Dynamic Meets Static: Ahead-of-time Compilation for Lua

Dominic J. Binks

Supervised by Professor Graham Hutton

School of Computer Science

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text: D. J. B.

18/04/2024

Abstract

Ahead-of-time compilation is the process of fully compiling a source code program to machine code before running the program, eliminating the need for any runtime environment when executing the code. While ahead-of-time compilation can be used for dynamic languages, it poses a significant challenge due to having a lack of knowledge when compiling. Therefore, dynamic languages languages tend to be interpreted or just-in-time compiled. By converting a dynamic language to a statically typed language, the process of creating an ahead-of-time compiler is greatly simplified, as the compiler can deduce the types of variables and expressions at compile time. Type inference can then be used to allow the programmer to omit the types of variables when writing a program, allowing the source code for this static variant to be extremely similar to the source code for the original dynamic language. In this dissertation I have explored this approach by developing a compiler for a statically typed variant of the Lua programming language. Benchmarking has shown this approach to produce good performance when compared to existing implementations of Lua, showing that it's possible to reduce the time it takes to run programs while maintaining a level of compatibility with other implementations.

Contents

1 Introduction

1.1 Compilers and Interpreters

A compiler is simply a program that takes a program written in a source language and translates it to an equivalent program in another target language [1]. Traditionally, the source language is a high-level language that is easy to program in, while the target language is usually a low-level language that is closer to the machine code that is run on the CPU. During this process, high-level constructs available in the high-level language must be converted into a series of more primitive instructions supported by the low-level language. This leads to the low-level language version being more verbose and less human-readable.

The most common type of a compiler is an ahead-of-time(AOT) compiler, which fully compiles a program into machine code before it is executed. AOT compilers can generally be split into two distinct parts: the front-end and the back-end, with both of these parts being broken down further into a series of separate steps. The first step is scanning, which involves breaking up the source code program into distinct pieces called tokens, which consist of things such as names, numbers, operators, and symbols such as parentheses. Parsing is then performed, which takes these tokens and arranges them into a tree structure called an abstract syntax tree(AST). During parsing, analysis is performed to ensure the source code program is well formed. If the source code program isn't well formed, then the tokens cannot be arranged into an AST, meaning a syntax error must be raised. For example, an expression given to a function call is checked to ensure it's a valid expression, and isn't just a list of operators without any operands. After parsing, semantic analysis is performed, usually taking the form of type checking or type inference. Type checking involves making sure that expressions have the correct type for the operations they're being used with, such as ensuring the operands used when performing addition are numbers. Type inference on the other hand involves working out the types of different expressions based upon context in which they're used, such as deducing a variable is a number if it's used as an operand during addition. In the case where these deductions contradict each other, a type error is raised. Lastly, the front-end takes the AST and uses it to produce some intermediate representation(IR) of the program. The back-end of the compiler then takes this intermediate representation and performs optimisations on it, such as loop unrolling, which converts a loop into a series of repeated instructions that represent multiple instances of the loop body. After performing optimisations, the back-end then produces machine code for the program that can be run on a computer's CPU, typically in the form of an executable file.

In contrast, an interpreter executes a source code program by directly performing the operations specified by each line of code in the program. This means that both a source code program and interpreter are needed to run the program, instead of just an executable file. As each line of code is executed, the interpreter updates its runtime environment based on the operation performed. For example if a variable is assigned a value, the runtime environment will be updated to store the variable's new value. This typically results in a

slower execution time due to the increased overhead associated with managing the runtime environment as each line is executed. Despite this, interpreters can resemble compilers, as they often initially compile the source code program into a bytecode representation. This bytecode representation can be more efficiently interpreted by a virtual machine, with this approach being taken by languages such as Java and Lua.

In addition to ahead-of-time compilers and interpreters, just-in-time(JIT) compilers exist as a combination of the two approaches. Like interpreters, JIT compilers also use a runtime environment to keep track of the state of the program as it runs. Despite this, instead of an interpreter performing the operation specified by a line of code, the source code or bytecode is converted to machine code "just-in-time", allowing for the operation to be performed more quickly and efficiently. Additionally, JIT compilation uses optimisations such as caching frequently used machine code routines to avoid having to recompile them. In many cases, this approach provides the best of both worlds, often leading to just-in-time implementations of languages being faster than their ahead-of-time and interpreted counterparts.

When compiling a program, one of the key tasks the compiler must do is figure out the type of each variable, so the correct amount of memory can be set aside to store values of that type. A dynamic language is a language that allows values of different types to be stored in the same variable throughout the program. This poses a major challenge to a compiler, as it means that the compiler is unable to know how much memory a particular variable requires. This contrasts with static languages, that only allow a variable to store values of a particular type. This means that the compiler can easily deduce what type of value a variable holds, and allocate the right amount of memory for that type. Therefore, a static language is much easier to compile ahead-of-time compared to a dynamic language. Dynamically typed languages can be compiled ahead-of-time using different methods, with these methods mainly revolving around adapting an existing interpreter. For example, Gualandi and Ierusalimschy showed how partial evaluation can be used to compile Lua ahead-of-time [8]. Partial evaluation involves taking different variants of the same piece of code, and then using the correct variant for the given value at runtime. In general though, these techniques are much more sophisticated than those required to create a compiler for a static language from scratch.

Overall, AOT compilers are generally used for static languages, while interpreters are used for dynamic languages. Despite this, there are cases where AOT compilers are used for dynamic languages, and interpreters for static languages. The decision of which approach to take is ultimately down to those implementing the programming language, and what they believe will help them fulfil their needs.

1.2 LLVM

LLVM is a collection of technologies that aim to aid those constructing compilers, with a particular focus on reusability [20]. LLVM provides an assembly language that can be used as an intermediate representation for compilation, that I will refer to as LLVM IR throughout this dissertation [19]. It uses a C style syntax that is human readable, but doesn't provide higher-level constructs like conditional statements or loops, instead

requiring the use of labels and branching.

The @ prefix denotes a global identifier used for functions and global variables, with global variables defining regions of memory that are allocated at compile time. This means the types of global variables must be known at compile time in order for the correct amount of memory to be allocated. As global variables just denote regions of memory, using them is similar to how you would use a pointer in a language like C.

The % prefix denotes a local identifier used for things such as register and label names, and doesn't require specifying the type of the value stored. Another major difference to C and assembly languages is that LLVM IR uses Static Single Assignment. This means that a register can only be assigned a value once. To implement mutable variables you must first allocate some memory, storing the pointer returned in a register, using this pointer at later date to access the memory. Luckily there are an unlimited number of registers available, with the registers being mapped to the registers and memory available for the particular architecture the code is compiled for.

LLVM IR has a number of types available, with the main ones of interest being **i1**(1-bit integer, used for booleans), **i32**(32-bit integer), and **double**(double-precision floating-point). Additionally, array types take the form **[l x t]** where **l** is an integer representing the length of the array, and **t** is the type of the elements. ***** can be appended to the end of a type to specify a pointer to that type, with multiple indicating a pointer to a pointer. Many instructions are provided such as **store** and **load** for storing and loading data to and from memory. **fadd** and **fmul** are used for floating point addition and multiplication. Finally, the **call** instruction can be used to call a function, allowing for easy code reuse.

1.3 Lua

The Lua Programming language has existed since 1993 [15], and has been widely used in a variety of settings such as programming IoT devices and adding custom content to numerous video games [21]. The standard implementation of Lua compiles source code into Lua bytecode, which is then interpreted using a virtual machine. A detailed description of the Lua programming language can be found in the book "Programming in Lua" [14]. I have picked out the key points from this book in order to give a brief overview of the language.

Lua has the typical features you would expect from an imperative programming language, such as operators, variables, conditional statements, loops, and functions. One notable difference is the absence of traditional arrays, with Lua instead providing associative arrays(also known as maps or dictionaries) in the form of tables. Tables can be used almost identically to arrays in other languages by using a numerical key. A slight catch is that the convention is to start indexing arrays from 1, in contrast with the 0-indexing used by the vast majority of other programming languages. Lua also has some more interesting features for more advanced use cases such as coroutines which provide similar functionality to user threads. Other useful functionality is also provided through standard libraries, such as the mathematical and I/O libraries. Lastly, Lua has an extensive C API which facilitates interoperability between the two languages, allowing Lua code to call C code and vice versa.

In terms of types, Lua has eight basic types: boolean, number, string, nil, function, userdata, thread, and table. The boolean, number, and string types are self explanatory, with the number type having integer and float subtypes depending on the number stored. The nil type is the type of the nil value, which is returned when there is no other appropriate value to return. This occurs in situations such as accessing an array index that hasn't been assigned a value. The function, userdata, thread, and table types are more interesting. Values of these types are objects, with variables of these types not actually containing these values, only references to them. The function, thread, and table types are used for functions, coroutines, and tables respectively. The userdata type is for allowing arbitrary C data to be stored in Lua variables.

Lua is a dynamically typed language, allowing the types of variables to change dynamically at runtime. Additionally, Lua has weak typing as automatic type conversion is performed. For example Lua would evaluate the expression **"123" + 4** as 127, implicitly converting the string **"123"** to the number 123. Finally, while Lua can be used to write procedural code, it is a multi-paradigm language. Object-oriented programming can be achieved by using tables to store functions and member variables, while first-class functions allow functional programming.

1.4 Project Overview

The overall idea behind this project was to create an ahead-of-time compiler for a statically typed variant of Lua, and then compare it to existing implementations of Lua. By compiling Lua programs ahead-of-time, my goal was to speed up the execution time by eliminating the need for a runtime environment. Additionally, I decided to select a dynamic language in order to see how converting it to a statically typed language might impact interoperability. Lua was a good candidate for this transformation as it is a dynamic language with simple syntax and strong documentation for the language itself. The Lua reference manual provided a clear description of how the language works, alongside the full context-free grammar for the language [24]. The variety of implementations of Lua also provided many opportunities to compare the performance of my compiler against other implementations, allowing me to know if I achieved my goal.

