CSE 341, Spring 2008, Lecture 8 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.

So far our uses of higher-order functions (i.e., functions taking or returning other functions) have used *closed functions*, meaning functions that only used variables that were arguments or defined inside the function. It is extremely powerful to let functions use variables that are in scope (i.e., in the environment) where the function was defined.

To understand how this works, realize that functions are values, but they are not just the code. They are really a pair (not ML pairs, but still something with two parts): The code and the environment that was current when the function was defined. We call this pair a *function closure* or just a *closure*. Before seeing why this is a good idea, it is worth going through several useless examples carefully. Consider:

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x+y)
```

When `f` is defined, the environment binds `x` to 1, so `f` is the increment function *no matter where it is used.* In the later expression `f (x+y)`, we evaluate `f` to the closure that has the code `fn y => x+y` and environment mapping `x` to 1. We evaluate the `x+y` at the call-site to 5 (since in the environment at the call-site `x` maps to 2 and `y` maps to 3) then execute the function body `x+y` in the environment with `x` mapping to 1 extended to map `y` to 5. So `z` ends up bound to 6.

Closures are more interesting and useful when they are created inside a function and the *free variables* (variables used but not defined in the function) refer to local bindings. For example:

```
fun f y = let val x = 2 in fn z => x + y + z end
```

This function returns a closure with code `fn x => x + y +z` and an environment mapping `x` to 2 and `y` to whatever the caller to `f` passed for `y`. So `f 6` would produce a closure that (when called later) always add 8 to its argument and `f ~1` would produce a closure that increments its argument.

This definition of closure is also important when passing a function to another function. For example, consider:

```
fun f g = let val x = 3 in g 2 end
val x = 4
fun h y = x + y
val z = f h
```

This code binds `6` to `z` because the closure passed to `f` always adds 4 to its argument. That is because we evaluate the body of `h` in the environment where `h` is defined (which here maps `x` to 4), not where the function is later called (in the body of `f` where there happens to be a different `x` bound to 3).

The rule that free variables in a function refer to the value they have in the environment where the function is *defined* is called *lexical scope.* The natural alternative — using the environment where the function is called — is called *dynamic scope.* There are several reasons to prefer lexical scope for variables. For example, under dynamic scope this code would try to call "the function 37" which makes no sense:

```
fun f x = x
fun g y = f y
val f = 37
val x = g 14
```

More generally, type-checking relies pretty fundamentally on lexical scope. However, the issue is not just type-checking. The good thing about lexical scope is that functions behave the same way no matter where they are used. So you can reason about and test a function without worrying about the environment being different at some place where it is called.

For the rest of this lecture and the next lecture, we will consider six different idioms where using function closures is elegant and helpful. Using closures, which often involves these idioms, and avoiding mutation are the essentials of programming in a functional style.

Creating similar functions

If we need multiple functions that differ in some small way, we can write a function that given an argument returns an appropriate function. In this silly but small example, the `addn` function returns a function that adds `n` to its argument, which we can then use to make any number of different adding functions. Crucially, the function `fn m => n+m` uses the `n` in the environment where it was defined:

```
val addn = fn n => fn m => n+m
val increment = addn 1
val add_two = addn 2
fun f n =
   if n=0
   then []
   else (addn n)::(f (n-1))
```

Combining functions

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 `('a -> 'b) * ('c -> 'a) -> 'c -> 'b`. Recall it is common (but not always the case) that higher-order functions have *polymorphic types* (types with `'a` etc. in them, the topic of lecture 10).

A second similar example uses `h` as a "back-up" function in case `g` returns `NONE`:

```
fun f (g,h) = fn x => case g x of NONE => h x | SOME y => y
```

If you are doing this sort of thing often ("if one thing is `NONE` try another thing"), abstracting it into a helper function that takes other functions can be helpful.

Passing functions with private data to iterators

Perhaps the most common and important use of closures is passing them to functions that recurse over data structures, like the `map` function we saw in the previous lecture:

```
fun map (f,lst) =
    case lst of
        [] => []
      | fst::rest => (f fst)::(map(f,rest))
```

All our examples last time passed functions for `f` that only used their arguments; passing closures that use free variables is much more powerful. For example, this use of `map` truncates values to be less than some value:

```
fun truncate (lst,hi) = map((fn x => if x > hi then hi else x), lst)
```

Another higher-order function over lists that is even more powerful than `map` is `fold`:

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

`fold` takes an "initial answer" `acc` and uses `f` to "combine" `acc` and the first element of the list, using this as the new "initial answer" for "folding" over the rest of the list. We can use `fold` to take care of iterating over a list while we provide some function that expresses how to combine elements. For example, to sum the elements in a list `lst`, we can do:

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

As with `map`, much of `fold`'s power comes from clients passing closures that can have "private fields" (in the form of free variables) for keeping data they want to consult. Similar to the `truncate` example, we could count how many elements are too large:

```
fun num_too_big(lst,hi) = fold((fn (x,y) => if y > hi then x+1 else x),0,lst)
```

This pattern of splitting the recursive traversal (`fold` or `map`) from the data-processing done on the elements (the closures passed in) is fundamental. In our examples, both parts are so easy we could just do the whole thing together in a few simple lines. More generally, we may have a very complicated set of data structures to traverse or we may have very involved data processing to do. It is good to *separate these concerns* so that the programming problems can be solved separately.

While functional programmers have been doing this for decades, it has recently gained much attention thanks to Google's MapReduce (and similar systems from other companies and open-source projects). This work was first done in about 2004 by people very familiar with languages like Scheme and ML and it is revolutionizing how large-scale data-intensive computations are done.

In a nutshell, a company like Google has so much data that it is spread across thousands or tens of thousands of computers and organized in a very complicated way. So one set of people works on writing functions like `map` and `fold` (a function pretty similar to `fold` is called `reduce`) that does all the communication and data-passing for the huge set of computers. There are also many complicated different things you might want to do with this data — handle search queries, look for patterns in news reporting, do spell-checking, etc. These can all be written just in terms of functions passed to `map` and `reduce` without any concern for how the computation is spread across the computers.

It turns out to be crucial that the functions passed to `map` and `reduce` do not do mutation such as assigning to some global variable. The `MapReduce` system assumes that some function `f` can be called on some argument `x` any number of times and it will always return the same answer and not have any affect on the state of any other variables. This is essential because when you have ten-thousand computers, one or more of them are likely to fail pretty often. When a computer fails, the system can just repeat any computation it did on a different computer since the lack of mutation ensures that it makes absolutely no difference what order the data is processed or how many times it is processed.

In summary, `MapReduce` boils down to 3 concepts:

- Building a fault-tolerant distributed system
- Using higher-order functions to provide a simple interface for the programmers doing the data-processing
- Avoiding mutation so that computations can be repeated and reordered by the system without affecting the result

This course has had a lot to say about 2 of these 3; the other we will not go anywhere near.