Exploring Languages with Interpreters and Functional Programming Chapter 13

H. Conrad Cunningham

24 September 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 use of a browser that supports the display of MathML. A good choice as of September 2018 is a recent version of Firefox from Mozilla.

13 List Programming

13.1 Chapter Introduction

This chapter introduces the list data type and develops the fundamental programming concepts and techniques for first-order polymorphic functions to process lists.

The goals of the chapter are to:

- introduce Haskell syntax and semantics for programming constructs related to polymorphic list data structures
- examine correct Haskell functional programs consisting of first-order polymorphic functions that solve problems by processing lists and strings
- explore methods for developing Haskell list-processing programs that terminate and are efficient and elegant
- examine the concepts and use of data sharing in lists

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

13.2 Polymorphic List Data Type

As we have seen, to do functional programming, we construct programs from collections of pure functions. Given the same arguments, a *pure function* always returns the same result. The function application is thus *referentially transparent*.

Such a pure function does not have *side effects*. It does not modify a variable or a data structure in place. It does not throw an exception or perform input/output. It does nothing that can be seen from outside the function except return its value.

Thus the data structures in purely functional programs must be *immutable*, not subject to change as the program executes.

Functional programming languages often have a number of immutable data structures. However, the most salient one is the list.

We mentioned the Haskell list and string data types in Chapter 5. In this chapter, we look at lists in depth. Strings are just special cases of lists.

13.2.1 List: `[t]`

The primary built-in data structure in Haskell is the list, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with the notation `[t]` to denote a list of zero or more elements of type `t`.

A *list* is is hierarchical data structure. It is either *empty* or it is a pair consisting of a *head element* and a *tail* that is itself a *list* of elements.

The Haskell list is an example of an *algebraic data type*. We discuss that concept in a later section.

A matching pair of empty square brackets (`[]`), pronounced "nil", represents the empty list.

A colon (`:`), pronounced "cons", represents the *list constructor* operation between a *head* element on the left and a *tail* list on the right.

Example lists include:

```
[]
2:[]
3:(2:[])
```

The Haskell language adds a bit of *syntactic sugar* to make expressing lists easier. (By syntactic sugar, we mean notation that simplifies expression of a concept but that adds no new functionality to the language. The new notation can be defined in terms of other notation within the language.)

The cons operations *binds from the right*. Thus

```
5:(3:(2:[]))
```

can be written as:

```
5:3:2:[]
```

We can write this as a comma-separated sequence enclosed in brackets as follows:

```
[5,3,2]
```

Haskell supports two list selector functions, `head` and `tail`, such that

`head (h:t)` $\Longrightarrow$ `h`

where `h` is the head element of list, and

`tail (h:t)` $\Longrightarrow$ `t`

where `t` is the tail list.

Aside: Instead of `head`, Lisp uses `car` and other languages use `hd`, `first`, etc. Instead of `tail`, Lisp uses `cdr` and other languages use `tl`, `rest`, etc.

The Prelude library supports a number of other useful functions on lists. For example, `length` takes a list and returns its length.

Note that lists are *defined inductively*. That is, they are defined in terms of a base element `[]` and the list constructor operation cons (`:`). As you would expect, a form of mathematical induction can be used to prove that list-manipulating functions satisfy various properties. We will discuss in Chapter 25.

13.2.2 String: `String`

In Haskell, a *string* is treated as a list of characters. Thus the data type `String` is defined as a *type synonym*:

```
type String = [Char]
```

In addition to the standard list syntax, a `String` literal can be given by a sequence of characters enclosed in double quotes. For example, `"oxford"` is shorthand for `['o','x','f','o','r','d']`.

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code `\&` is used to separate any character sequences that are otherwise ambiguous.

Example: `"Hello\nworld!\n"` is a string that has two newline characters embedded.

Example: `"\12\&3"` represents the list `['\12','3']`.

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings.

Consider a function to compute the length of a finite string:

```
len :: String -> Int
len s  =  if s == [] then 0 else 1 + len (tail s)
```

Note that the argument string for the recursive application of `len` is simpler (i.e. shorter) than the original argument. Thus `len` will eventually be applied to a `[]` argument and, hence, `len`'s evaluation will terminate.

How efficient is this function (i.e. its time and space complexity)?

