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. HASKELL and MIRANDA[1] are also functional programming languages; unlike ML, they are pure in the sense that they do not allow references and side-effects. HASKELL and MIRANDA are lazy languages, whereas ML is strict.

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.

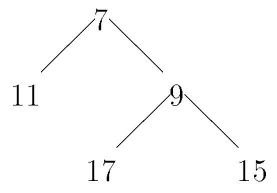
Like ADA, ML has language constructs for writing large programs. Roughly speaking, a *structure* in ML corresponds to a *package* in ADA; a *signature* corresponds to a *package interface* and a *functor* in ML corresponds to a *generic package* in ADA. However, ML admits structures (not just types) 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 *heap* is a binary tree of *items*, for example:

[1] MIRANDA is a trademark of Research Software Ltd.



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. *int*) or declared by the user (e.g. *item*).

```
type item = int;

infix leq;

fun (p: item) leq (q: item): bool =
  p <= q;

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;

val t = N(7, L 11, N(9, L 17, L 15));

fun top(L i) = i
  | top(N(i,_,_)) = i;
```

We start out by considering integer heaps only; therefore we first declare the type *item* to be an abbreviation for *int*. We then declare that *leq* is to be used as an infix identifier. Then we declare a function *leq* to be the pervasive `<=` on integers. We then declare the two functions *max* and *min* in which *leq* is used as an infix operator.

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 `datatype` declaration. `datatype` declarations are automatically recursive, i.e. data types can be declared in terms of themselves. This is illustrated by the declaration of *tree*. This data type has two *constructors*, L and N. Note that for example `7` is an item, but *L applied to* `7`, written L`(7)`, or just L `7`, is of type *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 *top* illustrates this. (*top* applied to a tree returns the item at the top of the tree). `(`$L\ i$`)` and `(`N`(`i`, _, _))` are examples of *patterns*. Applying a function (here *top*) to an argument (e.g. t) is done by matching the argument against the patterns till a matching pattern is found. For example *top* t evaluates to 7.

1.3 Recursive Functions

```
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
```

The function *depth* maps trees to integers; for instance *depth* t evaluates to 3. As spelled out in the declaration of *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 *depth* is *recursive*, i.e. defined in terms of itself. Another example of a recursive function is the function *isHeap* which when applied to a tree returns the value true if the tree is a heap and false otherwise.

Exercise 2 Write a function *size* which when applied to a tree returns the total number of items in the tree.

Exercise 3 The function *top* returns a minimal item of a heap. Write a recursive function *maxItem* which returns a maximal item.

1.4 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 *initHeap* which for given integer n returns a heap of depth n. This only makes sense for $n \geq 1$. This can be expressed in ML by raising an *exception* in the case $n < 1$. The effect of evaluating the expression `raise` e, where e is an exception, is to discontinue the current evaluation. Often, the exception will be *handled* by a `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.

```
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
```

Notice the `let` *dec* `in` *exp* `end` expression. To evaluate it, one first evaluates *initHeap*`(`n `- 1)` and binds the resulting value to t. Then one evaluates the body, N`(`*initial*`,` t`,` t`)` using this value for t. Notice that the scope of the declaration of t is the expression N(*initial*, t, t); 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 *leftSub* and *rightSub* which when applied to a tree returns the left and the right subtree, respectively.

Finally, we shall write a function *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.

```
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);

t;

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

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 l* above r in the tree?

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 *t1*.

Exercise 6 After the last declaration, what values are bound to *out2* and *t2*?

1.5 Structures

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

```
structure Heap =
struct
 type item = int;
 infix leq;
 fun (p: item) leq (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, L _) = ...
end; (* Heap *)

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

Identifiers declared in a structure are accessed from outside the structure by a *long identifier*, for instance *Heap.initHeap* (which can be read "the initHeap in Heap" or just "Heap dot initHeap"). In a large program containing many structures, long identifiers make it much easier for the reader to find the definition of an identifier.

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.

```
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.

1.7 Coersive 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 *HEAP*.

