Exploring Languages with Interpreters and Functional Programming Chapter 8

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

8 Evaluation Model

8.1 Chapter Introduction

This chapter introduces an evaluation model applicable to Haskell programs. As in the previous chapters, this chapter focuses on use of first-order functions and primitive data types.

A goal of this chapter and the next one is enable students to analyze Haskell functions to determine under what conditions they terminate normally and how efficient they are. This chapter presents the evaluation model and the next chapter informally analyzes simple functions in terms of time and space efficiency and termination.

How can we evaluate (i.e. execute) an expression that "calls" a function like the `fact1` function from Chapter 4?

We do this by rewriting expressions using a substitution model, as we see in this chapter. This process depends upon a property of functional languages called referential transparency.

8.2 Referential Transparency Revisited

Referential transparency is probably the most important property of modern functional programming languages.

As defined in Chapter 2, referential transparency means that, within some well-defined context (e.g. a function or module definition), a variable (or other symbol) *always* represents the *same value*.

Because a variable always has the same value, we can replace the variable in an expression by its value or vice versa. Similarly, if two subexpressions have equal values, we can replace one subexpression by the other. That is, "equals can be replaced by equals".

Pure functional programming languages thus use the same concept of a variable that mathematics uses.

However, in most imperative programming languages, a variable represents an address or "container" in which values may be stored. A program may change the value stored in a variable by executing an assignment statement. Thus these mutable variables break the property of referential transparency.

Because of referential transparency, we can construct, reason about, and manipulate functional programs in much the same way we can any other mathematical expressions. Many of the familiar "laws" from high school algebra still hold; new laws can be defined and proved for less familiar primitives and even user-defined operators. This enables a relatively natural equational style of reasoning

using the actual expressions of the language. We explore these ideas further in Chapters 25 and 26.

In contrast, to reason about imperative programs, we usually need to go outside the language itself and use notation that represents the semantics of the language.

For our purposes here, referential transparency underlies the substitution model for evaluation of expressions in Haskell< programs.

8.3 Substitution Model

The *substitution model* (or *reduction model*) involves rewriting (or reducing) an expression to a "simpler" equivalent form. It involves two kinds of replacements:

- replacing a subexpression that satisfies the left-hand side of an equation by the right-hand side with appropriate substitution of arguments for parameters
- replacing a primitive application (e.g. `+` or `*`) by its value

The term *redex* refers to a subexpression that can be reduced.

Redexes can be selected for reduction in several ways. For instance, the redex can be selected based on its position within the expression:

- *leftmost redex first*, where the leftmost reducible subexpression in the expression text is reduced before any other subexpressions are reduced
- *rightmost redex first*, where the rightmost reducible subexpression in the expression text is reduced before any other subexpressions are reduced

The redex can also be selected based on whether or not it is contained within another redex:

- *outermost redex first*, where a reducible subexpression that is not contained within any other reducible subexpression is reduced before one that is contained within another
- *innermost redex first*, where a reducible subexpression that contains no other reducible subexpression is reduced before one that contains others

We will explore these more fully in later chapters. In most circumstances, Haskell uses a *leftmost outermost redex first* approach.

In Chapter 4, we defined factorial function `fact1` as shown below. (The source code is in source file `Factorial.hs`).

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
          else
              n * fact1 (n-1)
```

Consider the expression from `else` clause in `fact1` with `n` having the value 2:

```
2 * fact1 (2-1)
```

This has two redexes: subexpressions `2-1` and `fact1 (2-1)`.

The multiplication cannot be reduced because it requires both of its arguments to be evaluated.

A function parameter is said to be *strict* if the value of that argument is always required. Thus, multiplication is strict in both its arguments. If the value of an argument is not always required, then it is *nonstrict*.

The first redex `2-1` is an innermost redex. Since it is the only innermost redex, it is both leftmost and rightmost.

The second redex `fact1 (2-1)` is an outermost redex. Since it is the only outermost redex, it is both leftmost and rightmost.

Now consider the complete evaluation of the expression `fact1 2` using leftmost outermost reduction steps. Below we denote the steps with $\Longrightarrow$ and give the substitution performed between braces.

