CSE341, Fall 2011, Lecture 12 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

We start by showing how ML modules can be used to separate bindings into different *namespaces*. We then build on this material to cover the much more interesting and important topic of using modules to hide bindings and types.

Modules for Namespace Management

To learn the basics of ML, pattern-matching, and functional programming, we have written small programs that are just a sequence of bindings. For larger programs, we want to use more structure in organizing our code. In ML, we can use *structures* to define *modules* that contain a collection of bindings. At its simplest, you can write `structure Name = struct` ***`bindings`*** `end` where `Name` is the name of your structure (you can pick anything; capitalization is a convention) and ***`bindings`*** is any list of bindings, containing values, functions, exceptions, datatypes, and type synonyms. Inside the structure you can use earlier bindings just like we have been doing "at top-level" (i.e., outside of any module). Outside the structure, you refer to a binding b in `Name` by writing `Name.`b. We have already been using this notation to use functions like `List.foldl`; now you know how to define your own structures.

Though we will not do so in our examples, you can nest structures inside other structures to create a tree-shaped hierarchy. But in ML, modules are *not* expressions: you cannot define them inside of functions, store them in tuples, pass them as arguments, etc.

Used like this, structures are providing just *namespace management*, a way to avoid different bindings in different parts of the program from shadowing each other. Java packages provide similar support for namespace management. Namespace management is very useful, but not very interesting. Much more interesting is giving structures *signatures*, which are types for modules. They let us provide strict *interfaces* that code outside the module must obey. ML has several ways to do this with subtly different syntax and semantics; we just show one way to write down an explicit signature for a module. Here is an example signature definition and structure definition that says the structure `MyMathLib` must have the signature `MATHLIB`:

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x =
    if x=0
    then 1
    else x * fact (x - 1)

val half_pi = Math.pi / 2.0

fun doubler y = y + y
end
```

Because of the `:> MATHLIB`, the structure `MyMathLib` will typecheck only if it actually provides everything the signature `MATHLIB` claims it does and with the right types. Signatures can also contain datatype, exception, and type bindings. Because we check the signature when we compile `MyMathLib`, we can use this information when we check any code that uses `MyMathLib`. In other words, we can just check clients *assuming* that the signature is correct.

`open`: an overused by convenient shortcut

If in some scope you are using many bindings from another structure, it can be inconvenient to write `SomeLongStructureName.foo` many times. Of course, you can use a val-binding to avoid this, e.g., `val foo = SomeLongStructureName.foo`, but this technique is ineffective if we are using many different bindings from the structure (we would need a new variable for each) or for using constructor names from the structure in patterns. So ML allows you to write `open SomeLongStructureName`, which provides "direct" access (you can just write `foo`) to any bindings in the module that are mentioned in the module's signature. The scope of an open is the rest of the enclosing structure (or the rest of the program at top-level).

A common use of `open` is to write succinct testing code for a module outside the module itself. Other uses of `open` are often frowned upon because it may introduce unexpected shadowing, especially since different modules may reuse binding names. For example, a list module and a tree module may both have functions named `map`.

Hiding things

Before learning how to use ML modules to hide implementation details from clients, let's remember that separating an interface from an implementation is probably the most important strategy for building correct, robust, reusable programs. Moreover, we can already use functions to hide implementations in various ways. For example, all 3 of these functions double their argument, and clients (i.e., callers) would have no way to tell if we replaced one of the functions with a different one:

```
fun double1 x = x + x
fun double2 x = x * 2
val y = 2
fun double3 x = x * y
```

Another feature we use for hiding implementations is defining functions locally inside other functions. We can later change, remove, or add locally defined functions knowing the old versions were not relied on by any other code. From an engineering perspective, this is a crucial separation of concerns. I can work on improving the implementation of a function and know that I am not breaking any clients. Conversely, nothing clients can do can break how the functions above work.

But what if you wanted to have two top-level functions that code in other modules could use and have both of them use the same hidden functions? There are ways to do this (e.g., create a record of functions), but it would be convenient to have some top-level functions that were "private" to the module. In ML, there is no "private" keyword like in other languages. Instead, you use signatures that simply mention less: anything not explicitly in a signature cannot be used from the outside. For example, if we change the signature above to:

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
end
```

then client code cannot call `MyMathLib.doubler`. The binding simply is not in scope, so no use of it will

type-check. In general, the idea is that we can implement the module however we like and only bindings that are explicitly listed in the signature can be called directly by clients.

