Exploring Languages with Interpreters and Functional Programming Chapter 15

H. Conrad Cunningham

10 October 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

15 Higher-Order Functions

15.1 Chapter Introduction

The previous chapters discussed first-order programming in Haskell. This chapter "kicks it up a notch" (to quote chef Emeril Lagasse) by adding powerful new abstraction facilities.

The chapter introduces the concepts of first-class and higher-order functions and constructs a library of useful higher-order functions to process lists. It continues the emphasis on Haskell programs that are correct, terminating, efficient, and elegant.

The chapter approaches the development of higher-order functions by generalizing a set of first-order functions having similar patterns of computation.

The Haskell module for this chapter is in `HigherOrderFunctions.hs`.

15.2 Generalizing Procedural Abstractions

A function in a programming language is a *procedural abstraction*. It separates the logical properties of a computation from the details of how the computation is implemented. It abstracts a pattern of behavior and encapsulates it within a program unit.

Suppose we wish to perform the *same* computation on a set of *similar* data structures. As we have seen, we can encapsulate the computation in a function having the data structure as an argument. For example, the function `length'` computes the number of elements in a list of any type.

Suppose instead we wish to perform a *similar* (but not identical) computation on a set of *similar* data structures. For example, we want to compute the sum or the product of a list of numbers. In this case, we may can pass the operation itself into the function.

This kind of function is called a *higher-order function*. A higher-order function is a function that takes functions as arguments or returns functions in a result. Most traditional imperative languages do not fully support higher-order functions.

In most functional programming languages, functions are treated as *first class* values. That is, functions can be stored in data structures, passed as arguments to functions, and returned as the results of functions. Historically, imperative languages have not treated functions as first-class values. (Recently, many imperative languages, such as Java 8, have added support for functions as first-class values.)

The higher-order functions in Haskell and other functional programming languages enable us to construct regular and powerful abstractions and operations.

By taking advantage of a library of higher-order functions that capture common patterns of computation, we can quickly construct concise, yet powerful, programs.

This can increase programmer productivity and program reliability because such programs are shorter, easier to understand, and constructed from well-tested components.

Higher-order functions can also increase the *modularity* of programs by enabling simple program fragments to be "glued together" readily into more complex programs.

In this chapter, we examine several common patterns and build a library of useful higher-order functions.

15.3 Defining `map`

Consider the following two functions, noting their type signatures and patterns of recursion.

The first, `squareAll`, takes a list of integers and returns the corresponding list of squares of the integers.

```
squareAll :: [Int] -> [Int]  squareAll :: [Int] -> [Int]
squareAll []     = []
squareAll (x:xs) = (x * x) : squareAll xs
```

The second, `lengthAll`,q takes a list of lists and returns the corresponding list of the lengths of the element lists; it uses the Prelude function `length`.

```
lengthAll :: [[a]] -> [Int]
lengthAll []       = []
lengthAll (xs:xss) = (length xs) : lengthAll xss
```

Although these functions take different kinds of data (a list of integers versus a list of polymorphically typed lists) and apply different operations (squaring versus list length), they exhibit the same pattern of computation. That is, both take a list of some type and apply a given function to each element to generate a resulting list of the same length as the original.

The combination of polymorphic typing and higher-order functions allow us to abstract this pattern of computation into a standard function.

We can abstract the pattern of computation common to `squareAll` and `lengthAll` as the (broadly useful) function `map`, which we define as follows. (In this chapter, we often add a suffix to the base function names to avoid conflicts with the similarly named functions in the Prelude. Here we use `map'` instead of `map`.)

```
map' :: (a -> b) -> [a] -> [b]   -- map in Prelude
map' f []     = []
map' f (x:xs) = f x : map' f xs
```

Function `map` *generalizes* `squareAll`, `lengthAll`, and similar functions by adding a higher-order parameter for the operation applied and making the input and the output lists polymorphic. Specifically, he function takes a function `f` of type `a -> b` and a list of type `[a]`, applies function `f` to each element of the list, and produces a list of type `[b]`.

Thus we can *specialize* `map` to give new definitions of `squareAll` and `lengthAll` as follows:

```
squareAll2 :: [Int] -> [Int]
squareAll2 xs = map' sq xs
                where sq x = x * x

lengthAll2 :: [[a]] -> [Int]
lengthAll2 xss = map' length xss
```

Consider the following questions.

