15–150: Principles of Functional Programming

Functors

Michael Erdmann*

Spring 2024

1 Topic: Using Functors to Build Structures

- putting structures together to build new structures
- functors as parameterized structure definitions
- transparent and opaque use of signatures

2 Motivation: Similar Code Patterns

Previously, we discussed:

```
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
```

as an interface that includes

- a type named `integer`

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

- 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`.

And we defined an implementation based on decimal digits (in reverse order, with least significant digit first, to make digitwise arithmetic easy).

We defined a structure `Dec`, and gave it the signature `ARITH`, as in:

```
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, _) = [ ]
    |  times (_, [ ]) = [ ]
    |  times (p, q::qs) =
         ((p * q) mod 10) :: carry ((p * q) div 10, times (p, 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
```

Examples:

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

Binary Representation

Recall as well that we also implemented a structure based on binary:

```
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, _) = [ ]
    |  times (_, [ ]) = [ ]
    |  times (p, q::qs) =
            ((p * q) mod 2) :: carry ((p * q) div 2, times (p, 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
```

The lesson is that there is a common design pattern at work. Almost the same structure, with easily identifiable modifications, works for decimal and for binary. And, in fact, we could do the same for base 5, base 42, or any integer base greater than 1.[1]

It is inconvenient to implement a different structure for each choice of digit base, re-writing all the code each time. This motivates the introduction of *functors*.

3 Using a Functor

What we want is a way to define a *parameterized* module, with a parameter that stands for a choice of integer to be used as *base*. For the decimal implementation we would choose 10, for binary choose 2, and so on. The solution is to use functors, a feature of the SML module system that makes it easy to define such a parametric module. Rather than give the general syntax and semantics of functors in full detail, we will work out an example and use it to introduce the main ideas.

First, recall the `ARITH` signature:

```
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
```

Now we'll introduce a new signature `BASE` which contains just a single integer named `base`. Every structure that implements this signature will define a specific integer value named `base`.

```
signature BASE =
 sig
   val base : int   (* parameter *)
 end
```

[1] It is possible to work with base 1 (which would be "unary" notation), but the code we are about to develop that works for bases greater than 1 doesn't behave well if we choose base 1.

Now we can define a *functor* – we'll call it `Digits` – that, when applied to a structure `B` with signature `BASE`, returns a structure with signature `ARITH` that implements integers as lists of digits in the range $0, \ldots,$ `B.base` $- 1$. The SML syntax for a functor definition is similar to a function definition, but with structures as arguments and signatures instead of types. Inside the body of the functor we introduce some local names to allow us to abbreviate: in particular, we use `b` to stand for the value `B.base`, where `B` is the argument structure supplied to the functor. The SML code in this functor body is obtained from the `Dec` code by using `b` strategically instead of 10. Equally well, it can be obtained from the `Binary` code by replacing 2 by `b`. We'll see shortly that we can actually recover two structures behaviorally equivalent to `Dec` and `Binary` by *calling* this functor on suitably chosen `BASE` structure arguments.

```
functor Digits(B : BASE) : ARITH =
struct
  val b = B.base     (* REQUIRE:  b > 1 *)
  type digit = int   (* use only digits 0 through b-1 *)
  type integer = digit list

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

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

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

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

In response to this functor declaration, the SML REPL says

```
functor Digits(B : sig val base : int end) :
              sig
                type integer
                val rep : int -> integer
                val display : integer -> string
                val add : integer * integer -> integer
                val mult : integer * integer -> integer
              end
```

We can use the functor `Digits` to obtain structures parameterized by different bases, as in these examples:

```
structure Dec = Digits(struct val base = 10 end)

structure Bin = Digits(struct val base = 2 end)
```

(Here we have used "anonymous" structure expressions such as

```
struct val base = 10 end
```

as the arguments in our functor applications.)

We may now further use these structures, for instance in function definitions (recall our earlier factorial implementations using `Dec` and `Bin`):

