Creating a Concurrent Compiler in Haksell

Luke Jackson

May 4, 2015

Abstract

Creating a compiler and virtual machine in Haskell to run Goal, a simple concurrent programming language of my own design based on Google Go. This dissertation can be treated as a introduction to implementing a compiler in Haskell, but a basic understanding of Haskell and Monads is assumed.

Contents

Chapter 1

Introduction & Overview

1.1 Introduction To Project

For my final year project I have created a compiler in Haskell for a simple programming language of my own design, which I based on Google's relatively new language Go. I called this language Goal, and it is a simplified version of Go but contains all of the features I was interested in implementing in my compiler, most importantly allowing concurrent programming.

The main focus of this project was to implement a parser, compiler and virtual machine for a modern concurrent programming language, not on designing a completely new language. Though I will briefly discuss the reasons behind creating Goal, it is important to understand the focus of this project is on compiler implementation not programming language design.

This document is split into five main sections; Designing Goal, Parsing, Code Generation & Intermediate Representations, Code Execution and Testing. In each of these sections I will go into more detail about how I approached each of these problems.

These next few pages will give a brief introduction to compilers and programming languages, also giving more information about Go and the motivation behind the project. I will also briefly discuss any development methodologies used and give a justification for certain technical decisions.

1.1.1 Introduction to Compilers

To understand what a compiler is, you need to first understand what a program is and, more importantly, what a programming language is. Programs have become a fundamental part of human existence. From performing simple calculations to creating applications that allow me to create documents such as this, they exist in every aspect of our life. The simplest definition of what a program is, is to think of a program as a recipe.

You have a list of resources that you need, followed by a series of instructions telling you what to do with these resources. Then, to run your program, you grab all the resources you need and follow the instructions.

Programs that run on computers have to be represented in someway that both humans and eventually computers can understand. Therefore programs are written in programming

languages, a good definition for programming languages can be found in [Aho et al.(2007)Aho, Lam, Sethi, and Ullma p, 1] where they state;

> Programming Languages are notations for describing computations to people and machines ... But, before a program (in this format) can be run it first must be translated into a form which can be executed by a computer.

So this brings us nicely to being able to define what a compiler is, a compiler is something that takes programs written in one programming language and translates it into another programming language. Often taking a program written in a high level language, that humans write code in, and translating them into lower level languages that computers can more easily understand and then execute.

Finally a virtual machine is something that emulates the a compute, in our case it is able to run code written in a given programming language. You will usually only see machines try to execute low level languages given the smaller, and more precisely defined, instruction set. As is the case with the virtual machine I have created. This highlights the need for compilers, so that they can be used to translate high level languages into more easily executable lower level languages, that are much harder for humans to create programs in.

Looking at these explanations it is clear to see the importance of compilers in the modern world. They allow people to create complicated programs and applications efficiently without having to concern themselves with low level details.

1.1.2 Introduction to My Project Structure

There are three main parts to my compiler; parsing, compiling and the virtual machine. To best understand the structure of my compiler it is good to look at the definition of the function *run* which is called to run Goal code.

```
run f  = exec (typeChecker (comp (parseGo f)))
```

Where f is the .gol file we want to run. You can see that we first call the parser which passes its result to the code generator (*comp*), which passes its result to a simple type checker before the function *exec* will run the compiled code.

This shows how my compiler is structured and this dissertation will walk through the creation of the parser, code generator and virtual machine and how they interact with each other.

1.1.3 Introduction to Goal

Goal is the language I have created my compiler for. It is a language I have created specifically for this project as a means of exploring compilers. It draws heavy inspiration from Go and uses the same basic syntax where possible. The main features of Goal are the ability to perform function recursion and to create and run concurrent programs.

The idea for Goal came when i was picking features from Go I was interested in compiling then decided it would be a nice idea to bring them together to create a more complete language. In my opinion during the course of this project Goal has evolved from a means to explore compilers and language design into a simple standalone language with it's own uses. I will discuss more about the design and functionality of Goal in chapter 2.

It is recommended to make use of the Goal Documentation when starting to write new programs in Goal. The provided documentation gives example syntax of every command you can use and provides examples of how to use them and how they are expected to behave.

```
global i = 20;

func main() {
    var c chan = Make(chan int);
    go pong();
    go ping(i);
    WaitOn(ping);
    Kill();
};

func ping(n int) {
    for (i = 0; i < n; i++) {
        c <- 1;
        Print("ping");
    };
};

func pong() {
    for (True) {
        WaitChan(c);
        l = <- c;
        Print("pong");
    };
};
```

Figure 1.1: An example program in Goal,

1.1.4 Introduction to Go

Go is an object oriented programming language created by Google relatively recently in 2007. It is used by Google for many different applications, most notably it powers dl.google.com, a service which contains the source for Chrome and The Android SDK. A good place to find out more about the history of Go and it's development is in the documentation section on golang.org.

In many ways Go is similar to C in terms of syntax and that it is also statically typed. There are however some interesting features thrown in, such as how Go handles concurrency.

In summary I would say Go (also referred to as golang) is a new language with quite a bit of potential, which can be seen by it's growing popularity. Although it is not revolutionary, it's mixture of simplistic syntax and more exciting features makes it a good language to emulate.

1.1.5 Introduction to Concurrency

Concurrency is the primary feature I was interested in implementing in my compiler. It is a key part of this project and also a fundamental part of modern programming.

The basic idea behind concurrent programming is you can do many things at one time. If we first define a sequential program as a sequence of operations carried out one at a time.

We can then define a concurrent program as a program that contains a set of sequential processes executing in parallel [Terry(1997), p. 414].

The importance of concurrent programming can be seen now more than ever with rising popularity of online services and cloud computing. Lets take one of the most popular websites in the world Google.com, with around 40,000 request a second there is a high chance that when you hit that search button, so are thousands of other people at exactly the same time. But instead of queuing every request and doing it sequentially a server will handle the requests concurrently, ensuring that many requests can be handled at the same time, and you don't have to wait for the thousands of people who clicked a few milliseconds ahead of you till you get your results.

There are plenty more examples of the importance of concurrency in modern technology, I'm sure if you think about most pieces of technology you use it would be easy to list numerous process that have to run concurrently. This is why I was interested in implementing concurrency into my compiler, because of it's usefulness and importance in modern programming languages.

1.2 Motivation

The biggest motivation behind this project was a desire to learn more about compiling and executing modern object oriented programming languages. I was also curious about programming language design and wanted to take the chance to really look at a programming language analytically.

1.2.1 Why Haskell

I chose to implement my compiler using Haskell. Haskell is a purely functional language with strong static typing. Functional languages are often seen as good platforms to use when creating a compiler because some of their features make it easier to handle tree data structures, which can be important during parsing and creating an intermediate representation of language. Also the use of pattern matching and efficiently being able to recurse makes Haskell a good choice for implementing a compiler and a virtual machine.

Another reason I chose Haskell was quite simply that it is a language I enjoy working with, and more importantly for a project like this, would like to learn more about. I was keen with this project to not only create something interesting but also learn more about functional programming. I hoped to use this project to explore some unique functional approaches to some of the problems creating a compiler can throw up. A good example of this is how I handled parsing and my use of Monadic Parser Combinators.

1.2.2 Why Create A New Language

There are so many great programming languages out there with fantastic supporting documentation that would be excellent choices to make compilers for. Which really begs the question, why did I go through the hassle of designing my own language to compile? The answer to this is pretty simple, with a project like this if I was to create a compiler for C++, for example, I would never have the time to create something that covers all of C++'s functionality within 9 months. Therefore no matter how well I had done with the project it

would always feel a little incomplete, but more importantly if someone was to use my compiler they could very well end up trying to use features my compiler didn't support which would be frustrating for me and anyone who wishes to use my compiler in the future.

Instead of having this problem I decided I would create a new language that I could provide supporting documentation for and strictly define what is and isn't possible with in it. Because the focus of this project was not on language design my approach to creating a new programming language was finding a language I liked, then picking key features I was interested in exploring. Before finally adding some extra functionality so that this new language could be useful on its own.

The language I liked the look of was Go and that is where most of the features for my language came from, with the key feature I was interested in exploring being concurrency. I decided to call the language I had created Goal.

1.2.3 Why Base This Language on Go

Go is not a language I was overly familiar with before the start of this project, but as soon as I looked into it I very much liked what I saw.

At the start of this project I was exploring different possibilities of languages I could use as inspiration for Goal. When I came across Go I discovered it had a very clean and easy to understand way of creating and running concurrent process, which was one of the key features I was interested in putting into Goal. From there I began looking into Go's design and discovered not only did it have a wealth of interesting features to look at but also a clean and understandable syntax, which would work nicely with the simplistic easy to use nature I wanted to create in Goal.

The most important factors where defiantly the way Go handle concurrency and channels but also there was an aspect of using this project as a way of familiarizing myself with another language. It also helped that Go came with such a wealth of easy to navigate online resources which was perfect for when I needed to check the exact functionality of certain expressions.

Overall I felt Go provided me with a simple syntax and good implementations of the main features I was interested in implementing. Which made it a good choice to use as the basis of Goal.

1.3 Development Process

The main method I used for development was Test Driven Development (TDD). This is a simple approach where you write your test cases first, then write your code so that is passes all your tests. I found this to be quite an appropriate approach due to the iterative nature of my development process.

