15–150: Principles of Functional Programming

Modules

Michael Erdmann*
Spring 2020

1 Topics

- Using modules to design large programs
- Using modules to encapsulate common idioms
- Signatures and structures
- Information hiding and abstract types

A signature describes an interface. It can include types, values, and exceptions.

A structure describes an implementation. SML determines types for all of the components of a structure, and checks that they are consistent with a given signature.

Components of a structure may be hidden (if they are not in the given signature) or visible. To implement an *abstract data type* one keeps the type implementation hidden, and only makes visible the operations that users need to build and manipulate values of that type. (The writeup for the next lecture will include further discussion of this idea.)

2 Background

SML has a module system that helps when designing large programs. With good modular design, you can

- divide your program up into smaller, more easily manageable, chunks called modules (or *structures*);
- for each chunk, specify an *interface* (or *signature*) that limits the way it interacts with the rest of the program.

Modularity can bring practical benefits:

- separate development – modules can be implemented independently

*Adapted with small changes from a document by Stephen Brookes.

- clients have limited access, which may prevent misuse of data
- easy maintenance – we can recompile one module without disrupting others, as long as we obey the interface constraints.

One can also use modules to group together related types and functions, and to encapsulate a commonly occurring pattern, such as a type equipped with certain operations. A good example of this is an *abstract data type*. Using modular design one can ensure that users of an abstract data type are only allowed to build values that are guaranteed to obey some desired properties, such as being a binary search tree. One may thus ensure that users never break the invariants associated with an implementation of the abstract data type. In turn, that facilitates design of efficient and correct code.

3 Main Ideas

A signature is an interface specification that lists some types, functions, and values. For example,

```
signature ARITH =
 sig
     type integer   (* abstract *)
     val rep : int -> integer
     val display : integer -> string
     val add : integer * integer -> integer
     val mult : integer * integer -> integer
 end
```

is an interface that includes

- a type named `integer`
- a function value named `rep`, of type `int -> integer`
- a function value named `display`, of type `integer -> string`
- function values named `add` and `mult`, each of type `integer * integer -> integer`.

Just introducing this signature by itself doesn't actually cause the creation or availability of any such types or values. (The SML REPL will just parrot back to us the signature definition.) To generate data we need to *implement* the signature, by building a *structure* that fills in the missing details. There are many different ways to do this, as we will see. Here is one, which we will refer to as the "standard" implementation, because it implements the type `integer` as the SML type `int`.

Before continuing, note that we included in the signature `ARITH` a type called `integer` for representing integers (in ways we will discuss), a function `rep` to create a representation for an integer from a value of type `int`, operations called `add` and `mult` for combining integer representations, and a function `display` that can be used to generate a string from an integer representation. To keep things clear, we use the term "integer representation" for a value of type `integer`, and "integer" for a value of type `int`.

```
structure Ints =
struct
    type integer = int
    fun rep (n:int):integer = n
    fun display (n:integer):string = Int.toString n
    val add:integer * integer -> integer = (op +)
    val mult:integer * integer -> integer = (op * )
 end
```

(In the last line, observe the space between `"*"` and `")"` to avoid confusion with comments.)

If we enter this into the SML REPL we get the response

```
structure Ints :
  sig
    type integer = int
    val rep : int -> integer
    val display : integer -> string
    val add : integer * integer -> integer
    val mult : integer * integer -> integer
  end
```

and again this looks suggestively like a typing statement: the structure `Ints` has the signature reported above. In fact this is *almost* the same signature as the one we called `ARITH`, and SML discovered this signature automatically, just as it figures out most general types for expressions. The only difference is that here we see that the type `integer` is known to be `int` and this is reported in the signature above.

It would have been just as acceptable to write

```
structure Ints =
struct
    type integer = int
    fun rep (n:int):integer = n
    fun display (n:int):string = Int.toString n
    val add:int * int -> int = (op +)
    val mult:int * int -> int = (op * )
 end
```

because the SML type inference engine works equally well.

We can indicate our intention to use this structure as an implementation of the `ARITH` signature, by writing

```
structure Ints : ARITH =
struct
    type integer = int
    fun rep (n:int):integer = n
    fun display (n:integer):string = Int.toString n
    val add:integer * integer -> integer = (op +)
    val mult:integer * integer -> integer = (op * )
 end
```

From this example you may see that signatures can play a similar role for structures as types do for values. This time the SML REPL will respond

```
structure Ints : ARITH
```

and we must go back and look at the signature to see that SML is telling us that `Ints` makes visible the following:

```
     type integer
     val rep : int -> integer
     val display : integer -> string
     val add : integer * integer -> integer
     val mult : integer * integer -> integer
```

but *not* the fact that `integer` is implemented as the type `int`. The SML results confirm our feeling that `Ints` really does implement `ARITH`, because inside the body of the `Ints` structure are declarations of exactly the things required, with types consistent with those in the signature.

