CSci 450: Org. of Programming Languages
More List Processing and Problem Solving

H. Conrad Cunningham

9 October 2017

Contents

Acknowledgements: In Summer 2016, I adapted and revised this module from chapters 7 and 10 of my *Notes on Functional Programming with Haskell*.

In Spring 2017, I continued to develop this module. This included adding discussion of the tupling technique from chapter 12 of my *Notes on Functional Programming with Haskell* and referring to examples used in other chapers.

I continue to develop this module in Summer and Fall 2017.

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

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of October 2017 is a recent version of Firefox from Mozilla.

TODO:

- Add chapter goals and outcomes
- Make problem solving section less dependent on external references such as the classic Bird and Wadler textbook. (Probably need to add new examples to this and other chapters)
- Add exercises

7 More List Processing and Problem Solving

7.1 Chapter Introduction

TODO

7.2 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]`

7.3 List Comprehensions

7.3.1 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]`

7.3.2 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 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:

~~~{.haskell}
    [ n*n | n<-[1..10], even n ]
~~~

a. Apply identity 4:

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

b. Apply identity 2:

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

c. Apply identity 3:

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

d. Apply identity 1:

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

7.3.3 Strings of spaces

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

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

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

7.3.4 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.)

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

7.3.5 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])
```

7.3.6 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 ]
```

7.3.7 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
```

7.3.8 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.

7.4 Problem Solving

I approach computing science with the following philosophy:

- Programming is the essence of computing science.
- Problem solving is the essence of programming.

Here I consider programming as the process of analyzing a problem and formulating a solution suitable for execution on a computer. The solution should be correct, elegant, efficient, and robust. It should be expressed in a manner that is understandable, maintainable, and reusable. The solution should balance generality and specificity, abstraction and concreteness.

In my view, programming is far more than just coding. It subsumes the concerns of algorithms, data structures, and software engineering. It uses programming languages and software development tools. It uses the intellectual tools of mathematics, logic, linguistics, and computing science theory. Etc.

7.4.1 Polya's insights

The mathematician George Polya (1887–1985), a Professor of Mathematics at Stanford University, said the following in the preface to his book *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving*.

> Solving a problem means finding a way out of a difficulty, a way around an obstacle, attaining an aim which was not immediately attainable. Solving problems is the specific achievement of intelligence,

> and intelligence is the specific gift of mankind: solving problems can be regarded as the most characteristically human activity. ...
>
> Solving problems is a practical art, like swimming, or skiing, or playing the piano: you learn it only by imitation and practice. ... if you wish to learn swimming you have to go into the water, and if you wish to become a problem solver you have to solve problems.
>
> If you wish to derive the most profit from your effort, look out for such features of a problem at hand as may be useful in handling the problems to come. A solution that you have obtained by your own effort or one that you have read or heard, but have followed with real interest and insight, may become a *pattern* for you, a model that you can imitate with advantage in solving similar problems. ...
>
> Our knowledge about any subject consists of *information* and *know-how*. If you have genuine *bona fide* experience of mathematical work on any level, elementary or advanced, there will be no doubt in your mind that, in mathematics, know-how is much more important than mere possession of information. ...
>
> What is know-how in mathematics? The ability to solve problems—not merely routine problems but problems requiring some degree of independence, judgment, originality, creativity. Therefore, the first and foremost duty ... in teaching mathematics is to emphasize *methodical work in problem solving.*

What Polya says for mathematics holds just as much for computing science.

In his book *How to Solve It*, Polya states four phases of problem solving. These steps are important for programming as well.

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan, checking each step.
4. Reexamine and reconsider the solution. (And, of course, reexamine the understanding of the problem, the plan, and the way the plan was carried out.)

7.4.2 Problem-solving strategies

There are many problem-solving strategies applicable to programming in general and functional programming in particular. We have seen some of these in the earlier chapters and will see others in later chapters. In this section, we highlight some of the general techniques.

7.4.2.1 Solve a more general problem first

That is, we solve a "harder" problem than the specific problem at hand, then use the solution of the "harder" problem to get the specific solution desired.

Sometimes a solution of the more general problem is actually easier to find because the problem is simpler to state or more symmetrical or less obscured by special conditions. The general solution can often be used to solve other related problems.

Often the solution of the more general problem can actually lead to a more efficient solution of the specific problem.

We have already seen one example of this technique: finding the first occurrence of an item in a list.

First, we devised a program to find all occurrences in a list. Then we selected the first occurrence from the set of all occurrences. (Lazy evaluation of Haskell programs means that this use of a more general solution differs very little in efficiency from a specialized version.)

We have also seen several cases where we have generalized problems by adding one or more *accumulating parameters*. These "harder" problems can lead to more efficient tail recursive solutions.

For example, consider the tail recursive Fibonacci program we developed in a previous chapter. We added two extra arguments to the function.

```
fib2 :: Int -> Int
fib2 n | n >= 0 = fibIter n 0 1
    where
        fibIter 0 p q         = p
        fibIter m p q | m > 0 = fibIter (m-1) q (p+q)
```

Another approach is to use the *tupling* technique. Instead of adding extra arguments, we add extra results.

For example, in the Fibonacci program `fastfib` below, we compute `(fib n, fib (n+1))` instead of just `fib n`. This is a harder problem, but it actually gives us more information to work with and, hence, provides more opportunity for optimization. (We formally derive this solution in a later chapter.)

