CPSC-354 Report

Erik Hombledal
Chapman University

December 22, 2021

Abstract

This report is divided into 3 parts. The first section is investigating the differences in computation time between Haskell and Python, and discussing the differences is language typings that account for this difference. In the second part, I investigate the concept of undecidability in programming and computation using Turing Machines, and the impact this has on mathematics and programming as a whole.Finally, in Part 3, I continue my investigation into the computation time of different programming languages, by adding additional languages and discussing the differences in execution time.

Contents

1 Introduction **2**

2 What Makes Python and Haskell Different? **3**

2.1 Introductions . . . 3
2.2 Functional vs Imperative Coding . . . 3
2.3 Lazy vs. Strict Evaluation . . . 4
2.4 The Many Types of Typings . . . 4
2.4.1 Strong vs. Weak Typing . . . 4
2.4.2 Static vs Dynamic Typing . . . 5
2.4.3 Explicit vs Implicit Typing . . . 5
2.5 Interpreted vs. Compiled Language . . . 6
2.6 Performing the Computational Comparison . . . 7
2.6.1 Results Analysis . . . 7
2.7 How Do We Apply This in the Real World? . . . 7
2.7.1 Calculation Speed . . . 7
2.7.2 Learning Complexity . . . 8
2.7.3 Overall Support . . . 8
2.8 Conclusion . . . 8

3 Undecidability and the Halting Problem **8**
3.1 Introductions . . . 8
3.2 The Turing Machine . . . 8
3.2.1 Turing Machine Basics . . . 9
3.2.2 Machine States . . . 9
3.2.3 Turing Machines and the Basis of Programming . . . 10
3.3 Undecidability in Programming . . . 11
3.4 The Halting Problem, Explained . . . 11
3.4.1 Why is the Halting Problem Undecidable? . . . 12
3.4.2 Why This Matters . . . 12
3.4.3 Other Undecidable Problems . . . 12

4 Execution Time Experiment **13**
4.1 Introduction . . . 13
4.2 Recap of Results from Part 1 . . . 13
4.3 Additional Languages . . . 14
4.3.1 Python, Done Properly . . . 14
4.3.2 Java . . . 15
4.3.3 C . . . 15
4.3.4 C++ . . . 16
4.4 Results Comparison . . . 16
4.5 Potential Future Algorithms to Test . . . 17
4.6 Testing Conclusions . . . 17

5 Overall Conclusions **17**

6 Sources **18**

1 Introduction

In this paper, I hope to investigate areas of research in Programming Languages that interest me, and that I wish to learn more about. Firstly, I wish to learn about what makes Haskell unique. We used the language all semester, but I wanted to have a more firm understanding of what made Haskell tick under the hood. To learn more about it, I compared it to Python, a language I was far more familiar with, to see what the differences were, and how that impacted execution time. For part 2 of the report, I have always been interested in some theoretical aspects of computation and logic, despite not knowing very much about them. So, I decided to learn about Turing Machines, and how those are used to explain logical phenomena like undecidabilty. Finally, I wanted to expand upon my work in Part 1, and see if comparing other languages to Haskell and Python revealed anything interesting about what each language is used for, and why.

2 What Makes Python and Haskell Different?

2.1 Introductions

In this section of the report, I will be investigating the computational time differences between Haskell and Python, and why these differences exist, despite both languages being written in C. To test this, I will be using large, identical list operations in both Python and Haskell, and then comparing the execution time between the two. To enable this, I will be discussing the differences between the two languages.

2.2 Functional vs Imperative Coding

It would not be fair to start a report on Haskell and Python without discussing the obvious difference between the two - Python being an imperative language, and Haskell being a functional one. Most programmers start out their careers with iterative languages, starting in either Python, or C++/C. Because of this, learning a language such as Haskell is the equivalent of turning the world on its head, as precious little you learned in iterative languages will apply conceptually.

As a refresher, iterative languages work by giving a computer a sequence of tasks, and the computer then performing them in order (or *iteratively.*) You declare a variable, you ask the computer to do some calculations to that variable, and then you have it return the result, which is now changed from the original variable you gave it. Easy, clean, and simple to wrap your head around.

Functional Programming Does Not Work This Way.

