SML Modules

Michael P. Fourman

February 2, 2010

In earlier lectures, we have discussed the use of signatures and structures to specify and group together related types and functions. In this note we introduce a final component of the ML modules system, *functors,* which support "top-down" software development, and the implementation of reusable code.

We begin with a brief review of signatures and structures, then introduce functors as a mechanism for code reuse, and finally discuss the use of functors in software development.

1 Signatures and Structures

As we have seen, an implementation of a data-structure, such as a queue, provides a type, together with a collection of functions. We can group together a collection of values using a record. However, to provide a queue package, we need to group together some values *and* some types. To do this we need a new kind of structuring mechanism, a *structure.* Values have types associated with them and, analogously, structures have *signatures.*

We can access the components of a structure using a *qualified name.* Thus `IntQueue.empty` would represent the empty queue, and the function to add an element to a queue could be accessed as `IntQueue.enq` (e.g. `IntQueue.enq(1, IntQueue.empty)`). Structures are *not* values. We cannot pass them to other functions, embed them in other values etc.

Signatures can be used to specify what we require of a structure. A structure is an *implementation* of, or *matches* a signature provided it declares the specified types and values (in any order). Signatures are like types: a most general signature is inferred for each structure, just as a most general type is inferred for each value; just as we can constrain the type inferred for a value using a type constraint, we can constrain the signature for a structure using a *signature constraint.* We can use as signature to hide values and

constructors, used to implement the structure, that should not be visible to the outside world.

2 Reusable Code

Functions such as `length` are polymorphic; we can apply the length function to a list of any type of value. Similarly, we could easily use lists to provide a polymorphic implementation of stacks; to implement the stack operations, we don't need to know what type of value is being manipulated. Life is not always so easy.

Consider the example of a Priority Queue. Our implementation of a priority queue of integers as an ordered list depends on the type of item in the queue, and, crucially, on the priority ordering we choose. Clearly we could use the "same" idea to implement other priority queues. We would like to parameterise our implementation, on a type of item, and a chosen priority ordering.

Given a type and an ordering on that type:

```
type Item
val > : Item * Item -> bool
```

we can implement a priority queue, using the given predicate (which we assume is a total ordering), as the priority order. The result is a structure, containing a new type representing the queue and a collection of operations that act on this type. From a collection of types an values (a structure) we construct a new structure; we express this construction as an ML *functor.* Just as a function takes a value and produces a new value, so a functor takes a structure and produces a new structure. We can use the code for `IntPQ` to implement the priority queue construction:

```
functor PQUEUE(type Item
               val > : Item * Item -> bool
              ):QueueSig =
struct
    type Item = Item
    exception Deq
    fun insert e [] = [e]:Item list
      | insert e (h :: t) =
        if e > h then e :: h :: t
                 else h :: insert e t

    abstype Queue = Q of Item list
    with
        val empty          = Q []
        fun isEmpty (Q []) = true
          | isEmpty _      = false

        fun enq(Q q, e)    = Q(insert e q)
        fun deq(Q(h :: t)) = (Q t, h)
          | deq _          = raise Deq
    end
end;
```

The syntax for functors uses a new keyword, `functor`. The functor name is followed by a list of formal parameters, and then a body just like that of a structure declaration. In the declaration of `PQUEUE` the formal parameters are specified as types and values. Structures, each constrained by a signature, can also be introduced as formal parameters to functors; we will see examples of this later.

How can we use this functor? We provide appropriate types and values to match the formal parameters. Suppose we want to work with queues of integers, greater integers having higher priority, we apply the functor `PQUEUE` as follows

```
structure IntPQ = PQUEUE(type Item = int
                         val op >  = op > : int * int -> bool)
```

The structure `IntPQ` behaves just like our earlier implementation. But now the code is reusable; we can create many different argument structures—and so many different priority queues—without rewriting the code implementing the priority queue. Using functors we can write reusable code to implement datastructures that depend on a collection of types and values.

