CPSC-354 Report

Ariel Gutierrez
Chapman University

December 21, 2021

Abstract

This report introduces the reader to functional programming and the basics of Haskell to prepare the reader to create their first functions using Haskell. Moreover, the reader is introduced to topics in Haskell such as type classes and monads. In the second portion of the report, the reader encounters structured programming, which led to Hoare Logic. At the end, the report walks the reader through a project in Haskell using string rewriting systems to implement a calculator for Roman numerals.

Contents

1 Introduction

Haskell is a struggle to learn at the beginning phases. The fact that Haskell is a functional programming language and not an imperative programming language is what makes learning Haskell a daunting task. Though it may be difficult to grasp at first, this report will walk you through the beginning phase of learning Haskell as well as making simple functions and programs with Haskell. We will briefly go over topics in

Programming Languages Theory that may help the reader grasp not only Haskell but also concepts such as Hoare Logic and structural programming.

2 Haskell

In this section, I will provide a brief introduction to Haskell for a beginning *Haskeller*. However, before learning about Haskell we should first should learn about functional programming and how it compares to imperative programming.

2.1 Functional Programming vs. Imperative Programming

Most of us are familiar with imperative programming, which is used by many programming languages such as Python, C++, and C. Imperative programming focuses on the programmer specifying which tasks the program should perform and how to track changes in states, usually done with conditionals and loops [1]. A programmer needs to keep track of the states in order to implement an algorithm and it is done by modifying the memory. In comparison, functional programming focuses the programmer focuses on specifying the information desired and what transformations a function has. The steps in making an algorithm in function programming is through function calls, including recursion.

Some advantages to functional programming over imperative programming are the increased readability of functions, easier testing and debugging due to the isolation of functions, and easier syntax which leads to less code. Moreover, a major difference between imperative programming and functional programming is that an imperative programming language typically distinguishes statements and expressions. An expression is a combination of values and functions that are combined and interpreted by the compiler to create a new value. A statement does not return a value, but instead computes a value by following the step by step process outlined by the programmer. A statement is a standalone unit of execution and does not return anything[19]. Overall, the purpose of an expression is to create a value, and the purpose of a statement is to have side-effects. For example, for an expression we can have $a+ = 2 + n$ so the value can change depending on the value of n and a, whereas in a statement we would only have $2 + 2$(we only have a standalone unit, in this case an integer).

In a functional programming language, such as Haskell, everything is an expression. The benefit of this is that expressions can be combined to form larger expressions and, unlike with statements, a series of expressions doesn't necessarily mean that there is any implied order of execution.

In order to write efficient code in Haskell we need to stray away from the conventions of imperative programming of telling a program how to solve a problem to telling a program the structure of a problem. In imperative programming one is more likely to write out a bunch of statements of what the program should do. Instead of thinking about the computations that the program should be doing, think about the patterns that you see in those computations. Is there any place where you see the same type of computation steps being called? Would the code be simpler by adding recursion? It is easy to get stuck on finding the most computationally quick solution, but it is difficult to find a readable, simple solution. Being able to focus on the definitions of a function instead of how to get to a solution is essential to programming with Haskell. It may be easier to think of Haskell functions as being related to mathematical definitions. This model of computation is called equation reasoning, since it takes a mathematical definitions and converts it into a function using recursion [PL].

2.2 Important Things to Note about Haskell

Haskell is frequently referenced as *lazy*; even Haskell.org calls Haskell a *lazy* language. It is important to note that just because Haskell is *lazy* and has simpler syntax than imperative languages, this does

not mean that Haskell is easier than an imperative programming language. Haskell isn't just called *lazy* because it has simple syntax, it is called lazy because in programming language theory, lazy evaluation is an evaluation strategy which delays the evaluation of an expression until its value is needed, whcih is called call by value[20]. The functions in Haskell do not evaluate their arguments until they are needed, such as when calling a function to compute in the main block. The importance of this is that it avoids repeated evaluations, which can reduce the running time of some functions over other evaluation strategies, such as call by name, where a same function is repeatedly evaluated.

Another benefit to being *lazy* is the ability to define control flow as abstractions instead of primitives and the ability to define potentially infinite data structures. This allows abstraction and allows for more straightforward implementations of certain algorithms. Moreover, Haskell is *lazy* because it uses less lines of code to compute the same thing that an imperative language would. Unnecessary lines of code, like if-else statements and loops, can be reduced to expressions using recursion; this is seen later in the report where we finally begin to build Haskell programs.

Another important thing to note about Haskell is about the *type classes*. You do not have to explicitly write out the types of arguments and outputs in a Haskell program. Haskell has types and these can be inferred by the compiler. Writing the types of the inputs and outputs is optional; however, it is important to note that labeling the types could be useful for readability and for documenting the usage of functions. Moreover, Haskell has type classes which means that the polymorphism used by Haskell allows different operators to be used on many different kinds of numbers and literals such as 1 and 2 can represent fixed and arbitrary precision integers [3]. This is also called *overloading*.

Furthermore, Haskell has *monads*. Monads are difficult concept to explain but they are frequently used in Haskell, such as in the IO system of Haskell and the *do* expression in Haskell [4]. Monads are a type of data structure - a data structure is a data type together with some operations and some laws that those operations should be able to satisfy. Instead of trying to understand the intricacies of a monad it is important to know why monads are useful for structuring a functional program. Monads are modular, which means that they let computations be composed from simpler computations and separate the expressions from the computations taking place.

