A companion booklet to *Functional Programming in Scala*

Chapter notes, errata, hints, and answers to exercises

compiled by Rúnar Óli Bjarnason

A companion booklet to ”Functional Programming in Scala”

Chapter notes, errata, hints, and answers to exercises

Rúnar Óli Bjarnason

This book is for sale at http://leanpub.com/fpinscalacompanion

This version was published on 2015-03-05

Contents

Answers to exercises

This section contains answers to exercises in the book. Note that usually only one answer is given per exercise, but there may be more than one correct answer. We highly recommend that you check out the book's source code repository from GitHub[1] and look at the answers given there for each chapter. Those answers are shown *in context*, with more detailed explanations, more varied solutions, and in some cases complete implementations that do not fit in this booklet.

Answers to exercises for chapter 2

Exercise 2.01

```
def fib(n: Int): Int = {
  @annotation.tailrec
  def loop(n: Int, prev: Int, cur: Int): Int =
    if (n == 0) prev
    else loop(n - 1, cur, prev + cur)
  loop(n, 0, 1)
}
```

0 and 1 are the first two numbers in the sequence, so we start the accumulators with those. At every iteration, we add the two numbers to get the next one.

Exercise 2.02

```
def isSorted[A](as: Array[A], gt: (A,A) => Boolean): Boolean = {
  @annotation.tailrec
  def go(n: Int): Boolean =
    if (n >= as.length-1) true
    else if (gt(as(n), as(n+1))) false
    else go(n+1)

  go(0)
}
```

Exercise 2.03

Note that `=>` associates to the right, so we could write the return type as `A => B => C`

[1] https://github.com/fpinscala/fpinscala

```
def curry[A,B,C](f: (A, B) => C): A => (B => C) =
  a => b => f(a, b)
```

NB: The `Function2` trait has a `curried` method already, so if you wanted to cheat a little you could write the answer as `f.curried`

Exercise 2.04

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C =
  (a, b) => f(a)(b)
```

NB: There is a method on the `Function` object in the standard library, `Function.uncurried` that you can use for uncurrying.

Note that we can go back and forth between the two forms. We can curry and uncurry and the two forms are in some sense "the same". In FP jargon, we say that they are *isomorphic* ("iso" = same; "morphe" = shape, form), a term we inherit from category theory.

Exercise 2.05

```
def compose[A,B,C](f: B => C, g: A => B): A => C =
  a => f(g(a))
```

Answers to exercises for chapter 3

Exercise 3.01

Three. The third case is the first that matches, with `x` bound to `1` and `y` bound to `2`.

Exercise 3.02

Although we could return `Nil` when the input list is empty, we choose to throw an exception instead. This is a somewhat subjective choice. In our experience, taking the tail of an empty list is often a bug, and silently returning a value just means this bug will be discovered later, further from the place where it was introduced.

It's generally good practice when pattern matching to use _ for any variables you don't intend to use on the right hand side of a pattern. This makes it clear the value isn't relevant.

```
def tail[A](l: List[A]): List[A] =
  l match {
    case Nil => sys.error("tail of empty list")
    case Cons(_,t) => t
  }
```

Exercise 3.03

If a function body consists solely of a match expression, we'll often put the match on the same line as the function signature, rather than introducing another level of nesting.

```
def setHead[A](l: List[A], h: A): List[A] = l match {
  case Nil => sys.error("setHead on empty list")
  case Cons(_,t) => Cons(h,t)
}
```

Exercise 3.04

Again, it's somewhat subjective whether to throw an exception when asked to drop more elements than the list contains. The usual default for `drop` is not to throw an exception, since it's typically used in cases where this is not indicative of a programming error. If you pay attention to how you use `drop`, it's often in cases where the length of the input list is unknown, and the number of elements to be dropped is being computed from something else. If `drop` threw an exception, we'd have to first compute or check the length and only drop up to that many elements.

```
def drop[A](l: List[A], n: Int): List[A] =
  if (n <= 0) l
  else l match {
    case Nil => Nil
    case Cons(_,t) => drop(t, n-1)
  }
```

Exercise 3.05

Somewhat overkill, but to illustrate the feature we're using a *pattern guard*, to only match a `Cons` whose head satisfies our predicate, `f`. The syntax is to add `if <cond>` after the pattern, before the `=>`, where `<cond>` can use any of the variables introduced by the pattern.