- Under what circumstances does `map' f xs` terminate? Do we have to assume anything about `f`? about `xs`?
- What is the time complexity of `map f xs`?
- What is the time complexity of `squareAll2 xs`? Of `lengthAll2 xs`?

15.4 Thinking about Data Transformations

Above we define `map` as a recursive function that transforms the elements of a list one by one. However, it is often more useful to think of `map` in one of two ways:

1. as a powerful list operator that transforms every element of the list. We can combine `map` with other powerful operators to quickly construct powerful list processing programs.

   We can consider `map` as operating on every element of the list "simultaneously". In fact, an implementation could use separate processors to transform each element: this is essentially the `map` operation in Google's `mapReduce` distributed "big data" processing framework.

   Referential transparency and immutable data structures make parallelism easier in Haskell than in most imperative languages.

2. as a operator node in a dataflow network. A stream of data objects flows into the `map` node. The `map` node transforms each object by applying the

argument function. Then the data object flows out to the next node of the network.

The lazy evaluation of the Haskell functions enables such an implementation.

Although in the early parts of these notes we give attention to the details of recursion, learning how to *think like a functional programmer* requires us to think about large-scale transformations of collections of data.

15.5 Generalizing Function Definitions

Whenever we recognize a computational pattern in a set of related functions, we can *generalize the function* definition as follows:

1. Do a *scope-commonality-variability (SCV) analysis* on the set of related functions.

   That is, identify what is to be included and what not (i.e. the scope), the parts of functions that are the same (the *commonalities* or *frozen spots*), and the parts that differ (the *variabilities* or *hot spots*)

2. Leave the commonalities in the generalized function's body.

3. Move the variabilities into the generalized function's header—its type signature and parameter list.

   - If the part moved to the generalized function's parameter list is an expression, then make that part a function with a parameter for each local variable accessed.

   - If a data type potentially differs from a specific type used in the set of related functions, then add a type parameter to the generalized function.

   - If the same data value or type appears in multiple roles, then consider adding distinct type or value parameters for each role.

4. Consider other approaches if the generalized function's type type signature and parameter list become too complex.

   For example, we can introduce new data or procedural abstractions for parts of the generalized function. These may be in the same module of the generalized function or in an appropriately defined separate module.

15.6 Defining `filter`

Consider the following two functions.

The first, `getEven`, takes a list of integers and returns the list of those integers that are even (i.e. are multiples of 2). The function preserves the relative order of the elements in the list.

```
getEven :: [Int] -> [Int]
getEven []       = []
getEven (x:xs)
    | even x    = x : getEven xs
    | otherwise = getEven xs
```

The second, `doublePos`, takes a list of integers and returns the list of doubles of the positive integers from the input list; it preserves the relative order of the elements.

```
doublePos :: [Int] -> [Int]
doublePos []          = []
doublePos (x:xs)
    | 0 < x     = (2 * x) : doublePos xs
    | otherwise = doublePos xs
```

Function `even` is from the Prelude; it returns `True` if its argument is evenly divisible by 2 and returns `False` otherwise.

What do these two functions have in common? What differs?

- Both take a list of integers and return a (possibly shorter) list of integers.

  However, the fact they use integers is not important; the key fact is that they take and return lists of the same element type.

- Both return an empty list when its input list is empty.

- In both, the relative orders of elements in the output list is the same as in the input list.

- Both select some elements to copy to the output and others not to copy.

  Function `getEven` selects elements that are even numbers and function `doublePos` selects elements that are positive numbers.

- Function `doublePos` doubles the value copied and `getEven` leaves the value unchanged.

Using the generalization method outlined above, we abstract the pattern of computation common to `getEven` and `doublePos` as the (broadly useful) function `filter` found in the Prelude. (We call the function `filter'` below to avoid a name conflict.)

```
filter' :: (a -> Bool) -> [a] -> [a]   -- filter in Prelude
filter' _ []    = []
filter' p (x:xs)
    | p x       = x : xs'
```

```
        | otherwise = xs'
                    where xs' = filter' p xs
```

Function `filter` takes a predicate `p` of type `a -> Bool` and a list of type `[a]` and returns a list containing those elements that satisfy `p`, in the same order as the input list. Note that the keyword `where` begins in the same column as the `=` in the defining equations; thus the scope of the definition of `xs'` extends over *both* legs of the definition.

Function `filter` does not incorporate the doubling operation from `doublePos`. We could have included it as another higher-order parameter, but we leave it out to keep the generalized function simple. We can use the already defined `map` function to achieve this separately.

