CPSC-354 Report

Andrew Dillon
Chapman University

December 21, 2021

Abstract

The following report will touch on three primary areas. Firstly, we will discuss the programming language Haskell. This section will include a brief history of the language, why it was created, it's namesake, as well as a brief tutorial to get started in the language. Part two will dive into the theory behind programming languages and explore topics such as Unique Normal Forms, Confluence, and Termination. Finally, we will compare Haskell with an imperative programming language: C++. Lets begin with our look at the language itself.

Contents

1 Part 1: Haskell

1.1 Introduction

Haskell is a non-strict, functional programming language first officially released in 1990. Haskell was by no means the first functional language, and certainly not the last, but it has had a profound effect on programmers and the way the functional programming languages were used, developed, and taught and it has earned a place among the best known and loved functional languages.

1.2 A Look at the History of Haskell

Development of the language started three years before its first official release in 1987 at a conference in Portland, Oregon on Functional Programming Languages and Computer Architecture [HH:BLC]. At the time, dozens of similar programming languages were emerging. These all had varying levels of complexity, power and popularity. The committee realized that this over abundance of similar languages was ironically hindering the growth and expansion of functional programming languages as a whole. So the committee set to work attempting to develop a standard, common language to foster growth and interest in this functional programming languages as a whole. The result was what they dubbed "The Haskell Report" [H1](not to be confused with this report, which is also about Haskell).

The Haskell Report, otherwise known as Haskell 1.0 was the first iteration of the language. This version officially released after nearly three years of work in 1990 [DOH]. Later versions of the language also came accompanied by their own reports, giving programmers a rich source of information and documentation on how the language operates and how it can be manipulated. Some of these reports are in fact available online now for anyone to read through [H98]. .

1.3 Haskell's Namesake

Haskell was named after American logician and mathematician Haskell Curry. He is perhaps best known for the technique of known as currying (which again comes from his name). Currying is the process of breaking down a single function or operation which takes in multiple arguments and simplifying it to a series of smaller functions which each only take a single argument. This is a core principle of Haskell. Curry actually has three different programming languages named after him, these being Haskell(his first name), Brook(His middle name), and Curry(his last name). [HBC] Unfortunately, he did not live long enough to see any of these three languages developed as he passed away in 1982.

1.4 Functional Languages vs Object Oriented Languages

As stated, Haskell is a functional programming language: but what exactly does this mean? And how does it differ from other languages? Languages like C++ or Python are both imperative, object oriented languages, meaning that they focus around the data fed into the system and handling it. Conversely functional programming languages, as their name implies, focus on the functions and processes a program needs to perform [FPLvIPL]. Essentially the difference between focusing on the data and focusing on how to handle it

Most functional programming languages such as Haskell are also lazy languages. This means that a function will evaluate expressions when and only if their values are actually used by a function. For example take the function:

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

This function will take in to separate integer values called x and y, but will always return the second value; y. What happens if the function were to be called like this?

```
f (1+1) (2+2)
```

If this were an eager programming language (this is what the opposite of a lazy language is called), the compiler would fist calculate $(1 + 1) = 2$ then it would add together $(2 + 2) = 4$. After this, the function would return the value passed in for y, which in our case would be 4. Calculating $1 + 1$ was unnecessary, as it was never used in the function. While this is a very small step in this example, a more complicated function could take up value or a situation where a function like this is called often, would begin to slow down the program, leading to it being less efficient.

How does Haskell differ? A lazy programming languages like Haskell work around this only preforming these calculations when required (a process known as "call by need"). In this example, Haskell would only perform the operation $(2 + 2) = 4$, as that is the only step this function ever actually requires [LG].

We will take a deeper look at the differences, advantages, and disadvantages of both functional and imperative programming languages in Part 3 later on in this report, but for now this brief introduction should give us enough understanding about what the core differences are between the two.

1.5 Pattern Matching

Pattern Matching is perhaps Haskell's greatest strength, as well as its most fundamental property [PL]. Pattern matching uses multiple function definition all with the same name, but different inputs. Here is an example of pattern matching with a very simple function. It will identify integers 1, 2, 3, 4, and 5 and convert text, otherwise it will tell the user that the value is out of range.

```
sayNumber :: (Integral a) => a -> String
sayNumber 1 = "One!"
sayNumber 2 = "Two!"
sayNumber 3 = "Three!"
sayNumber 4 = "Three!"
sayNumber 5 = "Five!"
sayNumber x = "Not between 1 and 5"
```

We will get into the semantics of Haskell in a later section, but for know we can just speak broadly. The first line declares a function named `sayNumber`. On the second line, we can see that if the value passed to it is a `1`, then it will return the string `"One!"`. The next four lines similarly handle input for `2` through `5`. The last line handles a generic case `x`, where the input value is not within the range 1 - 5.

In a way, it operates similar to an if-then-else statement as found in other languages like C++ or Java. In fact, for comparison, let's look at how this function could look in C++.

```
string sayNumber(int x)
{
    string numName = "";

    if(x == 1)
    {
        numName = "One!";
    }
    else if(x == 2)
    {
        numName = "Two!";
    }
    else if(x == 3)
```

```
    {
        numName = "Three!";
    }
    else if(x == 4)
    {
        numName = "Four!";
    }
    else if(x == 5)
    {
        numName = "Five!";
    }
    else
    {
        numName = "Not between 1 and 5" ;
    }

    return numName;
}
```

Even if we were to compress the brackets and remove the white space, this version of the function is much longer to write out.

1.6 A Brief Tutorial

The best way to truly understand a new language is by getting your hands dirty and actually writing with it. For our example, we will create a simple program to add and subtract numbers written in Roman Numerals. All the code for this example can be found in the GitHub repository liked here and at the end of this report [GH].

