15-150 Lecture 11:
Higher-Order Functions

Lecture by Dan Licata

Februrary 16, 2012

1 Functions as Arguments

Next, we're going to talk about the thing that makes functional programming *functional*:

> functions are values that can be passed to, and returned from, other functions

Consider the following two functions:

```
fun double (x : int) : int = x * 2
fun doubAll (l : int list) : int list =
    case l of
        [] => []
      | x :: xs => double x :: doubAll xs

fun raiseBy (l : int list , c : int) : int list =
    case l of
        [] => []
      | x :: xs => (x + c) :: raiseBy (xs,c)
```

How do they differ? They both do something to each element of a list, but they do different things (adding one, multiplying by `c`).

We want to write one function that expresses the pattern that is common to both of them:

```
fun map (f : int -> int , l : int list) : int list =
    case l of
        [] => []
      | x :: xs => f x :: map (f , xs)
```

The idea with `map` is that it takes a function `f : int -> int` as an argument, which represents what you are supposed to do to each element of the list.

`int -> int` is a type and can be used just like any other type in ML. In particular, a function like `map` can take an argument of type `int -> int`; and, as we'll see next time, a function can return a function as a result.

For example, we can recover `doubAll` like this:

```
fun doubAll l = map (double , l)
```

If you substitute `add1` into the body of `map`, you see that it results in basically the same code as before.

Anonymous functions Another way to instantiate `map` is with an anonymous function, which is written `fn x => 2 * x`:

```
fun doubAll l = map (fn x => 2 * x , l)
```

The idea is that `fn x => 2 * x` has type `int -> int` because assuming `x:int`, the body `2 * x : int`. To evaluate an anonymous function applied to an argument, you plug the value of the argument in for the variable. E.g. `(fn x => 2 * x) 3 |-> 2 * 3`. **Functions are values:** The value of `fn x => x + (1 + 1)` is `fn x => x + (1 + 1)`. You don't evaluate the body until you apply the function.

`doubAll` can be defined anonymously too:

```
val doubAll : int list -> int list = fn l => map (fn x => 2 * x , l)
```

Closures A somewhat tricky, but very very useful, fact is that anonymous functions can refer to variables bound in the enclosing scope. This gets used when we instantiate `map` to get `raiseBy`:

```
fun raiseBy (l , c) = map (fn x => x + c , l)
```

The function `fn x => x + c` adds `c` to its argument, where `c` bound as the argument to `raiseBy`. For example, in

```
    raiseBy ( [1,2,3] , 2)
|-> map (fn x => x + 2 , [1,2,3])
|-> (fn x => x + 2) 1 :: map (fn x => x + 2 , [2,3])
|-> 1 + 2 :: map (fn x => x + 2 , [2,3])
|-> 3 :: map (fn x => x + 2 , [2,3])
|-> 3 :: (fn x => x + 2) 2 :: map (fn x => x + 2 , [3])
 == 3 :: 4 :: map (fn x => x + 2 , [3])
 == 3 :: 4 :: (fn x => x + 2) 3 :: map (fn x => x + 2 , [])
 == 3 :: 4 :: 5 :: map (fn x => x + 2 , [])
 == 3 :: 4 :: 5 :: []
```

the `c` gets specialized to `2`. If you keep stepping, the function `fn x => x + 2` gets applied to each element of `[1,2,3]`. The important fact, which takes some getting used to, is that the function `fn x => x + 2` is *dynamically generated*: at run-time, we make up new functions, which do not appear anywhere in the program's source code!

Here's a puzzle: what does

```
let val x = 3
    val f = fn y => y + x
    val x = 5
 in
    f 10
 end
```

evaluate to?

Well, you know how to evaluate `let`: you evaluate the declarations in order, substituting as you go. So, you get

```
let val x = 3
    val f = fn y => y + 3
    val x = 5
 in
    f 10
 end
```

The fact that `x` is shadowed below is irrelevant; the result is `13`. This is one of the reasons why we've been teaching you the substitution model of evaluation all semester; it explains tricky puzzles like this in a natural way.

Polymorphism Another instance of the same pattern is:

```
fun showAll (l : int list) : string list =
    case l of
        [] => []
      | x :: xs => Int.toString x :: showAll xs
```

However, this is not an instance of `map`, because it transforms an `int list` into a `string list`.

We can fix this by giving `map` a polymorphic type:

```
fun map (f : 'a -> 'b , l : 'a list ) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map (f , xs)
```

Then both `doubAll` and `showAll` are instances:

```
fun doubAll l = map (fn x => 2 * x , l)
fun showAll l = map (Int.toString , l)
```

`map` is an example of what is called a *higher-order function*, a function that takes a function as an argument.