While I could've based my compiler off of an existing interpreter for Lua using the techniques I previously mentioned, I decided to build my compiler from scratch. I did this in order to demonstrate whether a simpler approach could provide a significant gain in performance. This approach is much more simplistic as it doesn't require in depth knowledge of an existing interpreter for the language. It simply requires a basic understanding of how to construct a compiler, alongside basic knowledge of type inference. Furthermore, if this approach proved successful, then it could more easily be adopted by others, allowing other languages to obtain similar gains in performance. This extra performance could be used to run programs more effectively on older or slower hardware, alongside offsetting less efficient implementations of programs. Dynamic programming languages such as Lua are generally more beginner friendly than statically typed languages, as you don't have to think about the types of values. Therefore, it is likely that less efficient implementations may be used due to the programmer's lack of experience.

When tackling this project I broke the work down into two phases: developing the compiler and evaluating it. The first step was creating a compiler that used explicit typing by implementing a scanner, parser, and code generator. The next step was adding type inference to allow implicit typing, to provide interoperability with with existing implementations. After adding type inference, my compiler was ready for evaluation, which was primarily achieved through the running of benchmarks. Overall, the steps taken to achieve the goal of this project are best outlined by my aims and objectives.

1.5 Aims and Objectives

1. Develop an ahead-of-time compiler for a statically typed variant of Lua
   (a) Create a compiler that uses explicit typing
   (b) Add type inference to allow implicit typing
2. Evaluate the benefits and drawbacks of this compiler compared to existing implementations of Lua
   (a) Run a set of benchmarks to observe the performance of the different implementations
   (b) Analyse the results of these benchmarks by comparing the performance of the different implementations
   (c) Investigate the interoperability of my implementation with existing implementations of Lua

1.6 Alternative implementations of Lua

There are many implementations of Lua that are similar to my compiler in the sense that they focus on providing interoperability with existing implementations, and don't aim to expand the language with extra features. LuaJIT is an alternative implementation of Lua that uses a JIT compiler [27], resulting in better performance compared to the reference implementation of Lua [28]. While I could've chosen to develop a JIT compiler, they are significantly harder to implement than an AOT compiler. As a novice to compiler construction at the beginning of this project, I decided that it was a wise decision to stick to creating an AOT compiler.

LuaAOT is a based on Lua 5.4.3 and allows Lua programs to be compiled ahead-of-time [7]. LuaAOT achieves this by using the partial evaluation technique for converting an interpreter into a compiler, and essentially compiles a Lua program by packaging it with the necessary parts of the interpreter needed to run the program. LuaAOT takes Lua bytecode as input, and produces a C code output that can be compiled with existing C compilers such as GCC or Clang. Therefore the key difference between this implementation and my compiler is that I wanted to see what could be achieved without using partial evaluation or other advanced techniques.

Ravi is based on Lua 5.3 and implements a large subset of the language alongside providing the programmer with the option of static typing [23]. It's compiler is written in C and

provides both AOT and JIT compilation, using a custom intermediate representation [22]. Like LuaAOT it compiles Lua programs to C programs that can be compiled with an existing C compiler. A key difference of my compiler is the choice of using LLVM IR as the target language instead of C. The key benefit of this is reducing the compilation time as there are less intermediate steps, as my compiler goes straight to LLVM IR without going through C. This contrasts the Clang C compiler which compiles C code to LLVM IR first, and then compiles the LLVM IR into an executable.

Teal aims to be a typed dialect of Lua, and compiles programs to Lua programs that can be run using existing implementations of Lua [26]. In this case, the addition of static typing is for help ensure the programmer constructs correct programs that are type safe, in contrast to my aim of improving performance. Furthermore, as the compilation target is Lua, another implementation of Lua has to be used to run the program.

1.7 Lua Variants

Nelua is a variant of Lua which focuses on trying to be a systems programming language that resembles Lua [2]. Like Ravi, it also compiles to C code, which makes sense given many of the additions Nelua provides directly match features of C. The addition of features such as manual memory management has made it more applicable for systems programming in exchange for compatibility with other Lua implementations. Titan is similar to Nelua, as it's another Lua-like language with static typing, with a compiler written in Lua [31]. Like Nelua it has added extra features, with plans to support explicit classes in the future. While these implementations are similar to Ravi and Teal in the sense they've added static typing to Lua, they are very much focused on moving away from standard implementations of Lua. Their goal is to add extra features to serve a different purpose, in contrast with my goal of maintaining compatibility.

2 Design

2.1 Language Features

In general, the key goal of my compiler was to maintain interoperability with existing implementations of Lua, with this being at the forefront of my mind when making design decisions. Be that as it may, due to the scope of this project I had to make some decisions that resulted in reduced interoperability. These decisions were made to ensure that I was able to deliver a robust piece of software that met the aims and objectives I set out for myself within the time I had available for this project. In general I had to reduce the language features supported by my compiler to the basic key features needed to implement some interesting algorithms. Being able to implement some interesting algorithms was an important target as it meant that my compiler could be used to compile non-trivial programs that could be used for benchmarking.

My compiler supports expressions of values and operators, variables, conditional statements, loops, arrays, printing, and functions. These were all the features I needed for the benchmarking programs I chose. Some notable omissions from this list are generic for loops, first-class functions, goto, local variables, and errors. While it would've been nice to support all these features, I had to spend time on other aspects of the compiler such as type inference. Furthermore, I felt that things like coroutines, metatables, and providing a C API were well beyond the scope of what could be achieved within this project.

For experienced Lua programmers, one of the most notable differences is the lack of idioms. Two good examples of Lua's idioms are being able to express **if not x then x = v end** as **x = x or v**, and **a ? b : c** as **(a and b) or c**. In the case of these idioms, if the values used are booleans, then the statements are equivalent without needing to handle them explicitly as idioms. In the case where the values are other types, or a mixture of types, they aren't available as the type system only allows the **and** and **or** operators to accept boolean arguments. On a similar note, while the **nil** value is implemented, it doesn't provide much use. In situations where there isn't an appropriate return value, **nil** isn't returned as it wouldn't match the return type. Instead, the majority of the cases where **nil** would be returned instead return a different but appropriate value such as 0 or result in undefined behaviour.

While my compiler supports Lua's arithmetic and boolean operators, it doesn't support the bitwise operators, or # and **..** for string length and string concatenation respectively. In the case of bitwise operators, LLVM IR only has instructions for bitwise operations on integer types. As I store numbers as doubles I would've had to handle checking that the value has an integer representation alongside converting it to and from an integer value. For the string operators, I would've had to adjust my approach of handling strings as fixed values.

Lastly, my compiler doesn't support any of Lua's libraries as I felt that it was significantly beyond the scope of this project to do so. One key function that is missing as a result

```
block ::= { stat } [ 'return' [ exp ] ]
stat ::= 'function' name '(' args ')' block 'end' | 'print' '(' exp ')' | 'do' block 'end' |
         'if' expr 'then' block { elf } [ els ] 'end' | 'while' exp 'do' block 'end' |
         'repeat' block 'until' exp | 'for' name asn ',' exp stp 'do' block 'end' | break |
         name [ '[' num ']' ] asn | name '(' explist ')' | ';'
args ::= name typ {',' name typ } | ϵ
elf ::= 'elseif' expr 'then' block
els ::= 'else' block
asn ::= typ '=' exp
typ ::= ':' type | ϵ
stp ::= ',' exp | ϵ
explist ::= exp { ',' exp } | ϵ
exp ::= nexp exp'
exp' ::= 'or' nexp exp' | ϵ
nexp ::= cexp nexp'
nexp' ::= 'and' cexp nexp' | ϵ
cexp ::= aexp cexp'
cexp' ::= '<' aexp cexp' | '<=' aexp cexp' | '>' aexp cexp' | '>=' aexp cexp' |
          '==' aexp cexp' | '∼=' aexp cexp' | ϵ
aexp ::= mexp aexp'
aexp' := '+' mexp aexp' | '-' mexp aexp' | ϵ
mexp ::= un mexp'
mexp' ::= '∗' un mexp' | '/' un mexp' | '//' un mexp' | '%' un mexp' | ϵ
un ::= ept | '-' un | 'not' un
ept ::= fin '^' ept | fin
fin ::= num | name '(' explist ')' | name '[' exp ']' | name | str | bool | nil | '{' explist '}' |
        '(' exp ')'
```

Figure 2.1: Context-free grammar for the Lua variant supported by my compiler

is **io.read()** which is used for user input. Therefore, a program must have all the values it needs to run within the source code itself. Even if I was to implement this function, it's usefulness would still be limited as it always returns a string value of the input. As I didn't implement anyway to convert between values different types, or implement implicit type conversions, the usefulness of these user inputs would be limited.

Overall, despite these design decisions deviating from traditional implementations, my compiler can still be used to compile programs that work with existing implementations, as can be seen in the evaluation section of this report. Furthermore, I believe the variant of Lua supported by my compiler is best summarised by the context-free grammar(CFG) that can be seen in figure 2.1. This CFG is represented using extended Backus-Naur form, and uses the option and control forms, denoted with [] and {} respectively [29].

The large number of non-terminals in the CFG are needed in order to impart the correct operator precedence levels, which correspond to traditional operator precedence levels for Lua. The specific precedence of each operator supported by my compiler is denoted in figure 2.2.

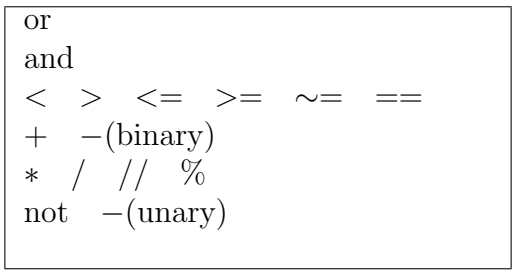

Figure 2.2: Precedence levels of operators my compiler supports, from lowest to highest priority

2.2 Types

The key feature that differentiates my implementation from the standard Lua implementation is static typing. This means that a variable can only store values of a single type throughout the program. Despite dynamic languages allowing different types, it is generally considered bad practice to use a variable to store values of different types. This is because it makes it harder to debug and understand code as a variable could have a different type of value than what you expect it to at a given point. Therefore, this limiting factor of static typing typically isn't much of an issue for well constructed programs.

