8
Higher-order functions and functional programming

We’ve laid the groundwork for programming with functions in Chapter 6, and provided some useful structures for data in Chapter 7, especially lists. In this chapter we show how higher-order functions serve as a mechanism to satisfy the edict of irredundancy. By examining some cases of similar code, we will present the use of higher-order functions to achieve the abstraction, in so doing presenting some of the most well known abstractions of higher-order functional programming on lists – map, fold, and filter.

8.1 The map abstraction

In Exercises 47 and 48, you wrote functions to increment and to square all of the elements of a list. After solving the first of these exercises with

```
# let rec inc_all (xs : int list) : int list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> (1 + hd) :: (inc_all tl) ;;
val inc_all : int list -> int list = <fun>
```

you may have thought to cut and paste the solution, modifying it slightly to solve the second:

```
# let rec square_all (xs : int list) : int list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> (hd * hd) :: (square_all tl) ;;
val square_all : int list -> int list = <fun>
```

These “apparently dissimilar” pieces of code bear a striking resemblance, a result of the cutting and pasting. And to the extent that they echo the same idea, we’ve written the same code twice, violating the edict of irredundancy. Can we view them abstractly as “instantiating an underlying identity”?

The differences between these functions are localized in their last lines, where they compute the head of the output list from the head

of the input list – in `inc_all` as `1 + hd`, in `square_all` as `hd * hd`. But the redundancies here are not merely the use of different values (as they were in Section 6.5), but different *computations* over values. Do we have a tool to characterize these different computations, what is done to the head of the input list in each case? Yes, the function! In `inc_all`, we are essentially applying the function `fun x -> 1 + x` to the head, and in `square_all`, the function `fun x -> x * x`. We can make this clearer by rewriting the two snippets of code as explicit applications of a function.

```
let rec inc_all (xs : int list) : int list =
  match xs with
  | [] -> []
  | hd :: tl -> (fun x -> 1 + x) hd :: (inc_all tl) ;;

let rec square_all (xs : int list) : int list =
  match xs with
  | [] -> []
  | hd :: tl -> (fun x -> x * x) hd :: (square_all tl) ;;
```

Now, we can take advantage of the fact that in OCaml functions are first-class values, which can be used as arguments or outputs of functions, to construct a single function that performs this general task of applying a function, call it `f`, to each element of a list. We add `f` as a new argument and replace the different functions being applied to `hd` with this `f`. Historically, this abstract pattern of computation – performing an operation on all elements of a list – is called a MAP. We capture it in a function `map` that abstracts both `inc_all` and `square_all`.

```
# let rec map (f : int -> int) (xs : int list) : int list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> f hd :: (map f tl) ;;
val map : (int -> int) -> int list -> int list = <fun>
```

The `map` function takes two arguments (curried), the first of which is itself a function, to be applied to all elements of its second integer list argument. Its type is thus `(int -> int) -> int list -> int list`. With `map` in hand, we can perform the equivalent of `inc_all` and `square_all` directly.

```
# map (fun x -> 1 + x) [1; 2; 4; 8] ;;
- : int list = [2; 3; 5; 9]
# map (fun x -> x * x) [1; 2; 4; 8] ;;
- : int list = [1; 4; 16; 64]
```

In fact, `map` can even be used to define the functions `inc_all` and `square_all`.

```
# let inc_all (xs : int list) : int list =
#   map (fun x -> 1 + x) xs ;;
val inc_all : int list -> int list = <fun>
# let square_all (xs : int list) : int list =
#   map (fun x -> x * x) xs ;;
val square_all : int list -> int list = <fun>
```

These definitions of `inc_all` and `square_all` don't suffer from the violation of the edict of irredundancy exhibited by our earlier ones. By abstracting out the differences in those functions and capturing them in a single higher-order function `map`, we've simplified each of the definitions considerably.

But making full use of higher-order functions as an abstraction mechanism allows even further simplification, via partial application.