An example of a monad in Haskell is an error monad. Haskell has a built-in monad called Maybe, which encapsulates an optional value. So a value of type Maybe a either contains a value of type a or it is empty. When writing functions in Haskell, we can use this error monad as a way to deal with errors or exceptional cases without resorting to using errors [19]. It is a very simple monad since all the errors are represented as nothing. As you can guess, the Maybe monad makes simpler computations, since the programmer does not need worry that much about errors stopping the program from compiling.

Moreover, monads are flexible, so they let functional programs be more adaptable than imperative programs because it lets the computation be in a single place instead of being computed throughout a whole program. Monads are also isolated, which means that the functions are isolated from the main block of the program. This allows Haskell to create a imperative type of computational structure without loosing the benefits of the recursion offered by Haskell [5]. In summary, it is easier to think of a monad as something that allows computation or action, which is distinctly different from a function in Haskell which exists as a strategy for computation or action.

2.3 The Syntax of Haskell

In this section, we will go over the basic syntax of Haskell so that we can then form our first functions. before starting, you should either have Haskell ready to run on your machine (use this to help download it onto you computer) or you can use the website replit.com to write and run Haskell code. Haskell code is

stored in a **.hs** file and can run the code using the interpreter ghci, or you can compile the code and run it using replit.com.

Strings are denoted using the double quotes "". If the string is only for a single character the double apostrophe is used '. There are also basic type classes for **numbers** as shown in the table below.

Example	Number Type
1	Integer, Int, floating point values
1.0, 1bef0	Floating point values
0x1, 0X1	Hexadecimal
-1	Negative number

Another basic data type in Haskell is a list. Lists, like in Python, can be used to hold any data type, such as a list of Integers or a list of single character strings. However, a difference is that lists in Haskell can be made in a variety of ways. For example, an empty list can be constructed using [], and if we have x as some data and xs is a list, then $x : xs$ is list with its first element being x and a list xs that follows it. Below are equivalent lists in Haskell:

```
-- the following are all equivalent lists in Haskell
[1,2,3,4]
1:(2:(3:(4:[])))
1:2:3:4:[]
```

Haskell also has its own built-in functions to deconstruct lists which are defined as:

```
head(x:xs) = x
tail(x:xs) = xs
```

Now we should learn the syntax of creating a function. The best way to do this is to write a simple function. One of the simplest functions I know is the identity function that given a number it returns that same number. First, make your **identity.hs** file to input your code. The first line of code is optional and it states the type of the arguments and the outcome of the function. We know that the identity functions has an input of a number and returns a number. I will use the data type Integer as the input so the identity function will return an Integer.

Moreover, we also want a name for this function. I will call it *ident* to imply that this will be the identity function; you can name your functions whatever you want as long as it is not a reserved word in Haskell. The first line should be written as

```
ident :: Integer -> Integer
```

As we can see, the first token is the name the function, which is then followed by two colons where we then show the data type of the input and an arrow that points to the data type of the output (in our case both are Integer). Next, we want to think about the definition of our identity function. I find it easier to find the mathematical definition of a function and then turn that into a Haskell function, but since the identity function is easy to implement, we don't need to do that. So the rest of our function is just the definition of the identity function:

```
ident :: Integer -> Integer
ident x = x
```

Note that the "=" sign is not used as assignment and is instead used the same as a mathematical definition. Also, note that there are no parentheses surrounding the first x token; we can do this if there is only one

token with the function, otherwise we would need parentheses.

You can run this code in ghci by first running ghci and running the command **:load identity.hs** to load the function, and then type **ident #**, where # is any number you want to test. You can also compile this and the code for compiling the code is shown here. To compile the code, you need a main block, which is used to test your functions. You begin a main block with the lines **main = do**. You follow this with whatever you want to compile. I usually print the outcome of a function for a case to screen using **print $**. Anything after that will be printed to the screen. Here is an example of how I could test our identity function with a main block.

```
main = do
  print $ ident 10
  print $ ident 5
```

The main block above prints the value of the identity function given the value 10 and on the other line it prints the value of the identity function given the value 5. Most functions are not as simple as the identity function and usually have base cases. The order of the expressions in your Haskell functions is important.

I find it useful if for a general computation you write what you do last first and have the general case at the bottom of the Haskell function. It is also helpful to first create the mathematical definition before starting to write your Haskell function.

This is just a brief introduction to basic data types and syntax in Haskell. If you would like to learn more here are some resources that have helped me understand the basic syntax of Haskell:

- Haskell Cheat Sheet
- Haskell First Steps
- Haskell Basics

2.4 Creating Haskell Programs

Now that we have a basic understanding of what is functional programming and the basic syntax and operations of Haskell, we can start building programs and functions in Haskell. The factorial. Recall that the factorial multiplies an integer by ever integer that precedes it. So $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$. Now that we have refreshed our memory, let's consider the actual mathematical definition of a factorial:

$$0! = 1 \qquad \text{[Base Case]}$$
$$n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1 \qquad \text{[General Case]}$$

In the earlier sections, we mentioned that Haskell uses pattern matching to implement expressions as a function. We will use this same idea to create our program. Below is the code for a factorial function using Haskell and another using Python. [Factorial Function in Haskell]

```
-- function that implements the factorial function in Haskell
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n-1)
```

```
-- function that implements the factorial function in Python
def fac(n):
```

```
    if n == 0:
        return 1
    else:
        return n * fac(n-1)
```

In the Haskell function there is only 3 lines. In fact, we can even omit the first line which provides the types since Haskell provides type inference. However, I suggest to use write the first line that defined the types for readability and to help us understand the actions of a function. The second line is the base case and the third line is the general case, which includes recursion. The order of these expressions is very important. The base case should always be first. It is important to keep what we do last as the first lines.