My compiler uses 4 of the basic Lua types: nil, boolean, number, and string, with values of these types generally working as you would expect them to. One minor change is that the number type doesn't have separate integer and float subtypes, with all numbers stored as an double. As none of the other language features I implemented would be able to take advantage of the different subtypes, I felt that this was a justified omission. Values of the string type also differ slightly as they are immutable, acting the same way as strings found in languages like Java, in contrast to traditional Lua strings which are mutable. This too isn't an issue as my implementation doesn't provide a way to mutate a string.

Furthermore, this means my compiler doesn't use the other 4 basic Lua types: userdata, function, thread, and table. The userdata and thread types aren't used as my implementation doesn't allow arbitrary C data to be stored, or threads to be created. It also doesn't use the table type as my implementation uses arrays instead of tables. As LLVM IR is an assembly language, implementing tables would entail a serious endeavour that isn't the focus of this project. Therefore, I decided to use traditional arrays for my implementation, as they have strong support in LLVM IR whilst still allowing my implementation to be used to compile interesting algorithms. In my implementation, an array type is indicated by appending **[i]** to the type of the elements contained in the array, where **i** denotes the number of elements. For example, **number[4]** would indicate an array of 4 numbers.

In addition to using static typing over dynamic typing, my implementation is more strongly typed than the standard implementation, due to a lack of implicit type conversions. This means that for example, while the code **"123" + 4** would return 127 when interpreted by the standard implementation, my compiler would throw an error as **"123"** is a string not a number.

2.3 Type Inference

Type inference is simply the process of deducing the types of expressions in a source code program. In the case of my compiler, it provides a key pillar in providing interoperability with existing Lua implementations. While my implementation allows the programmer to specify the type of a variable by appending it to the variable name after a **:**, the standard implementation doesn't. Lua programs intended for the standard implementation don't have to worry about specifying this typing information as it's not necessary for the interpreter's operation. Therefore in order to allow Lua programs to work with my compiler alongside other implementations, my compiler must be able to infer the types of variables from context.

Performing type inference requires establishing a type system, which simply specifies the rules used to assign types to expressions [30]. My compiler uses a type system based on the Hindley-Milner type system, using the Hindley-Milner algorithm to deduce types, and is based upon lecture notes by Lerner [16]. Many resources that explain type inference take a very formal approach, given that type inference itself is heavily based in the mathematics of type theory. These lecture notes on the other hand provide a much more practical approach to type inference, which I found to be much easier to understand, and more useful for actually implementing type inference.

For values such as the number **1.2** or the boolean value **true**, the type of these values is clearly number and boolean respectively, with these types referred to as type constants. When a variable is first encountered we don't know what its type is. This means that we can't assign it a type constant and must instead assign it a type variable as a form of placeholder for it's actual type. As we continue to go through the program inferring the types of different expressions, we are able to establish more constraints on these type variables. For example after assigning a variable a type variable, we may be able to conclude the variable contains an array of elements of some unknown type. Once the whole program has been traversed, this process collects and solves enough of these constraints to infer the type of each variable, function, and expression used when executing the program. However, it may be the case that at the end of this process the types of certain functions and expressions may still be expressed using type variables. This doesn't pose an issue, as these cases only arise for functions that are never called, and for the variables and expressions within these functions. Therefore this typing information isn't needed as we don't need to generate code for these functions as they're never called in the program.

In addition to type variables and type constants, the type system my compiler uses also uses functions types and array types to represent the types of functions and arrays respectively. A function type is specified as a list of the types that represent the function's arguments, alongside a single type that represents the type of the return value. The array type is specified as a type for the type of the elements stored in the array, alongside an integer that represents the number of elements the array stores.

When performing type inference, a type environment is used to keep track of the current type of each variable and function, and is simply a mapping of the name of the variable or function to its current type. As more information is deduced about the types of certain variables and functions the type environment is updated. My type system also uses a

type scheme environment to represent the types of operators. A type scheme is similar to a type, consisting of a list of unbound type variables, alongside a type that uses these unbound type variables. When inferring the type of an expression that uses an operator, a new function type for the operator is derived from the type scheme by replacing the unbound type variables with fresh type variables. This new type is then used to determine the type of the expression. The types of operators are represented as function types as an operator is simply an infix function that takes values of certain types and returns a single value of a particular type. This provides polymorphism as different instances of operators have different function types in different circumstances. A new function type is always constructed using fresh type variables when an operator is encountered. For example, this allows the = operator to be used in the both expression **1 == 1** and the expression **"abc" == "abc"**. This contrasts with expressions, variables, and functions in this type system which all have monomorphic types, which means the types are fixed throughout the duration of the program.

2.4 Structure

The structure of my compiler follows the structure of a compiler front-end I previously mentioned in the introduction. The input high-level language is a Lua program that is compatible with my variant of Lua, and the lower-level output is LLVM IR. As LLVM IR is an intermediate representation, my compiler is just a front-end as an existing LLVM IR compiler has to be used to produce an actual executable file. LLVM IR was a good choice for the compilation target as it supports compiling code for many different hardware architectures. This helps offset a key drawback of compilers compared to interpreters, which is only being able to run the code on CPUs that use the architecture the executable is compiled for.

3 Implementation

3.1 Overview

My implementation of a Lua compiler has been constructed using the Haskell programming language. Haskell has a heritage of being used to implement compilers and interpreters for a wide range of languages [9]. Furthermore, using a functional programming language like Haskell allowed me to create recursive decent parsers using parser combinators, with these parsers forming an instance of a monad [13]. Recursive decent parsing involves recursively parsing each non-terminal symbol in the grammar in a top-down fashion, with the base case for recursion being when a terminal symbol is reached [3]. An AST can then be built up from the results of these recursive calls. This parsing technique lends itself to being implemented Haskell due to the inherent recursive nature of the language itself. Parser combinators allow for a "more powerful" way of parsing compared to other methods, with the main benefit being that full backtracking is handled if necessary [11]. Using parser combinators in a functional programming language like Haskell allows parsers to be developed with ease, with the code closely relating to the context-free grammar of the language being implemented.

I initially created a simple compiler for the arithmetic expressions supported by Lua, directly drawing insipration from the approach taken in the compilers module offered by the University of Nottingham [4]. I then expanded this compiler to support variables, conditional statements, loops, functions, and arrays. This was done by iteratively updating the scanner, parser, type inference, and code generator. This approach meant that each language feature could be tested independently, aiding debugging. In the next sections I will give a general idea of how each part of the compiler was implemented. While I could explain each individual function in detail, this would be extremely excessive. Therefore, I will highlight the most noteworthy parts, that demonstrate the general techniques and approaches I've taken. This should provide a suitable foundation to dive into the source code if you want to fully understand how everything works.

3.2 Scanner

The first thing I did when constructing the compiler was define a type for the tokens that are scanned by the scanner.

```
data Token = Num Double          -- number
           | Op Operator         -- operator
           | Par Parenthesis     -- parenthesis
           | Str String          -- string
           | Nme String          -- variable/function name
           | Bool Bool           -- boolean
           | Tp TypeConstant     -- type
```

```
            | SCol              -- ;
            | Col               -- :
            | Eq                -- =
            | Cma               -- ,
            | Kword Keyword     -- keyword e.g. if, else, etc.
            deriving (Eq, Show)
```

This type's data constructors in turn use values of additional types to specify the exact token, such as values of the **Keyword** type for specifying the exact keyword that was scanned.

```
data Keyword = If      -- if
             | Then    -- then
             | ElseIf  -- elseif
             | Else    -- else
             | End     -- end
             ...
             deriving (Eq, Show)
```

I then defined a **Scanner** type that represents a scanner that takes a string and produces some output alongside the string left over. This output is wrapped in the maybe monad in order to handle unsuccessful operations. I was then able to define this type as an instance of the monad and alternative type classes, with the alternative type class being crucial for creating combinators. This step and subsequent steps are heavily based around examples given in the "Monadic parsing" chapter of the book "Programming in Haskell" [12].

```
newtype Scanner a = S(String -> Maybe (a, String))
```

Afterwards I constructed some helper scanners based around this type, such as a scanner for getting the front character from the string. This scanner also demonstrates the need for wrapping the result in the maybe monad, as it isn't possible to get a character from an empty string.

```
sFront :: Scanner Char
sFront = S(\s -> case s of
                   []     -> Nothing
                   (c:cs) -> Just (c, cs))
```

With these scanners in hand, I was then able to define a scanner that takes the string of a source code program and picks out all the tokens it can.

```
scanner :: Scanner [Token]
scanner =
  do
    sSpace
    ts <- many (
```

```
            sOp <|> sPar <|> sSCol <|> sCol <|> sEql <|>
            sCma<|> sKword <|> sTp <|> sNum <|> sBool <|>
            sNme <|> sStr)
    sSpace
    return ts
```

This scanner repeatedly tries using the scanner associated with each data constructor of **Token**, and finishes once none of the component scanners can extract any more tokens from the string. This is done by creating a combinator by combining each of the component scanners using the $< | >$ operator. This operator essentially works by trying to use the first scanner, and if **Nothing** is returned, trying the other scanner. Each of the component scanners work by looking at the front characters of the string to see if they correspond to a specific sequence of characters. For example, the scanner for boolean values looks for the sequences "true" and "false".

```
sBool :: Scanner Token
sBool =
  do
    sSpace
    b <- sString "true"
    return (Bool True)
  <|>
  do
    sSpace
    b <- sString "false"
    return (Bool False)
```

Two of the scanners work in a slightly different way than the rest: **sKword** and **sOp**. These scanners work by taking a list of strings and their corresponding token representation and turning it into a combinator using the **sStrings** function. Using **sStrings** prevents the need for writing out a separate case in do-notation for each keyword or operator. This is extremely useful for these cases as there are a large number of keywords and operators to handle.

```
sStrings :: [String] -> Scanner String
sStrings (x:[]) = sString x
sStrings (x:xs) = sString x <|> sStrings xs

sKword :: Scanner Token
sKword =
  do
    sSpace
    kw <- sStrings (map fst keywordMap)
    let tok = case (lookup kw keywordMap) of
                (Just t) -> t
                Nothing -> error ...
    return tok
```

```
keywordMap :: [(String, Token)]
keywordMap =
  [ ("if", Kword If)
  , ("then", Kword Then)
  , ("elseif", Kword ElseIf)
  , ("else", Kword Else)
  , ("end", Kword End)
  ...
  ]
```