8.2 Partial application

Although we traditionally think of functions as being able to take more than one argument, in OCaml functions always take exactly one argument. Here, for instance, is the `power` function, which appears to take two arguments, an exponent n and a base x, and returns x^n:

```
# let rec power (n, x) =
#   if n = 0 then 1
#   else x * power ((n - 1), x) ;;
val power : int * int -> int = <fun>
# power (3, 4) ;;
- : int = 64
```

Though it appears to be a function of two arguments, "desugaring" makes clear that there is really only one argument. First, we desugar the `let`:

```
let rec power =
  fun (n, x) ->
    if n = 0 then 1
    else x * power ((n - 1), x) ;;
```

and then desugar the pattern match in the `fun`:

```
let rec power =
  fun arg ->
    match arg with
    | (n, x) -> if n = 0 then 1
                else x * power ((n - 1), x) ;;
```

demonstrating that all along, `power` was a function (defined with `fun`) of one argument (now called `arg`).

How about this definition of `power`?

```
# let rec power n x =
#   if n = 0 then 1
```

```
#   else x * power (n - 1) x ;;
val power : int -> int -> int = <fun>
# power 3 4 ;;
- : int = 64
```

Again, desugaring reveals that all of the functions in the definition take a single argument.

```
let rec power =
  fun n ->
    fun x ->
      if n = 0 then 1
      else x * power (n - 1) x ;;
```

As described in Section 6.2, we use the term "currying" for encoding a multi-argument function using nested, higher-order functions, as this latter definition of `power`. In OCaml, we tend to use curried functions, rather than uncurried definitions like the first definition of `power` above; the whole language is set up to make that easy to do.

We can use the `power` function to define a function to cube numbers (take numbers to the third power):

```
# let cube x = power 3 x ;;
val cube : int -> int = <fun>
# cube 4 ;;
- : int = 64
```

But since `power` is curried, we can define the cube function even more simply, by applying the power function to its "first" argument only.

```
# let cube = power 3 ;;
val cube : int -> int = <fun>
# cube 4 ;;
- : int = 64
```

A perennial source of confusion is that in this definition of the `cube` function by partial application, no overt argument of the function appears in its definition. There's no `let cube x = ...` here. The expression `power 3` is already a function (of type `int -> int`). It *is* the cubing function, not just the result of applying the cubing function. This is PARTIAL APPLICATION: the applying of a curried function to only *some* of its arguments, resulting in a function that takes the remaining arguments.

The order in which a curried function takes its arguments thus becomes an important design consideration, as it determines what partial applications are possible. With partial application at hand, we can define other functions for powers of numbers. Here's a version of `square`:

```
# let square = power 2 ;;
val square : int -> int = <fun>
```

```
# square 4 ;;
- : int = 16
```

Understanding what's going on in these examples is a good indication that you "get" higher-order functional programming. So we pause for a little practice with partial application.

Exercise 51

A TESSERACT is the four-dimensional analog of a cube, so fourth powers of numbers are sometimes referred to as TESSERACTIC NUMBERS. Use the `power` function to define a function `tesseract` that takes its integer argument to the fourth power.

Now, `map` is itself a curried function and therefore can itself be partially applied to its first argument. It takes its function argument and its list argument one at a time, and applying it only to its first argument generates a function that applies that argument function to all of the elements of a list. We can partially apply map to the increment function to generate the `inc_all` function we had before.

```
# let inc_all = map (fun x -> 1 + x) ;;
val inc_all : int list -> int list = <fun>
```

But there are even further opportunities for partial application.[1] The addition function `(+)` itself is curried, as we noted in Section 6.2. It can thus be partially applied to one argument to form the increment function: `(+) 1`. (Recall the use of parentheses around the + operator in order to allow it to be used as a normal prefix function.) Notice how the types work out: Both `fun x -> 1 + x` and `(+) 1` have the same type, namely, `int -> int`. So the definition of `inc_all` can be expressed simply is as

[1] Partial application takes full advantage of the first-class nature of functions to enable compact and elegant definitions of functions. However, you should be aware that it does make type inference more difficult in the presence of polymorphism, an advanced topic discussed in Section 9.7 for the adventurous.

```
# let inc_all = map ((+) 1) ;;
val inc_all : int list -> int list = <fun>

# inc_all [1; 2; 4; 8] ;;
- : int list = [2; 3; 5; 9]
```

Similarly, `square_all` can be written as the mapping of the `square` function:

```
# let square_all = map square ;;
val square_all : int list -> int list = <fun>

# square_all [1; 2; 4; 8] ;;
- : int list = [1; 4; 16; 64]
```

Compare this to the original definition of `square_all`:

```
# let rec square_all (xs : int list) : int list =
#   match xs with
#   | [] -> []
#   | hd :: tl -> (hd * hd) :: (square_all tl) ;;
val square_all : int list -> int list = <fun>
```

Exercise 52

Use the `map` function to define a function `double_all` that takes an `int list` argument and returns a list with the elements doubled.

