Lazy Evaluation in Haskell

Michael Kulinich
Chapman University

October 17, 2021

Abstract

Many modern-day programmers generally prefer programming in imperative languages such as Python, C, or C++. After learning how to use one language effectively, one can swiftly start working with other languages because the thought process and method of answering problems are very similar. Little do they know that there actually exists another paradigm known as functional programming in which the goal is to bind everything in clean mathematical functions similar to what one would see in a mathematics course. In this way, programs are built solely from function application and evaluation. Then to break it down even more, in languages such as Haskell, everything is an expression, and the goal is to evaluate them in order to solve a problem. In this paper, we will observe and analyze the strategies for evaluating these expressions. ...

Contents

1 Introduction

1.1 General Remarks

For this paper, I will assume that the reader has a basic familiarity with Haskell, but if you are new to Haskell, that is not a problem. I will try to explain everything as clear as possible and work with simple examples. The first step before I get into anything is to make sure that you at least have Haskell installed. Haskell can be downloaded using the instructions from here. Assuming everything installs smoothly, we can

get started, but for any problems be sure to check out this website. In this paper, we will also be using GHCI to create interactive environments to run Haskell functions. To familiarize yourself with GHCI, I recommend looking at this website

1.2 Key Points

Imperative languages are usually composed of statements and expressions. An expression such as "5 * 2" is evaluated and modifies memory via an assignment operator to create a value. Expressions are entities that can be reduced into a value. On the other hand, statements are units of execution such as conditionals and loops that do not return values but compute by using side effects. A sequence of statements determines how to answer a certain problem. Functional languages, on the other hand, have programs that are built solely from function application and evaluation. In languages such as Haskell, everything is an expression, and the goal is to evaluate them in order to solve a problem. Haskell defaults to a method of evaluation known as lazy evaluation. Put simply, Haskell is lazy. It won't execute functions and calculate things until it's really forced to show you a result [LYAH]. Haskell uses lazy evaluation to evaluate expressions because it comes with several advantages.

- Function arguments are not computed before invocation
- Expressions are computed only if their value is necessary and it avoids repeated evaluation of expressions, thus making algorithms more efficient and reducing computation complexity
- Allows the ability to program with infinite lists
- Promises termination when possible

The goal of this paper is to understand what lazy evaluation really is, what makes Haskell lazy, and the advantages of this evaluation strategy. First, we will need to do a quick dive into Lambda Calculus because it is the building block of purely functional programming. Then, we will use Lambda calculus as the key to understanding the difference between two evaluation strategies for reducible expressions (redex). Finally, building upon these foundations, we will work on several examples of lazy evaluation.

2 Lambda Calculus

Before getting into how lazy evaluation is used in Haskell and its advantages, we first need to understand the idea behind lambda calculus. The motivation for looking at lambda calculus is because it elegantly demonstrates how expression evaluation can simply be seen as a sequence of reduction steps. This is also popularly known as rewriting, where we reduce/rewrite an expression into a simpler/shorter form. This process is repeated by any number of steps until we reach the normal form. An expression is in normal form when it cannot be reduced anymore. More concretely, a lambda expression is in normal form if it contains no more possible redexes. We will see how lambda expressions open up to two potential methods of expression evaluation: Normal-order evaluation and applicative-order evaluation.

2.1 What is Lambda Calculus?

Lambda calculus, also denoted as λ-calculus, is a convenient notation for functions and applications. What is a function? In mathematics, a function is something that relates an input to output. Say we want to write a function that squares an input, we can do this by $f(x) = x^2$. f is the function name, x is just an placeholder for the input, then x^2 is what we need to output. Now lets look at some examples of function in programming languages.In python, a similar function is written as

Similarly, we can create a function that does the same thing in Haskell

```
def square(x):
    return x*x
```

Figure 1: *A python function that squares an input*

```
square :: Int -> Int
square x = x*x
```

Figure 2: *A Haskell function that squares an input*

Remark: Before getting into it, lets go over this Haskell code first. The purpose of the first line

$$\text{square} :: \text{Int} \rightarrow \text{Int}$$

is to give our function *square* an explicit type declaration. The :: symbol is read as "has type of". Then the first parameter is the argument type that gets passed into our function, which in this case is an *Int*. The next parameter is the "return type" of the function, which in this case is also an *Int*. Without going into more advanced detail, there is no special distinction between the parameters and the return type [LYAH]. Therefore , the function type declaration can be read as: *function square takes in an argument of type Int for the and outputs an Int.* Declaring the types for your functions is not necessary, but it makes your code easier to read and understand. Also, I chose the argument to be of type *Int* just for demonstrations purposes. I could have chosen another type such as *float*. For more references about various types in Haskell, check out Tutorials Point.

