Exploring Languages with Interpreters and Functional Programming Chapter 27

H. Conrad Cunningham

04 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

27 Text Processing Example

27.1 Chapter Introduction

Chapter 26 illustrates how to synthesize function definitions from their specifications.

This chapter (27) applies these program synthesis techniques to a larger set of examples on text processing.

27.2 Text Processing Example

In this section we develop a text processing package similar to the one in Section 4.3 of the Bird and Wadler textbook [3]. The text processing package in the Haskell standard Prelude is slightly different in its treatment of newline characters.

A textual document can be viewed in many different ways. At the lowest level, we can view it as just a character string and define a type synonym as follows:

```haskell
type Text = String
```

However, for other purposes, we may want to consider the document as having more structure (i.e., view it as a sequence of words, lines, paragraphs, pages, etc). We sometimes want to convert the text from one view to another.

Consider the problem of converting a `Text` document to the corresponding sequence of lines. Suppose that in the `Text`{.haskell document, the newline characters `'\n'`{.haskell serve as *separators* of lines, not themselves part of the lines. Because each line is a sequence of characters, we define a type synonym `Line` as follows:

```haskell
type Line = String
```

We want a function `lines'` that will take a `Text` document and return the corresponding sequence of lines in the document. The function has the type signature:

```haskell
lines' :: Text -> [Line]
```

For example, the Haskell expression

```haskell
lines' "This has\nthree\nlines"
```

yields:

```haskell
["This has", "three ", "lines"]
```

Writing function `lines'` is not trivial. However, its inverse `unlines'` is quite easy. Function `unlines'` takes a list of `Line`s, inserts a newline character between each pair of adjacent lines, and returns the `Text` document resulting from the concatenation.

```
unlines' :: [Line] -> Text
```

Let's see if we can develop `lines'` from `unlines'`.

The basic computational pattern for function `unlines'` is a folding operation. Because we are dealing with the construction of a list and the list constructors are nonstrict in their right arguments, a `foldr` operation seems more appropriate than a `foldl` operation.

To use `foldr`, we need a binary operation that will append two lines with a newline character inserted between them. The following, a bit more general, operation `insert'` will do that for us. The first argument is the element that is to be inserted between the two list arguments.

```
insert' :: a -> [a] -> [a] -> [a]
insert' a xs ys = xs ++ [a] ++ ys -- insert.1
```

Informally, it is easy to see that `(insert' a)` is an associative operation but that it has no right (or left) identity element.

Given that `(insert' a)` has no identity element, there is no obvious "seed" value to use with `fold`. Thus we will need to find a different way to express `unlines'`.

If we restrict the domain of `unlines'` to non-nil lists of lines, then we can use `foldr1`, a right-folding operation defined over non-empty lists (in the Prelude). This function does not require an identity element for the operation. Function `foldr1` can be defined as follows:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Note: There is a similar function (in the Prelude), `foldl1` that takes a non-nil list and does a left-folding operation.

Thus we can now define `unlines'` as follows:

```
unlines' :: [Line] -> Text
unlines' xss = foldr1 (insert' '\n') xss
```

Given the definition of `unlines'`, we can now specify what we want `lines'` to do. It must satisfy the following specification for any *non-nil* `xss` of type `[Line]`:

```
lines' (unlines' xss) = xss
```

That is, `lines'` is the inverse of `unlines'` for all non-nil arguments.

The first step in the synthesis of `lines'` is to guess at a possible structure for the `lines'` function definition. Then we will attempt to calculate the unknown pieces of the definition.

Because `unlines'` uses a right-folding operation, it is reasonable to guess that its inverse will also use a right-folding operation. Thus we speculate that `lines'` can be defined as follows, given an appropriately defined operation `op` and "seed value" `a`.

```
lines' :: Text -> [Line]
lines' = foldr op a
```

Because of the definition of `foldr` and type signature of `lines'`, function `op` must have the type signature

```
op :: Char -> [Line] -> [Line]
```

and `a` must be the right identity of `op` and hence have type `[Line]`.

The task now is to find appropriate definitions for `op` and `a`.