```
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 *)
```

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

In practice, one should write down a signature *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 *HEAP* signature. We then imprint the view provided by *HEAP* on the declaration of the structure *Heap* by a *signature constraint*:

```
structure Heap: HEAP =
struct
  type item = int;
  ...
end; (* Heap *)

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

After this declaration of *Heap*, we cannot write *Heap*.*t*, since *t* is not mentioned in *HEAP*. However, we can write *Heap*.*replace* as *replace* is specified. Moreover, although *HEAP* does not specify that *item* should be *int*, the ML system discovers that *item* is in fact *int* in *Heap* and that is why `7` will be accepted as an *item* in the application *Heap*.*replace*(`7`, *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 `datatype` by specifying it as a `type`.

1.8 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 *item*, any binary function *leq* on items and any *initial* item, the signature *HEAP* is satisfied by the declarations we have already written. Let us specify the general requirements of a *Heap* structure.

```
signature ITEM =
 sig
  type item
  val leq: item * item -> bool
  val initial: item
 end;
```

What we are after is a structure which is parameterised on any structure, *Item*, say, which matches *ITEM*. In ML, a parameterised structure is called a *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
 infix leq;
 fun (p: item) leq (q: item) : bool =
      Item.leq(p,q)
 fun intmax(i: int, j) =
      if i <= j then i else j
 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 and coord
 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, t1) =
      StringHeap.replace("ape",
       StringHeap.initHeap(1));
val (out2, t2) =
      StringHeap.replace("man", t1);
```

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 called a module, but ML programmers often refer to a module meaning "a structure or a functor".

2 The Professor Game

We shall now present an example of programming with ML modules. We shall write a program which plays a game known as "The Professor Game". The game consists of 16 square pieces of cardboard, all of the same size, each showing four half professors in colourful clothing (see Figure 1). The object of the game is to place the 16 pieces side by side in a 4×4 matrix such that whenever two pieces meet, a full professor clad in colours to match emerges.

Figure 1: One of the 16 pieces of the Professor Game (colours unfortunately not reproducible)

The box in which the pieces are bought contains a slip claiming that there are "one million possibilities" and that "only 8 are correct".

2.1 About the Game

A closer inspection of the pieces reveals that the professors can be distinguished by the colour of their coats alone and that all coats are either lilac, blue, red or green. Only two of the pieces are identical. However, one quickly discovers that every piece has two "upper professors" and two "lower professors" and that every piece can be turned such that it has a lower professor on the northern and western edges and an upper professor on the southern and eastern edges. From this one sees that if a professor in some column has his head pointing up, then all his colleagues in the same column must have their heads pointing up as well. It is not clear, however, whether a solution can have professors in some column heading up and professors in a neighbouring column heading down. Unfortunately, the vendor does not give away any clues. Thus one really gets two games for the price of one: the *fixed orientation game* and the *variable orientation game*.

How can one proceed to find all the solutions to these games? In the case of the variable orientation game, there are $16! \times 4^{16}$ arrangements of 16 different pieces in a 4×4 matrix so it is clear that one has to prune this potentially enormous search space dramatically, if one wants any machine to find the solutions.

Perhaps the most obvious way to find all the solutions is to try to fill in the places in board in some particular order. In effect, one traverses a search tree. Let us say that a *situation* is a node in the search tree. The situation at the root of the search tree is that the board is empty. At each node, one creates a subtree for each remaining piece that matches the next place of the current situation.

Notice that the above algorithm is parameterised on what situations are and notably on what it means for a piece to match a situation.

One possible interpretation of situations is that a situation consists of a partially filled board and a position where the next piece is to be put, if the board is not full. One possible interpretation of matching is that a piece fits a situation if it can be put down at the next position of the board without causing a clash with neighbouring pieces that have already been placed.

These interpretations will prune the search significantly, but one can do better still by using other interpretations that place more constraints on matching. One possible way of imposing further constraints arises from the observation that there are some half professors that can never be matched up with their missing half. In the fixed orientation game, for example, two pieces have a blue head at their southern edges and no piece has a pair of blue legs on its northern edge. That means that those two pieces will have to be at the southern border of any solution. (Similarly, there are two more pieces that have red legs on their western edges than there are pieces with red heads on their eastern edges, so in every solution there must be at least two pieces with red legs on the western border; similar constraints apply to the two remaining borders.)

2.2 The Use of Functors

The exhaustive search outlined above is parameterised in several ways. It is parameterised on the notion of matching. Also, it is essentially the same for the fixed orientation and the variable orientation game. Finally, it is to a large extent independent of the way pieces and situations are represented. This is where the notion of a functor is extremely useful. We shall in fact write a single functor (called *Solver*) which is parameterised on structures that hide the implementation of situations and pieces and also hide the choices concerning orientation and matching. In fact, the *Solver* is not to know much about the game it is solving; for example, it is not to know how pieces are represented, how big the board is and how many colours there are. All it will do is to traverse a search tree, pruning it in the process, and collect all the solutions it finds. By varying the structures we apply *Solver* to, we can in effect get it to solve different games using a variety of constraints and heuristics to prune the search.

Similarly, the representation of the board and the operations that place pieces do not need to know exactly what pieces are. Therefore we shall write another functor (called *Board*) which is parameterised on a structure *Piece*.

Finally, we shall write a functor *Piece* which is to define the implementation of pieces and the elementary operations on them without revealing any of these details to any of the other functors.

If one tries to sketch the code of the solver, it soon becomes clear that there are two natural views of the search process. The first view, let us call it the *imperative* view, is that the partially filled board and the remaining pieces constitute a *state*. Putting down a matching piece changes (i.e. side-effects) the state. The second wiew, let us call it the *applicative* wiew, is that putting down a piece given some situation creates a new situation in its own right without "changing" any situation. These two views both seem perfectly reasonable, but they lead to different interfaces on the *Solver* functor. For example, in the imperative view, one might expect the *Board* parameter structure to be specified with the following operations for placing and removing pieces