Functional programming is based on is the idea that you declare concepts, and then you logically build on top of what you have previously stated to develop more complex computations(Part 1 - Source 1). To enable this, functional programming is absolute. If you tell Haskell that int myNumber = 5, you can not later state that myNumber = 7. Functional programming forbids this to ensure that functions will *always* do exactly what you tell them to. This might seem unnecessarily rigid, but is incredibly powerful later on down the road. For example...

```
a = 5
b = 7
SumValues a b = a + b;
--This will always return 12. You can't say later on that a = 8, making a+b not equal 12.
--Following this logic, you can not redefine Sumvalues to be something else. Sumvalues will
    always be a + b.
```

So, what do we now do with this simple function? Well, we use it to build more complex functions, using it as a building block. Because we ensured that functions can not be modified, we guarantee that our foundation is solid and unchanging. Continuing our previous example...

```
-- lets say we wanted to get the square of a sum. We can use our previously declared
    function to help us!
SumValues a b = a + b;
SquareSum a b = (SumValues a b) * (SumValues a b)
```

```
    --Squaresum 2 3 would return a value of 25!
```

This is very powerful, and allows you to build incredibly complex programs with very little repetition in your code. Being able to build functions on top of each other saves you from having to retread the same ground you would in an iterative program, as you have already done the work. Simply call your previous functions!

Now that we have finished with the obvious difference, we can go now and discuss some of the unique traits of Haskell, and see how they compare to Python.

2.3 Lazy vs. Strict Evaluation

When reading into the advantages of Haskell over other languages, you quickly realize that one of the major selling points is the ability for Haskell to be "lazy." What this means is that a lazy language will *not* run calculations unless they are needed somewhere else(Part 1 - Source 2). For example, if we have a list of integers:

```
xs = [1, 3, 5, 7, 9]
--and then ran the function double() which doubles the integers in our list...
doublexs = double(xs);
--nothing would happen computationally until you call the result!
print(doublexs[2]);
--this line would only call double() on the second index of xs, and that is it!
```

As you can imagine, this can save a large amount of computation time in larger programs, or when dealing with larger lists. However, in this particular case, Haskell is not unique. Python3 actually has the exact same functionality, performed in almost the exact same way. So, there is no real computational difference between Python and Haskell, at least in this specific category. However, as this is one of the main concepts that allows Haskell its speed compared to most other languages such as C, it wouldn't feel correct to not discuss it at least partially.

2.4 The Many Types of Typings

When it comes to comparing the typing between Haskell and Python, it is important to establish what exactly we mean by "typing." There are three major categories that you can compare a language's typing on:

2.4.1 Strong vs. Weak Typing

Firstly, you have whether the language is strongly typed or weakly typed. In a weakly typed language, variables can be implicitly converted into unrelated types, whereas in a strongly typed language, you would require an explicit conversion to be performed. (Part 1 - Source 3). Giving a quick example:

```
    --In a weakly typed language, this code snippet would work fine. In a strongly typed language,
        however, you would need to cast either the int or the string to the other type.
```

```
a = 5
b = "5"
c = concat(a, b)
d = add(a, b)
```

Both Haskell and Python are strongly typed, meaning that one type can not be implicitly converted into another. Because of this, both languages require casting to be performed if you are attempting to combine two types. The only exception to this is combining integers and floats, which are handled as a special case in each language for simplicity and convenience, despite being different types.

Generally, languages that are strongly typed are faster and safer, as the compiler does not have to spend time guessing what type you mean when you are attempting to combine two different categories of things. A strongly typed language prevents this situation from even occurring in the first place, and throws an error if you attempt to compile with this type uncertainty present.

2.4.2 Static vs Dynamic Typing

Next, we have the comparison between being statically typed and dynamically typed. Statically typed languages bind variables to both an object and a type, whereas dynamically typed languages only bind them to a specific object (Part 1 - Source 3). In practice, a dynamic language allows you to have a variable be equal to a string in one instance, then equal to an integer in another. As well, it means you can throw items of different types into the same list without issue. In a statically typed language, however, once you establish a variable, it is associated to the specific type you started with and can not be changed.

```
--In a dynamic typed language, this code would work, and you would add the string to the
    array. In a static language, it would throw an error.
list[] = [1, 3, 5, 7, 9]
list[1] = "Hello!"
print(array[])
--1, Hello!, 5, 7, 9
```

As can be expected, Haskell is a statically typed language. Being able to change the typing of a variable during runtime goes against the core tenets of functional programming, and would defeat the purpose of not allowing any changes to your foundation. However, Python is dynamically typed, and does allow these kinds of operations.