The second line

$$\text{square x} = \text{x*x}$$

is the function definition. We define a function *square* that takes an argument n and returns the square of that number, $n*n$.

In both languages, the function is given a name, *square*. Both functions take in an input which we call x, but this variable name is just a placeholder and we can realistically call it whatever we want and the function will not change. Now to apply a value to compute with the function, in Haskell we can say: *square 5 reduces to 5*. We will use this notation:

$$\text{square } 5 \rightarrow 25$$

Here , we are merely substituting 5 for variable x in function square. You might be wondering, why I am going over such simple high school algebra? Well to understand lambda calculus, we had to break down the basic components of a function. The main difference between what we've just seen and λ-calculus is that we can write functions without giving them explicit names, but the idea of applying a function to an argument and forming functions by abstraction still holds. Woah, that was a lot, let's see how this same function can be expressed in terms of λ-calculus by breaking down a quick example.

$$\begin{aligned}(\lambda x.x * x)5 &= 5 * 5 \quad [reduction] \\ &= 25\end{aligned}$$

In this examples, abstraction serves as the square function, represented by $\lambda x.x * x$ and the application is represented by applying the function $\lambda x.x * x$ to the argument 5. In essence, we are substituting every

occurrence of x with 5 in the lambda term. What is this computation of reduction more specifically? Let's define this as the β-reduction rule.

2.2 β-Reduction Rule

Let's look at the general definition for reducing lambda expressions.

- **β-reduction rule** If λx.M is a lambda term and N is another lambda term, then $(\lambda x.M)N \rightarrow M[N/x]$ that for the term M, each occurrence of x is substituted by N.

This idea of using substitution to reduce an expression, is very fundamental in lambda calculus and the theoretical building blocks of functional programming. As stated earlier, we use the β-reduction rule until we read normal form where we cannot reduce the redex any further.

3 Reduction Strategies

Reduction strategies can also be thought of as evaluation strategies. This implies that at each step during expression evaluation unless we are in normal form, there may be an expression we can reduce ti by applying a reduction definition. Let's look at a lambda expression that will apply a lambda term to another lambda term. $(\lambda x.x)((\lambda y.y)a)$. This expression can be reduced with two possible strategies.

$$
\begin{array}{c}
(\lambda x.x)((\lambda y.y)a) \\
\swarrow \quad \searrow \\
(\lambda y.y)a \quad \text{or} \quad (\lambda x.x)a \\
\searrow \quad \swarrow \\
a
\end{array}
$$

Figure 3: *Possible Reduction Strategies*

On the right-hand branch, we are evaluating the second lambda term

$$(\lambda y.y)a \rightarrow a$$

Then we apply this result, a into

$$\lambda x.x$$

in order to evaluate to the final form a. In programming, this method of evaluation is known as *call by value*, and more generally as "innermost reduction" or "applicative-order reduction".

On the left-hand branch, we are evaluating the first lambda term by substituting the whole right lambda term. We are basically replacing the variable x in

$$\lambda x.x$$

with the lambda term

$$(\lambda y.y)a$$

In programming, this method of evaluation is known as *call by name*, and more generally it is known as "outermost reduction" or "normal-order reduction." This strategy of evaluation is what we are more interested in because this is the essential idea behind lazy evaluation in Haskell.

3.1 Outermost Reduction Rule

I like to think of this rule as the "outermost reduction rule" instead of the "normal-order reduction rule" because innermost reduction is more conceptual. As previously mentioned, a redex is a reducible expression. So if we have an expression composed of two or more redexes, we reduce the leftmost first before reducing the sub-expressions inside of it. This can be seen in the example we had above where we have two possible redexes:

$$(\lambda x.x)((\lambda y.y)a)$$

For outermost reduction, we reduced the leftmost redex $(\lambda x.x)$ by applying/substituting the innermost redex $(\lambda y.y)a$ into the outermost redex. Furthermore, we are delaying the evaluation of the innermost redex (argument for the outermost redex) until we actually need to compute the expression. Functional languages such as Haskell follow this evaluation strategy. This makes Haskell lazy because it does not compute the argument until it finally has to.

