Ex.SUMexpr(e1,e2) => checkIntBin(TE,e1,e2)
| Ex.DIFFexpr(e1,e2) => checkIntBin(TE,e1,e2)
| Ex.PRODexpr(e1,e2) => checkIntBin(TE,e1,e2)
| Ex.LISTexpr [] =>
  let val new = Ty.freshTyvar ()
  in Ty.mkTypeList(TE.mkTypeTyvar new)
end
| Ex.LISTexpr(e::es) => tc (TE, Ex.CONSexpr(e,Ex.LISTexpr es))
| Ex.CONSexpr(e1,e2) =>
  let val t1 = tc(TE, e1)
  val t2 = tc(TE, e2)
  val new = Ty.freshTyvar ()
  val newt= Ty.mkTypeTyvar new
  val t2' = Ty.mkTypeList newt
  val S1 = Unify.unify(t2, t2')
    handle Unify.Unify=>
      raise TypeError(e2,"expected list type")
  val S2 = Unify.unify(S1 on newt,S1 on t1)
    handle Unify.Unify=>
      raise TypeError(exp,"element and list have different types")
  in S2 on (S1 on t2)
end
| Ex.EQexpr _ => raise Not Implemented "(equality)"
| Ex.CONDexpr _ => raise Not Implemented "(conditional)"
| Ex.DECLexpr(x,e1,e2) =>
  let val t1 = tc(TE,e1);
  val typeScheme = Ty.close(t1)
  in tc(TE.declare(x,typeScheme,TE), e2)
end
| Ex.RECDECLexpr _ => raise Not Implemented "(rec decl)"
| Ex.IDENTexpr x =>
  (Ty.instance(TE.retrieve(x,TE))
    handle TE.Retrieve _ =>
      raise TypeError(exp,"identifier ^x^ not declared")
  )
| Ex.LAMBDAXexpr _ => raise Not Implemented "(function)"

40
| Ex.APPLexpr _ => raise Not Implemented   "(application)"

)handle Unify.NotImplemented msg => raise Not Implemented msg

and checkIntBin(TE,e1,e2) =
  let val t1 = tc(TE,e1)
    val _ = Ty.unTypeInt t1
      handle Ty.Type => raise TypeError(e1,"expected int")
  val t2 = tc(TE,e2)
  val _ = Ty.unTypeInt t2
    handle Ty.Type => raise TypeError(e2,"expected int")
  in Ty.mkTypeInt()
  end;

fun typecheck(e) = tc(TE.emptyEnv,e)

end; (*TypeChecker*)

---

Then we extend the Type functor to match the TYPE signature:

functor Type() : TYPE =
  struct
    ...
    datatype TypeScheme = FORALL of tyvar list * Type
  
  fun instance(FORALL(tyvars,ty)) =
    let val old_to_new_tyvars = map (fn tv=>(tv,freshTyvar())) tyvars
      exception Find;
      fun find(tv,[])= raise Find
        | find(tv,(tv',new_tv)::rest)=
            if tv=tv' then new_tv else find(tv,rest)
      fun ty_instance INT = INT
        | ty_instance BOOL = BOOL
        | ty_instance (LIST t) = LIST(ty_instance t)
        | ty_instance (TYVAR tv) =
            TYVAR(find(tv,old_to_new_tyvars)
              handle Find=> tv)

    in
      ty_instance ty
    end

41
fun close(ty)=
let fun fv(INT) = []
|   fv(BOOL)= []
|   fv(LIST t) = fv(t)
|   fv(TYVAR tv) = [tv]
in FORALL(fv ty,ty)
end

Finally, the system is re-built as in Version 2, except that we have to provide and link
in an Environment functor which matches ENVIRONMENT.

Exercise 7  Extend Version 4 with if .. then .. else. (This extension has no
subtle implications for the type checking.)