Consider the evaluation of the expression `len "five"`. using the evaluation model from Chapter 8.

```
     len "five"
⟹   if "five" == [] then 0 else 1 + len (tail "five")
⟹   if False then 0 else 1 + len (tail "five")
⟹   1 + len (tail "five")
⟹   1 + len "ive"
⟹   1 + (if "ive" == [] then 0 else 1 + len (tail "ive"))
⟹   1 + (if False then 0 else 1 + len (tail "ive"))
⟹   1 + (1 + len (tail "ive"))
⟹   1 + (1 + len "ve")
⟹   1 + (1 + (if "ve" == [] then 0 else 1 + len (tail "ve")))
⟹   1 + (1 + (if False then 0 else 1 + len (tail "ve")))
⟹   1 + (1 + (1 + len (tail "ve")))
⟹   1 + (1 + (1 + len "e"))
⟹   1 + (1 + (1 + (if "e" == [] then 0 else 1 + len (tail "e"))))
```

```
⟹  1 + (1 + (1 + (if False then 0 else 1 + len (tail "e"))))
⟹  1 + (1 + (1 + (1 + len (tail "e"))))
⟹  1 + (1 + (1 + (1 + len "")))
⟹  1 + (1 + (1 + (1 + (if "" == [] then 0 else 1 + len (tail "")))))
⟹  1 + (1 + (1 + (1 + (if True then 0 else 1 + len (tail "")))))
⟹  1 + (1 + (1 + (1 + 0)))
⟹  1 + (1 + (1 + 1))
⟹  1 + (1 + 2)
⟹  1 + 3
⟹  4
```

If `n` is the length of the list `xs`, then `len s` requires `4*n` reduction steps involving the recursive leg (first 16 steps above), 2 steps involving the nonrecursive leg (next 2 steps above), and `n+1` steps involving the additions (last five steps). Thus, the evaluation requires `5*n+3` reduction steps. Hence, the number of reduction steps in proportional to the length of the input list. The time complexity of the function is thus O(`length s`{.haskell]).

The largest expression above is

```
1 + (1 + (1 + (1 + (if "" == [] then 0 else 1 + len (tail "")))))
```

This expression has `n + 2` (6) binary operators, 2 unary operators, and 1 ternary operator. Counting arguments (as discussed in Chapter 8), it has size `2 * (n + 2) + 2 + 3` (or `2*n+9`). Hence, the amount of space required (given lazy evaluation) is also proportional to the length of the input list. The space complexity of the function is thus O(`length s`).

13.2.3 Polymorphic lists

The above definition of `len` only works for strings. How can we make it work for a list of integers or other elements?

For an arbitrary type `a`, we want `len` to take objects of type `[a]` and return an `Int` value. Thus its type signature could be:

```
len :: [a] -> Int
```

If `a` is a variable name (i.e. it begins with a lowercase letter) that does not already have a value, then the type expression `a` used as above is a *type variable*; it can represent an arbitrary type. All occurrences of a type variable appearing in a *type signature* must, of course, represent the same type.

An object whose type includes one or more type variables can be thought of having many different types and is thus described as having a *polymorphic type*. (The next subsection gives more detail on polymorphism in general.)

Polymorphism and first-class functions are powerful abstraction mechanisms: they allow irrelevant detail to be hidden.

Other examples of polymorphic list functions from the Prelude library include:

```
head :: [a] -> a
tail :: [a] -> [a]
(:)  :: a -> [a] -> [a]
```

13.3 Programming with List Patterns

In the factorial examples in Chapter 4, we used integer patterns and guards to break out various cases of a function definition into separate equations. Lists and other data types may also be used in patterns.

Pattern matching helps enable the *form of the algorithm* to match the *form of the data structure*. Or, as others may say, it helps in *following types to implementations*.

This is considered elegant. It is also pragmatic. The structure of the data often suggests the algorithm that is needed for a task.

In general, lists have two cases that need to be handled: the empty list and the nonempty list. Breaking a definition for a list-processing function into these two cases is usually a good place to begin.

13.3.1 Summing a list of integers: `sum'`

Consider a function `sum'` to sum all the elements in a finite list of integers. That is, if the list is

$$v_1, v_2, v_3, \cdots, v_n,$$

then the sum of the list is the value resulting from inserting the addition operator between consecutive elements of the list:

$$v_1 + v_2 + v_3 + \cdots + v_n.$$

Because addition is an *associative* operation (that is, $(x + y) + z = x + (y + z)$ for any integers x, y, and z), the above additions can be computed in any order.