```
val place: situation * position * piece
     -> unit
val remove: situation * position -> unit
```

whereas in the applicative view, one would expect something like

```
val place: situation * position * piece
    -> situation
```

and no `remove` operation. Therefore we are forced to make a choice between the two views. We choose the applicative view, for no particular reason.

2.3 The Solver

The result signature of the *Solver* functor is as follows:

```
signature SOLVER=
sig
  type filename
  sharing type filename = string
  val solve: filename -> unit
end
```

We could of course have specified the result of the functor by the shorter

```
signature SOLVER=
sig
  val solve: string -> unit
end
```

but it would be less clear what the intention is.

What does the *Solver* need to know about pieces? It needs to know that there is a type of pieces (*piece*) and to get started, it needs a list of all the pieces of the game (*all_pieces*). Finally, it needs a function which can unfold the search tree as the search goes on. Hence we specify a function *expand_remaining*, which can expand a list l of remaining pieces to a list of pairs (p, l') where p is a piece which the *Solver* can try to match to the current situation and l' is the list of pieces that remain to be placed, in case the match is successful. In the case of the fixed orientation game, *expand_remaining*(l) and l have the same length, as every remaining piece is a candidate for matching; in the case of the variable orientation game, *expand_remaining*(l) is four times the length of l, as every remaining piece potentially can be placed in four different ways. (Recall that the *Solver* itself does not know which of the games it is solving.) Thus we arrive at the following interface to the *Solver*:

```
signature PIECE=
sig
  eqtype piece
  val all_pieces: piece list
  val expand_remaining: piece list ->
                (piece * piece list) list
end
```

The `eqtype` specification specifies that the type *piece* is to admit equality.

What does the *Solver* need to know about the board? It needs to know that there is a type which represents a current situation in the search (*situation*). To start the search, it needs an initial situation (*initial_situation*). Also, it needs a match operation (*match*) which can match a piece to a situation and either reject the piece or return a new situation. Moreover, it needs an operation for printing a list of situations (the solutions of the game). Finally, the *Solver* might find the same solutions more than once, so it also needs an operation (*thin*) which transforms a list of solutions to a list of solutions without duplicates.

```
signature BOARD=
sig
  type piece and situation
  val initial_situation: situation
  datatype result_of_match =
    REJECT
  | ACCEPT of situation
  val match: situation * piece ->
                          result_of_match
  val output_solutions:
    outstream -> situation list -> unit
  val thin:
    situation list -> situation list
end;
```

Why does *BOARD* specify the type *piece*? The answer is that the type constructor *piece* occurs in the specification of the value *match*. If we had not specified *piece* in *BOARD* then that occurrence of *piece* would refer to some global declaration of a type *piece*. But we do certainly not want to have to declare a type of pieces (and hence decide what pieces are) just because we are declaring the solver; therefore we specify *piece* as a type in the signature *BOARD*. In fact, as a matter of convention, we always make sure that our signatures are closed except for identifiers that occur in the initial basis (such as *int* and *unit*) or are bound to signatures.

The skeleton of the *Solver* appears below.

