Notes on Functional Programming

Foundations of Computer Science

February 11, 2016

Mapping

Code the following functions using explicit recursion:

- *A function* `squares` *that squares every element of a list:*

  ```
  squares : int list -> int list
  ```

  Example: `squares [1;2;3]` *should return* `[1; 4; 9]`

- *A function* `diags` *that creates a pair of two same values out of every element of a list:*

  ```
  diags : 'a list -> ('a * 'a) list
  ```

  Example: `diags [1;2;3]` *should return* `[(1,1); (2,2); (3,3)]`

Here is one way to write those functions:

```
let rec squares xs =
  match xs with
    [] -> []
  | x :: xs' -> (x*x)::squares xs'

let rec diags xs =
  match xs with
    [] -> []
  | x :: xs' -> (x,x)::diags xs'
```

Rewrite the previous functions using an explicit call to a helper function that does the "work" for every element of the list.

For example, for `squares`*:*

```
let square x = x * x

let rec squares xs =
  match xs with
    [] -> []
  | x :: xs' -> (square x)::squares xs'
```

Pretty straightforward:

```
let diag x = (x,x)

let rec diags xs =
  match xs with
    [] -> []
  | x :: xs' -> (diag x)::diags xs'
```

If you study your rewritten functions, you notice that they all have the same structure. In fact, if we replace the recursive function name by `YYY` and the helper function name by `ZZZ`, they all have the structure:

```
let rec YYY xs =
  match xs with
    [] -> []
  | x :: xs' -> (ZZZ x) :: YYY xs'
```

A good programming language should give you the ability to abstract away from a set of examples, exposing an underlying commonality, and giving you a way to express that commonality using an abstraction that lets you recover the original examples by instantiating that abstraction.

We can write a function that captures the above structure, and obtain the original functions as special cases of that *über* function.

The function is called `map`, and it takes as extra argument the *function* to be applied to every element of the list.

```
let rec map (f,xs) =
  match xs with
    [] -> []
  | x :: xs' -> (f x)::(map (f,xs'))
```

To use `map`, you need to pass it a function as a first argument. Thankfully, we have plenty of functions around, and we can recover our original functions:

```
let squares xs = map (square,xs)

let diags xs = map (diag,xs)
```

Passing functions around like that is made possible by the fact that functions in OCaml (and in many other languages) are just like any other value: they can be passed around to other functions, they can be returned from functions, they can be operated on.

We have been using functions by giving them names, and passing those names around. This is fine, but imagine what it would be like if every time you wanted to use, say, an integer, you had to name it. That is, if instead of writing `1+2`, you had to write

```
let one = 1 in
let two = 2 in
  one + two
```

That'd get old fast. You can use integers without naming them. It is also possible to use functions without naming then. These are called *anonymous functions*, and they are created in OCaml with the syntax:

```
fun x -> <expr>
```

where `<expr>` is an expression representing the body of the function. Thus, for example, we can rewrite our functions using `map` and anonymous functions as follows:

```
let squares xs = map ((fun x -> x * x), xs)

let diags xs = map ((fun x -> (x,x)), xs)
```

It tends to make OCaml happier when you wrap anonymous functions in parentheses, like `(fun x -> x * x)`, because it sometimes gets confused trying to figure out where the function body ends.

In fact, the function definition notation we have been using, such as:

```
let square x = x * x
```

is just a convenient abbreviation for:

```
let square = (fun x -> x * x)
```

which illustrates that `fun x -> x * x` is a value like any other, to which we happen to give name `square`. Understanding this, that functions are just values that can be given a name if we want to, is key to understanding all that follows.

Code the following functions using `map`*:*

- *A function* `triples` *that creates a triple* $(i, i+1, i+2)$ *for every element* i *in a list:*

```
triples : int list -> (int * int * int) list
```

Example: `triples [0; 10; 20]` *should return* `[(0,1,2); (10,11,12); (20,21,22)]`

- *A function* `thirds` *that extracts the third component of every triple in a list:*

  ```
  thirds : ('a * 'b * 'c) list -> 'c list
  ```

  Example: `thirds [(1,2,3); (4,5,6); (7,8,9)]` *should return* `[3; 6; 9]`

Here's the simplest way using anonymous functions:

```
let triples xs = map ((fun x -> (x,x+1,x+2)), xs)

let thirds xs = map ((fun (a,b,c) -> c), xs)
```

Let's pump up the difficulty a notch.

