Creating a Compiler and Executer in Haksell for Goal, a Simple Concurrent Programming Language Based on Google Go.

Luke Jackson

April 13, 2015

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 executor 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 & Executors

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 Ullman, 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 an executor is something that executes code written in a given programming language. You will usually only see executors created to execute low level languages given the smaller, and more precisely defined, instruction set. As is the case with the executor 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 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.

1.1.3 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 and even some newer ideas such as splices, which we will talk about later.

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.

1.1.4 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 how 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 & Methodologies Used

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
- Execution, using a Stack Based Virtual Machine

While developing my compiler and executer 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 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.

A key part of Goal's syntax is that each command must end with a semicolon, 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)].

2.1.2 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 3 main types; Integers, Booleans and Strings. 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;
i = "hello";
```

There would be no compiler or run time errors in this code and your variable i would have the value "hello" as each new assignment writes over the old value. 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.

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.

2.1.3 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.

2.1.4 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.

2.1.5 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. 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.

2.2 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

Bibliography

[Aho et al.(2007)Aho, Lam, Sethi, and Ullman] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers; Principles, Techniques & Tools*. Pearson, 2nd edition, 2007.

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

[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.