15-150 Lecture 19:
Signatures, Structures, and Type Abstraction

Lecture by Dan Licata

March 27, 2012

In these lectures, we will discuss the use of the ML module system for structuring large programs. Key concepts include *structures*, *signatures*, *abstract types*, *type classes*, and *functors*.

The goal is to figure out how to divide up programs that are so big that you can't fit them in your head at once. Our basic answer is:

1. divide programs up into chunks called *modules*, or *structures*.
2. limit which modules interact with each other.
3. for modules that do interact, specify their interaction with a *signature*, which summarizes the functionality of one module for use in another.
4. abstract over patterns of modules using *functors* (next time).

For example, the code for this HW is divided up into a variety of structures:

```
Sequenece
RealPlane
RatPlane
Mechanics
BBox
NaiveNBody
BarnesHut
Simulation
```

You are implementing `BarnesHut`. Your implementation relies on a bunch of our libraries: `RealPlane` and `RatPlane` are two different implementations of the plane—the former uses floating point numbers for actual simulation; the latter uses arbitrary-precision rationals, which are much slower, but good for predictable testing (because floating point addition/multiplication are not associative, if you do operations slightly differently than our solution, your output won't exactly match our tests). `Mechanics` is basic mechanics—like the `accOn` function from last time. `BBox` implements *bounding boxes*: a rectangle containing a bunch of points. `NaiveNBody` (the quadratic code from lecture last week) and `BarnesHut` are two different ways of computing accelerations. `Simulation` is the outer loop that ties it all together, using $p' = p + vt + 1/2at^2$ etc. to update bodies.

To make it easier to understand this code, we limit interactions—`Mechanics` deals only with one body at a time; other code lifts operations to `Sequence`s. We also specify interfaces using *signatures*.

1 Signatures

Here is the interface for sequences that we talked about last week, written out as a signature:

```
signature SEQUENCE =
sig
  type 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val reduce : (('a * 'a) -> 'a) -> 'a -> 'a seq -> 'a
  val tabulate : (int -> 'a) -> int -> 'a seq
  val nth    : int -> 'a seq -> 'a
end
```

This defines a *signature* named `SEQUENCE` (signatures are traditionally named in all caps). Our code *implements* this signature (provides something with this specification), whereas yours is the *client* (makes use of the types and functions provided). The type `'a seq` is what is called an *abstract type*: clients do not know its implementation. That is, the client (you) know that there is some type `'a seq` with the provided operations. The implementor picks a type to fill in for `'a seq` and implements the operations.

Another example is the signature `SPACE`, which describes the plane:

```
signature SPACE =
sig
    type vec
    type point
    val --> : point * point -> vec
    ...
end
```

Here we make the types `vec` and `point` abstract—unlike last time, where we knew that they were implemented as pairs of scalars. Right away, you can see that *there can be more than one implementation of a signature*: both `RealPlane` and `RatPlane` have the signature `SPACE`.

Signatures serve two purposes. The first is *communication*: as much as possible, you want to write code so that other people need only to read the signatures of your code, not the implementation. This means writing down the types and specs in a separate place from the implementation—e.g., you might annotate `map` with the spec that `map f <x1,...,xn> ==> <f x1,...,f xn>`, to demand that this spec hold for all implementations. The second is *information hiding.* That is, we deliberately hide information (operations in the implementation that are not in the signature; the identities of abstract types) from ourselves/others. Why?

- It breaks the code up into chunks that are easier to understand. You don't know the implementation of `'a seq`, so you don't need to think about it. All you know about it is what is said right there in the signature.

- It makes it easier to *evolve* the code over time. The client can pick another implementation of the same signature, without otherwise changing the code (e.g. switching between `RealPlane` and `RatPlane`). The implementor can change the implementation type, without the clients needing to change.

- It localizes reasoning about representation invariants. An abstract type uniquely identifies the code that acts on it. So if there is a violation of an invariant, you know what code to look at.

