CPSC-354 Report

Your Name
Chapman University

October 18, 2021

Abstract

Short introduction to your report ...

Contents

1 Introduction

NOT DONE. I will write my introduction once all parts are written. I find it easier to introduce my topics after I have written the bodies for all the respective parts. NOT DONE.

2 Haskell

2.1 A Brief Introduction

The vast majority of readers to this point are familiar, if not proficient, with the concept of object oriented programming. It is characterized by the representation of things as "objects", offering users the option to hide certain processes, pieces of code, from outsiders as well as to store those processes in their respective sections. Another key aspect of object oriented programming is the manipulation of data through variables.

Functional programming, on the other hand, is based on the evaluation of functions. The manipulation of variables, as seen in object oriented programming, is not possible—functional programming deals with immutable data. Loops are also not supported here. Instead, iterative data is managed by recursion.

This seemingly radical shift in paradigms sound daunting at first. And, to an extent, it is. However, all good things are difficult at the beginning—no one is born knowing how to program, yet you, reader, have made it this far. As such, rather than as an academic paper, I will treat this as a conversation.

2.2 Two Languages, One Calculator

We will first be analyzing the differences between object oriented and functional programming by building some functions for a simple calculator with a language from each paradigm—Python and Haskell, respectively. This will allow us to understand how functions, variables, and types behave for each language.

Let's begin with addition. For any two numbers, a, b, $a + b = c$, where c, is also a number. How about we code it in Python. We define a function and its parameters, in this case a and b, and implement the operation inside the body. It looks like this:

```
def addition(a, b):
    c = a + b
    return c
```

As insultingly trivial as this may seem at first, let's look a bit more in depth at this function. It receives two variables, creates a new one, and returns it. More so, it doesn't specify the required types for addition. It doesn't matter if the input is an integer, double, floating point, or even strings for concatenation, our humble function just does it.

Now, let's do it with Haskell. Haskell does not support type inference, so we must specify exactly what we wish to compute and with what type of input. Let's begin with natural numbers. The first step is to explain to our computer what is a natural number. It is comprised of all positive numbers as well as zero—many mathematicians dispute 0's membership in this number club with much fervour, however we have more important things to do, such as defining addition. Back to it.

Knowledge of discrete mathematics is important here, mainly because we must formally define our number system. Let us define a natural number as any number NN. This number is either 0, or a successor of it.

We define it within Haskell as seen below:

```
data NN = 0 | S 0
```

Now that we have defined the data type, let's move on to writing the function. Recall that Haskell is based on the evaluation of functions rather than variables inside functions. Our function adds two natural numbers: in other words, it takes in NN, adds it to another NN, and finally returns some NN, where NN could be any natural number. Elegantly put, $NN \to NN \to NN$.

The issue is that we have defined our natural numbers to be either 0 or its successors, therefore our addition will be dependent on this definition. As mentioned before, Haskell is based on recursion, so we will have to incorporate this as well. From Discrete Mathematics, we know that successors are commutative.

That is, adding $1 + 1$ is equivalent to adding $S(0 + 1)$. Furthermore, we know that the addition of zero and any number x is equal to x. Consider this our base case for our oncoming recursive implementation. Basing off from before, adding $S(\text{a})$, b, is equivalent to the sum of $S(\text{a} + \text{b})$.

Now that we have defined our data type, how our function treats the data type, as well as how it behaves, we can write it with Haskell. See below.

```
addN :: NN -> NN -> NN
addN O m = m
addN (S n) m = S (addN n m)
```

There we go, our first recursive function in Haskell. Now, let's up the ante a bit. Let's compare a slightly more complex function and analyze at the differences between the languages in which we implement it.

2.3 Fibonacci Sequence, Side Effects, and Benefits

We will first compare the implementation of the Fibonacci code for both languages. In Python, we could write it like this. [GFG]

```
toCalculate = 10

def fibonacci(n):
    a = 0
    b = 1

    # Check is n is less than 0
    if n < 0:
        print("Incorrect input")

    # Check is n is equal to 0
    elif n == 0:
        return 0

    # Check if n is equal to 1
    elif n == 1:
        return b
    else:
        for i in range(1, n):
            c = a + b
            a = b
            b = c
        return b

print(fibonacci(toCalculate))
```

Something interesting is happening here. We are creating a global variable *toCalculate*, giving it an initial value of 10, and passing it to our function. Inside, we see how our local variables a, b, c, are being modified per each iteration. More interestingly, we have the option to reference variables and modify them at will.