1.6.1 Breaking Down The Problem

Before we write the Haskell code for this project, lets briefly discuss how to attack the problem. Take a moment to think about how we might be able to accomplish this task before reading on.

There are likely a number of ways the challenge could be handled. This is how we will take care of it in this example. Roman Numeral inputs will be taken in as strings. For example, the string: `VI` will be read as the Roman Numeral for `6`. We can leverage the built in operations Haskell can perform on integers. Rather than defining addition and subtraction for a new data type, we can instead convert the Roman Numerals to integers, add and subtract as need be, and then convert them back to strings.

Though for this example we will only be working with addition, we can easily expand it to incorporate other functions such as multiplication or exponents. We just need additional functions to act on our converted numbers.

1.6.2 Headings

Before we can start coding, we will need to create a new project. Your file can have any name, but it must end in `.hs` for the compiler to recognize it as a Haskell program. Let's set up our file with some imports and a main section for testing. for this program we will need to import the following:

```
import Data.List
import System.IO
import Data.Char

main = do
```

```
return ()
```

1.6.3 Converting Roman Numerals to Arabic

Once we've created our file, our first step will be converting stings to integers. To do this, we will need four functions. The first one will be called `romanToArabic`. It will take in a value of type String and return an Integer. To define the function we type the following:

```
romanToArabic :: String -> Integer
```

We start with the name of our function followed by two colons. Next comes the data types of the inputs and output. We can have as many inputs as we want (which we will break down using currying) but can only have a single return value, which will always be the last type listed. As written, this function will take in a value of type String and return one of type Integer. So ff we were to take in two strings and return an integer, we would write:

```
exampleFunction1 :: String -> String-> Integer
```

Now we will define our function. This one will primarily be used to call two helper functions that we will write in a moment. One to isolate the characters of the roman numeral and get the values of them, called `GetCharacters`, and one called `convertIntegers` which add up each value.

```
romanToArabic :: String -> Integer
romanToArabic romanInput = convertIntegers(getCharacters romanInput)
```

Once again we start with the name of the function. We follow this by a name for our String input: `romanInput`. We set this equal to the functionality of our function. In our case, we are passing `romanInput` into our yet to be written function `getCharacters`. All of that is being passed into our `convertIntegers` function.

Now lets work on `getCharacters`. This function will convert our String into a list of Integers. We'll want a list of Integers rather than a single value so that we can appropriately deal with the values when we call `convertIntegers`. This will make more sense when we begin to handle work with the list. To do so, we declare our function like so.

```
getCharacters :: String -> [Integer]
```

Now we will use Haskell's pattern matching and recursion to define two versions of this function to execute based on the input. First we will want a base case to return an empty list of Integers if the string passed to it is empty. Our other case will be for non empty strings. We write our variable as `(x:xs)` This notation is borrowed from discrete mathematics [PL]. As Strings are just lists of characters, we can say that `x` is the first, or head, of that list. The colon shows that it is concatenated to the rest of the list `xs`. By writing the list as a head and it's tail, we can pick the list apart and deal with it's individual elements when we recurse. Here we pass the first character into our next function, `getValueOfCharacter` and then pass the rest of our list back into `getCharacters` recursively. We will take care of each character in the string, get it's value, until we reach the end (in the form of an empty string).

```
getCharacters :: String -> [Integer]
```

```
getCharacters [] = []
getCharacters (x:xs) = (getValueOfCharacter (toUpper x)) : getCharacters xs
```

The function `toUpper` is in the `Data.Char` import package. We'll want to use this function here to check than any characters passed in are evaluated as capital letters. For example, if the user enters `ii` this will ensure the value of 2 is given rather than an error.

Now our pattern matching skills will really come into play. We'll need a case for each possible character in a roman numeral. We will also need an additional case to prevent errors if the user enters an invalid character.

```
getValueOfCharacter :: Char -> Integer
getValueOfCharacter 'I' = 1
getValueOfCharacter 'V' = 5
getValueOfCharacter 'X' = 10
getValueOfCharacter 'L' = 50
getValueOfCharacter 'C' = 100
getValueOfCharacter 'D' = 500
getValueOfCharacter 'M' = 1000
getValueOfCharacter x = error "Invalid character: Roman Numerals can only use I, V, L, X, C, D, &
    M"
```

Our last step in converting to Integers is perhaps the most important. Now we need to define how all our list of Integers are added together. To do this we will need to define the function `convertIntegers`.

The function `convertIntegers` will take in our list of Integers and return a singular Integer value. Much like `getCharacters` we will have two cases, one for an empty list, and one to handle a none empty list.

Our list of Integers is a list the represents the value of each Roman numeral character in the order of the original String. At first, it seems that we can simply add up each value in the list, but this is not entirely right. The number `6` is written as `VI` in roman numerals, so simply adding up the values would give us `5` for `V` plus `1` for `I` which is `6`. The problem comes when we want to deal with numbers with preceding values. The number `4` is represented as `IV` (being I before V or 1 before 5). The proposed algorithm would add up the `I` plus `V` an result in an erroneous `6`. To account for these values, we need to check if the current value is smaller than the next. If it is, we subtract it instead of adding it to the total.

```
convertIntegers :: [Integer] -> Integer
convertIntegers [] = 0
convertIntegers (x:xs) = do
  if xs /= []
    then if x < head xs
    then convertIntegers xs - x
    else
      convertIntegers xs + x
      else
      convertIntegers xs + x
```

Finally, we can do our addition and subtraction. These will be fairly straight forward.

```
addRomanNumerals :: String -> String -> String
addRomanNumerals num1 num2 = arabicToRoman (romanToArabic(num1) + romanToArabic(num2))

subtractRomanNumerals :: String -> String -> String
subtractRomanNumerals num1 num2 = arabicToRoman(romanToArabic(num1) - romanToArabic(num2))
```

This is where we can expand our little Roman Numeral calculator by creating functions like `multiplyRomanNumerals` or `squareRomanNumerals` following the same format as our add and subtract functions.

