CPSC-354 Report

Krishna Narayan
Chapman University

December 22, 2021

Abstract

Programming languages are essentially to facilitating the development of the systems and software we use everyday. Functional programming languages like Haskell allow us to explore how we can use computers to adhere strictly to the world of mathematics and thus create more consistent, provable interpreters and compilers. Abstract reduction systems allow us to describe models of computation which programming languages are based around, and theorem proves like Isabelle allow us to prove that these models of computation are reliable. Functional languages can also be used in development of networking systems like chatrooms. Through their concise nature and lack of side effects, these projects become significantly easier to write in fewer lines. They are also easier to debug.

Contents

1   Introduction

This paper is split into three main sections – Haskell, Programming Languages Theory, and a Project.

Haskell is a purely-functional language. Functions in Haskell act similarly to those in mathematics, where a function is a black box with an input and an output and no other "side-effects" occur. The output is the only effect of the function. This is in contrast to a language like C, where memory might be altered which is outside the scope of the output of this function (think global variables, passing by reference). For this reason, Haskell, and functional programming languages in general, are valuable in academia and research for fields such as mathematics. In this section, we briefly explore Haskell's main features, what makes a functional programming language functional, and how we can get started with Haskell. We'll use Haskell to build an abacus virtual machine

Programming langugaes theory entails the theory behind designing and proving a programming language's functionality. We'll explore underlying models of computation and formalize them as abstraction reduction systems, which inherently hold traits (such as confluence and termination) that allow us to make proofs on tremendous graphs of computational relations. We'll explore a theorem prover, Isabelle. Isabelle will allow us to explore how developing an interpreter in a functional language lends itself to induction-based proofs. We'll also find that proving interpreter programs allows us to make deeper claims about higher-level programs to be written in the language we will interpret.

As a final project, we'll explore Haskell's applications to the design of a basic networking system – a chatroom. Applying Haskell to this situation will increase our comfort with Haskell and it's libraries. Furthermore, we'll get an idea of Haskell's conciseness in practice, and test the theory that Haskell is easier to debug due to it's declarative nature. We'll learn a little bit about application layer networking along the way.

2 Haskell

2.1 Programming Paradigms

I think most people start coding with imperative programming languages like C/C++, Python, or Java. I think that specific programming paradigm (imperative programming) might appear to these programmers to be the sole essence of a what a program "is" or can consist of. Haskell might be a first step in leaving Plato's cave for another realm of programming – declarative programming and the related functional programming.

I think before approaching Haskell it is a good idea to establish an understanding of the big picture of functional programming, and even before that to establish an understanding of the big picture of declarative programming and imperative programming.

2.1.1 Imperative programming

Imperative programming is based around the idea that a programs state can be altered. In the English language there exists the imperative and indicative moods of a verb. Take any verb – "eat" might be a good example. Eat is in the imperative mood when it gives a command – "Eat that pancake". This overarching concept of commanding applies to imperative programming – a programmer employs the imperative mood to have the program flow a control flow – "if this, do this[25]. for this, execute this". Here's write a quick piece of code to demonstrate the concept further, and for future reference:

```
int my_func() {
    int z = 0;
    for(int i = 0; i < 3; i ++) { //control flow
        if(y == 5) { //control flow
            HelloWorld();
            z ++; //add 3 to variable z
        }
    }
    return z;
}
```

Note how using words like for and if are indications of the use of commands to describe control flow. This is the essence of imperative programming – describing through code how something should be done.

2.1.2 Declarative programming

Now, introduce another programming paradigm, declarative programming. Declarative programming has a somewhat opposite approach to imperative programming. It's focus is on using declarations of final outcome in the context of other declarations to have the computer complete the program[5]. Lets consider an example of using declarations to accomplish a task. To follow along, write them in a Haskell REPL such as the GHCi:

```
x = 4 -- an expression x evaluates to 4
y = 5 -- an expression y evaluates to 5
x + y -- accomplish x + y
-- 9 is printed to standard output
```

The basic intent of declarative programming is therefore understood – focus on what the program should do rather than how it should do it. In the context of the two previous declarations, the declaration x + y is solved by the computer to be 9, and printed. A side note: Haskell is NOT declarative. It is functional. More later.

A more precise differentiation between declarative and imperative expressions needs to be defined now that we've talked about two programming paradigms composed of one or both: declarative expressions must be

constructed from referentially transparent sub-expressions[5]. Think about a program made up of expressions. Each expression evaluates to something. Any particular expression is said to be declarative in the case that if I were to replace that expression with respective evaluation, the program behavior would not change. So in the earlier example, `x = 4` evaluates to 4. If I replaced the `x` in `x + y` with 4, the program wouldn't change. $4 + y$ is still 9, and the program does nothing more than compute 9. Declarative programming languages only* use declarations – the output is described through declaration, and the computer bases it's computations around that description[7].

2.1.3 Functional programming

Functional programming is similar to declarative programming in that functional programming also does not use control flow directives[5]. That being said, it isn't limited specifically to declarative programming. Haskell is a functional programming language. Haskell programs consist of functions built from other functions and applying to other functions. Haskell is furthermore known as a *purely* functional programming language[6]. It's based around the idea that the functions which compose Haskell can have no affect on the program's state. What does this mean? A good way to think about this is to think about referential transparency. Referential transparency, for expressions, maintained the idea that if I were to replace the occurrence of an expression with it's computed value, the overall behaviour of my program will not change[26]. We can claim that a function is pure if it follows this same behaviour – replacing a functions occurrence with it's computed value will not change the state of the program[26].

What about the C function written earlier? Is this a pure function? What does it do? In essence, it declares an integer z, and prints it based of a number y. And it returns z. So a program wouldn't change if I replaced a function call to `my_func` by the computed value of z, right? Wrong, the program does change – because I call `HelloWorld()` within the function. By calling the function, I alter the state of the program outside of the scope of the function itself. HelloWorld has some effect on *something* that effects my program in some way. This property of referential transparency is no longer valid, because if I were to replace `my_func` with z, I would not have matched the original behaviour of calling `HelloWorld()`.

This is what is known as a **side-effect**, and it is the basis of functional impurity. Haskell avoids this completely. There's a bit more too this, but with a lack of side-effects, Haskell acts with a certain consistency. A lack of side-effects goes hand in hand with the ideas of declarative programming, the idea that things are declared to be something rather than changed to be something. States can't be altered freely. In this way, functions act a lot more like mathematical functions, with strict declarations and goals in terms of computation.

2.2 Lazy Evaluation

Another key aspect of Haskell is lazy evaluation. Haskell will not calculate data until a result is expected[12]. Let's say I called `my_func` from above (assuming it was a pure function), but didn't need the result shown until some later point in the program. This result then won't be calculated until said point. Wow, cool, I guess. It is cool, because if I had a function that needs to perform computations on an infinite amount of data, the values of said computations would only be computed as needed[12]. If I had a list `[1, 2, 3, 4 ... n, n + 1, n + 2]`, and some function that altered each value, my program wouldn't actually perform the any or all computations until results were needed, making data transformation much more efficient.

2.3 Static Typing

If you haven't coded Haskell, you'll probably pick up fairly quickly that Haskell is statically-typed, meaning that value types are known at compile time[10]. This is the same as Java, C, C++. This is in contrast to dynamic typing, where variables generally are reference-holders with no specific data-type associations. That being said, Haskell uses type-inference, so even if you do not specify a type for a certain piece of data, Haskell can probably figure it out[10].

2.4 GHC/GHCi

The topics mentioned above might not be completely understood to the beginner Haskell user, but as they use Haskell and come back to these essential elements of Haskell, a more thorough understanding will develop. For now, coding Haskell might be the best next step in learning. To get started with Haskell, we can use the GHC (Glasgow Haskell Compiler) to compile and run Haskell code[24]. I prefer the GHCi (GHC interpreter) as it simultaneously serves as a Haskell REPL and workspace where I can load and run multi-line programs. Assuming you've installed GHC/GHCi (use google), you can get started with the GHCi by typing `ghci` into your CLI.

