# 15-150 Fall 2013
# Lecture 16

Stephen Brookes

Tuesday, 22 October

## 1 Topics

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

## 2 Background

ML 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
- 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.

You 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*, and using modular design we can ensure that users of an a.d.t. are only allowed to build values that are guaranteed to obey some desired properties, such as being a binary search tree.

## 3 Main ideas

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

```
signature ARITH =
 sig
     type integer
     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 named `rep`, of type `int -> integer`
- a function named `display`, of type `integer -> string`
- functions named `add` and `mult`, 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 ML 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 ML type `int`.

Before continuing, note that we included in the signature `ARITH` a type intended to serve to represent integers, a function `rep` that can be used 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 extract 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
```

(Note that we put a space between the `*` and the `)`!)

If we enter this into the ML 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 ML 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 ML type inference engine works equally well. Also the order in which the function names are listed is not important.

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 ML REPL will respond

```
structure Ints : ARITH
```

and we must go back and look at the signature to see that ML 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 the signature alone does not tell us the fact that `integer` is implemented as the type `int`. 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`. And the following code fragment will type-check:

```
    Ints.add(Ints.rep 21, Ints.rep 21);
```

its type is `Ints.integer` and its value is `42`.

We've already seen some other uses of this kind of qualified name. The ML implementation contains several built-in structures with standard signatures – the ML 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 a different signature that makes fewer things visible to users of the structure. For example, the signature

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

doesn't include `add` or `display`. If we defined

```
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;
```

we would only be allowed to use

```
Ints2.rep
Ints2.mult
```

and the type `Ints2.integer`, but `Ints2.add` and `Ints2.display` would be disallowed.

The same problems would occur if we defined

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

Note that as here we can define a structure like this, 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 (* uses 0 through 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 L = foldl (fn (d, s) => Int.toString d ^ s) "" L
 end;
```

carry and times are “helper” functions, used only inside this structure. They aren’t listed in the signature, so we included their types in comments to help others understand our intentions.

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

- a type named integer (implemented 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 this structure, not part of the signature.

In the rest of this section we will discuss the behavior of these functions in more detail. To avoid always having to use qualified names like `Dec.add`, let's assume that we've entered the following text into the ML REPL loop:

```
open Dec;
```

That brings all of the signature items into scope so we can refer to them simply as `add` and so on. (However, the local items like `carry` are not in scope. We can't ask ML to evaluate `carry (0, [ ])`.)

Examples:

```
rep 123 = [3,2,1]
rep 0 = [ ]
rep 000 = [ ]
rep (12+13) = [5,2]
add([2,1], [3,1]) = [5,2]
```

Every value of type `Dec.integer` built from `rep`, `add`, `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                     *)
fun eval [ ] = 0
|      eval (d::L) = d + 10 * eval(L);

(* For all non-negative integers n, eval(rep n) = n *)
```

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

First, the following basic facts are easy to establish (by induction, as usual):

- For all `L:int list`, `inv(L) = 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) = true`.
- For all `L:int list` such that `inv(L) = 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`, there is a list `L` of decimal digits such that `rep n = L`. In fact there are many such lists, differing only in the number of "leading zeros".
- For all non-negative integers `n`, `rep(n)` is a list of decimal digits such that `eval(rep(n)) = n`.