```
functor Solver(
  structure L : LIST
  sharing type L.list = list
  structure B: BOARD
  structure P : PIECE
  sharing type P.piece = B.piece
  ): SOLVER=
struct
  open B P L;
...
end
```

The *Solver* is parameterised on a structure with signature *LIST*. This signature is defined in *The Edinburgh SML Library*.[2] One of the functions specified in *LIST* is *foldL* which informally is defined by the equation

$$foldL(f)(b)[x_1, \ldots, x_n] = f x_n(\ldots f x_1 b \ldots)$$

Why do we have the sharing constraints in the declaration of the *Solver*? Both *B* and *P* are specified to contain a type called *piece* but there is no a priori reason why these types should be the same, just because they have the same name: it is very easy to construct a real structure *Board* which matches *BOARD* and a real structure *Piece* which matches *PIECE* without *Board.piece* and *Piece.piece* being the same type. However, for the solver to make any sense, the pieces that *P* knows about and the pieces that *B* knows about must be of the same type. Indeed the body of the solver would not type-check if we omitted the sharing constraint.

[2]Dave Berry: The Edinburgh SML Library, Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh.

```
type remain = piece list
type job = situation * remain

fun new_jobs((sit,rem):job):job list =
 let
  fun filter [] = []
   | filter ((piece,rem')::rest) =
      case B.match(sit,piece) of
        REJECT => filter rest
      | ACCEPT sit' =>
           (sit',rem') :: filter(rest)
 in
  filter (P.expand_remaining rem)
 end
```

```
type solution = situation

fun solve_job job acc_sol: solution list  =
  case job of
    (sit,[]) => (* no pieces remain *)
      sit::acc_sol
  | (sit, remain)=>
      foldL
        solve_job
        acc_sol
        (new_jobs(sit,remain))
```

An example of a function in the body of *Solver* which refers to the specified components of L, B and P is the function *new_jobs* which from a *job* (i.e. a node in the search tree) constructs a list of new jobs to be tackled. It does so by using the *P.expand_remaining* and *B.match* operations. (Because of the `open` B P L, the qualifiers $P.$ and $B.$ could be omitted.)

The search for solutions is implemented by the following function, which uses *new_jobs*:

It is then straightforward to declare the function *solve*. The definition can be found in the Appendix, which contains a complete program listing.

2.4 The Board

The *Board* functor will need to know more about pieces than the *PIECE* signature specifies. In order to implement function *match*, the *Board* needs to be able to look at what half professors are at the four edges of a piece. Also, in order to implement the printing of solutions, the interface should provide an operation for printing a single piece. In fact, since printing a piece in a readable form spans over more than one line in the output, and since pieces in the same row of the board should be printed beside each other, it is convenient to specify a type of printable representations of pieces (*hbox*) and operations for creating, concatenating and printing hboxes.

In order to expand the original specification *PIECE* one could of course go back and edit the declaration of *PIECE*. However, this would mean that the *PIECE* interface to the *Solver* would specify many components that are not relevant to the *Solver* at all. Alternatively, we could just copy the declaration of *PIECE*, expand it with the editor and give it a new name. This, however, would mean that there was duplicated text in the two signatures, making maintenance harder. Fortunately, ML allows one to `include` a signature in another (much like one can `open` a structure in a structure or a signature). Thus the enrichment of *PIECE* is achieved by the following declaration:

```
signature FULL_PIECE=
sig
 include PIECE

 (* edges *)
 type edge
 val north: piece -> edge
 val south: piece -> edge
 val east: piece -> edge
 val west: piece -> edge
 val match_edges:
   edge * edge -> bool

 (* printing *)
 type hbox
 val box: piece -> hbox
 val end_row: hbox
 val concat:
   hbox -> hbox -> hbox
 val empty_piece: hbox
 val output_hbox:
   outstream -> hbox -> unit
end;
```

The skeleton of the *Board* functor appears below.