What is the sum of an empty list?

Because there are no numbers to add, then, intuitively, zero seems to be the proper value for the sum.

In general, if some binary operation is inserted between the elements of a list, then the result for an empty list is the *identity* element for the operation. Since $0 + x = x = x + 0$ for all integers x, zero is the identity element for addition.

Now, how can we compute the sum of a nonempty list?

Because a nonempty list has at least one element, we can remove one element and add it to the sum of the rest of the list. Note that the "rest of the list" is a simpler (i.e. shorter) list than the original list. This suggests a recursive definition.

The fact that Haskell defines lists recursively as a cons of a head element with a tail list suggests that we structure the algorithm around the structure of the *beginning* of the list.

Bringing together the two cases above, we can define the function `sum'` in Haskell as follows. This is similar to the Prelude function `sum`.

```
{- Function sum' sums a list of integers. It is similar to
   function sum in the Prelude.
-}
sum' :: [Int] -> Int
sum' []     = 0           -- nil list
sum' (x:xs) = x + sum' xs -- non-nil list
```

- As noted previously, all of the text between the symbol "`--`" and the end of the line represents a *comment*; it is ignored by the Haskell interpreter.

  The text enclosed by the `{-` and `-}` is a block comment, that can extend over multiple lines.

- This definition uses two *legs*. The equation in the first leg is used for nil list arguments, the second for non-nil arguments.

- Note the `(x:xs)` pattern in the second leg. The "`:`" denotes the list constructor operation *cons*.

  If this pattern succeeds, then the head element of the list argument is bound to the variable `x` and the tail of the list argument is bound to the variable `xs`. These bindings hold for the right-hand side of the equation.

- The use of the cons in the pattern simplifies the expression of the case. Otherwise the second leg would have to be stated using the `head` and `tail` selectors as follows:

  ```
  sum' xs = head xs + sum' (tail xs)
  ```

- We use the simple name `x` to represent items of some type and the name `xs`, the same name with an `s` (for sequence) appended, to represent a list of that same type. This is a useful convention (adopted from the classic Bird and Wadler textbook [Bird 1988]) that helps make a definition easier to understand.

- Remember that patterns (and guards) are tested in the order of occurrence (i.e. left to right, top to bottom). Thus, in most situations, the cases should be listed from the most specific (e.g. nil) to the most general (e.g. non-nil).

- The length of a non-nil argument decreases by one for each successive recursive application. Thus (assuming the list is finite) `sum'` will eventually be applied to a `[]` argument and terminate.

For a list consisting of elements 2, 4, 6, and 8, that is, `2:4:6:8:[]`, function `sum'` computes

```
2 + (4 + (6 + (8 + 0)))
```

giving the integer result 20.

Function `sum'` is backward linear recursive; its time and space complexity are both $O(n)$, where n is the length of the input list.

We could, of course, redefine this to use a tail-recursive auxiliary function. With *tail call optimization*, the recursion could be converted into a loop. It would still be order $O(n)$ in time complexity (but with a smaller constant factor) but $O(1)$ in space.

13.3.2 Multiplying a list of numbers: `product'`

Now consider a function `product'` to multiply together a finite list of integers.

The product of an empty list is 1 (which is the identity element for multiplication).

The product of a nonempty list is the head of the list multiplied by the product of the tail of the list, except that, if a 0 occurs anywhere in the list, the product of the list is 0.

We can thus define `product'` with two base cases and one recursive case, as follows. This is similar to the Prelude function `product`.

```
product' :: [Integer] -> Integer
product' []     = 1
product' (0:_)  = 0
product' (x:xs) = x * product' xs
```

Note the use of the wildcard pattern underscore "`_`" in the second leg above. This represents a "don't care" value. In this pattern it matches the tail, but no value is bound; the right-hand side of the equation does not need the actual value.

0 is the *zero element* for the multiplication operation on integers. That is, for all integers x:

$$0 * x = x * 0 = 0$$

For a list consisting of elements 2, 4, 6, and 8, that is, `2:4:6:8:[]`, function `product'` computes:

```
2 * (4 * (6 * (8 * 1)))
```

which yields the integer result 384.

