CSE341, Fall 2011, Lecture 9 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.

The previous lecture defined function closures and lexical scope. It also showed how closures make higher-order functions like map, filter, and fold much more useful because their function arguments can have "private data" (not unlike how objects in Java have private fields). This lecture covers four other general patterns where function closures are amazingly useful:

- Combining functions into larger functions (e.g., function composition)
- Currying: Multi-argument functions and partial application (and an unfortunate digression to the value restriction)
- Callbacks, as used in reactive programming
- Implementing an absract data type (ADT) as a record of functions (requires some rather fancy techniques, but hopefully the result is understandable)

Except for some syntatic sugar, there are no new semantics related to closures, just new idioms. For the callback idiom, we do finally admit that ML has support for mutable data, which requires introducing ML's refs and the primitives for building, accessing, and mutating them.

Combining Functions

Function composition

When we program with lots of functions, it is useful to create new functions that are just combinations of other functions. You have probably done similar things in mathematics, such as when you compose two functions. For example, here is a function that does exactly function composition:

```
fun compose (f,g) = fn x => f (g x)
```

It takes two functions `f` and `g` and returns a function that applies its argument to `g` and makes that the argument to `f`. Crucially, the code `fn x => f (g x)` uses the `f` and `g` in the environment where it was defined. Notice the type of `compose` is inferred to be `('a -> 'b) * ('c -> 'a) -> 'c -> 'b`, which is equivalent to what you might write: `('b -> 'c) * ('a -> 'b) -> ('a -> 'c)` since the two types simply use different type-variable names consistently.

As a cute and convenient library function, the ML library defines the infix operator `o` as function composition, just like in math. So instead of writing:

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))
```

you could write:

```
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
```

But this second version makes clearer that we can just use function-composition to create a function that we bind to a variable with a val-binding, as in this third version:

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

While all three versions are fairly readable, the first one does not immediately indicate to the reader that `sqrt_of_abs` is just the composition of other functions.

The Pipeline Operator

In functional programming, it is very common to compose other functions to create larger ones, so it makes sense to define convenient syntax for it. While the third version above is concise, it, like function composition in mathematics, has the strange-to-many-programmers property that the computation proceeds from right-to-left: "Take the absolute value, convert it to a real, and compute the square root" may be easier to understand than, "Take the square root of the converstion to real of the absolute value."

We can define convenient syntax for left-to-right as well. Let's first define our own infix operator that lets us put the function to the right of the argument we are calling it with:

```
infix |> (* tells the parser |> is a function that appears between its two arguments *)
fun x |> f = f x
```

Now we can write:

```
fun sqrt_of_abs i = i |> abs |> Real.fromInt |> Math.sqrt
```

This operator, commonly called the *pipeline operator*, is very popular in F# programming. As we have seen, there is nothing complicated about its semantics.

Other functions

There are many ways we might combine functions beyond just composition, so it is good to have a language that lets us define our own ways to define new higher-order functions. Here are two similar examples where we treat `g` as a "back-up" option if `f` returns `NONE` (in the first example) or raises an exception (in the second example).

```
(* val backup1 = fn : ('a -> 'b option) * ('a -> 'b) -> 'a -> 'b *)
fun backup1(f,g) = fn x => case f x of NONE => g x | SOME y => y

(* val backup2 = fn : ('a -> 'b) * ('a -> 'b) -> 'a -> 'b *)
fun backup2 (f,g) = fn x => g x handle _ => h x
```

Currying: Multi-argument functions and partial application

The next idiom we consider is very convenient in general, and is often used when defining and using higher-order functions like map, filter, and fold. We have already seen that in ML every function takes exactly one argument, so you have to use an idiom to get the effect of multiple arguments. Our previous approach passed a tuple as the one argument, so each part of the tuple is conceptually one of the multiple arguments. Another more clever and often more convenient way is to have a function take the first conceptual argument and return another function that takes the second conceptual argument and so on. Lexical scope is essential to this technique working correctly.

This technique is called *currying* after a logician named Haskell Curry who studied related ideas (so if you don't know that, then the term currying does not make much sense).

Defining and Using a Curried Function

Here is an example of a "three argument" function that uses currying:

```
val sorted3 = fn x => fn y => fn z => z >= y andalso y >= x
```

If we call `sorted3 4` we will get a closure that has `x` in its environment. If we then call this closure with `5`, we will get a closure that has `x` and `y` in its environment. If we then call this closure with `6`, we will get `true` because 6 is greater than 5 and 5 is greater than 4. That is just how closures work.

So `((sorted3 4) 5) 6` computes exactly what we want and feels pretty close to calling `sorted3` with 3 arguments. Even better, the parentheses are optional, so we can write exactly the same thing as `sorted3 4 5 6`, which is actually fewer characters than our old tuple approach where we would have:

```
fun sorted3_tupled (x,y,z) = z >= y andalso y >= x
val someClient = sorted3_tupled(4,5,6)
```

In general, the syntax `e1 e2 e3 e4` is implicitly the nested function calls `(((e1 e2) e3) e4)` and this choice was made because it makes using a curried function so pleasant.

Partial Application

Even though we might expect most clients of our curried `sorted3` to provide all 3 conceptual arguments, they might provide fewer and use the resulting closure later. This is called "partial application" because we are providing a subset (more precisely, a prefix) of the conceptual arguments. As a silly example, `sorted3 0 0` returns a function that returns `true` if its argument is nonnegative.

Partial Application and Higher-Order Functions

Currying is particularly convenient for creating similar functions with iterators. For example, here is a curried version of a fold function for lists:

```
fun fold f = fn acc => fn l =>
  case l of
    []     => acc
  | hd::tl => fold f (f(acc,hd)) tl
```

Now we could use this fold to define a function that sum's a list elements like this:

```
fun sum1 l = fold ((fn (x,y) => x+y) 0 l
```

But that is unnecessarily complicated compared to just using partial application:

```
val sum2 = fold (fn (x,y) => x+y) 0
```

The convenience of partial application is why many iterators in ML's standard library use currying with the function they take as the first argument. For example, the types of all these functions use currying:

```
val List.map = fn : ('a -> 'b) -> 'a list -> 'b list
val List.filter = ('a -> bool) -> 'a list -> 'a list
val List.foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

As an example, `List.foldl((fun (x,y) => x+y), 0, [3,4,5])` does not type-check because `List.foldl` expects a `'a * 'b -> 'b` function, not a triple. The correct call is `List.foldl (fn (x,y) => x+y) 0 [3,4,5]`, which calls `List.foldl` with a function, which returns a closure and so on.

There is syntactic sugar for defining curried functions; you can just separate the conceptual arguments by spaces rather than using anonymous functions. So the better style for our fold function would be:

```
fun fold f acc l =
  case l of
    []     => acc
  | hd::tl => fold f (f(acc,hd)) tl
```

Another useful curried function is `List.exists`, which we use in the callback example below. These library functions are easy to implement ourselves, so we should understand they are not fancy:

```
fun exists predicate lst =
    case lst of
      [] => false
    | hd::tl => predicate hd orelse exists predicate tl
```

Currying in General

While currying and partial application are great for higher-order funtions, they are great in general too. They work for any multi-argument function and partial application can also be surprisingly convenient. In this example, both `zip` and `range` are defined with currying and `countup` partially applies `range`. The `add_numbers` function turns the list `[v1,v2,...,vn]` into `[(1,v1),(2,v2),...,(n,vn)]`.

```
fun zip xs ys =
    case (xs,ys) of
        ([],[]) => []
      | (x::xs',y::ys') => (x,y) :: (zip xs' ys')
      | _ => raise Empty

fun range i j = if i > j then [] else i :: range (i+1) j

val countup  = range 1

fun add_numbers xs = zip (countup (length xs)) xs
```

Combining Functions to Curry and Uncurry Other Functions

Sometimes functions are curried but the arguments are not in the order you want for a partial application. Or sometimes a function is curried when you want it to use tuples or vice-versa. Fortunately our earlier idiom of combining functions can take functions using one approach and produce functions using another:

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

Looking at the types of these functions can help you understand what they do. As an aside, the types are also fascinating because if you pronounce `->` as "implies" and `*` as "and", the types of all these functions are logical tautologies.