2 Exists

Here's another example of a higher-order function, which captures the pattern of "is there some element of a list such that ...":

```
fun evenP (x : int) : bool = (x mod 2) = 0

fun hasEven (l : int list) : bool =
    case l of
        [] => false
      | x :: xs => evenP x orelse hasEven xs

fun doctor (l : string list) : bool =
    case l of
        [] => false
```

```
      | x :: xs => (x = "doctor") orelse doctor xs

fun exists (p : 'a -> bool, l : 'a list) : bool =
    case l of
        [] => false
      | x :: xs => p x orelse exists (p, xs)

fun hasEven (l : int list) : bool = exists(evenP, l)
fun doctor (l : string list) : bool = exists((fn x => x = "doctor"), l)
```

3 Currying

Currying is the idea that a function with two arguments is roughly the same thing as a function with one argument, that returns a function as a result. For example, here is the curried version of `map`:

```
fun map (f : 'a -> 'b) : 'a list -> 'b list =
    fn l =>
     case l of
         [] => []
       | x :: xs => f x :: map f xs
```

Instead of taking two arguments, a function and a list, it takes one argument (a function), and returns a function that takes the list and computes the result. Note that function application left-associates, so `f x y` gets parsed as `(f x) y`. Thus, a curried function can be applied to multiple arguments just by listing them in sequence, as in `map f xs` above. This is really two function applications: one of `map` to `f`, producing `map f : 'a list -> 'b list`; and one of `map f` to `xs`.

Thus far, currying may seem undermotivated: is it just a trivial syntactic thing? We will eventually discuss several advantages of curried functions:

- Today: It is easy to *partially apply* a curried function, e.g. you can write `map f`, rather than `fn l => map (f,l)`, when you want the function that maps `f` across a list. This will come up when you want to pass `map f` as an argument to another higher-order function, or when you want to compose it with other functions, as we will see below. This is a difference in readability, but not functionality.
- Another motivation for currying is that it lets you write programs that you wouldn't otherwise be able to write. But we won't get to this for a couple of lectures, when we talk about something called *staged computation*. Another example is *function-local state*, but we won't get to that until much later. The general idea is that currying lets you do real computation between the arguments to a function.

Because of this, there is special syntax for curried functions, where you write multiple arguments to a function in sequence, with spaces between them:

```
fun map (f : 'a -> 'b) (l : 'a list) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map f xs
```

This means the same as the explicit `fn` binding in the body, as above.

3.1 Fusion

If you have two maps in succession, this transforms a list, and then immediately transforms the resulting list again (and does nothing else with it). Thus, you might as well just compute the third tree right away, in one pass. This saves time (one traversal instead of two) and space (no need to create and then throw away the second tree).

In general, when `f` and `g` are total, we have

```
map f o map g == map (f o g)
```

which is called *fusion*. You'll look at the proof of this in HW. It is a generalization of the theorem about `raiseBy` from the lecture introducing lists.

4 Reduce

Consider the following two functions:

```
fun sum (l : int list) : int =
   case l of
        [] => 0
      | x :: xs => x + (sum xs)
fun join (l : string list) : string =
    case l of
        [] => ""
      | x :: xs => x ^ join xs
```

The pattern is "give some answer for the empty list, and for a cons, somehow combine the first element with the recursive call on the rest of the list." Here's is a curried version of `reduce` from lab:

```
fun reduce (c : 'a * 'a -> 'a) (n : 'a) (l : 'a list) : 'a =
    case l of
        [] => n
      | x :: xs => c (x , reduce c n xs)

fun sum (l : int list) : int = reduce (fn (x,y) => x + y) 0 l
fun join (l : string list) : string = reduce (fn (x,y) => x ^ y) "" l
```

5 Functional Decomposition of Problems

In functional programming, one way to think about problems is to decompose them into a series of tasks, represented as functions, and to solve the problem by composing these functions together.

Function composition is a builtin `o`, which is defined as follows:

```
fun (g : 'b -> 'c) o (f : 'a -> 'b) : 'a -> 'c =
  fn x => g (f x)
```

Function composition makes things more succinct. For example, the theorem on HW3 about `zip` and `unzip` is saying that `unzip o zip` $\cong$ `fn x => x`.

5.1 wordcount and longestline

For example, let's count all the words in a string. We can represent this as three tasks:

Start with a string

```
"hi there"
```

First, divide it up into a list of words:

```
["hi","there"]
```

Second replace each element with `1`

```
[1,1]
```

Third, sum up the list:

```
+ (1, 1)
```

Let's assume a function `words : string -> string list` for the first task. We can implement the second using `map (fn _ => 1) : string list -> int list`. And we defined `sum` above for the third.

Thus, we can implement

```
val wordcount : string -> int = sum o map (fn _ => 1) o words
```

by composing these tasks together.

Suppose we want to know the length of the longest line in a file:

```
val longestlinelength : string -> int =
   reduce Int.max 0  o  map wordcount  o  lines
```

Divide the file into lines, compute the number of words in each, and then take the max.

5.2 Stocks

Functional programming makes it easy to express computations as compositions of tasks, where the tasks themselves can often be expressed concisely using higher-order functions like `map` and `reduce`. This leads to short and elegant solutions to problems.