```
functor Board(
   structure List: LIST
   sharing type List.list = list
   structure Piece: FULL_PIECE
   ): BOARD =
struct
  open List Piece
  ...
end
```

Using the components specified in *FULL_PIECE* (and *LIST*) we have to write the body of the functor so that it matches the *BOARD* signature. That is, the *BOARD* signature is the agenda of what we have to do next. We first have to declare the type *situation*. This is achieved as follows:

```
type position = int * int
val k = 4 (* number of rows
          and columns *)

datatype 'a option =
   NONE | SOME of 'a
type board = piece option list list
type situation = board * position
```

A board is represented by a list of rows, each row being a list of 4 places, where each place is either empty or contains a piece. A situation is a board and a position which indicates an empty place where the next piece is going to be put, if it matches. We choose to fill the board sequentially, starting from the top left-hand corner (position (0,0)) completing a row at a time till we reach the bottom right-hand corner (position (3,3)). Position (i, j) indicates the place in the i^{th} row and j^{th} column.

The next item on the agenda is the value *initial_situation*. The initial situation is that we have an empty board and are ready to place a piece at position $(0, 0)$.

```
fun explode 0 x = []
  | explode n x = x :: explode (n-1) x

val initial_situation =
   (explode k (explode k NONE),
   (0,0));
```

We can continue going through the agenda till we have written all the components specified by the *BOARD* signature. We omit the details here; they appear in the Appendix. At this point we can compile the *Board* functor and when we get it to type-check, we can forget all about *Board* and *Solver* and turn our attention to the implementation of pieces.

2.5 The Piece

The *Piece* functor is particularly simple in that it need not be parameterised on any structure. Conceptually, *Piece* is one of the basic building blocks which the other modules build on. It is useful, however, to parameterise the *Piece* functor on a function which maps a quadruple of edges (representing an oriented piece) to a list of quadruple of edges (representing the turned versions of the piece). In the fixed orientation game, the list will have length 1; in the variable orientation game, the list will have length 4. The skeleton of the *Piece* functor is

```
functor Piece(
  val turns: 'a * 'a * 'a * 'a ->
          ('a * 'a * 'a * 'a)list
  ): FULL_PIECE=
struct
  ...
end
```

Now our agenda is the *FULL_PIECE* signature (which includes *PIECE*). The declaration of the type *piece* and its associated destructor operations is straightforward

```
datatype colour = RED | BLUE |
    LILAC | GREEN
datatype part = HEAD | LEGS
type edge = colour * part
type piece = edge * edge * edge * edge
fun north(n,_,_,_) = n
and west(_,w,_,_) = w
and south (_,_,s,_) = s
and east (_,_,_,e) = e;
```

The hardest bit of writing the entire system, writing down the representation of the 16 pieces without making mistakes, comes next:

```
val all_pieces = [
((LILAC,LEGS),(GREEN,LEGS)
 (RED, HEAD),(BLUE, HEAD)),
((LILAC,LEGS), (RED,LEGS),
 (RED, HEAD),(GREEN,HEAD)),
((LILAC,LEGS), (RED,LEGS),
 (BLUE,HEAD),(GREEN,HEAD)),
((LILAC,LEGS), (RED,LEGS),
 (BLUE,HEAD),(RED, HEAD)),
((LILAC,LEGS),(GREEN,LEGS),
 (BLUE,HEAD),(RED, HEAD)),
((LILAC,LEGS), (RED,LEGS),
 (BLUE,HEAD),(GREEN,HEAD)),

((GREEN,LEGS),(GREEN,LEGS),
 (LILAC,HEAD),(RED,HEAD)),
((GREEN,LEGS),(BLUE,LEGS),
 (LILAC,HEAD),(RED,HEAD)),
((GREEN,LEGS),(RED,LEGS),
 (LILAC,HEAD),(GREEN,HEAD)),
((GREEN,LEGS),(RED,LEGS),
 (LILAC,HEAD),(BLUE,HEAD)),


((BLUE,LEGS),(GREEN,LEGS),
 (LILAC,HEAD),(RED,HEAD)),
((BLUE,LEGS),(RED,LEGS),
 (GREEN,HEAD),(LILAC,HEAD)),
((BLUE,LEGS),(GREEN,LEGS),
 (LILAC,HEAD),(GREEN,HEAD)),
((BLUE,LEGS),(RED,LEGS),
 (GREEN,HEAD),(GREEN,HEAD)),