Having made this structure definition, we can use the data defined inside, but because they appear inside the structure body we need to use "qualified names". For example, `Ints.add` is a name we can use to call the `add` function defined inside `Ints`.

We've already seen some uses of this kind of qualified name. The SML implementation contains several built-in structures with standard signatures – the SML Basis Library. Among these are structures such as `String`, and the signature for `String` includes

```
   compare : string * string -> order
```

Similarly there is a structure called `Int`, whose signature includes

```
   compare : int * int -> order
```

To disambiguate between these two functions we call them

```
   String.compare : string * string -> order
   Int.compare : int * int -> order
```

(We chose to name our structure `Ints`, to avoid clashing with `Int`.)

We could have ascribed to a different signature that makes fewer things visible to users of the structure. For example, the signature

```
signature ARITH2 =
sig
   type integer
   val add : integer * integer -> integer
   val mult : integer * integer -> integer
end
```

doesn't include `rep` or `display`.

If we now define `Ints2` as on the next page, then we are only allowed to use `Ints2.add`, `Ints2.mult`, and the type `Ints2.integer`, but `Ints2.rep` and `Ints2.display` would be disallowed. They are invisible to a user outside the structure `Ints2`, since the signature `ARITH2` does not mention `rep` or `display`.

```
structure Ints2 : ARITH2 =
struct
    type integer = int
    fun rep (n:int):integer = n
    fun display (n:integer):string = Int.toString n
    val add:integer * integer -> integer = (op +)
    val mult:integer * integer -> integer = (op * )
 end
```

The same restrictions hold if we define

```
structure Ints2 : ARITH2 = Ints
```

Side note: Observe how we can define a structure by binding a name (`Ints2`) to an existing structure (`Ints`) and constrain its use to a given signature (`ARITH2`).

Decimal digit representation of integers

Now let's look at another way to implement `ARITH`: representing "integer" values as lists of decimal digits (in reverse order, with least significant digit first; this order makes digitwise arithmetic operations easy). We'll define a structure `Dec`, and give it the signature `ARITH`. Inside this structure we'll include some local functions and local type declarations, which we use inside the structure to help with the code implementation, but which (being locally scoped and not included in the signature `ARITH`) are not available to users of the `Dec` structure. This illustrates the usefulness of signatures as a way of *hiding information* that we don't want to be seen. We can easily prevent users from having access to helper functions that are needed inside the structure, simply by omitting them from the signature that we "ascribe" to the structure. In the example above we ascribed the signature `ARITH` to the structure named `Int`.

```
structure Dec : ARITH =
 struct
   type digit = int   (* use only digits 0,1,2,3,4,5,6,7,8,9 *)
   type integer = digit list

   fun rep 0 = [ ]  |  rep n = (n mod 10) :: rep (n div 10)

   (* carry : digit * integer -> integer *)
   fun carry (0, ps) = ps
    |  carry (c, [ ]) = [c]
    |  carry (c, p::ps) = ((p+c) mod 10) :: carry ((p+c) div 10, ps)

   fun add ([ ], qs) = qs
    |  add (ps, [ ]) = ps
    |  add (p::ps, q::qs) =
          ((p+q) mod 10) :: carry ((p+q) div 10, add(ps,qs))
```

```
   (* times : digit -> integer -> integer *)
   fun times 0 qs = [ ]
    |  times k [ ] = [ ]
    |  times k (q::qs) =
         ((k * q) mod 10) :: carry ((k * q) div 10, times k qs)

   fun mult ([ ], _) = [ ]
    |  mult (_, [ ]) = [ ]
    |  mult (p::ps, qs) = add (times p qs, 0 :: mult (ps, qs))

   fun display [ ] = "0"
     | display L = foldl (fn (d, s) => Int.toString d ^ s) "" L
 end
```

Notice that the structure `Dec` does indeed conform to the signature `ARITH`: it does define

- a type named `integer` (NOTE: This type is now implemented as `digit list` not as `int`)
- a function value named `rep`, of type `int -> integer`
- a function value named `display`, of type `integer -> string`
- function values named `add` and `mult`, each of type `integer * integer -> integer`.

The functions `carry` and `times`, and the type `digit`, are local to the structure, not part of the signature, so they are *not* visible externally. This means, for instance, that a user can refer to `Dec.add`, but `Dec.carry` makes no sense.

