CSE 341, Spring 2008, Lecture 11 Summary

Standard Disclaimer: These comments may prove useful, but certainly are not a complete summary of all the important stuff we did in class. They may make little sense if you missed class, but hopefully will help you organize and process what you have learned.

The end of lecture 10 described the basics of structures and signatures for separating different parts of our programs. Today we will use a larger example to discuss the advantages of using signatures that hide information about the structure.

Before discussing ML modules, let's notice that hiding implementation details is one of the most important concepts when developing software. We can already do this with functions. 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
```

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.

We would like to have these advantages for whole structures. The key is using signatures to do one or both of the following:

- Make certain bindings inaccessible
- Make types abstract (reveal that a type exists, but do not reveal its implementation)

These techniques help us replace a module implementation with another one without clients being able to tell. They *also* let us enforce invariants about arguments to our functions since abstract types can ensure that values are only created by functions defined within the module.

To see all this, we will consider an extended example. Here is a small library for positive, rational numbers. A positive rational number is a fraction where the numerator and denominator are both greater than 0. We have functions for creating rationals, adding two rationals, and converting a rational to a string. Of course, a real library would have many more functions, but this will suffice for us:

```
structure PosRat1 =
struct
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

  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) =>
           let val d = gcd(x,y) in
               if d=y
               then Whole(x div d)
               else Frac(x div d, y div d)
```

```
        end

    fun make_frac (x,y) =
        if x <= 0 orelse y <= 0
        then raise BadFrac
        else reduce (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)) => reduce (Frac(a*d + b*c, b*d))

    fun toString r =
        case r of
            Whole i => Int.toString i
          | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)

end
```

The code as written makes several *assumptions*, which should *at least* be documented in comments:

- `gcd` and `reduce` are really local helper functions for putting rationals into reduced form (e.g., `Frac(1,2)` instead of `Frac(2,4)` and `Whole 7` instead of `Frac(21,3)`). We do not want clients calling them since we do not want to be responsible for maintaining their behavior. Moreover, they do not work for negative numbers, so we definitely don't want clients thinking they do.
- `make_frac` will reject non-positive numbers. Therefore, if all rationals are created via `make_frac` and `add`, then all rationals will always be positive. Since `reduce` assumes this, that is an important *invariant*. To ensure it holds, clients should not use the `Whole` and `Frac` constructors directly; they should always call `make_frac`.
- Similarly, we keep all rationals in reduced form. Thanks to this invariant, `toString` will never return `"4/2"` or `"21/3"`, but again a client that creates its own rationals could break this property, e.g., with `PosRat1.toString(Frac(21,3))`.

A signature that simply described all the bindings in this structure would look like this:

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

If we give this signature to the structure (by starting it with `structure PosRat1 :> RATIONAL_ALL`, ML will check that all the types are right, but clients will still be able to do whatever they want with the constructors, exception, and functions we defined.

A more restrictive signature could hide the bindings that we do not want clients to know about. This effectively makes them private. Unlike in Java, we do this just via the signature, we do not have to change the structure body at all. We define:

```
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
```

and change the first line of the structure to `structure PosRat1 :> RATIONAL_A`.

This signature ensures clients do not call `gcd` or `reduce` (or have any idea they exist), so we could change the structure definition in various ways (such as changing their names) and we can feel better about having them fail with negative numbers. But we still have the problem that clients can create bad rationals directly (bypassing `make_frac`). The way to fix this is to remove the `datatype` binding as well! But that does not quite work because our signature mentions the type `rational` and if we remove the `datatype` beinding that type name will make no sense.

So what we want is a way to say that there is a type `rational` but clients cannot know anything about what the type is other than it exists. This is an *abstract type*, and it is a very powerful concept for letting programmers write libraries that have to be used in particular ways. Here is the appropriate signature:

```
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. That is always true, so we will stop mentioning it.)

The syntax is just to give a type binding without a definition. 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.

What `RATIONAL_B` took away from clients compared to `RATIONAL_A` is the constructors `Frac` and `Whole`. So clients cannot be rationals directly and they cannot pattern-match on rationals. They have no idea how they are represented internally.