We can type code after the `Prelude>` prompt. A quick sidetrack on "Prelude": Prelude itself referes to a standard set of definitions always included with Haskell. Imagine that this a preloaded module that you can use as you run GHCi, or a standard library for Haskell code. Reading these definitions is a great way to start to familiarize yourself with Haskell (see: Tour of Prelude. Note that some definitions might be outdated, as this is for Haskell 98). Like with any standard library, we can leverage these definitions to write Haskell programs. Writing a program to find palindromes, for example, can be easily done by using the `reverse` definition. Think smarter, not harder.

Back on track. In order to use GHCi, it's important to know a few commands. Here are the basics[24]:

- `:q` – quit GHCi. Can also use Ctrl-D.

- `:t` *expression* – Prints the type of expression.

- `:l` *module* – Loads a Haskell module into the GHCi.

There's a few more. You can check out the GHCi manuals for some further reading. There's a bit more technicality to loading modules – the GHCi technically deals with two categories of modules, those loaded and those actually in scope of the prompt. This is a bit complicated and the link above can explain.

For now, you can type Haskell expressions into the prompt. `1 + 2` is a valid expression, and the REPL should respond with `3`.

2.5 Hello, World!

From here, we'll write to programs. The first one prints "Hello, World!" and the second will virtualize an abacus to perform arithmetic. We'll do each in a .hs file, so we can test things from GHCi and also using the compile/run command `runhaskell`.

2.5.1 Code!

To follow along, all the code from this report can be found in this GitHub repository: CPSC354 Report Code. The file is called `helloworld.hs`.

2.5.2 Basic functions

To write our "Hello, World!" function, we need to to establish how functions work, and how they relate to typing. A pure function is a black box that translates input to output without side-effects[6]. We want to write a function that takes no input and returns the a list of characters, "Hello, World!". In Haskell, we can write the following into a file `helloworld.hs`:

```
helloWorld = "Hello, World!"
```

To reiterate, `helloWorld` is a function that returns a character list "Hello, World!". Test out the function in your CLI by first entering the directory containing helloWorld, running GHCi, and loading your file with `:l helloworld.hs`. Then, call the function by typing `helloWorld`. Notice that the REPL spits out our output

in quotes[8]. Technically, our function does not actually provide any output, it only returns a char string. We can tell exactly what the inputs and outputs for a function are by using `:t` with the function name. `helloWorld :: [Char]` shows use that helloWorld returns a char list, as expected.

2.5.3 Using IO

What if we wanted to print this to standard output, rather than implicitly having the REPL spit out the value? We can rewrite helloWorld (or make a new function):

```
helloWorld = putStrLn "Hello, World!"
```

`putStrLn` lets use put our string to standard output. Use `:t` on the new function to test of the typing has changed, and reload the module to test out the new function. You should see that the REPL this time prints the character to output such that it is formatted (no quotation marks).

2.5.4 Main and runhaskell

What if I want to write this in a file such that when I run the file, I run this function? I can do this using a function main, which serves as an entry point for the compiler. Main is a function that takes no arguments and returns to IO, much like our helloWorld function. Let's write a main function for our program, which calls helloWorld:

```
helloWorld = putStrLn "Hello, World!"
main = helloWorld
```

We can do two things from here – use the GHCi to test functions individually as we've done before (we can call main and also helloWorld by itself using the REPL) or we can use `runhaskell` to run our program based off of the main. We can do that by typing `runhaskell helloworld.hs`. Try both.

I think the GHCi is more useful here because we can test function types (you can also see function types using an extensive IDE) using `:t`, but using the compiler by itself is presumably less computer intensive.

2.6 A Basic Abacus

Now we can learn some fun Haskell stuff as we program an Abacus virtual machine/emulator.

2.6.1 Code!

To follow along, all the code from this report can be found in this GitHub repository: CPSC354 Report Code. The file is called `abacus.hs`.

2.6.2 What's an abacus?

An abacus is a calculator used in ancient times. It consists of different rows each with different beads on it. I recommend looking it up for visual reference. Not all abacus are the same, so we'll base our VM off of a seemingly standard layout I've been seeing on the internet. Physically, it consists of 10 rows with 10 beads per row. For rows N indexed from 0, each bead's numeric value is equal to it's associated row 10^N[3]. So a bead on row 0 has a value of one, a bead on row 2 has a value of 10, and so on. Each row represents a "place" in a base 10 digit. The 0 row represents the 1's place, the 1 row represents the 10's place, and so on. This set-up is used to "represent" a number in bead form. We can use this device to perform a few basic arithmetic operations[3]. For the report, we'll focus on addition and multiplication, and only do 3 rows for now.

2.6.3 Theoretical Addition

First, the prerequisites of using an abacus for addition are that the user knows how to count. How do we count a number on an abacus? We can take apart our number by it's digit places, and count them out on the abacus. We can say "shift right" to imply that we are pulling a bead to represent our value. Let's say we have the number 15. In the 10's places we have a 1, so we can shift 1 bead to the right in our row representing the 10's places (row 1). In the 1's place is a 5, so we can shift 5 beads on the row representing the 1's place. The abacus now visually represents 15.

How can we add a number to this number? Say we wanted to add 17 again $(15 + 17)$ such that the abacus visually represented 32. Let's try counting out 17, remembering the our board currently represented 15. To do this again, we'll need beads from the 0 and 1 row to represent digits in the 1's and 10's (1 and 5). As opposed to last time, where the board was we clear, we now only have 9 beads in row 1 and 5 beads in row 0. We count our single bead from row 1 to represent 10, that works. Now, we need to represent the 7, but we only have 5 beads to do it. Let's add the first 5 – now we have 10 beads shifted right, but still need to represent the final 2 of our 7. We can shift the current 10 beads left and exchange for a 10's bead so that we have 3 10's beads, and then get our next 2 from the 1's beads that are now left again. So we have 3 beads in row 1 (30) and we have 2 beads in row 0 (2). $30 + 2 = 32$.

The key two forms of calculation going on here: "bead" calculation via the abacus, and human arithmetic between single digit values. Implementation of human arithmetic will simply be done using Haskell's pre-existing plus and minus operators, so long as the operands are between 0 and 9.

Something important to remember is that the board, between operations, represents an single number. So we aren't adding two numbers at any time, we adding one number of the board to the current board which had been made to represent another number. This is a unary operation.

2.6.4 Haskell implementation

We can break up the abacus addition into a series of smaller functions and data types. First, I want to create a new type Row, which really just holds an int equal to the number of beads on the right of that row. The purpose of this is for clarity. My next step is to create a data type, Abacus, which holds three rows. Here, we can see why it's handy to the programmer to treat ints as rows instead of just as ints.

```
type Row = Int
data Abacus = Abacus Row Row Row deriving Show
```

We also use `deriving Show` here so that we can print our Abacus to standard out. We'll do this eventually as we need to test and finally show results. We can also set up our main that we can test more complicated function constructions.

```
main = do -- use the do keyword to define multiple outputs
    -- your code here
```

At the beginning, however, we can use the GHCi to test individual functions.

Now, we can start thinking about how we want to break down our problem. The essence of the abacus is that you can represent a number using the beads. Representing a number on a blank abacus is like adding a number to 0. For each every count in said number, we shift a bead to the right. We can build our `shiftRight` function such that given an abacus and a row, we can either output an abacus with the proper increment to that row, or we can do another `shiftRight` on the row above, and reset the current row if the current row would theoretically reach it's max of 10 with the next increment.

I don't plan on overcomplicating this program so that it works with a dynamic amount of rows, so we can use pattern matching to run the correct operation on the correct row given a certain parameter. Given a

shift right on the first row, I want to either increment the first row count or, if it's full, reset row to 0 and perform a shift right on the next row. The following code uses guards to define recursive cases.

```
shiftRight :: Abacus -> Int -> Abacus
shiftRight (Abacus r0 r1 r2) 0
    | r0 + 1 < 10 =
            Abacus (r0 + 1) r1 r2
    | otherwise = shiftRight(Abacus 0 r1 r2) 1
```

Line by line, a function shiftRight is typed so that it takes an Abacus and Int as input and returns an Abacus. The Int is the row to be incremented. The second line is the first pattern the computer tries to match based on input. The pattern explicitly requires that row 0 be chosen for incrementation. If it is, we continue into the section using guards. Guards work similarly to pattern matching, and allow us to establish base and recursive case for our function. The first guard outlines that when the row to be incremented can be incremented without overflow, we can return our abacus with minimal issues. The second, which is defaulted to after the first guard (hence, otherwise) calls shiftRight again, increment the row argument to 0 while also reseting the 0 row to 1.