Therefore, we can specialize `filter` to give new definitions of `getEven` and `doublePos` as follows:

```
getEven2 :: [Int] -> [Int]
getEven2 xs = filter' even xs

doublePos2 :: [Int] -> [Int]
doublePos2 xs = map' dbl (filter' pos xs)
                where dbl x  = 2 * x
                      pos x = (0 < x)
```

Note that function `doublePos2` exhibits both the `filter` and the `map` patterns of computation.

The standard higher-order functions `map` and `filter` allow us to restate the three-leg definitions of `getEven` and `doublePos` in just one leg each, except that `doublePos` requires two lines of local definitions. In subsequent sections, we see how to eliminate these simple local definitions as well.

Consider the following questions.

- Under what circumstances does `filter' p xs` terminate? Do we have to assume anything about `p`? about `xs`?
- What is the time complexity of `filter' p xs`? space complexity?
- What is the time complexity of `getEven2 xs`? space complexity?
- What is the time complexity of `doublePos2 xs`? space complexity?

15.7 Defining Fold Right (`foldr`)

Consider the `sum` and `product {.haskell}` functions we defined in a previous chapter, ignoring the short-cut handling of the zero element in `product`.

```
sum' :: [Int] -> Int               -- sum in Prelude
sum' []      = 0
```

```
sum' (x:xs) = x + sum' xs

product' :: [Integer] -> Integer -- product in Prelude
product' []     = 1
product' (x:xs) = x * product' xs
```

Both `sum'` and `product'` apply arithmetic operations to integers. What about other operations with similar pattern of computation?

Also consider a function `concat` that concatenates a list of lists of some type into a list of that type with the order of the input lists and their elements preserved.

```
concat' :: [[a]] -> [a]   -- concat in Prelude
concat' []       =  []
concat' (xs:xss) =  xs ++ concat' xss
```

For example,

```
sum' [1,2,3]           = (1 + (2 + (3 + 0)))
product' [1,2,3]       = (1 * (2 * (3 * 1)))
concat' ["1","2","3"] = ("1" ++ ("2" ++ ("3" ++ "")))
```

What do `sum'`, `product'`, and `concat'` have in common? What differs?

All exhibit the same pattern of computation.

- All take a list.

  But the element type differs. Function `sum'` takes a list of `Int` values, `product'` takes a list of `Integer` values, and `concat'` takes a polymorphic list.

- All insert a binary operator between all the consecutive elements of the list in order to reduce the list to a single value.

  But the binary operation differs. Function `sum'` applies integer addition, `product'` applies integer multiplication, and `concat'` applies `++`.

- All group the operations from the right to the left.

- Each function returns some value for an empty list. The function extends nonempty input lists to implicitly include this value as the "rightmost" value of the input list.

  But the actual value differs.

  Function `sum'` returns integer 0, the (right) identity element for addition.

  Function `product'` returns 1, the (right) identity element for multiplication.

  Function `concat'` returns `[]`, the (right) identity element for `++`.

  In general, this value could be something other than the identity element.

- All return a value of the same element type as the input list.

  But the input type differs, as we noted above.

This group of functions inserts operations of type `a -> a -> a` between elements a list of type `[a]`.

But these are special cases of more general operations of type `a -> b -> b`. In this case, the value returned must be of type `b` in the case of both empty and nonempty lists.

We can abstract the pattern of computation common to `sum'`, `product'`, and `concat’` as the function `foldr` (pronounced “fold right”) found in the Prelude. (Here we use `foldrX`{.haskell} to avoid the name conflict.)

```
foldrX :: (a -> b -> b) -> b -> [a] -> b  -- foldr in Prelude
foldrX f z []     = z
foldrX f z (x:xs) = f x (foldrX f z xs)
```

Function `foldr`:

- uses two type parameters `a` and `b`—one for the type of elements in the list and one for the type of the result
- passes in the general binary operation `f` (with type `a -> b -> b`) that combines (i.e. folds) the list elements
- passes in the “seed” element `z` (of type `b`) to be returned for empty lists

The `foldr` function “folds” the list elements (of type `a`) into a value (of type `b`) by “inserting” operation `f` between the elements, with value `z` “appended” as the rightmost element.

Often the seed value `z` is the right identity element for the operation, but `foldr` may be useful in some circumstances where it is not (or perhaps even if there is no right identity).

For example, `foldr f z [1,2,3]` expands to `f 1 (f 2 (f 3 z))`, or, using an infix style:

```
1 `f` (2 `f` (3 `f` z))
```

Function `foldr` does not depend upon `f` being associative or having either a right or left identity.

