CPSC-354 Report

Andrew Dillon
Chapman University

October 17, 2021

Abstract

Short introduction to your report ...

Contents

1 Introduction

This report will touch on three primary areas. Firstly, we will discuss the programming language Haskell. Next we will dive into the theory behind programming languages. Finally, we will discuss a project.

2 Haskell

Haskell is a non-strict functional programming language first released in 1990. It was by no means the first functional language, and certainly not the last, but it has had a profound effect on programmers.

2.1 A Look at the History of Haskell

Development of the language started three years earlier in 1987 at a conference in Portland, Oregon on Functional Programming Languages and Computer Architecture [HH:BLC]. At the time, dozens of similar programming languages were emerging, with varying levels of complexity and power. The committee realized that this over abundance of similar languages was hindering the growth and expansion of this type of language as a whole. So the committee set to work attempting to develop a standard, common language to foster growth and interest in this functional programming languages as a whole.

2.2 Haskell's Namesake

It was named after American logician and mathematician Haskell Curry (the languages Brooks and Curry are also named after him). He is perhaps best known for the technique of known as currying. Currying is the process of breaking down a single function which takes in multiple arguments into a series of smaller functions which each only take a single argument. This is a core principle of Haskell.

2.3 Functional Languages vs Object Oriented Languages

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

Functional programming languages are also lazy languages. This means that a function will evaluate expressions when and if their values are actually used by a function. For example take the function:

$$f :: Int -> Int -> Int$$

$$fxy = y$$

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

$$f(1+1)(2+2)$$

If this were an eager programming language, the compiler would fist calculate $(1+1) = 2$ then it would add together $(2+2) = 4$. After this, the function would return the value passed in for y, which in our case would be 4. Calculating $1+1$ was unnecessary, as it was never used in the function. While this is a very small step in this example, a more complicated function could take up value or a situation where a function like this is called often, would begin to slow down the program, leading to it being less efficient. A lazy programming languages like Haskell work around this only preforming these calculations when required (a process known as "call by need"). Haskell would only perform the operation $(2+2) = 4$, as that is the only step this function ever requires [LG].

2.4 Pattern Matching

Pattern Matching is perhaps Haskell's greatest strength, as well as its most fundamental property.

2.5 A Brief Tutorial

The best way to truly understand a new language is by getting your hands dirty and writing with it. We will create a simple program to add and subtract numbers written in roman numerals. Roman numeral inputs will be taken in as strings. This project can easily be expanded to include more mathematical operations.

Before we write the Haskell code for this project, lets briefly discuss how to attack the problem. We can leverage the built in operations Haskell can perform on integers. Rather than defining addition and subtraction for a new data type, we can instead convert the Roman Numerals to integers, add and subtract as need be, and then convert them back to strings.

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

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

```
main = do

  return ()
```

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

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

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

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

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

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

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

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

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

```
getCharacters :: String -> [Integer]
getCharacters [] = []
getCharacters (x:xs) = (getValueOfCharacter (toUpper x)) : getCharacters xs
```

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

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

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

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

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

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

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

Finally, we can do our addition and subtraction.

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

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

To typeset Haskell there are several possibilities. For the example below I took the LaTeX code from stackoverflow and the Haskell code from my tutorial.

```
-- run the transition function on a word and a state
run :: (State -> Char -> State) -> State -> [Char] -> State
run delta q [] = q
run delta q (c:cs) = run delta (delta q c) cs
```

This works well for short snippets of code. For entire programs, it is better to have external links to, for example, Github or Replit (click on the "Run" button and/or the "Code" tab).

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.

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

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

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

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