3.2 Innermost Reduction Rule

Innermost reduction (applicative-order reduction) is basically the opposite of outermost reduction. The innermost redex is reduced first, so essentially function arguments are evaluated first before they are substituted into the function. Functions in languages most imperative languages such as python and C/C++ follow this strategy.

3.3 Comparing Both Reduction Rules

Lets look at an example in Haskell demonstrating these two different possibilities of evaluation. We are going to compute $(2+3)^2$ using a Haskell function name *square*

```
-- square an int
square :: Int -> Int
square n = n * n
```

Outermost	**Innermost**
$square(2+3)$	$square(2+3)$
$=(2+3)*(2+3)$	$=square(5)$
$=5*(2+3)$	$=5*5$
$=5*5$	$=25$
$=25$	

Now lets look at the first column with outermost evaluation. In this method, we delay the evaluation of the argument. Instead, we reduce the outermost redex, which is the *square* abstraction. Following the definition of *square* above, we substitute $(2+3)$ into the variable n, then perform the squaring step of $n*n$

New lets observe the second column with innermost evaluation. In this method, we do not delay the evaluation of the argument. The first step in fact is to compute the expression (2+3). Basically, the argument is reduced before it is substituted into the function. This sort of reduction is what most programmers are used to. It is intuitive and generally preferred.

How can we compare the two methods? The first thing we can notice is that both strategies end up with the same result. This is an important note to make because in functional programming, if there are no side effects, we are guaranteed to end up with the same results from both evaluation methods. Also we can see that innermost evaluation reduces the entire expression in the three steps, compared to the four steps taken

by outermost reduction. In outermost reduction, we can see that we end up having to reduce the same expression $(2 + 3)$ on two separate occasions. That being said, it seems like innermost evaluation is more efficient and doesn't duplicate expressions.

Up until now, I have been building up a connection between outermost reduction and lazy evaluation. The two really sound the same right? Earlier I said that languages such as Haskell that default to lazy evaluation, do not compute function arguments before invocation. This sounds almost exactly the same as the strategy of outermost or normal-order expression evaluation. So this raises the question, if outermost evaluation is more inefficient because it needs to reduce the same, duplicate expression multiple times, then why should we care about it or lazy evaluation? Have I been wasting your time talking about lazy evaluation? The short answer is no... We have been building up how outermost reduction is essentially lazy evaluation, however, we are missing an extra ingredient. This ingredient is known as argument sharing or memoization. Argument sharing stores a value of a function to prevent re-computation. This is an essential ingredient for lazy evaluation because lazy evaluation avoids repeated evaluation.

4 Lazy Evaluation

First lets look at the example in the previous section and incorporate argument sharing.

Outermost

$$
\begin{aligned}
&square(2 + 3) \\
&= (2 + 3) * (2 + 3) \\
&= 5 * 5 \\
&= 25
\end{aligned}
$$

The example of reductions doesn't perfectly illustrate how re-computation is prevented, but I will try to clarify it. The important difference is found on the 2nd step. Unlike the outermost reduction example in the previous section, once we compute the first $(2+3)$ expression, we can automatically reduce both occurrences of the expression in one step. As a result, we have fully reduced the expression in the same number of steps as inner reduction.

4.1 What is Lazy Evaluation?

Lazy evaluation can be thought of as outermost reduction + argument sharing. Thus by combining these two things, lazy evaluation is an evaluation strategy which holds of evaluation of an expression until its value is needed and it avoids repeated evaluation [TP]. With lazy evaluation, Haskell evaluates a value when it is needed then updates all copies of that expression with the new value.

4.2 How is Haskell Lazy and What Are The Advantages?

The first advantage of lazy evaluation in Haskell is that expressions are computed only if their value is necessary in this context. Lets look at a couple examples regarding this point. We will create a similar function in Python and in Haskell to demonstrate this advantage. The first function takes in two arguments and just returns the first one. First we will look at the python code

```
def func(x, y):
    return x
```

Figure 4: *A python function that returns the first argument*

Now lets write the same function in Haskell:

```
func :: Int -> Int -> Int
func x y = x
```

Figure 5: *A Haskell function that returns the first argument*

Both functions take in two arguments and return the first one. Pretty simple right? It seems as if the function in both languages will always return the same thing, but it turns out they don't.

