Exploring Languages with Interpreters and Functional Programming Chapter 17

H. Conrad Cunningham

19 October 2018

Contents

Copyright (C) 2016, 2017, 2018, H. Conrad Cunningham
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.

17 Higher Order Function Examples

17.1 Chapter Introduction

The previous two chapters introduced the concepts of first-class and higher-order functions and their implications for Haskell programming.

This chapter looks at additional examples that use these higher-order programming concepts.

The Haskell module for this chapter's code is in `HigherOrderExamples.hs` except for the revised rational arithmetic module.

17.2 List-Breaking Operations

In a previous chapter we looked at the list-breaking functions `take` and `drop`. The Prelude also includes several higher-order list-breaking functions that take two arguments, a predicate that determines where the list is to be broken and the list to be broken.

Here we look at Prelude functions `takeWhile` and `dropWhile`. As you would expect, function `takeWhile` "takes" elements from the beginning of the list "while" the elements satisfy the predicate and `dropWhile` "drops" elements from the beginning of the list "while" the elements satisfy the predicate. The Prelude definitions are similar to the following:

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

dropWhile' :: (a -> Bool) -> [a] -> [a] -- dropWhile in Prelude
dropWhile'  p []  = []
dropWhile'  p xs@(x:xs')
    | p x        = dropWhile' p xs'
    | otherwise  = xs
```

Note the use of the pattern `xs@(x:xs')` in `dropWhile'`. This pattern matches a non-nil list with `x` and `xs'` binding to the head and tail, respectively, as usual. Variable `xs` binds to the entire list.

As an example, suppose we want to remove the leading blanks from a string. We can do that with the expression:

```
dropWhile ((==) ' ')
```

As with `take` and `drop`, the above functions can also be related by a "law". For all finite lists `xs` and predicates `p` on the same type:

```
takeWhile p xs ++ dropWhile p xs = xs
```

Prelude function `span` combines the functionality of `takeWhile` and `dropWhile` into one function. It takes a predicate `p` and a list `xs` and returns a tuple where the first element is the longest prefix (possibly empty) of `xs` that satisfies `p` and the second element is the remainder of the list.

```
span' :: (a -> Bool) -> [a] -> ([a],[a]) -- span in Prelude
span' _ xs@[]      =  (xs, xs)
span' p xs@(x:xs')
    | p x          =  let (ys,zs) = span' p xs' in (x:ys,zs)
    | otherwise    =  ([],xs)
```

Thus the following "law" holds for all finite lists `xs` and predicates `p` on same type:

```
span p xs == (takeWhile p xs, dropWhile p xs)
```

The Prelude also includes the function `break`, defined as follows:

```
break' :: (a -> Bool) -> [a] -> ([a],[a]) -- break in Prelude
break' p =  span (not . p)
```

17.3 List-Combining operations

In a previous chapter, we also looked at the function `zip`, which takes two lists and returns a list of pairs of the corresponding elements. Function `zip` applies an operation, in this case *tuple-construction*, to the corresponding elements of two lists.

We can generalize this pattern of computation with the function `zipWith` in which the operation is an argument to the function.

```
zipWith' :: (a->b->c) -> [a]->[b]->[c] -- zipWith in Prelude
zipWith' z  (x:xs)  (y:ys) = z x y : zipWith' z xs ys
zipWith' _  _     _        = []
```

Using a lambda expression to state the tuple-forming operation, the Prelude defines `zip` in terms of `zipWith`:

```
zip'' :: [a] -> [b] -> [(a,b)]  -- zip
zip'' = zipWith' (\x y -> (x,y))
```

Or can be written more simply as:

```
zip''' :: [a] -> [b] -> [(a,b)]  -- zip
zip''' = zipWith' (,)
```

The `zipWith` function also enables us to define operations such as the scalar product of two vectors in a concise way.