Because of this difference between Haskell and Python, there is some extra work the interpreter in Python has to do to to update the types of each variable and object as they potentially change, leading to additional computation time required.

2.4.3 Explicit vs Implicit Typing

Lastly, there is the difference between explicit and implicit typing. In explicit typing, you must declare every variable as having its own type, whereas in an implicit language, the interpreter/compiler assumes what the type is based on the operations you perform on it. As an example:

```
--in an implicit language, this would work fine and c would be implicitly declared an integer.
    In an explicit language, it would be an undeclared type error on c.
a = 5;
b = 9;
c = a + b;
```

In a very interesting turn of events, both Python *and* Haskell are implicitly typed. However, in Haskell's case, this is not a bad thing. As long as the variable's type is consistent after it is implicitly declared, Haskell would have no issue understanding what to do with that variable moving forward. So even though it might be a cause for concern on paper, in practice it is actually just as safe as declaring the type yourself.

2.5 Interpreted vs. Compiled Language

Last, but certainly not least, is discussing the difference between how Haskell and Python are interpreted and compiled. To start, it is important to clarify the difference between an interpreted language versus a compiled one, and what that actually means conceptually. When you attempt to run code on your computer, your machine needs to transform these written, human instructions into something the machine can understand and perform calculations on. In computing, there are two major ways to handle this - either through a compiled language or an interpreted language solution.

In a compiled language, the computer directly converts your code into instructions your machine can understand. However, these instructions are made explicitly for your computer, down to available memory, CPU speeds, RAM speed, and so on. Because of this, the code has to be recompiled on each new machine it needs to run on, as the specs of each machine will be different. As well, this method means that any change made to the code requires you to rebuild your executable. However, the payoff for this is faster computation speeds, as well as the ability to have some control over hardware operations such as memory allocation.

On the other hand, interpreted languages convert your human code into machine code through the use of an interpreter. This interpreter converts your code line by line as it runs your program. This is referred to as "just in time compilation." However, this additional overhead step of creating the machine instructions just before you need them does lead to a lot of increased computational overhead and inefficiency, and can slow down your program significantly (Part 1 - Source 9). The advantage, however, is that the interpreter will work regardless of what specs your current machine has, so code can be easily transferred and run on different machines without the need to compile. Historically, this was the major selling point of interpreted languages like Java. As well, no need to compile can save a lot of time while testing and debugging a program, as you do not need to waste computation time on recompiling the set of instructions each time you make a change.

Returning to our languages, we can see where the primary difference in speed comes from between Haskell and Python. Haskell, being a compiled language, is far faster at executing instructions than Python, an interpreted language. However, Python is able to run on machines without the need to compile, and is generally easier to develop on because of this.

2.6 Performing the Computational Comparison

To compare the computation speed between Haskell and Python, I created a program that creates 500,000 random integers in a list between 0 and 9, adds 1 to each of them, then sums the resulting list. I did the +1 and summation of both lists to get around the "laziness" of Haskell and Python, and to ensure both lists were properly generated.

In Python, I was able to create my program in Replit without issue, but I was unable to in Haskell because Replit does not contain some necessary libraries for Random. So, the Python program will be in Replit, and the Haskell program will be on Github.

Here are both of the programs included below, as well as a snippet of the Python program to see it conceptually.

Python Code - Replit (Part 1 - Source 4) (Part 1 - Source 5)

Haskell Code - Github (Part 1 - Source 6) (Part 1 - Source 7)

```
randomlist = []
for i in range (0, 500000): --for loop
  n = random.randint(0,9) -- generates randoms
  randomlist.append(n) --appends to list

for i in range (0, 500000):
  randomlist[i] = randomlist[i] + 1; --adds 1 to each

print(sum(randomlist)) -- sums list
```

2.6.1 Results Analysis

Upon running both of the programs, you can see the stark difference in execution time between Python and Haskell, even while controlling for laziness. After running the programs 5 times each, the Python program averaged around 2.6 seconds, whereas the Haskell program averaged 0.37 seconds. Despite performing identical list operations, the Haskell calculation was almost 5x faster. While some of this could be chalked up to using Replit to handle the Python calculations and the additional overhead, the time difference would not be anywhere near the magnitude we see in the results.