For example, lets set our arguments to these values

$$x = 5$$

$$y = (5/0)$$

Then lets run both function as see what we get. First lets try Python

```
>>> def func(x, y):
...     return x
...
>>> func(5, 5/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Figure 6: *Division by zero error in Python*

Oh no! What happened here? If you are an experienced programmer, you would no that the result of dividing anything by 0 is undefined, and thus printing an error. What we did was pass (5/0) as the argument for y, and since uses applicative/innermost reduction, it first computes the value of the argument before passing it into the function body. Thus, it computes (5/0) and throws an error before we ever get into the actual function.

Lets now try the same function in Haskell! [1]

```
Prelude> func x y = x
Prelude> func 5 (5/0)
5
```

Figure 7: *Lazy Evaluation example in Haskell with no division by zero error*

WOAH! It works like magic! Why did this not result in a divide by zero error? The reasoning is that in lazy evaluation, the computation of arguments is delayed. Also, expressions are evaluated only if there value is necessary. Turns out that the second argument never got evaluated, since it was not needed in the function body! This example is very basic and simple, but it demonstrates how only necessary expressions are evaluated. This can significantly reduce the time complexity of an algorithm because it won't do unnecessary work. This can be further demonstrated by the next example

[1] If you installed Haskell properly, you can open up an interactive environment that runs Haskell code

```
func :: Int -> Int -> Int
func x y
    | x < 10 = x
    | otherwise = y
```

Figure 8: *Lazy Evaluation example in Haskell*

Now, the function body consists of a notation that we haven't seen before, lets break it down before we analyze what this function does. The pipe characters, | are known as guards. Guards are a way to see whether some expression is true or false. If its true, then the function will return whatever is on the right of the equals sign. It is very similar to if-else statements. Check out this website for more information about guards. So now we can see that this function takes in two integer arguments, then checks to see if the first argument is less than 10; if it is, then it returns the first argument. If the first argument is greater then 10, then it will return the second argument. Lets try this function with various inputs.

$$a = sum[1, 2, 3]$$

$$b = sum[1..]$$

Sum is a built in Haskell function that sums of the values in a list. The second list includes two dots (..) which creates an infinite list from $1 \rightarrow \infty$. (Side note, thanks to lazy evaluation, Haskell lists can be infinite)

Ok, now when we pass these arguments to the function $a \rightarrow x$ $b \rightarrow y$. The function returns 6 almost instantly. Again, since the first argument x is less than 10, we did not have to compute the second argument. The second argument is the sum of an infinite list which is impossible to compute. The function would have never terminated if it wasn't for lazy evaluation. Lets see what happens when we swap arguments

$$a = sum[1..9999999]$$

$$b = sum[1, 2]$$

In this case however, we are not summing from $1 \rightarrow \infty$ but instead, to an arbitrarily large number. After running this you will see that there will either be a stack overflow because the computation requires too much memory, or it will take a VERY long time to compute. I highly recommend trying this on your own computer to see how much longer the second case actually runs.

In the previous example, I briefly touched upon lazy evaluation supports programming with infinite list, thus in Haskell, we are able to define infinite lists. This is because using lazy evaluation, expressions are only evaluated as much as required to produce the final result. Therefore, when declaring a list that has infinite length, Haskell doesn't actually create or initialize this list. It really defines a potentially infinite list that only evaluates as much as required by the given context [GHutton]. This is because only the specific components of that list that we need will actually be allocated in memory, not the whole thing.

```
Prelude> func x y | x < 10 = x | otherwise = y
Prelude> a = sum [1,2,3]
Prelude> b = sum [1..]
Prelude> func a b
6
```

Figure 9: *Output of a Haskell function*

5 Conclusions

Functional languages, on the other hand, have programs that are built solely from function application and evaluation. In languages such as Haskell, everything is an expression, and the goal is to evaluate them in order to solve a problem. Haskell defaults to a method of evaluation known as lazy evaluation. Put simply, Haskell is lazy. It won't execute functions and calculate things until it's really forced to show you a result [LYAH]. Haskell uses lazy evaluation to evaluate expressions because it comes with several advantages.

- Function arguments are not computed before invocation
- Expressions are computed only if their value is necessary and it avoids repeated evaluation of expressions, thus making algorithms more efficient and reducing computation complexity
- Allows the ability to program with infinite lists
- Promises termination when possible

References

[PL] Programming Languages 2021, Chapman University, 2021.

[LYAH] Learnyouahaskell.com

[TP] tutorialspoint.com

[GHutton] FP 16 - Lazy Evaluation