```
sp :: Num a => [a] -> [a] -> a
sp xs ys = sum' (zipWith' (*) xs ys)
```

The Prelude includes `zipWith3` for triples. Library `Data.List` has versions of `zipWith` that take up to seven input lists: `zipWith3` ··· `zipWith7`.

17.4 Rational Arithmetic Revisited

Remember the rational number arithmetic package developed in an earlier chapter. In that package's `Rational` module, we defined a function `eqRat` to compare two rational numbers for equality using the appropriate set of integer comparisons.

```
eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

We could have implemented the other comparison operations similarly.

Because the comparison operations are similar, they are good candidates for us to use a higher-order function. We can abstract out the common pattern of comparisons into a function that takes the corresponding integer comparison as an argument.

To compare two rational numbers, we can express their values in terms of a common denominator (e.g. `denom x * denom y`) and then compare the numerators using the integer comparisons. We can thus abstract the comparison into a higher-order function `compareRat` that takes an appropriate integer relational operator and the two rational numbers.

```
compareRat :: (Int -> Int -> Bool) -> Rat -> Rat -> Bool
compareRat r x y = r (numer x * denom y) (denom x * numer y)
```

Then we can define the rational number comparisons in terms of `compareRat`. (Note that we redefine function `eqRat` from the package in the earlier chapter.)

```
eqRat,neqRat,ltRat,leqRat,gtRat,geqRat :: Rat -> Rat -> Bool
eqRat   = compareRat (==)
neqRat  = compareRat (/=)
ltRat   = compareRat (<)
leqRat  = compareRat (<=)
gtRat   = compareRat (>)
geqRat  = compareRat (>=)
```

The Haskell module for the revised rational arithmetic module is in `RationalHO.hs`. The module `TestRationalHO.hs` is an extended version of the standard test script from Chapter 12 that tests the standard features of

the rational arithmetic module plus `eqRat`, `neqRat`, and `ltRat`. (It does not currently test `leqRat`, `gtRat`, or `geqRat`.)

17.5 Merge Sort

We defined the insertion sort in a previous chapter. It has an average-case time complexity of $O(n^2)$ where n is the length of the input list.

We now consider a more efficient function to sort the elements of a list into ascending order: *merge sort*. Merge sort works as follows:

- If the list has fewer than two elements, then it is already sorted.
- If the list has two or more elements, then we split it into two sublists, each with about half the elements, and sort each recursively.
- We merge the two ascending sublists into an ascending list.

We define function `msort` to be a polymorphic, higher-order function that has two parameters. The first (`less`) is the comparison operator and the second (`xs`) is the list to be sorted. Function `less` must be defined for every element that appears in the list to be sorted.

```
msort :: Ord a => (a -> a -> Bool) -> [a] -> [a]
msort _    []         = []
msort _    [x]        = [x]
msort less xs = merge less (msort less ls) (msort less rs)
    where n       = (length xs) `div` 2
          (ls,rs) = splitAt n xs
          merge _ [] ys     = ys
          merge _ xs []     = xs
          merge less ls@(x:xs) rs@(y:ys)
              | less x y  = x : (merge less xs rs)
              | otherwise = y : (merge less ls ys)
```

By nesting the definition of `merge`, we enabled it to directly access the the parameters of `msort`. In particular, we did not need to pass the comparison function to `merge`.

Assuming that `less` evaluates in constant time, the time complexity of `msort` is $O(n \log(n))$, where n is the length of the input list.

- Each call level requires splitting of the list in half and merging of the two sorted lists. This takes time proportional to the length of the list argument.
- Each call of `msort` for lists longer than one results in two recursive calls of `msort`.
- But each successive call of `msort` halves the number of elements in its input, so there are $O(\log(n))$ recursive calls.

So the total cost is $O(n \log(n))$. The cost is independent of distribution of elements in the original list.

We can apply `msort` as follows:

```
msort (<) [5, 7, 1, 3]
```

Function `msort` is defined in curried form with the comparison function first. This enables us to conveniently specialize `msort` with a specific comparison function. For example,

```
descendSort :: Ord a => [a] -> [a]
descendSort = msort (\ x y -> x > y)     -- or (>)
```

17.6 What Next?

This chapter and the two that proceeded it examined higher-order list programming concepts and features.