```
def dropWhile[A](l: List[A], f: A => Boolean): List[A] =
  l match {
    case Cons(h,t) if f(h) => dropWhile(t, f)
    case _ => l
  }
```

Exercise 3.06

Note that we're copying the entire list up until the last element. Besides being inefficient, the natural recursive solution will use a stack frame for each element of the list, which can lead to stack overflows for large lists (can you see why?). With lists, it's common to use a temporary, mutable buffer internal to the function (with lazy lists or streams, which we discuss in chapter 5, we don't normally do this). So long as the buffer is allocated internal to the function, the mutation is not observable and RT is preserved.

Another common convention is to accumulate the output list in reverse order, then reverse it at the end, which doesn't require even local mutation. We'll write a reverse function later in this chapter.

```
def init[A](l: List[A]): List[A] =
  l match {
    case Nil => sys.error("init of empty list")
    case Cons(_,Nil) => Nil
    case Cons(h,t) => Cons(h,init(t))
  }
def init2[A](l: List[A]): List[A] = {
  import collection.mutable.ListBuffer
  val buf = new ListBuffer[A]
  @annotation.tailrec
  def go(cur: List[A]): List[A] = cur match {
    case Nil => sys.error("init of empty list")
    case Cons(_,Nil) => List(buf.toList: _*)
    case Cons(h,t) => buf += h; go(t)
  }
  go(l)
}
```

Exercise 3.07

No, this is not possible! The reason is because *before* we ever call our function, `f`, we evaluate its argument, which in the case of `foldRight` means traversing the list all the way to the end. We need *non-strict* evaluation to support early termination—we discuss this in chapter 5.

Exercise 3.08

We get back the original list! Why is that? As we mentioned earlier, one way of thinking about what `foldRight` "does" is it replaces the `Nil` constructor of the list with the `z` argument, and it replaces the `Cons` constructor with the given function, `f`. If we just supply `Nil` for `z` and `Cons` for `f`, then we get back the input list.

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))), Nil:List[Int])(Cons(_,_))
Cons(1, foldRight(Cons(2, Cons(3, Nil)), Nil:List[Int])(Cons(_,_)))
Cons(1, Cons(2, foldRight(Cons(3, Nil), Nil:List[Int])(Cons(_,_))))
Cons(1, Cons(2, Cons(3, foldRight(Nil, Nil:List[Int])(Cons(_,_)))))
Cons(1, Cons(2, Cons(3, Nil)))
```

Exercise 3.09

```
def length[A](l: List[A]): Int =
  foldRight(l, 0)((_,acc) => acc + 1)
```

Exercise 3.10

It's common practice to annotate functions you expect to be tail-recursive with the `tailrec` annotation. If the function is not tail-recursive, it will yield a compile error, rather than silently compiling the code and resulting in greater stack space usage at runtime.

```
@annotation.tailrec
def foldLeft[A,B](l: List[A], z: B)(f: (B, A) => B): B = l match {
  case Nil => z
  case Cons(h,t) => foldLeft(t, f(z,h))(f)
}
```

Exercise 3.11

```
def sum3(l: List[Int]) = foldLeft(l, 0)(_ + _)
def product3(l: List[Double]) = foldLeft(l, 1.0)(_ * _)

def length2[A](l: List[A]): Int = foldLeft(l, 0)((acc,h) => acc + 1)
```

Exercise 3.12

```
def reverse[A](l: List[A]): List[A] =
  foldLeft(l, List[A]())((acc,h) => Cons(h,acc))
```

Exercise 3.13

The implementation of `foldRight` in terms of `reverse` and `foldLeft` is a common trick for avoiding stack overflows when implementing a strict `foldRight` function as we've done in this chapter. (We'll revisit this in a later chapter, when we discuss laziness).

The other implementations build up a chain of functions which, when called, results in the operations being performed with the correct associativity. We are calling `foldRight` with the `B` type being instantiated to `B => B`, then calling the built up function with the `z` argument. Try expanding the definitions by substituting equals for equals using a simple example, like `foldLeft(List(1,2,3), 0)(_ + _)` if this isn't clear. Note these implementations are more of theoretical interest - they aren't stack-safe and won't work for large lists.

```
def foldRightViaFoldLeft[A,B](l: List[A], z: B)(f: (A,B) => B): B =
  foldLeft(reverse(l), z)((b,a) => f(a,b))

def foldRightViaFoldLeft_1[A,B](l: List[A], z: B)(f: (A,B) => B): B =
  foldLeft(l, (b:B) => b)((g,a) => b => g(f(a,b)))(z)

def foldLeftViaFoldRight[A,B](l: List[A], z: B)(f: (B,A) => B): B =
  foldRight(l, (b:B) => b)((a,g) => b => g(f(b,a)))(z)
```

Exercise 3.14

`append` simply replaces the `Nil` constructor of the first list with the second list, which is exactly the operation performed by `foldRight`.

```
def appendViaFoldRight[A](l: List[A], r: List[A]): List[A] =
  foldRight(l, r)(Cons(_,_))
```

Exercise 3.15

Since `append` takes time proportional to its first argument, and this first argument never grows because of the right-associativity of `foldRight`, this function is linear in the total length of all lists. You may want to try tracing the execution of the implementation on paper to convince yourself that this works.