From what we know about `unlines'`, `foldr1`, `lines'`, and `foldr`, we see that the following identities hold. (These can be proved, but we do not do so here.)

```
unlines' [xs]         = xs                   -- unlines.1
unlines' ([xs]++xss) =
    insert' '\n' xs (unlines' xss)           -- unlines.2

lines' []             = a                    -- lines.1
lines' ([x]++xs)      = op x (lines' xs)     -- lines.2
```

Note the names we give each of the above identities (e.g., `unlines.1`). We use these equations to justify our steps in the calculations below.

Next, let us calculate the unknown identity element `a`. The strategy is to transform `a` by use of the definition and derived properties for `unlines'` and the specification and derived properties for `lines'` until we arrive at a constant.

```
    a
= { lines.1 (right to left) }
    lines' []
= { unlines'.1 (right to left) with xs = [] }
    lines' (unlines' [[]])
= { specification of lines' (left to right) }
    [[]]
```

Therefore we define `a` to be `[[]]`. Note that because of `lines.1`, we have also defined `lines'` in the case where its argument is `[]`.

Now we proceed to calculate a definition for `op`. Remember that we assume `xss /= []`.

As above, the strategy is to use what we know about `unlines'` and what we have assumed about `lines'` to calculate appropriate definitions for the unknown parts of the definition of `lines'`. We first expand our expression to bring in `unlines'`.

```
    op x xss
= { specification for lines' (right to left) }
    op x (lines' (unlines' xss))
= { lines.2 (right to left) }
    lines' ([x] ++ unlines' xss)
```

Because there seems no other way to proceed with our calculation, we distinguish between cases for the variable `x`. In particular, we consider the case where `x` is the line separator and the case where it is not, i.e., `x == '\n'` and `x /= '\n'`.

Case `x == '\n'`:

Our strategy is to absorb the `'\n'` into the `unlines'`, then apply the specification of `lines'`.

```
    lines' ("\n" ++ unlines' xss)
= { [] is the identity for ++ }
    lines' ([]  ++ "\n" ++ unlines' xss)
= { insert.1 (right to left) with a == '\n' }
    lines' (insert' '\n' [] (unlines' xss))
= { unlines.2 (right to left) }
    lines' (unlines' ([[]] ++ xss))
= { specification of lines' (left to right) }
    [[]] ++ xss
```

Thus `op '\n' xss = [[]] ++ xss`.

Case `x /= '\n'`:

Our strategy is to absorb the `[x]` into the `unlines'`, then apply the specification of `lines`.

```
    lines' ([x] ++ unlines' xss)
= { Assumption xss /= [], let xss = [ys] ++ yss }
    lines' ([x] ++ unlines' ([ys] ++ yss))
= { unlines.2 (left to right) with a = '\n' }
    lines' ([x] ++ insert' '\n' ys (unlines' yss))
```

= { `insert.1` (left to right) }

```
lines' ([x] ++ (ys ++ "\n" ++ unlines' yss))
```

= { ++ associativity }

```
lines' (([x] ++ ys) ++ "\n" ++ unlines' yss)
```

= { `insert.1` (right to left) }

```
lines' (insert' '\n' ([x]++ys) (unlines' yss))
```

= { `unlines.2` (right to left) }

```
lines' (unlines' ([[x]++ys] ++ yss))
```

= { specification of `lines'` (left to right) }

```
[[x]++ys] ++ yss
```

Thus, for `x /= '\n'` and `xss /= []`:

```
op x xss = [[x] ++ head xss] ++ (tail xss)
```

To generalize `op` like we did `insert'` and give it a more appropriate name, we define `op` to be `breakOn '\n'` as follows:

```
breakOn :: Eq a => a -> a -> [[a]] -> [[a]]
breakOn a x []               = error "breakOn applied to nil"
breakOn a x xss | a == x     = [[]] ++ xss
                | otherwise = [[x] ++ ys] ++ yss
                              where (ys:yss) = xss
```

Thus, we get the following definition for `lines'`:

```
lines' :: Text -> [Line]
lines' xs = foldr (breakOn '\n') [[]] xs
```

Let's review what we have done in this example. We have synthesized `lines'` from its specification and the definition for `unlines'`, its inverse. Starting from a precise, but non-executable specification, and using only equational reasoning, we have derived an executable definition of the required function.