The next chapter examines list comprehensions, an alternative syntax for higher-order list processing that is likely comfortable for programmers coming from an imperative programming background.

17.7 Exercises

TODO: Add Wally World Marketplace POP Project and other exercises?

1. Define a Haskell function

   ```
   removeFirst :: (a -> Bool) -> [a] -> [a}
   ```

   so that `removeFirst p xs` removes the first element of `xs` that has the property `p`.

2. Define a Haskell function

   ```
   removeLast :: (a -> Bool) -> [a] -> [a]
   ```

   so that `removeLast p xs` removes the last element of `xs` that has the property `p`.

   How could you define it using `removeFirst`?

3. A list `s` is a *prefix* of a list `t` if there is some list `u` (perhaps nil) such that `s ++ u == t`. For example, the prefixes of string `"abc"` are `""`, `"a"`, `"ab"`, and `"abc"`.

   A list `s` is a *suffix* of a list `t` if there is some list `u` (perhaps nil) such that `u ++ s == t`. For example, the suffixes of `"abc"` are `"abc"`, `"bc"`, `"c"`, and `""`.

A list `s` is a *segment* of a list `t` if there are some (perhaps nil) lists `u` and `v` such that `u ++ s ++ v = t`. For example, the segments of string `"abc"` consist of the prefixes and the suffixes plus `"b"`.

Define the following Haskell functions. You may use functions appearing early in the list to implement later ones.

a. Define a function `prefix` such that `prefix xs ys` returns `True` if `xs` is a prefix of `ys` and returns `False` otherwise.

b. Define a function `suffixes` such that `suffixes xs` returns the list of all suffixes of list `xs`. (Hint: Generate them in the order given in the example of `"abc"` above.)

c. Define a function `indexes` such that `indexes xs ys` returns a list of all the positions at which list `xs` appears in list `ys`. Consider the first character of `ys` as being at position 0. For example, `indexes "ab" "abaabbab"` returns `[1,4,7]`. (Hint: Remember functions like `map`, `filter`, `zip`, and the functions you just defined.)

d. Define a function `sublist` such that `sublist xs ys` returns `True` if list `xs` appears as a segment of list `ys` and returns `False` otherwise.

4. Assume that the following Haskell type synonyms have been defined:

```
type Word = String -- word, characters left-to-right
type Line = [Word] -- line, words left-to-right
type Page = [Line] -- page, lines top-to-bottom
type Doc  = [Page] -- document, pages front-to-back
```

Further assume that values of type `Word` do not contain any space characters. Implement the following Haskell text-handling functions:

a. `npages` that takes a `Doc` and returns the number of `Page`s in the document.

b. `nlines` that takes a `Doc` and returns the number of `Line`s in the document.

c. `nwords` that takes a `Doc` and returns the number of `Word`s in the document.

d. `nchars` that takes a `Doc` and returns the number of `Char`s in the document (not including spaces of course).

e. `deblank` that takes a `Doc` and returns the `Doc` with all blank lines removed. A blank line is a line that contains no words.

f. `linetext` that takes a `Line` and returns the line as a `String` with the words appended together in left-to-right order separated by space characters and with a newline character `'\n'` appended to the right end of the line. (For example, `linetext ["Robert", "Khayat"]` yields `"Robert Khayat\n"`.)

g. `pagetext` that takes a `Page` and returns the page as a `String`—applies `linetext` to its component lines and appends the result in a top-to-bottom order.

h. `doctext` that takes a `Doc` and returns the document as a `String`—applies `pagetext` to its component lines and appends the result in a top-to-bottom order.

i. `wordeq` that takes a two `Doc`s and returns `True` if the two documents are *word equivalent* and `False` otherwise. Two documents are word equivalent if they contain exactly the same words in exactly the same order regardless of page and line structure. For example, `[[["Robert"],["Khayat"]]]` is word equivalent to `[[["Robert","Khayat"]]]`.

17.8 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, 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 (this chapter), 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.

17.9 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.

17.10 Terms and Concepts

List-breaking (splitting) operators, list-combining operators, rational arithmetic, merge sort.