Exploring Languages with Interpreters and Functional Programming
Chapter 18

H. Conrad Cunningham

16 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.

18 More List Processing

18.1 Chapter Introduction

Previous chapters examined first-order and higher-order list programming. In particular, Chapter 15 explored the standard higher order functions such as `map`, `filter`, and `concatMap` and Chapter 16 explored function concepts such as function composition.

This chapter examines list comprehensions. This feature does not add new power to the language; the computations can be expressed with combinations of features from the previous chapters. But list comprehensions are often easier to write and to understand than equivalent compositions of `map`, `filter`, `concatMap`, etc.

The source file for the code in this chapter is in `MoreLists.hs`.

18.2 Sequences

Haskell provides a compact notation for expressing arithmetic sequences.

An arithmetic sequence (or progression) is a sequence of elements from an enumerated type (i.e. a member of class `Enum`) such that consecutive elements have a fixed difference. `Int`, `Integer`, `Float`, `Double`, and `Char` are all predefined members of this class.

- `[m..n]` produces the list of elements from `m` up to `n` in steps of one if `m <= n`. It produces the nil list otherwise.

  Examples:

  - `[1..5]` $\Longrightarrow$ `[1,2,3,4,5]`
  - `[5..1]` $\Longrightarrow$ `[]`

  This feature is implemented with Prelude function `enumFromTo` applied as `enumFromTo m n`.

- `[m,m'..n]` produces the list of elements from `m` in steps of `m'-m`. If `m' > m` then the list is increasing up to `n`. If `m' < m`, then it is decreasing.

  Examples:

  - `[1,3..9]` $\Longrightarrow$ `[1,3,5,7,9]`
  - `[9,8..5]` $\Longrightarrow$ `[9,8,7,6,5]`
  - `[9,8..11]` $\Longrightarrow$ `[]`

  This feature is implemented with Prelude function `enumFromThenTo` applied as `enumFromThenTo m' m n`.

- `[m..]` and `[m,m'..]` produce potentially infinite lists beginning with `m` and having steps 1 and `m'-m` respectively.

These features are implemented with Prelude functions `enumFrom` applied as `enumFrom m` and `enumFromThen` applied as `enumFromThen m m'`.

Of course, we can provide our own functions for sequences. Consider the following function to generate a geometric sequence.

A geometric sequence (or progression) is a sequence of elements from an ordered, numeric type (i.e. a member of both classes `Ord` and `Num`) such that consecutive elements have a fixed ratio.

```
geometric :: (Ord a, Num a) => a -> a -> a -> [a]
geometric r m n | m > n     = []
                | otherwise = m : geometric r (m*r) n
```

Example: `geometric 2 1 10` $\Longrightarrow$ `[1,2,4,8]`

18.3 List Comprehensions

18.3.1 Syntax and semantics

The *list comprehension* is another powerful and compact notation for describing lists. A list comprehension has the form

`[` *expression* `|` *qualifiers* `]`

where *expression* is any Haskell expression.

The *expression* and the *qualifiers* in a comprehension may contain variables that are local to the comprehension. The values of these variables are bound by the *qualifiers*.

For each group of values bound by the qualifiers, the comprehension generates an element of the list whose value is the *expression* with the values substituted for the local variables.

There are three kinds of *qualifiers* that can be used in Haskell: generators, filters, and local definitions.

1. A *generator* is a qualifier of the form

   pat `<-` *exp*

   where *exp* is a list-valued expression. The generator extracts each element of *exp* that matches the pattern *pat* in the order that the elements appear in the list; elements that do not match the pattern are skipped.

   Example:

   - `[ n*n | n <- [1..5]]` $\Longrightarrow$ `[1,4,9,16,25]`

2. A *filter* is a Boolean-valued expression used as a *qualifier* in a list comprehension. These expressions work like the `filter` function; only values that

make the expression `True` are used to form elements of the list comprehension.

Example:

- `[ n*n | even n ]` $\Longrightarrow$ `(if even n then [n*n] else [])`

Above variable `n` is global to this expression, not local to the comprehension.

3. A *local definition* is a qualifier of the form

   `let` *pat* `=` *expr*

   introduces a local definition into the list comprehension.

   Example:

   - `[ n*n | let n = 2 ]` $\Longrightarrow$ `[4]`

The real power of list comprehensions come from using several qualifiers separated by commas on the right side of the vertical bar `|`.