If we compare the number of lines between the two functions, the Python contains more lines and is bit more complicated writing wise. The Python function also uses an if-else statement to differentiate the base case and other cases. In comparison, Haskell uses 2 lines less and it does not need to specify the base case and it does not specify what the function should output. Some key advantages that we see from the Haskell function is the added readability. We can directly see the mathematical definition from the Haskell function, whereas with the Python implementation, it is harder to read the function.

Furthermore, testing is easier with the Haskell function because we can use ghci to load the function while with python you would need to make a main block in order to test the function. Haskell allows testing in isolation, either in a main block or through an interpreter.

Our Haskell code has no statements and instructions. It only contains expressions that does not mutate variables, like our Python code does. Python's recursive implementation is not as short and not as simple as our Haskell function. This is because Python only incorporates some of the features of a modern functional programming language.

Let's test our skills again by trying to implement the triangular numbers. First, we should think of the mathematical definition.

$$tri(0) = 0 \qquad \text{[Base Case]}$$
$$tri(n) = tri(n-1) + n \qquad \text{[General Case]}$$

Since we have obtained the mathematical definition, now we can easily translate that into our Haskell function. First, we make our optional first line that tells us that given an Integer we should obtain as output an Integer. Then the rest is similar to our mathematical function other than a few parentheses a dropped when not need. You can compile this here.

```
-- function that implements the triangular numbers function in Haskell
tri :: Integer -> Integer
tri 0 = 0
tri n = tri (n-1) + n
```

There are also other functions that are fairly easy to implement:

- Fibonacci Numbers
- Towers of Hanoi
- Return length of string
- Returns if a number is even

3 Programming Languages Theory

In this section, we will go over the basic theory of Hoare logic as well as the rules associated with it. We will go over some exercises to familiarize ourselves with using Hoare logic to prove the correctness of while loops.

3.1 Hoare Logic

Hoare Logic grew out of the methodology of structured programming. Structured programming is a programming paradigm that focuses on improving the readability, quality, and clarity of a computer program [6]. Today, almost every programmer learns about the importance of structured programming when they first begin to make computer programs. However, the concept of structured programming was not a well known concept around 1970.

3.1.1 Structured Programming

In Edsger Dijkstra's *Notes on Structured Programming* (1969), he goes over how a programmer's job is to organize a computation in a way to guarantee a solution. In the early years of programming, what mattered the most to a programmer back in the day was making a computer program that delivered a correct answer, but Dijkstra proposed another concept that would influence the way programmers would program in the future. Not only was it important to create a program that outputs the desired results, but it is also important to prove it is correct in a convincing manner.

It is interesting that Dijkstra compared a large disorganized program to that of a mathematical theorem stating the validity of a conclusion when ten full pages of conditions are satisfied [7]. As you can guess, a theorem where you need to meet a substantial number of conditions is not a useful tool as compared to a theorem with a few conditions, such as the Pythagorean theorem. The same applies to programming, in that a large disorganized program is not structurally useful or manageable because it is difficult to prove what it outputs. You cannot use a program as a tool if you cannot clearly explain why it works.

Programming in the early days of programming relied on jumps as a basic mechanism to control flow, but structured programming focuses on the use of the structured control flow constructs of selection and repetition [8]. This includes the use of if-then-else statements, while and for loops, block structures, and subroutines. Block structures are crucial to structural programming because it allows the programmer to treat a group of statements as a unit. A subroutine is a sequence of program instructions that perform a specific task and like block structures are treated as a unit [9]. The importance of a subroutine is that they can be used wherever in the program wherever that specific task needs to be used. A subroutine is also referred to as a function or a method. A programmer provides a definition for what should be done and wherever the subroutine is called in the program it is then performed.

The significance of structured programming was that the programs gained a higher readability. Through the added clarity, not only was it easier to understand the inner workings of the programs, but it also helped with debugging the programs.

Recall that structured programming has an emphasis on repetition. This repetition can be implemented with the use of a while loop. A while loop is used to repeat a specific block of code until a condition specified by the programmer is met [10]. With the increased use of the while loop, it became important to prove the correctness of it.

3.1.2 Background Information on Hoare Logic

In 1969, C. A. R. Hoare published the paper *An Axiomatic Basis for Computer Programming* in which he proposes using mathematical logic rules to prove the correctness of while loops.

The reason for the formation of this paper was that Hoare did not agree with Peter Landin and Christoper Strachey's definition of a programming language, in which they define a programming language in a simple functional notation that specified the effect of each command on a mathematically defined machine [12]. This definition seemed to rely on too many complicated abstract representations.

A recurrent theme in Hoare's paper is that an important property of a program is to provide a logical basis for proofs. Moreover, he argues that the practice of proving programs is likely to alleviate some problems which affect the computing world, such as using logical rules to help us define and explain the purpose of a subroutine in program documentation [11].

Not only did Hoare's paper provide a basis for Hoare logic, but it also showed how we can use it to create clarity in the programs we write. It also draws a specific emphasis on structured programming because if we have a subroutine that has a while loop that is incorrect, all we need to do is change the while loop to the correct version. This way we only made one correction in debugging a program instead of making multiple corrections as we would have to in an unstructured program.