Note that we're simply referencing the `append` function, without writing something like `(x,y) => append(x,y)` or `append(_,_)`. In Scala there is a rather arbitrary distinction between functions defined as *methods*, which are introduced with the `def` keyword, and function values, which are the

first-class objects we can pass to other functions, put in collections, and so on. This is a case where Scala lets us pretend the distinction doesn't exist. In other cases, you'll be forced to write `append _` (to convert a `def` to a function value) or even `(x: List[A], y: List[A]) => append(x,y)` if the function is polymorphic and the type arguments aren't known.

```
def concat[A](l: List[List[A]]): List[A] =
  foldRight(l, Nil:List[A])(append)
```

Exercise 3.16

```
def add1(l: List[Int]): List[Int] =
  foldRight(l, Nil:List[Int])((h,t) => Cons(h+1,t))
```

Exercise 3.17

```
def doubleToString(l: List[Double]): List[String] =
  foldRight(l, Nil:List[String])((h,t) => Cons(h.toString,t))
```

Exercise 3.18

A natural solution is using `foldRight`, but our implementation of `foldRight` is not stack-safe. We can use `foldRightViaFoldLeft` to avoid the stack overflow (variation 1), but more commonly, with our current implementation of `List`, `map` will just be implemented using local mutation (variation 2). Again, note that the mutation isn't observable outside the function, since we're only mutating a buffer that we've allocated.

```
def map[A,B](l: List[A])(f: A => B): List[B] =
  foldRight(l, Nil:List[B])((h,t) => Cons(f(h),t))

def map_1[A,B](l: List[A])(f: A => B): List[B] =
  foldRightViaFoldLeft(l, Nil:List[B])((h,t) => Cons(f(h),t))

def map_2[A,B](l: List[A])(f: A => B): List[B] = {
  val buf = new collection.mutable.ListBuffer[B]
  def go(l: List[A]): Unit = l match {
    case Nil => ()
    case Cons(h,t) => buf += f(h); go(t)
  }
  go(l)
  List(buf.toList: _*)
  // converting from the standard Scala list to the list we've defined here
}
```

Exercise 3.19

```
/*
The discussion about `map` also applies here.
*/
def filter[A](l: List[A])(f: A => Boolean): List[A] =
  foldRight(l, Nil:List[A])((h,t) => if (f(h)) Cons(h,t) else t)

def filter_1[A](l: List[A])(f: A => Boolean): List[A] =
  foldRightViaFoldLeft(l, Nil:List[A])((h,t) => if (f(h)) Cons(h,t) else t)

def filter_2[A](l: List[A])(f: A => Boolean): List[A] = {
  val buf = new collection.mutable.ListBuffer[A]
  def go(l: List[A]): Unit = l match {
    case Nil => ()
    case Cons(h,t) => if (f(h)) buf += h; go(t)
  }
  go(l)
  List(buf.toList: _*)
  // converting from the standard Scala list to the list we've defined here
}
```

Exercise 3.20

```
/*
This could also be implemented directly using `foldRight`.
*/
def flatMap[A,B](l: List[A])(f: A => List[B]): List[B] =
  concat(map(l)(f))
```

Exercise 3.21

```
def filterViaFlatMap[A](l: List[A])(f: A => Boolean): List[A] =
  flatMap(l)(a => if (f(a)) List(a) else Nil)
```

Exercise 3.22

To match on multiple values, we can put the values into a pair and match on the pair, as shown next, and the same syntax extends to matching on N values (see sidebar "Pairs and tuples in Scala" for more about pair and tuple objects). You can also (somewhat less conveniently, but a bit more efficiently) nest pattern matches: on the right hand side of the `=>`, simply begin another `match` expression. The inner `match` will have access to all the variables introduced in the outer `match`.