- Generators appearing later in the list of qualifiers vary more quickly than those that appear earlier. Speaking operationally, the generation "loop" for the later generator is nested within the "loop" for the earlier.

  Example:

  - `[ (m,n) | m<-[1..3], n<-[4,5] ]` $\Longrightarrow$
    `[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]`

- Qualifiers appearing later in the list of qualifiers may use values generated by qualifiers appearing earlier, but not vice versa.

  Examples:

  - `[ n*n | n<-[1..10], even n ]` $\Longrightarrow$ `[4,16,36,64,100]`
  - `[ (m,n) | m<-[1..3], n<-[1..m] ]` $\Longrightarrow$
    `[ (1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]`

- The generated values may or may not be used in the *expression*.

  Examples:

  - `[ 27 | n<-[1..3]]` $\Longrightarrow$ `[27,27,27]`
  - `[ x | x<-[1..3], y<-[1..2]]` $\Longrightarrow$ `[1,1,2,2,3,3]`

18.3.2 Translating list comprehensions

List comprehensions are syntactic sugar. We can translate them into core Haskell features by applying the following identities.

1. For any expression `e`,

```
[ e | True ]
```

is equivalent to:

```
[ e ]
```

2. For any expression `e` and qualifier `q`,

```
[ e | q ]
```

is equivalent to:

```
[ e | q, True ]
```

3. For any expression `e`, boolean `b`, and sequence of qualifiers `Q`,

```
[ e | b, Q ]
```

is equivalent to:

```
if b then [ e | Q ] else []
```

4. For any expression `e`, pattern `p`, list-valued expression `l`, sequence of qualifiers `Q`, and fresh variable `ok`,

```
[ e | p <- l, Q ]
```

is equivalent to:

```
let ok p = [ e | Q ] -- p is a pattern
    ok _ = []
in concatMap ok l
```

5. For any expression `e`, declaration list `D`, and sequence of qualifiers `Q`,

```
[ e | let D, Q ]
```

is equivalent to:

```
let D in [ e | Q ]
```

Function `concatMap` and boolean value `True` are as defined in the Prelude.

As we saw in a previous chapter, `concatMap` applies a list-returning function to each element of an input list and then concatenates the resulting list of lists into a single list. Both `map` and `filter` can be defined in terms of `concatMap`.

Consider the list comprehension:

```
[ n*n | n<-[1..10], even n ]
```

a. Apply identity 4:

```
let ok n = [ n*n | even n ]
    ok _ = []
in concatMap ok [1..10]
```

b. Apply identity 2:

```
let ok n = [ n*n | even n, True ]
    ok _ = []
in concatMap ok [1..10]
```

c. Apply identity 3:

```
let ok n = if (even n) then [ n*n | True ]
    ok _ = []
in concatMap ok [1..10]
```

d. Apply identity 1:

```
let ok n = if (even n) then [ n*n ]
    ok _ = []
in concatMap ok [1..10]
```

18.4 Using List Comprehensions

This section gives several examples where list comprehensions can be used to solve problems and express the solutions conveniently.

18.4.1 Strings of spaces

Consider a function `spaces` that takes a number and generates a string with that many spaces.

```
spaces :: Int -> String
spaces n = [ ' ' | i<-[1..n]]
```

Note that when `n < 1` the result is the empty string.

18.4.2 Prime number test

Consider a Boolean function `isPrime` that takes a nonzero natural number and determines whether the number is *prime*. (Remember that a prime number is a natural number whose only natural number factors are 1 and itself.)

```
isPrime :: Int -> Bool
isPrime n | n > 1 = (factors n == [])
    where factors m = [ x | x<-[2..(m-1)], m `mod` x == 0 ]
isPrime   _       =  False
```

18.4.3 Squares of primes

Consider a function `sqPrimes` that takes two natural numbers and returns the list of squares of the prime numbers in the inclusive range from the first up to

the second.

```
sqPrimes :: Int -> Int -> [Int]
sqPrimes m n = [ x*x | x<-[m..n], isPrime x ]
```

Alternatively, this function could be defined using `map` and `filter` as follows:

```
sqPrimes' :: Int -> Int -> [Int]
sqPrimes' m n = map (\x -> x*x) (filter isPrime [m..n])
```