Let's take a look at the second shiftRight pattern:

```
shiftRight (Abacus r0 r1 r2) 1
    | r1 + 1 < 10 =
            Abacus r0 (r1 + 1) r2
    | otherwise = shiftRight(Abacus r0 0 r2) 2
```

This pattern works almost identically to the first pattern, but shifts towards the next row. It also takes 1 as the target row to increment. If that row + 1 is greater than 10, the function is called again, this time targeting the final row and setting the middle row to 0. The final row, while shifting right, cannot reset and transfer its value to the a row above. Therefore, when this pattern has to shift to the next row, it instead returns a full abacus (`abacus 10 10 10`) which ultimately represents the value of 1110.

```
shiftRight (Abacus r0 r1 r2) 2
    | r2 + 1 < 10 =
            Abacus r0 r1 (r2 + 1)
    | otherwise = Abacus 10 10 10 -- return a full abacus
```

Now that we've establishing our pattern for shifting a single bead right for any given row, we can go about writing our main function of actually adding to our abacus. We know that two add a number to the board, we just need to continuously shift right and till we are finished. Representing a number on the board is just adding to the 0 board. What do we want out of this function? We want to provide an abacus and the number we want to add it to it, and we want it to output an abacus with the new value. So our function is typed: `addAbacus :: Abacus -> Int -> Abacus`.

A good way to avoid looping like you might do in imperative programming is by recursion. We want to loop by a count of X and repeatedly shift our abacus right. Only in the 0 case do we not want to shift right. We can plan this out recursively by saying that for our base case, when the number to add is 0, we output the abacus given. For our recursive case, we'll call addAbacus on an abacus shifted right by one, and we'll decrement the number to add by one. This is easier in practice than in explanation:

```
addAbacus (Abacus r0 r1 r2) x -- function takes args here
    | x == 0 = Abacus r0 r1 r2 -- on the case that x is 0, return the original abacus
    | otherwise = addAbacus (shiftRight (Abacus r0 r1 r2) 0) (x - 1)
    -- otherwise, return addAbacus that takes an abacus shifted right one, and an arg -1
```

We can test this out on a zeroed board, to ensure we get the right results. We can do this either using our main or GHCi.

```
main = do
    print $ addAbacus (Abacus 0 0 0) 138
```

This should print `Abacus 8 3 1`. This isn't backwards – remember that the first row represents the 1's place, then the 10's, then the 100's. Okay, we can represent a number on a zeroed board, so how do we add? Hooray – we already can. Abacuses, again, are unary. We just need to supply a new number to the board we already have. Let's add something to 138.

```
main = do
    print $ addAbacus (addAbacus (Abacus 0 0 0) 138) 229))
```

As output, we should get `Abacus 7 6 3`. Awesome, addition seems to work. We can also try with sums larger than the cap (1110) to make sure our final shiftRight pattern is working correctly.

Finally, let's write a function that converts an Abacus to an integer, so that we can read our output better. This function will take an abacus and return an integer. Each row of an abacus already represents an int, so we can multiply those by their respective place (1's, 10's or 100's) and add these values together. This function is really just a one-liner, ignoring typing.

```
abacusToInt :: Abacus -> Int
abacusToInt (Abacus r0 r1 r2) = r0 + r1*10 + r2*100
```

Let's test our Abacus to Int typecasting.

```
main = do
    print $ abacusToInt (addAbacus (addAbacus (Abacus 0 0 0) 138) 229)
```

This should return the value 167. Now we can test our Abacus easily. At this point we can conclude or program, or continue to use it to perform fun abstract calculations.

2.7   Tips and Tricks

I think the Abacus program went by pretty fast, and there a lot of tips and tricks along the way I wish I had pointed out. However, most of them didn't fit in very well. I also wouldn't really classify them as tricks, but moreso things to take note of when making the transition to Haskell.

The first I have is to constantly reinforce the idea that functions do not mutate data. Existing variables are immutable. If I pass in an abacus and a number to a function, say, shiftRight, I don't get the same abacus out. A new abacus is passed out of the function. Think of functions as a black box to a certain extent – pass something in, get something else out. Nothing is being changed.

The second is formatting when calling a function. Let's say I have the function shiftRight. shiftRight is typed as `shiftRight :: Abacus -> Int -> Abacus`. It takes two inputs and returns 1 output. When we call shiftRight, we need to write the arguments such that the compiler can only discern two of them. Use parentheses for this. An example would be `shiftRight (Abacus 0 r1 r2)1`. We use the parentheses to make it clear that Abacus, even with its 3 rows, is 1 argument. This may seem obvious, but miswriting these calls leads to many syntax errors.

The third tip is to take note of the differences between type, newtype, and data. We used type with Row so that we could effectively use Row as a synonym to Int. However, the same constructor was used. We didn't have to write Row in the same way we wrote Abacus. Using newtype would require a new constructor. Since Row didn't consist of anything more than the Int, we want to main Int construction and leverage

related Num operators. data declares a brand new type from a type name and variable types, and uses a constructor. Abacus uses the data keyword so that we can construct an abacus from 3 different rows. Using data types, especially recursive ones, is essential to a project such as making an arithmetic virtual machine.

2.8   Haskell Memory Allocation/Garbage Collection

I want to quickly talk about memory allocation and garbage collection in Haskell. If Haskell isn't altering data and instead creating new versions of data (see tip 1 in tips and tricks), Haskell must be allocating a ton of memory – a ton more than an imperative language. Since there is no explicit memory management functionality along the lines of malloc or free, all allocations and garbage collection is being done by the compiler. How does GHC facilitate this form of programming?

Like Java, objects created on the heap are usually accessed via pointers. However, Haskell is lazily evaluated, so often these objects (or functions treated as objects) might be uncomputed or suspended. These objects are known as thunks. GHC facilitates this form of memory management through a few notable compiler tricks[11].

The first is function in-lining. Function in-lining is when the compiler treats a function call such that the call is replaced with the written text of the function in the actual code. Effectively, this reduces the overhead of a function call.

The compiler uses strictness analysis to avoid thunks. The idea is that if an object does not require an argument for an evaluation, the object can be evaluated immediately and be passed before being destroyed.

3 Programming Languages Theory

3.1 A Quick Introduction

There are languages and then there are programming languages. Before diving into the theory behind programming languages I think we should distinguish the two, and also describes some.

3.1.1 Languages and Programming Languages

What is a language? In the context of human communication, language is a set of phrases or gestures that convey meaning. Language in this case has a structure, *grammar*, and a set of building blocks that make up each phrase or gesture, known as the *vocabulary*. Only once a sentence is grammatically correct can we hope to extract meaning.

What is a programming language, then? A programming language, first and foremost, is a means for a human to communicate with a computer (although we can technically also use programming languages to communicate with each other)[27]. Through this means of communications, we can provide a computer instructions to compute what we want it to. Written programming languages consisting of symbols are most common, but visual programming languages exist as well.

The key to programming languages is that a computer must be able to recognize the "phrase" or "sentence" it is given, and then it must be able to extract meaning/act upon it. How a computer programming language recognizes or understands the input it is given is entirely dependent on it's grammar. How it acts upon meaning generally involves translation into a lower-level language that can be executed at the hardware level.

The general design of a programming language is fundamentally similar to a language used between humans. Like the English languages, programming languages must have a defined syntax to regulate the set of phrases that the computer needs to understand to function properly. This syntax involves a grammar which functions similar to the grammar of something like English language.

3.1.2 Grammar

Because the only spoken language I know is English, I'll first use English to dissect how a very small subset of the English grammar works, and then we can compare it to how the grammar of a programming language might work.