2.7 How Do We Apply This in the Real World?

2.7.1 Calculation Speed

When comparing Haskell and Python, the calculation speed difference is quite staggering, even with such a computationally simple exercise. If you were to run a database in Python, handling potentially millions of lookups at once, it is hard to see how an interpreted language would even be functional in this circumstance. With a language like Haskell, however, it becomes much easier to handle large amounts of calculations quickly. As well, Haskell would have applications in real-time systems, where the added security and safety from functional programming languages, as well as the faster calculation speeds, would be incredibly beneficial.

2.7.2 Learning Complexity

In general, Haskell and functional programming as a whole are a lot more difficult for the average programmer to understand, with the vast majority of programmers working in iterative languages. Because of this, many people shy away from the advantages that functional programming could provide due to the steep learning curve. Personally, I understand the fear. If you do not have a solid background in mathematics, functional programming will appear far too mathematical for most people to even want to attempt.

2.7.3 Overall Support

Python is far, far more supported in the real world than Haskell. With the myriad libraries available to Python, it is quickly becoming the most used programming language in the world. Haskell, on the other hand, is a far more specialized tool. Haskell, and functional programming in general, just simply do support in order for them to become more widespread and supported in industry, and will most likely remain niche compared to C,C++, and Python.

2.8 Conclusion

Overall, this deep dive into Haskell and Python was informative, both about the two featured languages, and about programming languages as a whole. Realizing how many different ways there are to categorize languages, what they mean, and how they can influence how we code was very interesting to me. As well, it has also given me a deeper appreciation for functional programming, and the computational and organizational benefits it can provide. For my next section of the paper, I hope to cover the halting problem. As well, I will be expanding on the computational differences experiment further down in Part 3 by adding more languages to the comparison.

3 Undecidability and the Halting Problem

3.1 Introductions

In this section of the paper, I will be primarily discussing the computing concepts of Undecidability and the Halting Problem, and how they relate to programming languages as a whole. To do this, I will be using and explaining the concept of a Turing Machine, how it is the basis of all modern computing today, and why that makes all computers suffer from undecidability and the Halting Problem.

3.2 The Turing Machine

To begin, it is impossible to explain undecidabilty and the halting problem without a proper frame of reference. To help establish this, I will be explaining Turing machines, how they function, and why they are critical to the understanding of these above problems.

3.2.1 Turing Machine Basics

Turing Machines, as a concept, are relatively simple(Part 2 - Source 1). Developed by mathematician Allen Turing in 1936, they were designed to be a logical way to investigate the limitations of what can be computed. The basic idea is this - you have a machine (or a person, in Turing's original concept) that is fed an infinitely long roll of tape with symbols on it. The machine is capable of 3 operations, done in the following order:

- Read the symbol currently under the head
- edit(either replace with another symbol or remove) the symbol that is currently under the head
- move the tape left or right

Now, using this, we are able to form basic programs and calculations. In a basic example, let us assume we have a tape that has [0 0 1] written on it currently, with the head above the final [1]. If we provide instructions that the Turing Machine is to follow based on the symbols it reads, we are able to modify our initial values as we wish. Let us assume we give our Turing Machine the following instructions...

Symbol Read	Write Instruction	Move Instruction
Blank	None	None
0	Write 1	Move Tape Right
1	Write 0	Move Tape Right

So, what happens to our initial value of [0 0 1] if we follow the above instructions? Well, if we think it out logically, we can see what happens. Firstly, we start on our ending 1. The machine reads the 1, writes a 0, and then moves the tape to the right by one square, as was instructed. Currently, we have [1 0 1], with the head now above the middle 0. Next, we read that 0, replace it with a 1 and move the tape right, leaving us with [1 1 1]. Lastly, we reach the final digit on the left. As before, we read the value, modify it according to our instructions, and move the tape right. After this, we end up with [0 1 1], the bit inverted form of the initial value we gave! Despite the relative simplicity of the program we just devised, it is clear to see how this concept can be expanded on further, and the potential power that it holds in computing.

However, our above program is flawed. Once we finish reading our string of numbers, we reach a blank space. Based on our above instructions, we do nothing once we reach a blank. This means that our program is infinitely running, as it never receives an instruction to stop reading. So, how do we fix this problem? Simple - through the introduction of *machine states*.

3.2.2 Machine States

Machine states solve the problem we mentioned above by allowing symbols to mean different things based on the state condition of the machine (Part 2 - Source 1). For example, a blank space can be either do nothing, as in the original state of the program, or can be a termination order in a stop state. So, we can update our chart from above to reflect these changes.