Moreover, Hoare emphasized that the single most important property of a program is to check if it accomplishes the intentions of its user. If the intentions of a program's user can be described by making assertions about the values of variables at the termination of the program, then the correctness of the program can be described by the rules in Hoare's paper. These formalized set of rules can be used to form a formal definition of a programming language because it describes the desired properties of the programs [12].

Over the years, the idea that was proposed in Hoare's paper has evolved into Hoare logic and has been improved by other researchers and it is what we will be exploring in this section of this paper. Hoare logic is now defined as a formal system with a set of logical rules for reasoning about the correctness of computer programs [PL].

3.1.3 New notation

before we begin examining the rules of Hoare Logic, I will introduce how you should read the new notation for rules first with first a couple of examples. I am doing this so that we can familiarize ourselves and learn to read the more complicated axioms of Hoare Logic.

Take into consideration the follow rule (not a rule of Hoare Logic):

$$\frac{\{P\}S\{Q\}\ \{Q\}T\{R\}}{\{P\}S;T\{R\}}$$

3.1.4 Rules of Hoare Logic

These are the rules of Hoare Logic:

Composition

$$\frac{\{P\}\ \mathrm{S}\ \{Q\}\ \{Q\}\ \mathrm{T}\ \{R\}}{\{P\}\ \mathrm{S}\ ;\ \mathrm{T}\ \{R\}}$$

Assignment

$$\{P[\exp/x]\}x := \exp\ \{P\}$$

While

$$\frac{\{I \wedge B\}S\{I\}}{\{I\}\ \text{while}\ B\ \text{do}\ S\ \text{done}\ \{\neg B \wedge I\}}$$

If-Then-Else

$$\frac{\{B \wedge P\}\ \mathrm{S}\ \{Q\}.\ \{\neg B \wedge P\}\ \mathrm{T}\ \{Q\}}{\{P\}\ \text{if}\ B\ \text{then}\ S\ \text{else}\ T\ \text{endif}\ \{Q\}}$$

Weakening

$$\frac{P' \to P\ \{P\}\ \mathrm{S}\ \{Q\}\ Q \to Q'}{\{P'\}\ \mathrm{S}\ \{Q'\}}$$

3.1.5 Exercise

In the following exercise we will prove the partial correctness of various programs. Partial correctness does not consider termination.

```
-- program
   while (i < 100) do
      y := y + x
      i := i + 1
   done
```

Let's use the rule for while loops from the rules of Hoare Logic described in Section 3.1.4.

$$\frac{\{I \wedge B\}S\{I\}}{\{I\}\ \text{while}\ B\ \text{do}\ S\ \text{done}\ \{\neg B \wedge I\}}$$

We want to show that we can represent our program in the same form as the above rule. Doing this will ensure that our program in Exercise 1 is partially correct. First, let's find the Hoare triple. To do this, I will begin by trying to find the preconditions. This program needs a predefined value for i, y, and x.

We can find the invariants through a few methods. For example, we can use pattern matching with the mathematical definition of the while loop (while rule). With pattern we get that $B = (i < 100)$ and $S = \{y := y + x; i := i + 1\}$, but we are missing the not so obvious value of I, which represents an invariant. This would be difficult to find with pattern matching, however, we can also use an easier way by doing a sample computation of the program.

I will write the sample computation in a table and note that I introduced a new variable, *time*. Each time unit will denote an iteration of our while loop. The preconditions for the program are that $i < 100$. So, I set $i = 0$, $x = 2$ as our initial values for our program. We could set these to whatever value we want. The first few lines can be computed as shown below

time	i	x	y
0	0	2	0
1	1	2	2
2	2	2	4
3	3	2	6
⋮	⋮	⋮	⋮
99	?	?	?

Though we have easily computed the first few lines, we know that it would be a long and repetitive task to fill out the rest of the table. This is because the condition for the while loop is $i < 100$. We would have to do 100 computations, which is quite annoying. But what if we find a pattern amongst the computations?

First, we will try to find what the value for i will be at $time = 99$. If we look at the first computations, we see that $time$ and i have the same value. This means that i should equal 99; that gives us the rule $i = time$. Another easy computation is finding what the computation for x is; x always remains at 2, so we get the rule $x = 2$. All we have remaining is finding the rule for y. By looking at the y column, we see that it increases by 2 every time. More specifically, $y = time * 2$, but recall that $x = 2$ and $i = time$. Therefore, by using simple algebra we get the rule $y = x * time$. Below are our rules:

$$i = time \tag{1}$$
$$x = 2 \tag{2}$$
$$y = x * time \tag{3}$$

Each of the rules above is an invariant. An invariant is a condition that can be considered to be true during the execution of a computer program or during a portion of a program [13]. This means that it is a logical assertion that is made to always be true during execution time. During the 100 iterations of our program the rules above hold, so these are also invariants.

Now we can find the last row of our table.

time	i	x	y
0	0	2	0
1	1	2	2
2	2	2	4
3	3	2	6
⋮	⋮	⋮	⋮
99	99	2	198

The power of invariants is that we do not need to write out the whole table. If we want to find the values of an iteration given the time, we can easily find our answer. This shows us the importance of invariants.