If, for example, we want a queue of pairs of integers, with a particular priority ordering, this could be given by

```
local
    fun higher((x,y), (x',y')) = (x:int) > x'
                                 orelse (x = x' andalso (y:int) > y')
in
structure PairPQ = PQUEUE(type Item = int*int val op > = higher)
end
```

We can also apply the functor directly to a structure that provides the appropriate parameter types and values. We can apply the functor `PQUEUE` to any structure that matches the signature `OrdSig`

```
signature OrdSig =
sig
    type Item
    val > : Item * Item -> bool
end;
```

For example, we could package various operations on integers in a structure `IntItem`

```
infix 4 ==
structure IntItem =
struct
    type Item = int
    val op == = (op = : int * int -> bool)
    val op >  = (op > : int * int -> bool)
end;
```

and then can apply the functor `PQUEUE` to this structure

```
structure IntPQ = PQUEUE(IntItem)
```

Instead of providing arguments to functors piecemeal, it is often more convenient to implement structures, such as `IntItem`, that will match a variety of signatures, and so can be used in a wide variety of contexts.

3 Structured Software Development

As well as supporting the development of reusable code, functors provide a mechanism for documenting and controlling the dependencies between software components. To illustrate this we revisit the stack-based evaluator for expressions introduced in Practical 4.

The code given in the appendix to Practical 4 consists of five inter-related structures; the implementation of one structure uses types and values provided by another. In this code, the dependencies are expressed using qualified

names. Although this is a simple system, it requires close examination of the code, and a moment's thought, to determine the right order in which to compile the various structures. It probably took you some time to realise that, for the purposes of Practical 4, you only needed to look at one structure `Expn`.

Using functors, we can make the dependencies explicit. Here is the code needed to build the structure `TopLevel` from a collection of functors.

```
local
 structure Expn    = EXPN()

 structure Environment = ENVIRONMENT()

 structure Machine =
    MACHINE(structure Environment = Environment)

 structure Compile =
    COMPILE(structure Machine = Machine
                  and Expn = Expn)
in
structure TopLevel =
    TOPLEVEL( structure Machine = Machine
                    and Compile = Compile
                    and Environment = Environment
                    and Expn = Expn )
end;
```

The dependencies are made explicit: `Expn` and `Environment` have no dependencies, `Machine` depends on `Environment`, `Compile` depends on `Machine` and `Expn`, and `TopLevel` depends on everything else. Moreover, the code for the various functors can be written and modified independently, provided the interfaces are not changed. The language of functors can also help in specifying a coding task. For example, if in Practical 4 you had been asked to implement a functor with header

```
functor OPTIMISE(structure Expn:ExpnSig):OptimiseSig
```

you would have known immediately that only the signature `ExpnSig` was relevant to your task.

We use the rest of this note to look in more detail at the implementation of this example. The example provides code modelling the compilation and execution of code for a simple stack machine. The source language for the compiler consists of algebraic expressions.

Specification

We begin with the interfaces, given by signatures that will be used to specify the formal parameters to our functors. Starting at the beginning, the datatype `Expn` representing expressions is declared in a structure of its own. The signature carries a complete description of the type—this is the type we want; nothing else will do.

```
infixr 6 ** infixr 4 ++ ;
signature ExpnSig = sig
datatype Expn =
         Id    of string      (* identifiers    *)
       | Lit   of int         (* literals       *)
       | op ++ of Expn * Expn (* addition       *)
       | op ** of Expn * Expn (* multiplication *)
end;
```

The system we are modelling will allow us to make a sequence of declarations, analogous to ML `val` declarations. A declaration is represented as a `string*Expn` pair. The `TopLevel` structure will provide a function that compiles a list of declarations to produce an environment, together with a structure `Expn` representing expressions, an environment type, and a function to allow the user to lookup the values of identifiers in the environment. All this is specified in the signature `TopLevelSig`

```
signature TopLevelSig =
sig
   structure Expn : ExpnSig
   type Environment

   val compile : (string * Expn.Expn) list -> Environment
   val lookup  : Environment -> string -> int
end
```

The Environment will be implemented by a structure `Environment`. This will be used by other modules that will need access to an empty environment, and a function to add new bindings to the environment, as well as the lookup function already mentioned.

```
signature EnvironmentSig =
sig
   type Environment
   val empty  : Environment
   val lookup : Environment -> string -> int
   val enter  : (string*int) * Environment -> Environment
end
```

The stack machine is modelled by the structure `Machine` this provides a type of `Action`, the primitive machine instructions, and a function that produces the result of executing a sequence of actions in a given environment

```
signature MachineSig =
sig
      type Environment
      datatype Action = Add | Mul
                      | PushLit of int
                      | PushVal of string

      val execute : Environment ->
                      Action list -> int
end
```

The compiler produces machine code for a given expression.

```
signature CompileSig =
sig
 type Expn
 type Action
 val code : Expn -> Action list
end;
```

Each signature includes any types (except the built-in types) used in the types of the values it declares. This makes the signatures self-contained.