```
fastfib :: Int -> Int
fastfib n | n >= 0 = fst (twofib n)

twofib :: Int -> (Int,Int)
twofib 0 = (0,1)
twofib n = (b,a+b)
    where (a,b) = twofib (n-1)
```

7.4.2.2 Solve a simpler problem first

After solving the simpler problem, we then adapt or extend the solution to solve the original problem.

Often the mass of details in a problem description makes seeing a solution difficult. In the previous technique we made the problem easier by finding a more general problem to solve. In this technique, we move in the other direction: we find a more specific problem that is similar and solve it.

At worst, by solving the simpler problem we should get a better understanding of the problem we really want to solve. The more familiar we are with a problem, the more information we have about it, and, hence, the more likely we will be able to solve it.

At best, by solving the simpler problem we will find a solution that can be easily extended to build a solution to the original problem.

Consider a program to convert a positive integer of up to six digits to a string consisting of the English words for that number. For example, `369027` yields the string:

```
three hundred and sixty-nine thousand and twenty-seven
```

To deal with the complexity of this problem, we can work as follows:

a. Solve the problem of converting a two-digit number to words. (The single digit numbers and numbers in teens are special cases.)
b. Then extend the two-digit solution to three digits.
c. Then extend three-digit solution to to six digits.

See Section 4.1 of the classic Bird/Wadler textbook for the details of this problem and a solution.

The process of generalizing first-order functions into higher-order functions is another example of this "solve a simpler problem first" strategy. Recall how we motivated the development of the higher-order library functions such as `map`, `filter`, and `foldr`. Also consider the function generalization approach used in the cosequential processing case study.

7.4.2.3 Reuse off-the-shelf solutions to standard subproblems

We have been doing this all during this semester, especially since we began began studying polymorphism and higher-order functions.

The basic idea is to identify standard patterns of computation (e.g., standard prelude functions such as `length`, `take`, `zip`, `map`, `filter`, `foldr`) that will solve some aspects of the problem and then combine (e.g., using function composition) these standard patterns with your own specialized functions to construct a solution to the problem.

We have seen several examples of this in these notes and the homework assignments.

See section 4.2 of the classic Bird/Wadler textbook for a case study that develops a package of functions to do arithmetic on variable length integers. The functions take advantage of several of the standard prelude functions.

7.4.2.4 Solve a related problem

After solving the related problem, we then transform the solution of the related problem to get a solution to the original problem.

Perhaps we can find an entirely different problem formulation (i.e., stated in different terms) for which we can readily find a solution. Then that solution can be converted into a solution to the problem at hand.

For example, we can recast a problem in terms of mathematical or logical frameworks (e.g., sets, relations, graphs, finite state machines, grammars, or algebraic structures), solve the corresponding problem in those terms, and then interpret the result for the original problem. The simplification provided by the frameworks may reveal solutions that are obscured in the details of the problem. We can also take advantage of the theory and techniques that have been found previously for the mathematical frameworks.

Consider the problem of breaking a string of text into the list of its component lines.

This is not trivial. However, the "inverse" problem is trivial. All that is needed to convert a list of lines to a string of text is to insert linefeed characters between the lines.

We can first solve the inverse problem (line-folding) and then use it to calculate what the line-breaking program must be. (See Section 4.3 of the Bird/Wadler textbook and a later chapter in this course.)

7.4.2.5 Separate concerns

That is, we partition the problem into logically separate problems, solve each problem separately, then combine the solutions to the subproblems to construct a solution to the problem at hand.

As we have seen in the above strategies, when a problem is complex and difficult to attack directly, we search for simpler, but related, problems to solve, then build a solution to the complex problem from the simpler problems.

We have seen examples of this approach in earlier chapters and homework assignments. We separated concerns when we used stepwise refinement to develop a square root function, data abstraction in the rational number case study, and function pipelines.

Consider the development of a program to print a calendar for any year in various formats. We can approach this problem by first separating it into two independent subproblems:

a. building a calendar
b. formatting the output

After solving each of these simpler problems, the more complex problem can be solved easily by combining the two solutions. (See Section 4.5 of the classic Bird/Wadler textbook for the details of this problem and a solution.)

7.4.2.6 Divide and conquer

This is a special case of the "solve a simpler problem first" strategy. In this technique, we must divide the problem into subproblems that are the same as the original problem except that the size of the input is smaller.

This process of division continues recursively until we get a problem that can be solved trivially, then we combined we reverse the process by combining the solutions to subproblems to form solutions to larger problems.

Examples of divide and conquer from earlier chapters include the logarithmic exponentiation function `expt3` and the merge sort function `msort`.

Another common example of the divide and conquer approach is binary search. (See Section 6.4 of the classic Bird/Wadler textbook.)

There are, of course, other strategies that can be used to approach problem solving.

7.5 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..]))`

7.6 References

[**Bird-Wadler 1998**] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Polya 1945**] George Polya, *How to Solve It*, Princeton University Press, 1945.

[**Polya 1981**] George Polya, *Mathematical Discovery: On Understanding, Learning and Teaching Problem Solving*, Wiley, 1981.

[**Thompson 2011**] Simon Thompon. *Haskell: The Craft of Programming, Third Edition*, Pearson, 2011.

7.7 Terms and Concepts

TODO