Introducing our extended example

The rest of this lecture uses a small module that implements rational numbers. While a real library would provide many more features, ours will just support creating fractions, adding two of them together, and converting them to strings. Our library intends to (1) prevent denominators of zero and (2) keep fractions in reduced form (3/2 instead of 9/6 and 4 instead of 4/1). While negative fractions are fine, internally the library never has a negative denominator ($-3/2$ instead of $3/-2$ and $3/2$ instead of $-3/-2$). The structure below implements all these ideas, using the helper function `reduce`, which itself uses `gcd`, for reducing a fraction.

```
structure Rational1 =
struct
(* Invariant 1: all denominators > 0
   Invariant 2: rationals kept in reduced form *)
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

(* gcd and reduce help keep fractions reduced,
   but clients need not know about them *)
(* they _assume_ their inputs are not negative *)
  fun gcd (x,y) =
      if x=y
      then x
      else if x < y
      then gcd(x,y-x)
      else gcd(y,x)

   fun reduce r =
       case r of
           Whole _ => r
         | Frac(x,y) =>
           if x=0
           then Whole 0
           else let val d = gcd(abs x,y) in (* using invariant 1 *)
                    if d=y
                    then Whole(x div d)
                    else Frac(x div d, y div d)
                end

(* when making a frac, we ban zero denominators *)
   fun make_frac (x,y) =
       if y = 0
       then raise BadFrac
       else if y < 0
       then reduce(Frac(~x,~y))
       else reduce(Frac(x,y))

(* using math properties, both invariants hold of the result
   assuming they hold of the arguments *)
```

```
   fun add (r1,r2) =
       case (r1,r2) of
           (Whole(i),Whole(j))   => Whole(i+j)
         | (Whole(i),Frac(j,k))  => Frac(j+k*i,k)
         | (Frac(j,k),Whole(i))  => Frac(j+k*i,k)
         | (Frac(a,b),Frac(c,d)) => reduce (Frac(a*d + b*c, b*d))

(* given invariant, prints in reduced form *)
   fun toString r =
       case r of
           Whole i => Int.toString i
         | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)

end
```

Since `reduce` and `gcd` are helper functions that we do not want clients to rely on or misuse, one natural signature would be as follows:

```
signature RATIONAL_A =
sig
datatype rational = Frac of int * int | Whole of int
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

To use this signature to hide `gcd` and `reduce`, we can just change the first line of the structure definition to `structure Rational1 :> RATIONAL_A`.

Properties and Invariants of our Module

While this approach ensures clients do not call `gcd` or `reduce` directly (since they "do not exist" outside the module), this is *not* enough to ensure the bindings in the module are used correctly. What "correct" means for a module depends on the specification for the module (not the definition of the ML language), so let's be more specific about some of the desired *properties* of our library for rational numbers and the *invariants* that our code seeks to maintain to provide these properties:

- Property: `toString` always returns a string representation in reduced form
- Property: No code goes into an infinite loop
- Property: No code divides by zero
- Property: There are no fractions with denominators of 0
- Invariant: All denominators are greater than 0 (a negative fraction has a negative numerator)
- Invariant: All values of type `rational` returned by any function are already in reduced form

The properties are *externally visible*; they are what we promise clients. The invariants are *internal*; they are facts about the implementation that ensure the properties. The code above maintains the invariants and relies on them in certain places, notably:

- `gcd` will violate the properties if called with an arguments ≤ 0, but since we know denominators are > 0, `reduce` can pass denominators to `gcd` without concern.
- `toString` and most cases of `add` do not need to call `reduce` because they can assume their arguments are already in reduced form.
- `add` uses the property of mathematics that the product of two positive numbers is positive, so we know a non-positive denominator is not introduced.

A Better Signature: Using an *Abstract Type*

Unfortunately, under signature `RATIONAL_A`, clients must still be trusted not to break the properties and invariants! Because the signature exposed the definition of the datatype binding, the ML type system will not prevent clients from using the constructors `Frac` and `Whole` directly, bypassing all our work to establish and preserve the invariants. Clients could make "bad" fractions like `Rational.Frac(1,0)`, `Rational.Frac(3,~2)`, or `Rational.Frac(9,6)`, any of which could then end up causing `gcd` or `toString` to misbehave according to our specification. While we may have *intended* for the client only to use `make_frac`, `add`, and `toString`, our signature allows more.

A natural reaction would be to hide the datatype binding by removing the line
`datatype rational = Frac of int * int | Whole of int`. While this is the right intuition, the resulting signature makes no sense and would be rejected: it repeatedly mentions a type `rational` that is not known to exist. What we want to say instead is that there is a type `rational` *but clients cannot know anything about what the type is other than it exists.* In a signature, we can do just that with an *abstract type*, as this signature shows:

```
signature RATIONAL_B =
sig
type rational (* type now abstract *)
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

(Of course, we also have to change the first line of the structure definition to use this signature instead. That is always true, so we will stop mentioning it.)