Let's attempt to write out our Hoare triple. A Hoare triple includes formalizing a program with its pre-condition and post-condition. First, we will try to find the precondition to our while loop. You might be thinking, "We have our invariants, so can we go on?" and the answer to that is...NO! There is still a small problem with our invariants; it relies on the value of i. i could be anything we want it to be, so if we wanted we could even start at $i = 1$ or $i = 98$. We need to tweak our rule for i and consider the case of $i = 1$. For this case $i = time + 1$. If $i = 2$, then $i = time + 2$. Do you see the pattern forming? Let's remove the concrete value and replace it with a mathematical variable, m. This way our rule changes to $i = time + m$.

Another problem is with our rule $x = 2$. We solve this by doing the same thing we did before and apply a mathematical variable, n, that can be anything. Therefore, our rule changes to $x = n$. We can use algebra to change our previous variants to the ones below:

$$i = time + m$$
$$x = n$$
$$y = x * time$$

Recall that we said that the pre-condition should include the initial condition for $i = m$ and $y = 0$. We originally said that our pre-condition would be $\{y = 0 \wedge i = m\}$. Our post-condition would be the operation

that our program does, which would be $\{y = time * x\}$. Thus, our Hoare triple turns out to be

$$\{y = 0 \wedge i = m\} \text{ while } (i < 100) \text{ do } \{y := y + x; i := i + 1\} \text{ done } \{y = time * x\}$$

but according to the Hoare triple, the pre-condition and the post-condition invariants should be the same. Does that mean that we have the incorrect pre-condition and post-condition? No, we don't! By rewriting them we can find our Hoare triple! We can rewrite the post-condition by using our first invariant (1) and our third invariant (3) through simple algebra.

$$\begin{aligned} y &= x * time && [(3)] \\ &= x * (m - i) && [(1)] \end{aligned}$$

Thus, our new post-condition is $\{y = x * (m - i)\}$. Next, we want to show that we can get our pre-condition from our post-condition.

$$\begin{aligned} y &= x * (m - i) && [\text{post-condition}] \\ &= x * (m - m) && [\text{let } i = m] \\ &= x * 0 && [\text{subtraction}] \\ &= 0 && [\text{multiplication}] \end{aligned}$$

$$\begin{aligned} & y = x * (m - i) && [\text{post-condition}] \\ \iff & 0 = x * (m - i) && [\text{let } y = 0] \\ \iff & 0 = m - i && [\text{divide by } x] \\ \iff & i = m && [\text{add } i \text{ to both sides}] \end{aligned}$$

Thus, we have shown that our pre-condition is the same invariant as our post condition. Therefore, our Hoare triple is

$$\{y = x * (m - i)\} \text{ while } (i < 100) \text{ do } \{y := y + x; i := i + 1\} \text{ done } \{y = x * (m - i)\}$$

Our Hoare triple has the post-condition as $\{y = x * (m - i)\}$, but the rule for while loops says it should be in the form of $\{\neg B \wedge I\}$. Recall that $B = (i < 100)$, thus $\neg B = (i >= 100)$ so we get

$$\{y = x * (m - i)\} \text{ while } (i < 100) \text{ do } \{y := y + x; i := i + 1\} \text{ done } \{y = x * (m - i) \wedge i \geq 100\}$$

Our last step is important to bring in the pre-condition and post-condition since we need it for our partial correctness assertion. Our last is using the composition rule, our first rule of Hoare logic.

4 Project

For the project section of this report, we will be constructing a calculator that uses roman numerals. By the end of this project, we should be able to do addition, subtraction, and multiplication over roman numerals.

4.1 Short Introduction to Roman Numerals

before we begin with the implementation details of the project, we will briefly go over how to form roman numerals and how to read them. Roman numerals are the symbols used in a system of numerical notation based on the ancient Roman system [14]. In modern times we know use the Hindu-Arabic numerals, which are a set of 10 symbols (1, 2, 3, 4, 5, 6, 7, 8, 9, and 0) that represent numbers in the decimal number system, that originated from India [15]. Below I have the roman numerals and their corresponding Hindu-Arabic numeral, which is the numbering system we use in the modern world.

Roman Numeral	Hindu-Arabic Numeral
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

The Roman numeral symbols above cane be combined to make newer numbers. For example, here are a few combinations of Roman numerals and their corresponding Hindu-Arabic numeral.

Roman Numeral	Hindu-Arabic Numeral
I	1
II	2
III	3
IV	4
V	5
VI	6
IX	9

As you can see there are some rules when forming numbers using Roman numerals. One rule is that when a symbol appears after a larger (or equal) symbol it is added[16]. So $VII = V + I + I = 5 + 1 + 1 = 7$ and $LXXI = L + X + X + I = 50 + 10 + 10 + 1 = 71$. Another rule is that if a symbol appears before a symbol with a larger value then it should be subtracted[16]. So $IV = V - I = 5 - 1 = 4$ and $XCII = C - X + I + I = 100 - 10 + 1 + 1 = 92$.

Now let's learn to convert Hindu-Arabic numerals to Roman numerals. If we want to write the number 1234 into Roman numerals, we first break the number into thousands, hundreds, tens and ones. Using our table that has the conversions from Roman numeral to Hindu-Arabic numeral, then we get that

$$\begin{aligned} 1234 &= 1000 + 200 + 30 + 4 && \text{[Break number down]} \\ &= M + CC + XXX + IV && \text{[Conversions]} \\ &= MCCXXXIV \end{aligned}$$

As you can see a number that we can write down in four symbols now takes 8 symbols to write down in Roman numerals. This emphasizes how troublesome it must have been to perform arithmetic operations on this number system.