```
fun decfact (0:int) : Dec.integer = Dec.rep 1
  | decfact n = Dec.mult(Dec.rep n, decfact(n-1))

Dec.display(decfact 100)

fun binfact (0:int) : Bin.integer = Bin.rep 1
  | binfact n = Bin.mult(Bin.rep n, binfact(n-1))

Bin.display(binfact 100)
```

Check the results!

We could also have built these structures in other ways, e.g., by declaring structures with names `Base10` and so on, to use in the functor applications:

```
structure Base10 = struct val base = 10 end
structure Base2 = struct val base = 2 end

structure Dec = Digits(Base10)
structure Binary = Digits(Base2)
```

And as usual we can use these structures, for instance by using qualified names:

```
Dec.rep 42 ==> [2,4] : Dec.integer
Bin.rep 42 ==> [0,1,0,1,0,1] : Bin.integer
```

4 Ascribing Signatures

We just defined two structures, `Dec` and `Binary` ascribing to the same signature `ARITH`. In each of these structures, the type `integer` is defined to be `int list`. For better or worse, the identity of these types is visible and exploitable. Here is an example to show that SML "knows" that these types are identical.

```
- Bin.rep 42;
val it = [0,1,0,1,0,1] : Bin.integer
- it : Dec.integer;
val it = [0,1,0,1,0,1] : Dec.integer
- it : int list;
val it = [0,1,0,1,0,1] : int list
```

SML lets us check that a value of the reported type `Bin.integer` can also be used at the type `int list`. In fact, SML will even allow us to do crazy things like mix up the operations from different digitwise implementations, because they all (visibly) share the same underlying type `int list`:

```
 - Dec.add(Bin.rep 42, Bin.rep 42);
val it = [0,2,0,2,0,2] : Dec.integer
```

or (even worse)

```
- Bin.add(Dec.rep 42, Dec.rep 42);
val it = [0,0,1,2] : Bin.integer
```

That is worse, because the list isn't even a list of binary digits, and even if we interpret it as a decimal digit list it gives the sum of 42 and 42 as 2100, not 84.

One really should prevent users of one digital arithmetic structure from accidentally calling operations from another. The SML module system offers some ways to do this.

Using `datatype` Constructors to Hide Abstract Types

As we saw in a previous lecture, one can prevent components of a structure from being visible to users by omitting them from the signature ascribed to by the structure. Still, suppose we want the fact that `ARITH` has an internal `integer` type to be known, so we want it in the signature. However, we also want to prevent a client from manipulating the representation of `integer` directly.

One way to achieve that, as we have seen, is to modify the functor definition to make the type `integer` into a `datatype` whose constructors are not directly accessible (because they are not in the `ARITH` signature). This is a generally applicable method, and in this particular case we can get by with a `datatype` with a single constructor (as below). Unfortunately, doing this will also require us to modify the code for all of the functions inside the functor body, since those functions will now be operating on a different type from before.

```
functor Digits(B : BASE) : ARITH =
struct
  val b = B.base     (* REQUIRE:  b > 1 *)
  type digit = int   (* use only digits 0 through b-1 *)
  datatype digits = D of digit list
  type integer = digits

  fun rep 0 = D [ ]
    | rep n = let
                 val (D L) = rep (n div b)
              in
                 D((n mod b) :: L)
              end

   . . . adjusted similarly . . .

 end
```

Observe how we had to re-do the `rep` function to perform pattern matching on the result of the recursive call (to extract a digit list), then wrap the cons'd list inside the constructor so that the result type of `rep` is correct (we need the `digits` datatype, not just a `digit list` type).

If we do this also in the other components of the structure, then redefine the structures `Dec` and `Binary` exactly as before, with the same signature ascriptions, the code fragment