Now that we have all of our pieces we are ready to run the program. Or at least almost ready. We still need some way of interacting with the function. We can either interact with the functions directly through the command line or we can write some examples to test the program. These examples would best be written in our `main` we started earlier.

Here is what that could look like

```
main = do

  print $ "** TESTING addRomanNumerals **"

  print $ "IV - I = " ++ show (subtractRomanNumerals "IV" "I")

  print $ "X + X = " ++ show (addRomanNumerals "X" "x")


  return ()
```

Again, all the code for this project can be found in this GitHub Repository [GH].

2 Part 2: Programming Languages Theory

2.1 Introduction

An abstract reduction system can have Unique Normal Forms, Confluence, and or Termination. Not all systems have each of these three attributes. Some may only have two of these qualities, or just one, but not every combination is possible. In this section, we will take a look at the definitions of these concepts as well as see how they can be represented in diagrams. We will then investigate which of these combinations are actually possible.

2.2 Abstract Reduction System

An abstract reduction system, also called an abstract rewrite system or an abstract replacement system, is a way to break down and represent a program that transforms data and its possible outputs [EOU]. These systems can often be represented by a series of rules that define how a string may get rewritten.

Here is a simple example. Here we have rules that will act on strings containing the letters a,b,c, and d, and rewrite them into alphabetical order.

```
ba -> ab
ca -> ac
da -> ad

cb -> bc
db -> bd

dc -> cd

x  -> x
```

These rules each define one step to rearranging the letters in a string. The final rule represents the "default case" where two letters are already in order, such as "ab".

Abstract reduction systems can have a number of different attributes. Three very important qualities include having normal forms, confluence, and termination. We'll look at some simple mathematical equations to explain these concepts.

2.3 Normal Forms

A normal form is something that cannot be reduced any further. For a very simple example, we can look at the calculation:

$$1 + 2 + 3$$

As you likely know, this can be reduced in the following manner:

$$1 + 2 + 3$$
$$3 + 3$$
$$= 6$$

Once we arrive at 6, we cannot reduce any further. In this example, 6 would be our normal form.

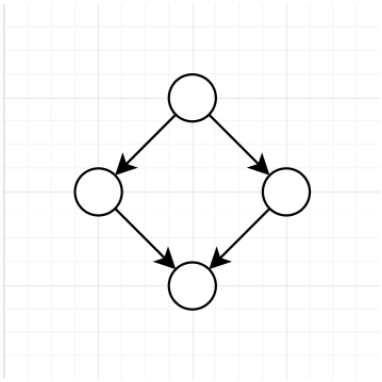
We can also say in this example that 6 is a *Unique* normal form. This means that 6 is the only possible deduction of the given function. There is noway for us to arrive at 7 or 5 or any other value. Consider the example:

$$\sqrt{9}$$

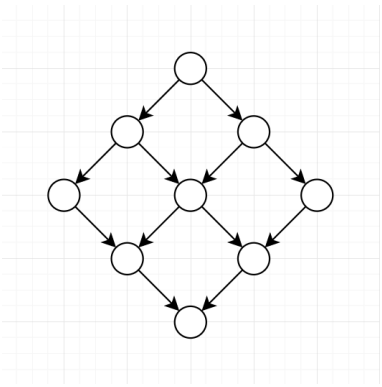
When taking the square root of a value, we will always have two possible results. Here we can arrive at both +3 and -3. As there are multiple possible conclusions, we cannot say that square roots have unique normal forms.

2.4 Confluence

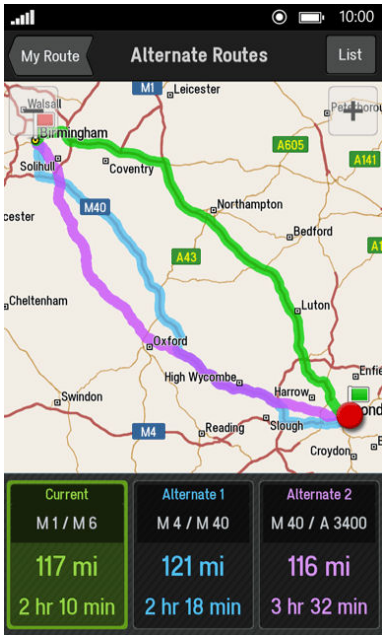
Confluence is the idea that all dividing calculations will reconvene to the same answer. This "diverging diamond" diagram is perhaps the simplest way to visually understand the concept.[PL]



An abstract reduction system is considered confluent if all possible paths reconnect. So if we expand this diagram, it'll look like this.



One way to think about confluence is a road map. If we have a starting point, such as home, and a destination, such as the grocery store, we likely have more than one route that we could take to get there. As we can see in the image below [GPS], all three of these colored paths lead to the same destination, despite taking different paths to get there. They could therefore be considered to be confluent.



What does this look like in math? Let's again look at

$$1 + 2 + 3$$

As we know, addition is associative, meaning we can choose to either add the

$$1 + 2$$

or the

$$2 + 3$$

. Using parenthesis we can visualize it like this.

$$(1 + 2) + 3 or 1 + (2 + 3)$$

Now lets reduce these individually.

$$\begin{array}{cc} 1 + 2 + 3 & 1 + 2 + 3 \\ (1 + 2) + 3 & 1 + (2 + 3) \\ 3 + 3 & 1 + 5 \\ 6 & 6 \end{array}$$

We can see that no matter which part of the calculation we do first, we still arrive at the same conclusion, therefore it is confluent.