I initially started work by focusing on being able to compile and execute a small subset of simple features such as if statements, variable assignments and simple arithmetic. Then using TDD I wrote tests for each new piece of functionality I wanted to add, expanding the subset of features I was able to handle. Due to the nature of TDD I was able to do this without worrying about breaking earlier features as I could just ensure all my original tests still passed.

As was mentioned before, I split the implementations of my compiler into three main sections;

- Parsing
- Code Generation & Intermediate Representations
- Creating a Stack Based Virtual Machine

While developing my compiler and virtual machine I often found I would implement each feature in two steps; First ensuring I could create a suitable intermediate representation and updating my virtual machine to ensure it could handle the new feature. Then, once this was complete, I would update my parser to handle this new command.

Often when working on multiple features at a time, it would be more efficient to update the code generation and virtual machine first for all the new features. Then update the parser second, rather than adding each new piece of functionality one at a time.

I will go into more detail about the creation and running of my tests in Chapter 5.

Chapter 2

Designing Goal

This chapter deals with how I approached designing a new language and how I went about choosing the features I wished for Goal to contain. I will discuss some reasoning behind why certain features differ from how they are handled in Go and also talk about how I tried to make the language as complete as possible.

I think it is also important to note here that the main focus of this project is not on the design and creation of a new language but rather the implementation of the compiler itself.

2.1 Picking Features

As I mentioned before, rather than designing Goal from scratch I chose to instead choose features from Go I was interested in, then find a way of putting them into Goal. I did not create Goal with a target audience or with potential uses in mind, hence the lack of a specification or market research. Instead I created Goal as a means to let me implement a compiler that could handle a number of different interesting features.

Therefore you will find the features Goal can handle my seem quite varied and not completely complimentary of each other, but I do feel that Goal still has many uses.

As much as Goal was created simply as a tool for creating an interesting compiler, I did also attempt to make it as user friendly as possible. I have created supporting documentation for Goal that has full examples of the syntax it uses and detailed explanations of how to use each of it's features.

A good summary of my approach to creating Goal is that I filled it with features I wanted to explore, then added extra functionality to try and make the language as easy to use, complete and useful as possible.

2.1.1 Types and Scope

A type system defines what type of variables are allowed to be used in a language. A variables type can be defined by saying [Cooper and Torczan(2012), p. 164];

> The type (of a variable) specifies a set of properties held in common by all values of that type ... For example, an integer might be any whole number i in the range $-2^{31} \leq i < 2^{31}$.

Go has a very broad type system and static typing. Static typing is where any type errors are checked during compiling as opposed to dynamic typing where type errors are checked at run time. By saying Go has a broad type system I'm saying it allows for a lot of different types.

In Goal I decided not to focus on allowing lots of different types and instead only allow 2 types; Integers and Booleans. This was mainly so I could spend more time working on other features and not worrying about creating an extensive type checking system, whilst still having enough types to be able to create some interesting programs.

This also lead to allowing very basic variable declarations and assignments. You do not need to declare a type when initially declaring a variable and variables can be created on the fly. For example if you were to write;

```
i = 4;
i = True;
```

There would be no compiler or run time errors in this code and your variable i would have the value True as each new assignment writes over the old value. More specifically because booleans are treated as integers i will have the value 1. This is the biggest difference Goal and Go, it is actually much closer to how a language like Python handles variable declarations and puts more emphasis on the user keeping track of their variables.

It must be noted strings can be used and outputted using the $Print()$ command but you cannot save strings as a variable only use it to print out specific phrases. The command $Show()$ can be used to output numbers and variables and any other numerical expression.

A variable's scope defines where it can be accessed from. In Goal I decided to keep this very simple by only allowing variables to have two different scopes; global or local. A global variable is defined by using the keyword global before the definition and once declared can be accessed anywhere in your code. A local variable is just declared as shown above with no key word and can only be accessed with in the function you declared it in. More information about how variable scope works in Goal can be found in the accompanying documentation.

2.1.2 Basic Commands

By basic commands I mean the set of statements you expect to find in most object orientated languages. In Goal this includes;

- If Statements
- While Loops
- For Loops
- Output Commands
- Return Statements

All of these almost exactly follow the same syntax rules and have the same functionality that you find in Go. There are some small differences such as in Goal you are required to hold your conditional expression in brackets in each command but other than that there is not much difference. Again for more clarification on how these statements work you can find examples in the Goal documentation.

```
func main() {
    facRunner();
};

func facRunner() {
    fa = fac(5);
    Show(fa);
};

func fac(n int) int {
    if (n == 0) {
        return 1;
    };
    return n * facB(n-1);
};
```

Figure 2.1: Example of a program using function recursion and showing how to define different types of functions

2.1.3 Functions

Functions are treated very similarly in Goal as they are in Go. The main difference is that you can only have functions that either return Booleans or Integers, or your functions can be void. Function recursion is allowed as are nested function calls.

2.1.4 Concurrency

For concurrency I wanted to use the same simple technique that allowed you to run any void function as a concurrent process with the use of a keyword. Meaning if you wanted to run two functions $funA()$ and $funB()$ concurrently all you needed to do was write;

```
go funA();
go funB();
```

This code would start both the processes as independent subroutines and would execute them in parallel. I also added some useful library functions that I felt would be helpful for creating concurrent programs. These are not found in Go but are important to the way Goal works. The two most important ones are $Wait()$, which will halt a program until all subroutines have finished executing, and $Kill()$ which will immediately stop all running subroutines. There are a few other library functions I have included and again more information can be found in Goal's Documentation.

I also needed to implement a way to pass information between subroutines, although you could use global variables it is safer to have a more controlled method. This is why I implemented channels. These are also present in Go. You can declare a channel anywhere and it has global scope. The way to declare a new channel is shown below.

```
var c chan = Make(chan int);
```

They behave like stacks so you can push a value onto a channel in one subroutine then pop it out in another subroutine. Channels have a first in first out system. I have implemented channels almost exactly the same as how they are implemented in Go.

2.1.5 Syntax

A language's syntax can be said to describe the form that commands and expressions in a language must take [Terry(1997), p. 72]. In the case of programming languages it defines how you must write your code so that it can perform the computations you wish.

The syntax for Goal is pretty simple and almost identical to that of Go, with some minor differences. I decided to follow Go's syntax rules not only for simplicity but also so it was easy to see where Goal got it's functionality from. I felt it would make sense to give it syntax rules close to the language it was emulating.

While Loops;

```
for (x < 8) {
        x += 2;
};
```

If Statements;

```
if (x == 13) {
        x --;
} else if (x == 14){
        x ++;
} else if (x == 13){
        x -= 2;
} else {
        x += 2;
};
```

Output commands

```
Show(NUMBER);
Print("STRING");
```

Figure 2.2: Examples of Goal Syntax

A key part of Goal's syntax is that each command must end with a semi-colon, including if statements, functions and for loops. A more detailed outline and examples of valid Goal syntax can be seen in the accompanying documentation for Goal with examples of syntactically correct Goal [Jackson(2015)].

2.2 Differences From Go

Although the language I have created is based on Go there are quite a few noticeable differences. The biggest difference is that Go is statically typed and Goal I have made dynamically typed, although there is a very basic type checker in place at compile time.

This mainly means that if you convert your Go code to Goal you need to take care that you don't overwrite any variables.

Other noticeable differences come from the output functions being part of Goals standard library where as in Go you eed to import a package to output to a console.

Obviously the scope of Goal is much smaller as it has a much more limited type system and does not allow for multiple classes or files. Though it is noticeably similar and does contain many of the key features that Go contains

2.3 Possible Uses

Towards the end of this project I began to reflect on Goal as a standalone language and not just as something for me and felt it could have some uses of it's own.

Where Go is most popular at the moment is for a

Although I don't think Goal has a future competing with Go as tool for implementing large scale servers. I do feel that given Goal's simplistic syntax and easily understandable ,and usable, features it could have some use as a tool for teaching programming. More specifically helping people to understand how concurrent programs work, and giving them the tools to create their own exciting concurrent programs.

You can see I have provided with my project some example programs and as part of that I have included some programs I feel highlight how Goal could be used as a teaching tool in the future. I will revisit this idea as part of my evaluation of the project as whole.

Chapter 3

Parsing

Parsing is a fundamental part of compiling, you can think of it as the front end of the compiler. In simplistic terms it takes in a program as raw text and then builds a data structure that represents the program the user has written. In my project the raw text is the code from a .gol file and the data structure is the intermediate representation of Goal.

Although I will be talking a lot about the intermediate representation I've created to parse my text into, I will not be going into much detail about the design of that data structure. That is covered in more detail in the next chapter. In this section a preexisting understanding of Monads in Haskell is assumed.

3.1 Introduction to Using Monadic Parser Combinators

The technique I used to parse data in my compiler was to make use of Monadic Parser Combinators. If we think of a parser in Haskell as something that takes in a string and returns a data structure, we can think of parser combinators as high order functions that take in several parsers as its input and returns a new parser as it's output.

A lot of the comments in the section makes use of two papers on monadic parser combinators both [Hutton and Meijer(1996)] and [Hutton and Meijer(1998)] also making use of a library of parsers provided in the later.

We can first look at the type of the parsers provided;

```
newtype Parser a = Parser (String -> [(a,String)])
```

You can see that our parser takes in a string returning a list of results. We can see that our parser will have the type $ParserProg$ as we need our parser to output the type $Prog$ which is the type of my intermediate representation and the type which my code generator takes in.

The best way to understand how parser combinators work is too look at a few examples that you will find in Parsing.lhs provide by [Hutton and Meijer(1998)].