Examples:

```
   Dec.rep 123 ==> [3,2,1]
   Dec.rep 0 ==> [ ]
   Dec.rep 000 ==> [ ]
   Dec.rep (12+13) ==> [5,2]
   Dec.add([2,1], [3,1]) ==> [5,2]
   Dec.display(Dec.add([2,1], [3,1])) ==> "25"
```

Every value of type `Dec.integer` built from `Dec.rep`, `Dec.add`, `Dec.mult` is a list of decimal digits. Explain why.

To establish the "correctness" of this implementation, we introduce the following helper functions:

```
(* inv : int list -> bool *)
fun inv [ ] = true
  | inv (d::L) = 0 <= d andalso d <= 9 andalso inv L

(* eval : int list -> int
   For all non-negative integers n, eval(rep n) ==> n *)
fun eval [ ] = 0
  | eval (d::L) = d + 10 * eval(L)
```

The purpose of these functions is to help us make a sensible specification of what it means for this implementation to be "correct".

First, one can establish the following facts (e.g., by induction):

- For all `L:int list`, `inv(L)` $\cong$ `true` iff `L` is a list of decimal digits.
- For all non-negative integers `n`, `rep(n)` evaluates to a list `L` such that `inv(L)` $\cong$ `true`.
- For all `L:int list` such that `inv(L)` $\cong$ `true`, i.e., for all lists `L` of decimal digits, `eval L` is a non-negative integer. We call this *the integer represented by* `L`.
- For all non-negative integers `n`, `rep(n)` is a list of decimal digits such that `eval(rep(n)) ==> n`.

Some examples:

```
Dec.rep 1230 ==> [0,3,2,1]

eval [0,3,2,1] ==> 0 + 10 * (eval [3,2,1])
               ==> 0 + 10 * (3 + 10 * (2 + 10 * (1 + 10 * eval [ ])))
               ==> 0 + 10 * (3 + 10 * (2 + 10 * (1 + 10 * 0)))
               ==> 0 + 10 * (3 + 10 * (2 + 10 * (1 + 0)))
               ==> 1230

eval [0,3,2,1,0] ==> 1230

Dec.display(Dec.mult(Dec.rep 10, Dec.rep 20)) ==> "200"
```

We have introduced some tools (`eval` and `inv`) to help us talk accurately about what it means for a value of type `Dec.integer` to be a list of decimal digits, and for such a value to "represent" a given integer `n`. We can also use these tools to define "correctness" for the operations `Dec.add` and `Dec.mult`:

- For all values `L,R:int list`, if `inv(L)` $\cong$ `true` and `inv(R)` $\cong$ `true`, then `Dec.add(L, R)` evaluates to a list `A` such that `inv(A)` $\cong$ `true`, and `eval(A)` $\cong$ `eval(L) + eval(R)`.
- For all values `L,R:int list`, if `inv(L)` $\cong$ `true` and `inv(R)` $\cong$ `true`, then `Dec.mult(L, R)` evaluates to a list `A` such that `inv(A)` $\cong$ `true`, and `eval(A)` $\cong$ `eval(L) * eval(R)`.

To prove these results about `Dec.add` and `Dec.mult` we'll need lemmas about the behavior of `carry` and `times`. What lemmas? We need the following, which one may prove by induction: (here `carry` and `times` refer to the function implementations within structure `Dec`).

- For all values `L:int list` and `c:int`, if `inv(L)` $\cong$ `true` then `inv(carry(c, L))` $\cong$ `true` and `eval(carry(c, L))` $\cong$ `c + eval(L)`.
- For all values `L:int list` and `c:int`, if `inv(L)` $\cong$ `true` then `inv(times(c, L))` $\cong$ `true` and `eval(times(c, L))` $\cong$ `c * eval(L)`.

Exercise: prove these lemmas, and use them to prove the above properties of `Dec.add` and `Dec.mult`. You may need the following fact: for all `n:int`,

`n` $\cong$ `10 * (n div 10) + (n mod 10)`.