Symbol Read	Write Instruction	Move Instruction	*Next State*
Blank	None	None	*Stop State*
0	Write 1	Move Tape Right	*State 0*
1	Write 0	Move Tape Right	*State 0*

So now, when we read a 0 or a 1, we set the machine's next state to *state 0*, which is the default run state of our machine. However, when we land on a blank space, we now change the machine state to be the stop state, which stops our program.

Following this logic, we can use machine states for more than simply stopping our program. For example, let's say we want to re-invert the bits we inverted in our above example, and return them back to normal using our same machine, during the same calculation. How would we accomplish this? Through machine states!

Thinking logically, we can imagine what we would need our machine to do. Firstly, we would need our machine to invert the bits as in the above example, always moving the tape to the right and inverting until we reach a blank. Once we do reach a blank, we change the machine state to *state 1*, and begin working backwards, re-inverting the bits and moving the tape to the left. We then enter the stop state when we reach a blank space within *state 1*. This logic can be seen in an expanded chart from above.

Machine State	Symbol Read	Write Instruction	Move Instruction	Next State
State 0	Blank	None	Move Tape Left	State 1
	0	Write 1	Move Tape Right	State 1
	1	Write 0	Move Tape Right	State 0
State 1	Blank	None	Move Tape Right	Stop State
	0	Write 1	Move Tape Left	State 1
	1	Write 0	Move Tape Left	State 1

So now, whenever we reach a blank while in state 0, we move the tape left, transition to state 1, and then reverse the process. While this exercise might not be incredibly *useful*, as we are simply changing bits back and forth, this concept of states is the basis of all programming languages. Through the introduction of more states and more symbols, we are capable of simulating *any possible mathematical calculation or program.* (Part 2 - Source 2).

3.2.3 Turing Machines and the Basis of Programming

As was mentioned previously, Turing Machines are capable of calculating any solvable mathematical calculation or simulating any computer program, as long as they have sufficient symbols and states (AKA infinite memory). We deem any programming language or computer capable of computing every Turing-Computable function as "Turing-complete." (Part 2 - Source 3). As well, a Turing-complete machine must be able to be simulated by a universal Turing Machine (A Turing Machine capable of simulating other Turing Machines). However, as current computers do not have infinite runtime or infinite memory, they are not truly Turing-complete according to the mathematical definition. Despite this, they are considered close enough

for any practical computing purpose, as they would be considered Turing-complete if they did have infinite resources.

This concept of Turing-completeness forms the basis for all modern programming languages and computing. As long as your language is Turing-Complete, it is able to simulate all solvable mathematical functions, and is able to be translated into any other Turing-complete language (Part 2 - Source 4). Almost all languages currently in use today, from iterative languages like C/C++ and Python, to pure functional languages like Haskell, are Turing complete. This means that almost all computer programming languages are universal, and able to be translated from language to language without issue. The only true difference between any Turing-complete language is syntax and implementation.

Now that we understand the basics of Turing machines and what that means for programming and computation, we are now able to further expand upon the concept of undecidability, and what that truly means for mathematics.

3.3 Undecidability in Programming

The concept of undecidability in computing is quite simple. If we are able to construct a Turing machine which will halt in a finite amount of time for any possible input and be given a definitive "yes" or "no" answer, that problem is decidable. If we are not able to do so, it is considered undecidable. These problems of undecidability have profound implications for not just programming, but mathematics as a whole.

Back when Turing first created his concept of the Turing Machine in his 1937 paper, he was developing them to solve a specific, unanswered question in mathematics. This problem, called the "Entscheidungsproblem" (decision problem in English), was the idea that all of mathematics should be solvable, and reducible to base algorithms and proofs. Once these "truths" were successfully reduced, all of mathematics could be built upon these base algorithms, and all of mathematics would be formally definable (Part 2 - Source 2). However, Turing showed that this is impossible, and that there are some mathematical functions that are unknowable. Even knowing all of the preconditions, as well as all possible states the Turing machine might take during the execution of a particular algorithm, there are still some algorithms where it is *mathematically impossible* to know the results.

The proof that Turing used in his paper to disprove the Entscheidungsproblem, as well as the simplest to understand conceptually, is known as the Halting Problem.

