15-150 Lecture 11:
Higher-Order Functions

Lecture by Dan Licata

September 29, 2011

1 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.

2 Higher-order functions on trees

In lab, we saw the following definition of trees:

```
datatype 'a tree =
    Empty
  | Leaf of 'a
  | Node of 'a tree * 'a tree
```

Last time, we saw `map` for lists. Here's `map` for trees, curried:

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

This applies `f` to the data at each `Leaf`: if we have a tree

```
val t = Node(Empty,Node(Leaf v1,Leaf v2))
```

then `map (f,t)` will compute

```
Node(Empty,Node(Leaf (f v1),Leaf(f v2)))
```

Another useful higher-order function is `reduce`. Consider

```
fun sum (t : int tree) : int =
    case t of
        Empty => 0
      | Leaf x => x
      | Node(t1,t2) => (sum t1) + (sum t2)

fun join (t : string tree) : string =
    case t of
        Empty => ""
      | Leaf x => x
      | Node(t1,t2) => join t1 ^ join t2
```

In each of these, we return something for `Empty`, and return the value at the leaf for `Leaf`, and do something to the two recursive results for `Node`. We can abstract this pattern as follows:

```
fun reduce (n : 'a * 'a -> 'a) (e : 'a) (t : 'a tree) : 'a =
   case t of
      Empty => e
    | Leaf x => x
    | Node (l,r) => n (reduce n e l , reduce n e r)
```

We recover the original functions as instances:

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

Because `reduce` is curried, these can also be written more simply using *partial application*: you can apply a curried function to only some of its arguments, resulting in a function of the remaining arguments:

```
val sum : int tree -> int = reduce (fn (x,y) => x + y) 0
val join : string tree -> int = reduce (fn (x,y) => x ^ y) ""
```

3 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` ≅ `fn x => x`.

3.1 wordcount and longestline

Spring, 2012: these were challenge problems in lab.

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 tree with one word at each `Leaf`:

```
Node (Leaf "hi", Leaf "there")
```

Second replace each leaf with `1`

```
Node (Leaf 1, Leaf 1)
```

Third, sum up the tree:

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

Let's assume a function `words : string -> string tree` for the first task. We can implement the second using `map (fn _ => 1) : string tree -> int tree`. 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.

3.2 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.

Last semester, we had a guest lecture from Jane Street Capital, a high-frequency trading firm that's a big commercial user of ML. 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 tree -> 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'll do it on a tree for parallelism, using the following helper functions:

```
(* cf. zip on lists in HW 3 *)
val zip ('a tree * 'b tree) -> ('a * 'b) tree

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

(* find the maximum element of a tree of trees of integers *)
val maxAll : (int tree) tree -> int
```

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

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

Now we can implement `bestGain` as a composition of three functions, feeding the output

```
val bestGain : int tree -> 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.

In step (3), we want to take the max of all of the `int`'s at the leaves of this tree. This should be familiar from Lecture 1:

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

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

3.3 Deforestation

Note, Fall 2012: This will be covered in lecture later, when we get to sequences.

Note that `maxAll` is basically

```
val maxAll : int tree tree -> int =
   reduce Int.max minint o map maxT
```

When we have a `reduce` right after a `map`, you'll notice that something kind of inefficient happens: in one pass, you build up a new tree with the transformed values. But the only thing you ever do with this tree is to consume it in the `reduce`. This takes extra time (two passes instead of one) and space (to store the new tree, which is immediately consumes).

A better solution is to combine a `map` and a `reduce` into one pass, yielding `mapreduce`:

```
fun mapreduce (l : 'a -> 'b) (e : 'b) (n : 'b * 'b -> 'b) (t : 'a tree) : 'b =
    case t of
        Empty => e
      | Leaf x => l x
      | Node (t1,t2) => n (mapreduce l e n t1, mapreduce l e n t2)
```

For example, `maxAll` can be written as

```
val maxAll : int tree tree -> int =
   mapreduce maxT Int.max minint
```

This is called *deforestation*, because you are eliminating trees.

All semester, we've been talking about templates for structural recursion. The template for `'a tree` says that you give three cases, one for empty, one for a leaf (in terms of its value), and one for a node (in terms of the two recursive results). But this is *exactly what `mapreduce` is*! It has one argument `e` for `Empty`, one argument `l` that tells you what to do with the value stored at a `Leaf`, and one `n` that tells you how to combine the recursive results on a `Node`.

That is, using higher-order functions, we can codify informal templates as actual code! The advantage of this is that we can build up new templates, new abstractions, as part of programming. We will exploit this a bunch later in the semester.

3.4 Fusion

An idea similar to deforestation is fusion: If you have two maps in succession, this transforms a tree, and then immediately transforms the resulting tree 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.

4 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
```

Another way to do it is using higher-order functions:

```
val sum = reduce (fn (x,y) => x + y) 0

(* upto n returns a tree whose leaves are <0,1,...,n> *)
fun upto n = tabulate (fn x => x) (n + 1)

fun mult (c : poly, d : poly) : poly =
    fn e => sum (map (fn i => c i * d (e - i)) (upto e))
```

Other examples Here are some other examples of functional data structures:

- A dictionary mapping `key`s to `value`s: `key -> value`
- ... where some keys may not occur:
- A dictionary mapping `key`s to `value`s: `key -> value option`
- Sets of `elt`s, as characteristic functions: `elt -> bool`.
- Shapes, from last time: `point -> bool`.