((RED,LEGS),(GREEN,LEGS),
 (BLUE,HEAD),(LILAC,HEAD)),
((RED,LEGS),(BLUE,LEGS),
 (BLUE,HEAD),(RED,HEAD))]
```

Writing the rest of the functor body is not very hard; the details appear in the Appendix.

2.6 Building the System

Finally, we can build the system. We generate three structures (*Piece*, *Board*, and *Solver*) by applying the functors we have already written:

```
structure FixedPiece = Piece(
  fun turns x = [x]);

structure FixedBoard = Board(
  structure List = List
  structure Piece = FixedPiece);

structure FixedSolver = Solver(
  structure L = List
  structure B = FixedBoard
  structure P = FixedPiece);

open FixedSolver;
```

One can now test the solver by

```
solve("fixedsolutions")
```

which will print a nicely formatted list of all the solutions to the fixed orientation game (there are 5) on file `fixedsolutions`.

2.7 Variations and Extensions

To get a solver for the variable orientation game, one needs to modify the parameter *turns* to *Pieces* before reapplying the functors.

```
structure VariablePiece = Piece(
 fun turns ··· = [···]);

structure VariableBoard = Board(
 structure List = List
 structure Piece = VariablePiece);

structure VariableSolver = Solver(
 structure L = List
 structure B = VariableBoard
 structure P = VariablePiece);

open VariableSolver;
```

It turns out that one more change is needed. The *thin* function in the *Board* functor does not remove solutions that arise from each other by turning the entire board. How can we make another version of the *Board* functor which has a more sophisticated *thin* operation? The problems are the same as when we wanted to enrich the *PIECES* signature, and the solution is similar. We just make a new functor based on the old one:

```
functor EnrichedBoard(
  structure List: LIST
  sharing type List.list = list
  structure Piece: FULL_PIECE
  ): BOARD =
struct
  structure Old = Board(
    structure List = List
    structure Piece = Piece)
  open Old;