Function `foldr` is backward recursive. If the function application is fully evaluated, it needs a new stack frame for each element of the input list. If its list argument is long or the folding function itself is expensive, then the function can terminate with a *stack overflow* error.

In Haskell, `foldr` is called a *fold* operation. Other languages sometimes call this a *reduce* or *insert* operation.

We can specialize `foldr` to restate the definitions for `sum'`, `product'`, and `concat'`.

```
sum2 :: [Int] -> Int                -- sum
sum2 xs = foldrX (+) 0 xs

product2 :: [Int] -> Int            -- product
product2 xs = foldrX (*) 1 xs

concat2:: [[a]] -> [a]              -- concat
concat2 xss = foldrX (++) [] xss
```

As further examples, consider the folding of the Boolean operators `&&` ("and") and `||` ("or") over lists of Boolean values as Prelude functions `and` and `or` (shown as `and'` and `or'` below to avoid name conflicts):

```
and', or' :: [Bool] -> Bool  -- and, or in Prelude
and' xs = foldrX (&&) True xs
or'  xs = foldrX (||) False xs
```

Although their definitions look different, `and'` and `or'` are actually identical to functions `and` and `or` in the Prelude.

Consider the following questions.

- Under what circumstances does `foldrX f z xs` terminate? Do we have to assume anything about `f`? about `xs`?
- What is the time complexity of `product2`? of `concat2`?

15.8 Using `foldr`

The fold functions are very powerful. By choosing an appropriate folding function argument, many different list functions can be implemented in terms of `foldr`.

For example, we can implement `map` using `foldr` as follows:

```
map2 :: (a -> b) -> [a] -> [b] -- map
map2 f xs = foldr mf [] xs
    where mf y ys = (f y) : ys
```

The folding function `mf y ys = (f y):ys` applies the mapping function `f` to the next element of the list (moving right to left) and attaches the result on the front of the processed tail. This is a case where the folding function `mf` does not have a right identity, but where `foldr` is quite useful.

We can also implement `filter` in terms of `foldr` as follows:

```
filter2 :: (a -> Bool) -> [a] -> [a]  -- filter
filter2 p xs = foldr ff [] xs
    where ff y ys = if p y then (y:ys) else ys
```

The folding function `ff y ys = if p x then (y:ys) else ys` applies the filter predicate `p` to the next element of the list (moving right to left). If the predicate evaluates to `True`, the folding function attaches that element on the front of the processed tail; otherwise, it omits the element from the result.

We can also use `foldr` to compute the length of a polymorphic list.

```
length2 :: [a] -> Int   -- length
length2 xs  = foldr len  0  xs
    where len _ acc = acc + 1
```

This uses the `z` parameter of `foldr` to initialize the count to 0. Higher-order argument `f` of `foldr` is a function that takes an element of the list as its left argument and the previous accumulator as its right argument and returns the accumulator incremented by 1. In this application, `z` is not the identity element for `f` but is a convenient beginning value for the counter.

We can construct an "append" function that uses `foldr` as follows:

```
append2 :: [a] -> [a] -> [a]   -- ++
append2 xs ys = foldr (:) ys xs
```

Here the the list that `foldr` operates on the first argument of the append. The `z` parameter is the entire second argument and the folding function is just `(:)`. So the effect is to replace the `[]` at the end of the first list by the entire second list.

Function `foldr` 1s a backward recursive function that processes the elements of a list one by one. However, as we have seen, it is often more useful to think of `foldr` as a powerful list operator that reduces the element of the list into a single value. We can combine `foldr` with other operators to conveniently construct list processing programs.

15.9 Defining Fold Left (`foldl`)

We designed function `foldr` as a backward linear recursive function with the signature:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

As noted:

```
foldr f z [1,2,3] ==  f 1 (f 2 (f 3 z))
                  ==  1 `f` (2 `f` (3 `f` z))
```

Consider a function `foldl` (pronounced "fold left") such that:

```
foldl f z [1,2,3]  ==  f (f (f z 1) 2) 3
                   ==  ((z `f` 1) `f` 2) `f` 3`
```

This function folds from the left. It offers us the opportunity to use parameter `z` as an accumulating parameter in a tail recursive implementation. This is shown below as `foldlX`, which is similar to `foldl` in the Prelude.

```
foldlX :: (a -> b -> a) -> a -> [b] -> a  -- foldl in Prelude
foldlX f z []     = z
foldlX f z (x:xs) = foldlX f (f z x) xs
```

Note how the second leg of `foldlX` implements the left binding of the operation. In the recursive call of `foldlX` the "seed value" argument is used as an accumulating parameter.

