CPSC-354 Report

Corey Spielman
Chapman University

December 19, 2021

Abstract

In this report, the theory of programming languages is explored through the use of Haskell, as well as Lambda Calculus. This report regards the widespread unforeseen vulnerabilities built into complex programming languages, as well as explores the benefits of recursion over algebraic data types.

Contents

1 Introduction

Programming languages can be divided into many different types, ranging from object-orientated, iterative, imperative, logic-based, functional, and so forth.

All these different types of languages have definite differences in the way they are organized, however many modern day programming languages can support many different types of programming styles.

So what does the basis of a programming language fall onto, and how does the intended style of programming affect the real-world when the code is compiled?

To answer these questions, I take an in-depth exploration into the structural integrity of the Haskell programming language, compared with how a typical modern non-functional programming language interprets and compiles the two.

In addition, I explore the real-world consequences of unintended coding side effects, and how a newcomer could familiarize themselves in a new coding environment.

---

2 Haskell

2.1 The Fruit

Haskell compared to another language (like C#) is like apples compared with oranges.

Yes, they are both similar in the sense that they can be used to write computer code, and in the sense that both examples are fruit.

But really taking a look reveals how different they look, how the colors are completely different, and most importantly, how they taste completely different.

Haskell is built along the same structural line that principles every other language.

In-depth, this means that they all take human readable code, and use a compilation process to convert the software into a faster and more efficient form for the computer, a computer readable form.

These languages use a version of a type, which in the most basic form is just a value that can be fixed, determined at run time by a user, or determined using mathematics depending on which is more efficient for the hardware itself.

The real *fruit of the labor*, is how the languages are predetermined to operate, creating two major categories of computer languages to fall under; non functional programming languages, & functional programming languages.

Haskell is a firsthand example of a functional programming language, as it has a set of strict rules it must follow, and does not allow deviations in the form of complexities.

The variables it allows to be created and modified are under predetermined guidelines, meeting the criteria that they can not be added onto.

In most well recognized object oriented programming languages and/or imperative programming languages (for example C#), a data type can exist in multiple different types, such as a boolean, an integer, a character, etc.

The problem here is that with each of these added complexities, there become more and more ways for the system to fail.

With each new type comes its own problems, causing all sorts of unforeseen consequences throughout the years.

An infamous real-world example of an unaccounted software vulnerability in a complex imperative programming language's code was the case of the Ariane 5.

[FB] This was a rocket designed to carry a satellite into orbit by the European Space Agency in 1996, and everything should have gone perfectly except for one small mistake.

The memory size of a variable was not prefixed, and as such the rocket was only able to travel a short distance before it exploded.

This is recognized today as an Integer Overflow, and can be fixed with a few lines of code. Unfortunately, at the time of launch, this was unforeseen by the developers.

As such the computer tried to put a 64 bit integer into a 16 bit variable, which ultimately was the cause of the explosion (as reported by the investigators).

After the debris had been cleaned up, an estimated $370 million was lost due to the explosion.

Luckily this was one of the more expensive mistakes caused by overlooked software vulnerabilities.

If the very same scripts were written using functional programming languages, there would quite literally be no room for these types of mistakes.

Haskell does not allow for the possibility of an Integer Overflow, and as such would not be able to explode the rocket in that way.

At least in theory, with no room for error, Haskell would be preferable as a programming language in extremely high stake scenarios such as rockets climbing to the stratosphere.

2.2 The Weight

The two programming paradigms present very clear and strict strengths, as well as weaknesses.

In the case of a non-functional programming language, with each new type of operator comes it's own rules it follows.

Any deviations from these predetermined states will cause an error with the compiler.

In best cases, this will simply throw an error message to the users console, and can be fixed without further problems.

However, with each added complexity, more and more possible situations must be taken into account, to prevent further problems.

So with each new complexity added there are more problems to be solved. This can be both a strength, or a weakness depending on the situation at hand.

If you are writing a small-time script with only a simple purpose, sometimes these possible situations won't be acted upon, leading to no issues in the run-time for the code.

As long as it stays that way with no additions, there will be no problems.

On the other hand, if the code is being used in a high staked operation, such as interstellar navigation, global communication, or national security to name a few, every single possible future *must* be taken into account before the software is deployed.

As you can see, this is very dependent on the situation, but in the long term it is an issue that should be taken into account when programming.

A functional programming language, on the other hand, is extremely strictly typed, and must 100% follow the rules set out to guide it.

There is no room for additions to a variable, it is set as one thing and that is it.

There is no need to define integers from characters, as the programming language functions only on the core values.

Without any room to add complexity to the types, Haskell while being a strictly typed functional programming language does not need to check for such problems like memory allocation, overflows, etc.

It all boils down to the circumstances of what the code is being used for, and depending on the use this can be seen as an advantage through it's disadvantages.

2.3 The Approach

An important thing to note about the two types of programming languages is that the run time of the code largely does not depend on the style itself [IC].

Rather, the run time of the program depends on how much complexity is stored, must be interpreted, compiled, and then deployed.