If a function is not associative it will not necessarily be confluent. Let's take a look at one that has caused a bit of controversy in the mathematical community. [WI22]

$$-3^2$$

Depending on how you interpret the statement, you can arrive at two different conclusions. If we interpret it as being "the negative of 3 squared" we will get:

$$-(3^2)$$

$$-1 * (3 * 3)$$

$$-1 * (9)$$

$$-9$$

If however we interpret it as being "negative 3 to the second power" we will calculate:

$$(-3)^2$$

$$-3 * -3$$

$$9$$

"9" is not equal to "-9", therefore without a consistent order of operations, we do not have confluence. Following "PEMDAS", most mathematicians believe that the former calculation is correct. Though some people still argue for the latter.

2.5 Termination

A program is considered terminating if every possible function or calculation has a definitive stopping point. This means that there can be no infinite loops. This needs to be true of potential paths for a given program. If only some of the functionality comes to an end, it is not considered terminating, but instead normalizing. Going back to our first example, "1 + 2 + 3 = 6". It terminates at 6.

Essentially, a terminating function is one that is not infinite. Pascals Triangle is one example of an infinite series in mathematics.[PT]

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ & 1 & & 4 & & 6 & & 4 & & 1 \\ 1 & & 5 & & 10 & & 10 & & 5 & & 1 \\ 1 & & & & ... & & & & & & 1 \end{array}$$

This pattern can continue infinitely and therefore does not terminate.

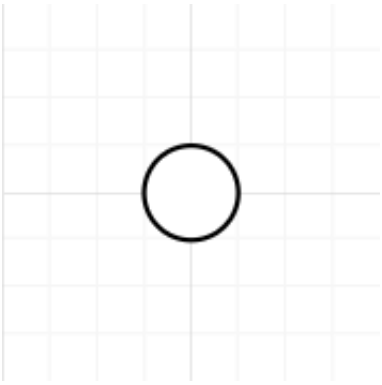
2.6 Combinations of these Attributes

As we have discussed, Abstract Reduction Systems can have normal forms. They can be confluent or not, and they can either Terminate or be infinite. Each system will be on one side or the other for each of these conditions. For example, a system could have unique normal forms, confluence and terminate, or perhaps it could terminate, but not have unique normal forms or confluence.

We'll try to draw it. diagrams to determine which ones are possible and which are impossible. But not every combination of these are possible. Let's take a look at each possible scenario and determine which are possible and which are impossible.

2.6.1 Case 1: Unique Normal Form, Confluent, and Terminating

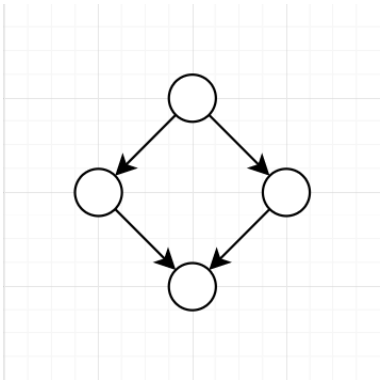
Our first case is when all three are true. Firstly, the diagram must end with a single normal form, thus it must end with a single dot.

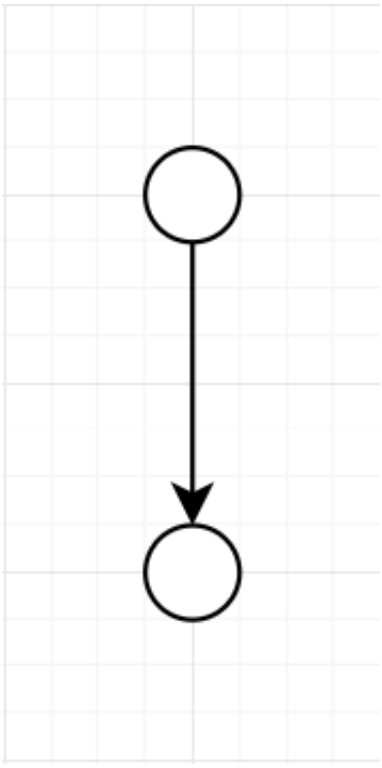


Now to deal with confluence and termination. Before we add anything else to the diagram, let's check to if either of these properties are currently satisfied. Confluence means that any time the program can choose between two paths, that the paths will reunite. As our diagram currently stands, we have confluence. There are no separate paths to take, therefor, all path divisions do reconnect.

Now lets check for termination. To ensure termination, we need to not have any loops in the program. As it stands, there is no way to loop. Therefor our simple one step diagram fully represents an abstract reduction system which has unique normal forms, confluence, and termination.

This single dot diagram is by no means the only example of an Abstract Reduction System that us has all three of these properties, but it is the simplest. Another example would be the diamond diagram that we used to explain confluence in the first place. Additionally, A model wherein there is a calculation to perform, but no choices to make would also meet all three of these requirements for this situation.



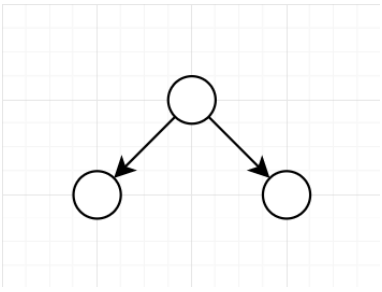


We can therefor say that Case 1 is valid.