After using the scanner function to scan as many tokens as possible, the remaining string is checked to ensure it is empty. If the program is well formed and syntactically correct then there should be nothing left in this string as all characters should be part of a token that's scanned. Therefore, if there is any string left my compiler raises an error stating that the syntax of the program is invalid. This check is performed by the **scanSrc** function, which packages scanning into a simple function that takes a program string as input and provides a list of tokens as an output.

```
scanSrc :: String -> [Token]
scanSrc s = case (scan scanner s) of
              (Just (ts, l)) -> if l == "" then ts
                                else error "Invalid␣syntax"
              Nothing -> error "Unable␣to␣scan␣program"
```

3.3 Parser

I begun constructing my parser by first defining an AST type to represent the structure of the parsed program. With a couple of exceptions, the constructors of the AST type correspond to the productions involving terminal symbols in the CFG for the language. For constructs such as for-loops this is quite easy to see, but becomes more difficult for expressions involving operators, given the many non-terminal symbols needed to impart the correct operator precedence and associativity.

```
data AST = Register Int
         | Number Double
         | Name String
         | String String
         | Boolean Bool
         | Nil
         | Type Type
         | Statements [AST]
         | Assignment AST AST AST
         ...
         deriving (Eq, Show)
```

I then defined a parser type that takes a list of tokens and produces some output alongside the tokens left over. This parser type was then defined as an instance of the monad and alternative type classes. Afterwards, this type was used to define some helper parsers, such as for getting the token at the head of the list.

```
newtype Parser a = P([Token] -> Maybe (a, [Token]))

pFront :: Parser Token
pFront = P(\ts -> case ts of
                    []     -> Nothing
                    (t:s) -> Just (t, s))
```

This parser type and the helper parsers closely resemble the scanners previously mentioned, as they're constructed in the exact same fashion, with the major difference being changing the input value from strings to lists of tokens. It would've been possible for me to skip out scanning to tokens first, going straight from a string to an AST. Despite this, I decided that using tokens as an intermediate step made this transition much easier to understand.

Using this parser type and the helper parsers, I constructed a parser for each terminal and non-terminal in the CFG. Some of these parsers were extremely simple to construct such as the parser for the **elf** non-terminal. By comparing the code for this parser with the CFG in figure 2.1, you can see that the parser closely resembles the **elf** non-terminal's production.

```
pElf :: Parser AST
pElf =
  do
    pToken (Kword ElseIf)
    e <- pExp
    pToken (Kword Then)
    b <- pBlock
    return (IfThen e b)
```

On the other hand, some of the parsers where much harder to construct, with the by far the hardest parser to construct being the parser for the **stat** non-terminal. You might expect that this parser was the hardest to construct due to large number of productions. This wasn't the case though as the use of parser combinators made the process of adding a list of productions a painless process. Instead, the main difficulty was parsing the production for conditional statements. This production was hard to parse due to the number of possible combinations: if-then, if-else, if-elseif-else, and if-elseif. These combinations are a result of the production using both the option and repetition control forms, which indicate optional and repeated non-terminals. I made many small adjustments through trial and error to this production, alongside the **elf** and **els** non-terminals, to see what worked best. Eventually I ended up with the production that can be seen in figure 2.1, alongside the following parser for the production:

```
do
  pToken (Kword If)
  cond <- pExp
  pToken (Kword Then)
  b <- pBlock
  elfs <- many pElf
  case elfs of
    [] -> (do -- if else
              els <- pEls
              pToken (Kword End)
              return (IfElse cond b els)
            <|>
            do -- if
              pToken (Kword End)
              return (IfThen cond b))
    xs -> do
            (do -- if elseif else
               els <- pEls
               pToken (Kword End)
               let xs' = foldr (\(IfThen c b) ->
                                  \x -> IfElse c b x) els xs
               return (IfElse cond b xs')
             <|>
             do -- if elseif
               pToken (Kword End)
               let xs' = case xs of
                           [x] -> x
                           _ -> foldr (\(IfThen c b) ->
                                         \x -> IfElse c b x)
                                  (head xs)
                                  (tail xs)
               return (IfElse cond b xs'))
```

While this code likely isn't the most elegant solution, I believe it isn't too verbose of an implementation. The **foldr** function in particular came in very handy for dealing with a chain of **elseif** conditions.

Similarly to the scanner, my code also checks to ensure that there are no tokens left at the end of parsing, raising an invalid syntax error in the case that there are any tokens left. This check is performed by the **parseTokens** function, which takes a list of Tokens and produces the corresponding AST.

```
parseTokens :: [Token] -> AST
parseTokens ts = case (parse pBlock ts) of
                   (Just (ast, l)) ->
                     if l == [] then ast
                     else error "Invalid␣syntax"
                   Nothing -> error "Unable␣to␣parse␣tokens"
```

When working on both the scanner and the parser, I initially constructed them without using a monadic style, instead using nested case statements. This allowed me to quickly construct a working scanner and parser, without the need for spending time defining the **Scanner** and **Parser** types as instances of the monad and applicative type classes. While these implementations served their purpose of allowing me to get things up and running quickly, they were not at all scalable. This was due to each extra token being parsed requiring an extra nested case statement. This meant that this approach would be extremely prohibitive when adding features such as for-loops which require parsing a longer list of tokens. Therefore, I switched to using a monadic style that allowed me to easily construct scanners and parsers using do notation.

3.4 Type Inference

While the implementation of the scanner and parser are extremely similar, implementing type inference was a completely different task. I had previously been referring to "Programming in Haskell" to guide my implementation, but for type inference I had to seek out a difference reference material. As previously mentioned, I eventually ended up using Lerner's lecture notes [16]. One difficulty faced when referring to these lecture notes was that all examples where written in the ML programming language. Therefore, I had to spend some time converting the provided function types into Haskell types. These types gave me a good idea of what each function was meant to do, in the case that there was no implementation given alongside the type. My implementation generally follows the notes, but adjustments were made to better suit my needs. These adjustments included using my own AST type as the input type, and introducing a separate array type. The most important step in implementing type inference was defining the structure of the types that are inferred, in the form of the Haskell data types **Type** and **TypeConstant**.

```
data Type = TyVar String
          | TyCon TypeConstant
          | TyArr [Type] Type
          | TyArray Type Int
          | TyBlank
          | Typeless
          deriving Eq

data TypeConstant = TNil
                  | TBoolean
                  | TNumber
                  | TString
                  deriving Eq
```

The **Type** type encapsulates all the different possible types returned by the type inference. The **TyVar** and **TyCon** data constructors are fairly self explanatory, representing type constants and type variables respectively. The **TyArr** constructor represents an arrow type, used for functions, with the list of types representing the types of the arguments, and the other singular type representing the return type.

The **TyArray** constructor represents an array that contains elements of the specified type, with the integer value indicating the number of elements. Initially a **TypeConstant** value was used instead of a **Type** value. This had to be changed in order to account for situations where we know a variable has an array type, but don't know the exact type of its elements. Therefore, changing it to a **Type** value allows a fresh type variable to be used to indicate this lack of knowledge. Additionally, in this case the length of the array is also unknown, so it is is set to the negation of the element type's variable number. This serves as a placeholder until the actual length is known.

Lastly, the **TyBlank** and **Typeless** may seem confusing additions but they serve important purposes. The **TyBlank** type is used as a placeholder in AST constructs where a type can optionally be specified, but isn't specified by the user, such as when declaring a variable. The **Typless** type is returned when a specific AST construct doesn't have a type, which is the case for assignment and conditional statements.

The two most important functions used for type inference are **inferAST** and **unify**. The **inferAST** function takes an **AST** value and returns a **Type** value that represents the type of the AST, and has the type **inferAST :: AST** $->$ **TST Type**. Here the **Type** value is wrapped in the state transformer monad **TST**. Maintaining a state is needed to keep track of the type environment, type scheme environment, and the next fresh type variable to use.

In the case of **AST** values that represent primitive values such as numbers, strings, booleans, and nil, this function simply returns the corresponding type constant.

```
inferAST :: AST -> TST Type
inferAST (Number n) = return (TyCon TNumber)
inferAST (String s) = return (TyCon TString)
inferAST (Boolean b) = return (TyCon TBoolean)
inferAST Nil = return (TyCon TNil)
```

For high-level constructs such as a conditional statement, the types of the component ASTs are inferred recursively.

```
inferAST (IfElse c t e) =
  do
    cT <- inferAST c
    unify cT (TyCon TBoolean)
    inferAST t
    inferAST e
    return Typeless
```

This example also demonstrates the **unify** function being used to ensure that the type of the condition is a boolean, with **unify** throwing a unification error if this isn't the case. This function is explained in more detail later in this section. Something else which you might notice as being slightly odd is the need for inferring the types of the then and else bodies represented by **t** and **e** respectively, despite not storing the types returned and passing them to **unify**. This is because these recursive calls to **inferAST** may call

the **unify** function themselves. This in turn updates the type environment stored in the state in order to store or update the types of any variables or functions encountered. For example, consider the case where **t** includes a variable assignment.

```
inferAST (Assignment (Name var) (Type t) val) =
  do
    newT <- freshVar
    mVarT <- getMType var
    let varT = case mVarT of
                 (Just t') ->
                   if (t' == t) || (t == TyBlank)
                   then t'
                   else error (var ++ "␣has␣type␣" ++
                               (show t') ++ "␣not␣type␣" ++
                               (show t))
                 Nothing -> if t == TyBlank then newT else t
    addEnv var varT
    valT <- inferAST val
    unify varT valT
    return Typeless
```

In this case **addEnv var varT** is used to add **var**'s type **varT** to the type environment. Another interesting case of the **inferAST** function is the case for binary operators.

```
inferAST (BinOp op l r) =
  do
    opSch <- getOpSch (show op)
    opT <- instantiate opSch
    lT <- inferAST l
    rT <- inferAST r
    retT <- freshVar
    let newArrT = (TyArr [lT, rT] retT)
    subst <- unify newArrT opT
    return (applySubstType subst retT)
```

In this case, **getOpSch (show op)** is used to get the type scheme for the operator **op**, and **instantiate opSch** is used to replace the unbound type variables with fresh type variables. **inferAST** is then called recursively to get the types of the operands, which are then combined with a fresh type variable to get a **TyArr** value for this expression which can be unified with the instantiated operator type. Unlike the previous example, the return value of **unify** is stored, with this value being any substitutions that result from this unification. A substitution is simply a mapping from a type variable to a different type.

```
type Subst = [(String, Type)]
```

A list is used here to easily handle situations where there are more than one substitution.