Also note how the types of `foldr` and `foldl` differ.

Often the beginning value of `z` is the left identity of the operation `f`, but `foldl` (like `foldr`) can be a quite useful function in circumstances when it is not (or when `f` has no left identity).

15.10 Using `foldl`

If $\oplus$ is an associative binary operation of type `t -> t -> t` with identity element `z` (i.e. $\oplus$ and `t` form the algebraic structure know as a monoid), then, for any `xs`,

```
foldr (⊕) z xs = foldl (⊕) z xs
```

The classic Bird and Wadler textbook [Bird 1988] calls this property the *first duality theorem*.

Because `+`, `*`, and `++` are all associative operations with identity elements, `sum`, `product`, and `concat` can all be implemented with either `foldr` or `foldl`.

Which is better?

Depending upon the nature of the operation, an implementation using `foldr` may be more efficient than `foldl` or vice versa.

We defer a more complete discussion of the efficiency until we study evaluation strategies further in a later chapter.

As a rule of thumb, however, if the operation $\oplus$ is *nonstrict* in either argument, then it is usually better to use `foldr`. That form takes better advantage of lazy evaluation.

If the operation $\oplus$ is *strict* in both arguments, then it is often better (i.e. more efficient) to use the optimized version of `foldl` called `foldl'` from the standard Haskell module `Data.List`.

The append operation `++` is nonstrict in its second argument, so it is better to use `foldr` to implement `concat`.

Addition and multiplication are strict in both arguments, so we can implement `sum` and `product` functions efficiently with `foldl'`, as follows:

```
import Data.List    -- to make foldl' available
sum3, product3 :: Num a =>  [a] -> a -- sum, product
sum3 xs     = foldl' (+) 0 xs
product3 xs = foldl' (*) 1 xs
```

Note that we generalize these functions to operate on polymorphic lists with a base type in class `Num`. Class `Num` includes all numeric types.

Function `length3` uses `foldl`. It is like `length2` except that the arguments of function `len` are reversed.

```
length3 :: [a] -> Int   -- length
length3 xs  = foldl len 0  xs
    where len acc _ = acc + 1
```

However, it is usually better to use the `foldr` version `length2` because the folding function `len` is nonstrict in the argument corresponding to the list.

We can also implement list reversal using `foldl` as follows:

```
reverse2 :: [a] -> [a]   -- reverse
reverse2 xs = foldl rev [] xs
    where rev acc x = (x:acc)
```

This gives a solution similar to the tail recursive `reverse` function from a previous chapter. The `z` parameter of function `foldl` is initially an empty list; the folding function parameter `f` of `foldl` uses `(:)` to “attach” each element of the list as the new head of the accumulator, incrementally building the list in reverse order.

Although cons is nonstrict in its right operand, `reverse2` builds up that argument from `[]`, so `reverse2` cannot take advantage of lazy evaluation by using `foldr` instead.

To avoid a stack overflow situation with `foldr`, we can first apply `reverse` to the list argument and then apply `foldl` as follows:

```
foldr2 :: (a -> b -> b) -> b -> [a] -> b  -- foldr
foldr2 f z xs = foldl flipf z (reverse xs)
    where flipf y x = f x y
```

The combining function in the call to `foldl` is the same as the one passed to `foldr` except that its arguments are reversed.

15.11 Defining `concatMap` (flatmap)

The higher-order function `map` applies its function argument `f` to every element of a list and returns the list of results. If the function argument `f` returns a list, then the result is a list of lists. Often we wish to flatten this into a single list, that is, apply a function like `concat` defined in a previous section.

This computation is sufficiently common that we give it the name `concatMap`. We can define it in terms of `map` and `concat` as

```
concatMap' :: (a -> [b]) -> [a] -> [b]
concatMap' f xs = concat (map f xs)
```

or by combining `map` and `concat` into one `foldr` as:

```
concatMap2 :: (a -> [b]) -> [a] -> [b]
concatMap2 f xs = foldr fmf [] xs
    where fmf x ys = f x ++ ys
```

Above, the function argument to `foldr` applies the `concatMap` function argument `f` to each element of the list argument and then appends the resulting list in front of the result from processing the elements to the right.