  An example of this occurs in `BBox`: internally, `BBox` is represented as a pair of the lower-left and upper-right corners. Last semester, students had access to this representation, and screwed things up, by putting the points in the wrong order (then `BBox.contained` doesn't work). This year, we used the module system to make `BBox.bbox` an abstract type, so you can't have this bug! Because only the functions in the implementation of `BBox`, not client code, can access the internal representation, you ever see a `bbox` whose points are in the wrong order, it's our fault, not yours.

- It lets you use the type system to find bugs at compile-time, by making different concepts have different types, even if the underlying representations are the same.

  An example of this is `point` and `vec` in `SPACE`s: both are implemented as pairs of scalars, but they mean different things, and using one in place of the other is a bug. To convert a point to a vector or vice versa, you need to remember to specify the tail, and now the type system forces you to do this.

- It allows you to type-check and compile different parts of your program independently, before the parts they depend on have been written.

We'll see more examples of all of these later on.

2 Structures

Here's an example of the implementation/client divide:

```
(* implementation *)
structure ArraySeq : SEQUENCE = struct ... end

(* client *)
fun accels(bodies : body ArraySeq.seq) : vec ArraySeq.seq =
  ArraySeq.map (fn b1 => sum bodies (fn b2 => accOn(b1,b2)))
```

The implementation is what is called a *structure*, which implements the abstract type and the corresponding operations. The client code is the quadratic n-body code from last week. The thing to note is that you refer to components of structures using *dot notation*: you write `ArraySeq.map` for the `map` component of `ArraySeq`. This includes both values, like `ArraySeq.map`, and types, like `ArraySeq.seq`.

It's important to understand that the type is a component of the structure—a common misconception is that the types come from te signature, whereas the values come from the structure. Unlike langauges like Java, where an interface defines a single abstract type, ML signatures can describe packages that define many related abstract types at once. This is important when you want one block of code that knows the implementation of all of them, and then these implementations to be hidden from the clients—e.g. with `point` and `vec` above!

Structures must define the types in the signature and implement the values on them. For example, in our sequence implementation

```
structure ArraySeq : SEQUENCE =
struct
   type 'a seq = 'a array
   val map = <some code on arrays>
   ...
end
```

You can have different implementations of the same signature and its abstract types. Why would you want this? They might have

- The same behavior, with different costs. E.g. array-based sequences have $O(1)$ `nth` but $O(n)$ `cons` (you need to copy). Tree-based sequences have $O(\log n)$ `nth` and `cons`. So you might want to choose between them depending on which operation you use more often.
- Different but "similar enough" behavior. E.g. `RatPlane` and `RealPlane` behave differently, but both implement the plane in terms of some notion of number, and are useful for different purposes.

Here's a second implementation of sequences as trees:

```
structure Seq : SEQUENCE =
struct
  datatype 'a tree = Leaf of 'a | Empty | Node of 'a tree * 'a tree
  type 'a seq = 'a tree

  fun map f (t : 'a seq) = case t of
      Leaf x => Leaf (f x)
    | Empty => Empty
    | Node (t1 , t2) => Node (map f t1, map f t2)

  fun reduce n e t = ...
  fun tabulate f n = ...

  (* helper function is not visible to clients *)
  fun numLeaves t =
      case t of
          Empty => 0
        | Leaf _ => 1
        | Node (t1,t2) => numLeaves t1 + numLeaves t2

  fun nth i s =
      case (s , i) of
        (Empty , _) => raise Fail "out of range"
      | (Leaf x , 0) => x
      | (Leaf x , _) => raise Fail "out of range"
      | (Node (t1 , t2) , n) =>
          let val s1 = numLeaves t1
          in
              case n < s1 of
                  true => nth n t1
```

```
                     | false => nth (n - s1) t2
            end
end
```

What does it mean for a structure to satisfy a signature?

1. The structure provides a definition for each declaration in the signature. In particular, the structure can have more declarations, in which case the ones not in the signature are not visible from the outside. Note that `fun` declarations can be used to provide values of function type (because functions are values!).

