8
Higher-order functions and functional programming

Having laid the groundwork for programming with functions, in this chapter we present the edict of irredundancy, and show how higher-order functions serve as a mechanism to satisfy the edict.

Recall that abstraction is the process of viewing a set of apparently dissimilar things as instantiating an underlying identity. Plato in his *Phaedrus* has Socrates adduce two rhetorical principles. The first Socrates describes as

> That of perceiving and bringing together in one idea the scattered particulars, that one may make clear by definition the particular thing which he wishes to explain. (Plato, 1927)

that is, a principle of abstraction. (Socrates's second principle shows up at the end of this chapter.)

Abstraction in programming is this process applied to code, and can be enabled by appropriate language constructs. Programming abstraction is important because it enables programmers to satisfy perhaps the most important edict of programming:

Edict of irredundancy:
Never write the same code twice.

A standard technique that beginning programmers use is "cut and paste" programming – you find some code that does more or less what you need, perhaps code you've written before, and you cut and paste it into your program, adjusting as necessary for the context the code now appears in. There is a high but mostly hidden cost to the cut and paste approach. If you find a bug in one of the copies, it needs to be fixed in all of the copies. If some functionality changes in one of the copies, the other copies don't benefit unless they are modified too. As documentation is added to clarify one of the copies, it must be maintained for all of them. When one of the copies is tested, no assurance is thereby gained for the other copies. There's a theme here. Having written the same code twice, all of the problems of debugging, maintaining, documenting, and testing code have been similarly multiplied.

The edict of irredundancy is the principle of avoiding the problems introduced by duplicative code. Rather than write the same code twice, the edict calls for viewing the apparently dissimilar pieces of code as instantiating an underlying identity, and factoring out the common parts using an appropriate abstraction mechanism.

Given the emphasis in the previous chapters, it will be unsurprising to see that the abstraction mechanism we turn to first is the function itself. 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 44 and 45, 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`. Do we have a tool to characterize 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, that is, one at a time), 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 (`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 6.1, 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.[1]

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

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>
```

[1] A perennial source of confusion is that in this definition of `cube` by partial application, no overt argument appears in the 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. Understanding what's going on in these examples is a good indication that you "get" higher-order functional programming.

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

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.[2] The addition function itself is curried, as we noted in Section 6.1. 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

```
# 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>
```

[2] 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, a topic we'll discuss in Section 9.7.

8.3 The fold abstraction

Let's take a look at some other functions that bear a striking resemblance. Exercises 41 and 42 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), the value to return on the empty list and the operation to apply to the head of the list and the recursively processed tail.

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

Notice the two additional arguments – `f` and `init` – which correspond exactly to the two places that `sum` and `prod` differed. (We place these as the first two arguments looking ahead to the ability to partially apply them.) In summary, the type of `fold` is `(int -> int -> int) -> int -> int list -> int list`.

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 `x_0` is

```
f x_1 (f x_2 (f x_3 ( ⋯ (f x_n x_0)⋯)))    .
```

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

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

or, taking advantage of partial application of the `fold`,

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

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

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

Exercise 49 *The function that we've called `fold` operates "right-to-left" producing*

```
f x_1 (f x_2 (f x_3 ( ⋯ (f x_n x_0)⋯)))     .
```

For this reason, it is sometimes referred to as `fold_right`. The symmetrical function `fold_left` operates left-to-right, calculating