8.3 *The fold abstraction*

Let's take a look at some other functions that bear a striking resemblance. Exercises 44 and 45 asked for definitions of functions that took, respectively, the sum and the product of the elements in a list. Here are some possible solutions, written in the recursive style of Chapter 7:

```
# let rec sum (xs : int list) : int =
#   match xs with
#   | [] -> 0
#   | hd :: tl -> hd + (sum tl) ;;
val sum : int list -> int = <fun>

# let rec prod (xs : int list) : int =
#   match xs with
#   | [] -> 1
#   | hd :: tl -> hd * (prod tl) ;;
val prod : int list -> int = <fun>
```

As before, note the striking similarity of these two definitions. They differ in just two places (highlighted above): an initial value to return on the empty list and the operation to apply to the next element of the list and the recursively processed suffix of the list.

This abstract pattern of computation – combining all of the elements of a list one at a time with a binary function, starting with an initial value – is called a FOLD. We repeat the abstraction process from the previous section, defining a function called `fold` to capture the abstraction.

```
# let rec fold (f : int -> int -> int)
#              (xs : int list)
#              (init : int)
#            : int =
#   match xs with
#   | [] -> init
#   | hd :: tl -> f hd (fold f tl init) ;;
val fold : (int -> int -> int) -> int list -> int -> int = <fun>
```

Notice the two additional arguments – `f` and `init` – which correspond exactly to the two places that `sum` and `prod` differed.[2] In summary, the type of `fold` is `(int -> int -> int) -> int list -> int -> int`.

The `fold` abstraction is simply the repeated embedded application of a binary function, starting with an initial value, to all of the elements of a list. That is, given a list of n elements `[x_1, x_2, x_3, ..., x_n]`, the fold of a binary function `f` with initial value `init` is

[2] Ideally, these two arguments – `f` and `init` – would be placed as the first two arguments of `fold` so that they could be conveniently partially applied. (In fact, the Haskell functional programming language uses that argument order for their fold functions.) By convention, however, the argument order for this fold operation in OCaml is as provided here, allowing for partially applying the `f` argument but not `init`. The `init` argument is placed at the end to reflect its use as the rightmost element being operated on during the fold. As you'll see later, the alternative `fold_left` function uses the Haskell argument order.

```
f x_1 (f x_2 (f x_3 ( ··· (f x_n init)···)))    .
```

Now `sum` can be defined using `fold`:

```
# let sum lst =
#   fold (fun x y -> x + y) lst 0 ;;
val sum : int list -> int = <fun>
```

or, noting that + is itself the curried addition function we need as the first argument to `fold`:

```
# let sum lst = fold (+) lst 0 ;;
val sum : int list -> int = <fun>
```

The `prod` function, similarly, is a kind of fold, this time of the product function starting with the multiplicative identity 1.

```
# let prod lst = fold ( * ) lst 1 ;;
val prod : int list -> int = <fun>
```

A wide variety of list functions follow this pattern. Consider taking the length of a list, a function from Section 7.3.1.

```
let rec length (lst : int list) : int =
  match lst with
  | [] -> 0
  | _hd :: tl -> 1 + length tl ;;
```

This function matches the fold structure as well. The initial value, the length of an empty list, is `0`, and the operation to apply to the head of the list and the recursively processed tail is to simply ignore the head and increment the value for the tail.

```
# let length lst = fold (fun _hd tlval -> 1 + tlval) lst 0 ;;
val length : int list -> int = <fun>
#
# length [1; 2; 4; 8] ;;
- : int = 4
```

The function that we've called `fold` operates "right-to-left" producing

```
f x_1 (f x_2 (f x_3 ( ··· (f x_n init)···)))    .
```

For this reason, it is sometimes referred to as `fold_right`; in fact, that is the name of the corresponding function in OCaml's `List` module. The symmetrical function `fold_left` operates left-to-right, calculating

```
(f ··· (f (f (f init x_1) x_2) x_3) x_n)    .
```

where `init` is as before an initial value, and `f` is a binary function taking as arguments the recursively processed *prefix* and the next element in the list.