For a list consisting of elements 2, 0, 6, and 8, function `product'` "short circuits" the computation as:

```
2 * 0
```

Like `sum'`, function `product'` is backward linear recursive; it has a worst-case time complexity of $O(n)$, where n is the length of the input list. It terminates because the argument of each successive recursive call is one element shorter than the previous call, approaching the first base case.

As with `sum'`, we could redefine this to use a tail-recursive auxiliary function, which could evaluate in $O(1)$ space with tail call optimization.

Note that `sum'` and `product'` have similar computational patterns. In Chapter 15, we look at how to capture the commonality in a single higher-order function.

13.3.3 Length of a list: `length'`

As another example, consider the function for the length of a finite list that we discussed earlier (as `len`). Using list patterns we can define `length'` as follows:

```
length' :: [a] -> Int
length' []     = 0              -- nil list
length' (_:xs) = 1 + length' xs -- non-nil list
```

Note the use of the wildcard pattern underscore "`_`". In this pattern it matches the head, but no value is bound; the right-hand side of the equation does not need the actual value.

Given a finite list for its argument, does this function terminate? What are its time and space complexities?

This definition is similar to the definition for `length` in the Prelude.

13.3.4 Remove duplicate elements: `remdups`

Consider the problem of removing adjacent duplicate elements from a list. That is, we want to replace a group of adjacent elements having the same value by a single occurrence of that value.

As with the above functions, we let the form of the data guide the form of the algorithm, following the type to the implementation.

The notion of adjacency is only meaningful when there are two or more of something. Thus, in approaching this problem, there seem to be three cases to consider:

- The argument is a list whose first two elements are duplicates; in which case one of them should be removed from the result.
- The argument is a list whose first two elements are not duplicates; in which case both elements are needed in the result.
- The argument is a list with fewer than two elements; in which case the remaining element, if any, is needed in the result.

Of course, we must be careful that sequences of more than two duplicates are handled properly.

Our algorithm thus can examine the first two elements of the list. If they are equal, then the first is discarded and the process is repeated recursively on the list remaining. If they are not equal, then the first element is retained in the result and the process is repeated on the list remaining. In either case the remaining list is one element shorter than the original list. When the list has fewer than two elements, it is simply returned as the result.

If we restrict the function to lists of integers, we can define Haskell function `remdups` as follows:

```
remdups :: [Int] -> [Int]
remdups (x:y:xs)
    | x == y = remdups (y:xs)
    | x /= y = x : remdups (y:xs)
remdups xs = xs
```

- Note the use of the pattern `(x:y:xs)`. This pattern match succeeds if the argument list has at least two elements: the head element is bound to `x`, the second element to `y`, and the tail list to `xs`.
- Note the use of guards to distinguish between the cases where the two elements are equal (`==`) and where they are not equal (`/=`).
- In this definition the case patterns overlap, that is, a list with at least two elements satisfies both patterns. But since the cases are evaluated top to bottom, the list only matches the first pattern. Thus the second pattern just matches lists with fewer than two elements.

What if we wanted to make the list type polymorphic instead of just integers?

At first glance, it would seem to be sufficient to give `remdups` the polymorphic type `[a] -> [a]`. But the guards complicate the situation a bit.

Evaluation of the guards requires that Haskell be able to compare elements of the polymorphic type `a` for equality (`==`) and inequality (`/=`). For some types these comparisons may not be supported. (For example, suppose the elements are functions.) Thus we need to restrict the polymorphism to types in which the comparisons are supported.

We can restrict the range of types by using a *context* predicate. The following type signature restricts the polymorphism of type variable `a` to the built-in *type class* `Eq`, the group of types for which both equality (`==`) and inequality (`/=`) comparisons have been defined:

```
remdups :: Eq a => [a] -> [a]
```

Another useful context is the class `Ord`, which is an *extension* of class `Eq`. `Ord` denotes the class of objects for which the relational operators `<`, `<=`, `>`, and `>=` have been defined in addition to `==` and `/=`.

Note: Chapter 22 explores the concepts of type class, instances, and overloading in more depth.

In most situations the type signature can be left off the declaration of a function. Haskell then attempts to infer an appropriate type. For `remdups`, the type inference mechanism would assign the type `Eq [a] => [a] -> [a]` . However, in general, it is good practice to give explicit type signatures.