Before moving on to Haskell, we must quickly look at pattern matching. Recall piece-wise functions from math. These functions behave differently based on the value of the given input. See the example below.

$$fib(x) = \begin{cases} 1, \text{ if } x = 0 \\ 1, \text{ if } x = 1 \\ fib(x-1) + fib(x-2), \text{ if } x >= 2 \end{cases}$$

See how the Python code is somewhat similar. It has different procedures for each specified case. Haskell's pattern matching works in a similar, slightly more elegant way in comparison to Python, and closer to our piece-wise function. We specify different cases for a function so that Haskell knows what to do in each case. When we call the function, Haskell compares the given arguments to the argument patterns we have written. As mentioned in *A Gentle Introduction to Haskell*, "Pattern matching provides a way to 'dispatch control' based on structural properties of a value."[GH] Now that we have a better understanding of pattern matching, we can translate our piece-wise function to Haskell.

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

See how Haskell's implementation is almost identical to our mathematical representation. However, a valid question would be, "why is this better than Python?" One of the main reasons is because it is more prone to be "mistake free". We can call our Haskell **deterministic**. It will always give the same result given a respective input.

More so, our Haskell code has no external state nor variables that are being mutated. In fact, our Fibonacci sequence is defined upon itself, function-wise rather than in terms of variables. This is crucial because functional programming languages, such as Haskell, don't allow for "side effects". As mentioned by Computerphile, an example of this is a mutable state, in essence a state that you can "mutate, you can update it, [...] so it can change from one value to another as the program progresses." [CPh] In synthesis, this method of implementation allows for clarity and security, as all our "function dependencies are stated explicitly" [MI].

2.4 Caesar Cipher

A cipher is defined as any method of transforming a message to hide its meaning. It has been used throughout history for efficient and secure communication among parties. One of the most famous and simple ones is the Caesar Cipher—named after Julius Caesar, who used it in his private messages. It is denominated as a substitution cipher, where each letter in the message is replaced by another letter, usually through a specified key. For example, if we wanted to encode "abc" with a key of 2, the result would be "cde". Although it serves no real security methods in today's times, it is a powerful example when it comes to considering the manipulation of strings regarding our comparison between languages.

To code it, we need to have a string to encipher and a key to know how much the shift will be. From there on, we loop through the string and change the individual character by the amount specified by the key. This is done by representing our string through modular arithmetic, where each letter is assigned a number—for example, A=0, B=1, ..., Z=26. Through this method, we can use the following equation for determining how each letter will be updated. [WK]

$$Encipher_n(x) = (x + n) mod 26$$

We use mod26 for cases where our key causes the encrypted letter to shift past 26. In essence, if our letter is Z and the key is 3, $(26 + 3) mod 26 = 3$, which changes Z to C. The code below shows a possible implementation in Python.

```
def Encipher(text,s):
    result = ""

    # loop through text
    for i in range(len(text)):
        char = text[i]
```

```
        char = char.lower()

        if char == ' ': # to keep spaces
            result += ' '
        else:
            result += chr((ord(char) + s-65) % 26 + 65).lower()

    return result
```

Notice how, again, our Python code is dependent on manipulation of variables. On the same note, take a look at how we don't have to specify individual character manipulation since the language allows us to do this all from the same function.

For Haskell, the situation is a bit more complicated. We must specify the behavior for each character as well as for the string as a whole. I suggest we begin by looking the shifting procedure for a single character, as seen below.

```
shiftedchar c n = chr((mod ((ord c - ord 'a') + n) 26) + ord 'a')
```

Notice how we are applying the same modular arithmetic equation specified previously. See below:

$$(ordc - ord'a') + n$$
$$x + n$$

All of this is then being modded by 26. Now, we can see the big picture and analyze how our code is being enciphered. [MWS]

```
lowercase m = words (map (toLower) m)

shiftedchar c n = chr((mod ((ord c - ord 'a') + n) 26) + ord 'a')
shiftedword w n = map (`shiftedchar` n) w

encipher m n = unwords(map (`shiftedword` n) (lowercase m))
```

Our enciphering program takes a message and a key, applies the shifted word function to each word as separated by spaces, which itself applies the shiftedchar function to actually shift the letters. Notice how our Haskell function is much more concise than our Python one. Similarly, it acts solely on the functions themselves in a deterministic and side-effect-free manner. No variables are used nor manipulated.

For deciphering, the code is rather similar. By the cyclical nature of the modulo operator on our shifting procedure, we can denote our deciphering equation as follows:

$$Decipher_n(x) = (x - n) mod 26$$

The following code is adapted from the enciphering code, changing the necessary manipulations such that an enciphered string can be read in its normal form—given we know the key, of course.

```
lowercase m = words (map (toLower) m)

normalchar c n = chr((mod ((ord c - ord 'a') - n) 26) + ord 'a')
normalword w n = map(`normalchar` n) w

decipher m n = unwords(map (`normalword` n ) (lowercase m))
```

Finally, the entire program would look as follows, including the necessary imports to run properly. To run it, run the following commands.