We can use the Dec structure to perform arithmetic calculations on integer representations that would, if done directly using the built-in type int, encounter overflow problems. Here is an example: computing the factorial of an integer (e.g., 100) whose factorial is too large to be an allowed value of type int.

```
(* fact : int -> Dec.integer *)
fun fact n =
    if n=0 then Dec.rep 1 else Dec.mult (Dec.rep n, fact (n-1))
```

Note the type: fact takes an SML integer and returns a list of decimal digits.
For all non-negative n, eval(fact n) represents the factorial of n.

```
List.rev(fact 100) ==>
[9,3,3,2,6,2,1,5,4,4,3,9,4,4,1,5,2,6,8,1,6,9,9,2,3,8,8,5,6,2,6,6,7,0,0,
 4,9,0,7,1,5,9,6,8,2,6,4,3,8,1,6,2,1,4,6,8,5,9,2,9,6,3,8,9,5,2,1,7,5,9,
 9,9,9,3,2,2,9,9,1,5,6,0,8,9,4,1,4,6,3,9,7,6,1,5,6,5,1,8,2,8,6,2,5,3,6,
 9,7,9,2,0,8,2,7,2,2,3,7,5,8,2,5,1,1,8,5,2,1,0,9,1,6,8,6,4,0,0,0,0,0,0,
 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

(Note: we reverse the list, because when we write out a number in decimal notation the least significant digits go on the right, not the left.)

Thus, the factorial of 100 is

```
93326215443944152681699238856266700490715968264
38162146859296389521759999322991560894146397615
6518286253697920827223758251185210916864000000000000000000000000
```

```
 Dec.display(fact 100) ==>
 "93326215443944152681699238856266700490715968264
38162146859296389521759999322991560894146397615
6518286253697920827223758251185210916864000000000000000000000000"
```

Binary representation

Actually, there is nothing special about decimal: we could use binary just as well. The following structure is a binary digit implementation of `ARITH`:

```
structure Bin : ARITH =
 struct
   type digit = int   (* use only 0 and 1 *)
   type integer = digit list
   fun rep 0 = [ ]  |   rep n = (n mod 2) :: rep (n div 2)

   (* carry : digit * integer -> integer *)
   fun carry (0, ps) = ps
    |  carry (c, [ ]) = [c]
    |  carry (c, p::ps) = ((p+c) mod 2) :: carry ((p+c) div 2, ps)

   fun add ([ ], qs) = qs
    |  add (ps, [ ]) = ps
    |  add (p::ps, q::qs) =
           ((p+q) mod 2) :: carry ((p+q) div 2, add (ps,qs))

   (* times : digit -> integer -> integer *)
   fun times 0 qs = [ ]
    |  times k [ ] = [ ]
    |  times k (q::qs) =
             ((k * q) mod 2) :: carry ((k * q) div 2, times k qs)

   fun mult ([ ], _) = [ ]
    |  mult (_, [ ]) = [ ]
    |  mult (p::ps, qs) = add (times p qs, 0 :: mult (ps,qs))

   fun display [ ] = "0"
     | display L = foldl (fn (d, s) => Int.toString d ^ s) "" L
 end
```

We could again compute factorials, now in binary representation:

```
(* fact : int -> Bin.integer *)
fun fact n = if n=0 then Bin.rep 1 else Bin.mult (Bin.rep n, fact (n-1))
```

```
 Bin.display(fact 100) ==>
  "110110011000010010110010011101100001110010101110111000010010000000110100
   101010010100011010101010010111011110110100100000011010001011011101001011
   0011011101111110011010011100001110101100100001101101011011001010010100001
   1101000110010000111001101111110001000000111001000101110100010101010111000
   011001100101010010100001000001100011011101100101100111011011100101110110
   100101110111010001011000000101110101000100111001101011100011000011010000
   000000000000000000000000000000000000000000000000000000000000000000000000
   000000000000000000000"
```

4 Binary Search Trees

Now we revisit binary search trees of integers, to reinforce the benefits of information hiding. Recall that a binary search tree is a binary tree that is sorted, as defined in the lecture on tree sorting. For simplicity we do not include deletion as an allowed operation, but it is not difficult to augment the signature and structures to incorporate deletion.

```
signature TREE =
sig
   datatype tree = Leaf | Node of tree * int * tree   (* concrete *)
   val empty : tree
   val insert : int * tree -> tree
   val trav : tree -> int list
end
```

The type `tree` and the constructors `Leaf` and `Node` will be visible to users. So will be the functions `insert` and `trav`.