Efficiency

Finally, you might wonder which is faster, currying or tupling. It almost never matters; they both do work proportional to the number of conceptual arguments, which is typically quite small. For the performance-critical functions in your software, it *might* matter to pick the faster way. In the version of the ML compiler we

are using, tupling happens to be faster. In widely used implementations of OCaml, Haskell, and F#, curried functions are faster so they are the standard way to define multi-argument functions in those languages.

The Value Restriction: See Next Lecture

Once you have learned currying and partial application, you might try to use it to create a polymorphic function. Unfortunately, uses like this do not work in ML:

```
val mapSome = List.map SOME (*turn [v1,v2,...,vn] into [SOME v1, SOME v2, ..., SOME vn]*)
val pairIt = List.map (fn x => (x,x)) (*turn [v1,v2,...,vn] into [(v1,v1),(v2,v2),...,(vn,vn)]*)
```

Given what we have learned so far, there is no reason why this should not work, especially since all these functions do work:

```
fun mapSome xs = List.map SOME xs
val mapSome = fn xs => List.map SOME xs
val pairIt : int list -> (int * int) list = List.map (fn x => (x,x))
val incrementIt = List.map (fn x => x+1)
```

The reason is called the *value restriction* and it is sometimes annoying. It is in the language for good reason: without it, the type-checker might allow some code to break the type system. This can happen only with code that is using mutation and the code above is not, but the type-checker does not know that.

The simplest approach is to ignore this issue until you get a warning/error about the value restriction. When you do, turn the val-binding back into a fun-binding like in the first example above of what works.

The next lecture, on type inference, will discuss the value restriction in some more detail.

Callbacks (and a Digression to Introduce ML Refs

The next common idiom we consider is implementing a library that detects when "events" occur and informs clients that have previously "registered" their interest in hearing about events. Clients can register their interest by providing a "callback" — a function that gets called when the event occurs. Examples of events for which you might want this sort of library include things like users moving the mouse or pressing a key. Data arriving from a network interface is another example. Computer players in a game where the events are "it is your turn" is yet another.

The purpose of these libraries is to allow multiple clients to register callbacks. The library implementer has no idea what clients need to compute when an event occurs, and the clients may need "extra data" to do the computation. So the library implementor should not restrict what "extra data" each client uses. A closure is ideal for this because a function's type `t1 -> t2` does not specify the types of any other variables a closure uses, so we can put the "extra data" in the closure's environment.

If you have used "event listeners" in Java's Swing library, then you have used this idiom in an object-oriented setting. In Java, you get "extra data" by defining a subclass with additional fields. This can take an awful lot of keystrokes for a simple listener, which is a (the?) main reason the Java language added anonymous inner classes (which you do not need to know about for this course, but we will show an example later), which are closer to the convenience of closures.

To see an example in ML, we will finally introduce ML's support for mutation. Mutation is okay in some settings. In this case, we really do want registering a callback to "change the state of the world" — when an event occurs, there are now more callbacks to invoke. In ML, most things really cannot be mutated. Instead you must create a *reference*, which is a container whose contents can be changed. You create a new reference with the expression `ref e` (the initial contents are the result of evaluating `e`). You get a reference `r`'s current contents with `!r` (not to be confused with negation in Java or C), and you change `r`'s contents with `r := e`. The type of a reference that contains values of type `t` is written `t ref`.

Our example uses the idea that callbacks should be called when a key on the keyboard is pressed. We will pass the callbacks an `int` that encodes which key it was. Our interface just needs a way to register callbacks. (In a real library, you might also want a way to unregister them.)

```
val onKeyEvent : (int -> unit) -> unit
```

Clients will pass a function of type `int -> unit` that, when called later with an `int`, will do whatever they want. To implement this function, we just use a reference that holds a list of the callbacks. Then when an event actually occurs, we assume the function `on_event` is called and it calls each callback in the list:

```
val cbs : (int -> unit) list ref = ref []
fun onKeyEvent f = cbs := f::(!cbs) (* The only "public" binding *)
fun onEvent i =
   let fun loop fs =
        case fs of
          [] => ()
        | f::fs' => (f i; loop fs')
    in loop (!cbs) end
```

Most importantly, the type of `onKeyEvent` places no restriction on what extra data a callback can access when it is called. Here are different clients (calls to `onKeyEvent`) that use different bindings of different types in their environment. (The `val _ = e` idiom is common for executing an expression just for its side-effect, in this case registering a callback.)