This is crucial as it means that from a strictly programming view, there isn't a reason to choose one over the other, to get the most efficient run time.

When a newcomer wants to learn a functional programming language, they have a ton of resources to help them explore.

The best way for a newcomer to learn in particular is by actually trying and failing over and over, learning from their mistakes to push them forward until they understand the problem at whole.

Most often a newcomer to a functional programming language will take the approach of trying to write in the style of an object orientated language, or an imperative language.

This means that they will take a stance from the beginning, trying to write something in a completely foreign mindset.

The problem here is that they are using their past knowledge of how a programming language works, and usually this is the best approach to trying something new.

However, in this case this technique is actually the key to their downfall.

By approaching Haskell or any other functional programming language like it were a more standard type of language, they are thinking about the wrong type of problems, and are already working on those solutions.

They *must* take a brand new approach to their thinking, *and define in their own minds that their code needs to be structurally organized in a different manner* than they thought.

Haskell and another programming language (for example C#) start off the same way, where the user must define the work space as they planned out.

They must create and define the variables necessary for the script to operate, but after that is where the two diverge into completely different spaces.

There are actually some programming languages that are created by design to offer multi-paradigm programming, instead of trying to focus solely on one approach[IC].

Java has the reputation across the globe as one of the most pure object orientated languages, however it's not 100% object orientated.

With Haskell, everything operates purely on the base functions. In short, this means that new variables are generated as an output, instead of changing the variables state.

This is completely backwards when compared with a non-functional programming language, where variables are designed to be constantly modified and updated.

There are several advantages to this, one most notably being there is no need to worry about a variables shared state throughout the program, as the original definition will never be modified.

Additionally, depending on the type of program being written, this can lead to less complexity throughout the program, making it easier to understand and navigate.

In a non functional programming language, such as an object orientated programming language, there exists a whole other world of classes and objects not found within a functional programming language.

As a quick summary, a class is what actually defines the data structure of the object, which determines on what kind of value the object takes over.

Within are the member functions that actually establish how the variables should operate, so the system is built in place to support modifications to the limit.

2.4 The Inside

The Objects themselves are merely an instance of the classes, and the objects can have a wide variety of data inside, ranging from a simply data type to a complex list.

Whereas in Haskell, there is none of that to be found. Haskell takes the same principle, but adjusts the angle a bit so it accomplishes the same problem even though a completely different approach was utilized.

Haskell is a functional programming language, which means it only operates on functions.

Functions themselves are simple methods of conversion from an existing data input, and modifying the output based on it's goal.

This is the most important part of the theory behind Haskell, that it only operates on functions.

In the example of recursion with the Fibonacci Sequence, the original variables are defined from the start, and the function is written in a way that allows recursive use for the entire infinitely long sequence of numbers.

There is absolutely no way to modify the variables after they have been defined, which is what makes the functional programming language so difficult for a newcomer to pick up.

They are used to the standard programming language where variables are mutable throughout the program, and this is the opposite of that case.

Since the variables are not mutable, there exists no concern for worries about unforeseen consequences in that regard.

There is one type of variable and once it's set, it's not being modified or erased unless a function utilizes it for something else.

With a function as simple as possible, the parallel processing done across the computer can actually be more efficient.

In addition to this, the simpler a function is the easier it is to test.

This can be useful for resolving every single bug possible, if that is necessary for the utilization of the program.

Having a function be as simple as possible like in this case is absolutely possible to write in a non functional programming language, such as an object orientated programming language, however typically it doesn't actually make much sense.

It wasn't the intended design of the programming language to be written in such a way, and as such usually becomes more complicated than originally intended.

The downside to having a functional programming language like Haskell only operate using functions is that most of the time, it is the function itself that is prioritized instead of the data in question.

Having a function as described above would be the most efficient way of getting a solution to a problem, however it is there that the function stops being useful.

It is created with the purpose of having a single function, with no room for deviation or mutability.

One of the main remarks about Haskell one might hear from time to time is the thought process that the programming language itself is rather lazy.

Unlike in imperative languages, author Bartosz Milewski goes on to write, when writing functions in Haskell, a function is really a function [LZ].

The difference between one of these functions and a normal function can be found when determining what it is that makes a pure function so pure.

A pure function must return the exact same parameters each and every time it is called using the same arguments.

What this means is that a function does not have any state registry, and along those same lines it can't follow an external state either.

Each and every time it is called in the software, it must be reanalyzed to determine what it stands for.

This could be seen as inefficient, but it also does have its practical applications as well.

Additionally, a pure function shouldn't have any side effects.

If a function is called once, it should be the same result as calling the same function twice, but then discarding the result that it got for the first time around.

If the result of any function is discarded, Haskell never actually calls the function.

This is done for a variety of reasons, but most importantly for efficiency.

Because a pure function has no side effects, Haskell is able to execute functions in parallel with each other super easily.

When someone says that Haskell is "lazy", they don't mean it in the literal sense.

Laziness in this context refers to the fact that Haskell only calculates operations it needs to.

If a function is called in the software, Haskell waits until the last possible moment to evaluate it.