In the English language, the grammar encodes certain words (or structures of words) with specific meanings or roles inside of the sentence[28]. Consider the structure of words, "John went to the store." The English grammar defines this as a sentence. The grammar then defines that this subject is constructed of a subject, "John", and a predicate, "went to the store"[28]. Now we've split this sentence up into two more structures. The first, "John" is a encoded to be a noun, which typically refers to an object. The second, the predicate, consists of "Went", a verb in past tense, "to", a preposition, and "the store", a noun[28].

For the sake of brevity we did not fully deconstruct the sentence, but the idea is that the English language uses a defined grammar to assign meaning to a sentence, and the many parts which make up a sentence. One thing to note is how we approached deconstructing the meaning of this sentence through the grammar. First we analyzed the whole sentence, and then we analyzed the subject and then the predicate. From there, we would continue to analyze the composition of the predicate. Before we can even understand the meaning of the sentence we must recursively analyze the sentence part by part. A computer effectively does the same thing.

Let's think about a more simple language that we could use to communicate with computer. When we think of computation our first instinct might be calculation, so let's build a language that allows us to write calculations. The first thing to do is think of our calculator language as one that consists of a set of expressions, similar to how the English language might be a set of sentences (assuming their grammatical

correctness). Expressions can be made up of other expressions (remember the idea of recursively analyzing a sentence part by part?) – if not, we should define them in our grammar.

3.1.3 Backus-Naur form

We know that our language will be made up of expressions. How do we define how these expressions are formatted, and then decide how each on is encoded or evaluated? We can do this by using the Backus-Naur form (BNF), a specified form of defining programming language grammar that the computer can reference to interpret expressions[27]. An example of BNF might be `Plus. Exp ::= Exp "+"Exp ;`[27]. Our language consists of expressions, and now we've created a new definition in our grammar for a "plus" expression. This idea of defining certain expressions for analysis can be extended to the rest of a calculator's grammar:

```
Plus. Exp ::= Exp "+" Exp1 ;
Subtr. Exp ::= Exp "-" Exp1 ;
Times. Exp1 ::= Exp1 "*" Exp2 ;
Div. Exp1 ::= Exp1 "/" Exp ;
Num. Exp2 ::= Integer ;

coercions Exp 2 ;
```

We've created these abstract of rules (grammar) to govern certain structures of words. Also through BNF, we've defined the the syntax of our language – how words should or characters should be ordered within our language. The decision to have a plus expression be written by a programmer as `1+2` rather than `+ 1 2` is a syntactical decision. The syntax matches directly to the grammar rules.

3.1.4 Concrete and Abstract Syntax

In the context of programming languages, where a computer must "understand" user input, syntax such as `Exp "+"Exp1` is known as *concrete* syntax[15]. Concrete syntax is exactly what the expression will look like to the users[15]. For a computer, this syntax can be difficult to understand when different operations have different levels of precedence, and when operands are intertwined with operands in complicated orders[15]. To circumvent these difficulties, the computer/compiler instead operate using parse trees and abstract syntax trees, data structures created by lexing and parsing expressions using the grammar we created in our BNF[15]. A demonstration is in order to somewhat explain how our grammar is referenced when creating abstract syntax trees.

Consider the expression `1 + 2` and the grammar we've defined. Through the grammar, we know that this is a `plus` operation with operands 1 and 2, which are both integers (`Num. Exp2 ::= Integer ;`). We now create a tree to represent this expression that includes all the important parts of our grammar, such that a computer could easily "linearize" or convert the contents of a tree into a series of linear instructions for computer instructions. The tree might look like this:
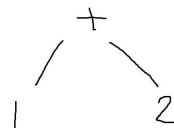

3.1.5 Syntax and Semantics

An important concept of programming languages and language in general is the differentiation between syntax and semantics. In the English language, English speakers know when a sentence is syntactically

correct but not semantically correct. Take the sentence ""Colorless green ideas sleep furiously", written by Noam Chomsky. The sentence is syntatically correct, but no English speaker could reasonably derive any idea from this. The same idea applies to programming languages – syntactically correct expressions can also be semantically incorrect.

3.2 Abstract Reduction Systems

Programming languages and language exist as complex relations between syntax and semantics. In the design of programming languages, an especially important concept is the idea of rewriting, or methods of rewriting expressions in various ways. Rewriting is quite literally the act of zero-to-n steps of computation, and many declarative programming languages and theorem provers leverage this idea to implement functionality. Rewriting introduces the concept of an Abstract Reduction System (or Abstract Rewriting System), which is also known as an ARS[14]. An ARS comprises the set of all possible expressions and the relations between these expressions. Each relation corresponds to a computation/or a reduction between all possible computations. The abstract reduction system correlated to our language is the set of all A allowed by our language, and also the relations between these reductions. Using this mathemetical theorization of our language, we can start to making proofs about our programming language and programming languages in general.

The defining propeties of an abstract reduction system are directly related to how computations form different graphs within these systems. The best way to approach definitive properties of abstract reduction systems such as confluence and termination is by visualizing the ARS as a graph. Let's think of the language of arithmetic as an ARS, where A is the set of arithmetic expressions, and R are the reductions/computational steps relating each one. I could visualize a short portion of the ARS as:
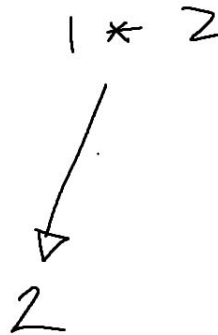
$1 * 2$ and 2 are members of set A, and the arrow represents a single step computation which is a member of the set of relations of all members in A. This set is set R. Understanding this visualization is useful for understanding potential properties of ARS such as confluence, termination, and the existence of normal forms.

3.2.1 Normal Forms

A member of set A in an ARS is said to be a normal form this member can no longer be reduced[29]. In our earlier example, 2 can no longer be reduced by the rules of arithmetic, and therefore it is in normal form. $1 * 2$ is not in normal form, because arithmetic specifies multiplication as a form of reduction.

3.2.2 Confluence

In the previous ARS, there is only 1 possible reduction of the expression $1 * 2$. What if I had a more complicated expression which branched into multiple other expressions through multiple reductions? Take

the expression $1 * 2 * 3$, for example. The ARS might look like:
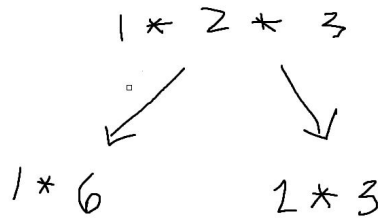
We've now branched off into two different expressions that are part of set A. However, if we were to continue discovering our ARS by reducing these expressions, we would eventually (in this case, immediately) find that our ARS branches together into a single expression:
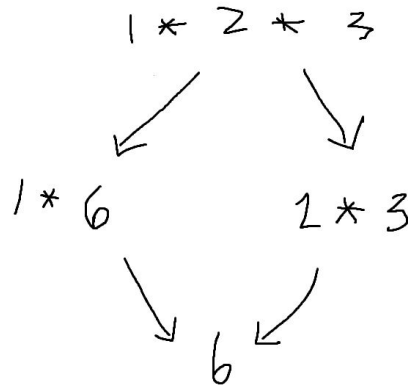

This suggests that the expressions $1 * 6$ and $2 * 3$ are joinable. Expressions are joinable when they reduce to the same element over an arbitrary amount of computational steps. In this case, both expressions reduce to the same element (6) after a single step. This expression is in normal form.

We can describe an ARS as confluent when for any element of A that reduces to two distinct elements, these two distinct elements will be joinable[29]. In other words, the visual structure of the ARS drawn above should apply to all elements. All peaks should form a valley, all expressions that reduce into multiple other expressions will eventually join back together into a single expression (in normal form). If this isn't the case, the ARS is not confluent.

3.2.3   Termination

Termination could be considered to be a simpler concept. An ARS is terminating if every element has a normal form, and there is no infinite chain of reductions[30]. Imagine a language with only the letters x and y and I define the rules of my language so that all x reduce to y and y reduce to x. There are no normal forms, so for any given element in our system, said element could never reduce to a state where reduction is no longer.
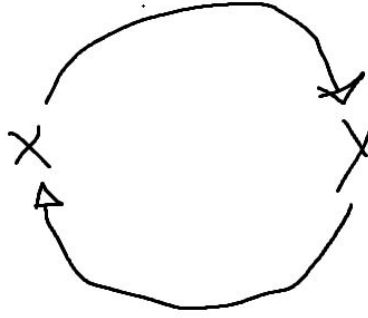
The drawing above demonstrates the infinitely-circling pattern of reduction. This ARS is therefore not terminating. Arithmetic is terminating, however, given that there are described normal forms (numbers).