Like the previous functions, `remdups` is backward linear recursive; it takes a number of steps that is proportional to the length of the list. This function has a recursive call on both the duplicate and non-duplicate legs. Each of these recursive calls uses a list that is shorter than the previous call, thus moving closer to the base case.

13.3.5 More list patterns

The following table shows Haskell parameter patterns, corresponding arguments, and the result of the attempted match.

Pattern	**Argument**	**Succeeds?**	**Bindings**
`1`	`1`	yes	none
`x`	`1`	yes	`x` ← `1`
`(x:y)`	`[1,2]`	yes	`x` ← `1`, `y` ← `[2]`
`(x:y)`	`[[1,2]]`	yes	`x` ← `[1,2]`, `y` ← `[]`
`(x:y)`	`["olemiss"]`	yes	`x` ← `"olemiss"`, `y` ← `[]`
`(x:y)`	`"olemiss"`	yes	`x` ← `'o'`, `y` ← `"lemiss"`
`(1:x)`	`[1,2]`	yes	`x` ← `[2]`
`(1:x)`	`[2,2]`	no	none
`(x:_:_:y)`	`[1,2,3,4,5,6]`	yes	`x` ← `1`, `y` ← `[4,5,6]`
`[]`	`[]`	yes	none
`[x]`	`["Cy"]`	yes	`x` ← `"Cy"`
`[1,x]`	`[1,2]`	yes	`x` ← `2`
`[x,y]`	`[1]`	no	none
`(x,y)`	`(1,2)`	yes	`x` ← `1`, `y` ← `2`

13.4 Data Sharing

Suppose we have the declaration:

```
xs = [1,2,3]
```

As we learned in the data structures course, we can implement this list as a singly linked list `xs` with three cells with the values `1`, `2`, and `3`, as shown in the figure below.

Consider the following declarations (which are illustrated in Figure 13-1):

```
ys = 0:xs
zs = tail xs
```

where

- `0:xs` returns a list that has a new cell containing `0` in front of the previous list
- `tail xs` returns the list consisting of the last two elements of `xs`

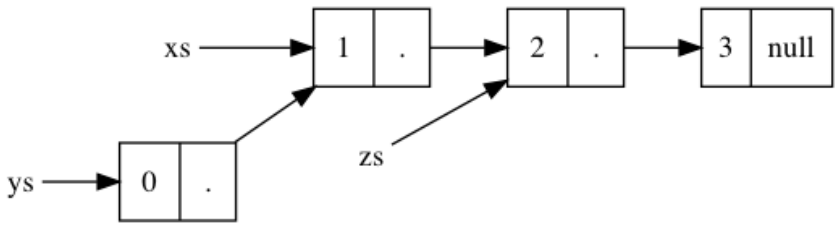

Figure 13-1: Data sharing in lists

If the linked list `xs` is immutable (i.e. the values and pointers in the three cells cannot be changed), then neither of these operations requires any copying.

- The first just constructs a new cell containing `0`, links it to the first cell in list `xs`, and initializes `ys` with a reference to the new cell.
- The second just returns a reference to the second cell in list `xs` and initializes `zs` with this reference.
- The original list `xs` is still available, unaltered.

This is called *data sharing*. It enables the programming language to implement immutable data structures efficiently, without copying in many key cases.

Also, such functional data structures are *persistent* because existing references are never changed by operations on the data structure.

Consider evaluation of the expression `head xs`. It must create a copy of the head element (in this case `1`). The result does not share data with the input list.

Similarly, the list returned by function `remdups` (defined above) does not share data with its input list.

13.4.1 Preconditions for `head` and `tail`

What should `tail` return if the list is nil?

One choice is to return a nil list `[]`. However, it seems illogical for an empty list to have a tail.

Consider a typical usage of the `tail` function. It is normally an error for a program to attempt to get the tail of an empty list. Moreover, a program can efficiently check whether a list is empty or not. So, in this case, it is better to consider `tail` a partial function.

Thus, Haskell defines both `tail` and `head` to have the precondition that their parameters are non-nil lists. If we call either with a nil list, then it will terminate execution with a standard error message.