Most other programming languages take the opposite approach, and try to evaluate all functions when the code is compiled.

Haskell though, just kind of assumes that since the function isn't in use right now, it may not be in use at all.

Again, this is extremely useful to the efficiency of the software, as it can save valuable run time for the user.

2.5 The Core

Within Haskell there is a specialized class called a Monad. Haskell's Monad class sets the guidelines as an interface for programming arbitrary data, or control structures.

According to the Haskell wiki, all common monads are members of the Monad class.

```
-- Haskell's Monad Class

class Monad m where
  (>>=) :: m a -> ( a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

All instances of the class Monad have a strict set of rules they must follow, which are called the Monad Laws[MD].

```
-- The Monad Laws

return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Additionally, Haskell is built upon a specific type of mathematics that is unseen in most other programming languages. I'm talking of course, about recursion over algebraic data types.

The best example of this can be seen in a similar fashion to how Professor Kurz represented the Fibonacci sequence [PL].

When creating a function, one must take into account the name of the function, the equations that define the function, and by approaching the smallest value of data, rather than the largest[AK].

```
-- The Fibonacci Sequence in Haskell

fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

It is crucial a newcomer to Haskell as a functional programming language can clue into the differences between the methods of approach.

On one hand, the program can be directed to be overtly complex, take into account millions of possibilities and have a calculation solved for any that are tried.

On the other hand, instead of focusing on getting as much data measured as possible to fit in, it's crucial to make sure that even from the first steps.

In short, approaching the smallest amount of data can be the more beneficial approach overall.

Because Haskell has no way to create mutable objects, all of it's objects must be immutable.

This can be seen by taking an example like the Fibonacci Sequence, and analyzing how the variables are declared.

They are static, defined as 0 and 1, and will not be modified or changed in any way when the program is compiled.

This is crucial for developers, because given the right circumstances, there are many situations where writing the program with only immutable objects would actually be extremely beneficial to the core strength of the program.

2.6 Specifically...

Specifically in an object orientated programming language, whether or not the object is mutable or immutable can be left up to the authors choice.

This may sound like a good thing, but with this newfound freedom comes severe drawbacks.

The specifications of the object orientated programming language are what define when an object can be mutable, but many languages also utilize a method involving displaying particular objects like strings, as immutable objects.

This leads to all sorts of benefits, like the actual run time of the code receiving a significant speed advantage, and can help the developers of the code itself with the readability.

Most importantly though, is that immutable objects are extremely useful when working in a specifically multi threaded situation.

This is because there won't be a need to worry about any other thread accessing the same resource and trying to modify it.

Haskell may lose some benefits without being able to use mutable objects in it's code, but what it loses it more than makes up for with it's efficiency.

Overall, Haskell does lose some ability without being able to utilize mutable objects, but it's not enough to bring down the language, in fact far from it.

This disadvantage of only being able to use immutable objects is actually what allows the functional programming language to function in the first place.

When compared with my favorite programming language, Python, Haskell seems almost backwards.

But that's really the whole point, is that by simplifying the language so such an extent, the overall complexity reduces to an extremely manageable amount.

Haskell was planned out from the start to be a functional programming language, with very little room for adjustment.

Python, on the other hand, was written to be as widely-adapted as plausible.

Unlike Haskell, Python was written and designed to be a multi paradigm programming language.

It's really left for the authors of their programs to determine the best approach to write the code.

Specifically, [IC] some of Pythons built in data types are only immutable.

This can be seen with its Boolean's, Strings, Tuples, and more.

It also allows for other data types to be mutable, sort of restricting which can and can't be mutable, but at the same time allowing for more freedom for the authors.

An example of this in action can be seen by looking at the Fibonacci Sequence, as written in Python [JP].

