15-150 Fall 2013
Lecture 17

Stephen Brookes

Thursday, 24 October

1 Topics

Continuing from the previous lecture. . .

- Using modules to design large programs
- Using modules to encapsulate common idioms
- Signatures and structures: interfaces and implementations
- Information hiding and abstract types
- Using functors to build structures

A signature describes an interface. It can include types, values (of given types), and exceptions (although we didn't include any exceptions in our examples).

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

Structures may also be used in signatures, in which case to implement you will need a structure containing a structure.

Components of a structure may be hidden (if they are not in the given signature) or visible. A type component (a type defined in a structure) is an "abstract type" if it is not in the signature. To implement an *abstract data type* keep the type implementation hidden, and only make visible the operations that users need to build and manipulate values of that type.

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 use modules to group together related types and functions, and to encapsulate a commonly occurring pattern, such as a type equipped with certain operations; this notion (describing a whole family of related instances) is similar to a typeclass. And another situation for which the module system helps greatly is with the design and implementation of an *abstract data type*: 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 (usually describable as a *representation invariant*), such as being a binary search tree. We can thus ensure that users never break the invariant, and we can design the code that implements the operations on data to take advantage of the invariant to improve efficiency and achieve correctness.

TODAY:

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

3 Main ideas

Last time we introduced

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

and explained that this signature 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`.

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 0 through 9 *)
   type integer = digit list

   (* rep : int -> integer *)
   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)
   (* add : integer * integer -> integer *)
   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))

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

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

Note that `carry` and `times` are "helper" functions, used only inside this structure (and not available for use outside because they aren't mentioned in the signature). We included their types in comments to help document our intentions, since these types don't appear in the signature.

Examples:

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

We showed last time that every value of type `Dec.integer` built from `Dec.rep`, `Dec.add` and `Dec.mult` is a list of decimal digits. To prove this "correctness" property for this implementation, we introduced two helper functions: a *representation invariant* (stating that a list of integers is a list of decimal digits) and an *abstraction function* (describing what abstract value – of type `int` –

is represented by a concrete value – of type `Dec.integer`, which is `int list` – that satisfies the representation invariant).

```
(* 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)
```

We stated that the following results are provable:

- For all `L:int list`, `inv(L) = true` iff `L` is a list of decimal digits.
- For all non-negative integers `n`, `Dec.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 `Dec.rep n = L`. In fact there are many such lists, differing only in the number of trailing zeros (which correspond to leading zeros when we convert to a string using `Dec.display`). Actually for all `n` ≥ 0, `Dec.rep n` has no redundant zeros.
- For all non-negative integers `n`, `Dec.rep(n)` is a list of decimal digits such that `eval(Dec.rep(n)) = n`.

Some examples:

```
Dec.rep 1230 = [0,3,2,1]]
eval [0,3,2,1] = 123
Dec.rep 01230 = [0,3,2,1]
eval [0,3,2,1,0,0,0,0] = 123
Dec.display(Dec.mult(Dec.rep 10, Dec.rep 20)) = "200"
```

We also stated the following results, which mention the `carry` and `times` functions from inside the `Dec` structure body: [1]

- For all values `L:int list` and `c:int`, if `inv(L) = true` and $0 \leq$ `c` ≤ 9 then `inv(Dec.carry(c, L)) = true` and `eval(Dec.carry(c, L)) = c + eval L`.
- For all values `L:int list` and `c:int`, if `inv(L) = true` and $0 \leq$ `c` ≤ 9 then `inv(Dec.times(c, L)) = true` and `eval(Dec.times(c, L)) = c * eval L`.
- For all values `L,R:int list`, if `inv(L) = true` and `inv(R) = true`, then `Dec.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 `Dec.mult(L, R)` evaluates to a list `A` such that `inv(A) = true` and `eval(A) = eval L * eval R`.

A key ingredient in establishing these results was the fundamental property linking `div` and `mod`: for all non-negative values `n:int`,

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

We also showed that the following code fragment, which uses the data listed in `ARITH` as implemented by `Dec`, computes the factorial of 100 (as a list of decimal digits):

```
open Dec;

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

display (fact 100);
```

[1]Even though the Ml scope rules make these functions invisible outside the function body, we allow ourselves the luxury of mentioning them here because we're doing math, not running a program!

Binary representation

But there's nothing special about decimal: we could use binary, as in:

```
structure Binary : ARITH =
 struct
   type digit = int
   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;
```

This structure also conforms to the ascribed signature `ARITH`.