Suppose we have the prices for stocks for a stock over time:

Day 1	Day 2	Day 3	Day 4
$20	$25	$24	$30

Let's write a function `bestGain : int list -> int` that computes the best profit you could have made on the stock, were you to buy and sell on any two days. Here's the algorithm, on the above example:

1. Form pairs $(buy, sells)$, where buy is a price you could have bought at, and $sells$ is the prices you could then sell at:

   ```
   (20, [25,24,30])
   (25, [24,30])
   (24, [30])
   (30, [])
   ```

We can do this by pairing each price with the suffix of the data after that point.

2. For each pair $(buy, sells)$, for each element $sell$ of $sells$, compute the difference $sell - buy$:

```
[5,4,10]
[~1,5]
[6]
[]
```

3. Take the max of all of these. In this case, the answer is 10.

Note that the output of each step is the input to the subsequent step. This means we can implement this algorithm as a composition of three functions, one for each step.

We use the following helper functions:

```
(* See homework 3 *)
val zip : ('a list * 'b list) -> ('a * 'b) list

(* compute a list whose elements are all the suffixes of the input *)
val suffixes : 'a list -> ('a list) list
```

For step (1), we implement a function `withSuffixes` that pairs each price with the suffix of the list after that price.

```
fun withSuffixes (t : int list) : (int * int list) list = zip (t, suffixes t)
```

For step (3), we want to take the max of all of the `int`'s in a list of lists. This should be familiar from Lecture 1:

```
val maxL : int list -> int = reduce Int.max minint
val maxAll : (int list) list -> int = maxL o map maxL
```

Note that `minint` is an `int` that is $\leq$ all other `int`'s.

Now we can implement `bestGain` as a composition of three functions:

```
val bestGain : int list -> int =
      maxAll                                                          (* step 3 *)
    o (map (fn (buy,sells) => (map (fn sell => sell - buy) sells))) (* step 2 *)
    o withSuffixes                                                    (* step 1 *)
```

For step (1), we use `withSuffixes`.

For step (2), the input is the `(int * int tree) tree` resulting from step (1). We use a `map` to do something to each pair `(buy,sells)` (e.g. the pairs `(20, <25,24,30>)` and `(25, <24,30>)` ... in the above example). Note that this outer map is partially applied, because the argument to `o` is a function. The body of the outer `map` is again a call to `map`, time over `sells`, to do something to each `sell` price—in particular, to compute `sell - buy`. This leaves us with an `(int tree) tree`, whose leaves are all of the `sell - buy` differences.

For step (3), we use `maxAll`.

In case currying is confusing you, here is what the above example looks like if we uncurry everything:

```
fun map (f : 'a -> 'b, l : 'a list) : 'b list =
    case l of
        [] => []
      | x :: xs => f x :: map (f,xs)

fun reduce (c : 'a * 'a -> 'a, n : 'a, l : 'a list) : 'a =
    case l of
        [] => n
      | x :: xs => c (x , reduce (c, n, xs))

fun maxL (l : int list) : int = reduce (Int.max , minint , l)
fun maxAll (l : (int list) list) : int = maxL (map (maxL, l))

fun withSuffixes (l : int list) : (int * int list) list =
    zip (l, suffixes l)

fun bestGain (l : int list) : int =
    maxAll (map (fn (buy,sells) => (map (fn sell => sell - buy, sells)),
                 withSuffixes l))
```

6 Functions as Data

Now that we know that functions can be returned as results from other functions, we can start to put this idea to use. For example, returning to the polynomial example from before, we can represent a polynomial

```
c0 + c1 x + c2 x^2 + ... cn x^n
```

How can we represent such a table? As a function that maps each exponent to its coefficient:

```
   fn x => case x of
      0 => c0
    | 1 => c1
    | 2 => c2
    | ...
    | n => cn
    | _ => 0
```

This carries the same information as the coefficient list representation, but it also scales to series, which may have infinitely many terms.

Here's an example polynomial:

```
type poly = int -> int

(* x^2 + 2x + 1 *)
val example : poly =
     fn 0 => 1
      | 1 => 2
      | 2 => 1
      | _ => 0
```

The function that adds two polynomials takes two functions as input, and produces a function as output:

```
fun add (p1 : poly, p2 : poly) : poly = fn e => p1 e + p2 e
```

This is like Jeopardy: the answer of `add(p1,p2)` has the form of a question—what exponent would you like the coefficient of? In the case of addition, the answer is that the coefficient of `e` is the sum of the coefficients in the summand. Note that the function we return refers to the arguments `p1` and `p2`—this is called a *closure*, and we say that the function *closes over* `p1` and `p2`. When you call `add`, you generate a new function that mentions whatever polynomials you want to sum.

For multiplication, we need to do a *convolution*: the coefficient of e is

$$\sum_{i=0}^{e} c_i d_{e-i}$$

We render this in SML by writing a simple local, recursive helper function, which loops from `e` to `0`:

```
fun mult (c , d) =
    fn e => let
                fun convolution i =
                    case i of
                        ~1 => 0
                      | _ => (c i) * (d (e - i)) + convolution (i - 1)
            in
                convolution e
            end
```

More generally, we could represent a dictionary mapping `key`s to `value`s as functions `key -> value`.