You may be thinking about how one should go to write very large numbers over 1000. To counter this problem, in Roman numerals, a dash over a symbol means that the value of the symbol is multiplied by 1000. Though for this project we will not deal with these symbols, below is a chart that has the values of these symbols in case you were curious.

Roman Numeral	Hindu-Arabic Numeral
V	5000
X	10,000
L	50,000
C	100,000
D	500,000
M	1,000,000

4.2 Implementing Roman Numerals into Haskell

In string rewriting, there are equivalence classes, which are transitive, symmetric, and reflexive. This means that two strings are in an equivalence class if we can apply the string rewrite rules and go from one string to the other. Now if we apply this to Roman numerals, we can see that there are some rules we can apply when combining Roman numerals when doing arithmetic. For example, IV and $IIII$ would be in the same equivalence class because they represent the same symbol. However, IV would be the normal form of $IIII$. Likewise, X and VV would be in the same equivalence class because they share a same value, where X is the normal form. As we noted before, this makes implementing a calculator very troublesome since if these rules did not exist, we could just add the symbols with their corresponding value to get the Hindu-Arabic numeral.

4.2.1 String Rewriting Rules for Roman Numerals

before in the previous section of the report, we had the string rewrite rules written down for us, but now we need to come up with the string rewrite rules for Roman numerals. First, it would be easy to go through expanding the Roman numerals. For instance, $IV = IIII$ and $IX = XIIII$. Below are the expanding string rewrites for Roman numerals:

Expanding Roman Numerals
IV→IIII
IX→VIIII
XL→XXXX
XC→LXXXX
CD→CCCC
CM→DCCCC

Now that we have these rules written out, let's implement them in Haskell. We can implement this into a function, which we can call onto our Roman Numeral strings, whenever we want to expand them.

Before we start, we will first need to import some packages and add a line of code; this goes at the top of the program file:

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T -- strict type
import qualified Data.Text.IO as TIO -- print T.Text
import qualified Data.Maybe as M -- maybe monad
```

The first line allows us to use Haskell's Strings as a Data.Text data type. Data.Text is a time and space-efficient implementation of Unicode text, which we will use to write out our strings of Roman numerals. We will use this package instead of Haskell's String data type because Haskell's String is very limited in its features and functions; implementing a rewriting system for Roman numerals would be too daunting of a task. The Data.Text package, which uses strict types-it requires that an entire string fit into memory at once [17]. Data.Text.IO simply allows us to print out a Data.Text data type to command line output. Data.Maybe encapsulates an optional value; a value of type a either contains a value of type a or is empty.

Using this package is a good way to deal with errors, and it is simple error monad [18]. Now let's make our function, *expand*, using this data type:

```
-- function that expands a Roman Numeral
expand :: T.Text -> T.Text
expand x
    | T.isPrefixOf (T.pack "IV") x = T.append (T.pack "IIII") (expand (T.drop 2 x)
    | T.isPrefixOf (T.pack "IX") x = T.append (T.pack "VIIII") (expand (T.drop 2 x)
    | T.isPrefixOf (T.pack "XL") x = T.append (T.pack "XXXX") (expand (T.drop 2 x)
    | T.isPrefixOf (T.pack "XC") x = T.append (T.pack "LXXXX") (expand (T.drop 2 x)
    | T.isPrefixOf (T.pack "CD") x = T.append (T.pack "CCCC") (expand (T.drop 2 x)
    | T.isPrefixOf (T.pack "CM") x = T.append (T.pack "DCCCC") (expand (T.drop 2 x)
    | not (T.null x) = T.cons (T.head x) (expand (T.tail x))
    | otherwise = T.empty
```

This function, takes what we had in the table for the Roman numeral expansions through the use of guards. Guards are a way of testing whether some properties of a value are true or false. So for the first six cases, we are checking if the first two elements of the D.Text data type input match any of the strings we want to expand. If there is a match then we add the expansions of the string and the remainder of the string and call the function expand again; this is how we introduce recursion into our function. The second to last case tells Haskell that if the string is not empty, we drop the first element and do expansion on the rest, and our last case is our base case where we return an empty string.

Now we need to think about how we should implement reducing a Roman numeral; for example, $IIIII$ should reduce to V and VV should reduce to X. Below I list the reductions that we need to consider:

Reducing Roman Numerals
IIII→IV
IIIII→V
VIIII→IX
VV→X
VIX→XIV
XXXX→XL
XXXXX→L
LXXXX→XC
LL→C
LXC→CLX
CCCC→CD
CCCCC→D
DCCCC→CM
DD→M
DCM→MCD

To implement the rules to reduce Roman numerals, we need a whole bunch of rules because of the rules found in writing Roman numerals. If we just were to reverse what we did in the expand function, we would run into some problems with the ordering of the Roman numeral. This is why we have rules such as $VIX \rightarrow XIV$, which puts the Roman numerals in the correct order. However, this may not fix all of our ordering problems so we need another function that will attempt to fix the order between letters. Implementing the rules in the table above in Haskell gives us the functions *reduce* and *romanOrder*.