If we first look at the *item* parser. Which just consumes the first character of a string, and fails if the string is null.

```
item    :: Parser Char
item    =  P (\inp -> case inp of
                        []     -> []
                        (x:xs) -> [(x,xs)])
```

We can see that on it's own this parser isn't particularly useful, but with the use of parser combinators we can put it to some use. Now say we want to make use of this parser to parse digits. We need to first look at how we allow for conditional parsing, compared to *item* which will parse anything.

If we look at *sat* we can see how it makes use of *item* to allow for parsing on some condition;

```
sat     :: (Char -> Bool) -> Parser Char
sat p   =  do x <- item
              if p x then return x else failure
```

You may also notice the use of $failure$ which has type $Parser\ a$ and is a parser that will always fail regardless of input.

We can now make use of *sat* to create a parser to parse digits;

```
digit   :: Parser Char
digit   =  sat isDigit
```

This highlights how parser combinators work, by combing several parsers we are able to begin to create something useful .Using the same technique we could easily create parsers for specific characters or to only parser upper or lower case characters.

This is an incredibly brief introduction into a very interesting topic and it is recommend that the interested reader refer to either of papers referenced earlier to learn more about creating monadic parser combinators.

3.2 Goal Syntax Rules and Justifications

As was discussed in the previous structure Goal gets most of its syntax rules from Go. There are however some unique syntax rules I decided to implement in Goal that you will not find in Go. There were two main reasons why syntax rules my differ from Go.

The first being it was a design choice. For example if you look at variable declaration in Go it can be done one of 2 ways.

```
var i int = 42
j := 42
```

Although I could have implemented variable declarations to look like this I decided because variables can be declared on the fly and because of the simplistic nature of my type system I may as well keep declarations and assignments the same, and as simple as possible. Hence in Goal all you need to write to declare or assign a variable is;

```
l = 42;
```

There are several examples of these changes in syntax for design choice, such as global variable definitions and the requirement of brackets to hold conditional statements. These small changes where made merely as a way to tidy up Goal and make it have a more complete and consistent syntax.

The other reason Goal's syntax varies from Go is because of the limitations within the implementation of my parser.

For example, in Go you don't need to use semi colons at the end of a command. Where as I decided that in Goal I would make it necessary to include semi colons at the end of every command, including if statements and function declarations. For example;

```
func main() {
    i = 2;
    j = 0;
    for (i < 100){
        i = square(i);
        j ++;
    };
    Show(j);
};

func square(n int) int {
        return n * n;
};
```

You can see in this simple program how every command even function definitions and for loops need to finish with a semi colon.

This is because it makes it easier to split up commands based on every time I see a semi colon. The reasons I made some of these choices was not because it was not possible for me to implement different syntax rules, but because the main focus of this project was not on parsing and if a small change to syntax meant a quicker implementation sometimes I felt it necessary.

3.3 Parser Implementation

My parser implementation was focused around using monadic parser combinators. As part of this I used a collection of functions provided from the paper. These can be seen in my abstract in the Parsing.lhs file. What these functions do is provide a series of simple parsers to use when creating more complicated parser combinators.

Then it was simply a question of building up parsers to handle more restrictive rules. A good way to understand this is to walk through a simple example of parsing comparative expressions.

Although this section gives a walk through of how I used parser combinators, it should not be considered a standalone introduction. The interested reader should look at which provides a more in depth introduction and more detailed descriptions about monadic parser combinators.

3.3.1 Example of Parser Implementation

If we first start with defining the data structure, also known as the intermediate representation we wish to parse down to, we can then look to define the language which we are parsing;

```
data Expr       = CompExpr Op Expr Expr | Val Number

data Op         = EQU | NEQ

type Number     = Int
```

This is a pretty simple data structure representing two types of conditional expressions. We now need to define a grammar that is acceptable for our language. A grammar, or formal grammar, is a set of rules by which valid sentences (or in our case commands and expressions) in a language are constructed [1]. In this case when we talk about a language we are not talking about a programming language in the typical sense, instead we define a language as a set of strings made of symbols that our limited by rules that are specific to that language [2].

To define our grammar it is good to look at some examples of what is acceptable in our little language, if we say we are only interested at looking at conditionals where you use == to signify equals and != to signify two values being not equal;

```
4 == 7
5 != 5
42 == 42
12 != 34
```

You may notice that the expressions we are allowing don't have to be true. In this example we are allowing the user to express comparative expressions our parser has no need to deal with what is and isn't allowed computationally as long as our grammar allows it, the expression can be as bizarre as you want. This shows how we have to be careful defining our grammar to only allow what we want. For example if we look at our data structures we actually allow for some nested comparisons.

[1] http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/06-Formal-Grammars.pdf
[2] http://en.wikipedia.org/wiki/Formal_language

You can see that in our data structure the following expression is valid;

```
CompOp EQU
        (CompOp NEQ
                (Val 5)
                (CompOP EQU
                        (Val 4)
                        (Val 7)))
        (CompOp NEQ
                (Val 3)
                (Val 5))
```

Which would look like this;

```
(5 != (4 == 7)) == (3 == 5)
```

Now you can argue this makes sense and should be allowed but I feel like in our simple language it's not what we want to include, also we need to consider if we end up including more types of comparisons, such as $>$ or $\geq$. Things will start to get messy quickly. Regardless of the justification for this example we won't allow nested comparisons.

There are two ways to handle this problem. We can either alter our data structure so that it is no longer allowed, or we can create a grammar that does not allow for nested comparisons in our language. It is better to restrict our grammar than our intermediate representation. So we now need to define our grammar. In simple terms we can describe it as;

```
<NUMBER1> <COPARISON OPERATION> <NUMBER1>
```

Though we need to create a more formal definition. Lets say a sentence in our language has to consist of one natural number followed by either a == or ! = then another natural number. We will allow white space between the numbers and comparison operators.By natural numbers we are referring to any whole number.

To create a parser to handle this grammar we will need to make use of some parsers provided in Parsing.lhs;

- *natural*, will parse natural numbers
- *symbols* will parse the given symbol s

Making use of this we can use of Haskell's do notation we can also generate a parser to parse the different conditional symbols == and ! =.

```
        parseOp   :: Parser Op
        parseOp   = do symbol "=="
                       return (EQU)
                    +++ do symbol "!="
                           return (NEQ)
```

Using parse combinators means we can combine multiple parsers. Using this approach we can now create code to evaluate conditional expressions allowed in our language;

```
evalCompExpr    :: String -> Expr
```

```
evalCompExpr xs = case (parse compExpr xs) of
                    [(e,[])]  -> e
                    [(_,o)]   -> error ("unused "++ o)
                    []        -> error "invalid"

parseComp         :: Parser Expr
parseComp         = do n1 <- natural
                       o  <- parseOp
                       n2 <- natural
                       return (CompOp o n1 n2)
```

This example shows how you build a simple parser using monadic parser combinators and the Haskell script provided . It shows the the process of building up a grammar to define your language and also begins to show the process of combing parsers.

3.3.2 Analysis and Expansion of Parser Example

Looking at this example you can see the process of slowly building up a grammar and the parsers to enforce it.

There are a number of important ideas shown in this example, one of the most important concepts is the idea of making sure your parser is more restrictive than your data structure. There are several aspects of my intermediate representation that allow for features that are restricted in my grammar. The reason this is done is because it makes generating code simpler without really making my parser any more complex.

Another good thing too take away from this is the idea of abstracting problems out our final parser could have easily looked like this;

```
parseComp         :: Parser Expr
parseComp         = do n1 <- natural
                       do symbol "=="
                          n2 <- natural
                          return (CompOp EQU n1 n2)
                        +++ do symbol "!="
                               n3 <- natural
                               return (CompOp NEQ n1 n3)
```

But because of the creation of our *parseOp* function not only were we able to have a much cleaner implementation, also making it much easier to expand our parser to include more comparison operations.

One concept that is not touched upon in this example is the idea of having a hierarchy of bindings. If we take the idea of operator precedences within mathematical expressions if we want a multiplication to be done before an addition in our language we need to generate an intermediate representation that will show this.

By combining multiple parsers we can create a hierarchy were an addition expression is dealt with before anything else by splitting on the + symbol before anything else. Then nesting other arithmetic expressions within the output of the first parser. Examples of this technique can be seen throughout the implementation of my parser.

3.4 Potential for Expansion

Parsing is a topic that could easily become the main focus of a project like this. The idea behind using parser combinators was that they had been something I was interested in exploring and the more i used them the more I began to see how they could take up ore of this project. Instead I decided to focus this project in other areas but that does leave a list of areas I would be interested in coming back too.

One of the main things you could expand within my parser would be to add some lexical analysis before beginning parsing. Lexical analysis is used to identify a valid set of words used in the language [Chattapoadhyay(2005), p. 13]. It is useful a it means you don not begin actually parsing before being certain that only valid commands exist in each program.

Another big area for expansion would have been spending more time looking into being able to add more types. You will notice looking at my code that my compiler and executor can handle doubles. But unfortunately due to limitations with my parsing I was decided against including it in the final release of Goal because I wanted to focus more time elsewhere. Though if this project was picked up in the future it would be one of the first features to update.

Chapter 4

Code Generation & Intermediate Representation

Code generation is where I take the parsed input and then generate code from a low level instruction set that can then be run by my virtual machine. I decided to implement my own low level language and create a virtual machine to run code written in that language. Thus meaning that I would be compiling the parsed input down to code that used an instruction set that I had defined.