```
Bin.add(Dec.rep 42, Dec.rep 42)
```

results in a type error, as desired. Problem solved, albeit at the cost of code redesign.

Using Opaque Ascription to Hide Abstract Types

Another way to solve the problem, also supported by the SML module system, is to ascribe to the signature `ARITH` *opaquely* and otherwise *leave the original functor definition the same*. An opaque ascription is called opaque because the implementation details of the types defined in the structure (`integer` here) are hidden; the fact that `integer` and `int list` are the same is not made visible outside the structure. An ascription like `Dec:ARITH` that uses a colon (`:`) is called a *transparent* ascription. As the name suggests, when we ascribe to a signature *transparently*, implementation details of types defined in the structure are made visible, provided the types are mentioned in the signature. (In both transparent and opaque cases, things not mentioned in the signature stay invisible outside the structure body.)

To make an *opaque* ascription we use colon-greater (`:>`), as in

```
functor Digits(B : BASE) :> ARITH =
struct
   ... as before ...
end
```

and then

```
structure Binary = Digits(struct val base = 2 end)
structure Dec = Digits(struct val base = 10 end)
```

SML again responds with

```
structure Binary : ARITH
structure Dec : ARITH
```

but we will see different results when we use the opaquely ascribed structures from what we saw earlier:

```
- Bin.rep 42;
val it = - : Bin.integer   (* not val it = [0,1,0,1,0,1] : Bin.integer *)

- Dec.rep 42;
val it = - : Dec.integer   (*  not val it = [2,4] : Dec.integer *)

- Dec.display (Dec.rep 42);
val it = "42" : string     (* just to confirm that the hidden value is correct *)

- Bin.display(Bin.rep 42);
val it = "101010" : string (* similarly *)

- Bin.add(Dec.rep 42, Dec.rep 42);
stdIn:107.1-107.35 Error: operator and operand don't agree [tycon mismatch]
  operator domain: Bin.integer * Bin.integer
  operand:         Dec.integer * Dec.integer
  in expression:
    Bin.add (Dec.rep 42,Dec.rep 42)
```

If we ascribe a signature opaquely to a structure, be aware that (as above) the evaluation results and types reported by the SML REPL may be rather uninformative. In the example shown above, we used `Dec.display` to generate a result of type `string`, in order to confirm that the value of type `Dec.integer` computed by `Dec.rep 42` must have been the intended list of decimal digits. If you use opaque ascription make sure you have allowed enough operations to be available to help with debugging your code. And make sure you haven't hidden too much.

- When you use transparent ascription it is your responsibility to hide what you want hidden.
- When you use opaque ascription it is your responsibility to reveal what needs to be revealed.

5 Functor Syntax

Alternative Declaration Format

The SML syntax for functor declarations also allows us to insert the keyword `structure` before the parameter name, as in:

```
functor Digits(structure B : BASE) : ARITH =
struct
  val b = B.base     (* REQUIRE:  b > 1 *)
  type digit = int   (* use only digits 0 through b-1 *)
  type integer = digit list
    ... as before ...
 end
```

If we enter this declaration the SML REPL responds slightly differently than before:

```
functor Digits(<param> : sig structure B : <sig>  end) :
               sig
                 type integer
                 val rep : int -> integer
                 val display : integer -> string
                 val add : integer * integer -> integer
                 val mult : integer * integer -> integer
               end
```

(only the parameter information looks different). Although this format (explicitly saying "`structure`" in the functor header) is more verbose, it is sometimes convenient, for instance when writing a functor that takes several arguments. The more streamlined syntax does not allow multiple arguments.

Functors with Several Arguments

So far we have seen one example of a functor, and if we pursue the analogy with *functions*, our example involved a "first-order" functor from structures to structures. And our functor had a single parameter. To define a functor with several arguments, we must use the verbose format (say `structure` before each argument name) and separate the arguments by a space (or new line) *not by a comma*.

Here is an example. We can define a "type class" of *monoids*, where a monoid is a type equipped with a binary operation (called *addition*) and a

designated special value (called *zero*). In math, monoids also have certain algebraic properties, e.g., *zero* behaves like an additive zero and addition is associative. But here we aren't concerned with imposing algebraic laws, since they cannot be enforced merely by type-checking or the module system. So here is a signature `MONOID` that encapsulates the notion of monoids:

```
signature MONOID =
  sig
     type t   (* parameter *)
     val add : t * t -> t
     val zero : t
  end
```

Here are some implementations:

```
structure AdditiveIntegers : MONOID =
  struct
     type t = int
     val add = (op + )
     val zero = 0
  end

structure MultiplicativeIntegers : MONOID =
  struct
    type t = int
    val add = (op * )
    val zero = 1
  end
```

Mathematicians know that the *Cartesian product of two monoids is also a monoid*. Correspondingly, there ought to be a simple way to combine two structures with the signature `MONOID` and obtain their "product", another structure with signature `MONOID`. This can be encapsulated as a functor with two argument structures:

```
functor Prod (structure A:MONOID
              structure B:MONOID) : MONOID =
struct
   type t = A.t * B.t
   val zero = (A.zero, B.zero)
   fun add((a1, b1), (a2, b2)) =
        (A.add(a1, a2), B.add(b1, b2))
end
```

Notice that we tagged the two arguments with the keyword `structure`, and did *not* separate them with a comma but with a newline (a space would be fine too).

The SML REPL responds with

```
functor Prod(<param>: sig
                        structure A : <sig>
                        structure B : <sig>
                      end) :
            sig
              type t
              val add : t * t -> t
              val zero : t
            end
```

To use this functor we apply it, also using a verbose format for the parameters, e.g.,

```
- structure S = Prod(structure A=AdditiveIntegers
                     structure B=MultiplicativeIntegers)
```

to which the SML REPL says

```
structure S : MONOID
```

6 Dictionaries Revisited

Recall that we defined a parameterized type of trees with data at the nodes. If we use data of the form (k, v), where k is a "key" value belonging to some type equipped with a comparison function, we can use trees to represent "dictionaries" and design insertion and lookup functions that use key comparison for efficiency.

To do this in a modular manner, we begin with two signatures:

- `ORDERED`, an interface providing a type named `t` and a "comparison" function of type `t * t -> order`.
- `DICT`, an interface with a parameterized type `'a dict`, a structure `Key` with signature `ORDERED`, a value `empty` and functions `insert` and `lookup`. We also include a function `trav` that's useful for debugging.

```
signature ORDERED =
 sig
   type t   (* parameter *)
   val compare : t * t -> order
 end

signature DICT =
  sig
    structure Key : ORDERED   (* parameter *)
    type 'a dict              (* abstract  *)
    val empty : 'a dict
    val insert : 'a dict -> Key.t * 'a -> 'a dict
    val lookup : 'a dict -> Key.t -> 'a option
    val trav : 'a dict -> (Key.t * 'a) list
  end
```

Some notes:

- It is fine to have a signature containing a structure. When we implement this signature, we will build a structure containing another structure.
- We decided to curry all the function types in this `DICT` signature
- The details of this signature `DICT` are slightly different from those presented in the lecture code. All the core ideas are the same.

Here is an example implementation of `ORDERED`:

```
structure Integers : ORDERED =
 struct
    type t = int
    val compare = Int.compare
  end
```

Dictionaries as Association Lists

One way to implement `DICT` is to use integers as keys, and lists of integer-value pairs as dictionaries:

```
structure Assoc : DICT =
 struct
   structure Key = Integers
   type 'a dict = (Key.t * 'a) list
   val empty = [ ]
   fun insert D (k, v) = ...
   fun lookup D k = ...
 end
```

Notice how the use of integers as keys is largely secondary. Within the structure `Assoc` we can refer to `Key.t` and `Key.compare`, never needing to know that these refer to integers.

Dictionaries as Binary Search Trees

Alternatively, we could implement dictionaries using binary search trees (instead of lists) of integer-value pairs. Again, the precise key type as integer is irrelevant. Again, we can simply refer to `Key.t` and `Key.compare`. Any key type ascribing to `ORDERED` will work. So, let's introduce a functor:

```
functor BSTDict(Key : ORDERED) : DICT =
 struct
   structure Key : ORDERED = Key
   datatype 'a tree = Leaf | Node of 'a tree * (Key.t * 'a) * 'a tree
   type 'a dict = 'a tree