Code the following function using `map`*:*

- *A function* `distribute` *creating tuples with the same given element as first component from items in a list:*

  ```
  distribute : 'a * 'b list -> ('a * 'b) list
  ```

  Example: `distribute (1, ["a"; "b"; "c"])` *should return* `[(1,"a"); (1,"b"); (1,"c")]`

The easiest way is to use an anonymous function:

```
let distribute (a,xs) = map ((fun x -> (a,x)), xs)
```

But what if we wanted to give the anonymous function a name, and define it before using it in `map`? That's the tricky bit. If we define it locally inside `distribute`, it's still easy:

```
let distribute (a,xs) =
  let mkPair x = (a,x) in
  map (mkPair,xs)
```

But what if, mischievously, we wanted to define the helper function `mkPair` *outside* of `distribute`? Can we do it? Try, go ahead.

When you try to write and use such a `mkPair`, you see that it gets called by `map` with an element of the list. There is no way to tell `map` to pass `a` as well, unless we change the definition of `map`, somehow.

But we can pull a trick, by remembering that *functions are values* and therefore can be returned from other functions. We use a function `createMkPair` that takes the `a` and creates the appropriate `mkPair` function *that works with that* `a` and that can be passed into `map`:

```
let createMkPair a =
  fun x -> (a,x)

let distribute (a,xs) = map (createMkPair a, xs)
```

Function `createMkPair` returns a new function, and it is *that* new function that we give as an argument to `map` (via the call `createMkPair a`).[1]

Note the type of `createMkPair`: it has type `'a -> 'b -> 'a * 'b`. To understand this type, you have to know that `->` associates to the right. So `'a -> 'b -> 'a * 'b` is to be understood as `'a -> ('b -> 'a * 'b)`, and this tells you the whole story: it is a function that expects a value of type `'a` and returns a function of type `'b -> 'a * 'b`, that is, a function that expects a value of type `'b` and returns a tuple of type `'a * 'b`.

Functions returning a new function are rather common, and OCaml has some nice notation for them that lets us "fake" having multiple argument functions. (We used tuples earlier to fake multiple argument functions—this provides an alternative.)

The function definition

```
let add x y = x + y
```

is just an abbreviation for

```
let add x = fun y -> x + y
```

To call such a function, remember that it is a function that expects one argument and returns a function that itself returns one argument, and thus we need to write something like:

```
(add 1) 2
```

[1]Nomenclature trivia. The function returned from `createMkPair` refers to the `a` that is passed as an argument to `createMkPair`. In order for this to make sense, the system has to remember the value for the `a` that was passed in when it returns the function. Internally, the system does this by associating an environment with the returned function containing the values for the free variables in the function. A function and its associated environment is usually called a *closure*.

(which evaluates to `3`, of course). Because application associates to the left, we can in fact simply write

```
add 1 2
```

which pleasantly reflects the shape of the function definition, `let add x y = x + y`.

Again: `add` looks like it's a two argument function, but really, it's a function returning a function. Its type makes that clear: it is `int -> int -> int`.[2]

Thus, in our `distribute` example, we can define

```
let createMkPair a x = (a,x)

let distribute (a,xs) = map (createMkPair a, xs)
```

The curried notation extends. For instance,

```
let add x y z = x+y+z
```

is an abbreviation for

```
let add x = fun y -> (fun z -> x+y+z)
```

and of course, it is also equivalent to

```
let add x y = fun z -> x+y+z
```

and in fact if we remember that defining functions in the first place is an abbreviation, the three definitions above are also equivalent to:

```
let add = fun x -> (fun y -> (fun z -> x+y+z))
```

The curried notation also extends to anonymous functions, so that `fun x y -> <expr>` is really an abbreviation for `fun x -> (fun y -> <expr>)`, and thus the above definitions for `add` are also equivalent to:

```
let add = fun x y z -> x+y+z
```

You get the gist.

So what's the difference between writing `add` as a curried function versus having it take a tuple as an argument? Compare:

```
let add (x,y,z) = x+y+z
```

[2] More nomenclature trivia. A function such as `add`, written in such a way that it looks like a multi-argument function but really takes them one after the other in a cascade of functions, is called a *curried* function—named after the logician Haskell Curry.

If you try to call `add (1,2)`, you will get a type error:

```
# add (1,2);;
Characters 4-9:
  add (1,2);;
      ^^^^^
Error: This expression has type 'a * 'b
       but an expression was expected of type int * int * int
```

whereas if you define

```
let add x y z = x+y+z
```

and you pass only two arguments to `add`:

```
# add 1 2;;
- : int -> int = <fun>
```

No error at all. You get a function back. Which makes perfect sense. (What does the function you get back do, though?) Of course, if you then try to use the result as an integer, such as trying to evaluate `10+(add 1 2)`, then OCaml will complain with an error message that you're trying to add an integer to a function.