2. Values must have the specified types, knowing the implementation of the abstract types.

As an example of the first point, `numLeaves` is not visible to clients, because it is not mentioned in the signature. As an example of the second, when defining each function (e.g. `map`), you can pattern match on a `'a seq` because you know it is an `'a tree`.

Type abstraction `Seq.seq` is an abstract type: the *only* thing a client can do is use the operations provided in the interface. Why?

There is a subtlety, which we will talk about more next time: in fact, ML does propagate the definitions of types into the signature, so clients learn that `'a seq = 'a tree`. There are good reasons for why you want this, which we will talk about next time; but it's not what you want here.

This seems to contradict what I've been saying about `'a seq` being an abstract type. However, it doesn't! And the reason why it doesn't is already on the board: First, declarations not in the signature are not visible to clients. Second, this includes datatypes and constructors. Third, the only thing you can do with a datatype is pattern-match or construct a value, and for both of these you need to know the datatype constructors. So, if clients do not get access to the datatype constructors, they can't do anything with a value of that datatype except pattern-match.

Thus:

> **Methodology:** To make a type abstract, define it to be a datatype that is not exported in the signature.

If you always do this, clients of a module will only know exactly what is in the signature.

2.1 Client Evoluation

We can easily change the client code to use this implementation, by replacing all references to `ArraySeq` with `TreeSeq`.

```
(* client *)
fun accels(bodies : body TreeSeq.seq) : vec TreeSeq.seq =
  TreeSeq.map (fn b1 => sum bodies (fn b2 => accOn(b1,b2)))
```

In fact, there is an easier way: We can make a structure binding that defines the name `Seq` to be `TreeSeq`.

```
structure Seq : SEQUENCE = TreeSeq
(* client *)
fun accels(bodies : body Seq.seq) : vec Seq.seq =
  Seq.map (fn b1 => sum bodies (fn b2 => accOn(b1,b2)))
```

Because TreeSeq satisifies th same signature, this code is guaranteed to still type check, and will probably still work, as long as the different implementations of the signature in fact behave the same.

Methodologically, it is good to name structures with something descriptive about their particular implementation of the signature; e.g. ArraySeq and TreeSeq.

2.2 Implementation Evolution

As an example of implementation evolution, the above nth function is inefficient, because it computes the size of the tree at each step—thus it takes linear rather than logarithmic time. We can fix this by storing, at each node, the size of the left subtree. Because the type 'a tree is abstract, we can make this change *without changing any client code.*

```
structure TreeSeq : SEQUENCE =
struct

    (* representation invariant:
       in Node (t1 , s1 , t2) , s1 is the number of leaves in t1
       *)
    datatype 'a tree = Leaf of 'a | Empty | Node of 'a tree * int * 'a tree
    type 'a seq = 'a tree

    fun map f t = case t of
        Leaf x => Leaf (f x)
      | Empty => Empty
      | Node (t1 , s1 , t2) => Node (map f t1, s1, map f t2)

    fun nth i s =
        case (s , i) of
            (Empty , _) => raise Fail "out of range"
          | (Leaf x , 0) => x
          | (Leaf x , _) => raise Fail "out of range"
          | (Node (t1 , s1 , t2) , n) =>
                case n < s1 of
                    true => nth n t1
                  | false => nth (n - s1) t2
    ...
end
```

We change the datatype definition to store an integer at each node. In functions that construct sequences, we need to compute this information—e.g. in map, where the map preserves the size. In nth, we use the value to direct the search.

Because the type 'a tree is not exported from this module, it is only the implementor, and not the client, who can break the representation invariant that the integer is the size of the left subtree: if you ever see a tree that does not satisfy this invariant, you know that it is the implementation of TreeSeq's fault. The abstract type *uniquely identifies* the code that acts on it. Put another way, if you prove that each function in the signature preserves this invariant, then you know that *any larger program in which you use this module* will maintain the invariant!