```
-- The Fibonacci Sequence in Python

n_terms = int(input ("How many terms the user wants to print? "))
n_1 = 0
n_2 = 1
count = 0
if n_terms <= 0:
   print ("Please enter a positive integer, the given number is not valid")
elif n_terms == 1:
   print ("The Fibonacci sequence of the numbers up to", n_terms, ": ")
   print(n_1)
else:
   print ("The fibonacci sequence of the numbers is:")
   while count < n_terms:
       print(n_1)
       nth = n_1 + n_2
      # At last, we will update values
       n_1 = n_2
       n_2 = nth
       count += 1
```

Compared with the same operation written in Haskell, this method of writing out the Fibonacci Sequence looks extremely complex at a quick view.

This is a great example of how by adding mutable objects in the program, the complexity of the program itself changes in an exponential manner.

There are many complexities taken into account in the aforementioned script, which are absent from the same implementation in Haskell.

Most notably is the implementation of the while loop.

This is an example of something that couldn't possibly exist in a functional programming language, but in an object orientated programming language there are no issues.

In fact, it's used to keep the code from becoming even more complex.

Another important factor to take into account is that while the base operations of the functions remain the same (which is to calculate the Fibonacci Sequence to a certain result), the ability to add more complexity allows for further specific analysis into the exact term of the sequence.

Finally, when comparing the two types of programming languages, there are what authors Mariana Berga and Rute Figueiredo [IC] describe as two main ways to begin planning a program.

The two main methods are Imperative and Declarative, which can be described in a similar manner to how a programmer should approach their code.

In a functional programming language like Haskell, the language itself is designed to be approached only in a declarative method.

What this means is that when declarative programming, there is a set goal that must be reached, and the program provides the easiest path to reach that goal.

In a non functional programming language, most of the time the author sets out with an imperative state of mind.

Here the programs original goal ends up relying on the order of operations it receives to modify the program.

2.7 Behind the Scenes

In the opinion of the author, the most useful external sources for learning a new programming language are visually based.

Rather than a block of text describing the issue into over detail, with more visual content the material becomes easier to comprehend.

I believe one online resources is the best at this, and that is YouTube.

While it is true that anybody can upload to the site, when someone with a comprehensive background on the material is using the platform to share their information, I believe that is the best way to share information, especially with visual learners.

Most often with online blogs and tutorials, there is only one perspective being shown, and that is what the author has typed out.

This is problematic, because often there are hundreds of unaccounted situations that occur due to only having a singular perspective shown.

By adding representations to the posts such as a video reference, or diagrams to illustrate the topic, these problems can mostly be resolved.

This is why YouTube may be the best online resource for external research.

The author of the video has a singular topic to express much in the same way of a text based article, but has the added benefits of being able to visually illustrate at a live pace from a planned perspective, on top of the benefits of natural audio.

There is a narrative perspective in the sense of what the author is trying to communicate, a visual perspective from the sense that the author can animate situations, or draw them out, or even act them out.

Additionally, there is an audio based perspective added on both, which means the author has a third way to invoke their ideas in the viewer.

Many see YouTube as a platform for binge watching, but with the right educational background, I believe it is the best platform for external online research.

However, even with as much behind the scenes work put into YouTube by thousands of engineers, it still fell pray to one of the most common mistakes made in programming languages - an integer too big in value.

On December 21st, 2012 [FB], the viral YouTube video "Gangnam Style" did the unthinkable- it surpassed 2,147,483,647 views on the platform.

That is nearly *1/4th of the entire global human population!* From a human perspective this was amazing news, but for a programming perspective it was when all hell was let loose.

2.8 Remarks

Similar to the event that caused the Ariane 5 to explode on liftoff, an integer too big in value was the issue.

Upon further investigation, the maximum count for a 32 bit integer is 2,147,483,647.

This is the exact number of maximum views possible for a YouTube video, or at least was until the company changed the platform to support 64 bit integers.

So when the latest viral video hit the platform with a running start, it not only surpassed Justin Bieber as the most viewed video on the platform, it also broke the platform completely. There were all sorts of problems that stemmed from this complication, but most notably in the public eye was the failure of the view tracker itself.

As once the two billionth, one hundred and forty seventh million, four hundred eighty third thousand, six hundred and forty-eighth person just happened to click on the video, they completely melted the expectations set out long ago by the original programmers.

As stated before it was actually a simple fix of upgrading the managing system to support up to 64 bit integers instead of 32 bit integers, but it goes to show that sometimes even the most unexpected circumstance could happen at any given time.

By upgrading and modifying the system, the bugs were worked out and everything was able to return to normal once again.

This same floating bit error has happened many times throughout history, usually leading to all sorts of catastrophic unforeseen consequences.

Some of which actually ended up affecting some real, living people.

Fifteen years ago in 2007, PayPal was still in the prime of its youth, as a continuously developing online service for transferring money.

At the time, there was nothing else so simple, services still required paper checks or cash, and the only way to do so to an individual online would require a bank transfer.

This gap in the market is what led to the eventual success of the company, but not before it faced some issues with its original programming.

Fifteen years ago [FB], there was an incident that occurred on the online money management service PayPal, which led to an innocuous but still incredibly unfortunate mistake.