```
(f ⋯ (f (f (f x_0 x_1) x_2) x_3) x_n)     .
```

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

Exercise 50 ✏ *A cousin of the* `fold_left` *function is the function* `reduce`*,*[3] *which 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.* □

[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).

Exercise 51 ✏ *Perhaps surprisingly, the function* `map: (int -> int) -> int list -> int list` *can itself be written in terms of* `fold : (int -> int -> int) -> int -> int list -> int list`*. Provide a definition for* `map` *that involves just a single call to* `fold`*.* □

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, 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 52 ✏ *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 53 ✏ *Provide definitions of* `evens`*,* `odds`*,* `positives`*, and* `negatives` *in terms of* `filter`*.* ☐

Exercise 54 ✏ *Define a function* `reverse : int list -> int list`*, which returns the reversal of its argument list. Instead of using explicit recursion, define* `reverse` *in terms of* `map`*,* `fold`*, and* `filter`*.* ☐

Exercise 55 *Define a function* `append : int list -> int list -> int list` *(as described in Exercise 46) to calculate the concatenation of two integer lists. Again, avoid explicit recursion, using* `map`*,* `fold`*, and* `filter` *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. Determining the best place to carve is an important skill, the basis for REFACTORING of code, which is 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 our pursuit of mechanisms for capturing more abstractions, by allowing generic programs that operate over various types, a technique called *polymorphism*.

8.5 Problem set 2: Higher-order functional programming

8.5.1 Background

This assignment focuses on programming in the functional programming paradigm, with special attention to the idiomatic use of higher-order functions like `map`, `fold`, and `filter`. In doing so, you will exercise important features of functional languages, such as recursion, pattern matching, and list processing.

8.5.2 Setup

First, create your repository in GitHub Classroom for this homework by following this link. Then, follow the GitHub Classroom instructions found in Section **??**.

For this assignment, you will be working in the file `mapfold.ml`.

Reminders

Compilation errors In order to submit your work to the course grading server, your solution must compile against our test suite.

Changing the names or type signatures of the functions that we ask you to write will cause your code to fail to compile against our unit tests, which expect those names. Consequently, you should not change the names or types of those functions. If there are problems that you are unable to solve, you must still write a function that matches the expected type signature, or your code will not compile. (When we provide stub code, that code will typically match the expected type signature and compile to begin with.) If you are having difficulty getting your code to compile, please visit office hours or post on Piazza. Emailing your homework to your TF or the Head TFs is *not* a valid substitute for submitting to the course grading server. Please start early, and submit frequently, to ensure that you are able to submit before the deadline.

Testing is required As with the previous problem set, we ask that you explicitly add tests to your code in the file `mapfold_tests.ml`. You'll want to provide at least one test per code path (that is, each match case and conditional branch). We've provided the very simple unit testing framework from Section 6.5 in the file `test_-simple.ml` and demonstrate its use in the file `mapfold_tests.ml`.

Design and style Good design and style are important aspects of this problem set and all problem sets in the course. Programming with attention to design and style is not only useful in making code more readable but also in helping identify syntactic and type errors in your functions. For this and all assignments, your code should adhere to the style guide in Chapter B.

Compilation with makefiles In the previous problem set, we provided a `Makefile` that allowed you to compile your code without directly calling `ocamlbuild`, the compiler that takes your `.ml` files and turns them into executable `.byte` files.

For this problem set, a `Makefile` might again prove helpful in compiling your code, especially if you don't want to compile it all at once. This time, you will be given the opportunity to write the `Makefile` yourself. Take a look at Subsection **??** for instructions on how to create a `Makefile`.

8.5.3 Higher order functional programming

Mapping, folding, and filtering are important techniques in functional languages that allow the programmer to abstract out the details of traversing and manipulating lists. Each can be used to accomplish a great deal with very little code. In this problem set, you will create a number of functions using the higher-order functions `map`, `filter`, and `fold`. In OCaml, these functions are available as `List.map`, `List.filter`, `List.fold_right`, and `List.fold_left` in the `List` module.

The file `mapfold.ml` contains starter code for a set of functions that operate on lists. For each one, you are to provide the implementation of that function using the higher-order (mapping, folding, filtering) functions directly. The point is to use the higher-order functional programming paradigm idiomatically. The problems will be graded accordingly: a solution, even a working one, that does not use the higher-order functional paradigm, deploying these higher-order functions properly, will receive little or no credit. For instance, solutions that require you to change the `let` to a `let rec` indicate that you probably haven't assimilated the higher-order functional paradigm. However, you should feel free to use functions you've defined earlier in the problem set to implement others later where appropriate.

Remember to provide unit tests for all of the functions in `mapfold.ml` in a file `mapfold_tests.ml`. We have included an example of its use in the starter code for `mapfold_tests.ml`. You should provide at least one test per code path for every function that you write on this problem set.

8.5.4 *Submission*

Before submitting, please estimate how much time you spent on the assignment by editing the line:

```
let minutes_spent_on_pset () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) the assignment took you to complete. For example, if you spent 6 hours on this assignment, you should change the line to:

```
let minutes_spent_on_pset () : int = 360 ;;
```

We also ask that you reflect on the process you went through in working through the problem set, where you ran into problems and how you ended up resolving them. What might you have done in retrospect that would have allowed you to generate as good a submission in less time? Please provide your thoughts by editing the value of the `reflect` function. It should look something like

```
let reflection () : string =
  "...your reflections go here..." ;;
```

Make sure your code still compiles. Then, to submit the assignment, follow the instructions found in Section **??**.