Some examples:

```
rep 1230 = [0,3,2,1]]
eval [0,3,2,1] = 0 + 10*(eval [3,2,1])
               = 10 * (3 + 10 * (2 + 10 * (1 + 10 * eval [ ])))
               = 10 * (3 + 10 * (2 + 10 * (1 + 10 * 0)))
               = 10 * (3 + 10 * (2 + 10 * (1 + 0)))
               = 123

eval [0,3,2,1,0] = 123

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

So now we've 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 `add` and `mult`:

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

To prove these results about `add` and `mult` we'll need lemmas about the behavior of `carry` and `times`. What lemmas? We need the following, which are easy to prove by induction on list length:

- For all `L:int list` and `c:int`, if `inv(L) = true` and $0 \leq$ `c` $\leq 9$, then `inv(carry(c, L)) = true` and `eval(carry(c, L)) = c + eval L`.
- For all `L:int list` and `c:int`, if `inv(L) = true` and $0 \leq$ `c` $\leq 9$, then `inv(times(c, L)) = true` and `eval(times(c, L)) = c * eval L`.

Here is a quick proof sketch for the lemma about `carry`.

> For all `L:int list` and `c:int` such that `inv(L) = true` and $0 \leq$ `c` $\leq 9$, `inv(carry(c, L)) = true` and `eval(carry(c, L)) = c + eval L`.

Proof: by induction on the length of `L`.

- Base case: for `L = [ ]`. Let $0 \leq$ `c` $\leq 9$. Note that `inv [ ] = true`, and `carry(c, [ ]) = [c]`. Since $0 \leq$ `c` $\leq 9$, `inv` [c] = `true`. And `eval [c] = c = c + 0 = c + eval [ ]`.
- Inductive step: Let `L = d::ds`, `inv(d::ds) = true`, and $0 \leq$ `c` $\leq 9$.

  Assume as induction hypothesis that for all lists `R` shorter than `d::ds` and all c′, if `inv(R) = true` and $0 \leq$ c′ $\leq 9$, then `inv(carry(c', R)) = true` and `eval(carry(c', R)) = c' + eval R`.

  By assumption, $0 \leq$ `d` $\leq 9$ and `inv(ds) = true`.

  We have, by the function definitions:

  ```
  carry(c, d::ds) = ((c+d) mod 10) :: carry((c+d) div 10, ds)
  ```

  Let `c'` be the value of `(c+d) div 10`. Then $0 \leq$ c′ $\leq 1$. (Why?)

  Since `ds` is a shorter list than `d::ds` and `inv(ds) = true`, it follows from the induction hypothesis that `inv(carry(c', ds)) = true` and `eval(carry(c', ds)) = c' + eval ds`.

  By definition of `eval` we then have

```
    eval (carry(c, d::ds))
    = eval ((c+d) mod 10) :: carry(c', ds))
    = (c+d) mod 10 + 10 * eval(carry(c', ds))
    = (c+d) mod 10 + 10 * (c' + eval ds)
    = (c+d) mod 10 + 10 * c' + 10 * eval ds
    = (c+d) mod 10 + 10 * ((c+d) div 10) + 10 * eval ds
    = (c+d) + 10 * eval ds
    = c + (d + 10 * eval ds)
    = c + eval(d::ds),
```

as required.

Exercise: prove the lemma about `times`, which can be done similarly to the lemma about `carry`. Then use the two lemmas to prove the above properties of `add` and `mult`. Note the vital importance in these proofs of the fundamental property that relates `div` and `mod`: for all `n:int`,

```
n = 10 * (n div 10) + (n mod 10).
```

So we've verified the correctness of this implementation of arithmetic: integer values are represented faithfully and the addition and multiplication operations on represented integers produce results faithful to the abstract operations `+` and `*`.

Moreover we can use this `Dec` structure to do arithmetical 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 -> integer *)
fun fact n =
    if n=0 then rep 1 else  mult (rep n, fact (n-1));
```

Note the type: `fact` takes an ML 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
651828625369792082722375825118521091686400000000000000000000000
```

```
 display(fact 100) =
 "93326215443944152681699238856266700490715968264
38162146859296389521759999322991560894146397615
651828625369792082722375825118521091686400000000000000000000000"
```

## Binary representation

Actually there's nothing special about decimal: we could use binary just as well. The following structure is the binary digit version.

```
structure Bin : ARITH =
 struct
   type digit = int (* uses 0, 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 L =  foldl (fn (d, s) => Int.toString d ^ s) "" L
 end;
```