ML lets us do one more slightly fancy thing if we want. `RATIONAL_B` enforces the invariants we want but it makes it a little unpleasant to make rationals that are whole numbers since clients have to call `make_frac(6,1)`. We could add a function to our structure like this:

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

and a line to our signature like this:

```
val make_whole : int -> rational
```

But the `make_whole` function is an unnecessary wrapper just like `if e then true else false`. The constructor `Whole` is *already* a function that takes an `int` and returns a `rational` just like we want. So instead we can make no change to the structure and instead just add to our signature:

```
val Whole : int -> rational
```

This makes sense because constructors are two things: functions that create values of the datatype and things you can use in patterns. This addition to our signature just exposes one of the two — clients know `Whole` is a function, but they do not know it is a constructor.

We can now consider in general, what it means for a structure `Name` to be a legal implementation of the signature `BLAH`:

- It must define all bindings mentioned in `BLAH` (and it can define more).
- The types of the bindings in `Name` must match those in `BLAH`, but they can be more general (e.g., the implementation can be polymorphic even if the signature says it is not — an example comes later).
- Types can be made abstract.

So far, our more restrictive signatures for `PosRat1` have succeeded in ensuring clients: (1) do not create non-positive rationals, (2) do not create non-reduced rationals, (3) do not pass bad arguments to `gcd` or `reduce`. Now we can consider the other advantage of restrictive signatures: we can change structure implementations and *know* that client behavior cannot change.

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:

```
structure PosRat2 :> RATIONAL_A (*or B or C*) =
struct
  datatype rational = Whole of int | Frac of int*int
  exception BadFrac

   fun make_frac (x,y) =
       if x <= 0 orelse y <= 0
       then raise BadFrac
       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) =>
                   let val d = gcd(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
```

Notice the following:

- `PosRat2` does *not* have signature `RATIONAL_ALL` because it does not define `gcd` or `reduce` (they are local to `toString`).
- While `PosRat2` does have signature `RATIONAL_A`, if we give `PosRat1` and `PosRat2` these signatures, *clients can tell the difference!*. For example, `PosRat1.toString(Frac(21,3)` returns `"21/3"` but `PosRat2.toString(Frac(21,3)` returns `"7"`. So this signature is *not* restrictive enough to ensure changing `PosRat1` to `PosRat2` will not change client behavior.
- `RATIONAL_B` and `RATIONAL_C` *are* restrictive enough to ensure the two structures are totally equivalent for any possible client.

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 PosRat3 :> RATIONAL_B =
struct
   type rational = int*int
   exception BadFrac

   fun make_frac z =
       let val (x,y) = z in
           if x <= 0 orelse y <= 0
           then raise BadFrac
           else z
       end

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

   fun toString (x,y) =
       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 (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 `PosRat2` 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 is the identity function unless it raises an exception, but *clients do not know that* since they cannot tell that `rational` is `int*int`. They *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`, ensuring that all rationals are positive. Our structure does *not* match `RATIONAL_A` since it does not provide `rational` as a datatype with constructors `Frac` and `Whole`.

Also notice that our structure as defined does not have signature `RATIONAL_C` because there is no `Whole` function. But we could easily add one – all we need is a function of type `int->rational` that has the behavior we want:

```
fun Whole i = (i,1)
```

No client can distinguish our "real function" from the previous structures' use of the `Whole` constructor as a function.

Finally, suppose we had written `make_frac` like this:

```
fun make_frac x = x
```

This does not enforce our invariant that forbids non-positive numbers, but it does still match signatures `RATIONAL_B` and `RATIONAL_C`. That is intersting because within the module, `make_frac` has type `'a->'a`, but outside it has type `int*int -> rational`. Our previous version above passed signature matching because internally it had type `int*int -> int*int` and `rational=int*int`. The polymorphic version also matches because we are allowed to *specialize* polymorphic functions when doing signature matching. So the type-checker figures out that to match `int*int -> rational` against `'a->'a` it need to first replace `'a` with `int*int` and then use the definition `rational=int*int`. Less formally, the fact that `make_frac` *could* have a polymorphic type does not mean the singature *has* to give it one.