A key part of this section is looking at how I designed and implemented an intermediate representation of Goal in the form of a data structure. This intermediate representation was then simpler to compile into my low level instruction set. The importance of a good intermediate representation can be seen in [Cooper and Torczan(2012), p. 221] where they state;

> Most passes in the compiler read and manipulate the IR form of code. Thus, decisions about what to represent and how to represent it play a crucial role in both the cost of compilation and it's effectiveness.

In this section I will go into detail about the design of the data structure that you are required to parse down to, and which represents all of the features I implemented in my language. I will also be discussing some of the more interesting features that I implemented and how they were dealt with by the code generator.

Although in this section I will be talking a lot about code generated using the instruction set defined in my executer, I will not be going into detail about the design of the executor or the low level language I created. For more information on this you can go to the next section that does focus more on the design and execution of the low level instruction set.

4.1 Intermediate Representation

Having a good intermediate representation of the program being compiled is very important part of creating a compiler. It is important to make sure that you do not misinterpret the program you are compiling and a good way to do this is to create a data structure that clearly represents what your program is doing, whilst ensuring this data structure is easy enough to compile into your target language.

4.1.1 Introduction to Intermediate Representations

An intermediate representation of a program is where you create some data structure of your program that can be more easily interpreted and handled by your compiler. It is also important to ensure your IR[1] is designed in such a way that you capture the meaning of the program you wish to represent, and do not end up misrepresenting your program.

There are several approaches to generating an IR. These can been seen in [Cooper and Torczan(2012), p. 223] where they state there are three main approaches to generating an IR;

- Graphical IRs, This uses tree or graph data structures.
- Linear IRS, This will closely resemble pseudo-code.
- Hybrid IRS, A combination of both Linear and Graphical approaches.

I chose to go with a Hybrid IR. A most suitable approach for me because, as stated in [Chattapoadhyay(2005), p. 113], Linear IR are often used when compiling for stack base virtual machines, which is the architecture of the virtual machine that I created. Also I wanted to take advantage of how easily Haskell allows you to create recursive data structures, that can be used when generating trees you would find in Graphical IRs. Hence why I decided to use a Hybrid IR.

This approach provided me with a relatively high level representation of programs, meaning it gave a representation that would closely resemble the pseudo-code of the program you are trying to compile. This makes it nice format to work with as it makes it easier to visualize what is actually occurring during the later stages of code generation.

It is important to note there are several different approaches I could have used when creating an intermediate representation and the choice to use a high level approach was both a design choice and also to do with ease of implementation.

To best understand the approach I used to create the data structure for my intermediate representation it is good to follow a simple example of how you take a specific command then translate that into my intermediate representation.

4.1.2 Example Creating an Intermediate Representation

I will now show an example of the process I went through when generating my IR. The example I will use is creating a data structure that best represents an if statement. This is what an if statement looks like in Goal;

```
if (x < 1) {
        return x;
};
```

You can see that there are two main parts to this statement. The condition directly after the if command, $(x > 1)$ and then the code ,in this case *return* x;, inside of the curly brackets.

What is important to notice is that you could replace the return statement with any other acceptable code, even another If statement. Whereas the expression after the if is limited only to be certain things. In this case we could rewrite a general definition for an if statement to look like this;

[1] IR is shorthand for Intermediate Representation

```
if EXPRESSION {
        PROGRAM
}
```

This can be described by saying; If some expression is true, then execute the program inside the curly brackets.

Now all we need to do is define what EXPRESSION and PROGRAM can be. Then we can create a data type in Haskell code to represent If statements that looks like this;

```
data IfStatement = If Expr Prog
```

A program is quite easy to define, it's just a one or more instructions written to perform a specific task[2]. Therefore we must define all of the instructions that can be used to make a program in our data type Prog. Looking at our example that needs to include return statements, but I also said we were allowed to have nested if statements. Therefore we can create a recursive data structure that will allow this;

```
data Prog = If Expr Prog | Return Expr
```

An expression is slightly harder to define. In the case of If statements we know our expression needs to do something. It needs to give us some condition which we can then decide is true or false.

We can now create a data type for our expressions. We will need to allow for nested expressions but also allow for different comparison operators such as equals or less than. This can be shown here;

[2] $wikipedia.org/wiki/Computer_program$

```
data Expr = ExprComp Op Expr Expr
          | Val Number
          | Var Name

data Op   = GET | LET | NEQ | EQU
```

Although this example gives us quite a strict definition of what an expression needs to be for an If statement later on we will need to come up with a more general definition that is more applicable to other instances of using expressions (such as in arithmetic operations). The language specification for Go describes an expression by saying; "An expression specifies the computation of a value by applying operators and functions to operators". Another way to think of an expression is it is any valid unit of code that resolves to a single value [3].

Now that we have our definitions for what Expressions and Programs can be, we can begin to group what instructions belong in which section. With all our expressions being defined in the data type Expr and all our instructions used to make programs in the data type Prog.

Therefore if we are only considering our If statement example from earlier (ignoring how we defined the variable x) we end up with the following data structure;

```
data Prog   = If Expr Prog | Return Expr

data Expr   = ExprComp Op Expr Expr
            |Val Number
            |Var Name

data Op     = GET | LET | NEQ | EQU

type Name   = Char

type Number = Int
```

The above data structure now enables you to create high level intermediate representation of the original If statement we looked at, through the use of Haskell data types.

You can see here how the data structure has a broad scope actually allowing expressions such as $(4 < 6)! = (4 < (5 > 6))$, which don't really make sense. But I felt it was better to allow for these bizarre expressions, and handle any stricter rules in parsing stages. Rather than create an IR that was too restrictive.

4.1.3 Analysis and Expansion of Creating an IR Example

The key point of the above example is to understand that each part of a languages features can be, and needs to be, strictly defined. I found it important to remember the data structure I am creating has a purpose other than just being a new representation of Goal. It needs to actually extract the important pieces of information from the code, such that you are left with a series of concise statements that hold all the information necessary for you to start to rebuild the program using a new instruction set.

The example also highlights the process of abstracting out each part of a languages functionality, then choosing the appropriate structure to use to represent them. An important

[3] url for quote

part of this example is showing the use of recursive data structures to generate tree like data types.

For example look at the code;

```
if (x > 1){
        if (x > 5){
                if (x == 6) {
                        return 1;
                };
        };
};
```

It will be represented using data structure from our example by the following Haskell expression;

```
If (ExprComp GET (Var 'x') (Val 1))
        (If (ExprComp GET (Var 'x') (Val 5))
                (If (ExprComp EQU (Var 'x') (Val 6))
                        (Return (Val 1))))
```

This can also be visualized in a tree structure with branches to the right being the outcome if the condition is true and branches to the left if the condition is false;

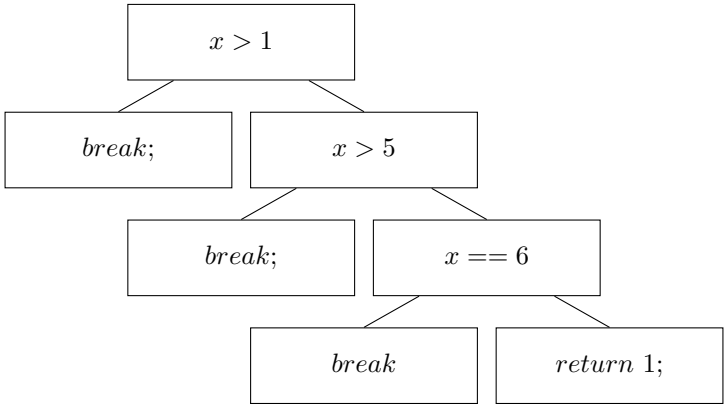

Figure 4.1: Visual representation of tree structure of nested if statements

What figure 4.1 shows you is the importance of using recursive data structures when allowing an arbitrary number of nested commands. You will see that as you start to expand your data structure to include multiple function declarations, or even large conditional expressions, building a tree like data structure is a good way to represent any program.

Overall this example helps to introduce all the key concepts behind the creation of the IR I used in my compiler.

4.1.4 My Intermediate Representation of Goal

Below I show the data structure I created to as an intermediate representation of Goal;

```
data Prog         = CreateChan Name
```

```
                    | PushToChan Name Expr
                    | Assign Name Expr
                    | Show Expr
                    | Print String
                    | If Expr Prog
                    | IfElse Expr Prog Prog
                    | ElseIf Expr Prog [ElseIfCase'] Prog
                    | While Expr Prog
                    | Seqn [Prog]
                    | Empty
                    | Return (Maybe Expr)
                    | Func FName [Name] Prog
                    | Main Prog
                    | VoidFuncCall Name [Expr]
                    | GoCall Name [Expr]
                    | Wait
                    | Kill

data ElseIfCase' = Case Expr Prog
```

You can see that for almost every command you can use in Goal it has been broken down the same way that was shown in my example and represented in a psuedocode style whilst allowing the creation of tree like structures through the use of the recursive data structure.

One of the most important elements of my IR is the $Seqn[Prog]$ command. This is one of the key aspects of the IR and what allows for multiple commands within functions and even for allowing multiple functions.

4.2 Code Generation

Code generation is one of the most important aspects of compiling. It is the process of translating an intermediate representation of a program into code, without losing that programs meaning. In my case it meant creating functions that could translating my IR into code made up of the instruction set my executor could handle.

