Notes for Lecture 10 (Fall 2022 week 5, part 2): Higher-order functions

Jana Dunfield

October 2, 2022

The code for this lecture is in `lec10.hs`.

1 Higher-order functions

Haskell is a functional programming language, which correctly suggests that functions are a central feature. Functions in Haskell are more powerful than functions in many other languages, though in recent years, some popular languages have adopted some of the features of Haskell's functions.

Two key features of Haskell's functions are (1) we can pass functions as arguments, and (2) we can return new functions. These features make Haskell functions *higher-order*.

We'll start with the first of these features.

1.1 Functions as arguments

Consider some functions on lists. One of them, `double_list`, takes a list of integers and returns a list with every element multiplied by 2.


```
double_list :: [Integer] -> [Integer]
double_list []       = []
double_list (x : xs) = (x * 2) : (double_list xs)
```


Given the empty list `[]`, we return the empty list. Given a list whose head is `x` and whose tail is `xs`, we return a new list whose head is `x * 2` and whose tail is the tail of the given list with every element multiplied by 2.

For example, `double_list [3, 1, 2]` returns `[6, 2, 4]`.

Another function, `triple_list`, is similar but multiplies elements by 3:

```
triple_list :: [Integer] -> [Integer]
triple_list []       = []
triple_list (x : xs) = (x * 3) : (triple_list xs)
```

We can generalize these to return a list with every element multiplied by some number `k`.

```
multiply_list :: Integer -> [Integer] -> [Integer]
multiply_list k []       = []
multiply_list k (x : xs) = (x * k) : (multiply_list k xs)
```

For example, `multiply_list 2 [3, 1, 2]` returns `[6, 2, 4]`, just like `double_list`; `multiply_list 0 [3, 1, 2]` returns `[0, 0, 0]`, because every element gets multiplied by zero.

We could have written `multiply_list` to take `k` as the *second* argument. The advantage of having `k` be the first argument is that we can conveniently specialize `multiply_list` to specific `k`:

```
quadruple_list :: [Integer] -> [Integer]
quadruple_list = multiply_list 4

-- (equivalent:  quadruple_list xs = multiply_list 4 xs)
```

We can write a function that adds a number to every element in a list (that is, returns a new list with a number added to each element of a given list).

```
add_to_list :: Integer -> [Integer] -> [Integer]
```

Before I show the code, think about how you would write this function, with `multiply_list` as a model.

I wrote add_to_list by copying and pasting multiply_list, renaming, and changing * to +.

```
add_to_list :: Integer -> [Integer] -> [Integer]
add_to_list k []       = []
add_to_list k (x : xs) = (x + k) : (add_to_list k xs)
```

For example, add_to_list 4 [100, 200] returns 104, 204.

We started with double_list and triple_list, noticed that the definitions of each were very similar, and generalized to multiply_list. We generalized by passing the number 2 or 3 as an argument, rather than writing it within each function definition. The function multiply_list returns a list with each element multiplied by a number; the function add_to_list returns a list with a number added to each element.

```
...
multiply_list k (x : xs) = (x * k) : (multiply_list k xs)
...
add_to_list   k (x : xs) = (x + k) : (add_to_list k xs)
```

If we try to abstract over these two functions, we get: "a function something_list returns a list that has had something done with each element". We generalize by passing the "something" that is "done" as an argument: we pass a *function* as an argument.

It is traditional to call the function we are about to write (not the function being passed as an argument, the function that takes a function as an argument) "map". Haskell defines a function with this name already, so we will call our version mymap. It is also traditional to call "something" f.


```
mymap :: (Integer -> Integer) -> [Integer] -> [Integer]
mymap f []       = []
mymap f (x : xs) = (f x) : (mymap f xs)
```

The second clause applies f (whatever it is) to the head x.

We can now recover the behaviour of multiply_list 9 with:

```
multiply_list_by_9 :: [Integer] -> [Integer]
multiply_list_by_9 = mymap (\y -> y * 9)
```

And we can define a function that subtracts a given number from each element:

```
subtract_from_list :: Integer -> [Integer] -> [Integer]
subtract_from_list k = mymap (\x -> x - k)
```

In these functions I passed lambdas (anonymous functions) to mymap. Lambdas are often convenient as arguments to higher-order functions: a single operation like subtracting an integer is probably not worth a named function declaration.

The following exercise will lead into lec11, on *polymorphism*.

Exercise 1. Comment out the type declaration for `mymap` in `lec10.hs`, and reload the file.

What type do you get for `mymap`?

Can you pass `(\x -> x > 0)` as the first argument to `mymap`?

Can you pass the Boolean negation function `not` as the first argument `mymap`? What does the second argument need to be?

Can you pass `(\s -> "A" ++ s ++ "Z")` as the first argument to `mymap`?