The technique used is a familiar one in many areas of mathematics:

1. We guessed at a form for the solution.
2. We then calculated the unknowns.

Note: The definition of `lines` and `unlines` in the standard Prelude treat newlines as line *terminators* instead of line separators. Their definitions follow.

```
lines :: String -> [String]
lines "" =  []
lines s  = l : (if null s' then [] else lines (tail s'))
           where (l, s') = break ('\n'==) s
```

```
unlines :: [String] -> String
unlines  = concat . map (\l -> l ++ "\n")
```

27.2.1 Word processing

Let's continue the text processing example from the previous subsection a bit further. We want to synthesize a function to break a text into a sequence of words.

For the purposes here, we define a *word* as any nonempty sequence of characters not containing a space or newline character. That is, a group of one or more spaces and newlines separate words. We introduce a type synonym for words.

```
type Word = String
```

We want a function `words'` that breaks a line up into a sequence of words. Function `words'` thus has the following type signature:

```
words' :: Line -> [Word]
```

For example, expression

```
words' "Hi there"
```

yields:

```
["Hi", "there"]
```

As in the synthesis of `lines'`, we proceed by defining the "inverse" function first, then we calculate the definition for `words'`.

All `unwords'` needs to do is to insert a space character between adjacent elements of the sequence of words and return the concatenated result. Following the development in the previous subsection, we can thus define `unwords'` as follows.

```
unwords' :: [Word] -> Line
unwords' xs = foldr1 (insert'  ' ') xs
```

Using calculations similar to those for `lines'`, we derive the inverse of `unwords'` to be the following function:

```
foldr (breakOn' ') [[]]
```

However, this identifies zero-length words where there are adjacent spaces. We need to filter those out.

```
words' :: Line -> [Word]
words' = filter (/= []) . foldr (breakOn' ') [[]]
```

Note that

```
words' (unwords' xss)  =  xss
```

for all `xss` of type `[Word]`, but that

```
unwords' (words' xs)  =   xs
```

for some `xs` of type `Line`. The latter is undefined when `words' {.haskell} xs` returns `[]`. Where it is defined, adjacent spaces in `xs` are replaced by a single space in `unwords' (words' xs)`.

Note: The functions `words` and `unwords` in the standard Prelude differ in that `unwords [] = []`, which is more complete.

27.2.2 Paragraph processing

Let's continue the text processing example one step further and synthesize a function to break a sequence of lines into paragraphs.

For the purposes here, we define a *paragraph* as any nonempty sequence of nonempty lines. That is, a group of one or more empty lines separate paragraphs. As above, we introduce an appropriate type synonym:

```
type Para = [Line]
```

We want a function `paras'` that breaks a sequence of lines into a sequence of paragraphs:

```
paras' :: [Line] -> [Para]
```

For example, expression

```
paras' ["Line 1.1","Line 1.2","","Line 2.1"]
```

yields:

```
[["Line 1.1","Line 1.2"],["Line 2.1"]]
```

As in the synthesis of `lines'` and `words'`, we can start with the inverse and calculate the definition of `paras'`. The inverse function `unparas'` takes a sequence of paragraphs and returns the corresponding sequence of lines with an empty line inserted between adjacent paragraphs.

```
unparas' :: [Para] -> [Line]
unparas' = foldr1 (insert' [])
```

Using calculations similar to those for `lines'` and `words'`, we can derive the following definitions:

```
paras' :: [Line] -> [Para]
paras' = filter (/= []) . foldr (breakOn []) [[]]
```

The `filter (/= [])` operation removes all "empty paragraphs" corresponding to two or more adjacent empty lines.

Note: There are no equivalents of `paras'` and '`unparas'` in the standard prelude. As with `unwords`, `unparas'` should be redefined so that `unparas' [] = []`, which is more complete.

27.2.3 Other text processing functions

Using the six functions in our text processing package, we can build other useful functions.

1. **Count the lines in a text.**

   ```
   countLines :: Text -> Int
   countLines = length . lines'
   ```

2. **Count the words in a text.**

   ```
   countWords :: Text -> Int
   countWords = length . concat . (map words') . lines'
   ```

   An alternative using a list comprehension is:

   ```
   countWords xs =
       length [ w | l <- lines' xs, w <- words' l]
   ```

3. **Count the paragraphs in a text.**

   ```
   countParas :: Text -> Int
   countParas = length . paras' . lines'
   ```

4. **Normalize text by removing redundant empty lines and spaces.**

   The following functions take advantage of the fact that `paras'` and `words'` discard empty paragraphs and words, respectively.