Exercise 8  [For the extra keen] Extend Version 4 to cope with lambda abstraction
(fn) and application. First, you have to introduce arrow types with constructors and
destructors. Then you have to change the type of close so that it takes two arguments,
namely a type environment and a type. It should return the type scheme that is obtained
by quantifying on all the type variables that occur in the type but do not occur free in
the type environment.

Then you can modify the type checker. When you type check a lambda abstraction,
you just bind the formal parameter to the trivial type scheme which is just a fresh type
variable (no quantified variables). Thus the type environment can now contain type
schemes with free type variables.

An application tc(TE,e) now yields two arguments, namely a type t and a substitu-
tion S; the idea is that if you apply the substitution S to the type environment TE, which
now can contain free type variables, the expression e has the type t. When an expression
consists of more than one subexpression, the type environment gradually becomes more
and more specific by applying the substitutions produced by the checking of the subex-
pressions one by one. Moreover, the substitution returned from the whole expression is
the composition of these individual substitutions. (You have to extend the TYPE signature
(and the Type functor) with composition of substitutions.

Finally, you can extend the unification algorithm to cope with arrow types. (This will
also use composition of substitutions.)

4.5  Acknowledgement

The parser and evaluator and all the signatures related to them are due to Nick Rothwell.
Appendix A: The bare Interpreter

(* interp1.sml : VERSION 1: the bare interpreter *)

signature INTERPRETER =
 sig
   val interpret: string -> string
   val eval: bool ref
   and tc : bool ref
 end;

(* syntax *)

signature EXPRESSION =
 sig
   datatype Expression =
      SUMexpr of Expression * Expression |
      DIFFexpr of Expression * Expression |
      PRODexpr of Expression * Expression |
      BOOLexpr of bool |
      EQexpr of Expression * Expression |
      CONDexpr of Expression * Expression * Expression |
      CONSexpr of Expression * Expression |
      LISTexpr of Expression list |
      DECLexpr of string * Expression * Expression |
      RECDECLexpr of string * Expression * Expression |
      IDENTexpr of string |
      LAMBDAXexpr of string * Expression |
      APPLexpr of Expression * Expression |
      NUMBERexpr of int
 end

(* parsing *)

signature PARSER =
 sig
   structure E: EXPRESSION

exception Lexical of string
exception Syntax of string
val parse: string -> E.Expression
end

(* environments *)

signature ENVIRONMENT =
  sig
  type 'object Environment

  exception Retrieve of string

  val emptyEnv: 'object Environment
  val declare: string * 'object * 'object Environment
    -> 'object Environment
  val retrieve: string * 'object Environment -> 'object
end

(* evaluation *)

signature VALUE =
  sig
    type Value
    exception Value

    val mkValueNumber: int -> Value
        and unValueNumber: Value -> int

    val mkValueBool: bool -> Value
        and unValueBool: Value -> bool

    val ValueNil: Value
    val mkValueCons: Value * Value -> Value
        and unValueHead: Value -> Value
        and unValueTail: Value -> Value

    val eqValue: Value * Value -> bool
    val printValue: Value -> string
end

signature EVALUATOR =
  sig
structure Exp: EXPRESSION
structure Val: VALUE
exception Unimplemented
val evaluate: Exp.Expression -> Val.Value
end

(* type checking *)
signature TYPE =
  sig
    type Type
  end

(* constructors and destructors *)
exception Type
val mkTypeInt: unit -> Type
  and unTypeInt: Type -> unit

val mkTypeBool: unit -> Type
  and unTypeBool: Type -> unit

val prType: Type -> string
end

signature TYPECHECKER =
  sig
    structure Exp: EXPRESSION
    structure Type: TYPE
    exception NotImplemented of string
    exception TypeError of Exp.Expression * string
    val typecheck: Exp.Expression -> Type.Type
  end;

(* the interpreter *)

functor Interpreter
  (structure Ty: TYPE
   structure Value: VALUE
   structure Parser: PARSER
   structure TyCh: TYPECHECKER
   structure Evaluator: EVALUATOR
     sharing Parser.E = TyCh.Exp = Evaluator.Exp
     and TyCh.Type = Ty
and Evaluator.Val = Value
)