We can also define `filter` in terms of `concatMap` as follows:

```
filter3 :: (a -> Bool) -> [a] -> [a]
filter3 p xs = concatMap' fmf xs
    where fmf x = if p x then [x] else []
```

The function argument to `concatMap` generates a one-element list if the filter predicate `p` is true and an empty list if it is false.

Some other languages (e.g. Scala) call the `concatMap` function by the name `flatmap`.

15.12 What Next?

The chapter introduced the concepts of first-class and higher-order functions and generalized common computational patterns to construct a library of useful higher-order functions to process lists.

The next chapter continues to examine those concepts and their implications for Haskell programming.

15.13 Exercises

1. Suppose you need a Haskell function `times` that takes a list of integers (type `Integer`) and returns the product of the elements (e.g. `times [2,3,4]` returns `24`). Define the following Haskell functions.

    a. Function `times1` that uses the Prelude function `foldr` (or `foldr'` from this chapter).

b. Function `times2` that uses backward recursion to compute the product. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `product`.)

c. Function `times3` that uses forward recursion to compute the product. (Hint: use a tail-recursive auxiliary function with an accumulating parameter.)

d. Function `times4` that uses function `foldl'` from the Haskell library `Data.List`.

2. For each of the following specifications, define a Haskell function that has the given arguments and result. Use the higher order library functions (from this chapter) such as `map`, `filter`, `foldr`, and `foldl` as appropriate.

   a. Function `numof` takes a value and a list and returns the number of occurrences of the value in the list.

   b. Function `ellen` takes a list of character strings and returns a list of the lengths of the corresponding strings.

   c. Function `ssp` takes a list of integers and returns the sum of the squares of the positive elements of the list.

3. Suppose you need a Haskell function `sumSqNeg` that takes a list of integers (type `Integer`) and returns the sum of the squares of the negative values in the list.

   Define the following Haskell functions. Use the higher order library functions (from this chapter) such as `map`, `filter`, `foldr`, and `foldl` as appropriate.

   a. Function `sumSqNeg1` that is backward recursive. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `sum`.)

   b. Function `sumSqNeg2` that is tail recursive. (Use recursion directly. Do not use the list-folding Prelude functions such as `foldr` or `sum`.)

   c. Function `sumSqNeg3` that uses standard prelude functions such as `map`, `filter`, `foldr`, and `foldl`.

   d. Function `sumSqNeg4` that uses list comprehensions (Chapter 18).

4. Define a Haskell function

```
    scalarprod :: [Int] -> [Int] -> Int
```

   to compute the scalar product of two lists of integers (e.g. representing vectors).

   The *scalar product* is the sum of the products of the elements in corresponding positions in the lists. That is, the scalar product of two lists `xs` and `ys`, of length `n`, is:

$$\sum_{i=0}^{i=n} xs_i * ys_i$$

For example, `scalarprod [1,2,3] [3,3,3]` yields `18`.

5. Define a Haskell function `map2` that takes a list of functions and a list of values and returns the list of results of applying each function in the first list to the corresponding value in the second list.

15.14 Acknowledgements

In Summer 2016, I adapted and revised much of this work from the following sources:

- chapter 6 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]
- my notes on *Functional Data Structures (Scala)* [Cunningham 2016] which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [Chiusano 2015]

In 2017, I continued to develop this work as Chapter 5, Higher-Order Functions, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous Higher-Order Functions chapter into three chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 5.1-5.2 became the basis for new Chapter 15 (this chapter), Higher-Order Functions, section 5.3 became the basis for new Chapter 16, Haskell Function Concepts, and previous sections 5.4-5.6 became the basis for new Chapter 17, Higher-Order Function Examples.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

15.15 References

[Bird 1988]: Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[Bird 1998]: Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall, 1998.

[Bird 2015]: Richard Bird. *Thinking Functionally with Haskell*, Second Edition, Cambridge University Press, 2015.

[Chiusano 2015]: Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[Cunningham 2014]: H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[Cunningham 2016]: H. Conrad Cunningham, *Functional Data Structures (Scala)*, 2016. (Lecture notes based, in part, on chapter 3 [Chiusano 2015].)

[Thompson 2011]: Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

15.16 Terms and Concepts

Procedural abstraction, functions (first-class, higher-order), modularity, interface, function generalization and specialization, scope-commonality-variability (SCV) analysis, hot and frozen spots, data transformations, think like a functional programmer, common functional programming patterns (map, filter, fold, concatMap), duality theorem, strict and nonstrict functions.