4.2.1 Brief Introduction to Target Language Instruction Set

To understand how code generation works it is good to get a brief understanding of how my instruction set is structured and behaves. This section is just a brief introduction to the concepts that are necessary to understand in order for me to explain the process of generating code, more information on executing the code and the design of the instruction set used in my executor can be found in chapter 5.

My instruction set was designed so that you can translate any intermediate representation, no matter how complicated, into a list of instructions that can then be executed one by one.

There are two important things to understand about my virtual machine, the platform which executes the code we are generating. First, is that it uses a stack so any operations we wish to perform must do so using a stack. Secondly code execution is controlled by a program counter which starts at 0 and increments by one after every instruction, unless an

instruction wants to jump to a another section of code then the program counter will be set based on where the in the code you want to go.

Don't worry if you are not to familiar with these concepts as they are gone over in more detail in the next chapter.

4.2.2 Example Code Generation

I will now show an example of how you would generate code for an instruction from our data set. We will carry on with our earlier example of looking at an if statement now we know what the IR for this instruction is we should look at the code we wish to generate.

```
if(x < 1){
        return x;
};
```

Will become;

```
If (CompExpr LET (Var x) (Val (Integer 1))) (Return (Var x))
```

Now I will show the code we need to generate;

```
[PUSHV "x",
 PUSH (Integer 1),
 COMP LET,
 JUMPZ i,
 PUSHV "x",
 RSTOP,
 LABEL i]
```

To better understand this code you may want to refer quickly to chapter 5, but to give a brief understanding the variable x is compared with 1 then if this condition is true 1 is placed onto the stack, if not 0 is. Then $JUMPZ\ i$ checks the top of the stack to see if it finds a 0, if so it will jump too $LABEL\ i$ if not it moves on to return x.

To generate code we will need to use pattern matching. If we first generate our the type of our compiler function *comp* we can see more easily exactly what it needs to do.

```
comp  :: Prog -> Code
```

Where *Code* is the type that represents a list of instructions. If we look at our if statement we know that it is split nicely into two parts; the *Expr* and the *Prog*. Therefore we can use this in our code generator.

If we first focus on creating a function that handles creating a function which can generate code for our expression.

```
compE                    :: Expr -> Code
compE (CompExr o e1 e2)= compE x ++ compE y ++[COMP o]
compE (Val n)           = [PUSH n]
compE (Var v)           = [PUSHV v]
```

This may seem like a bit of a jump but it's pretty simple to see whats going on here. It's good to think of the *Expr* type as things that leave a result on the stack, and then our *Prog* type as commands that do things based on whats on top of the stack.

Making *compE* recursive is also useful in case we want to include more types of expressions, for example it allows us to easily expand this function to handle expressions such as

$(5 + 6) == (12 - 1)$. We can make sure that nay restrictions on expressions are dealt with in our parsers.

Now we need to look at how we can incorporate this function into solving our initial problem, generating code for if and return statements. Lets first try this;

```
compP             :: Prog -> Code
compP (If e p1)   = ifDealer e p1
compP (Return e) = compE e ++ [RSTOP]
```

Return is pretty simple to understand as it's just placing something on the stack and then using the call $RSTOP$. We now need to define our $ifDealer$ which creates the code for if statements. Remembering that we still want to allow all the features we mentioned in our earlier example.

```
ifDealer     :: Expr -> Prog -> Code
ifDealer e p =
        compE e ++ [JUMPZ 1] ++ compP p ++ [LABEL 1]
```

This looks suitable but there are some issues, what if we have nested if statements, we would be using the same the same label everywhere so how would we know where to jump to? We need a way to generate a unique label for each time we want to use the $LABEL$ command.

```
    type State      =  Label

    data ST a       =  S (State -> (a, State))

    apply           :: ST a -> State -> (a,State)
    apply (S f) x   =  f x

 instance Monad ST where
    return x        =  S (\s -> (x,s))
    st >>= f        = S (\s -> let (x,s') =
                           apply st s in apply (f x) s')

    fresh           :: ST Label
    fresh           = S (\s -> (s, s+1))
```

Figure 4.2: Definition of the state monad

This is where we use the state monad. It can be used to generate a new label whenever we want and helps us make sure that we never have duplicated labels. In figure 4.2 you can see the definition of the state monad and the funtions $fresh$ which is used every time we want to generate a new label.

We now need to update our $compP$ and $ifDealer$ functions to make use of the state monad;

```
compP             :: Prog -> ST Code
compP (If e p1)   = ifDealer e p1
compP (Return e) = return (compE e ++ [RSTOP])
```

```
ifDealer     :: Expr -> Prog -> Code
ifDealer e p =
        do i  <- fresh
           pc <- compP p
           return (compE e ++[JUMPZ i]++ pc ++[LABEL i])
```

This now will allow us to generate code for an arbitrary number of nested if statements. Though we now have the issue of that we now return $ST\ Code$, when we need to be passing $Code$ to our virtual machine.

Therefore we need to create another function to get rid of this for us. Which still has the type $Prog -> Code$ that we want. We can do this by making use of the *apply* function seen in figure 4.2.

```
comp    :: Prog -> Code
comp p  =  fst (apply (compP p) (0))
```

Now we can generate code in a format that can be run by our virtual machine.

4.2.3 Analysis and Expansion of Code Generation Example

The example is very useful in showing one of the most important features of my code generation section, the use of the state monad to ensure that the creation of each label is unique and allows for looping and conditional expressions in the language I am compiling.

It also shows the importance of using pattern matching and recursive functions to be able to efficiently generate the correct output and also allow for an arbitrary number of nested expressions with the code being compiled. It also helps to highlight why Haskell is such a useful language to use in compilers more specifically in projects where code generation is the major focus.

The example can also be expanded to include another Monad. The Writer Monad, which can be used to tidy up outputting the code. The inclusion of this Monad also requires the use of a Monad of transformer.

Overall this example highlights very nicely exactly how my code generator works. In very similar fashion to the designing of my IR I wanted to make sure I could handle all commands exactly as needed through the use of pattern matching. Then, use recursion, to allow for some arbitrary number of nested commands. Finally I used monads to ensure the creation of unique labels and tidy up my implementation.

Chapter 5

Code Execution using a Stack Based Virtual Machine

This chapter goes into detail about the method I used to execute the code generated by the code generator. The previous sections have mainly focused on compiling programs down to an instruction set. In this section I will go into detail about the design of that instruction set and the method I used to run it.

When creating a compiler you must take in a source language, the code you wish to compile, and then output a target language. A target language is the language you wish to compile into. In simpler terms, your compiler is taking in programs in your source language and then outputting the same programs but they are now represented using your target language.

For a project like this there are two approaches you can take when choosing what your target language should be. You can either find a preexisting low level language like ARM code or Java Byte Code. Or you can create your own instruction set and a virtual machine that is capable of executing those instructions. I decided to make my own instruction set and a virtual machine to execute it.

5.1 Introduction to Stack Based Virtual Machines

There are two things too define here before we can start talking about stacked based virtual machines . We need to first define what is a virtual machine, then secondly what is a stack machine.

A virtual machine can be described as a self contained operating environment that behaves as if it is a separate computer[1] . This can be useful because it means no matter what machine you are running your VM[2] on any application you run on your VM will run the same regardless of the physical machine you are using.

A good example of a virtual machine is if you have ever played a retro games console emulators on your phone or computers. Those are examples of fully functional virtual

[1] `http://www.webopedia.com/TERM/V/virtual_machine.html`

[2] VM is short hand for virtual machine

machines running old programs independently of the machine they are running on.

A stack machine is a machine or (virtual machine) that uses a pushdown stack instead of set registers to evaluate different expressions[3]. A pushdown stack is a data structure with two main operations;

- Push, which puts something onto the stack.
- Pop, which removes the last element put on to the stack.

You can think of a stack like a PEZ sweet dispenser. The first bit of candy you put in goes straight to the bottom of your dispenser and if you put more in they get pilled in on top. This is the same way you push objects onto a stack. Then when you wish to eat your candy, the last piece you put in comes back out first. This is the equivalent of popping the stack where the last object you pushed onto the stack comes out first. This is called a last in first out system.

Therefore a stack machine is quite simply a machine that uses a stack instead of allocated registers to handle expressions. If you look at it shows you how the expression $2 - 1$ would be evaluated using a stack in a stack machine.

Now we understand what a stack machine is and what a virtual machine is it becomes quite easy to understand what a stack based virtual machine is. Quite simply it is a virtual machine that uses a stack architecture, effectively you can think of the virtual machine I created as a simple stack machine emulator.

5.2 Implementing a Stack Based Virtual Machine in Haskell

The way the executor I crated works is it takes in code written from a simple instruction set then the virtual machine will execute the code one instruction at a time. For a quick overview of how my stack based virtual machine works you can look at the type of the main functions that handle executing code.

```
exec    :: Code -> String
```

Which simply starts a call to the recursive function;

```
exec'   ::exec' type goes here
```

[3] http://en.wikipedia.org/wiki/Stack_machine

What looking at *exec* shows is all that is needed to start the executor is the code that has been generated by the code generator. Then looking at $exec'$ you can see the different data structures that are used in implementing my virtual machine in my executor. The main components are;

- The Code
- The Program Counter
- The Stack
- Memory
- Channels
- List of Subroutines

The code is simply referring to the code you are currently executing and the program counter tells you where in that code you are. The stack is used like registers would be, where you will hold values you are currently interested in handling. Memory is where you store any variables that you will need to refer back to. Subroutines and channels are used for concurrency and I will go into more detail about them later.