```
-- function that reduces a Roman Numeral
reduce :: T.Text -> T.Text
reduce x
    | T.isSuffixOf (T.pack "VV") x = reduce (romanOrder (T.append (T.dropEnd 2 x) (T.pack "X")))
```

```
    | T.isSuffixOf (T.pack "IIIII") x = reduce (romanOrder (T.append (T.dropEnd 5 x) (T.pack "V")))
    | T.isSuffixOf (T.pack "VIIII") x = reduce (T.append (T.dropEnd 5 x) (T.pack "IX"))
    | T.isSuffixOf (T.pack "IIII") x = reduce (T.append (T.dropEnd 4 x) (T.pack "IV"))
    | T.isSuffixOf (T.pack "VIX") x = reduce (T.append (T.dropEnd 3 x) (T.pack "XIV"))
    | T.isSuffixOf (T.pack "LL") x = reduce (romanOrder (T.append (T.dropEnd 2 x) (T.pack "C")))
    | T.isSuffixOf (T.pack "XXXXX") x = reduce (romanOrder (T.append (T.dropEnd 5 x) (T.pack "L")))
    | T.isSuffixOf (T.pack "LXXXX") x = reduce (T.append (T.dropEnd 5 x) (T.pack "XC"))
    | T.isSuffixOf (T.pack "XXXX") x = reduce (T.append (T.dropEnd 4 x) (T.pack "XL"))
    | T.isSuffixOf (T.pack "LXC") x = reduce (T.append (T.dropEnd 3 x) (T.pack "CLX"))
    | T.isSuffixOf (T.pack "DD") x = reduce (romanOrder (T.append (T.dropEnd 2 x) (T.pack "M")))
    | T.isSuffixOf (T.pack "CCCCC") x = reduce (romanOrder (T.append (T.dropEnd 5 x) (T.pack "D")))
    | T.isSuffixOf (T.pack "DCCCC") x = reduce (T.append (T.dropEnd 5 x) (T.pack "CM"))
    | T.isSuffixOf (T.pack "CCCC") x = reduce (T.append (T.dropEnd 4 x) (T.pack "CD"))
    | T.isSuffixOf (T.pack "DCM") x = reduce (T.append (T.dropEnd 3 x) (T.pack "MCD"))
    | not (T.null x) = T.snoc (reduce (T.init x)) (T.last x)
    | otherwise = T.empty
```

The function *reduce* is reminiscent to our function *expand* except that we do everything in the reverse order. Notice that in *reduce* we first append the strings and at last we recursively call itself onto the string, whereas with *expand* we first recursively call itself onto the rest of the string and then we append them together. Moreover, the order that we put the cases is important! Imagine if we had a normal numeral in the form of $VVIIIII$; if we had the case of $IIII$ before $IIIII$, then we would possibly get $XVII$ as an answer since it would compute as follows, which is what we do not want:

$$\begin{aligned} VVIIIII &= XIIIII \\ &= XIVI \\ &= XVII \end{aligned}$$

```
-- function that calls onto itself numerous times to implement the order rules of Roman Numerals
romanOrder :: T.Text -> T.Text
romanOrder x
    | let i = M.fromJust (T.findIndex ('I' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'I') x = T.snoc (romanOrder (T.append front (T.tail back))) 'I'
    | let i = M.fromJust (T.findIndex ('V' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'V') x = T.snoc (romanOrder (T.append front (T.tail back))) 'V'
    | let i = M.fromJust (T.findIndex ('X' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'X') x = T.snoc (romanOrder (T.append front (T.tail back))) 'X'
    | let i = M.fromJust (T.findIndex ('L' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'L') x = T.snoc (romanOrder (T.append front (T.tail back))) 'L'
    | let i = M.fromJust (T.findIndex ('C' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'C') x = T.snoc (romanOrder (T.append front (T.tail back))) 'C'
    | let i = M.fromJust (T.findIndex ('D' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'D') x = T.snoc (romanOrder (T.append front (T.tail back))) 'D'
    | let i = M.fromJust (T.findIndex ('M' ==) x),
        let (front, back) = T.splitAt i x,
        T.isInfixOf (T.singleton 'M') x = T.snoc (romanOrder (T.append front (T.tail back))) 'M'
    | otherwise = T.empty
```

The function *romanOrder* finds the index of the desired Roman numeral symbol and then splits the string at that index to get two strings, the front and the back, where both strings don't include the desired Roman numeral symbol. Then, it recursively calls itself onto the concatenated string of front and back and moves the symbol it took out to the end. This ensures that it can be called as many times as possible onto itself.

4.2.2 Implementing Arithmetic over Roman Numerals

Now that we have finished the hard part, we can now implement basic arithmetic using Roman numerals. To implement **addition** let's first do a small exercise. Let's say we want to add IV and IX, then we first would want to expand both of the Roman numerals and then we concatenate them together, put them in the right order, and then reduce the ordered concatenated string.

$$IV + IX = IIII + VIIII \quad \text{[Expand]}$$
$$= IIIIVIIII \quad \text{[Concatenate]} \quad = VIIIIIIII \quad \text{[Apply Order]} = VVIII \quad \text{[Reduce]} \quad = XIII \quad \text{[Reduce]}$$

Doing these operations should suffice to give us the correct answer. Now let's implement this in Haskell.

```
-- function that adds Roman Numerals
addRN :: T.Text -> T.Text -> T.Text
addRN "" n = n
addRN m "" = m
addRN m n = reduce (romanOrder (T.append (expand m) (expand n)))
```

Now we need to implement **subtraction** and **multiplication**. This will be a harder process than it was for addition, since we need the help of another function. We will need to write a function that decrements Roman Numerals. For a Roman numeral that ends with I we just remove the I. One that ends with IV should then become III. A Roman numeral that ends with IX should become $VIII$ It seems like we need to keep on defining cases for each type of ending that is possible, so that is exactly what we will do with the use of guards, as shown in the function below.