1. **ghci *the name you give this file***
2. **encipher *message* *key*** or **decipher *encrypted message* *key***

```
import Data.Char

main :: IO ()
main = return ()

lowercase m = words (map (toLower) m)

shiftedchar c n = chr((mod ((ord c - ord 'a') + n) 26) + ord 'a')
shiftedword w n = map (`shiftedchar` n) w

normalchar c n = chr((mod ((ord c - ord 'a') - n) 26) + ord 'a')
normalword w n = map(`normalchar` n) w


encipher m n = unwords(map (`shiftedword` n) (lowercase m))
decipher m n = unwords(map (`normalword` n ) (lowercase m))
```

For further practice, I recommend incorporating further shifting possibilities to the code above. For example, our code right now only works with alphabetical characters. However, we could extend it such that it can encipher punctuation marks and numbers.

2.5 Haskell, irl

If you are like me from a few weeks ago, you might still be leaning towards Python due to its close relationship semantically to English, facility to write programs, and an understandable comfortable relationship with OOP/imperative programming. However, as programmers, efficiency is one of the most important things we strive for. Haskell, by nature a compiled language—in essence converted directly into machine code for the computer to execute—, is, and will always be, faster than Python—or any other interpreted language.

"Alright, then," you might say, "I still see no use of it in real life." In that case, let's bring up Facebook and its spam filtering. As of 2021, Facebook boasts over 2.89 billion active users. [FB] For such a big platform, the monitoring of spam is something very difficult to achieve. In comparison, consider the 1.5 billion users Gmail has and the constant spam you can expect to receive [GM]. One of the ways in which Facebook is so successful at monitoring and filtering spam, malware, and other types of abuse is through a system called Sigma. It automatically detects such content and deletes it such that it doesn't appear on one's feed.

Sigma serves as a *rule engine*. Every interaction on Facebook is evaluated by sigma against a set of rules that are coded within it. These rules are classified within different groups, which allows for Facebook to determine the types of interactions that occur. The source code for Facebook is running at all times through Sigma. Let's ponder about this for a second. The source code for a platform consisting of 2.89 billion active users is constantly being run through code for monitoring types of interactions. Clearly, this type of program must be able to run efficiently.

How does Haskell fit in? Recall that Haskell doesn't allow for side effects and requires strongly typed implementation. This ensures that rules don't interact with nor modify each other, preventing Sigma from crashing. More so, testing is easier since rules don't interact with each other. Finally, strong types help eliminate bugs from entering Sigma. Other reasons include performance as compared to the language with

which Facebook would previously fight spam with: FXL. As mentioned by Facebook, "FXL's slower performance [required] writing anything performance-critical in C++ and putting it in Sigma itself. This had a number of drawbacks, particularly the time required to roll out changes." [SIG]

Regarding performance itself, efficiency is crucial for such a big platform. Facebook states that "Requests to Sigma result from users performing actions on Facebook, such as sending a message on Messenger, and Sigma must respond before the action can take place." For the 25 most common types of requests received by Sigma, Haskell outperforms FXL in all but two cases, often three times faster and for an average of 20-30 percent increased efficiency. [SIG]

I truly recommend reading more on this topic, as it is one of the single most pertaining examples of Haskell's efficiency and industry-wide use. Other companies that also use Haskell include AT&T, Chordify, Deutsche Bank, Eaton, NVIDIA, and many others. [HA]

2.6 Haskell, Booleans, and Introducing Logic

3 Programming Languages Theory

In this section you will show what you learned about the theory of programming languages.

4 Project

In this section you will describe a short project. It can either be in Haskell or of a theoretical nature,

5 Conclusions

Short conclusion.

References

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

[FB] Statista

[GH] Case Expressions and Pattern Matching, *A Gentle Introduction to Haskell*.

[GFG] Python Program for Fibonacci Numbers, Geeks for Geeks.

[GM] Gmail Wikipedia

[CPh] Programming Paradigms - Computerphile, ComputerPhile.

[MI] Pure Functions and Side Effects, Maksim Ivanov.

[MWS] Introduction to Haskell, Caesar Shift Cipher MultiWingSpan.

[WK] Caesar cipher Wikipedia.

[SIG] Fighting spam with Haskell Facebook engineering.

[HA] Haskell in Industry Haskell.