We know the how these systems of expressions are defined, so we can roughly observe that termination might hold true for one system and not the other. However, in general, it's necessary to prove that an ARS does not terminate. Proving that an ARS is terminating means that one can find a measure function [30].

On measure functions - imagine expressions a and b in an ARS. If said ARS terminates, there is a measure function $\phi : A \to \mathbb{N}$ such that when a reduces to b, the measure function would provide the implication $\phi(a) > \phi(b)$ [30]. This measure function assumes the idea that reduction can be measured in some abstract way, and therefore the implication is that the reductions doesn't exist in the scope of an infinite chain where reductions can't be measured in reference to a start and an end.

3.2.4   Invariance (and Invariants)

ARS's can be defined by their own invariants, or specific properties that do not change at any point in our system [21]. Imagine even the most simple non-terminating system where the string a reduces to a, the single rule being $a \to a$. One could intuitively describe an invariant of this system to be that the the number of a's doesn't change, or the number of elements (length) doesn't change. Imagine a more complex, terminating system where $aa \to a$, and $bb \to b$. An invariant here might be that an expression with only b's never changes the number of $a's$, or vice versa.

3.2.5   ARS Conclusions

While there are more properties of ARS to be discussed, the ones above provide a detailed picture of how we can draw conclusions about complex computational models, and make proofs about them. We can describe languages as confluent and terminating, and thus relate them to certain invariants and use these traits to prove certain functionalities to given languages. The next section, which will introduce Isabelle, will even more thoroughly demonstrate the value of analyzing languages for these attributes.

3.3   Isabelle

A key point of programming languages and programming in general is being able to prove the properties of programs are inherent or correct. For example, in creating something like an interpreter, we need to prove that properties like constant folding, simplification, and substitution always work correctly. Otherwise, we can't prove that any program that we write in our language is correct.

Theorem provers like Isabelle and can automated proofs on functional programs. Usually, in order to fully prove a program, we might need to use induction such that for every input our output is correct. Isabelle

let's us write theories, which are essentially programs, and then we can make claims about said programs (lemmas) and automate their proofs by simply determine what element of our program we are using induction on.

3.4 Some notes on Isabelle

Before delving into programs, I have some thoughts to share about the Isabelle programming IDE and Isabelle itself. First, Isabelle's programming IDE provides VIM emulation, which is extremely handy. Furthermore, Isabelle uses sections for proofs, and this can serve as a great way to split up exercises. The programs were below were done as a result of exercises, and as one can see in the code, they're easily organized by sections. Third, writing functions themselves in Isabelle is very similar to basic Haskell in terms of syntax. However, there are variations, so using Haskell as a fallback to at first make sure that your program is functionally sound is a good idea. Then, you can delve into how the Isabelle syntax might be incorrect. Work through programs line by line, and take full advantage of Isabelle's line by line evaluation. That's enough ramble for now, but maybe these observations are helpful for following the next few sections.

3.4.1 Isabelle and IMP

At this point, let's open up Isabelle and starting writing some functions to interpret a language, and then let's prove these functions. Let's say we want to write proofs on a very simple version of the language IMP. In this case, IMP consists of variables, integers, and for now a simple Plus expression that contains two expressions [13]. Let's create our expression datatype `datatype aexp = N int | V vname | Plus aexp aexp`. V is function that takes a type `vname`, for variable name. This type doesn't actually exist but we want it to be a synonym for a basic string. So above this line, we'll add `type_synonym vname = string`. In addition, we'll include a type synonym for integers called `val`, and we'll include a function that returns the value of a variable given the name: `type_synonym state = "vname \<Rightarrow> val"`. Notice the use of quotations here not define strings but to chain together sequences of commands. We now have our structural foundation for IMP:

```
type_synonym vname = string
datatype aexp = N int | V vname | Plus aexp aexp

type_synonym val = int
type_synonym state = "vname \<Rightarrow> val"
```

3.4.2 Evaluating arithmetic expressions

Let's write a simple function that we can use to evaluate arithmetic expressions. This is relatively simple. We just need to match each type of expression, and for `Plus` we can leverage Isabelle's existing `+` operator. One thing to note is that we want to keep a state with our function while it's iterate such that on the account of a variable, we can extract the value of that variable. In the case of a simple natural number, our function will disregard state and simply evaluated to the natural number. However, in the case of a variable, the state will be output, with the variable name applied to that state such that the value of the variable is returned. With this in mind, we can write our function `aval`:

```
fun aval :: "aexp \<Rightarrow> state \<Rightarrow> val" where
"aval (N n) s = n" |
"aval (V x) s = s x" |
"aval (Plus a1 a2) s = aval a1 s + aval a2 s"
```

It's worth again emphasizing that Plus is split into two recursive calls of `aval`, which are summed using a pre-defined `+` operator.

3.4.3 Constant Folding

With evaluation in place, the first program we want to write and prove is one that performs constant folding. Constant-folding is otherwise known as replacing sub-expressions by their reduced forms [13]. Something to consider is that the only expressions that can be folded are `Plus` expressions holding two natural numbers, as these numbers can be added by the `+` to make a simpler expression.

Let's write another function in Isabelle to fold constants. We first need to decide what cases we need. To reiterate, in the case of an integer, return an integer, as we can't fold a single integer. The same applies to the a single variable. Therefore:

```
fun asimp_const :: "aexp \<Rightarrow> aexp" where
"asimp_const (N n) = N n" |
"asimp_const (V x) = V x" |
```

The output in the case of a `Plus` expression depends on on the sub-expressions of plus. Both expressions are natural numbers, then we can fold constants and return the sum of these two numbers. Otherwise, we'll just return the Plus statement as it is:

```
fun asimp_const :: "aexp \<Rightarrow> aexp" where
"asimp_const (N n) = N n" |
"asimp_const (V x) = V x" |
"asimp_const (Plus a1 a2) =
  (case (asimp_const a1, asimp_const a2) of
    (N n1, N n2) \<Rightarrow> N(n1 + n2)|
    (b1, b2) \<Rightarrow> Plus b1 b2)"
```

Something to note about this third case is that `asimp_const` is recursively applied to the arguments of the Plus symbol. This way, the entirety of our expression is folded through recursion. Sub-expressions eventually return base cases, and these base cases are then registered in our case patterns. The formatting of case statements makes the program look more complicated than most recursive programs, but the idea is it exactly the same except that we split our evaluation in our third pattern.

Now that we've proven our function, we can attempt to it's functionality using Isabelle. One lemma we can prove about our program is that if we were to evaluate a program, it would be evaluated equivalently to the same program had it been applied to the constant-folding function. Or, we propose that `aval (asimp_const a)s = aval a s`. In other words, we want to prove that `asimp_const` doesn't change the meaning or semantics of our original program `a`. To prove this, we would use induction on a program `a`. Let's consider proving the base cases:

```
"asimp_const (N n) = N n" |
"asimp_const (V x) = V x" |
```

These can be trivially proven, given they are cases where `asimp_const` does effectively nothing, as there are no sub-expressions to simplify. The case-statement, however, cannot be trivially proven:

```
"asimp_const (Plus a1 a2) =
  (case (asimp_const a1, asimp_const a2) of
    (N n1, N n2) \<Rightarrow> N(n1 + n2)|
    (b1, b2) \<Rightarrow> Plus b1 b2)"
```

This case statement serves as a basis for the inductive hypothesis that `aval (asimp_const a_i)s = aval a_i s` where `i=1,2`. Essentially, for either of the expressions in the plus statement, we can hypothesize that each of these programs will be semantically equivalent after constant folding. Then, we can prove our case statement by running through the steps of deriving an expression in which we can substitute our inductive

hypothesis to prove that our program works for all cases of a. We won't do this, because we have our hands on a theorem-prover!