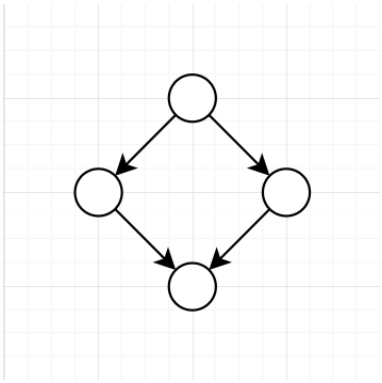
Case 1: Unique Normal Form, Confluent, and Terminating ✓

2.6.2 Case 2: Non-Unique Normal Form, Confluent, and Terminating

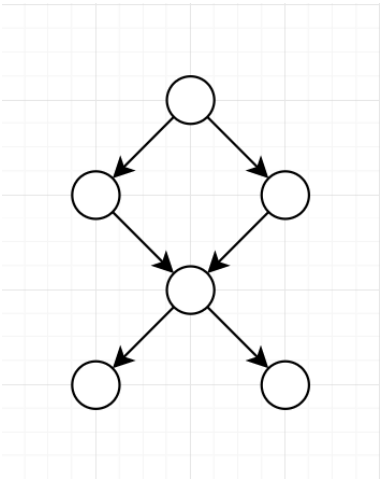
Our second case involves both confluence and Termination, but does not have unique normal forms. Let's think about how we would draw this out. As we know it must not have unique normal forms, we'll start with two separate ending dots as separate paths from a starting point.



As each of these points is a normal from, and as there are no other paths to take, we can say that this Abstract Reduction System Terminates as well. It is however not Confluent, as these two normal forms do not reconnect. To fix the confluence we'd need to connect the two like this:



This, as we just recently discussed in Case 1, also has Unique Normal Forms. To correct this we'd need to draw two splitting paths at some point in the diagram:

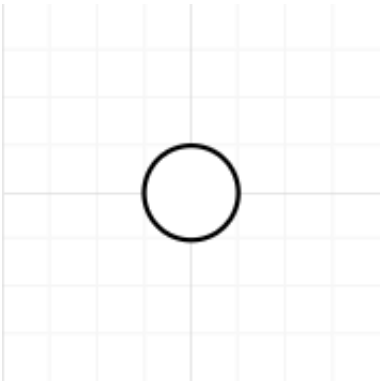


This brings back our Non-Unique Normal Forms and keeps the Termination, but we once again lose Confluence. We could combine the paths again, but we'd just be stuck in an endless loop bouncing between Confluence and Non-Unique Normal Forms. Due to this, we can say that there is no possible way to represent an Abstract Reduction System that is both Confluent and Terminating but does not have Unique Normal Forms [[PL]].

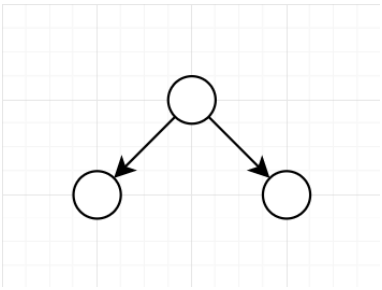
~~Case 2: Non-Unique Normal Form, Confluent, and Terminating~~

2.6.3 Case 3: Unique Normal Form, Non-Confluent, and Terminating

Our third case features both Unique Normal Forms and Termination, but is not Confluent. In a way, it is almost the opposite of Case 2. To go about drawing it, we'd want to start with a single dot again. This will cover both Unique Normal Forms and Termination.



Now as we discussed in case one, this single dot diagram is confluent due to it not having any diverging paths that need to be re-conjoined. To eliminate confluence, we'd need some sort of splitting path.



This does however run us into trouble with not having a single or Unique Normal From. Much like Case 2, we are stuck alternating between having satisfying our Confluence aspect or our Unique Normal Forms. We can therefor say that Case 3 is also impossible.

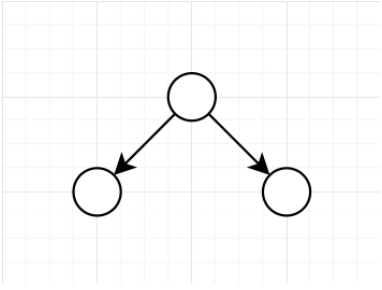
~~Case 3: Unique Normal Form, Non-Confluent, and Terminating~~

Would

2.6.4 Case 4: Non-Unique Normal Form, Non-Confluent, and Terminating

At this rate, it may seem like the only possible combination is when all three attributes are true. Let's look at Case 4 to see if this statement is true.

Case 4 is an Abstract Reduction System which does not have Unique Normal Forms, is not confluent, but does Terminate. How could we go about drawing this? We'd need something that has dividing paths that has at least two different end points. We could draw this divergence like so:

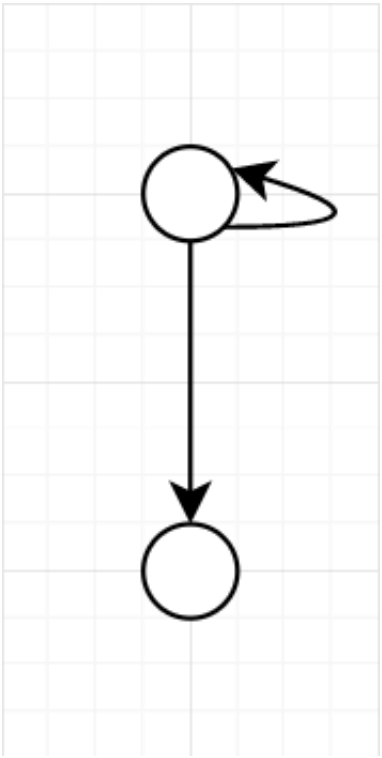


This has two possible ending points, therefor it does not have Unique Normal Forms. It is also not Confluence, as these two individual paths never meet back up. Does it Terminate, though? Yes, it does. There are no loops for the program to potentially get stuck in. Therefor it satisfies each of the criteria for this case. We can therefore say that:

Case 4: Non-Unique Normal Form, Non-Confluent, and Terminating ✓

2.6.5 Case 5: Unique Normal Form, Non-Confluent, and Non-Terminating

Case 5 describes an Abstract Reduction System in which there is no Confluence, no Termination, but that does have Unique Normal Forms. This means that the system must have in it somewhere a potentially infinite loop, but also have the option to arrive a a single unique conclusion.



