Exploring Languages with Interpreters and Functional Programming Chapter 30

H. Conrad Cunningham

04 April2022

Contents

30 Infinite Data Structures 2
30.1 Chapter Introduction 2
30.2 Infinite Lists 2
30.3 Iterate 3
30.4 Prime Numbers: Sieve of Eratosthenes 4
30.5 Circular Structures 6
30.6 What Next? 7
30.7 Chapter Source Code 7
30.8 Exercises 7
30.9 Acknowledgements 7
30.10Terms and Concepts 7
30.11References 7

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.

30 Infinite Data Structures

30.1 Chapter Introduction

One particular benefit of *lazy evaluation* is that functions in Haskell can manipulate "infinite" data structures. Of course, a program cannot actually generate or store all of an infinite object, but lazy evaluation will allow the object to be built piece-by-piece as needed and the storage occupied by no-longer-needed pieces to be reclaimed.

This chapter explores Haskell programming techniques for infinite data structures such as lists.

TODO: Write Introduction, including goals of chapter.

TODO: - Complete chapter. Improve the writing. - Update and expand discussion of infinite computations. - Recreate the missing Haskell source code files for this chapter. Ensure it works for Haskell 2010.

30.2 Infinite Lists

Reference: This section is based, in part, on discussions in the classic Bird and Wadler textbook [1:7.1] and Wentworth's tutorial [3].

In Chapter 18 , we looked at generators for infinite arithmetic sequences such as `[1..]` and `[1,3..]`. These infinite lists are encoded in the functions that generate the sequences. The sequences are only evaluated as far as needed.

For example, `take 5 [1..]` yields:

```
[1,2,3,4,5]
```

Haskell also allows infinite lists of infinite lists to be expressed as shown in the following example which generates a table of the multiples of the positive integers.

```
multiples :: [[Int]]
multiples = [ [ m*n | m<-[1..]] | n <- [1..] ]
```

Thus `multiples` represents an infinite list, as shown below (not valid Haskell code):

```
[ [1, 2, 3, 4, 5, ... ],
  [2, 4, 6, 8,10, ... ],
  [3, 6, 9,12,14, ... ],
  [4, 8,12,16,20, ... ],
  ...
]
```

However, if we evaluate the expression

```
take 4 (multiples !! 3)
```

we get the terminating result:

```
[4,8,12,16]
```

Note: Remember that the operator `xs !! n` returns element `n` of the list `xs` (where the head is element `0`).

Haskell's infinite lists are not the same as *infinite sets* or *infinite sequences* in mathematics. *Infinite lists* in Haskell correspond to *infinite computations* whereas infinite sets in mathematics are simply definitions.

In mathematics, set $\{x^2 \mid x \in \{1,2,3\} \wedge x^2 < 10\} = \{1,4,9\}$.

However, in Haskell, the expression

```
show [ x * x | x <- [1..], x * x < 10 ]
```

yields:

```
[1,4,9
```

This is a computation that never returns a result. Often, we assign this computation the value `1:4:9:`$\perp$ (where $\perp$, pronounced "bottom" represents an undefined expression).

But the expression

```
takeWhile (<10) [ x * x | x <- [1..] ]
```

yields:

```
[1,4,9]
```

30.3 Iterate

Reference: This section is based in part on a discussion in the classic Bird and Wadler textbook [1:7.2].

In mathematics, the notation f^n denotes the function f composed with itself n times. Thus, $f^0 = id$, $f^1 = f$, $f^2 = f.f$, $f^3 = f.f.f$, $\cdots$.

A useful function is the function `iterate` such that (not valid Haskell code):

```
iterate f x = [x, f x, f^2 x, f^3 x, ... x ]
```

The Haskell standard Prelude defines `iterate` recursively as follows:

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

For example, suppose we need the set of all powers of the integers.

We can define a function `powertables` would expand as follows (not valid Haskell code):

```
[ [1, 2, 4,  8, ...
  [1, 3, 9, 27, ...
  [1, 4,16, 64, ...
  [1, 5,25,125, ...
  ...
]
```

Using `iterate` we can define `powertables` compactly as follows:

```
powertables :: [[Int]]
powertables = [ iterate (*n) 1 | n <- [2..]]
```

As another example, suppose we want a function to extract the decimal digits of a positive integer. We can define `digits` as follows:

```
digits :: Int -> [Int]
digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (/10)
```

Let's consider how `digits 178` evaluates (not actual reduction steps).

```
    digits 178
⟹
    reverse . map (mod10) . takeWhile (/= 0) [178,17,1,0,0,
    ...]
⟹
    reverse . map (mod10) [178,17,1]
⟹
    reverse [8,7,1]
⟹
    [1,7,8]
```

30.4 Prime Numbers: Sieve of Eratosthenes

Reference: This is based in part on discussions in the classic Bird and Wadler textbook [1:7.3] and Wentworth's tutorial [3, Ch. 9].