We said that any piece of code using the data prescribed in this signature should be executable on *any* structure that implements this signature. So an obvious question is: What happens if we try the factorial calculation again?

```
open Binary;

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

display(fact 100);
```

What happens?

```
val it =
  "11011001100001001011001001110110000111001010111011100001001000000001101#"
  : string
```

Is this OK? Yes, its fine. We've done the same calculation as before and reported the answer using binary digits rather than decimal. If we were to figure out what integer is represented by this binary list, we would get the same mathematical answer as the result we found using decimal digits. (Unfortunately, we can't do this check in ML because of overflow.)

And we can easily re-do the entire correctness analysis in binary rather than decimal, along the following lines.

Introduce helper functions

```
(* inv2 : int list -> bool *)
fun inv2 [ ] = true
|   inv2 (d::L) = 0 <= d andalso d <= 1 andalso inv2 L;

(* eval2 : int list -> int                    *)
fun eval2 [ ] = 0
|      eval2 (d::L) = d + 2 * eval(L);
```

and show that:

- For all `L:int list`, `inv2(L) = true` iff `L` is a list of binary digits.
- For all non-negative integers `n`, `Binary.rep(n)` evaluates to a list `L` such that `inv2(L) = true`.
- For all `L:int list` such that `inv2(L) = true`, i.e. for all lists `L` of binary digits, `eval2 L` is a non-negative integer. We call this *the integer represented by* `L` *in binary.*

- For all non-negative integers `n`, there is a list `L` of binary digits such that `Binary.rep n = L`. In fact there are many such lists, differing only in the number of "leading zeros" (which show up in `Binary.rep n` as zeros at the tail end of the list).
- For all non-negative integers `n`, `Binary.rep(n)` is a list of binary digits such that `eval(Binary.rep(n)) = n`.
- For all values `L:int list` and `c:int`, if `inv2 L = true` and $0 \leq$ `c` ≤ 1 then `inv2(Binary.carry(c, L)) = true` and `eval2(Binary.carry(c, L)) = c + eval2 L`.
- For all values `L:int list` and `c:int`, if `inv2(L) = true` and $0 leq$`c` ≤ 1 then `inv2(Binary.times(c, L)) = true` and `eval2(Binary.times(c, L)) = c * eval2 L`.
- For all values `L,R:int list`, if `inv2(L) = true` and `inv2(R) = true`, then `Binary.add(L, R)` evaluates to a list `A` such that `inv2(A) = true` and `eval2(A) = eval2 L + eval2 R`.
- For all values `L,R:int list`, if `inv2(L) = true` and `inv2(R) = true`, then `Binary.mult(L, R)` evaluates to a list `A` such that `inv2(A) = true` and `eval2(A) = eval2 L * eval2 R`.

Here the relevant mathematical property is: for all non-negative values `n:int`,

```
n = 2 * (n div 2) + (n mod 2).
```

We deliberately avoid giving the proof details, because they can easily be obtained by systematically replacing 10 by 2 in the details for the decimal case!

The lesson is that there is a common design pattern at work here. Almost the same structure, with easily identifiable modifications, works for decimal as for binary. And in fact we could do the same for base 5, base 42, any base integer greater than 1.[2]

It's clearly a bad idea to have to re-define a different structure for each choice of digit base, re-writing all the code.

This motivates the introduction of *functors.*

[2]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. In fact the correctness analysis breaks down when the base is 1 – figure out where!

4 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 ML 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
     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
 end;
```

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 ML 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 ML 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;
  type digit = int (* use 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 ML 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 to obtain structures, as in these examples:

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

structure Binary = Digits(struct val base = 2 end);

structure Unary = Digits(struct val base = 1 end);
```

(Here we've used "anonymous" structure expressions such as

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

as the arguments in our functor applications.)

Now we could use these structures, as in:

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

fun binfact(n:int) : Binary.integer =
   if n=0 then Binary.rep 1 else Binary.mult(Binary.rep n, binfact(n-1));

print(Dec.display(decfact 100));

print(Binary.display(binfact 100));
```

Check the results!

Also check what happens if we use `Unary`. Explain why things go wrong!

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 Dec = Digits(Base10);
```

And as usual we can use these structures either by opening them or by using qualified names. For example

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

What happens if we try `Unary.rep 42`?

5 Ascribing signatures

As we've seen, you can prevent components of a structure from being visible to users by omitting them from the signature ascribed to the structure.

We've shown you how to use ascriptions like `Dec:ARITH`.