An unassuming man from Pennsylvania was momentarily the richest person on the planet, when PayPal suffered from an integer float issue, similar to what happened to YouTube.

Seemingly out of the blue, Chris Reynolds became worth $92 *quadrillion.*

It was of course, by total accident.

He had unfortunately not discovered the best get rich scheme in history, and was not actually worth more than one thousand times the total combined growth domestic product of the entire planet Earth.

It was due to an automatic crediting error found within the source code for PayPal.

Just like in the case of YouTube, the success of the platform skyrocketing was not taken into account when writing the base code to define the service.

This ultimately led to the error paying out in the favor of a completely unrelated individual, valuing him at ninety two quadrillion dollars.

Luckily that was as bad as the problem became, which was not bad at all in the eyes of software programming as a whole.

The mistake only affected numbers for an online service, and fortunately did not cause an actual issue in the real world.

The crediting error was quickly noticed by PayPal, and resolved almost instantly.

The exact issue in the program operating PayPal was determined to be an issue with a 64 bit integer, just like in the case of YouTube.

The exact amount of money credited in the individuals online account was extremely significant, as it was easily recognized as a factor of the 64 bit integer.

2.9 The Remains

The major stumbling blocks in learning Haskell can be described with an example.

When peeling an apple, there is widely a lot of different ways to approach it.

One could utilize tools for mathematically similarly sized slices, and could use those tools to remove the stem and seeds.

This is an efficient way to peel an apple, but it's not the only way.

One could use almost any sort of blade, not even worry about the stem or seeds, and the job would get done in the same way, assuming it was given proper time.

When peeling an orange, though, it's usually a lot simpler of a process.

There is no need to even worry about the stem, or the seeds of the fruit, like in the case that Haskell does not have to ever worry about an object becoming mutable.

Early on I mentioned that while both a non functional programming language and a functional programming language were similar in the sense that they are both programming languages, the way they operate from the ground up is completely different from each other.

Functional programming languages like Haskell get the benefit of having a stricter guideline when it comes to the core code itself, but non functional programming languages like C#, or Python get the benefit of having the author decide how to approach their problem.

Which is better, is ultimately decided by the author of the program, not the program itself.

---

3 Programming Languages Theory

3.1 Lambda Calculus

Lambda calculus is a universal model of calculations, which operates using abstraction and application with binding and substitution.

Abstraction is the concept of creating or removing a spatial object.

Application represents the function at hand with binding passed as the argument for its domain, and substitution is the range.

Binding is used as the identifier for a name with a spatial object (E.g. data in code), while substitution is the act of replacing all instances of a value with a constant.

With this lambda calculus can be used to create a simulation of any Turing machine.

The essence of Lambda Calculus boils down to the facts that the process takes in lambda terms, and performs reduction operations on them.

Lambda calculus is entirely dependant upon mathematics, and so can be trusted absolutely. With this in mind, it makes sense lambda calculus must follow three very important rules[LC].

The Rules of Lambda Calculus:

Syntax	Name	Brief Description
x	Variable	A character or string representing a parameter or a mathematical/ logical value.
(λx.M)	Abstraction	Function definition (M being a known lambda term). The variable x becomes bound in the expression.
(M N)	Application	Applying a function to an argument. M and N are lambda terms.

Breaking down these steps, it's easy to see why Lambda Calculus is a great fit for a Turing machine - due to how simple the language is written.

A Turing machine must follow strict mathematics, and by using Lambda Calculus it can take full advantage of its resources.

In the first rule, the language only takes in x as a simple string or character. This is just about as basic as it gets, and from there the rules build upon one another.

Since Lambda Calculus is a high level programming language, the program only relies on using lambda calculus, and does not rely on any other mathematical dependencies.

With no distractions in the language, it provides a clear path to implement a Turing Machine, and so is called a Turing Complete language.

Another way to explain why this is considered Turing Complete is due to the fact that recursive functions are the bread and butter of the program.

They are so fundamentally ingrained as the main tools, that one would have to go out of their way to write a Turing Machine that doesn't use recursion.

It is for these two main reasons that Lambda Calculus is considered Turing Complete, and can help us identify one of the main overarching themes of the theory of programming languages.

When it comes to a purely functional programming language, one of the main goals of efficiency state that reusing code with planned recursion can lead to the fastest run time.

When solely focusing on the run time of applications, better results are returned when recursion is implemented, and the complexity of the system is set to a minimum.

3.2 Recursive Computing

This is a popular online cartoon, depicting different data structures as a solution to a problem (here, a dish has caught on fire, and the data structures serve as liquids)[PM].

If this were a realistic scenario, there would be several options, notably water that can quickly and efficiently put out the fire.

However, some solutions like in the case of gasoline actually end up adding fuel to the fire.

In the cartoon depicted above, various Functional Data Structures, C Foreign Function Interface, and Profiling are depicted as either sufficient, or exactly made for these circumstances to solve the problem.

Foldl', though, is compared to a gasoline substance, because when it comes to the run time of an application in a functional programming language, it's not the best choice available (nor does anyone actually fight fire with fire, and walk away unscathed).

Foldl' refers to the group of functions whose goal is to analyze a recursive data structure, and by using a known combination algorithm, can recombine the recursively ordered different parts into a single value.

Foldl and Foldr are almost the same operation, however foldl associates to the left, while foldr associates to the right[HF].