These substitutions are then applied to the return type so that in the case that the return type is updated during unification, the up to date type is returned by this function.

The most complex case of **inferAST** is for inferring the type of a function, and shows how this process isn't always trivial.

```
inferAST (Function (Name n) args b) =
  do
    mNT <- getMType n
    case mNT of
      (Just _) -> error (n ++ "␣already␣declared")
      Nothing ->
        do
          argTs <- inferASTs args
          retT <- freshVar
          addEnv n (TyArr argTs retT)
          inferAST b
          retTs <- getRetTs b
          let retTsTl = if (length retTs) <= 1 then []
                        else (tail retTs)
          subst <- unifyLists retTs retTsTl
          let retT' = case retTs of
                        [] -> (TyCon TNil)
                        _  ->
                          applySubstType subst (head retTs)
          argTs' <- inferASTs args
          addEnv n (TyArr argTs' retT')
          return (TyArr argTs' retT')
```

First **getMType** is used to check if the name of this function has already been used for a variable or a different function. If it hasn't, then the types of the arguments are inferred using **inferASTs** which simply performs **inferAST** multiple times, returning a list of the types. These types are combined with a fresh type variable to create a new function type which is then added to the environment. The reason the function is added to the environment here is to handle recursive functions, ensuring that a recursive call to **inferAST** for a function call statement doesn't raise an error when trying to find the type of the function called. After this, **getRetTs b** is used to find the types of each expression returned using a **return** statement, with all of these types being unified together with **unifyLists** to ensure that all the types are the same. In the case that there are no return statements, **retT'** is set to the nil type constant, but if there are any return statements, **retT'** is set to the return type found when unifying the return types. **inferASTs args** is called again in order to easily get the updated types of all the arguments after inferring the types in the function body. These argument types and the return type are used to add the function to the type environment again. This overrides the previous type of the function in the type environment in order to ensure the type has the correct return type rather than a type variable. Finally this new function type is returned.

As previously mentioned, the other key component of type inference is the **unify** function.

This function takes two types and returns any resulting substitutions made by replacing a type variable with another type variable or type constant. The return value is embedded in the state transformer monad in order to be able to apply any substitution performed to the type environment.

My implementation of the Hindley-Milner algorithm is based off the outline provided by Lerner's lecture notes, and can be summarised as:

- If either type contains a type variable that doesn't occur in the other type then return a substitution that maps the type variable to the other type.
- If both types are a type constant and the type constants are the same then return no substitution.
- If both types are an arrow type with the same number of arguments, then return the substitutions given by unifying the corresponding argument and return types.
- If both types are an array type with the same length then unify the types of the elements.
- If both types are an array type and either type uses a type variable, then unify the types of the elements and update uses of the placeholder length value.
- Otherwise return a unification error.

By repeatedly applying this algorithm to pairs of types, the number of type variables in the type environment is whittled down as they're replaced with type constants. Compared to **inferAST**, the **unify** function was relatively easy to implement, as it closely resembles the summary of the algorithm above, as can be seen in the cases for type variables.

```
unify :: Type -> Type -> TST Subst
unify (TyVar v) t =
  if occurs v t && (TyVar v) /= t
  then unificationError v (show t)
  else
    do
      let subst = [(v, t)]
      applySubstTypeEnvt subst
      return subst
unify t (TyVar v) =
  if occurs v t && (TyVar v) /= t
  then unificationError v (show t)
  else
    do
      let subst = [(v, t)]
      applySubstTypeEnvt subst
      return subst
```

The implementation of **unify** did require some tweaking though, especially to account for the tweaks made to the **TyArray** constructor that I mentioned previously.

```
unify x@(TyArray t1 i1) y@(TyArray t2 i2) =
  if i1 == i2 || i1 < 0 || i2 < 0
  then
    if i1 == i2
    then unify t1 t2
    else
      do
        let var = minimum [i1, i2]
        let len = maximum [i1, i2]
        setArrayLength var len
        unify t1 t2
  else unificationError (show x) (show y)
unify x y = unificationError (show x) (show y)
```

Both of the array type cases in the summary are combined into one case of the **unify** function, with a nested conditional being used to handle the different cases. **minimum [i1, i2]** is used to get the negative placeholder length, which is then stored in **var**. **maximum [i1, i2]** is used to get the actual length in the case that one of the array types has a known element type and a positive length. If both array types use placeholder lengths, then the value stored in **len** doesn't matter as it will still be negative, meaning it will be replaced once it's unified with an actual length.

Bringing this all together is the **inferTypes** function that starts the chain of recursive **inferAST** calls. This call is initialised with an empty type environment, a type scheme environment that includes all the operators, and an initial type variable of 0.

```
inferTypes :: AST -> TypeEnvt
inferTypes ast = envt
                 where
                   (_, (envt, _, _)) =
                     appTST (inferAST ast) (typeEnv, schEnv, 0)
```

3.5 Code Generator

The last major component of my compiler is the code generator. Similarly to type inference, the code generator is extremely recursive in nature. The code generator recursively generates the code for AST sub-trees, correctly arranging the lines of code produced. It can be split into three separate parts: generating code for variables, generating code for strings and functions, and generating code for the rest of the program. These three parts of the code are inserted at specific points in a template LLVM IR file that acts as a wrapper for the code generated. This template file includes declarations for the C standard library functions used, constants used for printing, alongside functions for performing floor division and printing booleans. After inserting the code into the template, the code

is written to an LLVM IR file that can easily be compiled into an executable.

The Haskell function that performs code generation is **generateCode**, which takes the program AST and the type environment obtained through type inference, and returns three lists of strings for the lines of LLVM IR code generated for the three different parts.

```
generateCode :: AST -> TypeEnvt ->
                    ([String], [String], [String])
generateCode (Statements xs) envt =
  let
    vars = generateVars (Map.toList envt)
    (c, s) = (appGST (generate xs) (GenState { reg = 0
                                             , lbl = 1
                                             , env = envt
                                             , code = []}))
  in
    (vars, checkBreak (code s), checkBreak c)
```

The code generation for variables performed by **generateVars** involves assigning each variable in the program to a global variable. This function goes through the type environment obtained from type inference and finds each variable. For each variable it then generates an LLVM IR instruction for assigning a default value to that variable, returning the list of instructions generated. For example a variable **x** with type **number** would generate the LLVM IR instruction **@x = global double 0.0**.

The code generation for strings and variables and the main body of the program is performed by the **generate** function. This function uses the state transformer monad to store the next register number, next label number, the type environment, and the LLVM IR code generated for strings and functions. Strings need to be handled separately as they're stored in a global variable which is used whenever the string is referenced. This global variable needs to be placed higher up in the template file so that it can be accessed by functions alongside the main function. These variables have the same name as the string they store, and have the prefix '-' in order to differentiate them from LLVM IR instructions in the case the string has the same value. While a Lua program doesn't have a main function as an entry point for a program, an LLVM IR program uses **@main** as the entry point for the program. Functions need to be handled separately as LLVM IR doesn't allow nesting functions, meaning function declarations can't be placed in the main function of the LLVM IR program. Instead they're placed higher up in the template file, so they can be accessed in the main function that's further down in the template. Lastly, a record type is used to encapsulate this state, in order to simplify access and manipulation.

Applying **generate xs** to the initial state returns the main body code, alongside the end state from which the code for strings and functions can be extracted. When returning the code for strings and functions and the code for the main body of the program, the **checkBreak** function is applied to the two lists. This function ensures that there are no placeholders for **break** statements left, as this would indicate that a **break** statement has been used outside a loop, meaning an error should be raised.

The **generate** function takes a list of ASTs as it takes a list of ASTs embedded in the **Statements** AST constructor. Therefore the auxiliary function **generate'** is used to generate the LLVM IR code for a single AST. The **generate'** function takes an **AST** and returns two values wrapped in the state transformer monad. The first value is a **String** that contains the register the result value of the code generation is stored in. For example, in the case of generating the LLVM IR code **%r0 = fadd double 1.0, 2.0** for adding 1 and 2 together, this value would be **"%r0"** as this register stores the result of the operation. In the case that the code generated doesn't have a particular result value, such as the code for a conditional statement, this value is just an empty string. The second value is a list of strings of the LLVM IR code produced, in the correct order to perform the operation corresponding to the **AST**.

Another key similarity between code generation and type inference is how the code generation for primitive values is extremely simple. All these cases simply return the correct result value for the value in the **AST**. Generating code for a string also involves adding the string to the strings and functions code in the state.

```
generate' (Number i) = return (show i, [])
generate' (String s) = addString s >>
                       return (getStringRegister s, [])
generate' (Boolean b) = if b
                        then return ("1", [])
                        else return ("0", [])
generate' (Nil) = return ("null", [])
```

One particular construct that was particularly challenging to generate LLVM IR code for was for-loops. The key pain point of this was the fact that for Lua for-loops the sign of the step value determines the comparison made between the counter variable and the condition value. For example the for-loop **for i = 0, 6, 2** performs the comparison i <= 6 as the step value **2** is positive. The for-loop **for i = 6, 0, -2** performs the comparison i >= 0 as the step value **-2** is negative.

Therefore, when generating code for for-loops, I first had to generate code that assigns an intermediate value to 1 or -1 depending on if the step value is greater than or less than 0. As a side note, this point also demonstrates a drawback of compiling ahead-of-time. When interpreting a for-loop, the step value can be checked to ensure it's not 0. Unfortunately, this check can't be performed ahead-of-time as the step value can be a variable for which we have no idea of the value it will have when the code executes. Going back to code generation, this intermediate value is then multiplied by the counter variable and the condition value before performing the comparison. This is because in the case that the step value is positive, a multiplication by 1 has no affect on either value, resulting in the comparison i <= 6 using the previous example. In the case of a negative step value, both sides will be multiplied by -1 giving -i <= -0, which is mathematically equivalent to i >= 0, which is the desired comparison to be performed when the step value is negative.