   (* empty : 'a dict *)
   val empty = Leaf
```

```
  (* lookup : 'a dict -> Key.t -> 'a option *)
  fun lookup Leaf k = NONE
    | lookup (Node (D1, (k', v'), D2)) k  =
         (case Key.compare (k,k') of
             EQUAL    =>  SOME v'
           | LESS     =>  lookup D1 k
           | GREATER  =>  lookup D2 k)

  (* insert : 'a dict -> key * 'a -> 'a dict *)
  fun insert Leaf (k, v) = Node(Leaf, (k, v), Leaf)
   |  insert (Node(l, (k', v'), r)) (k, v) =
        (case Key.compare(k, k') of
            EQUAL   => Node(l, (k,v), r)
          | LESS    => Node(insert l (k,v), (k',v'), r)
          | GREATER => Node(l, (k',v'), insert r (k,v)))

    fun trav Leaf = [ ]
     |  trav (Node(l, (k,v), r)) = (trav l) @ (k,v) :: (trav r)
end
```

Using the `BSTDict` Functor

```
structure S = BSTDict(Integers)

fun build [ ] = S.empty
 |  build (x::L) = S.insert (build L) (x, Int.toString x)

fun flatten Ls = foldr (op @) [ ] Ls

fun splits [ ] = [([ ], [ ])]
 |  splits (x::r) =
       ([ ], x::r) :: (map (fn (l1,l2) => (x::l1,l2)) (splits r))

fun perms [ ] = [[ ]]
 |  perms (a::l) =
    let
      val t = flatten (map splits (perms l))
    in
      map (fn (l1, l2) => l1@[a]@l2) t
    end
```

```
fun forall p [ ] = true
  | forall p (x::L) = (p x) andalso forall p L

val true = forall bst (map build (perms [1,2,3,4,5,6,7,8]))
```

Here `bst` is a function for checking if a value of tree type is actually a binary search tree:

```
fun sorted [ ] = true
 |  sorted ((k,_)::L) =
      forall (fn (k',_) => S.Key.compare(k,k') <> GREATER) L

fun bst T = sorted (S.trav T)
```

The above code fragment generates the list of all dictionary values expressible as `build L`, where `L` ranges over the list of permutations of `[1,2,3,4,5,6,7,8]`. Then it checks that each of these trees is indeed a binary search tree.

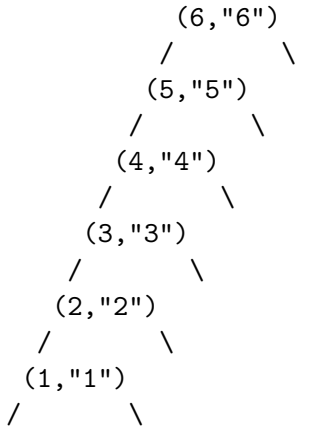
```
- forall bst (map build (perms [1,2,3,4,5,6,7,8]));
val it = true : bool
```

So this looks like our testing is producing satisfactory results: if we insert 8 different things (or at least, the 8 integers given above) into a dictionary, in any order, we always get a binary search tree.

Of course, the resulting trees need not be balanced. Indeed, consider:

```
- val T = build [1,2,3,4,5,6];
val T = Node (Node (Node #,(#,#),Leaf),(6,"6"),Leaf) : string tree
```

In pictorial form, this tree `T` looks like



As expected, that is not a balanced tree.

(We will discuss a Red/Black tree implementation of balanced trees in the next lecture.)

7 Advanced Features

SML does not have *higher-order* modules in its language definition. Some SML implementations do, for instance SML/NJ. SML/NJ allows a programmer to include functors in signatures, and functors in structures. And one can define signatures for functors, functors that return functors, and functors that take functors as arguments. In fact, one can even *curry* a functor of several arguments. Moreover, just as well-typed expressions have a most general type, well-typed structures have a most general signature.

We do not explore these concepts in this course.