And we've defined three structures, `Dec`, `Binary`, and `Unary`, each with the same signature `ARITH`. In each of these structures the type `integer` is defined to be `int list`. Here's an example to show that ML "knows" that these types are identical and the module system allows you to exploit that fact: in other words, the identity of thee types is visible to users:

```
- Binary.rep 42;
val it = [0,1,0,1,0,1] : Binary.integer
- it : Unary.integer;
val it = [0,1,0,1,0,1] : Unary.integer
- it : int list;
val it = [0,0,1,2] : int list
```

So ML lets us check that a value of the reported type `Binary.integer` can also be used at the type `int list`. In fact, ML will even allow you 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(Binary.rep 42, Binary.rep 42);
val it = [0,2,0,2,0,2] : Dec.integer
```

or (even worse)

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

This 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.

You really should prevent users of one digital arithmetic structure from accidentally calling operations from another. The ML module system offers some ways to do this. One way is to modify the functor definition to make the type `integer` into a *datatype* whose constructors are not visible (because they're 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 you to modify the code for all of the functions inside the functor body, since they will now be operating on a different type from before.

```
functor Digits(B : BASE) : ARITH =
struct
  val b = B.base;
  type digit = int (* use 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;
```

See 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 (the `digits` datatype, not the `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

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

no longer passes the type system. Problem solved, albeit at the cost of a code redesign.

Another way to solve the problem, also supported by the ML module system, is to ascribe the signature `ARITH` *opaquely* and *leave the original functor definition the same.* An opaque ascription is so-called because when we ascribe a signature opaquely to a structure 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. Similarly, and again as the name suggests, when we ascribe 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

```
structure Binary :> ARITH = Digits(struct val base = 2 end);
structure Dec :> ARITH = Digits(struct val base = 10 end);
```

Incidentally, despite its resemblance, `:>` is not a smiley or emoticon `:-)`.

ML 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.

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

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

```
- Binary.add(Dec.rep 42, Dec.rep 42);
stdIn:107.1-107.35 Error: operator and operand don't agree [tycon mismatch]
  operator domain: Binary.integer * Binary.integer
  operand:         Dec.integer * Dec.integer
  in expression:
    Binary.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 buy the ML 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've allowed enough operations to be available to help with debugging your code! And make sure you haven't made too many things and details to be hidden. When you use transparent ascription it's your responsibility to hide what you want hidden; when you use opaque ascription it's your responsibility to reveal what needs to be revealed.

A note on functor syntax

The ML syntax for functor declarations also allows you to insert the keyword `structure` before the parameter name in the above example, as in:

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

If we enter this declaration the ML REPL responds slightly differently, with In response to this functor declaration, the ML REPL says

```
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 may be preferred, mainly because it generalizes nicely to cases where you want to write a functor of several arguments whereas the more streamlined syntax doesn't.

6 Functors with several arguments

So far we've 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 you 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
     val add : t * t -> t
     val zero : t
  end
```

Here are some obvious 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.

The ML 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=Additive structure B=Multiplicative);
```

to which the ML REPL says

```
structure S : MONOID
```

7 Large integers, revisited yet again

We used factorials as an example where we encounter large integer values that cause overflow with the conventional type `int`.

We did this for pedagogic reasons, but we could have avoided lots of the fuss by taking advantage of another ML library module.

Standard ML has a signature `INTEGER`, and several library structures that implement it. Each of these implementations provides a type of signed integers of either a fixed or arbitrary precision, together with arithmetic and conversion operations. For the fixed precision implementations, most arithmetic operations raise the exception `Overflow` when their result is not representable.

Many ML implementations also contain a structure named `IntInf` that also implements the INTEGER interface but with some extras. In addition to the INTEGER operations, it provides some operations useful for programming with arbitrarily large integers. Operations in `IntInf` that return a value of type `IntInf.int` don't raise the `Overflow` exception. Note that, as it extends the `INTEGER` interface, `IntInf` defines a type `int`.

We could have defined

```
fun fact (n:IntInf.int) : IntInf.int =
      if n=0 then 1 else n * fact(n - 1);
```

The ML REPL's response is:

```
val fact = fn : IntInf.int -> IntInf.int
```

If we then enter

```
- fact 100;
```

we get the result

```
val it =
  93326215443944152681699238856266700490715968264381621468592963895217599#
  : IntInf.int
```

This agrees with the calculation done using the `Dec` implementation!

8 Advanced features

Although we won't explore them this semester, the ML module system also allows *higher-order* modules. You can include functors in signatures, and functors in structures. And you can define signatures for functors, functors that return functors, and functors that take functors as arguments. In fact, you 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.

9 Reading

You can find an alternative discussion of the ML module system in

`http://www.cs.cmu.edu/~rwh/introsml/modules/sigstruct.htm`