Apart from generating the code for performing the comparison between the counter and condition values, the rest of the code generation is fairly typical of the code generation for other **AST**s. The code for the body of the loop is generated, alongside getting fresh

labels from the state in order to generate the branch statements needed to correctly wire up all the parts of the loop.

```
generate' (ForLoop i@(Assignment v _ _) c s b) =
  do
    m <- freshReg
    let mi = read (drop 2 m) :: Int
    (_, is) <- generate' i
    (_, ms) <-
      generate' (IfElse
                   (BinOp LessThan s (Number 0))
                   (Assignment (Register mi)
                      (Type (TyCon TNumber)) (Number (-1)))
                   (Assignment (Register mi)
                      (Type (TyCon TNumber)) (Number 1)))
    (c', cs) <-
      generate' (BinOp LessEqual
                   (BinOp Multiplication (Register mi) v)
                   (BinOp Multiplication (Register mi) c))
    (_, ss) <-
      generate' (Assignment v (Type (TyCon TNumber))
                 (BinOp Addition v s))
    (_, bs) <- generate' b
    (l0, r0) <- freshLbl
    (l1, r1) <- freshLbl
    (l2, r2) <- freshLbl
    let
      m' = m ++ "␣=␣alloca␣double"
      b = branchPrint r0 :: String
      cb = condBranchPrint c' r1 r2
      ls = ([m'] ++ is ++ ms ++ [b] ++ [l0] ++ cs ++ [cb] ++
            [l1] ++ bs ++ ss ++ [b] ++ [l2])
      brk = branchPrint r2 :: String
    return ("", setBreak brk ls)
```

Another construct that was more difficult to generate code for was functions. Firstly, the type environment has to be checked to ensure this function doesn't include any type variables. A type variable in the function's type indicates that the function isn't used, meaning we don't need to generate code for the function. Additionally, the parameters used by functions in Lua are local variables. Therefore, during parsing the prefix '-' is added to these variables to differentiate them from global variables with the same name. Additionally, the values of these parameters have to be copied into freshly allocated memory to provide mutability. Pointers to these regions of memory are stored in local registers that begin with an additional prefix "-r", to differentiate them from the global variables and parameters. Before generating code for the body of the function all references to parameters have to be renamed to refer to these new registers instead of the original parameters. Lastly, if the return type is **nil** then **return null** is added to the end of the function. This is because a function is given the return type **nil** if no explicit return value

is given.

In the **generate'** cases for each AST construct, a series of functions ending in "Print" are used for producing the actual lines of LLVM IR code. For example, when generating the LLVM IR code for a binary operator, the **binOpPrint** function is used. These printing functions all return a **PrintfType** type value, with **binOpPrint** being no exception to this. These **PrintfType** values are constructed using the **printf** function, which is similar to the **printf** function in the C programming language. **printf** takes a string with a series of string placeholders indicated by **%s**, and gives a function that when applied to string arguments populates the string placeholders one by one with the strings provided. By providing **printf** with a string for a line of LLVM IR code, using **%s** to indicate where registers or types should be placed, these printing functions can be used to easily generate the LLVM IR code needed. The code generation for a binary operator clearly demonstrates this.

```
generate' (BinOp op x y) =
  do
    (x', xs) <- generate' x
    (y', ys) <- generate' y
    r <- freshReg
    t <- inferTypeConstant x
    let
      o = case op of
            Equal -> equalPrint t x' y' :: String
            NotEqual -> notEqualPrint t x' y' :: String
            _ -> binOpPrint op x' y' :: String
      a = assignPrint r o :: String
    return (r, xs ++ ys ++ [a])
```

The case statement here is to ensure that printing for equal and not equal operators is handled separately. The polymorphic nature of these operators makes their code generation more difficult, as the LLVM IR line template depends on the type of the operands. That aside, this function simply generates the code for the operands **x** and **y**, and uses **binOpPrint** to get the printing function for the given operator **op**. The placeholders in this function's string are then replaced with the registers **x'** and **y'**. The type annotation **:: String** has to be added after using a printing function to specify the **PrintfValue** should now be treated as a **String**. **assignPrint** is another printing function, and simply assigns the register in the first argument to the instruction in the second argument.

One issue I initially had when generating code was ensuring that all numerical labels where in strict numerical order, as required by LLVM IR for numerical labels. Despite ensuring this, I ended up having a situation when compiling the LLVM IR would raise a labelling error. In the end I decided to prefix all labels with a '-' character. Not only did this solve this compilation issue, it also meant that the labels didn't have to be in numerical order as they were now non-numerical as they now began with a '-' character.

You may suspect that all these prefixes may collide, but I paid special attention to ensure that they didn't overlap. The prefix "@-" indicates a string, while the prefix "%-" indicates

a label. A local identifier of the form **%-x-y** indicates the parameter **y** of function **x**. Lastly a local identifier of the form **%-r-x-y** indicates allocated memory in function **x** for parameter **y**.

3.6 Creating Executables

As my compiler is simply a front-end compiler that produces an LLVM IR file, an existing compiler must be used to compile this into an executable that can be run. While llc [18] would be a good choice for this, I instead opted to utilise Clang [17]. Despite being a C compiler, Clang also supports compiling LLVM IR, as it uses LLVM for the back-end of the compiler. The main reason for this choice was to be able to easily link the LLVM IR file with the C standard library. This is needed as my compiler produces LLVM IR that utilises functions from the C standard library. These functions are used to perform functionality that would be extremely cumbersome to implement myself, such as printing to the console and performing exponentiation with floating point values.

Overall, all the different parts of the compiler I've previously mentioned are all linked together in the **main** function in the **Main.hs** file. This function acts as the entry point when running the compiler, once it's compiled using GHC. The first thing this function does is create a string from the Lua file given as a command line argument. This string is than scanned into a list of tokens, which is then parsed into an AST. The AST is then used to infer the types of each variable and function, with the resulting type environment being given alongside the AST to the code generator. Lastly, the code generated is inserted into the lines of a template file, resulting in a LLVM IR file being produced. Therefore, when running the compiler it is important to have this template file in the same directory so that it can be read in by the compiler. Lastly Clang is called to compile the LLVM IR file into an executable file with the same name as the Lua and LLVM IR files. The executable produced is a ".out" file as I developed this compiler on a Linux machine, so designed the compiler to output executables that could be run on a Linux machine. This is also evident in the compilation target listed in the LLVM IR template file. Therefore, my compiler is known to work on Linux, and would definitely require adjustments to work on other platforms.

4 Evaluation

4.1 Benchmarking

In order to observe the performance of my implementation in relation to existing Lua implementations, I decided that some benchmarking needed to be done. The first decision to make was deciding on the implementations to compare against. I decided to compare against the standard interpreted implementation of Lua, LuaAOT, and LuaJIT. I chose these three implementations as it allowed me to compare my implementation to implementations that are interpreted, AOT compiled, and JIT compiled.

The next decision was to choose the programs to use to benchmark the performance of these implementations. As I previously alluded to, I decided to pick three different algorithms to test. The three algorithms I ended up using were computing numbers in the Fibonacci sequence, a solution to the rod cutting problem, and the Floyd-Warshall algorithm. In addition to deciding the algorithms to use, I had to decide on the input to give to these algorithms. In general, I picked an input that ensured a long enough run time to differentiate between the time it takes for the different implementations to run the program. Longer run times ensured the implementations had more time to diverge from each other, making the relationship between their run times more clear. Additionally, I used inputs that gave outputs that could easily be checked to ensure the program is working correctly.

To time how long a program takes to run with each implementation I used the GNU **time** command, which times how long a process takes to run in milliseconds [6]. For my implementation and LuaAOT I timed both how long the program took to compile, alongside how long it took to run. This allows the compile time and run time to be combined, enabling a more detailed comparison of the implementations. When collecting results, I timed the compilation or execution time five times in order to be able to take the mean average, to get a more accurate value. Each of these timings was performed using a 4GHz AMD Ryzen 5 5500U CPU alongside 14GB RAM. All of the data I collected can be viewed in the appendix.

4.1.1 Fibonacci Sequence

The Fibonacci sequence is probably one of the most famous integer sequences in the world, and can be described with the recurrence relation:

$$\begin{aligned} &F_0 = 0 \\ &F_1 = 1 \\ &F_n = F_{(n-1)} + F_{(n-2)} \end{aligned}$$

Using the recurrence relation above, it's relatively easy to construct a program that re-

cursively computes the nth number in the sequence. I decided that computing the 35th number in the sequence was the right balance of letting run times diverge without letting them be too long. I used the following Lua function to perform the computation:

```
function fib(n)
    if n == 0 then
        return 0
    elseif n == 1 then
        return 1
    else
        return fib(n-1) + fib(n-2)
    end

    return -1
end
```

As you can see, the algorithm is extremely recursive, with seeing how my compiler handles recursion being the main reason I selected this algorithm.