Implementation

We don't want to dwell on the details of the code used to implement the system—it uses tree traversal and lists in a straightforward way, and doesn't introduce any new ideas—but the functor `TOPLEVEL` does illustrate a subtlety that often arises when using functors and structures. We begin with the more-straightforward examples. The code is just like that given in the appendix

to Practical 4. It is described there.

```
infixr 6 ** infixr 4 ++ ;
functor EXPN() = struct
datatype Expn =
         Id    of string      (* identifiers    *)
       | Lit   of int         (* literals       *)
       | op ++ of Expn * Expn (* addition       *)
       | op ** of Expn * Expn (* multiplication *)
end;

functor ENVIRONMENT():EnvironmentSig =
struct
    exception Lookup
    type Environment = (string * int) list

        val empty =  []

        fun lookup ((k,e)::t) key = if(k = key)then e
                                                else lookup t key
          | lookup _ _ = raise Lookup

        fun enter (entry, entries) = entry :: entries
end;
```

```
functor MACHINE(structure Environment:EnvironmentSig) =
struct
type Environment = Environment.Environment
datatype Action = PushLit of int
                | PushVal of string
                | Mul
                | Add

fun execute env code =
let exception Eval

    fun v s = Environment.lookup env s

    fun run(args, PushLit n :: ops) = run(  n :: args, ops)
      | run(args, PushVal e :: ops) = run(v e :: args, ops)
      | run(a::b::args, Mul :: ops) = run(a*b :: args, ops)
      | run(a::b::args, Add :: ops) = run(a+b :: args, ops)
      | run([result],           []) = result
      | run _ = raise Eval
in
    run([], code)
end
end

infixr 4 ++
infixr 6 **;
functor COMPILE( structure Expn:ExpnSig
                       and Machine:MachineSig
               ):CompileSig =
struct
      open Expn Machine
      fun codeacc (Id s,   rest) = PushVal s :: rest
        | codeacc (Lit n,  rest) = PushLit n :: rest
        | codeacc (a ++ b, rest) = codeacc(a, codeacc(b, Add :: rest))
        | codeacc (a ** b, rest) = codeacc(a, codeacc(b, Mul :: rest))

   fun code expn = codeacc(expn, [])
end;
```

```
functor TOPLEVEL(
    structure Machine: MachineSig
          and Compile: CompileSig
          and Environment: EnvironmentSig
          and Expn : ExpnSig
 sharing type Compile.Action = Machine.Action
 sharing type Environment.Environment = Machine.Environment
 sharing type Compile.Expn = Expn.Expn
):TopLevelSig = struct
    structure Expn = Expn
    type Environment = Environment.Environment
    val lookup = Environment.lookup

    local
    fun adddecs ((s,e) :: decs) env =
           let val v = Machine.execute env (Compile.code e)
           in adddecs decs (Environment.enter((s,v), env)) end
      | adddecs [] env = env
    in
        fun compile decs = adddecs decs Environment.empty
    end
end;
```

The parameter specification for the `TOPLEVEL` functor includes a number of *sharing constraints.* This is the subtlety we spoke of: consider the following line of code

```
v = Machine.execute(env)(Compile.code(e))
```

From the signatures `MachineSig` and `CompileSig` we see that types of the functions are

```
Machine.execute : Environment -> Action list -> int
Compile.code : Expn -> Action list
```

The signatures `MachineSig` and `CompileSig` both include a type called `Action`, but there is no guarantee that implementations of these signatures will both use the *same* implementation for this type. However, when we write the line of code given above, we certainly intend that the functor will only be applied to structures, `MachineSig` and `CompileSig`, that use the same implementation for the type `Action`. The sharing constraint tells the compiler that this is what we intend (so it can type check the code), and when we apply the functor the system checks that the constraining is satisfied. Unexpected type errors when developing a new functor may result from

a missing sharing constraint.