```
structure Bst : TREE =
struct
   datatype tree = Leaf | Node of tree * int * tree

   val empty = Leaf

   fun insert (x, Leaf) = Node(Leaf, x, Leaf)
     | insert(x, Node(T1, y, T2)) =
         (case Int.compare(x,y) of
             LESS => Node(insert(x, T1), y, T2)
           | EQUAL => Node(T1, y, T2)  (* we do not keep duplicates *)
           | GREATER => Node(T1, y, insert(x, T2)))

   fun trav Leaf = [ ]
     | trav (Node(T1, x, T2)) = trav T1 @ (x :: trav T2)
end
```

To a user outside the structure `Bst`, the names `Bst.tree`, `Bst.empty`, `Bst.insert`, and `Bst.trav` are in scope. Perhaps less obviously, so are `Bst.Leaf` and `Bst.Node`.

Every tree built from `Bst.empty` using the `Bst.insert` operation is guaranteed to be a valid binary search tree, because `Bst.Leaf` is a binary search tree, and `Bst.insert(x, T)` is a binary search tree whenever `T` is a binary search tree.

However, since `Bst.Node` and `Bst.Leaf` are in scope outside the structure, a user could construct a value of type `Bst.tree` that is not a valid binary search tree. Example:

```
                                                                2
   Bst.Node(Bst.Leaf, 2, Bst.Node(Bst.Leaf, 1, Bst.Leaf))     i.e.,      \
                                                                  1
```

We can revise our signature and structure design to prevent this. See next page.

```
signature TREE =
sig
   type tree  (* abstract *)
   val empty : tree
   val insert : int * tree -> tree
   val trav : tree -> int list
end
```

For any structure `Bst` ascribing to this revised signature, `Bst.empty`, `Bst.insert`, and `Bst.trav` are in scope for external users, as is the type `Bst.tree`. But, that's all! The specific `tree` implementation is not available for use since it is not even specified in the signature. One says that the tree implementation is *abstract*.

```
structure Bst : TREE =
struct
   datatype tree = Leaf | Node of tree * int * tree

   (* The datatype constructors Leaf and Node are not in TREE, *)
   (* so will not be in scope outside of this structure body.  *)

   val empty = Leaf

   fun insert (x, Leaf) = Node(Leaf, x, Leaf)
     | insert(x, Node(T1, y, T2)) =
         (case Int.compare(x,y) of
              LESS => Node(insert(x, T1), y, T2)
            | EQUAL => Node(T1, y, T2)  (* we do not keep duplicates *)
            | GREATER => Node(T1, y, insert(x, T2)))

   fun trav Leaf = [ ]
     | trav (Node(T1, x, T2)) = trav T1 @ (x :: trav T2)
end
```

The structure implementation above looks identical to what we had before, but now the signature is different. `Bst.Node` and `Bst.Leaf` are *not* in scope for an external user. Thus the signature prevents a user from inventing bogus tree values.

A user still can *see* that trees are constructed via `Node` and `Leaf` but cannot pattern-match on or use those constructors directly. For instance, in the REPL:

```
- val T = Bst.insert(2, Bst.empty);
val T = Node (Leaf,2,Leaf) : Bst.tree
```

However, an attempt to use `Bst.Leaf` or `Bst.Node` is now an error:

```
- Bst.Leaf;
stdIn:94.1-94.5 Error: unbound variable or constructor: Leaf in path Bst.Leaf
```

The writeup for next lecture will discuss *opaque ascription* as an even stronger method for hiding data, in which a user would not even be able to see that trees are constructed using `Node` and `Leaf`.

We could have achieved something similar as follows:

```
structure Bst : TREE =
struct
   datatype foo = Leaf | Node of foo * int * foo

   type tree = foo

   (* Again, the datatype constructors Leaf and Node are not in TREE, *)
   (* so will not be in scope outside of this structure body.         *)

   val empty = Leaf

   fun insert (x, Leaf) = Node(Leaf, x, Leaf)
     | insert(x, Node(T1, y, T2)) =
         (case Int.compare(x,y) of
             LESS => Node(insert(x, T1), y, T2)
           | EQUAL => Node(T1, y, T2)  (* we do not keep duplicates *)
           | GREATER => Node(T1, y, insert(x, T2)))

   fun trav Leaf = [ ]
     | trav (Node(T1, x, T2)) = trav T1 @ (x :: trav T2)
end
```

Every value of type `Bst.tree` built from `Bst.empty` and `Bst.insert` is guaranteed to be a valid binary search tree. These are the ONLY ways users can build values of type `Bst.tree`.

5 Functors

The discussion of decimal digitwise arithmetic, and the very similar code dealing with binary arithmetic, is an example of a rather common occurrence: there may be an entire family of closely related ways to do something (here, arithmetic on integers), parameterized by some easily identifiable feature (here, the choice of *base* for the digits: 10 or 2, but any positive integer would work just as well).

Soon we will see that the SML module system offers an elegant way to take advantage of recurring common patterns in *code design*. We do not have to write an entire family of structures, one for each choice of base. Instead, we will be able to write a "function" that operates on structures, called a *functor*.