) : INTERPRETER=

struct
val eval = ref true (* toggle for evaluation *)
and tc = ref true (* toggle for type checking *)
fun interpret(str)=
  let val abstsyn = Parser.parse str
  val typestr = if !tc then
    Ty.prType(TyCh.typecheck abstsyn)
  else "(disabled)"
  val valuestr = if !eval then
    Value.printValue(Evaluator.evaluate abstsyn)
  else "(disabled)"

  in valuestr ^ " : ": " ^ typestr
end

handle Evaluator.Unimplemented =>
  "Evaluator not fully implemented"
  | TyCh.NotImplemented msg =>
    "Typechecker not fully implemented " ^ msg
  | Value.Value => "Run-time error"
  | Parser.Syntax msg => "Syntax Error: " ^ msg
  | Parser.Lexical msg => "Lexical Error: " ^ msg
  | TyCh.TypeError(_,msg) => "Type Error: " ^ msg
end;

(* the evaluator *)

functor Evaluator
(structure Expression: EXPRESSION
  structure Value: VALUE):EVALUATOR=

struct
  structure Exp = Expression
  structure Val = Value
exception Unimplemented

local
  open Expression Value
  fun evaluate exp =
    case exp
      of BOOLExpr b => mkValueBool b
| NUMBERexpr i => mkValueNumber i  
| SUMexpr(e1, e2) => 
|     let val e1' = evaluate e1  
|           val e2' = evaluate e2  
|     in  
|           mkValueNumber(unValueNumber e1' +  
|         unValueNumber e2')  
| end  
| DIFFexpr(e1, e2) =>  
|     let val e1' = evaluate e1  
|           val e2' = evaluate e2  
|     in  
|           mkValueNumber(unValueNumber e1' -  
|         unValueNumber e2')  
| end  
| PRODexpr(e1, e2) =>  
|     let val e1' = evaluate e1  
|           val e2' = evaluate e2  
|     in  
|           mkValueNumber(unValueNumber e1' *  
|         unValueNumber e2')  
| end  
| EQexpr _ => raise Unimplemented  
| CONDexpr _ => raise Unimplemented  
| CONSexpr _ => raise Unimplemented  
| LISTexpr _ => raise Unimplemented  
| DECLEXPR _ => raise Unimplemented  
| RECDECLexpr _ => raise Unimplemented  
| IDENT expr _ => raise Unimplemented  
| LAMBDAexpr _ => raise Unimplemented  
| APPLexpr _ => raise Unimplemented  

in  
val evaluate = evaluate  
end  
end;

(* the typechecker *)

functor TypeChecker
(structure Ex: EXPRESSION
structure Ty: TYPE)=
struct
structure Exp = Ex
structure Type = Ty
exception NotImplemented of string
exception TypeError of Ex.Expression * string

fun tc (exp: Ex.Expression): Ty.Type =
case exp of
  Ex.BOOLexpr b => raise NotImplemented
                  "(boolean constants)"
  | Ex.NUMBERexpr _ => Ty.mkTypeInt()
  | Ex.SUMexpr(e1,e2) => checkIntBin(e1,e2)
  | Ex.DIFFexpr _ => raise NotImplemented "(minus)"
  | Ex.PRODexpr _ => raise NotImplemented "(product)"
  | Ex.LISTexpr _ => raise NotImplemented "(lists)"
  | Ex.CONSexpr _ => raise NotImplemented "(lists)"
  | Ex.EQexpr _ => raise NotImplemented "(equality)"
  | Ex.CONDexpr _ => raise NotImplemented "(conditional)"
  | Ex.DECLEXP _ => raise NotImplemented "(declaration)"
  | Ex.RECDECLexpr _ => raise NotImplemented "(rec decl)"
  | Ex.IDENTEXP _ => raise NotImplemented "(identifier)"
  | Ex.LAMBDAexpr _ => raise NotImplemented "(function)"
  | Ex.APPLexpr _ => raise NotImplemented "(application)"