```
-- Foldl vs Foldr

foldl (+) 0 [1, 2, 3]
    (((0+1)+2)+3)

foldr (+) 0 [1, 2, 3]
    (1+(2+(3+0)))
```

Even though both are given the same parameters, they do in fact return different results.

From a basic programming perspective, this choice may seem arbitrary.

However, if the program is complex enough and important in the real world, the run time can be greatly diminished if the wrong choice was made here.

3.3 Reversible Computing

The idea of reversible computing is to break everything down into logic gates - in the forms of OR, NOR, NOT, and more.

Using theses logic gates, every single piece of data one could need can be written and passed through the gates in binary form.

As an example of a logic gate in action, here is a reference chart for a 2 input AND gate. [RC].

2 Input AND Gate:

Input	Input	Output
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Another name for this type of gate is a truth table, and it must follow the rules that are laid out for it.

In order to successfully pass a '1' as the output, it relies on the truth of the AND operation that guides the process.

In this case, there is only one scenario where the output can be a '1'. That is when both requirements of the AND gate have been met, not just one.

If just one input A or B is a '1', the AND condition has not been met, and so the output would remain a '0'.

It is easy to figure out the output from two inputs, but what if we looked at the same logic gates in reverse?

Taking the above 2 Input AND Gate as an example, if you know the output is a '1', you know what both the inputs must be, as there is only one option.

However, if you know the output is a '0', there is no way to determine with just this information what both of the inputs are.

There are simply too many options that A and B could fulfill, and all lead to the same result.

With this in mind, this is an example of irreversible computing, the non-helpful version of reversible computing.

To determine the difference between the two, this NOT gate will represent a different type of logic gate, so the two can be compared.

Input	Output
A	B
0	1
1	0

The key difference between these two logic gates is the way they setup the outputs to depend on the inputs.

In the case of this NOT gate, taking a reversible stance is quite easy compared to the AND gate.

If you know the output B = '1', then there is only one choice for the input A.

Similarly, if you know the output B = '0', there is only one choice for the input, which is different than if B = '1'.

This is in essence the core of reversible computing, the ability to be able to work backwards in the calculations.

Reversible Computing has one final, but major difference that sets it apart from most other computational methods.

It's actually due to the dissipated heat due to the thermodynamics of the system.

According to [RC], it's theoretically possible to construct a computer that is solely built with reversible gates.

The coolest part about this (pun intended) is that in theory, the computer will be so efficient that it won't dissipate any heat (heat generated from non-reversible computing).

This would save energy drawn from the Power Supply Unit, due to a lower temperature inside as less cooling is needed.

Similarly, the Central Processing Unit and the Graphics Processing Unit (if available) would both be running at lower temperatures than 'normal'.

One of the biggest problems with an over-powerful computer is the heat generated from all of the processing power.

It is for this reason a lot of companies have 'server rooms', where the goal is to keep the hardware running at a cool temperature.

By reducing a lot of the wasted heat from computers, a new standard of computer efficiency could be brought to the world, forever changing the course of computer architecture.

In addition this would save electricity which is constantly powering the computer, leading to a lower energy bill for all.

Furthermore with cooler components, the hardware itself should in theory be preserved better than if stored at a high temperature.

And last but not least, if a truly only reversible computer was constructed, it would probably be able to run much faster, with less hang time between programs.

3.4 The Halting Problem

To break it down, the Halting Problem literally is about a computer program halting, or not halting.

The idea of a halting machine is that a basic computer program is written to determine if a computer program will halt or not.

The goal is to be able to solve if first order logic is decidable or not.

This is the same way of asking if we can automatically test if the premises lead to the conclusion.

[HP] There is some form of an input that is able to pass on the information about its input and its source, but here's where it gets tricky.

The computer program passes its information onto the original function, and then instead of following the rules it performs the opposite action that is expected from what the function thinks the input will do.

Basically, the halting problem expects a proof of the impossible, of every single possible computer program

The Halting Problem uses a Turing Machine in the form of both a computer and program, and when presented with the case as described above, it becomes undecidable, because it is proven to be unsolvable.

This is significant, because it provides proof to a side of computer science where a class of applications will never be perfectly defined.

To introduce this subject to a first time learner, I would have them compare it to the effects of an endless while loop, so they can see the same type of result in a simplified manner.

As an example of the problem, one can visualize an input/ output machine, titled A. Machine A can only take in inputs, and respond with 0 for False, or 1 for True.

This program will eventually halt, which is the goal of almost all modern programming.

If the program does not halt, it must mean it's caught in an infinite loop.

Alan Turing was one of the first to tackle this issue, concluding with a proof that shows it's impossible for any such program to be able to solve.

To explain this, he describes an arbitrary machine called B, which is able to solve the halting problem.

It takes in an input in the form of a machine running software, and in turn outputs a 0 or a 1 depending on if the program halts, or not.

By modifying B and staying within the confines of the problem, the paradox is closely examined at all points.

If Machine B outputs a 1, it will then be told to loop forever.

But if Machine B outputs a 0, it will then be told to halt. This Machine is now called Machine C.

And so if Machine C were able to take Machine C as a form of input, it would lead to a paradox.

If the program does eventually halt, Machine C will then not halt and be stuck in an infinite loop.

And by the same logic if the program does not eventually halt, the program will then halt.

This is a classic example of a paradox in action, and infinite loops more often than not are problematic at their core.

Usually when a programmer is met with an infinite loop, it's due to poor coding practices where not every complexity was taken into account.

With the Halting Problem, when one wants to see if a program will halt or not its impossible to take in every pair of inputs.

To get around this type of issue, modern day programmers usually try to write subroutines which are both guaranteed to halt at some point, and/ or to specifically halt before a prefixed time.

Since the Halting Problem can not be solved practically, it is considered by many to be undecidable.

3.5 The Knuth-Bendix Algorithm

The Knuth-Bendix algorithm is a complex set of systematic formulas representing transformation of a set of equations to a confluent term rewriting program [KB].

When the algorithm is able to successfully run, it will solve the word problem for the algebra needed.

In this case, the word problem can be arbitrarily defined as the problem of choosing between two known expressions, and determining if the two are equivalent or not.

It does this with a complex series of equations that allow a path from any point of the problem to the solution.

Equations can coexist in a rewritten system, however when programming sometimes only one option is acceptable.

The act of transforming an equation can be seen in this example:

Initial Equation: $0 + x = x$
Rewritten System: $0 + x \rightarrow x$

The word problem for the algebra tries to determine if two different expressions represent the same thing.

An example can be seen by examining this formula:

$$-((-y + y) + (x + -x)) = x + (-(y + x) + y)$$

They may not appear identical to either a human or a computer at first glance, but after some investigating the two terms are equivalent to each other.

The Knuth-Bendix algorithm follows four central axioms:

(1) 0 + x = x
(2) x + 0 = x
(3) -x + x = 0
(4) (x + y) + z = x+ (y + z)

Using just these four axioms, anything can be solved.

As an example, to prove that - -x = x for any x, only five steps are needed.

To begin, Axiom (2) states - -x = - -x + 0.

Axiom (3) states - -x = - -x + (-x + x)

By Axiom (4) - -x = (- -x + -x) + x

Axiom (3) again proves - -x = 0 + x

Concluding by Axiom (1) - -x = x

Using this systematic way of reasoning, the algorithm can solve most word problems for the algebra needed.

3.6 Hoare Logic

Hoare Logic is the name of a system of rules whose end goal is to correct computer programs.

Hoare Logic is build from the ground up upon a few different sets of rules, with the main feature revolving around the Hoare triple.

In its most basic form, the Hoare triple looks like this [HL]:

```
-- Hoare Triple

{P}C{Q}
```

In this form, 'P' is the precondition assertion, while 'Q' is the post-condition assertion.

It's important to note the post-condition is only established after the precondition has been met.

Finally, 'C' in the triple above has the role of the command, the actual action being taken.

Since Hoare logic can be divided into axioms and rules, all the constructs of an imperative programming language can be fulfilled, including pointers, concurrency, jumps, etc.

But it's extremely important to note that with only basic Hoare Logic, termination can not be proven!

C will not terminate if P holds the state before running C that Q is to hold.

If one wants to achieve total correctness, it can be proven by adding onto the While rule.

---

4 Project

4.1 The Exterior

As a Computer Scientist major who's minoring in Game Development, I wanted to take the opportunity of implementing a project in Haskell to combine both of my two leading traits.

I spent a long time researching different games people have created through Haskell, but unfortunately most of them ended up being extremely complex, and not well suited as a short project.

After a while, I found a template on a code review website that builds a basic model of the game Rock Paper Scissors, but it wasn't written in the most efficient manner.

I spent some time debugging the issues that were pointed out with the code, as well as took an in-depth exploration into how the software is able to perform its expected target [SF].

```
-- Rock Paper Scissors in Haskell

-- Instructions to play: Enter your move (Rock, Paper, or Scissors) as standard input when
    executing the game.

import System.IO
import System.Random
import Control.Monad
import Control.Arrow

data Throw = Rock | Paper | Scissors
  deriving (Show, Read, Enum, Bounded)

instance Random Throw where
  random = randomR (minBound, maxBound)

  randomR (a, b) g = first toEnum $ randomR (a', b') g
    where a' = fromEnum a
          b' = fromEnum b

beats :: Throw -> Throw -> Bool
Rock `beats` Scissors = True
Paper `beats` Rock = True
Scissors `beats` Paper = True
_ `beats` _ = False

play :: Throw -> Throw -> String
play p1 p2
  | p1 `beats` p2 = "You win!"
  | p2 `beats` p1 = "You lose."
  | otherwise   = "No winner this time! It's a Tie."
main :: IO ()
main = do
  putStr "Please enter a move: Rock, Paper, or Scissors? \n"
  hFlush stdout
  p1 <- getLine >>= readIO
  p2 <- liftM (fst . random) getStdGen
```