The key concept to take way is that memory is for storing objects for the long term where as the stack is for passing around values and holding them temporarily.

5.2.1 Explanation of Instruction Set

Chapter 4 shows how the code is generated from an instruction set, and in figure 5.1 you can see the data structure I created in Haskell that represents my instruction set.

To understand how this instruction set works it is good too look at a couple of examples of generated code from the instruction set and what exactly different instructions mean. Then we can move on to looking at more detailed example of how you go about building an executor for this instruction set in Haskell.

If we first look at a variable assignment. If we wanted to compile the code: $x = 7$; it becomes;

```
PUSH (Integer 7)
POP "x"
```

Or more accurately in Haskell, since Code has the type $[Inst]$, it would look like this;

```
[PUSH (Integer 7), POP "x"]
```

This uses two of the most important instructions. $PUSH$ Takes a number as an argument and pushes it onto the stack, POP takes a name as an argument and removes the head of the stack saving it in memory, overwriting any existing variables with the same name in the same scope in memory. You can use $PUSHV$ which takes a name as argument to push a variable from memory onto the stack.

In code generation one of the few features of the instruction set I went into in any detail was the idea behind the $LABEL$ and $JUMP$ instructions. So you may remember that compiled code for the expression;

```
data Inst  = PUSH Number
           | PUSHV Name
           | POP Name
           | SHOW
           | PRINT String
           | DO ArthOp
           | COMP CompOp
           | JUMP Label
           | JUMPZ Label
           | LABEL Label
           | FUNC FName
           | FEND
           | VCALL Name
           | CALL Name
           | STOP
           | RSTOP
           | MAIN
           | PUSHC Name
           | POPC Name
           | CHANNEL Name
           | WAIT
           | KILL
           | GO Name

type Label =  Int
```

Figure 5.1: Haskell data structure used to represent all the instructions in my instruction set.

```
for (x < 5) {
        x++;
};
```

Will look like this;

```
[LABEL 0, PUSHV "x", PUSH (Integer 5), COMP LET,
JUMPZ 1, PUSHV "x", PUSH (Integer 1), DO ADD,
JUMP 0, LABEL 1]
```

To understand this better we can annotate the code to show what is going on at each instruction, this is shown in figure 5.2. It is a good way to explain several instructions that I will be using frequently throughout this chapter.

This gives a more detailed insight into how $JUMP$, $JUMPZ$ and $LABEL$ are used to represent conditional expressions on a lower level. It also introduces the idea of how you perform operations involving two values, where you have instructions that pop the top two elements of the stack then push the result of the operation back onto the stack.

```
LABEL 0           -- places a label with name 0

PUSHV "x"         -- pushes variable x onto the stack

PUSH (Integer 5) -- pushes the number 5 onto the stack

                     performs "<" comparison between the
COMP LET          -- top 2 elements of the stack, places 0
                     ontop of the stack if it's false and
                     1 if true

JUMPZ 1           -- jumps to label 1 if the head of the
                     stack is 0

PUSHV "x"         -- pushes varaible x onto the stack

PUSH (Integer 1) -- pushes number 1 onto the stack

DO ADD            -- adds the top two items on top of the
                     stack, places result on top of stack

POP "x"           -- pops the item from top of stack and
                     saves it memory with name "x"

JUMP 0            -- Jumps to the label 0

LABEL 1           -- places label 1
```

Figure 5.2: Annotated example of what each instruction is doing in a simple while loop

5.2.2 Example Virtual Machine

I will now run through an example of how I created a virtual machine to handle my instruction set. To do this I will start with a data structure representing a much smaller set of instructions and then show how I build a virtual machine capable of executing code built from these instructions.

The instructions I will use in my example can be shown in the following data structure;

```
data Inst  = PUSH Number
           | PUSHV Name
           | POP Name
           | DO ArthOp
           | JUMP Label
           | JUMPZ Label
           | LABEL Label
```

This is a relatively small set of instructions in comparison to what I used in my final project, but you should be familiar with most of the commands, as they were all introduced in the previous section.

Now we have our instruction set we can move on to defining everything we will need to make it work.

```
type Name   = String

type Number = Int

type Label  = Int

data ArthOp = ADD | SUB
```

Now we have some raw data types to use, we now need to think about what we will need in our virtual machine to make it work. For this example we will definitely need a memory and a stack. So we add;

```
type Code    = [Inst]

type Stack   = [Number]

type Memory  = [(Name, Number)]
```

It make sense that memory should be a list of tuples because memory is a series of values with an allocated name used to reference them. It also makes sense that our Stack just needs to be a list of numbers. We now need to decide what our executors type should be. I think its good if it takes in some code and returns us the stack. But the problem with that is how do we pass about all the components of our virtual machine. well we need too create a helper function thats going to do all the hard work. Seeing as we just need the code to start our executor we can write;

```
exec     :: Code Stack
exec c   = exec' c 0 [] []

exec'    :: Code -> Int -> Stack -> Memory -> Stack
```

So now we need to decide how we will handle our code. If our code is just a list of instructions then we can use our program counter to find the element in the list we should be dealing with then use a case analysis to say what to do depending on our instruction.

Lets pretends we have already defined some functions to speed things up.

- *pop* will take in the stack and return the stack after popping it
- *push* will take in the stack and a value, pushing that value to the top of the stack, then returning the updated stack
- *pushv* will do the same as *push* but takes in a name and memory and pushes the variable from memory to the stack
- *save* will take the head of the stack and a name, then save that value to memory
- *jump* will take in the code and a label number and return a program counter that is set to the location of that label.
- *jumpz* does the same as *jump* but takes in the stack as well only performing a jump if the head of the stack is 0, otherwise it just increments the program counter as normal.

We will also need to define a function *do* that will take take in the stack and an *ArthOp*. Then it will return the stack after performing the required operation on the top two elements of the stack and pushing the result on top. I will define this function below as an example, so as to give you an idea as to how some of the other functions could be implemented.

```
do       :: ArthOp -> Stack -> Stack
do o s   = case o of
              ADD -> push (v2 + v1) ns
              SUB -> push (v2 - v1) ns
           where
              v1   = head s
              v2   = head (tail s)
              ns   = (pop (pop s))
```

Now that we have these functions we can easily create a recursive function to handle a long list of instructions.

```
exec'     :: Code -> Int -> Stack -> Memory -> Stack
exec'c pc s m
          = if pc >= (length s) then s
            else
              case c !! pc of
                POP n    -> exec' c (pc+1) (pop s) (save s m)
                PUSH v   -> exec' c (pc+1) (push v s) m
                PUSHV n  -> exec' c (pc+1) (pushv n m s) m
                LABEL l  -> exec' c (pc+1) s m
                JUMP l   -> exec' c (jump l c) s m
                JUMPZ l  -> exec' c (jumpz l c s) (pop s) m
                DO o     -> exec' c (pc+1) (do o s) m
```

The above function is a very basic example of how the main function in my executor works. Using recursion and pattern matching I am able to iterate over a list of instructions and create multiple functions to assist me in creating an efficient implementation of a virtual machine.

5.3 Handling More Advanced Features

After using the example to show you how my executor works, I will now discuss some of the more interesting features I implemented in my executor and how I handled these features.

5.3.1 Memory Design and Implementation

Memory is handled very simply in this project it has the same type it was given in the example in section 5.2.2 and variables are referenced in almost exactly the same manner. The only major difference is that I included variable scope in my project and as such had to handle a number of different possible outcomes.

Handling Variable Scope

The main difference from the example is that memory in my executor is partitioned when it is initialized. My parser does not allow for variables the empty string as the name. Therefore to partition memory I initiate memory by creating a list of ten empty values , $(\text{""}, 0)$. Then all global memory is stored at the head of the list and local memory is stored at the tail, meaning memory is split in two. This makes it easy to handle scope because every time there is a change of scope, e.g. you begin to execute a function, you can simply drop everything after the list of empty variables as they will no longer be in scope.

This is a very simple approach to partitioning memory and one of the weaker aspects of this project. The current implementation does not facilitate the use of pointers, due to the complete dropping of the previous local memory on a change of scope, but this would not be too difficult to update my executor to handle. Though it may require restructuring the way I currently handle memory.

5.3.2 Stack Management

Manging a stack is a very important part of implementing an efficient and, more importantly, working stack based virtual machine.

Handling Function Calls

Handling function calls is one of the more interesting things to implement in a stack. If we first look at what happens in our executor when we hit the two commands that deal with function calls; either $VCALL$ (for void function calls) or $CALL$, a function called *handleCall* is called. The purpose of this function is to execute the function call before moving onto the next command. This seems simple enough but there are two complications; functions that take in arguments and functions that return values.

If we first look at handling arguments, one approach could be to change the data type of the function call instructions to now incorporate arguments is as well like this;

```
data Inst     = ...
              |CALL Name [Argument]
              |VCALL Name [Argument]
              ...

type Argument = Number
```

This looks like a good solution at first, but what if we want to call a function using an expression or a variable as arguments. Calls like $fun(a + 5, (4 * b) - c)$; would start to get very messy to deal with.

A better approach would be to use stack frames. A stack frame is a frame of data that is put on top of the stack. In the case of function calls it means putting all the arguments onto a the stack, then ensuring they are popped off in the correct order.