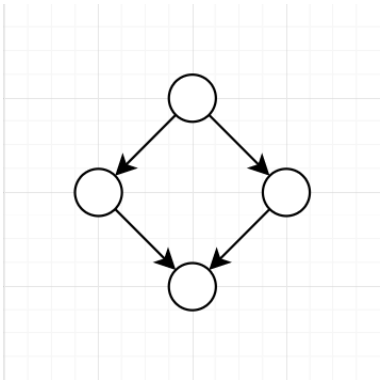
Does this diagram satisfy our the "Non-Confluent" aspect of this case? Let's take a look at it. A system is considered Confluent if all diverging paths eventually reconnect. As the two paths in this diagram end in different places, this system is cannot be considered Confluent. Therefor, Case 5 is possible.

Case 5: Unique Normal Form, Non-Confluent, and Non-Terminating ✓

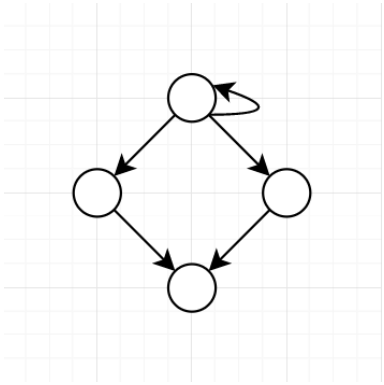
2.6.6 Case 6: Non-Unique Normal Form, Confluent, and Non-Terminating

Case 6 is an Abstract Reduction System that only has Confluence. This means that it is both Non-Terminating and does not have Unique Normal Forms. Based on some of our previous cases, it would seem that this case might also be impossible. This stems from the fact that both Case 2 and Case 3 also have opposing attributes for Confluence and Unique Normal Forms. It may stand to reason that this case could also be impossible. Before jumping to any conclusions though, Let's take a look at it to see if that is so.

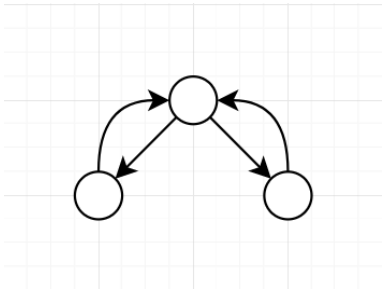
Confluence states that all paths must eventually reconvene. Let's start diagramming with our standard diamond diagram. From here, it may seem impossible for us to add on to it in any way that we could meet our other requirements.



But as we know, we need some way of preventing termination through an infinite loop to prevent Termination. Simply adding another branch which loops would make the Abstract Reduction system Non-Confluent.



At first this may seem impossible. How can we add to a Confluent System to make it non terminating? Adding on to the system is actually just the problem. If the point where these paths reconvene the same as the starting point, then we've created infinite loops while also forcing all paths to reconnect.

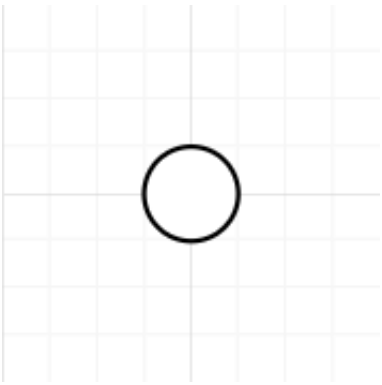


In this case, the two branches and their "returning loops" must be opposites of each other. For example, if one branch were to add 3 to a value, the next step would subtract 3, returning to its initial position. Therefor, this is once again, a valid case.

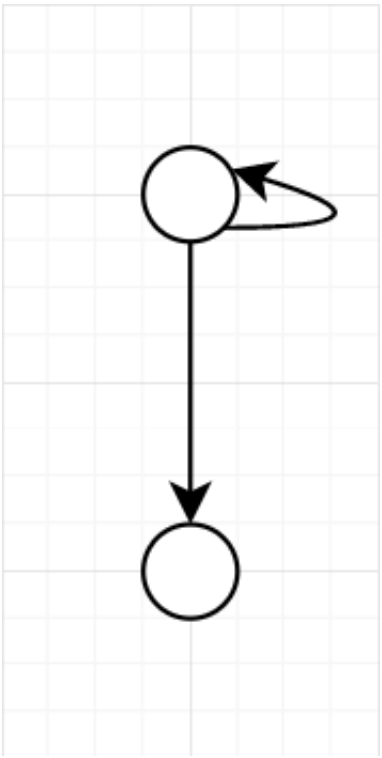
Case 6: Non-Unique Normal Form, Confluent, and Non-Terminating ✓

2.6.7 Case 7: Unique Normal Form, Non-Confluent, and Non-Terminating

Case 7 only features Unique Normal Forms, without Confluence or Termination. As Case 6 proved, Unique Normal Forms is not inherently tied to Confluence, so this case could be possible. Let's think about how we could go about drawing this out. We know it has to end in a Unique Normal From, so we can start with a single dot to end with.



We could make this system Non-Terminating by adding in a loop that returns back to itself as we did for Case 5.

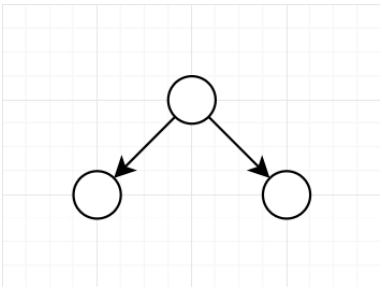


Our last challenge is to make this system Non-Confluent. How could we go about doing that without affecting the Unique Normal Forms? Adding a diverging path at any point in this diagram would create a second Normal From. Therefore, Case 7 is another impossible Abstract Reduction System to create.