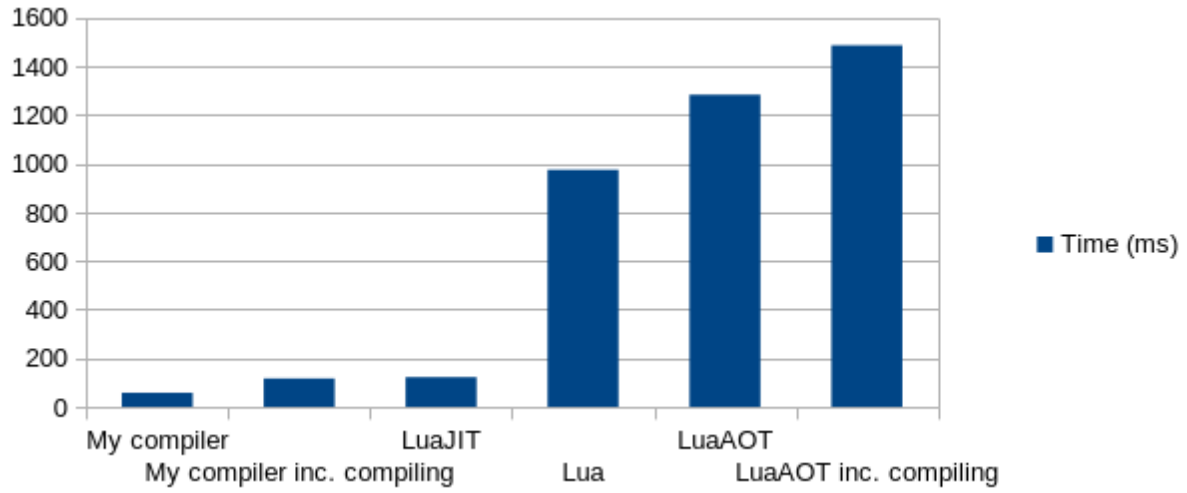

Figure 4.1: Time to execute the Fibonacci program

The results shown in figure 4.1 show my implementation performed the best, even when factoring in the compile time. Despite this LuaJIT was only a few milliseconds behind, with Lua and LuaAOT trailing behind significantly. It makes sense that my compiler and LuaJIT are close to each other as they both use machine code to perform operations. Moreover, my compiler and LuaJIT are approximately ten times faster than the standard Lua implementation, which shows how much this impacts performance. Lua and LuaAOT are close together as LuaAOT is based off of a Lua interpreter, meaning that both Lua and LuaAOT will likely be performing extremely similar instructions to each other. Additionally, this figure also shows that my compiler handles recursion well, as it performed well in this scenario.

4.1.2 Rod-Cutting Problem

The rod cutting problem involves figuring out the maximum amount of money that can be made by cutting a steel rod of length n into rods of shorter lengths. The input is the length n of the rod and a list of prices for each length of rod from 1 to n. I used a naive recursive solution to ensure a longer run time that could be more easily timed, using the **CUT-ROD** function from the textbook "Introduction to Algorithms" [5]. A minor difference of my implementation compared to the pseudocode is the addition of an additional argument **i**. This was done to ensure that the **cutRod** function uses a local version of **i**, as it's a function parameter. If this isn't done, then issues will occur when calling **cutRod** recursively, as the value **i** will be overwritten by the recursive calls, causing incorrect looping. This issue could be solved by changing loop counter variables to be locally scoped. The general way this algorithm works is by looping over each possible size rod that can be cut off and adding it's value to the most money that can be made from the rod left over, taking the maximum value of the current maximum and this case.

```
function cutRod(p, n, i)
    if n == 0 then
        return 0
    end

    q = -99
    for i = 1, n do
        q = max(q, p[i] + cutRod(p, n-i, 0))
    end

    return q
end
```

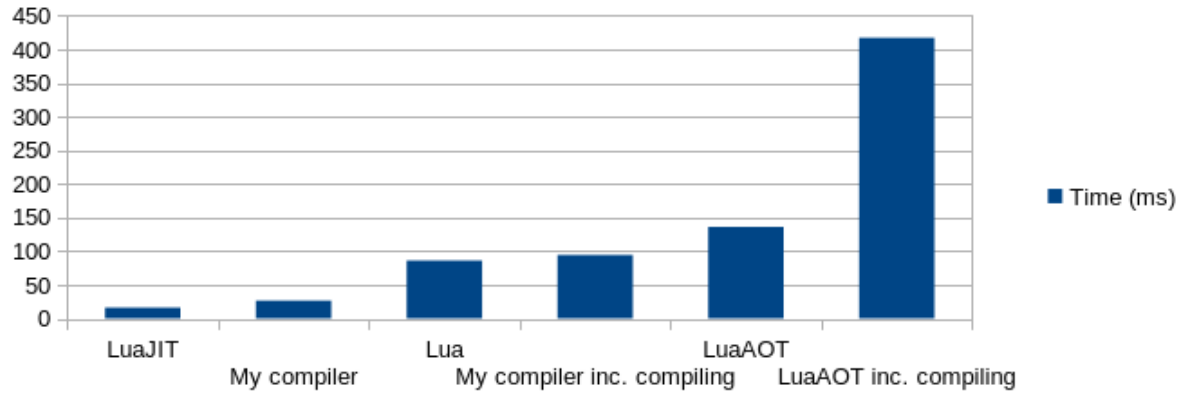

Figure 4.2: Time to execute the rod cutting program

The results shown in figure 4.2 show that for the rod cutting problem, LuaJIT has the best performance, with my implementation following in a fairly close 2nd when only considering the time to run. The performance gain is slightly less here, as LuaJIT and my compiler are only around four times faster. One possible explanation for this is my compiler handling recursive calls better than the standard implementation, with a lower

number of recursive calls being used by this program. When factoring in the compilation time of my implementation, it performs worse than the standard Lua interpreter, but still performs better than LuaAOT. This shows that when including compilation time, a faster time isn't guaranteed. A key trade off of AOT compilation is being able to run a program quickly once it's compiled, but with the possibility of long compilation times. LuaAOT is a key example of this trade off, with extremely long compile times resulting from the large number of files that have to be linked together at compile time. In most cases, a program tends to be compiled once and then run many times. This means that my compiler provides the most benefit when compiling infrequently, as opposed to a situation such as testing a program where you might want to recompile often.

4.1.3 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is another common algorithm, and is used for solving the all-pairs shortest paths problem. This algorithm compliments the previous two as it is an iterative algorithm not a recursive algorithm. I selected it to help indicate if any of the implementations perform better or worse when recursion isn't used. Like my implementation for the rod cutting problem, my implementation of the Floyd-Warshall algorithm is based upon a pseudocode example from "Introduction to Algorithms". The function that implements the algorithm simply loops over every node **k** in the graph and checks to see if any path between two nodes **i** and **j** would be shorter if it went through **k**. Before running this function a basic graph is initialised, and afterwards some printing of edge weights is done.

```
function floydWarshall()
    for k = 1, v do
        for i = 1, v do
            for j = 1, v do
                if dist[getInd(i, j)] >
                    dist[getInd(i, k)] + dist[getInd(k, j)]
                then
                    dist[getInd(i, j)] = dist[getInd(i, k)]
                                         + dist[getInd(k, j)]
                end
            end
        end
    end
end
```

While this function works with my implementation alongside existing implementations, a minor adjustment had to be made to the declaration of the array used to store the graph. While traditional implementations of Lua allow the statement **dist = {}**, my implementation requires specifying the type of the array if an empty array initialisation is used. This meant that for the program to work with my compiler, this line had to be changed to **dist:number[16384] = {}**.

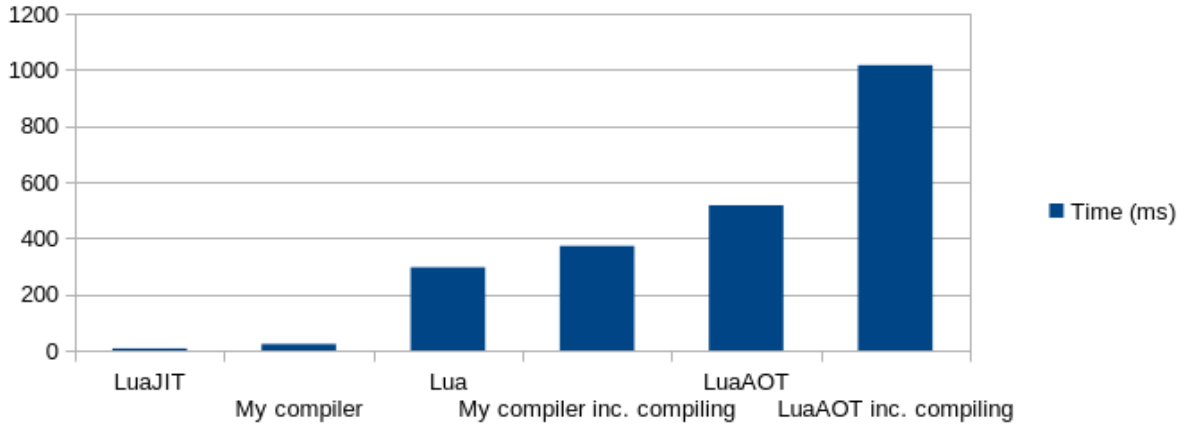

Figure 4.3: Time to execute the Floyd-Warshall algorithm program

By looking at figure 4.3, we can see that the ordering of the times is identical to the ordering for the rod cutting program. Additionally, there is a larger gap between LuaJIT and my compiler in this case, with my compiler being three times slower. Despite this, my compiler is still approximately ten times faster than the standard implementation of Lua. Another interesting observation is that LuaAOT is almost twice as slow as the standard implementation of Lua. You might expect this to be due to the longer run times of these implementations giving more time for their times to diverge. This isn't the case though as the run times shown in figure 4.1 are longer but are closer together. Therefore, it's likely this is due to a particular feature of the program, such as using loops over recursion, with LuaAOT preferring recursive programs over iterative ones. This could also explain why my implementation is three times slower than LuaJIT, and would suggest that my compiler doesn't handle iterative approaches as well as recursive ones.

4.1.4 Overall Result

Overall, the benchmarking shows that my compiler is close to the performance of LuaJIT, and significantly faster than Lua, when only considering the time to run the executable. When factoring in compilation times, my compiler's performance is more similar to the standard implementation of Lua. This suggests that my compiler's compilation times are in line with the runtime overhead of the standard Lua interpreter. Be that as it may, this compilation overhead is only a concern in situations where programs are frequently compiled. Additionally, there is evidence to suggest that my compiler has better performance when a recursive approach is used over an iterative approach. On the whole, I believe it's fair to say that I achieved my goal of improving performance by using AOT compilation.

4.2 Testing Correctness

Throughout the development process I tested my code in order to ensure that my compiler was working correctly. At the beginning of the project, this was done by creating simple test programs relating to the feature I was currently implementing. These test programs were inputted into the compiler, with adjustments being made to the compiler to ensure

that a particular part of the compiler worked correctly. In general each of these programs focused on a single language feature such as loops or arrays, but there was some overlap to ensure that different features worked well together.