Exercise 53

Define the higher-order function `fold_left : (int -> int -> int) -> int -> int list -> int`, which performs this left-to-right fold.

Because addition is associative, a list can be summed by either a `fold_right` as above or a `fold_left`. The definition analogous to the one using `fold_right` is

```
# let sum lst = fold_left (+) 0 lst ;;
val sum : int list -> int = <fun>
```

but (because the list argument of `fold_left` is the final argument) this can be further simplified by partial application:

```
# let sum = fold_left (+) 0 ;;
val sum : int list -> int = <fun>
```

Exercise 54

Define the `length` function that returns the length of a list, using `fold_left`.

Exercise 55

A cousin of the `fold_left` function is the function `reduce`,[3] which is like `fold_left` except that it uses the first element of the list as the initial value, calculating

```
(f ⋯ (f (f x_1 x_2) x_3) x_n)    .
```

Define the higher-order function `reduce : (int -> int -> int) -> int list -> int`, which works in this way. You might define `reduce` recursively as we did with `fold` and `fold_left` or nonrecursively by using `fold_left` itself. (By its definition `reduce` is undefined when applied to an empty list, but you needn't deal with this case where it's applied to an invalid argument.)

[3] The higher-order functional programming paradigm founded on functions like `map` and `reduce` inspired the wildly popular Google framework for parallel processing of large data sets called, not surprisingly, MapReduce (Dean and Ghemawat, 2004).

8.4 *The filter abstraction*

The final list-processing abstraction we look at is the FILTER, which serves as an abstract version of functions that return a subset of elements of a list, such as the following examples, which return the even, odd, positive, and negative elements of an integer list.

```
# let rec evens xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> if hd mod 2 = 0 then hd :: evens tl
#                 else evens tl ;;
val evens : int list -> int list = <fun>

# let rec odds xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> if hd mod 2 <> 0 then hd :: odds tl
#                 else odds tl ;;
val odds : int list -> int list = <fun>

# let rec positives xs =
#   match xs with
```

```
#   | [] -> []
#   | hd :: tl -> if hd > 0 then hd :: positives tl
#                  else positives tl ;;
val positives : int list -> int list = <fun>

# let rec negatives xs =
#   match xs with
#   | [] -> []
#   | hd :: tl -> if hd < 0 then hd :: negatives tl
#                  else negatives tl ;;
val negatives : int list -> int list = <fun>
```

We leave the definition of an appropriate abstracted function `filter : (int -> bool) -> int list -> int list` as an exercise.

Exercise 56

Define a function `filter : (int -> bool) -> int list -> int list` that returns a list containing all of the elements of its second argument for which its first argument returns `true`.

Exercise 57

Provide definitions of `evens`, `odds`, `positives`, and `negatives` in terms of `filter`.

Exercise 58

Define a function `reverse : int list -> int list`, which returns the reversal of its argument list. Instead of using explicit recursion, define `reverse` by mapping, folding, or filtering.

Exercise 59

Define a function `append : int list -> int list -> int list` (as described in Exercise 49) to calculate the concatenation of two integer lists. Again, avoid explicit recursion, using map, fold, or filter functions instead.

❧

We've used the same technique three times in this chapter – noticing redundancies in code and carving out the differing bits to find the underlying commonality. The result is a set of higher-order functions – `map`, `fold_left`, `fold_right`, and `filter` – that are broadly useful.

Determining the best place to carve up code into separate factors to take advantage of the commonalities and maximizing the utility of the factors is an important skill, the basis for REFACTORING of code, the name given to exactly this practice. And it turns out to match Socrates's second principle in *Phaedrus*:

> PHAEDRUS: And what is the other principle, Socrates?
> SOCRATES: That of dividing things again by classes, where the natural joints are, and not trying to break any part after the manner of a bad carver. (Plato, 1927)

This principle deserves its own name:

Edict of decomposition:
Carve software at its joints.

The edict of decomposition arises throughout programming practice, but plays an especial role in Chapter 18, where it motivates the programming paradigm of object-oriented programming. For now, however, we continue in the next chapter our pursuit of mechanisms for capturing more abstractions, by allowing generic programs that operate over various types, a technique called *polymorphism*.

8.5 Supplementary material

- Lab 2: Simple data structures and higher-order functions
- Problem set A.2: Higher-order functional programming