and checkIntBin(e1,e2) =
let val t1 = tc e1
  val _ = Ty.unTypeInt t1
    handle Ty.Type=>
    raise TypeError(e1,"expected int")
val t2 = tc e2
val _ = Ty.unTypeInt t2
    handle Ty.Type=>
    raise TypeError(e2,"expected int")
in Ty.mkTypeInt()
end;

val typecheck = tc

end; (*TypeChecker*)
(* the basics -- nullary functors *)

functor Type() : TYPE =
  struct
    datatype Type = INT
      | BOOL
  
  exception Type

  fun mkTypeInt() = INT
  and unTypeInt(INT)=() 
    | unTypeInt(_) = raise Type

  fun mkTypeBool() = BOOL
  and unTypeBool(BOOL)=() 
    | unTypeBool(_) = raise Type

  fun prType INT = "int"
  | prType BOOL = "bool"
end;

functor Expression(): EXPRESSION =
  struct
    type 'a pair = 'a * 'a

    datatype Expression =
      SUMexpr of Expression pair 
      | DIFFexpr of Expression pair 
      | PRODexpr of Expression pair 
      | BOOLexpr of bool 
      | EQexpr of Expression pair 
      | CONDexpr of Expression * Expression * Expression 
      | CONSexpr of Expression pair 
      | LISTexpr of Expression list 
      | DECLexpr of string * Expression * Expression 
      | RECDECLexpr of string * Expression * Expression 
      | IDENTexpr of string 
      | LAMBDAXexpr of string * Expression 
      | APPLEXexpr of Expression * Expression 
      | NUMBERexpr of int

49
end;

functor Value(): VALUE =
  struct
    type 'a pair = 'a * 'a

    datatype Value = NUMBERvalue of int |
                    BOOLvalue of bool |
                    NILvalue |
                    CONSvalue of Value pair
  end

exception Value

val mkValueNumber = NUMBERvalue
val mkValueBool = BOOLvalue
val ValueNil = NILvalue
val mkValueCons = CONSvalue

fun unValueNumber(NUMBERvalue(i)) = i |
    unValueNumber(_) = raise Value

fun unValueBool(BOOLvalue(b)) = b |
    unValueBool(_) = raise Value

fun unValueHead(CONSvalue(c, _)) = c |
    unValueHead(_) = raise Value

fun unValueTail(CONSvalue(_, c)) = c |
    unValueTail(_) = raise Value

fun eqValue(c1, c2) = (c1 = c2)

(* Pretty-printing *)
fun printValue(NUMBERvalue(i)) = makestring(i) |
  printValue(BOOLvalue(true)) = "true" |
  printValue(BOOLvalue(false)) = "false" |
  printValue(NILvalue) = "[]" |
  printValue(CONSvalue(c)) = "[" ^
      printValueList(c) ^ "]"
  and printValueList(hd, NILvalue) = printValue(hd) |
  printValueList(hd, CONSvalue(tl)) =
      printValue(hd) ^ "," ^ printValueList(tl) |

50
printValueList(_) = raise Value
Appendix B: Files

The following files are available in the directory `/usr/cheops/mads/course`

- `interp1.sml` Version 1 (as included in Appendix A).
- `interp2.sml` ... `interp4.sml` The other versions.
- `build1.sml` the structure declarations needed to build Version 1.
- `build2.sml` ... `build4.sml` Similarly for the other versions.
- `parser.sml` The parser functor.

To build Version 3, say, you type the following (assuming you have copied the files to your directory):

```plaintext
use "interp3.sml";
use "parser.sml";
use "build3.sml";
```

Since the parser functor is completely closed, you dont have to include it more than once in every session, although you will probably want to build your system several times while you experiment with the extensions.