Most functions in OCaml tend to be written in curried form. From now, I shall do so as well.

Note that the curried version of `map` is available int he OCaml library as `List.map`.

Let's finish with an example showing both functions passed as arguments and returned as results.

First, define the following curried functions:

```
let add n m = n + m
let mult n m = n * m
```

Easy enough. Partial application, that is, giving less than the full number of arguments to a curried function, leads to some interesting behavior. For instance, `add 5` gives you back a function that always adds `5` to its input; `mult 6` gives you back a function that always multiplies its input by `6`:

```
# let f = add 5;;
val f : int -> int = <fun>
# f 10;;
- : int = 15
# let g = mult 6;;
val g : int -> int = <fun>
```

```
# g 10;;
- : int = 60
```

We can write a functional *composition* operator that takes two functions f and g and composes them together into a single function, corresponding to the mathematical operation $g \circ f$.

```
let compose g f = fun x -> g (f x)
```

Function `compose` has type `('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)`

Thus, for example, `compose (add 5) (mult 6)` gives you back a function (what does it do?) and when you apply that function to `3`, you get back `23`:

```
# let w = compose (add 5) (mult 6);;
val w : int -> int = <fun>
# w 3;;
- : int = 23
# w 10;;
- : int = 65
```

Of course, the definition of `compose` above is equivalent to:

```
let compose g f x = g (f x)
```

Filtering

Code the following functions:

- *A function* `filter` *that takes a predicate and a list and returns the list of elements that satisfy the predicate:*

  ```
  filter : ('a -> bool) -> 'a list -> 'a list
  ```

  E.g., `filter (fun x -> x>0) [0;1;-2;3;-4;5]` *should return* `[1;3;5]`

- *A function* `removeEmpty` *that takes a list of sublists and returns the list of all non-empty sublists :*

  ```
  removeEmpty : 'a list list -> 'a list list
  ```

  E.g., `removeEmpty [[1;2]; []; [3]; []]` *should return* `[[1;2];[3]]`

Function `filter` is a straightforward recursive function over lists:

```
let rec filter p xs =
  match xs with
    [] -> []
  | x :: xs' -> if (p x) then x :: (filter p xs')
                else filter p xs';;
```

Function `removeEmpty` can be defined directly as an explicitly recursive function, but it can also be defined in terms of `filter`:

```
let removeEmpty xss =
  filter (fun xs -> match xs with [] -> false | _ -> true) xss
```

Note that `filter` is available from the OCaml library as `List.filter`.

Can you implement `filter` *directly using* `map`*?*

In other words, can you define

```
  let filter p xs = map ...
```

If so, do it. If not, why not?

There is no way for `map` by itself to be able to express `filter`, because `map` has the property that the list it returns has always the same size as the list it is passed as argument. Function `filter`, on the other hand, can potentially shrink the size of the list.

Now, `map` by itself cannot implement `filter`, but a slight variant of `map` can:

```
let rec map_append f xs =
  match xs with
    [] -> []
  | x :: xs' -> (f x) @ (map_append f xs')
```

Intuitively, while `map f [x;y;z]` returns the list `[f x; f y; f z]`, the call `map_append g [x;y;z]` returns the list `[x1; x2; ...; xm; y1; y2; ...; yn; z1; z2; ... zp]`, where `g x = [x1; x2; ...; xm]`, `g y = [y1; y2; ...; yn]`, and `g z = [z1; z2; ...; zp]`.

Code the following functions using `map_append`*:*

- *A function* `flatten` *that takes a list of sublists and returns a new list with all the sublists' elements in it, in order:*

  ```
  flatten : 'a list list -> 'a list
  ```

  E.g., `flatten [[1; 2]; [3; 4]; [5]; []; [6; 7]]` *should return* `[1; 2; 3; 4; 5; 6; 7]`.

- *The function* `filter` *above.*

These functions are directly implemented using `map_append`:

```
let flatten xs = map_append (fun x -> x) xs

let rec filter p xs =
  map_append (fun x -> if (p x) then [x] else []) xs
```

Note that `flatten` uses the identity function `fun x -> x` as the transformation.[3]

Functions `map` and `map_append` have a lot in common. Can we make precise what they have in common and write a single function that can do whatever `map` and `map_append` can do?