```
  putStrLn $ "Game on! " ++ show p1 ++ " vs. " ++ show p2
  putStrLn $ play p1 p2
```

There are some basic flaws with this setup, but the game can be trusted to play as expected each time, which is the most important aspect.

4.2 The Interior

The game rock paper scissors is basic in its logic, as there are only three moves that can be selected as a valid move.

The player is given a choice of what move to make, but in theory can type anything they want.

With this in mind, the game is setup so that only three options will allow for the game to advance.

The player must enter a move of either Rock, Paper, or Scissors.

The game starts here in the main function, where the player begins to setup their attack.

This is done by prompting user input from the player in the form of a string that requests a move is chosen.

Finally, this value is stored to be used later.

If the player enters any combination of characters besides these three moves, the program will not advance.

One of the main issues with the way the code is setup is that since the 3 main moves are constructors, they must use a capital character to begin.

This is why Rock Paper and Scissors all must have a capital letter when playing the game.

Of course if one wanted to take capitalization into account, there are multiple methods to resolve this type of conflict.

The easiest way would be to compare the values of what the player enters with the known move set, and match the case to fit the notion needed for the code to compile.

After the player selects a move, then the game can begin.

The players move is saved under the name p1 for later use, and the computers AI who randomly selects one of the three moves is stored under the name p2.

Each of the possible moves in the game is then stored for later use as a Throw.

Each Throw then is compared with the boolean value of the opponents Throw, so the game can determine whether or not the two values are identical.

When the Throws are compared with each other and they can be aligned with the chart of what beats what, then the form of the game starts to take place.

Each of the four possibilities that a player could take are set up initially.

To begin, if Rock and Scissors are selected, Rock will always beat Scissors.

If Paper and Rock are chosen, then Rock will always lose to Paper.

Third, if Scissors and Paper are the two chosen, then Scissors will always beat Paper.

Finally, when the throws are compared as equal with each other, the last test case of NULL beats NULL is selected.

The most important part of the way these rules are setup is that there is no room for deviation.

For example, there is no way that Paper can lose to Rock, or no way that rock can lose to scissors.

After the player makes a decision on what move to use, the game will show the player what the battle is expected to look like.

In this situation it's not the most advanced, as the game will display to the user their choice vs the computers AI choice.

Next in the games run time, it must determine what to do with the winner selected.

For this case, a simple statement to check whether or not p1 beats p2 can be utilized.

If p1 pulls ahead and beats p2, then the game can simply print the player won, and end.

However is p2 manages to outplay p1, the game will let the user know they've lost.

Finally, if the case of no winners is undertaken, like in the case of the AI selecting the same choice as the player, then the game will note to the player that they have tied.

This at it's essence is the core game-play loop, allowing for random chances in a battle that is always predictable in its calculations.

Overall the game does run as expected, but only with the assumption that the players move is entered initially.

The problem is that there exists a slight chance of a new player running the game for the first time, without entering their move initially.

If they had not read the instructions to play, the game wouldn't play as they expected.

In practical use this does not make a difference, as from a human perspective it's just a matter of waiting to prompt user input.

When it comes to the run-time of the program, the change is so minute that from a realistic standpoint it would not change anything obvious.

---

5 Conclusions

Haskell is a universal programming language that operates on the principles of mathematics to operate as one of the most efficient forms of software.

In theory, anything can be programmed using Haskell, but sometimes it may be more convoluted to do so.

Because Haskell is such a reliable programming language, it can be seen as predictable, which really is one of the goals of a well written programming language.

Due to Haskell's limits when it comes to types, from the surface the language as a whole may seem very bland, or basic.

But this is the key defining feature that truly sets Haskell apart from other languages - it's predictability.

When the outcome can be pre-planned from the start, with no variation other than what is instructed, the run time of a script can be incredibly optimized.

With other programming languages, variables can have all sorts of unexpected complexities, that years later can be the source of a massive problem.

Anything from a satellite falling out of orbit, to bank fraud or identity theft.

With no underlying complexities that could potentially break everything, Haskell pulls ahead here.

Since there is no room for failure in Haskell, at least when compared with other languages in a basic sense, this becomes one of the best software choices for a developer to use.

By keeping the programming language functional at its lowest possible form, developers can cut out anything that slows down the code, or could lead to an error, or could be exploited by a hacker.

Removing complexity from the system may make some actions more difficult, but with no room for error this is sometimes the preferred route.

Apples and oranges are both delicious choices of fruit.

They grow unlike each other, act different from one another, and taste different as a whole.

Yes they are two completely different fruit, but the key is that they are in fact both fruit.

Much alike a functional programming language, and a non functional programming language, both have their advantages and disadvantages.

When it comes down to actually using one though, unless the situation is particularly specific they can both get the same job done.

In the end, it really depends on the use case of the program for developers to determine the most efficient way to tackle their goal.

References

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

[FB] 10 Famous Bugs in The Computer Science World, Geeks for Geeks, 2019.

[IC] Functional Programming Vs OOP: Comparing Paradigms, Imaginary Cloud, 2021.

[LZ] Pure Functions, Laziness, I/O, and Monads, School of Haskell, 2014.

[MD] Monads, Haskell Wiki, 2021.

[AK] Recursion over algebraic data types in Haskell: Examples, Alexander Kurz, 2020.

[JP] Python Program to Print the Fibonacci sequence, Java T Point, 2021.

[PM] Haskell Performance Measurements, Bohdan Liesnikov, 2018.

[HF] Higher Order Functions Data Types, Godfrey Cummings, 2016.

[LC] Lambda Calculus, Wikipedia, 2021.

[HP] Turing Machine Halting Problem, Tutorials Point, 2021.

[RC] What is Reversible Computing? Medium, 2020.

[HL] Hoare Logic, Wikipedia, 2021.

[KB] Knuth-Bendix Completion Part 1: Introduction, Jin Xing Lim, 2020.

[SF] Rock Paper Scissors in Haskell, Simon Forsberg, 2014.