3.4 The Halting Problem, Explained

The halting problem asks a simple question. Given a program P and input I, can a program H determine whether program P terminates or runs infinitely, given P and I as inputs? Intuitively, our brains may decide that this problem is solvable. At first glance, why wouldn't it be? If you know the program, as well as the input itself, you should be able to calculate what happens. We should be able to tell if a program will halt or not in our heads. Written out in pseudocode, the halting problem looks like this.

```
--H is a program that determines if P halts on a given input I.
```

```
if ((H(P, I)) == TRUE) --A True result means that P halts, a false result means that P does
    not halt.
{
    run_forever(); --does not matter what this function is/does, as long as it runs forever
}
--So if P halts, we run forever. If P does not halt, we do not call run_forever(), and
    terminate.
```

At this point, your brain might have noticed that something is not quite right. Within the above code, there is a contradiction. If P halts, H runs forever. If P runs forever,H halts. But, the definition of P means that it only returns a true value if the program halts. If it is run within H, it no longer halts, as H runs forever. This leads into the main problem of undecidability.

3.4.1 Why is the Halting Problem Undecidable?

The halting problem is undecidable through a proof of contradiction (Part 2 - Source 5). There are two possible outcomes of this system -

- H returns true. This means that program P with input I halts. However, by definition of P, if H returns true we enter an infinite loop, meaning that P(I) does not halt after all.
- H returns false. This means that Program P with input I does not halt. By definition of P, if H returns false we terminate our program, so P(I) actually halts after all.

In both cases, a contradiction arises. This contradiction occurs due to the assumed existence of H, as its existence allows us to create a Program P that does not function correctly. Because of this, H can not exist. Therefore, we can not decide if H is true or not.

3.4.2 Why This Matters

This concept of unsolvability has profound impacts on mathematics and computational sciences as a whole. Through thought experiments like the Halting problem, we are able to prove that the field of mathematics and computability are not, and will never be, fully solved. There will always be questions that are impossible to answer, despite our intuition telling us otherwise. As almost all programming languages are created using the concept of Turing-completeness, all programming languages will also have calculations that will always have an undeterminable answer. However, understanding these limitations of computing, and what they mean, will allow us to better form more comprehensive views of mathematics and the world, and get more efficient and accurate calculations in the future. Computation and theoretical programming are nowhere close to solved sciences, and these examples of unsolvability show just how much further we have to go.

3.4.3 Other Undecidable Problems

Due to the nature of unsolvability, there are theoretically infinite unsolvable mathematical problems. Ranging from logic problems, topology, matrices, to formal language grammar, there is no field of mathematics that

is free of unsolvable problems, as they are all based on the similar concept of unknowable outputs. However, there is one unsolvable problem that I found particularly interesting, known as the Mortal Matrix Problem. The concept is as follows:

Given a finite set of n x n matrices of integers, can they be multiplied together in some order, with potential repetition, to equal the zero matrix? This problem has been proven unsolvable for a set of six 3x3 matrices, as well as a set of two 15x15 matrices (Part 2 - Source 6). This concept is based upon the Post Correspondence Problem in table form, another unsolvable problem within computation. Essentially, it is impossible to determine due to infinite looping within the solution, leading to an unsolvable problem.

The reason I find this so interesting is because matrix multiplication is *everywhere*. From graphics calculations to search engine optimizations, any program built on the concept of matrices and linear algebra must be designed with this concept in mind, to avoid inconsistent or unsolvable results. Whereas the Halting problem was more theoretical, and did not have as much real world application, this unsolvability problem has real world impact, and must be actively designed around.

4 Execution Time Experiment

4.1 Introduction

In this final section of the paper, I will be expanding further upon the experiment set out in Part 1. I will be testing additional languages' computation times, and how that is related to the typing of each language tested, and what that means for deciding on programming languages for a project. In addition to covering Haskell and Python more in depth, I plan to also introduce tests for C, C++, and Java. Finally, I will be designing additional algorithms to test the languages on as a potential future area of study, and why I believe these algorithms would be computationally interesting.

4.2 Recap of Results from Part 1

As a refresher for what was covered in Part 1, I designed an experiment to compare the computation time of Haskell and Python. The algorithm I designed was quite simple - create an array of 500,000 entries, fill each index with a random number between 1 - 9, add 1 to each entry, and then sum the list. The Python algorithm used in testing is included below for reference.

```
randomlist = []
for i in range (0, 500000): --for loop
  n = random.randint(0,9) -- generates randoms
  randomlist.append(n) --appends to list

for i in range (0, 500000):
  randomlist[i] = randomlist[i] + 1; --adds 1 to each

print(sum(randomlist)) -- sums list
```

The results were as follows - On average, the Python program took 2600 milliseconds to run, while the Haskell version of the same program took 350 ms. Something of note, however, is that the Python test was run in Replit in Part 1, whereas Haskell was run locally. In Part 3, I will be redoing the Python portion of this experiment on a local installation of Python, to further control for outside variables to allow for a more fair comparison.