The discussion about stack usage from the explanation of `map` also applies here.

```
def addPairwise(a: List[Int], b: List[Int]): List[Int] = (a,b) match {
  case (Nil, _) => Nil
  case (_, Nil) => Nil
  case (Cons(h1,t1), Cons(h2,t2)) => Cons(h1+h2, addPairwise(t1,t2))
}
```

Exercise 3.23

This function is usually called `zipWith`. The discussion about stack usage from the explanation of `map` also applies here. By putting the `f` in the second argument list, Scala can infer its type from the previous argument list.

```
def zipWith[A,B,C](a: List[A], b: List[B])(f: (A,B) => C): List[C] =
  (a,b) match {
    case (Nil, _) => Nil
    case (_, Nil) => Nil
    case (Cons(h1,t1), Cons(h2,t2)) => Cons(f(h1,h2), zipWith(t1,t2)(f))
  }
```

Exercise 3.24

There's nothing particularly bad about this implementation, except that it's somewhat monolithic and easy to get wrong. Where possible, we prefer to assemble functions like this using combinations of other functions. It makes the code more obviously correct and easier to read and understand. Notice that in this implementation we need special purpose logic to break out of our loops early. In Chapter 5 we'll discuss ways of composing functions like this from simpler components, without giving up the efficiency of having the resulting functions work in one pass over the data.

```
@annotation.tailrec
def startsWith[A](l: List[A], prefix: List[A]): Boolean = (l,prefix) match {
  case (_,Nil) => true
  case (Cons(h,t),Cons(h2,t2)) if h == h2 => startsWith(t, t2)
  case _ => false
}
@annotation.tailrec
def hasSubsequence[A](sup: List[A], sub: List[A]): Boolean = sup match {
  case Nil => false
  case _ if startsWith(sup, sub) => true
  case Cons(h,t) => hasSubsequence(t, sub)
}
```

Exercise 3.25

```
def size[A](t: Tree[A]): Int = t match {
  case Leaf(_) => 1
  case Branch(l,r) => 1 + size(l) + size(r)
}
```

Exercise 3.26

We're using the method `max` that exists on all `Int` values rather than an explicit `if` expression.

Note how similar the implementation is to `size`. We'll abstract out the common pattern in a later exercise.

```
def maximum(t: Tree[Int]): Int = t match {
  case Leaf(n) => n
  case Branch(l,r) => maximum(l) max maximum(r)
}
```

Exercise 3.27

Again, note how similar the implementation is to `size` and `maximum`.

```
def depth[A](t: Tree[A]): Int = t match {
  case Leaf(_) => 0
  case Branch(l,r) => 1 + (depth(l) max depth(r))
}
```

Exercise 3.28

```
def map[A,B](t: Tree[A])(f: A => B): Tree[B] = t match {
  case Leaf(a) => Leaf(f(a))
  case Branch(l,r) => Branch(map(l)(f), map(r)(f))
}
```

Exercise 3.29

```
def mapViaFold[A,B](t: Tree[A])(f: A => B): Tree[B] =
  fold(t)(a => Leaf(f(a)): Tree[B])(Branch(_,_))
```

Like `foldRight` for lists, `fold` receives a "handler" for each of the data constructors of the type, and recursively accumulates some value using these handlers. As with `foldRight`, `fold(t)(Leaf(_))(Branch(_,_)) == t`, and we can use this function to implement just about any recursive function that would otherwise be defined by pattern matching.

```
def fold[A,B](t: Tree[A])(f: A => B)(g: (B,B) => B): B = t match {
  case Leaf(a) => f(a)
  case Branch(l,r) => g(fold(l)(f)(g), fold(r)(f)(g))
}

def sizeViaFold[A](t: Tree[A]): Int =
  fold(t)(a => 1)(1 + _ + _)

def maximumViaFold(t: Tree[Int]): Int =
  fold(t)(a => a)(_ max _)

def depthViaFold[A](t: Tree[A]): Int =
  fold(t)(a => 0)((d1,d2) => 1 + (d1 max d2))
```

Note the type annotation required on the expression `Leaf(f(a))`. Without this annotation, we get an error like this:

```
type mismatch;
  found   : fpinscala.datastructures.Branch[B]
  required: fpinscala.datastructures.Leaf[B]
    fold(t)(a => Leaf(f(a)))(Branch(_,_))
                                    ^
```

This error is an unfortunate consequence of Scala using subtyping to encode algebraic data types. Without the annotation, the result type of the fold gets inferred as `Leaf[B]` and it is then expected that the second argument to `fold` will return `Leaf[B]`, which it doesn't (it returns `Branch[B]`). Really, we'd prefer Scala to infer `Tree[B]` as the result type in both cases. When working with algebraic data types in Scala, it's somewhat common to define helper functions that simply call the corresponding data constructors but give the less specific result type:

```
def leaf[A](a: A): Tree[A] = Leaf(a)
def branch[A](l: Tree[A], r: Tree[A]): Tree[A] = Branch(l, r)
```