```
-- function that decrements a Roman numeral by one
minusOneRN :: T.Text -> T.Text
minusOneRN x
    | T.isSuffixOf (T.pack "I") x = T.dropEnd 1 x
    | T.isSuffixOf (T.pack "IV") x = T.append (T.dropEnd 2 x) (T.pack "III")
    | T.isSuffixOf (T.pack "V") x = T.append (T.dropEnd 1 x) (T.pack "IV")
    | T.isSuffixOf (T.pack "IX") x = T.append (T.dropEnd 2 x) (T.pack "VIII")
    | T.isSuffixOf (T.pack "X") x = T.append (T.dropEnd 1 x) (T.pack "IX")
    | T.isSuffixOf (T.pack "XL") x = T.append (T.dropEnd 2 x) (T.pack "XXXIX")
    | T.isSuffixOf (T.pack "L") x = T.append (T.dropEnd 1 x) (T.pack "XLIX")
    | T.isSuffixOf (T.pack "XC") x = T.append (T.dropEnd 2 x) (T.pack "LXXXIX")
    | T.isSuffixOf (T.pack "C") x = T.append (T.dropEnd 1 x) (T.pack "XCIX")
    | T.isSuffixOf (T.pack "CD") x = T.append (T.dropEnd 2 x) (T.pack "CCCXCIX")
    | T.isSuffixOf (T.pack "D") x = T.append (T.dropEnd 1 x) (T.pack "CDXCIX")
    | T.isSuffixOf (T.pack "CM") x = T.append (T.dropEnd 2 x) (T.pack "DCCCXCIX")
    | T.isSuffixOf (T.pack "M") x = T.append (T.dropEnd 1 x) (T.pack "CMXCIX")
    | otherwise = T.empty
```

With this function, we can now create our functions for subtraction and multiplication. The implementation of these two functions is similar to how we implement successor numbers in Haskell.

```
-- function that subtracts Roman Numerals
subtrRN :: T.Text -> T.Text -> T.Text
subtrRN T.empty n = T.empty
subtrRN m T.empty = m
```

```
subtrRN m n = subtrRN (minusOneRN m) (minusOneRN n)
```

```
-- function that multiplies Roman Numerals
multRN :: T.Text -> T.Text -> T.Text
multRN T.empty n = T.empty
multRN m T.empty = T.empty
multRN m n = addRN (multRN (minusOneRN m) n) m
```

With that we have created a program that does addition, subtraction, and multiplication on Roman numerals. All that is left is testing the program. To test the program, I wrote the following code in the main loop; this should execute with the following command:

```
-- function that multiplies Roman Numerals
multRN :: T.Text -> T.Text -> T.Text
multRN "" n = T.empty
multRN m "" = T.empty
multRN "I" n = n
multRN m "I" = m
multRN m n = addRN (multRN (minusOneRN m) n) n
```

Now just add all the functions along with the import statements to a file and call it **romanNumeralCalc.hs**. You can add test cases to the end of the file in the main loop. Here are my test cases that I used to ensure that my code worked.

```
main = do
    TIO.putStrLn (addRN "IV" "LXVI") -- LXX
    TIO.putStrLn (addRN "X" "C") -- CX
    TIO.putStrLn (subtrRN "C" "X") -- XC
    TIO.putStrLn (subtrRN "CXXIII" "C") -- XXIII
    TIO.putStrLn (multRN "V" "X") -- L
    TIO.putStrLn (multRN "C" "VI") -- DC
    TIO.putStrLn (multRN "VIII" "CXXV") -- M
```

To compile the code simply use the command **ghc -o main romanNumeralCalc.hs** and then run it with **./main**. The program code can be found on Github through this link.

5 Conclusions

Throughout this report, we covered various important topics. For example, we introduced the concept of functional programming, the type of programming language that Haskell is, and some important things about Haskell. These include being a *lazy*, monads, and type classes. Then, we covered some basic syntax of Haskell and created our own functions. In the second part of the report, we went over Programming Language Theories, specifically Hoare Logic. Finally, we made a program to make a simple calculator using Roman numerals using our newfound knowledge of Haskell. Overall, we gained a new way of thinking through the use of functional programming, which could help us further along in our futures.

References

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

[1] Functional programming vs. imperative programming (LINQ to XML), Chapman University, 2021.

[2] Haskell Basics, School of Haskell, 2021

[3] Type Classes and Overloading, A Gentle Introduction to Haskell, Version 98

[4] About Monads, A Gentle Introduction to Haskell, Version 98

[5] All about Monads, Haskell Wiki, 2021

[6] Hoare Logic, Wikipedia, 2021

[7] Notes on Structured Programming, Edsger Dijkstra, 1969

[8] Structured Programming, Wikipedia, 2021

[9] What are Subroutines and Functions, IBM, 2021

[10] The "While" Loop, University of Utah: School of Computing, 2021

[11] An Axiomatic Basis for Computer Programming, C. A. R. Hoare, 1969

[12] The 1980 ACM Turing Award Lecture, C. A. R. Hoare, 1980

[13] Invariant, lojban, 2021

[14] Roman numeral, Britannica, 2021

[15] Hindu-Arabic Numerals, Britannica, 2021

[16] Roman Numerals, Math is fun, 2021

[17] Data.Text, Hackage Haskell, 2012

[18] Data.Maybe, Hackage Haskell, 2012

[19] Expressions vs. Statements, F for Fun and Profit, 2012

[20] , Wikipedia, 2021

[21] Semi-Thue system, Wikipedia, 2021