This new feature, which makes sense only in signatures, is exactly what we want. It lets our module define operations over a type without revealing the implementation of that type. The syntax is just to give a type binding without a definition. The implementation of the module is unchanged; we are simply changing how much information clients have.

Now, how can clients make rationals? Well, the first one will have to be made with `make_frac`. After that, more rationals can be made with `make_frac` or `add`. There is *no other way*, so thanks to the way we wrote `make_frac` and `add`, all rationals will always be in reduced form with a positive denominator.

What `RATIONAL_B` took away from clients compared to `RATIONAL_A` is the constructors `Frac` and `Whole`. So clients cannot create rationals directly and they cannot pattern-match on rationals. They have no idea how they are represented internally. They don't even know `rational` is implemented as a datatype.

Abstract types are a Really Big Deal in programming.

A Cute Twist: Expose the `Whole` function

By making the `rational` type abstract, we took away from clients the `Frac` and `Whole` constructors. While this was crucial for ensuring clients could not create a fraction that was not reduced or had a non-positive denominator, only the `Frac` constructor was problematic. Since allowing clients to create whole numbers directly cannot violate our specification, we could add a function like:

```
fun make_whole x = Whole x
```

to our structure and `val make_whole : int -> rational` to our signature. But this is unnecessary function wrapping; a shorter implementation would be:

```
val make_whole = Whole
```

and of course clients cannot tell which implementation of `make_whole` we are using. But why create a new binding `make_whole` that is just the same thing as `Whole`? Instead, we could just *export the constructor as a function* with this signature and *no changes or additions to our structure*:

```
signature RATIONAL_C =
sig
type rational (* type still abstract *)
exception BadFrac
val Whole : int -> rational (* client knows only that Whole is a function *)
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

This signature tells clients there is a function bound to `Whole` that takes an `int` and produces a `rational`. That is correct: this binding is one of the things the datatype binding in the structure creates. So we are exposing *part* of what the datatype binding provides: that `rational` is a type and that `Whole` is bound to a function. We are still hiding the rest of what the datatype binding provides: the `Frac` constructor and pattern-matching with `Frac` and `Whole`.

Rules for Signature Matching

So far, our discussion of whether a structure "should type-check" given a particular signature has been rather informal. Let us now enumerate more precise rules for what it means for a structure to *match* a signature. (This terminology has nothing to do with pattern-matching.) If a structure does not match a signature assigned to it, then the module does not type-check. A structure `Name` matches a signature `BLAH` if:

- For every val-binding in `BLAH`, `Name` must have a binding with that type *or a more general type* (e.g., the implementation can be polymorphic even if the signature says it is not — see below for an example). This binding could be provided via a val-binding, a fun-binding, or a datatype-binding.
- For every non-abstract type-binding in `BLAH`, `Name` must have the same type binding.
- For every abstract type-binding in `BLAH`, `Name` must have some binding that creates that type (either a datatype binding or a type synonym).

Notice that `Name` can have any additional bindings that are not in the signature.

Equivalent Implementations

Given our property- and invariant-preserving signatures `RATIONAL_B` and `RATIONAL_C`, we know clients cannot rely on any helper functions or the actual representation of rationals as defined in the module. So we could replace the *implementation* with any *equivalent implementation* that had the same properties: as long as any call to the `toString` binding in the module produced the same result, clients could never tell. This is another essential software-development task: improving/changing a library in a way that does not break clients. Knowing clients obey an abstraction boundary, as enforced by ML's signatures, is invaluable.

As a simple example, we could make `gcd` a local function defined inside of `reduce` and know that no client will fail to work since they could not rely on `gcd`'s existence. More interestingly, let's change one of the invariants of our structure. Let's *not* keep rationals in reduced form. Instead, let's just reduce a rational right before we convert it to a string. This simplifies `make_frac` and `add`, while complicating `toString`, which is now the only function that needs `reduce`. Here is the whole structure, which would still match signatures `RATIONAL_A`, `RATIONAL_B`, or `RATIONAL_C`:

```
structure Rational2 :> RATIONAL_A (* or B or C *) =
struct
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

   fun make_frac (x,y) =
       if y = 0
       then raise BadFrac
       else if y < 0
       then Frac(~x,~y)
       else Frac(x,y)

   fun add (r1,r2) =
       case (r1,r2) of
           (Whole(i),Whole(j))   => Whole(i+j)
         | (Whole(i),Frac(j,k))  => Frac(j+k*i,k)
         | (Frac(j,k),Whole(i))  => Frac(j+k*i,k)
         | (Frac(a,b),Frac(c,d)) => Frac(a*d + b*c, b*d)

   fun toString r =
       let fun gcd (x,y) =
               if x=y
               then x
               else if x < y
               then gcd(x,y-x)
               else gcd(y,x)

           fun reduce r =
               case r of
                   Whole _ => r
                 | Frac(x,y) =>
                   if x=0
                   then Whole 0
                   else
                       let val d = gcd(abs x,y) in
                           if d=y
```