4.3 Additional Languages

The additional languages I chose to investigate were Java, C, and C++. For Java, I chose it due to its frequency of use in industry today, as well as its historical stigma of being a slow language to run. I wanted to see if the rumors were true, and how it compared against other popular programming languages. The reason I included C was due to its historical significance, as well as the fact that C++, Java, Python, and Haskell are all C derivative programming languages. I wanted to see if the original language these are all based off of is faster due to its simplicity. Finally, I will be testing C++. To differentiate C++ from C, I chose to use vectors and vector operations instead of standard array operations, to see the difference in computation time between the two.

All programs tested below are available on GitHub.

4.3.1 Python, Done Properly

In addition to testing C/C++ and Java, I wanted to retest the Python portion of the experiment on a local installation of Python, instead of Replit. I did not expect it to make a substantial difference in computation time, but I chose to retest for the sake of consistency. After porting the code on Replit to a local installation, the results were as follows:

Trial No.	Execution Time (ms)
1	529ms
2	558ms
3	571ms
4	553ms
5	524ms
AVERAGE	547 ms

Looking back, I realize I did a disservice to Python by doing my initial testing on Replit, instead of locally. While I expected it to cause some difference, I did not expect it to be 5x faster on a local installation compared to Replit. This brings the results much, much closer than before, with Python only being about 30 percent slower than Haskell. While this means my initial findings in Part 1 are still accurate, the improved results will help in comparisons between Python and other languages, as well as for the sake of scientific fairness.

4.3.2 Java

After Python, I began my investigation into Java. Creating the test algorithm was simple, and was able to compile and run without issues. For consistency's sake, the algorithm was run on the same computer as the previous Python and Haskell examples, on a local installation of Java. The results from the experiment were as follows:

Trial No.	Execution Time (ms)
1	73ms
2	88ms
3	74ms
4	90ms
5	66ms
AVERAGE	78.2ms

Overall, the speed of the Java execution was shocking to me. However, upon further research, I realized where my mistake in understanding was. Java, like C and C++, is a compiled language. This means that it will always be faster than an interpreted language like Python. The historical slowness does not come from the actual execution time, however, but from the implementation of libraries and the JVM. As neither of these were being tested here, as the timer did not start until the compiling was already done, the calculation speed was only slightly slower than C++. Overall, I found this very interesting, and dispelled one of my beliefs about Java being a slow language!

4.3.3 C

Same as before, convert the code to the new language, compile and run. For this, I expect very fast results, due to the efficiency and speed C is known for performing its calculations, due to its efficient compiled nature.

Trial No.	Execution Time (ms)
1	9.8ms
2	12.6ms
3	9.5ms
4	10.3ms
5	11.8ms
AVERAGE	10.8ms

There is a reason C is considered the king of fast execution times, and the preferred language for any critical, speed focused system - and I think this little experiment proved why. By being a static, compiled, and strongly typed language, everything about C is designed to create as fast of an execution time as possible, with as little overhead as possible. However, I did not expect just how much faster C truly is compared to everything else. C is over 50x faster than Python at this experiment, and 7x faster than Java. I see why all other languages tested are built off of C - there truly is no other better or faster programming language to branch off of.

4.3.4 C++

As mentioned previously, I wanted to ensure my results between C and C++ were different and both scientifically valuable. Because of this, I chose to make use of a C++ exclusive feature - vectors. Vectors are considered the "improved" form of arrays, and are suggested over using regular arrays in C++ programs. I wanted to see if this substitution had any impact on overall execution time.

Trial No.	Execution Time (ms)
1	17.2ms
2	15.4ms
3	16.9ms
4	15.9ms
5	12.3ms
AVERAGE	15.5ms

As was expected, C++ was slightly slower than C, but not by much. C++ still took a cozy second place, and was still 5x faster than its closest competitor, Java. While the performance loss of swapping from arrays to vectors was small, it still might be a consideration in some systems with hard deadlines. However, for the average use case, there is no sizable execution time difference between C arrays and C++ vectors. As with C and Java, the compiled nature of C++ makes it ideal for projects that value execution speed. However, the need to compile can slow down development and add development time.