18.4.4 Doubling positive elements

We can use a list comprehension to define (our, by now, old and dear friend) the function `doublePos`, which doubles the positive integers in a list.

```
doublePos5 :: [Int] -> [Int]
doublePos5 xs = [ 2*x | x<-xs, 0 < x ]
```

18.4.5 Concatenating a list of lists of lists

Consider a program `superConcat` that takes a list of lists of lists and concatenates the elements into a single list.

```
superConcat :: [[[a]]] -> [a]
superConcat xsss = [ x | xss<-xsss, xs<-xss, x<-xs ]
```

Alternatively, this function could be defined using Prelude functions `concat` and `map` and functional composition as follows:

```
superConcat' :: [[[a]]] -> [a]
superConcat' = concat . map concat
```

18.4.6 First occurrence in a list

Consider a function `position` that takes a list and a value of the same type. If the value occurs in the list, `position` returns the position of the value's first occurrence; if the value does not occur in the list, `position` returns 0.

Strategy: *Solve a more general problem first, then use it to get the specific solution desired.*

In this problem, we generalize the problem to finding *all* occurrences of a value in a list. This more general problem is actually easier to solve.

```
positions :: Eq a => [a] -> a -> [Int]
positions xs x = [ i | (i,y)<-zip [1..length xs] xs, x == y]
```

Function `zip` is useful in pairing an element of the list with its position within the list. The subsequent filter removes those pairs not involving the value `x`. The "zipper" functions can be very useful within list comprehensions.

Now that we have the positions of all the occurrences, we can use `head` to get the first occurrence. Of course, we need to be careful that we return 0 when there are no occurrences of `x` in `xs`.

```
position :: Eq a => [a] -> a -> Int
position xs x = head ( positions xs x ++ [0] )
```

Because of lazy evaluation, this implementation of `position` is not as inefficient as it first appears. The function `positions` will, in actuality, only generate the head element of its output list.

Also because of lazy evaluation, the upper bound `length xs` can be left off the generator in `positions`. In fact, the function is more efficient to do so.

18.5 What Next?

This chapter examined list comprehensions. Although they do not add new power to the language, programs involving comprehensions are often easier to write and to understand than equivalent compositions of other functions.

The next two chapters discuss problem solving techniques. Chapter 19 discusses systematic generalization of functions. Chapter 20 surveys various problem-solving techniques uses in this textbook and other sources.

18.6 Exercises

1. Show the list (or string) yielded by each of the following Haskell list expressions. Display it using fully specified list bracket notation, e.g. expression `[1..5]` yields `[1,2,3,4,5]`.

   a. `[7..11]`

   b. `[11..7]`

   c. `[3,6..12]`

   d. `[12,9..2]`

   e. `[ n*n | n <- [1..10], even n ]`

   f. `[ 7 | n <- [1..4] ]`

   g. `[ x | (x:xs) <- [Did, you, study?] ]`

   h. `[ (x,y) | x <- [1..3], y <- [4,7] ]`

   i. `[ (m,n) | m <- [1..3], n <- [1..m] ]`

j. `take 3 [ [1..n] | n <- [1..] ]`

2. Translate the following expressions into expressions that use list comprehensions. For example, `map (*2) xs` could be translated to `[ x*2 | x <- xs ]`.

   a. `map (\x -> 2*x-1) xs`

   b. `filter p xs`

   c. `map (^2) (filter even [1..5])`

   d. `foldr (++) [] xss`

   e. `map snd (filter (p .  fst) (zip xs [1..]))`

18.7 Acknowledgements

In 2016 and 2017, I adapted and revised my previous notes to form Chapter 7, More List Processing and Problem Solving, in the 2017 version of this textbook. In particular, I drew the information on More List Processing from:

- chapter 7 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]

In Summer 2018, I divided the 2017 More List Processing and Problem Solving chapter back into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 7.2-7.3 (essentially chapter 7 of [Cunningham 2014]) became the basis for new Chapter 18, More List Processing (this chapter), and the Problem Solving discussion became the basis for new Chapter 20, Problem Solving.

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.

18.8 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.

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

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

18.9 Terms and Concepts

Sequence (arithmetic, geometric), list comprehension (generator, filter, local definition, multiple generators and filters), syntactic sugar, translating list comprehensions to function calls, prime numbers, solve a harder problem first.