If we look at what happens if we were to call a function that takes in 2 arguments we can see that both arguments get pushed onto the stack before calling the function, this is shown below.

```
fun(12, 30);     -->  PUSH (Integer 30)
                      PUSH (Integer 12)
                      CALL "fun"
```

Now to understand what happens in *handleCall* we must look at what the function does. In simple terms it calls $funExec'$, a function which behaves almost exactly as our main executor except with some limits to what can be done inside a function (mainly to do with concurrent processes), and then moves onto the next instruction by updating the program counter and calling $exec'$ after updating the stack and memory.

Lets look at the code to be executed now in $funExec$ if we see that function fun looks like this;

```
func fun(a int, b int){
        ...
};
```

So the way in which the arguments would be passed is to pass the stack into $funExec'$ then pop of the items in the correct order. This can be show here;

```
-main executor;            -function executor;
...
PUSH (Integer 30)
PUSH (Integer 12)
CALL "fun"
            -- pass stack -->
                           FUNC "fun
                           POP "a"
                           POP "b"
                           ...
```

Now all your arguments are set up in memory to use within the function. It is important to note that when calling functions rather than just insert the function code into the current executing code I actually cause a new instance of an executor to run and passed around the stack and correct memory scopes.

This was a design choice as I felt that this approach gave me more control as to how I wanted to implement recursion and how I wanted functions to behave in general. I also felt it was a cleaner approach that more closely mirrored how I wanted functions to be treated.

Dealing with functions that return values is very similar to how we handled arguments, just the other way round. If we look at an assignment;

```
j = 13;  --> PUSH (Integer 13)
             POP "j"
```

You can see that when dealing with an assignment you are popping the top of the stack. Therefore if our assignment looked like this;

```
j = gun();
```

We must ensure that the value our function $gun()$ returns is left on top of the stack to be popped of our assignment. The instruction $RSTOP$ is used to signify a return value and will break out of $funExec'$. Therefore if we combine both our previous examples we can show how a function which takes in arguments will be called and how the stack is updated and passed around.

Lets look at the function;

```
func fun(a int, b int){
        Return a + b;
};
```

Then how it would behave if we use it as part of assignment;

```
j = fun(5, 8);
```

Looking at figure 5.3 we can see how this simple command is dealt with by the executor and the function executor functions.

This examples shows how stack frames are used to handle passing arguments and the returning of values in functions in my virtual machine.

Handling Recursion

Another interesting feature to implement in a stack based virtual machine is allowing function recursion. Function recursion is when you allow a function to call itself inside it's

```
                                    ]
  -main executor;              -function executor;
  ...
  PUSH (Integer 30)
  PUSH (Integer 12)
  CALL "fun"
              -- pass stack -->
                               FUNC "fun
                               POP "a"
                               POP "b"
                               PUSHV "a"
                               PUSHCV "b"
                               DO ADD
                               RSTOP
              <-- pass stack --
 POP "j"
```

Figure 5.3: Example of how functions are called, including passing arguments and returning a value

```
func fac(n int) int {
    if (n == 0) {
        return 1;
    };
    return n * fac(n-1);
};
```

Figure 5.4: Example of a recursive function fac, and example of how it works

function definition. An example of this can be seen below in figure 5.4 function that will give you the factorial of the number

This not only gives an example of how a recursive function can work, but also helps to show what a powerful tool recursion can be in solving different sorts fo problems.

To implement recursion I simply had to ensure that each stack frame generated for each new function was properly created and passed on. There was no special trick to allowing recursion other than ensuring my initial implementation of calling functions worked as intended then it was simply a question of allowing the function executor to be allowed to handle call function calls which would run a new instance of the function executor at the call step, ensuring the correct stack was passed and returned from $funExec'$.

5.3.3 Implementing Concurrency

How Goal Handles Concurrency

This section is a brief recap of what was explained in section 2.1.4 where we talked about how Goal was designed to handle concurrency. There are three important features of Goal that needed to be implemented and handled by my compiler and executor;

- *go* keyword, using this command with a void function starts a new instance of that method as a concurrent process, know as a sub routine.
- $Kill()$ command, this stops all concurrent processes
- $Wait()$ command, this will cause a program to wait at this point until all sub routines have stopped running.

There are also channels which are used to pass information between subroutines, these are described in greater detail in section 2.1.4 and in the Goal documentation.

How my Virtual Machine Handles Concurrency

Handling the creation and running of concurrent processes is definitely one of the more exciting features of my project. To understand how my executor deals with concurrency it is good to look at the type of my main executor function;

```
exec' :: Code ->  Int -> Stack -> Mem -> Bool -> [Channel] -> (Seq GoRoutine, Int) ->  EndParam
```

The bits we are interested in for handling concurrency are the types;

- $[Channel]$
- $(Seq\ GoRoutine, Int)$
- $EndParam$

These are types represent the way in which I pass around subroutines and channels in my executor.

If we first talk about how I handle channels we can then discuss subroutines in more detail . Channels are not held in memory but instead are stored independently. There are three instructions that can effect channels;

- $CHANNELn$, which creates a new channel
- $PUSHCn$, which takes whats on the top of the stack and pushes into onto the correct channel
- $POPCn$, which pops the correct channel and pushes the popped item onto the stack

These instructions can not only be executed by the main executor but also be executed within functions. Therefore it is important to look at our type $EndParam$ which our function executor will return;

```
EndParam  = ((Stack, Mem), ([Channel], (S.Seq GoRoutine, Int)))
```

You can see that $EndParam$ returns all the data structures used within my virtual machine, most importantly you can see that it returns a list of channels. Showing you can be updating your channels within functions and once that function has been executed it will return the $EndParam$ which will update the channels you have used in your functions.

Thus meaning code such as;

```
func main() {
        var c chan = Make(chan int);
        funA();
        j = <- c;
        Show(j);
};

func funA() {
        c <- 1;
        c <- 2;
        funcB();
};

func funC() {
        c <- 3;
};

OUTPUTS : 3
          DONE
```

Is perfectly valid due to the nature of how I execute functions and the data structure used to store and pass channel data.

Channels are regulated by a series of functions that are called each time a channel operation is performed. For example there are functions that check that you are not creating a new channel with the same name as a pre-existing channel and functions that check you are not trying to push and pop to a non existent channel.

We will now start talking about how sub routines are created and handled.

When a sub routine is created it's information is represented by the data type *GoRoutine*. This is a data structure that holds everything that a sub routine needs to run as independent process (excluding channel data). This structure is show below;

```
data GoRoutine   = Go Code Int Stack Mem
```

This gives every subroutine;

- Code to execute
- A program counter
- An independent stack
- A local memory

To handle multiple subroutines they are handled by the type ($Seq\ GoRoutine$, Int). The Seq monad is used to handle a list of items, I felt it was safer to use in order to maintain the order of items.

Now we know what format our subroutines will be passed round our executor we now need a way of executing them concurrently. Here it is important that to realize we are not creating truly concurrent processes in the sense that they run at exactly the same time. Instead what we are doing is interweaving the execution of processes to create the illusion of them executing in parallel.

To do this at each step of our executor we call the function *subRoutsHandler*. This will execute ten commands of on of our subroutines for every one command executed by out main executor.

You may have noticed that our subroutines are passed around in a tuple with an integer. This integer is used to keep track of which process is currently being executed.

To better understand what is going on we should look at how the $WAIT$ and $KILL$ commands are handled by the main executor, $exec'$;

```
...
KILL -> exec' c (pc+1) s m g ncs (S.fromList [], 0)
WAIT -> if (S.null gs) then
           exec' c (pc+1) s m g ncs (S.fromList [], 0)
        else
           exec' c pc s m g ncs (ngs, ngc)
...
  where
     con     = subRoutsHandler gs gc cs
     ncs     = snd con
     ngs     = fst (fst con)
     ngc     = snd (fst con)
```

You can see here how *subRoutsHandler* will be called and how the different values it needs to return are placed back into the virtual machine.

This code also show how $KILL$ simply removes all running subroutines and $WAIT$ stops until there are no more processes left to run, as every time a subroutine is finished running it is removed from the list.

To summarize a sequence of subroutines are handled by a separate executor at each step in my executor, this separate executor executes several commands before returning the updated information of the subroutines. The current subroutine being executed is kept track of using a counter.

Chapter 6

Testing

Testing is not only a very important part of a project like this, but also poses some interesting challenges for us.

6.1 Using Test Driven Development

In my introduction I explained that the way I approached implementing this project was to use test driven development. To recap, this is a development methodology where you write your tests first then write code to ensure your tests pass.

This process was very constructive in the first stages of my development, but became a bit frustrating as the project developed. Specifically when I started making changes to existing features in my intermediate representation. This was because a lot of my initial tests were wrote testing my back-end of my compiler, ensuring correct code generation and that programs ran correctly on my virtual machine. Therefor a lot of initial test programs where constructed using my IR.

The problem this caused was if I made any changes to the structure of my IR it could make many of my old tests redundant. The same problem could be seen when I moved on to making a parser. Where any changes in my languages syntax forced me to go back and update all previous tests.

To handle this problem that using TDD in an iterative process causes, I would when updating tests ensure that every test I created was still necessary. For example there was no point worrying about maintaining the $ifTest_2$ in figure 6.1 if my parser was now working and I was able to just write a program to test in Goal.

This meant that over time my tests changed from Haskell data structures, to simply writing an example program in Goal that I wanted my compiler to handle. Eventually this left me testing code by having numerous Goal source files my compiler had to run. Which led to the creation of my test suite.