  fun thin l = ...
end;
```

For this solution to work, one has to remove the result signature from the original declaration of *Board*; otherwise the components that become accessible via `open` in the new functor will not suffice to write the new *thin* function.

Introducing constraints that further prune the search tree can be done by modifications to the *Board*; one changes the definition of the *situation* type to include constraints and modifies *match* accordingly. Note that no change to the *Solver* is required.

3 Assessment of the Modules System

Having presented several examples of ML modules, let us address the question: what are the advantages and disadvantages of the modules system, seen from the perspective of the working programmer. Also, it it relevant to ask from a language design point of view whether it really is necessary to introduce a modules system with the power of the ML Modules. Would something simpler do, for example having structures and signatures, but not functors? Finally, we shall outline the key insights of semantics that underlie not just the formal definition of the language but also (and probably more importantly) the usefulness of the modules system.

3.1 Functors and Abstraction

Some programming languages have a modules system that corresponds to having structures and signatures but no functors. Why are functors important to programming?

There are several reasons for this, but an important one has to do with being able to abstract from implementation details. The simplest kind of abstraction one can want of a modules system is the ability to hide information about an already declared object, typically by restricting the scope of identifiers that are to be internal to the module. In ML, the `abstype` construction and the ability to constrain structure declarations by signatures are of this kind. However, what is obtained in this way is simply the hiding of details of implementation that *have already been done*, not the ability to *postpone* implementation. The ability to postpone implementation only exists in languages where it is possible to parameterise on modules. This is why functors are so important.

In my experience, the most common reason for writing a module as a functor is not that one wants to apply it to more than one argument, but simply that one wants to postpone making certain implementation decisions. The ability to have functions as parameters to functions is not enough to achieve this, partly because one often wants to parameterise on a type together with a collection of operations on it, and partly because the thing one wants to parameterise on is a declaration rather than an expression.

3.2 Top-down versus Bottom-up Program Development

It is important that the constructs a programming language provides fit naturally with the way programmers develop programs. Some texts on program development advocate a strict top-down development of programs; in terms of ML modules, one way of imposing a top-down discipline would be to insist that all signatures be completed before the writing of structures and functors begins.

Other texts advocate a bottom-up approach; in terms of ML modules, this could be achieved by programming with structures whenever possible and using signatures just to hide information in already declared structures.

As programmers know, neither the top-down nor the bottom-up paradigm is an accurate description of how programming takes place in practice. There is no reason why one should feel ashamed of being unable to follow either discipline. Program development involves a mixture of problems one has thought about and problems one knows exists but has not yet thought about. There is no way one can make sure that the solution of the unsolved problems will not require solving some of the already solved problems in another way.

One finds that programming involves both programming in the small (writing functions and declaring datatypes) and programming in the large (structuring of programs) and that these two interact. It is therefore extremely important that a modules system be able to support the interaction between the programming in the small and the programming in the large.

The section on the Professor Game is intended to demonstrate that ML does support this interaction. Writing a functor body is programming in the small. As the programming of the functor body progresses, one discovers what external types and operations one requires and these are then moved to the parameter list of the functor. Naturally one tries to organize these requirements into groups of related requirements, thereby in effect creating new signatures or changing already existing ones.

To sum up, the interaction between low-level implementation work and high-level structuring work, which is characteristic of real program development, is made possible largely through the fact that the language has functors.

3.3 Experience with Type-checking

ML Modules are subject to static checking just as the Core language is subject to type-checking. Let us use the term *elaboration* to cover both kinds of static analysis, and let us use the term *elaborater* for the part of a compiler which implements elaboration.

The advantage of elaboration in the Core language is that once a program gets through the elaborater, most of programming mistakes have been corrected. The advantage of elaboration of Modules is even more important: once a functor gets through the elaborater, one can put the functor aside and concentrate on other things. Because the body of the functor elaborated, it certainly has a high degree of formal consistency, and because the functor in its entirety elaborated, the argument and result signatures of the functor really are interfaces to the module. In most cases, these interfaces are all one needs to consult when one subsequently needs to refer to the functor.

In one project, the *ML Kit*, by Nick Rothwell, David Turner and the author, about 8000 lines of ML modules were written in about 6 weeks. All modules were written as closed functors, in whatever order seemed appropriate, so it was impossible to run programs while the programming went on. When the last basic functors were finally written, the whole system was built using functor applications. Getting the applications to elaborate took about one and a half hours, after which one had a running system. There were still mistakes in the code, but building the system and getting it to run reasonably well so quickly was possible only because the type-checking of the functions in the modules had removed most of the mistakes in the individual modules and the elaboration of the modules in general had ensured that they could be put together in a consistent manner.

3.4 Modules do not Solve all Programming Problems

One might think that a major cost connected with using modules, in terms of programming effort, is that one has to write all these signatures that do not really *do* anything. There is no denying that signatures make the the programs longer. However, in my experience, it is not a burden to write signatures, because many signatures arise naturally as specifications of parameter structures when one writes functor bodies. A much more real difficulty, still in my experience, is to assess when it is good to abstract problems and hide information (thereby creating more modules and interfaces) and when it is better to package things together in a single functor.

There are difficult problems to do with coordinating programmers efforts that modules do not solve. It is nice when one can simply write down a signature and just give it to another programmer and ask him or her to write a functor that matches it. In reality, however, signatures are often revised as a result of the insight which writing functors provides, so that method often does not work. It is virtually impossible to get the overall structure of the program right in the first attempt and whenever one revises structure, the changes can affect the work of several programmers.

Regardless of the language they use, conscientious programmers always devote considerable energy to organizing their programs into some kind of units and try to ensure that there are clean interfaces between the various units. What ML's module system provides is essentially the ability to express the results of this effort in the language itself thereby making it possible for a compiler to check that a high degree of consistency between modules exist even when parts of the system have not yet been written.

3.5 The Semantics of Modules

We have argued that signatures and functors are good to have. Let us now look briefly at their semantics.

To make things simple, let us first restrict the modules system as follows: omit functors altogether and omit signature declarations but allow signature expressions to constrain structure expressions:

$$\mathit{strexp} : \mathit{sigexp}$$

Of course this language is not terribly useful, but the semantics is simple. One can define what it is for a structure expression *strexp* to elaborate to a structure S in some basis B, written

$$B \vdash \mathit{strexp} \Rightarrow S$$

(The basis B contains an environment which gives the meaning of the free identifiers in *strexp*.) The semantics of matching is simple: if *strexp* elaborates to S then *strexp* : *sigexp* elaborates to S', where S' is obtained from S by deleting the components that are not mentioned in *sigexp* (provided of course that S satisfies all the specifications listed in *sigexp*).

Next, let us introduce signature declarations and identifiers. This gives us the ability to write signatures before we write structures. Also, we achieve a separation of signature expressions from the concept of matching. Signatures become objects in their own right that can be named, included in other signatures, and so on. As a consequence one needs to be able to elaborate a signature expression to some semantic object which represents the signature expression in the context it was declared. This semantic object is called a *signature* in the definition of the language. The terminology is that a signature expression *sigexp* elaborates to a signature Σ in basis B, written

$$B \vdash \mathit{sigexp} \Rightarrow \Sigma$$

Consider the signature