To first declare our lemma in Isabelle, we can write lemma `"aval (asimp_const a)s = aval a s"`. We know that we are performing induction on `a`, so we can then start the automated proof by write `apply (induction a)`. This sets up the groundwork for automation, but something to note is that for automation to work in this case, the proof needs to be split by case for our last pattern. This can be explicitly hinted at by writing `apply (auto split: aexp.split`. At this point, we can test our automation by typing done. Altogether, the proof looks like this:

```
lemma "aval (asimp_const a) s = aval a s"
  apply (induction a)
  apply (auto split: aexp.split)
  done
```

Isabelle shows that the goals of the proof are to inductively prove each of the patterns. The third one is split into two cases. Isabelle confirms that the proof is sound. Assuming Isabelle itself has been proven, we know that our program for folding constants doesn't change the semantics of expressions, and could be ready for production or deployment in a real interpreter.

3.4.4 Optimal

Let's write another program that returns a boolean rather than an arithmetic expression. This program will be called `optimal`, and it will determine whether an expression is in an optimal state. An optimal state can be described as one where all expressions are reduced as much as possible, or all sub-expressions are folded where possible [13]. This is synonymous to the normal state of an expression.

An important mindset in writing recursive programs is one where you attempt to represent your exact logic through code, without thinking things out to be a series of instructions. For example, if we theorize our new program `Optimal`, what does it return, and in what cases? What will `Optimal` do in the most literal sense? Here's a list of thoughts:

- In the case of an integer, optimal will return true. (This expression can no longer be optimized).

- In the case of a variable, optimal will true. (This expression can no longer be optimized).

- In the case of a Plus, optimal will be true only if both sub-expressions are true, or optimal.

We can directly implement these ideas or declarations about the goals of this program into our Isabelle code. The base cases are trivial to implement:

```
"optimal (N n) = True" |
"optimal (V x) = True" |
```

Implementing the case-statement for `Plus` is slightly more difficult only in that there are multiple combinations of true and false for sub-expressions a1 and a2. We could either write out all of the cases exactly how they should be, or we could supplement our functions with other functions that read the combinations elsewhere. For simplicity, we'll write all of the cases down. Our function will therefore read:

```
fun optimal :: "aexp \<Rightarrow> bool" where

"optimal (Plus (N n1) (N n2)) = False" |
"optimal (Plus a1 a2) =
  (case (optimal a1, optimal a2) of
  (False, True) \<Rightarrow> False|
  (True, False) \<Rightarrow> False|
```

```
(False, False) \<Rightarrow> False|
(True, True) \<Rightarrow> True)"
```

How can we prove that `optimal` is actually correct for all programs in declaring that each of them are or are not optimal? For one, we know that our function `asimp_const a` will return optimal expressions where the semantics don't change from the original expression. Therefore, every program applied to `asimp_const` should be optimal. We can start with this lemma: `optimal (asimp_const a)= True`. We again want induction on a program `a` in this case, and we need to be split our proof according to the case statement.

```
lemma "optimal (asimp_const a) = True"
  apply (induction a)
  apply (auto split: aexp.split)
  done
```

Isabelle can solve all of the appropriate sub-goals, and we have proven that our `Optimal` function works for all programs in our language.

3.4.5 Full simplification of expressions

A more difficult function to implement and prove is one that performs full simplification of an arithmetic expression. In this case, full simplification might imply that we perform constant folding such that or program is optimal, but we also move variables and integers such that integers are summed regardless of where they are in the expression. This is synonymous with the property of associativity. Thinking about associativity is important to understanding the declarative mindset behind implementation.

We can establish our bases cases first, which are identical to the constant-folding base-cases:

```
"full_asimp (N n) = N n" |
"full_asimp (V x) = V x" |
```

Again, `Plus` will involve a case statement, only this time we'll be simply rearranging all cases where two natural numbers exist. Something to note is that we don't need to worry about more than two natural numbers, as these will be simplified through recursive calls similar to those in previous functions. We'll also include a pattern such that if a `Plus` expression has two natural numbers, they're summed. This functionality is identical to that of aval. In total, the function looks like this:

```
fun full_asimp :: "aexp \<Rightarrow> aexp" where
"full_asimp (N n) = N n" |
"full_asimp (V x) = V x" |
"full_asimp (Plus a1 a2) =
  (case (full_asimp a1, full_asimp a2) of
    (N n1, Plus (N n2) (V x)) \<Rightarrow> (Plus (V x) (N (n1 + n2)))|
    (N n1, Plus (V x) (N n2)) \<Rightarrow> (Plus (V x) (N (n1 + n2)))|
    (Plus (V x) (N n1), (N n2)) \<Rightarrow> (Plus (V x) (N (n1 + n2)))|
    (Plus (N n1) (V x), (N n2)) \<Rightarrow> (Plus (V x) (N (n1 + n2)))|
    (N n1, N n2) \<Rightarrow> N(n1 + n2)|
    (b1, b2) \<Rightarrow> (Plus b1 b2))"
```

This is another occurence where we effectively have the same patterns multiple times, but we simply have to cover all arrangement of terms here.

We want to prove that our when an arithmetic expression is entered into `full_asimp`, the expression that is output evaluates (check this using `aval`) to the same expression that is input.

```
lemma "aval (full_asimp a) s = aval a s"
```

```
apply (induction a)
apply (auto split: aexp.split)
done
```

We can prove this by induction on `a`. We also need to use a split to guide Isabelle in either of the directions of the case statement.

3.4.6 Proving substitution

Substitution becomes less difficult as a programmer embraces the narrative of declarative programs. The implementation exists exactly as one might expect. The logic operates as follows:

- If there is no variable that can be substituted, return the expression.

- If there is a variable that matches the variable specified for substitution, replace the variable with the value to substitute.

- If we're operating on a plus statement, apply substitute to the components of the plus operands.

```
fun subst :: "vname \<Rightarrow> aexp \<Rightarrow> aexp \<Rightarrow> aexp" where
"subst x a (N n) = N n"|
"subst x a (V y) = ( if x=y then a else (V y) )"|
"subst x a (Plus p1 p2) = Plus (subst x a p1) (subst x a p2)"
```

Now, we want to prove the **substitution lemma** that an expression should evaluate to the same normal form regardless of when substitution takes place, or `aval (subst x a e)s = aval e (s(x := aval a s))` [13]. We prove this with simple induction on e.

```
lemma "aval (subst x a e) s = aval e (s(x := aval a s))"
  apply (induction e)
  apply (auto)
  done
```

3.4.7 Isabelle Conclusions

For now, we can wrap up on proving the various facilities behind IMP's interpretation mechanisms. We demonstrated how Isabelle takes full advantages of the nature of functional programming languages and their purity. I think learning Isabelle and proving various features of languages is a great introduction to the intersection between functional programming and mathematics. Furthermore, proving a programs validity is a neccesary step to developing a proper foundation for the development of systems and applications.

4 Project

4.1 A Chatroom in Haskell

For a final project, I decided to explores the concise nature of functional programming by taking on a task generally approached with imperative languages – creating a multi-threaded chatroom. To do this, I explored sockets and threading in Haskell and developed a client-server architecture, much like I would in a language like C++ or Java.

4.1.1 Code!

To follow along, all the code from this report can be found in this GitHub repository: haskell-chat. There are two files – one opens a server and one opens a client, and are named respectively. Each one has the word "byte" prepend to indicate that these programs use byte strings, rather than socket-to-handle based handlers for strings. Handlers make programming easier, but aren't as low-level. It's worth noting that the client and server are separate programs. Lot's of examples of Haskell socket programming use a single file. I figured this would be a more flexible setup for user use.

4.1.2 Development Setup

I highly recommend using a terminal multiplexer for this project, as debugging multiple programs that interact with one another requires flexibility in switching between terminals. I used `TMUX`.

I chose not to use a project builder (`stack`). While this project does require dependencies for concurrency and networking, using `stack` seemed unnecessary for a single dependency install (not including GHC). Furthermore, `stack` did not lend itself to using multiple executables, as it requires a main module for a single program, when I needed different mains established for different executables. The whole operation became paradoxical.