The Greek mathematician Eratosthenes described essentially the following procedure for generating the list of all *prime numbers*. This algorithm is called the *Sieve of Eratosthenes*.

1. Generate the list 2, 3, 4, $\cdots$
2. Mark the first element p as prime.
3. Delete all multiples of p from the list.
4. Return to step 2.

Not only is the 2-3-4 loop infinite, but so are steps 1 and 3 themselves.

There is a straightforward translation of this algorithm to Haskell.

```
primes :: [Int]
primes = map head (iterate sieve [2..])

sieve (p:xs) = [x | x <- xs, x `mod` p /= 0 ]
```

Note: This uses an intermediate infinite list of infinite lists; even though it is evaluated lazily, it is still inefficient.

We can use function `primes` in various ways, e.g., to find the first 1000 primes or to find all the primes that are less than 10,000.

```
take 1000 primes
takeWhile (<10000) primes
```

Calculations such as these are not trivial if the computation is attempted using arrays in an "eager" language like Pascal—in particular it is difficult to know beforehand how large an array to declare for the lists.

However, by *separating the concerns*, that is, by keeping the computation of the primes separate from the application of the boundary conditions, the program becomes quite modular. The same basic computation can support different boundary conditions in different contexts.

Now let's transform the `primes` and `sieve` definitions to eliminate the infinite list of infinite lists. First, let's separate the generation of the infinite list of positive integers from the application of `sieve`.

```
primes        = rsieve [2..]

rsieve (p:ps) = map head (iterate sieve (p:ps))
```

Next, let's try to transform `rsieve` into a more efficient definition.

```
    rsieve (p:ps)
= { rsieve }
    map head (iterate sieve (p:ps))
= { iterate }
    map head ((p:ps) : (iterate sieve (sieve (p:ps)) ))
= { map.2, head }
    p : map head (iterate sieve (sieve (p:ps)) )
= { sieve }
    p : map head (iterate sieve [x | x <- ps, x `mod` p /= 0 ])
= { rsieve }
```

```
    p : rsieve [x | x <- ps, x `mod` p /= 0 ]
```

This calculation gives us the new definition:

```
    rsieve (p:ps) = p : rsieve [x | x <- ps, x `mod` p /= 0 ]
```

This new definition is, of course, equivalent to the original one, but it is slightly more efficient in that it does not use an infinite list of infinite lists.

30.5 Circular Structures

Reference: This section is based, in part, on discussions in classic Bird and Wadler textbook [1:7.6] and of Wentworth's tutorial [3, Ch. 9].

Suppose a program produces a data structure (e.g., a list) as its output. And further suppose the program feeds that output structure back into the input so that later elements in the structure depend on earlier elements. These might be called *circular*, *cyclic*, or *self-referential* structures.

Consider a list consisting of the integer one repeated infinitely:

```
    ones = 1:ones
```

As an expression graph, `ones` consists of a cons operator with two children, the integer `1` on the left and a recursive reference to `ones` (i.e., a self loop) on the right. Thus the infinite list `ones` is represented in a finite amount of space.

Function `numsFrom` below is a perhaps more useful function. It generates a list of successive integers beginning with `n`:

```
    numsFrom :: Int -> [Int]
    numsFrom n = n : numsFrom (n+1)
```

Using `numsFrom` we can construct an infinite list of the natural number multiples of an integer `m`:

```
    multiples :: Int -> [Int]
    multiples m  = map ((*) m) (numsFrom 0)
```

Of course, we cannot actually process all the members of one of these infinite lists. If we want a terminating program, we can only process some finite initial segment of the list. For example, we might want all of the multiples of 3 that are at most 2000:

```
    takeWhile ((>=) 2000) (multiples 3)
```

We can also define a program to generate a list of the Fibonacci numbers in a circular fashion similar to `ones`:

```
    fibs :: [Int]
    fibs = 0 : 1 : (zipWith (+) fibs (tail fibs))
```

Proofs involving infinite lists are beyond the current scope of this textbook. See the Bird and Wadler textbook for more information [1].

TODO: Finish Chapter

30.6 What Next?

TODO

30.7 Chapter Source Code

TODO

30.8 Exercises

TODO

30.9 Acknowledgements

In Summer 2018, I adapted and revised this chapter from chapter 15 of my *Notes on Functional Programming with Haskell* [2].

These previous notes drew on the presentations in the 1st edition of the Bird and Wadler textbook [1], Wentworth's tutorial [3], and other functional programming sources.

I incorporated this work as new Chapter 30, Infinite Data Structures, 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.

30.10 Terms and Concepts

Infinite data structures, lazy evaluation, infinite sets, infinite sequences, infinite lists, infinite computations, bottom $\perp$, `iterate`, prime numbers, Sieve of Eratosthenes, separation of concerns, circular/cyclic/self-referential structures.

30.11 References

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

[2] 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

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