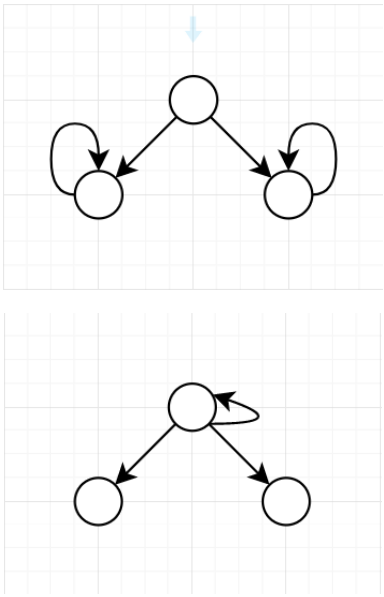
~~Case 7: Unique Normal Form, Non-Confluent, and Non-Terminating~~

2.6.8 Case 8: Non-Unique Normal Form, Non-Confluent, and Non-Terminating

Finally, Case 8 is one in which the system does not have Unique Normal Forms, is not Confluent and does not Terminate: the exact opposite of Case 1. To draw this case, let's start by first drawing a diverging path. This will give us both Non-Unique Normal Forms and a Non-Confluent System.



Now all we need to do is add in some sort of loop to make it non terminating. We can either add a loop at our starting point or to one (or both) of our end points.



Case 8: Non-Unique Normal Form, Non-Confluent, and Non-Terminating ✓

2.7 Conclusions and Inferences

Case Number	Unique Normal Forms	Confluent	Terminating	Is Possible
1	true	true	true	true
2	false	true	true	false
3	true	false	true	false
4	false	false	true	true
5	true	true	false	true
6	false	true	false	true
7	true	false	false	false
8	false	false	false	true

Looking at the table, we can see that only three of these combinations are completely impossible: Case 2, Case 3, and Case 7. What can we glean from this table?

Firstly, if an Abstract Reduction System has Unique Normal Forms, it must also be confluent. Thinking about this, that makes sense. If every possible path needs to end at the same result, then any split in the path to that result must eventually recombine. However, this statement does not hold true the other way around. A system can be confluent without having unique normal forms. Case 6 proves this to us showing an example where a system is Confluent but does not have Unique Normal Forms.

Written more succinctly:

If something has unique normal forms, it must also be confluent.

If something is confluent it does not necessarily have unique normal forms.

Or otherwise stated:

$$UniqueNormalForms \Rightarrow Confluence$$

$$Confluence \nRightarrow UniqueNormalForms$$

3 Part 3: A Short Comparison Project

3.1 Introduction

Functional programming languages and imperative programming languages are both very different. They have their own advantages, disadvantages, and best use situations. This section will dive into the differences between the two types and a case study comparing Haskell with an imperative programming language. In our case, we'll be be putting Haskell up against a C++; comparing the differences and similarities, advantages and disadvantages, as well as looking at some more subjective view points about the two.

3.2 Differences

As stated, these two types of languages have some very significant differences. These difference cover not only the way the language is written, but the way a programmer will approach a problem. Here are some of these differences.

3.2.1 Syntax

As with any two languages, Haskell and C++ have different syntaxes. Many of the differences in syntax comes from the fact that Haskell is a functional language and C++ is imperative. For example, C++ uses brackets to truncate code into sections. Functions, loops, and most importantly, classes and objects are set apart in brackets. In Haskell, functions tend to be much simpler, as they only handle one case. Haskell organizes it's functions in list, which while not entirely unlike C++, is less truncated.

3.2.2 Order of Execution

In languages like C++, the order the code is written in plays a major role in determining how a program will execute. For example, the code written on line 1 will be executed before the code on line 2, before the code on line 3, and so on. In stark contrast, a program written in Haskell cares little about the order its written in. This is due to Haskell's use of pattern matching (as we have discussed a few times previously)[FPLvIPL]. Rather than following the order, the compiler searches for the function that matches the inputs given, executes those commands and returns to the calling function. This is not to say that the order plays no roll in Functional Programming Languages, just that it plays less of a roll than it does in Imperative Languages like C++.

3.2.3 Flow Control

Building off of the order of execution, is the idea of hope the flow of a program is controlled [FPLvIPL]. Again, Haskell primarily uses pattern matching based on function calls to determine the overall flow of the program. C++ of course uses function calls and recursion, but most of the control is governed by loops and conditional statements such as If-Then-Else blocks or switch statements.

3.2.4 State and State Changes

In Haskell, everything defined in a constant. Once the value of something is set, it cannot be changed. In C++, programmers have the option of defining both constants (who's values cannot be altered during the running of the program) or variables (which as the name implies, can vary and change during a program's execution) [DBFaIP]. Rather than changing a variables value, Haskell will cleverly utilize new constants in place of a state change.

3.2.5 Encapsulation

Encapsulation is the idea or linking associated data into larger pieces of manageable [WIE] For example, a student's name, birthday, and student ID number could all be grouped together in a single student object [FPvOOP]. At least, this is the case for C++ and other imperative programming languages. C++ and other imperative programming languages tend to be Object Oriented, the main goal of which is to encapsulate data an deal with it in these larger units.

While it is possible to take an object oriented approach in Haskell, it is not the main purpose of the language [OOPH]. Why is this? The answer to this question ties partially back to how both of these languages deal with state. Being object oriented is difficult when there are no variables to change

3.2.6 Similarities?

Though they have many differences, there are some similarities. According to Luis Pedrol's retrospective article *I tried Haskell for 5 years and here's how it was*, "Like C++, Haskell is (in part) a research project with a single initial Big Idea and a few smaller ones. In Haskell's case, the Big Idea was purely functional lazy evaluation (or, if you want to be pedantic, call it "non-strict" instead of lazy). In C++'s case, the Big Idea was high level object orientation without loss of performance compared to C," [ITHF5Y].