```
     fact1 2
⟹   { replace fact1 2 using definition }
     if 2 == 0 then 1 else 2 * fact1 (2-1)
⟹   { evaluate 2 == 0 in condition }
     if False then 1 else 2 * fact1 (2-1)
⟹   { evaluate if }
     2 * fact1 (2-1)
⟹   { replace fact1 (2-1) using definition, add implicit parentheses }
     2 * (if (2-1) == 0 then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate 2-1 in condition }
     2 * (if 1 == 0 then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate 1 == 0 in condition }
     2 * (if False then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate if }
     2 * ((2-1) * fact1 ((2-1)-1))
⟹   { evaluate leftmost 2-1 }
     2 * (1 * fact1 ((2-1)-1))
⟹   { replace fact1 ((2-1)-1) using definition, add implicit parentheses }
     2 * (1 * (if ((2-1)-1) == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate 2-1 in condition }
     2 * (1 * (if (1-1) == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate 1-1 in condition }
     2 * (1 * (if 0 == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
```

```
⟹   { evaluate 0 == 0 }
     2 * (1 * (if True then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate if }
     2 * (1 * 1)
⟹   { evaluate 1 * 1 }
     2 * 1
⟹   { evaluate 2 * 1 }
     2
```

The rewriting model we have been using so far can be called *string reduction* because our model involves the textual replacement of one string by an equivalent string.

A more efficient alternative is *graph reduction*. In this technique, the expressions are represented as (directed acyclic) expression graphs rather than text strings. The repeated subexpressions of an expression are represented as shared components of the expression graph. Once a shared component has been evaluated once, it need not be evaluated again.

In the example above, subexpression `2-1` is reduced three times. However, all of those subexpressions come from the initial replacement of `fact1 2`. Using graph reduction, only the first of those reductions is necessary.

```
     fact1 2
⟹   { replace fact1 2 using definition }
     if 2 == 0 then 1 else 2 * fact1 (2-1)
⟹   { evaluate 2 == 0 in condition }
     if False then 1 else 2 * fact1 (2-1) }
⟹   { evaluate if }
     2 * fact1 (2-1)
⟹   { replace fact1 (2-1) using definition, add implicit parentheses }
     2 * (if (2-1) == 0 then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate 2-1 because of condition (3 occurrences in graph) }
     2 * (if 1 == 0 then 1 else 1 * fact1 (1-1))
⟹   { evaluate 1 == 0 }
     2 * (if False then 1 else 1 * fact1 (1-1))
⟹   { evaluate if }
     2 * (1 * fact1 (1-1))
⟹   { replace fact1 ((1-1) using definition, add implicit parentheses }
     2 * (1 * (if (1-1) == 0 then 1 else (1-1) * fact1 ((1-1)-1))
⟹   { evaluate 1-1 because of condition (3 occurrences in graph) }
     2 * (1 * (if 0 == 0 then 1 else 0 * fact1 (0-1))
⟹   { evaluate 0 == 0 }
     2 * (1 * (if True then 1 else  0 * fact1 (0-1))
```