13.4.2 Dropping elements from beginning of list

We can generalize `tail` to a function `drop'` that removes the first `n` elements of a list as follows, (This function is called `drop` in the Prelude.)

```
drop' :: Int -> [a] -> [a]
drop' n xs | n <= 0 = xs
drop' _ []          = []
drop' n (_:xs)      = drop' (n-1) xs
```

Consider the example:

```
drop 2 "oxford" ⟹ ··· "ford"
```

This function takes a different approach to the "empty list" issue than `tail` does. Although it is illogical to take the `tail` of an empty list, dropping the first element from an empty list seems subtly different. Given that we often use `drop'`‘ in cases where the length of the input list is unknown, dropping the first element of an empty list does not necessarily indicate a program error.

Suppose instead that `drop'` would trigger an error when called with an empty list. To avoid this situation, the program might need to determine the length of the list argument. This is inefficient, usually requiring a traversal of the entire list to count the elements. Thus the choice for `drop'` to return a nil is also pragmatic.

The `drop'` function is tail recursive. The result list shares space with the input list.

The `drop'` function terminates when either the list argument is empty or the integer argument is 0 or negative. The function eventually terminates because each recursive call both shortens the list and decrements the integer.

What is the time complexity of `drop'`?

There are two base cases. For the first leg, the function must terminate in $O(max(1,$`n` $))$ steps. For the second leg, the function must terminate within $O($`length xs` $)$ steps. So the function must terminate within $O(min(max(1,$`n` $),$`length xs` $))$ steps.

What is the space complexity of `drop'`?

This tail recursive function evaluates in constant space when optimized.

13.4.3 Taking elements from the beginning of a list

Similarly, we can generalize `head'` to a function `take` that takes a number `n` and a list and returns the first `n` elements of the list.

```
take' :: Int -> [a] -> [a]
take' n _ | n <= 0 = []
take' _ []         = []
take' n (x:xs)     = x : take' (n-1) xs
```

Consider the following questions for this function?

- What is returned when the list argument is nil?
- Does evaluation of this function terminate?
- Does the result share data with the input?
- Is the function tail recursive?
- What are its time and space complexities?

Consider the example:

```
take 2 "oxford" ⟹ ··· "ox"
```

13.5 What Next?

This chapter examined programming with the list data type using first-order polymorphic functions. The next chapter continues the discussion of list programming, introducing infix operations and more examples.

13.6 Exercises

1. Answer the following questions for the `take'` function defined in this chapter:
   - What is returned when the list argument is nil?
   - Does evaluation of the function terminate?
   - Does the result share data with the input?
   - Is the function tail recursive?
   - What are its time and space complexities?
2. Write a Haskell function `maxlist` to compute the maximum value in a nonempty list of integers. Generalize the function by making it polymorphic, accepting a value from any ordered type.
3. Write a Haskell function `adjpairs` that takes a list and returns the list of all pairs of adjacent elements. For example, `adjpairs [2,1,11,4]` returns `[(2,1), (1,11), (11,4)]`.
4. Write a Haskell function `mean` that takes a list of integers and returns the mean (i.e. average) value for the list.
5. Write the following Haskell functions using tail recursion:
   a. `sum''` with same functionality as `sum'`
   b. `product''` with the same functionality as `product'`

13.7 Acknowledgements

In Summer 2016, I adapted and revised much of this work from previous work:

- chapter 5 of my *Notes on Functional Programming with Haskell* [Cunningham 2014] which is influenced by [Bird 1988] (later editions are [Bird 1998] and [Bird 2015])
- 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 4, List Programming, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous List Programming chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 4.1-4.4 became the basis for new Chapter 13 (this chapter), List Programming, and previous sections 4.5-4.8 became the basis for Chapter 14, Infix Operators and List Programming Examples. I moved the discussion of "kinds of polymorphism" to new Chapter 5 and "local definitions" to new Chapter 9.

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.

13.8 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].)

13.9 Terms and Concepts

Type class (`Eq`, `Ord`, context predicate), lists (polymorphic, immutable, persistent, data sharing, empty/nil, nonempty), string, list and string operations (cons, head, tail, pattern matching, wildcard pattern, length), inductive definitions, operator binding, syntactic sugar, type synonym, type variable, type signature, follow the types to implementations, let the form of the data guide the form of the algorithm, associativity, identity element, zero element, termination, time and space complexity, adjacency,