Modules and Abstraction

Functors provide yet another abstraction mechanism: when we write the body of a functor we can only rely on the specification of the formal parameters—there is no implementation. So if we always use functors, and always access our abstract data types via formal parameters, we don't need to use abstype as a extra abstraction barrier. However, using the `abstype` mechanism is the only way to erect an impregnable barrier in ML. We will make our implementations of datastructures such as queues abstract using `abstype`; when using functors to organise the code for an application, we take a more relaxed view and normally rely on the disciplined use of functors to ensure that we are not making unwarranted use of implementation details.

Styles of use

Books that talk about software engineering usually have a section that discusses bottom-up and top-down program development. In the bottom-up approach you start, as the name suggests, from the bottom and work your way towards the goal, i.e. the problem to be solved. Your initial environment might not support sets, for example, and so your first step might be to implement a set package. Using this you can then write a package to support graphs. At each step you are increasing your programming 'vocabulary' until you eventually reach the stage where you can express the solution to the original problem in a concise form. In the top-down approach you express the solution to the original goal in terms of the solutions to some subgoals. You then develop algorithms to solve the subgoals, each of these depending on solutions to further subgoals. This process is repeated until you reach a goal that is trivial enough to be solved using the basic building blocks of the language directly.

Both of these approaches have their problems. The bottom-up approach is not goal directed – one might waste time developing some code that may never be used in the final program. The advantage of this strategy is that the code can be run and tested at each step. The top-down approach is clearly driven by the goal to be solved. Unfortunately, at each stage the solution of a goal depends on the code for subgoals that have yet to be solved. It can be difficult to test the program until all of the code has been written. In reality, of course, people will use a mixture of these two approaches. The hope is

that they will reap the benefits of both approaches. The danger is that they will suffer the disadvantages of both.

Structures and functors can be used to support both styles of program design. In the bottom-up approach the basic building blocks are structures. We start by building some simple structures, and then apply functors to these to create more complex structures. We continue this process until we eventually produce a structure representing the final solution. At each stage in the development we will have a collection of structures that can be tested. In the top-down approach the basic building blocks are functors. We start by writing a functor to solve the main goal. This will be parameterised on signatures representing solutions to some subgoals. We can type-check this functor, but we cannot evaluate the code until we apply the functor to structures matching the argument signatures. To produce these we must first solve the subgoals using other functors. These will also depend on signatures representing smaller subgoals. You can visualise the process as building a tree, starting from the root. Eventually the leaves will be simple enough that we can construct structures for them directly, rather than using functors. At this point you can apply the functors to the structures, bottom-up, until you produce a structure representing the final program. The functor application phase is similar to program linking in a more conventional language. This analogy can be carried further, as functors can be separately compiled and loaded in some ML systems. There is nothing stopping you mixing these two styles of development. Intermediate goals can be identified and solved in a top-down fashion, and the resulting structures can then be used as the basis of a bottom-up solution of the original goal.

Top-down development of functors

When developing structures, you will have discovered that error messages for a large block of faulty code may be un-informative. A useful strategy is to develop the code interactively at the top level, and then to package it as a structure once it is correct. When developing a functor we cannot use this strategy as our code depends on the formal parameters.

PolyML provides a function `PolyML.ifunctor` to assist in the interactive development of functors. Here is an example of its use

```
> PolyML.ifunctor "C" "CompileSig";
val it = () : unit
> open C;
type Expn   type Action   val code = <undefined> : Expn -> Action list
>
```

The function takes two string arguments, the first will be used as the name of a *dummy structure* matching the signature given by the second. Opening this dummy structure gives a collection of dummy types and values as specified by the signature. Using these, you can develop and type-check the functor code interactively. Since the dummy values haven't been implemented, you can't run it.

To develop a functor with parameters including structures and sharing constraints, just declare a signature corresponding to the parameter specification. For example, to develop the functor `TOPLEVEL` we could have declared a signature `ArgumentSig`

```
signature ArgumentSig = sig
    structure Machine: MachineSig
          and Compile: CompileSig
          and Environment: EnvironmentSig
          and Expn : ExpnSig
 sharing type Compile.Action = Machine.Action
     and type Environment.Environment = Machine.Environment
     and type Compile.Expn = Expn.Expn
end;
```

and then used `ifunctor` as follows

```
> PolyML.ifunctor "Args" "ArgumentSig";
val it = () : unit
> open Args;
structure Environment : EnvironmentSig
structure Expn : ExpnSig
structure Machine : MachineSig
structure Compile : CompileSig
```