```
⟹   { evaluate if }
     2 * (1 * 1)
⟹   { evaluate 1 * 1 }
     2 * 1
⟹   { evaluate 2 * 1 }
     2
```

In general, the Haskell compiler or interpreter uses a leftmost outermost graph reduction technique. However, if the value of a function's argument is always needed for a computation, then an innermost reduction can be triggered for that argument. Either the programmer can explicitly require this or the compiler can detect the situation and automatically trigger the innermost reduction order.

Haskell exhibits *lazy evaluation*. That is, an expression is not evaluated until its value is needed, if ever. An outermost reduction corresponds to this evaluation strategy.

Other functional languages such as Scala and F# exhibit *eager evaluation*. That is, an expression is evaluated as soon as possible. An innermost reduction corresponds to this evaluation strategy.

8.4 Time and Space Complexity

We state efficiency (i.e. time complexity or space complexity) of programs in terms of the "Big-O" notation and asymptotic analysis.

For example, consider the leftmost outermost graph reduction of function `fact1` above. The number of reduction steps required to evaluate `fact1 n` is `5n + 3`.

We let the number of steps in a graph reduction be our measure of time. Thus, the *time complexity* of `fact1 n` is O(`n`), which means that the time to evaluat `fact1 n` is bounded above by some (mathematical) function that is proportional to the value of `n`.

Of course, this result is easy to see in this case. The algorithm is dominated by the `n` multiplications it must carry out. Alternatively, we see that evaluation requires on the order of `n` recursive calls.

We let the number of *arguments* in an expression graph be our measure of the *size* of an expression. Then the *space complexity* is the maximum size needed for the evaluation in terms of the input.

This size measure is an indication of the maximum size of the unevaluated expression that is held at a particular point in the evaluation process. This is a bit different from the way we normally think of space complexity in imperative algorithms, that is, the number of "words" required to store the program's data.

However, this is not as strange as it may at first appear. As we in later chapters, the data structures in functional languages like Haskell are themselves *expressions* built by applying constructors to simpler data.

In the case of the graph reduction of `fact1 n`, the size of the largest expression is `2n + 16`. This is a multiplication for each integer in the range from 1 to `n` plus 16 for the full `if` statement. Thus the space complexity is O(`n`).

The Big-O analysis is an asymptotic analysis. That is, it estimates the order of magnitude of the evaluation time or space as the size of the input approaches infinity (gets large). We often do worst case analyses of time and space. Such analyses are usually easier to do than average-case analyses.

The time complexity of `fact1 n` is similar to that of a loop in an imperative program. However, the space complexity of the imperative loop algorithm is O(1). So `fact1` is not space efficient compared to the imperative loop.

We examine techniques for improving the efficiency of functions below. In later chapters, we examine reduction techniques more fully.

8.5 Termination

A recursive function has one or more recursive cases and one or more base (nonrecursive) cases. It may also be undefined for some cases.

To show that evaluation of a recursive function terminates, we must show that each recursive application *always* gets closer to a termination condition represented by a base case.

Again consider `fact1` defined above.

If `fact1` is called with argument `n` greater than 0, the argument of the recursive application in the `else` clause always decreases to `n - 1`. Because the argument always decreases in integer steps, it must eventually reach 0 and, hence, terminate in the first leg of the definition.

If we call `fact1` with argument 0, the function terminates immediately.

What if we call `fact1` with its argument less than 0? We consider this issue below.

8.6 What Next?

This chapter introduced an evaluation model applicable to Haskell programs. It provides a framework for analyzing Haskell functions to determine under what conditions they terminate normally and how efficient they are.

The next chapter informally analyzes simple functions in terms of time and space efficiency and termination.

8.7 Exercises

1. Given the following definition of Fibonacci function `fib`, show the reduction of `fib 4`.

```
fib :: Int -> Int
fib 0          = 0
fib 1          = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
```

2. What are the time and space complexities of `fact6` as defined in the previous exercise?

3. Given the following definition of `fact6`, show the reduction of `fact6 2`.

```
fact6 :: Int -> Int
fact6 n = factIter n 1

factIter :: Int -> Int -> Int
factIter 0 r         = r
factIter n r | n > 0 = factIter (n-1) (n*r)
```

4. What are the time and space complexities of `fact6` as defined in the previous exercise?

8.8 Acknowledgements

In Summer and Fall 2016, I adapted and revised much of this work from parts of chapters 1 and 13 of my *Notes on Functional Programming with Haskell* [Cunningham 2014].

In 2017, I continued to develop this work as Chapter 3, Evaluation and Efficiency, of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Evaluation and Efficiency chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 3.1-3.2 became the basis for new Chapter 8, Evaluation Model (this chapter), and previous sections 3.3-3.5 became the basis for Chapter 9, Recursion Styles and Efficiency. I also moved the discussion of preconditions and postconditions to the new Chapter 6.

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

8.9 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 1996]: Simon Thompson. *Haskell: The Craft of Programming*, First Edition, Addison Wesley, 1996; Second Edition, 1999; Third Edition, Pearson, 2011.

8.10 Terms and Concepts

Referential transparency, reducible expression (redex), reduction strategies (leftmost vs. rightmost, innermost vs. outermost), string and graph reduction models, time and space complexity, termination preconditions, postconditions, contracts.