VSCode's Haskell extensions were extremely valuable for development. Haskell's official extension make use of the Haskell language server, which provides valuable input on typos, errors, and required inputs. As I added certain functions provided by certain libraries, the language server even wrote the necessary boilerplate code for me, which was extremely valuable.

4.1.3 Project Background

As stated, making a chatroom requires a server and a client program. The server program will facilitates any clients that tap into the server program.

Clients connect to the server using sockets. Sockets are a network abstractions for internet connections between processes [16]. They exist as a triad of three key identifiers for how and where a connection should occur between programs. The first is an IP address. An IP address is an identifier for a host network interface (such as a WiFi card) that is connected to the internet. The second is a port number, which represents an communication endpoint to a specific process. Third is a transport protocol, which holds specifications for how data should be transferred (UDP is an unreliable transport protocol, TCP is reliable, etc.) [16].

After a client establishes a connection to the server using a socket's job, it's the server's job to handle requests by the client. In this case, we want to make a specific chat program. Therefore, the server should be able to take messages sent by the client and broadcast them to anyone else connected to the server (broadcast). Our client program should be able to receive messages from the server, and facilitate user input to send as a message.

As a final note on the project, both the server and the client need the ability to both read from clients and write to clients concurrently. This way, a client can still receive messages even while being prompted to send a message. We implement concurrency by using threads. Threads are allocated from process memory

and provide pseudo-concurrency in comparison to something like a multi-processing, where processes. That being said, threads can access shared memory while processes cannot.

4.2 Haskell Libaries

Assuming we know standard implementations such as getting user input and printing to the screen, how will we implement our more complicated requirements, such as sockets and threading? We can use Haskell libraries to accomplish these tasks.

4.2.1 Sockets

We want to get sockets set up, so let's set up a basic socket program between a single server and a client. To do this, we are going to use `network`, a networking library for Haskell. We can install this through Haskell's package manager by typing `cabal install network`, where `cabal` is the package manager. When we have a program that needs sockets (both client and server programs) we can include `import Network.Socket` into our file.

Now that we have sockets at our fingertips, we can go over some of the core functionality in the library. Here's a short list of those core functions with brief descriptions [20].

- `socket :: Family -> SocketType -> Protocol Number -> IO Socket` - used to create a new socket. The family referes to the address family. In our case, the simplest family to use is the address family of IPv4. The socket type refers to type of data transfer. Here we can specify which transport protocol we want to use, depending on our choice of `stream` or `datagram`. Finally we can select a protocol number. The default works (zero).

- `bind :: Socket -> SocketAddr -> IO ()` - used to bind a socket to a specific address, which consists of a port number and a host address. By binding sockets to well-known addresses, servers can be consistently accessed by sockets at more ephemeral addresses.

- `connect :: Socket -> SocketAddr -> IO ()` - used to connect a remote socket address (such as one we've binded a socket to). Clients will use this to connect to a server.

- `accept :: Socket -> IO (Socket, SockAddr)` - used to connect a remote socket address (such as one we've binded a socket to). Clients will use this to connect to a server.

These functions are used to create sockets, and establish basic connections between them. With connections established, we also need some form of data transfer. There are a few ways to do this. One of the simpler ways to facilitate data transfer is to turn a socket into a "handle" using the pre-existing `System.IO` library. Handles ultimately act as an abstraction for what we all know is ultimately byte transfer, and will therefore make our job easier. However, we might want to think about scalability and expanding our program (maybe implemented the IRC protocol) so we'll focus on simple byte transfer without handles for now. Luckily, `Network.Socket` provides a small library `Network.Socket.ByteString` that faciliates sending strings of bytes between sockets. Here's a function rundown.

- `sendAll :: Socket -> IO Int` - sends a byte string from a given socket. In order to receive from this socket, another socket will have to be connected and also having a blocking `recv` call.

- `recv :: Socket -> Int -> ByteString` - receive the specified amount of bytes (`Int` in to the socket. This call is blocking, meaning the program will not continue pass this function call until the bytes have been received.

These functions are all we'll need to establish necessary socket connections in Haskell and to send data between programs. At this point we can talk about basic concurrency in Haskell.

4.2.2 Threading + Concurrency

Concurrency/threading in Haskell is easier to implement. First, `Concurrent` doesn't require an install using the cabal package. We can import it using `import Control.Concurrent`. The main function we'll need to use is `ForkIO` [17]:

- `ForkIO :: IO -> IO ThreadId` - Given some sort of input/output function, run said function in a given thread and return the Id of the thread. We can chose whether or not we actually want to use the ThreadId.

This gives a better on how we might organize our project. The server might split a read and write function into two individual threads for each client that connects to our server.

4.3 Server Programming

Now we can program our server. We need to first establish a network connection for our server using a socket. We can do this using some of the basic socket functions described about. We can create our socket in main.

```
main :: IO ()
main = do
  sock <- socket AF_INET Stream 0 -- create a new socket
  setSocketOption sock ReuseAddr 1 -- instant address reuse
  bind sock (SockAddrInet 4000 0) -- bind our socket to a well-known port
```

The function `setSocketOption` is used but wasn't listed above. While not essential to socket programming, `setSocketOption` makes debugging much easier because it allows ports to be instantly re-used [16]. For example, if I ran my server with a socket on port 4000, I might want to simply use `Ctrl-C` to shut the server down and go back to coding. However, if the socket wasn't closed, the port will still be in use. This way, I allow reuse of the port address, without having to worry about properly closing the socket.

The choice to use port 4000 is mostly random, except that it's not within the main range of ports dedicated to specific applications (examples being HTTP at port 80, or SSH at 22). The default protocol number was also used.

Now that we've established our socket, we want to continuously open ourselves up to connections from other sockets. We can do this using a loop. We can create a loop by repeatedly calling a function. In a way, this recursion without a base case, which is fine, because in theory we want our server to run forever. Given that this loop listens for clients, we'll call it a listenLoop:

```
listenLoop :: Socket -> IO ()
listenLoop sock =
  listen sock 2 -- listen for up to 2 socket connections
  putStrLn "Listening on port 4000..."
  (conn, _) <- accept sock -- accept a socket
  putStrLn "New connection found."
  sendAll conn $ C.pack "Welcome to server -- type '!quit' to quit" -- send msg to socket
  listenLoop sock -- loop
```

Important to note is that `accept` is blocking, so this loop won't proceed until a connection is found. Accept also returns both a `Socket` and a `SockAddr`. We only need the `Socket`, so that is all we'll extract from the output of the call to `accept`.

Also notable is the use of `C.pack`. `C.pack` translates our string (in this case, `"Welcome to server -- type '!quit' to quit"`) into bytes so that sendAll can transfer bytes to all socket connections.

We'll append a function call to `listenLoop` to the end of our main:

```
main :: IO ()
main = do
  ...
  listenLoop sock
```

No we can think about how we might receive bytes from an incoming connection. We'll call this function `dealWithConnection`.

```
dealWithConnection :: Socket -> IO ()
dealWithConnection conn = do
  msg_bytes <- recv conn 1024 -- prepare to receive 1024 bytes
  let msg_string = C.unpack msg_bytes -- unpack bytes into a string
  case msg_string of
    "!quit" -> do -- in the case of a quit message
      putStrLn "server recv quit, close connection"
      sendAll conn $ C.pack "goodbye"
      close conn
    _ -> do -- otherwise
      putStrLn("1") -- an ack that message was received (only on server)
      dealWithConnection conn
```

For now, this function continuously looks to receive bytes from a connection. A received message is unpacked, and the function then decides what to do based on the message. If the message is a request to quit the program (decided to be `"!quit"`), a goodbye message is sent back and the connection is closed. Otherwise, for now, we'll simply acknowledge that we received a message. We then call the function again to repeat this process.

Here, things become a bit more complicated. First, we need to make sure that this function is running concurrently for every client we have. Clients need to be served concurrently. To do this, we can call this function from our listenLoop within a call to `ForkIO`. At the end of our listen loop, we'll write this call:

```
listenLoop :: Socket -> IO ()
listenLoop sock =
  ...
  forkIO(dealWithConnection conn chan ident) -- pass it to all threads
  listenLoop sock -- loop
```

Next, we need to broadcast the message received in our `dealWithConnection` function. It seems extraneous to keep track of each client connection and then loop through each one to relay what was sent. Instead, we can use a `channel`, an abstraction of shared memory between threads provided by the `concurrency` library. Channels will allow us to read and write to shared memory between threads.