Here is the code for `map` and `map_append`, next to each other:

```
  let rec map f xs =
    match xs with
      [] -> []
    | x :: xs' -> (f x) :: (map f xs')

  let rec map_append f xs =
    match xs with
      [] -> []
    | x :: xs' -> (f x) @ (map_append f xs')
```

We see they both have the structure:

```
  let rec MMM f xs =
    match xs with
      [] -> []
    | x :: xs' -> (f x) XXX (MMM f xs')
```

where `XXX` is the combination function, `::` for `map`, and `@` for `map_append`. We can replace that `XXX` by a function that is passed as an argument, and we obtain `map_general_1`:

```
let rec map_general_1 comb f xs =
  match xs with
```

[3]That `flatten` is `map_append` with an identity function as transformation suggests that there is a special relationship between `map_append` and `flatten`. Indeed, if instead of defining `map_append`, we had defined `flatten` directly, we could derive `map_append` as follows:

```
let map_append f xs = flatten (map f xs)
```

```
    [] -> []
  | x :: xs' -> comb (f x) (map_general_1 comb f xs')
```

and we can now write, as desired:

```
let map f xs = map_general_1 (fun x ys -> x :: ys) f xs

let map_append f xs = map_general_1 (fun x ys -> x @ ys) f xs
```

In fact, passing in both `comb` and `f` as arguments to `map_general_1` is unnecessary: a suitable combination function can play the role of both `comb` and `f`:

```
let rec map_general comb xs =
  match xs with
    [] -> []
  | x :: xs' -> comb x (map_general comb xs')
```

and to see that we have not lost any generality, we can implement the original `map_general_1` using `map_general`:

```
let map_general_1 comb f xs =
  map_general (fun x ys -> comb (f x) ys) xs
```

Folding

Can we go even more general? If you look carefully at `map_general`, or feed it to OCaml, you see it has type

```
('a -> 'b list -> 'b list) -> 'a list -> 'b list
```

In particular, the result of calling `map_general` is always a list.

Now, there are functions on lists that do not return lists, but still have a lot in common with `map_general`. For instance, `sum`, with its obvious recursive definition:

```
let add m n = m + n

let rec sum xs =
  match xs with
    [] -> 0
  | x :: xs' -> add x (sum xs')
```

where I use `add` in `sum` to emphasize the commonality with `map_general`.

Look at the code for `map_general` and `sum` next to each other. One difference is that `sum` embeds its combination function `add` directly in the code, which is fine. We can imagine pulling it out as an argument:

```
let rec sum_general comb xs =
  match xs with
    [] -> 0
  | x :: xs' -> comb x (sum_general comb xs')
```

Now, the code for `map_general` and `sum_general` are very similar, and share the following structure:

```
let rec FFF comb xs =
  match xs with
    [] -> YYY
  | x :: xs' -> comb x (FFF comb xs')
```

where `YYY` is the difference between the functions, namely the value returned by the function on an empty list. So we can do as we did for `map_general`, and simply make that difference a parameter to the function. We get the following function, with one of its common names:

```
let rec fold_right comb xs base =
  match xs with
    [] -> base
  | x :: xs' -> comb x (fold_right comb xs' base)
```

(That function is also sometimes called `reduce`.) After a moment's thought, we see that:

```
let map_general comb xs = fold_right comb xs []

let sum xs = fold_right add xs 0
```

Function `fold_right` is very interesting. It really does nothing except recurse. Everything else is delegated to the functions given as arguments. It is the *essence* of structural recursion on lists. Most recursive functions on lists, as long as they recurse on the tail of the list, can be implemented in term of `fold_right` for the appropriate combination function and base value.

Code the following functions using `fold_right`*:*

- *The function* `removeEmpty` *we saw earlier:*

  ```
  removeEmpty : 'a list list -> 'a list list
  ```

- *A function* `heads` *that takes a list of sublists and return the first element of each sublist, skipping over empty lists:*

  ```
  heads : 'a list list -> 'a list
  ```

  E.g., `heads [[1;2];[];[3]]` *should return* `[1;3]`*.*

- *A function* `concat` *that takes a list of strings and returns the result of concatenating all the strings together in the order in which they appear in the list.*

```
concat : string list -> string
```

  E.g., `concat ["goodbye ","cruel ";"world"]` *should return* `"goodbye cruel world"`.

At first, the trick is to first write down the function using explicit recursion, and then read off the appropriate combination function and base value. With practice, you can then come up with them directly:

```
let removeEmpty xs =
  fold_right (fun x r -> match x with [] -> r | _ -> x::r)
             xs []

let heads xs =
  fold_right (fun x r -> match x with [] -> r | y::_ -> y::r)
             xs []

let concat xs =
  fold_right (fun x r -> x^r) xs ""
```