4.4 Results Comparison

The average results of each language is laid out below:

Language	Avg Execution Time (ms)
Haskell	358ms
Python	547ms
Java	78.2ms
C	10.8ms
C++	15.3ms

Overall, I believe the results found re-enforce commonly held beliefs in industry. Compiled languages, overall, are just so, so much faster than the interpreted versions. If you require speed, you go with C or C++. If you require portability, you go with something like Java. If you want portability and ease of development, or are in an area where execution time does not matter, go with Python. The only confusing aspect of my results was the slow overall speed of Haskell code compared to C or Java. Even after ensuring I was running the code as compiled and not interpreted, the speed difference was still quite large. Haskell should be closer to the other compiled languages, instead of being closer to Python. Upon further research, I realized that some of the slowness in the Haskell code comes from how print() and the UTF encoding is handled in Haskell (Part 3 - Source 1), and not based on the actual execution time. I admit, however, that this area requires further research to understand fully.

4.5 Potential Future Algorithms to Test

I think there are many different areas I could test all of these languages on. From I/O operations like reading/writing to a file, calculations that let lazy languages like Python and Haskell excel, as well as seeing how each language handles multithreaded programs. If I had to design another experiment, however, I think I would focus further on the differences between compiled and interpreted languages. Are there any calculations or programs that interpreted languages could do as well as compiled ones? Are there ways to reduce the gap between the two? I think I would predominantly focus on using Haskell, as the ability to run haskell as both an interpreted and compiled language would be invaluable for ensuring a fair overall test.

4.6 Testing Conclusions

The results I found in this experiment were very interesting, and opened a lot of avenues of future research. In particular, I wish to explore functional programming languages more, and see if there are ways to more accurately measure execution time of these programs. As well, investigating the difference in Haskell between ghc(compiled) and runhaskell(interpreted) could be a very scientific way to compare algorithms, as there are few other languages that allow you to swap between an interpreted and compiled form like that. While the results I found did not reveal anything too surprising or novel, seeing my beliefs re-enforced through scientific experimentation was a very encouraging feeling.

5 Overall Conclusions

Overall, I found this paper a very valuable learning experience. In a traditional exam format, you would cram a day or two before an exam, absorb as much as you possibly can, throw it all onto paper, and then forget it all immediately after. I think being able to choose topics you are interested in, and then use that as a springboard for research into programming language theory, was a far more engaging way of interacting and solidifying the content of this course. For example, I think lambda calculus and string rewriting are a little bit dense and boring, and would have disliked having to study them for an exam. However, being able to focus on a concept I do enjoy, like running little algorithm experiments or investigating borderline philosophical problems with the Halting Problem was a much more enjoyable way to learn this course. While I understand writing a paper of this length and depth is not for everyone, I think the opportunity for me was much appreciated. In the future, however, I think you could make this report open to groups. It might help some that are less confident in writing focus more on the technical research aspect, and leave the writing to those who might have less of a grasp of the theoretical aspect, but are far better at condensing and explaining complex subjects.

6 Sources

PART 1 SOURCES

1. Learnyouahaskell.com/introduction
2. https://towardsdatascience.com/what-is-lazy-evaluation-in-python-9efb1d3bfed0
3. https://pythonconquerstheuniverse.wordpress.com/2009/10/03/static-vs-dynamic-typing-of-programming-languages/
4. https://www.freelancinggig.com/blog/2019/01/07/haskell-vs-python-what-you-need-to-know/
5. https://www.tutorialspoint.com/generating-random-number-list-in-python
6. https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution
7. https://stackoverflow.com/questions/5968614/how-to-get-a-programs-running-time-in-haskell
8. https://stackoverflow.com/questions/30994484/haskell-generate-and-use-the-same-random-list
9. https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/

PART 2 SOURCES

1. https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html
2. https://plato.stanford.edu/entries/turing-machine/
3. https://chortle.ccsu.edu/StructuredC/Chap01/struct01$_5$.*html*
4. https://dev.to/gruhn/what-makes-a-programming-language-turing-complete-58fl
5. https://brilliant.org/wiki/halting-problem/
6. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.45rep=rep1type=pdf

PART 3 SOURCES

1. https://stackoverflow.com/questions/57124974/why-is-this-simple-haskell-program-so-slow