CPSC-354 Report - Part 1

Erik Hombledal
Chapman University

November 13, 2021

Abstract

In this 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.

1 What Makes Python and Haskell Different?

To start, I will begin with an introduction on the differences between Python and Haskell, and how these differences can impact performance during execution.

1.1 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, you know, *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(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.

1.2 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(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.

1.3 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 -

1.3.1 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. (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.

1.3.2 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 (Source 3). In practice, a weak 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 static 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.

1.3.3 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.

1.4 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. 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 (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 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 for you to run, as well as a snippet of the Python program to see it conceptually.

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

Haskell Code - Github (Source 6) (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
```

3 Results and Applications

3.1 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 0.75 seconds. Despite performing identical list operations, the Haskell calculation was almost 3x 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.

3.2 How Do We Apply This in the Real World?

3.2.1 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.

3.2.2 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.

3.2.3 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.

4 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 deper 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, and potentially tie it to Python and Haskell as I discussed here.

5 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/