   ```
   normalize :: Text -> Text
   normalize = unparse . parse

   parse :: Text -> [[[Word]]]
   parse = (map (map words')) . paras' . lines'

   unparse :: [[[Word]]] -> Text
   unparse = unlines' . unparas' . map (map unwords')
   ```

   We can also state `parse` and `unparse` in terms of list comprehensions.

   ```
   parse xs =
       [ [words' l | l <- p] | p <- paras' (lines' xs) ]

   unparse xssss =
       unlines' (unparas' [ [unwords' l | l<-p] | p<-xssss])
   ```

Section 4.3.5 of the Bird and Wadler textbook [3] goes on to build functions to fill and left-justify lines of text.

27.3 What Next?

Chapter 26 illustrates how to synthesize (i.e., derive or calculate) function definitions from their specifications. This chapter (27) applies these program synthesis techniques to larger set of examples on text processing.

No subsequent chapter depends explicitly upon the program synthesis content from these chapters. However, if practiced regularly, the techniques explored in this chapter can enhance a programmer's ability to solve problems and construct correct functional programming solutions.

27.4 Exercises

TODO

27.5 Acknowledgements

In Summer 2018, I adapted and revised this chapter and the next from Chapter 12 of my *Notes on Functional Programming with Haskell* [9].

These previous notes drew on the presentations in the first edition of the classic Bird and Wadler textbook [3] and other functional programming sources [1,2,15,17,18]. They were also influenced by my research, study, and teaching related to program specification, verification, derivation, and semantics [[4]; [5]; [6]; [7]; [8]; [10]; [11]; [12]; [13]; [14]; [16]; vanGesteren1990].

I incorporated this work as new Chapter 26, Program Synthesis, and new Chapter 27, Text Processing (this chapter), in the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming* and continue to revise it.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

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.

27.6 Terms and Concepts

Program synthesis, synthesizing a function from its inverse, text processing, line, word, paragraph, terminator, separator.

27.7 References

[1] Richard Bird. 1998. *Introduction to functional programming using Haskell* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] Richard Bird. 2015. *Thinking functionall with Haskell* (First ed.). Cambridge University Press, Cambridge, UK.

[3] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[4] K. Mani Chandy and Jayadev Misra. 1988. *Parallel program design: A foundation.* Addison-Wesley, Boston, Massachusetts, USA.

[5] Edward Cohen. 1990. *Programming in the 1990's: An introduction to the calculation of programs.* Springer, New York, New York, USA.

[6] H. Conrad Cunningham. 1989. The shared dataspace approach to concurrent computation: The Swarm programming model, notation, and logic. PhD thesis. Washington University, Department of Computer Science, St. Louis, Missouri, USA.

[7] H. Conrad Cunningham. 2006. *A programmer's introduction to predicate logic.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/notes/haskell_notes.pdf

[8] H. Conrad Cunningham. 2006. *Notes on program semantics and derivation.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/reports/umcis-1994-02.pdf

[9] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/csci450/notes/haskell_notes.pdf

[10] Edsger W. Dijkstra. 1976. Updating a sequential file. In *A discipline of programming.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 117--122.

[11] Edsger W. Dijkstra and Wim H. J. Feijen. 1988. *A method of programming.* Addison-Wesley, TBD.

[12] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate calculus and program semantics.* Springer, New York, New York, USA.

[13] David Gries. 1981. *Science of programming.* Springer, New York, New York, USA.

[14] David Gries and Fred B. Schneider. 2013. *A logical approach to discrete math.* Springer, New York, New York, USA.

[15] Robert R. Hoogerwoord. 1989. The design of functional programs: A calculational approach. PhD thesis. Eindhoven Technical University, Eindhoven, The Netherlands.

[16] Anne Kaldewaij. 1990. *Programming: The derivation of algorithms.* Prentice Hall, New York, New York, USA.

[17] Simon Thompson. 1996. *Haskell: The craft of programming* (First ed.). Addison-Wesley, Boston, Massachusetts, USA.

[18] E. Peter Wentworth. 1990. *Introduction to functional programming using RUFL*. Rhodes University, Department of Computer Science, Grahamstown, South Africa.