6.2 Test Suite

For this project I felt the best way to test my compiler was to actually write code that I wanted to run on it, and see if it could. There are several ways I could do this. Either

```
ifTest_2     :: Prog
ifTest_2   = If (Val (Integer 0))
                (Assign "Luke" (Val (Integer 21)))

whileTest_3  :: Prog
whileTest_3  =
    Seqn [(Assign "l" (Val (Integer 5))),
          (Assign "j" (Val (Integer 0))),
          (While (Var "l")
                 (Seqn [
                       (Assign "j" (ExprApp ADD (Var "j")
                                   (Val (Integer 1)))),
                       (Assign "l" (ExprApp SUB (Var "l")
                                   (Val (Integer 1))))
                       ])
           )]
```

Figure 6.1: Examples of the sort of tests programs I would run to check the back end of my compiler.

by creating one large Goal program containing all the features I wished to test, or I could create a number of separate programs each one checking several key features.

I decided to go with the latter approach, creating a number of test programs which had standardized output that was easy to understand. These tests ranged from simple programs checking the behavior of conditional expressions to more complex programs checking features such as recursion.

The important part for each program was that it outputted a single line of text that told you if the program run successfully. This was in the format "testName : success/fail".

I then created a folder that held all of these tests so that I was able to write a Haskell program that could run all my tests so I could immediately see if any changes I had made to my code ended up causing problems with my previous tests.

This explains how ended up creating and running a test suite to run on my compiler. I would then check by using the standardized output of my test, to see if any tests had failed.

6.3 Quickcheck

Another method I used to test my compiler was Quickcheck. Quickcheck is a Haskell library used for random testing of program properties[1]. It is relatively simple to use and is very helpful in testing functions that are required to have certain properties.

They way it works is you define some property you want to be true then use Quickcheck to check that property 100 times with random input. For example I needed to create a function that would add 2 integers, therefore I could check the following property;

```
propAdd l j  =
      ((addNumbers (Integer l) (Integer j))==Integer (l+j))
```

[1] https://hackage.haskell.org/package/QuickCheck

```
func main() {
    l = fun(20);

    if (l == 20){
        Print("testSeven : success");
    } else {
        Print("testSeven : fail");
    };
};

func fun(n int) int {
    l = 0;
    for (n > 0) {
        l++;
        n--;
    };
    return l;
};
```

Figure 6.2: Examples of a test program you would find in my test suite

Then I would have to make the call;

```
quickCheck propAdd
```

Which will give me the output;

```
"+++ OK, passed 100 tests"
```

I used Quickcheck primarily to tests functions used in my virtual machine. I found it particularly useful when testing stack operations.

There were some problems with using Quickcheck though, specifically in checking more complicated functions. For example I would only use Quickcheck to test functions with limited input and output. I would not use Quickcheck to test my entire virtual machine but I would use it to test specific functions used within my virtual machine.

This was because I felt that it was better to use a test suite to test the main aspects of my compiler then use Quickcheck as a supplement to help me test specific functions.

6.4 Bugs

As with any large scale project testing through up some interesting bugs within my code. Rather than outline every bug I discovered I will go over two of the more interesting problems I came across and how I solved them.

Specifically I will talk about two bugs that caused me problems that I felt although I was able to provide solutions for, I maybe could have tackled better.

6.4.1 Haskell's Lazy Evaluation

Lazy evaluation is where you only evaluate expressions if you need to. It is the most common method used for executing Haskell and caused me some problems. It means that sometimes you can not be 100% certain of how you program will be executed.

The way I discovered the bug was when I was testing void function calls. I had the problem that the following code;

```
func main() {
        fun();
};
```

Would do nothing regardless of the definition of $fun()$. This got me confused but Final I found out that by making the small change;

```
global j = 0;

func main() {
        fun();
        Show(j);
};
```

Would cause my program to behave as I wanted. This lead me to thinking there was a problem with how Haskell was evaluating my virtual machine code. The issue seemed to disappear if there was some interaction with the stack and some forced output.

The good thing about having my compiler broken up into three main sections means you can run each part individually until you find the problem. Hence why I was able to quickly realize the problem was in my virtual machine.

Unfortunately I found this problem quite late on in my project, which lead me to increase the size a scope of my test suite. Therefore I was not able to come up with the sort of elegant solution I would have liked to.

Instead I had to solve this problem by forcing my virtual machine to perform a stack operation and give some output as the last thing it does. This can be seen by the creation and use of the $TRICK$ instruction which forces the virtual machine to perform something even if most of the work is going to be done by the function executor.

6.4.2 Output

Another problem I had was getting my virtual machine to provide output. Ther are two commands that need to be dealt with in this respect;

- $SHOW$, Which will take the item on top o the stack, pop it and output it.
- $PRINTs$, This is the only command which can handle strings, it will simply output the string s.

The problem I had initially was the function I created to generate output was not working. I tried to make use of the $unsafePerformIO$ function to handle generating output without changing my virtual machines type declarations but unfortunately it was to no avail.

The solution I came up with was to make use of the Haskell *trace* function this can be seen below;

```
SHOW -> trace (getNumberAsString (head s))
              (exec' c (pc+1) (pop s) m g ncs (ngs, ngc) )
```

Although this does solve the problem of providing output for my virtual machine it is something I would have liked to revisit. This is due to the unsafe nature the *trace* function is only meant to be used in debugging rather than for production code.

Chapter 7

Closing Statements

7.1 Conclusion

In conclusion during the course of writing this dissertation I achieved what I set out to do. I created a compiler that can handle a concurrent programming language. On top of that I also manged to create my own programing language and a virtual machine to run it.

With regards to the language I created I feel that it accurately portrays it's inspiration from Go whilst at the same time having a lot of individuality. I feel the way I designed Goal to handle concurrency not only makes it easy to use but useful for the creation of concurrent programs. I also think that Goal can ended up having some practical uses, especially as a tool for teaching concurrency due to its easy to understand syntax and how simple it is to implement concurrent programs.

In terms of the dissertation write up I felt that overall this project can be used for people in the future who are interested in implementing there own compiler in Haskell. This document strives to serve as a tool people can use in the future to learn more about implementing compilers and virtual machines in Haskell.

This project was never setting out to be ground breaking, but I aimed to create something that could be used to help people in the future. Primarily by writing a document that provided people with an insight into how to go about creating the different aspects of a compiler and virtual machine in Haskell, and to highlight some off the different approaches available to someone tackling this sort of problem in the future.

This project started as a means for me to learn more about compilers, language design and Haskell. In the end I felt not only did learn about these topics but also created something that could help others to learn about the same things I ended up learning about.

I hope that Goal goes on to find some use in the future either as a teaching tool or as a future project for someone to expand on. I also hope that this document helps other people answer the sort of questions I had at the start of this project.

7.2 Reflection

The initial aim of this project was to create a compiler in Haskell for a concurrent programming language. Over time the project has expanded to include creating an executor and even designing my own language to compile. Although there were changes to the process, I

still managed to achieve the primary goal of implementing a concurrent compiler.

There were several advanced features in my compiler implementation but on reflection I would say the project can be boiled down to a few interesting aspects;

- The design and creation of a new concurrent programing language, Goal.
- Parsing using monadic parser combinators.
- Allowing function recursion.
- Allowing the running of concurrent processes within my compiler.

There are many aspects of this project that I am proud of, but being able to implement concurrency is obviously the highlight.

Although this project was a success I would argue there are several areas for expansion or possible improvements, as with most programming languages. Throughout this write up I have touched on several aspects that could be expanded and I feel on reflection three main features would be the first to be added;

- Expanding the type system
- Allowing arrays
- Pointers

Adding types, as was discussed, came down to an issue with parsing and deciding that other features where more important, even though my executor is set up to handle extra numerical types, I decided not to ship something half done.

With pointers and arrays these were two features that interested me but I felt again I was more interested in focusing my efforts elsewhere. Even though these feature could be areas to expand the project I do not feel without these features Goal is any less useful. As there are still work rounds.

In reflection I feel this project was very successful not just in what I initially set out to achieve in creating a concurrent compiler, but also it succeeded in creating a new programing language, Goal, and an executor to run it.

Importantly I feel although this project is complete it has been created in such a way that if someone was interested in expanding Goal it would only take an understanding of Haskell and Monads and a bit of enthusiasm, to start adding new features. Which is something I'm glad I achieved.

Overall I am very pleased with the compiler I created. I feel the language I initially created as a tool to explore compiler design and concurrency has, due to the completeness of my compiler implementation, changed from a simple experiment to a complete language with it's own uses. Which is a very good side effect of a what started as a simple exploration into creating compilers in Haskell.

Bibliography

[Aho et al.(2007)Aho, Lam, Sethi, and Ullman] AV. Aho, MS. Lam, R. Sethi, and JD. Ullman. *Compilers; Principles, Techniques & Tools*. Pearson, 2nd edition, 2007.

[Chattapoadhyay(2005)] Santanu Chattapoadhyay. *Compiler Design*. Prentice-Hall of India, 1st edition, 2005.

[Cooper and Torczan(2012)] Kieth D. Cooper and Linda Torczan. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2012.

[Hutton and Meijer(1996)] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[Hutton and Meijer(1998)] Graham Hutton and Erik Meijer. Functional pearls monadic parsing in haskell, 1998.

[Jackson(2015)] Luke Jackson. Goal programming language documentation, 2015.

[Terry(1997)] P. D. Terry. *Compilers and Compiler Generators; An Introduction With C++*. Thompson Computer Press, 1st edition, 1997.