Pedrol goes on to describe some more similarities he found in his experience. He says that the error messages to be similar. Additionally, the versatility each language offers gives programmers a variety of ways to tackle the same problem.

3.3 Which Is Better?

So, with all of these differences, which language is better? Is there a better language, or is the better language a situational decision one needs to make when programming?

3.3.1 Reasons For Haskell

There are those who believe that Haskell is the better language. Proponents of the language argue that "it is pure and does not mix other programming paradigms into the language. This forces you to learn functional programming in its most pure form. You avoid falling back on old habits and learn an entirely new way to program," [CvH].

It is also ideal if the goal is to use a functional language. This may sound like a strange argument, but if the situation calls for a functional approach, it would be far easier to write the program using Haskell than attempt the same with C++ or a language that was intended to be objected oriented. While C++ can be used in a functional way, Haskell being a purely functional language from the get go makes it the ideal choice for situations when a functional language is preferred.

While it may be difficult to read for newcomers to the language, Haskell functions actually do a good job of laying out their functionality to programmers. By using pattern matching, it's easy to identify the path that a function call will take, rather than having to weave in and out of different if-then-else and switch statements [ITHF5Y].

Additionally, Haskell has a number of well designed libraries that can help programmers build their code without needing to reinvent everything.

3.3.2 Reasons for C++

Others claim that C++ is the better language. There are a variety of reasons people use to support this claim. One of C++'s most known attributes is it's speed. "Has imperative, object-oriented and generic programming features, while also providing the facilities for low level memory manipulation. C++ compiles directly to a machine's native code, allowing it to be one of the fastest languages in the world, if optimized," [CvHwatD]. While Haskell is fast, C++ has the advantage of only being one step above assembly [Q].

There seem to also be far more programmers who use C++ than there are who use Haskell. According to Stack Share, "C++ has a broader approval, being mentioned in 199 company stacks and 371 developers stacks; compared to Haskell, which is listed in 33 company stacks and 47 developer stacks," [CvHwatD]. Having more users also opens up the language to having more resources, both in the forms of books, blogs, and tutorials, as well as being able to find people such as teachers or colleagues to lend assistance.

This is not to say that there are no resources for Haskell, books like *Real World Haskell* by Bryan O'Sullivan, John Goerzen, and Don Stewart is a great way to learn the language [T10BtLH]. Similarly, Miran Miran Lipovača's *Learn You a Haskell* and its accompanying website give programmers plenty of ideas and resources to find out more about the language [LYAH]. The point still stands, though, that there are far more resources on C++.

Additionally, C++ has far more similarities to other languages like Python and Java, and an even closer relationship with both C and C#. Programming theory and paradigms from these languages can be more easily transferred over to C++ than to Haskell.

3.3.3 The Better Language

There is no perfect, "One-size-fits-all" language. If you were building a house, you wouldn't use a hammer for everything: sometimes you'd use a screwdriver, sometimes the table saw, sometimes the sand paper. In the end it comes down to a case by case decision about what both the programmer and the programmer needs. If a functional language is required, Haskell is a great option. On the other hand, if the program would work best with lots of objects and be written in an imperative way, C++ would probably serve the program better.

4 Conclusions

In summary, Haskell is a functional programming language, which is a language based around the creation and execution of functions to break down problems on a case by case basis as opposed to being designed around the data the program will handle. It is notable not only for the currying it employs (based on it's namesakes life work and research), but also for the role it played in bringing more attention to idea, development, and use of functional programming languages as a whole. We went over an example of a simple project to introduce the logic and basic syntax of Haskell, all the code for which is included in the GitHub Repository linked here [GH].

In Part two, we discussed some of the theory associated with functional programming languages. In particular we discussed the concepts of unique normal forms, confluence, and termination. We used both mathematical comparisons as well as numerous diagrams to illustrate these concepts. Additionally we took a look into which of combinations of these properties can be mixed together in order to form different types of systems.

Part three focused on comparing functional programming languages with imperative, object oriented languages. To do this, we took Haskell and compared it to the ever popular C++, examining differences in syntax, structure, and purpose. Finally, we took a look some opinions and arguments to see if we could determine which language was superior, though we eventually decided that the better languages is very subjective and changes on a case by case basis. There is no one size fits all language, only the one best used for the situation at hand

References

[HH:BLC] A History of Haskell:Being Lazy With Class, Paul Hudak et al, 2007.

[GH] Andrew Dillon, 2021

[T10BtLH] Alex Turner, 2021

[ASCII] ASCII Art Generator — 2021

[BHH] Brief History of Haskell, Jeremy Singer, 2021.

[DOH] Cleaverism 2021

[CRDH] Convert a Roman number to a decimal number in Haskell (Code) — Haskell is not difficult, Gautam Mokal, 2021.

[OOPH] Edsko de Vries, 2018

[FPLvIPL] Functional programming vs. imperative programming (LINQ to XML), 2021

[DBFaIP] Geeks For Geeks, 2021

[HBC] Jonathan P. Seldin

[LG] Lazy is Good, Jeremy Singer, 2021.

[ITHF5Y] Luis Pedrol, 2017

[LYAHSiF] Miran Lipovača, 2011

[LYAH] Miran Lipovača, 2011

[H1] Mission Valley Software 2021

[EOU] Oxford University, 2019

[AHOH] Pablo Buiras, 2016

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

[FPvOOP] Priya Pedamkar, 2021

[Q] Quora, 2021

[GPS] Rafe Blandford, 2013

[WI22] RD, 2021

[CvH] Slant, 2021

[WIE] Sumo Logic, 2021

[CvHwatD] Stackshare, 2021

[H98] The Haskell 98 Language and Libraries, 2002

[PT] William L. Hosch, 2013