```
                        then Whole(x div d)
                        else Frac(x div d, y div d)
                    end
    in
        case reduce r of
            Whole i   => Int.toString i
          | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)
    end
end
```

If we give `Rational1` and `Rational2` the signature `RATIONAL_A`, both will type-check, but clients can still distinguish them. For example, `Rational1.toString(Rational1.Frac(21,3))` produces `"21/3"`, but `Rational2.toString(Rational2.Frac(21,3))` produces `"7"`. But if we give `Rational1` and `Rational2` the signature `RATIONAL_B` or `RATIONAL_C`, then the structures are equivalent for any possible client. This is why it is important to use restrictive signatures like `RATIONAL_B` to begin with: so you can change the structure later without checking all the clients.

While our two structures so far maintain different invariants, they do use the same definition for the type `rational`. This is not necessary with signatures `RATIONAL_B` or `RATIONAL_C`; a different structure having these signatures could implement the type differently. For example, suppose we realize that special-casing whole-numbers internally is more trouble than it is worth. We could instead just use `int*int` and define this structure:

```
structure Rational3 :> RATIONAL_B (* or C *)=
struct
   type rational = int*int
   exception BadFrac

   fun make_frac (x,y) =
       if y = 0
       then raise BadFrac
       else if y < 0
       then (~x,~y)
       else (x,y)

   fun Whole i = (i,1)

   fun add ((a,b),(c,d)) = (a*d + c*b, b*d)

   fun toString (x,y) =
       if x=0
       then "0"
       else
           let fun gcd (x,y) =
                   if x=y
                   then x
                   else if x < y
                   then gcd(x,y-x)
                   else gcd(y,x)
               val d = gcd (abs x,y)
               val num = x div d
               val denom = y div d
```

```
        in
            Int.toString num ^ (if denom=1
                                then ""
                                else "/" ^ (Int.toString denom))
        end
end
```

(This structure takes the `Rational2` approach of having `toString` reduce fractions, but that issue is largely orthogonal from the definition of `rational`.)

Notice that this structure provides everything `RATIONAL_B` requires. The function `make_frac` is interesting in that it takes an `int*int` and return an `int*int`, but clients do not know the actual return type, only the abstract type `rational`. And while giving it an argument type of `rational` in the signature would match, it would make the module useless since clients would not be able to create a value of type `rational`. Nonetheless, clients *cannot* pass just any `int*int` to `add` or `toString`; they must pass something that they know has type `rational`. As with our other structures, that means rationals are created only by `make_frac` and `add`, which enforces all our invariants.

Our structure *does not* match `RATIONAL_A` since it does not provide `rational` as a datatype with constructors `Frac` and `Whole`.

Our structure *does* match signature `RATIONAL_C` because we explicitly added a function `Whole` of the right type. No client can distinguish our "real function" from the previous structures' use of the `Whole` constructor as a function.

The fact that `fun Whole i = (i,1)` matches `val Whole : int -> rational` is interesting. The type of `Whole` in the module is actually polymorphic: `'a -> 'a * int`. ML signature matching allows `'a -> 'a * int` to match `int -> rational` because `'a -> 'a * int` is more general than `int -> int * int` and `int -> rational` is a correct abstraction of `int -> int * int`. Less formally, the fact that `Whole` has a polymorphic type inside the module does not mean the signature has to give it a polymorphic type outside the module. And in fact it cannot while using the abstract type since `Whole` cannot have the type `'a -> int * int` or `'a -> rational`.

Different modules define different types

While we have defined different structures (e.g., `Rational1`, `Rational2`, and `Rational3`) with the same signature (e.g., `RATIONAL_B`), that does *not* mean that the bindings from the different structures can be used with each other. For example, `Rational1.toString(Rational2.make_frac(2,3))` will not type-check, which is a good thing since it would print an unreduced fraction. The *reason* it does not type-check is that `Rational2.rational` and `Rational1.rational` are *different types*. They were not created by the same datatype binding even though they happen to look identical. Moreover, outside the module we do not *know* they look identical. Indeed, `Rational3.toString(Rational2.make_frac(2,3))` really needs not to type-check since `Rational3.toString` expects an `int*int` but `Rational2.make_frac(2,3))` returns a value made out of the `Rational2.Frac` constructor.