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ => timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
    onKeyEvent (fn j => if i=j
                        then print ("you pressed " ^ Int.toString i ^ "\n")
                        else ())

val _ = printIfPressed 4
val _ = printIfPressed 11
val _ = printIfPressed 23
```

Implementing an ADT

This last example is the fanciest and most subtle. It is not the sort of thing programmers typically do — there is usually a simpler way to do it in a modern programming lnaguage. It is included as an advanced example to demonstrate that a record of closures that have the same environment is a lot like an object where the functions are methods and the bindings in the environment are private fields and methods. There are no new language features here, just lexical scope. It suggests (correctly) that functional programming and object-oriented programming are more similar than they might first appear (a topic we will revisit later in the course; there are important differences).

The key to an abstract data type (ADT) is requiring clients to use it via a collection of functions rather than directly accessing its private implementation. Thanks to this abstraction, we can later change how the data type is implemented without changing how it behaves for clients. In an object-oriented language, you might implement an ADT by defining a class with all private fields (inaccessible to clients) and some public methods (the interface with clients). We can do the same thing in ML with a record of closures; the variables that the closures use from the environment correspond to the private fields.

As an example, consider an implementation of a set of integers that supports creating a new bigger set and seeing if an integer is in a set. Our sets are mutation-free in the sense that adding an integer to a set produces a new, different set; we could just as easily define a mutable version using the references introduced above. In ML, we could define a type that describes this interface:

```
datatype set = S of { insert : int -> set, member : int -> bool, size : unit -> int }
```

Roughly speaking, a set is a record with two fields, each of which holds a function. It would be simpler to write:

```
type set = { insert : int -> set, member : int -> bool, size : unit -> int }
```

but this does not work in ML because `type` bindings cannot be recursive. So we have to deal with the mild inconvenience of having a constructor `S` around our record of functions defining a set even though sets are each-of types, not one-of types. Notice we are not using any new types or features; we simply have a type describing a record with fields named `insert` and `member`, each of which holds a function.

Once we have an empty set, we can use its `insert` field to create a one-element set, and then use that set's `insert` field to create a two-element set, and so on. So the only other thing our interface needs is a binding like this:

```
val empty_set = ... : set
```

Before implementing this interface, let's see how a client might use it (many of the parentheses are optional but may help understand the code):

```
fun use_sets () =
    let val S s1 = empty_set
        val S s2 = (#insert s1) 34
        val S s3 = (#insert s2) 34
        val S s4 = #insert s3 19
    in
        if (#member s4) 42
        then 99
        else if (#member s4) 19
        then 17 + (#size s3) ()
        else 0
    end
```

Again we are using no new features. `#insert s1` is reading a record field, which in this case produces a function that we can then call with 34. If we were in Java, we might write `s1.insert(34)` to do something similar. The `val` bindings use pattern-matching to "strip off" the `S` constructors on values of type `set`.

There are many ways we could define `empty_set`; they will all use the technique of using a closure to "remember" what elements a set has. Here is one way:

```
val empty_set =
    let
        fun make_set lst = (* lst is a "private field" in result *)
            let (* contains a "private method" in result *)
                fun contains i = List.exists (fn j => i=j) lst
```

```
        in
            S { insert = fn i => if contains i
                                 then make_set lst
                                 else make_set (i::lst),
                member = contains,
                size   = fn () => length lst
              }
        end
in
    make_set []
end
```

All the fanciness is in `make_set` and `empty_set` is just the record returned by `make_set []`. What `make_set` returns is a value of type `set`. It is essentially a record with three closures. The closures can use `lst`, the helper function `contains`, and `make_set`. Like all function bodies, they are not executed until they are called.