```
signature SIG =
sig
  type t
  val x: t
end
```

Here t is not some particular type, but rather an identifier which stands for some unknown type. Thus one can think of a signature as a polymorphic (or *generic*) structure, in which the specified types and structures are universally quantified. Using the notation of the language definition, a signature Σ is of the form

$$\Sigma = (N)S$$

where S is a structure, N is a set of names of types and structures specified in the signature expression and the parenthesis indicate universal quantification. It is similar to the form of type schemes

$$\sigma = \forall\alpha_1, \ldots, \alpha_n.\tau$$

in the Core language.

Now we can replace the primitive notion of syntactic matching by a definition of what it is for a structure S to match a signature $\Sigma = (N)S'$. Matching breaks down into two parts:

S *matches* $(N)S'$ if $(N)S' \geq S^- \prec S$, for some S^-

The relation $(N)S' \geq S^-$ is called *instantiation* and is concerned with instantiating the generic names N to the actual names in S. It is similar to the instantiation of type schemes to types in the Core:

$$\forall\alpha_1, \ldots, \alpha_n.\tau' \succ \tau$$

The other relation, $S^- \prec S$, is called *enrichment*, and allows S to have more components and more polymorphism than the signature.

Whenever $B \vdash sigexp \Rightarrow \Sigma$ then Σ has to be as general as possible, meaning that whenever S is a structure which satisfies all the specifications of *sigexp* then S matches Σ. Conversely, Σ must not be so general that there are structures that match Σ but do not satisfy all the specifications of *sigexp*. In other words, Σ should be such that for all stuctures S, S matches Σ if and only if S satisfies all the specifications in *sigexp*. The technical terminology for this is that Σ should be *principal* for *sigexp* in B.

Finally, let us add functors to the language. Consider the declaration

`functor` $f(X$: *sigexp*$)$:*sigexp*$'$=*strexp*

Two requirements must be met. First, it must be possible to elaborate f without first having elaborated the structures to which it is subsequently applied (since these may not yet exist). Second, once f has been elaborated successfully, it must be possible to apply it to any argument that satisfies the specifications of *sigexp* (for otherwise *sigexp* could not justly be called an interface to f).

Notice that to meet the first requirement, it is essential that signature expressions elaborate to semantic objects in their own right (rather than serving the purpose of restricting the view of existing structures), for *sigexp* is now used to specify the formal structure X.

Inside the functor body, X should be bound to a formal structure which matches the parameter signature. In the semantics, this is achieved by letting this formal structure be a "fresh" instance of the parameter signature Σ.

It is crucial that Σ be principal for *sigexp*. If Σ were matched by some structure that did not satisfy the specifications of *sigexp* then we would not know how to interpret a type error discovered in the functor body: would the error be a genuine error or just be due to the elaborater picking an instance of Σ which did not satisfy the specifications of *sigexp*? Conversely, if there existed a structure S which satisfied all the specifications of *sigexp* but did not match Σ, how could we know that it would make sense to apply the functor to all structures which satisfy the specifications in *sigexp*? Thus principality is crucial for ensuring the second requirement.

It can be proved that whenever a signature expression elaborates at all, then it elaborates to a principal signature. Thus it really is possible to elaborate functors just once (namely when they are declared) and one can be certain that one can apply the functor to any argument that satisfies the parameter specifications.