One way to use a channel for broadcasting to have the server put any messages received from the client into the channel, while running a concurrent thread which reads from this channel. We can try to implement this, but the first thing to remember is that we want to work with a single channel. Therefore, we should create our channel in main, and then pass into our listen loop, which will then pass it to threads that receive messages from connections. We then want `dealWithConnection` to spawn another thread specifically for reading this channel concurrent to dealings with the channel.

```
main = do
  ...
  bind sock (SockAddrInet 4000 0)
  chan <- newChan -- CREATE A NEW CHANNEL
  listenLoop sock chan
...
```

```
listenLoop sock chan = do
  ...
  forkIO(dealWithConnection conn chan) -- pass it to all threads
  listenLoop sock chan

...
dealWithConnection conn chan ident = do
  reading_thread <- forkIO(readChannel conn chan ident)
  ...
  killThread reading_thread
```

We store the output thread id into a variable `reading_thread`, that we can later kill. Our thread spawn is based around the function `readChannel`, which is we'll implement to read from the channel, and send to the socket connection related to the given thread.

```
readChannel conn chan = do
  msg_from_chan <- readChan chan
  sendAll conn $ C.pack msg_from_chan
  readChannel conn chan
```

`readChannel` continues to loop until the `dealWithConnection` is finished, at which point the thread is spawened again (unless the socket is disconnected). This ensures that the thread is always appropriately killed.

At this point, we've finished the basic functionality of the server program.

4.4 Client Programming

The client is similar to the server, in that we need to create a socket. We then need to connect it to our well-known port. Since I'm running my client and server on the same computer, I don't need to worry about IP addresses, as both program run on the same IP address.

```
main :: IO ()
main = do
  sock <- socket AF_INET Stream 0 -- create a new sock
  connect sock (SockAddrInet 4000 0) -- connect to well known port
  forkIO(recvLoop sock) -- receive messages in a separate thread
  sendLoop sock -- send messages in main thread
  close sock -- close when loops finish
```

From here, we just need to implement simple receive and send loops, similar to those from the server:

```
sendLoop sock = do
  msg_string <- getLine
  sendAll sock $ C.pack msg_string
  case msg_string of
    "!quit" -> putStrLn "exiting..."
    _ -> do
      sendLoop sock

recvLoop sock = do
  msg <- recv sock 1024
  print("Received >>> " ++ C.unpack msg)
  recvLoop sock
```
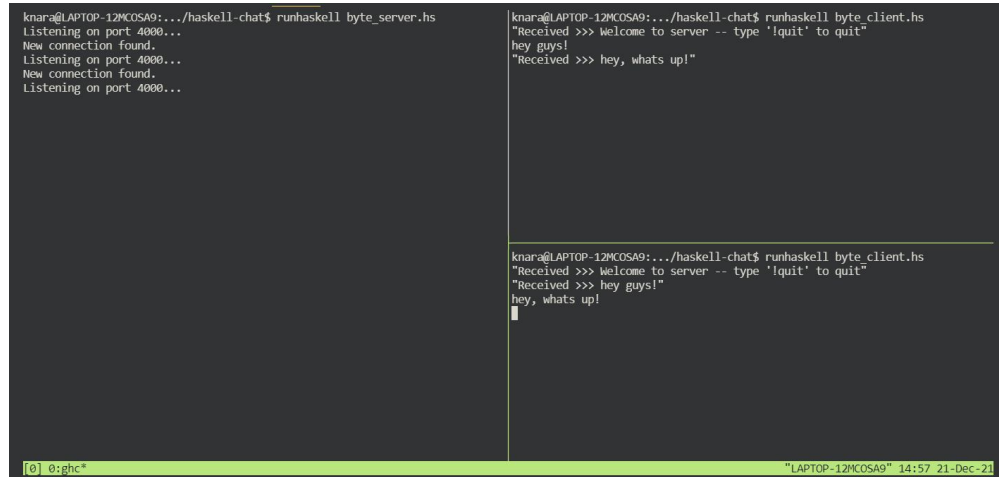
With that, we can start our server and then our clients. Clients can then talk to on another. For a slightly more complicated implementation using user identification, check the GitHub code.

4.5   Final Remarks

In conclusion, the project was mostly a success. Here's a screenshot of the advanced client-server architecture in action. In this situation, the server uses a unique identifier for each connection to ensure than no connection receives its own messages.

Unfortunately, there are still issues with reliability. I attribute to this issues with concurrency. My goal to implement atomics to fix this issues, if Haskell will allow it. I might try and use the `dupChan` again, but I've had quite a few issues with messages being broadcasted 4 or 5 more times than they should've been.

While the syntax is more concise, I would not argue that Haskell does not make this program easier to write (for me). I found it difficult to understand how to approach a problem oriented around objects (sockets, clients, and servers) with a language more equipped to handle mathematics. That being said, Haskell's syntax is similar to Python in it's clarity and conciseness. Writing this project has given me a taste for Haskell's strengths, so I'll continue to use it when I can. Some argue that Rust makes you a better programmer – I'd argue that Haskell might make someone a better mathematician. However, I wouldn't say that functional programming is completely coherent with computer programming as a concept.

5   Conclusions

As we enter the age of machine learning and cryptocurrency, it's no suprise that functional programming languages, which lend themselves to program proofs, are becoming for fashionable for industry-professionals and academics alike. Haskell's mimicry of the world of mathematics is arguably a mimicry of reality, and as such, Haskell provides a sense of reliability. At the same time, it allows us to program the very systems we would usually create with languages such as C.

Nevertheless, the fundamentals remain. Use of abstract reduction systems to model computation will forever remain the pillar of support for programming languages, and will continue to inform the design of systems for quite some time.

In a true testament to functional programming, my journey into networks with Haskell and theorem-proving in Isabelle will continue a concurrent fashion. The idea that these projects are so far from each other in concept and yet completely related is profound, and I'm looking to forward to seeing how functional programming will be used in the future.

References

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

[1] Guards vs. If-Then-Else Cases, Stack Overflow, 2021.

[2] Types and Type Classes, Learn You A Haskell, 2021.

[3] How To Use An Abacus, Ignite Spot Accounting Services, 2021.

[4] Functional Programming, Wikipedia, 2021.

[5] Declarative Programming, Wikipedia, 2021.

[6] Purely Functional Programming, Wikipedia, 2021.

[7] Functional Programming Vs Declarative Programming Vs Imperative Programming, Stack Overflow, 2021.

[8] Haskell REPL, Replit, 2021

[9] Implementing Functional Languages: A Tutorial, Simon L Peyton Jones, David R Lester, 2000.

[10] Types in Haskell, Haskell Wiki, 2021.

[11] How Do Haskell Compilers Decide Whether to Allocate on the Heap or the Stack, Stack Overflow, 2011.

[12] Laziness, Write You A Haskell, 2021.

[13] Concrete Semantics, 2021

[14] Abstract Rewriting System, Wikipedia

[15] Abstract and Concrete Syntax, Chalmers University, 2021

[16] Network Socket, Wikipedia, 2021

[17] Control-Concurrent, Hackage, 2021

[18] Commonly Used Ports, Host Papa Support

[19] Implement a Chat Server, 2019

[20] Network.Socket, Hackage, 2021

[21] Invariants, Alexander Kurz, 2021

[22] Rewriting: Introduction, Alexander Kurz, 2021

[23] Haskell Network Programming - TCP Client and Server, mchaver, 2021

[24] GHCi User Guide, Haskell, 2021

[25] The Imperative Mood, Aba English, 2021

[26] What is Referential Transparency, Stack Overflow, 2021

[27] Programming Languages, Wikipedia, 2021

[28] Grammar, Wikipedia, 2021

[29] Confluence and Normal Forms, Alexander Kurz, 2021

[30] Termination, Alexander Kurz, 2021