After the majority of the implementation was complete, I then turned my attention to using unit tests. These unit tests ensured that any minor adjustments I made to my compiler to fix a bug didn't result in another part of my compiler breaking. I constructed these unit tests using the HUnit framework for Haskell [10]. For each part of the compiler(scanner, parser, type inference, and code generator) I created a HUnit test. Each of these tests contained assertions based around a series of simple Lua programs that each focused on a different language feature such as conditionals or loops. These tests helped me to quickly narrow down the location of any error to a particular part of the compiler and the particular construct that was causing the error.

Lastly, the benchmarking I performed acted as a form of integration testing, as benchmarking my compiler involved linking all the parts of my compiler together. Furthermore, as the benchmarking programs were more complex than the programs used for unit testing, they demonstrated that the compiler was working correctly for more complex programs. These programs where more complex as different language features were being used together, such as using function calls to index arrays. I checked that the benchmarking programs were working correctly by ensuring the output was correct for the given input, such as checking the Fibonacci program outputted 9227465, as this is the 35th Fibonacci number.

4.3 Interoperability

My benchmarking showed that my compiler is mostly interoperable with existing Lua implementations when programs are written with my compiler in mind. It didn't reflect how well my compiler works with existing programs written by other people. The ability for my compiler to work with existing programs without any modifications is another indicator of its interoperability. To find some existing Lua programs I utilised Rosetta Code, which gives examples of the same program in many different programming languages [25]. I looked at the programs implemented in Lua, and the picked out some well known programs. For each of the programs I then identified the main areas of incompatibility with my compiler, with these listed in figure 4.4.

As figure 4.4. shows, all the programs identified wouldn't work with my compiler due to various incompatibilities. Even in the case of an existing implementation of the Fibonacci sequence, my compiler is incompatible due the language features used. This isn't surprising though as my compiler lacks many features supported by Lua. Therefore, there is a major need to add more language features to my compiler in order to achieve usable interoperability with existing programs. Except for the programs that used tables, the incompatibility of these programs wasn't due to any of the design decisions I made when constructing my compiler. Therefore many of these programs would work if minor adjustments were made to ensure that just the features supported my compiler are used.

One upside of this analysis is that it provides a clear indication of the extra language features that are the most important to add in order to improve interoperability. The

Program	Incompatibilities
Ascending primes	local keyword, length operator, tables
Base64 encode data	local keyword, bitwise operators, IO library
Catalan numbers	local keyword, metatables, string library
Dijkstra's algorithm	local keyword, tables, iterators, concatenation operator
Fibonacci sequence	math library
Jewels and stones	local keyword, length operator, string library
Knapsack problem	tables, local keyword, string library, iterators
Substitution cipher	local keyword, tables, length operator, math library, iterators
Towers of Hanoi	local keyword, IO library, length operator, goto

Figure 4.4: Incompatibilities of existing Lua programs with my compiler

key incompatibility that is found in all but one of the programs is not having support for the local keyword. Therefore implementing the local keyword is clearly the first step in improving interoperability. The lack of tables is also a major limiting factor to seamless interoperability. Most of the uses of tables could be refactored to use arrays, but this requires lots of extra effort, and doesn't allow seamless interoperability. Tables could be implemented using an existing hash map implementation or by constructing a new implementation from scratch in LLVM IR. This is a serious undertaking though, so I don't regret my choice of using arrays for this project.

4.4 Summary

Overall, my compiler doesn't quite match the performance of LuaJIT. Despite this, it still provides better performance than the standard implementation of Lua and LuaAOT when just considering the time for the compiled executable to run. This performance improvement is the main benefit of my implementation, with a longer overall program run time leading to my implementation having a significantly quicker execution time. On the other hand, in it's current state my implementation's main drawback is the lack of interoperability with existing programs. This drawback can often be offset by rewriting the program to just use the language features available. Moreover, if a program is written with my compiler in mind, it can easily be run using other implementations with minimal or no modifications needed.

5 Conclusion

5.1 Project Management

Throughout this project I used a Gantt chart to track my progress. By comparing figures 5.1 and 5.2 you can clearly see the progress I've made between my interim report and now, alongside the adjustments I made along the way. Possibly the most significant change is increasing the time I took away from this project over the Christmas holiday and exam season. In hindsight I naively underestimated the workload of my other modules over this period, meaning I had to compress my schedule for the rest of the project. Another major difference was allocating more time to add additional features such as functions and arrays. I extremely underestimated the complexity associated with implementing these features, needing an extra two weeks in addition to the week I allocated. Luckily this was offset by needing less time for running benchmarks and evaluating interoperability, being able to do these tasks while writing this report. Overall, while I definitely could've done a better job at planning my time out, I was still able to achieve the goal I set out for myself. If I was to do this project again I would've definitely ensured I left a couple weeks free to account for tasks taking longer than expected.

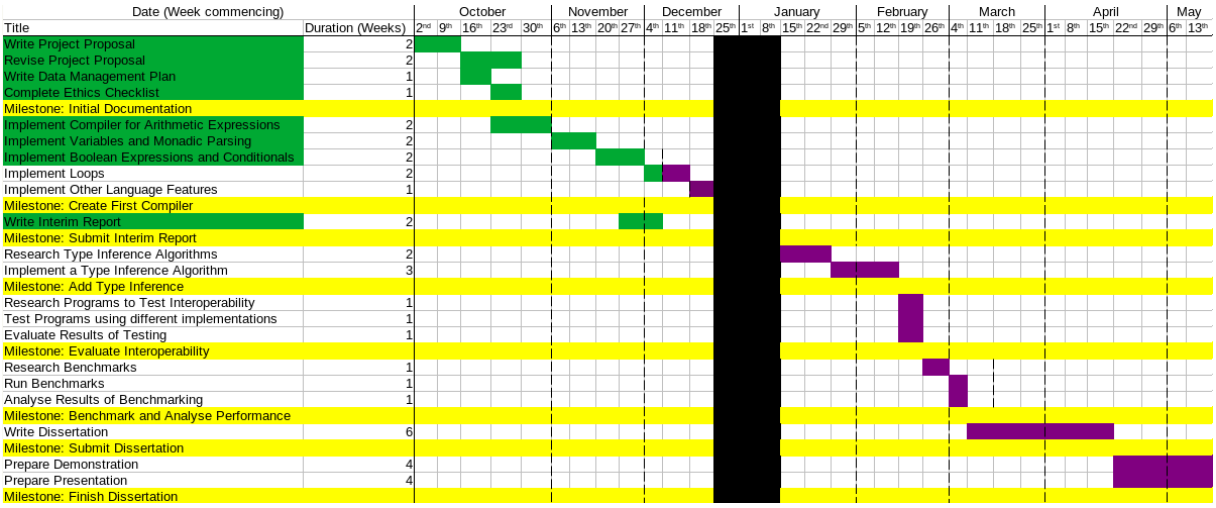

Figure 5.1: Interim report Gantt chart

While my Gantt chart helped me to manage the project on a long term basis, the key driver of my daily productivity was a simple to-do list that listed the immediate tasks I needed to perform. By looking at my Gantt chart I figured out what the next tasks I needed to complete were, and broke them up into smaller steps that I put on my to-do list. I also experimented with using GitLab issues to manage individual tasks, as it meant I could then link my Git commits to them. GitLab issues seemed like something worth experimenting with as I was using a GitLab repository to store all my code to ensure that it was always backed up. Another useful benefit of this was having the ability to go back and see previous versions of my code, which was handy on a number of occasions. Despite this, I found that GitLab issues were harder to use as I had to log on to GitLab every

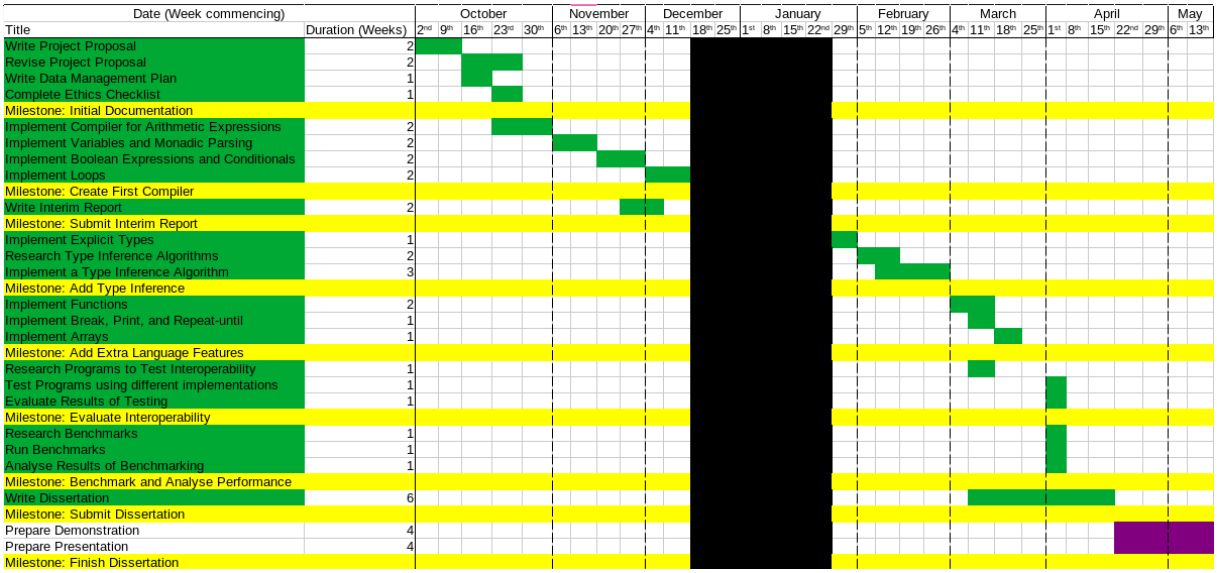

Figure 5.2: Current Gantt chart

time I wanted to make a change, compared to being able to quickly open my to-do list. As a result I stopped using GitLab issues, and reverted back to using a simple to-do list. If I was to do this project again I would definitely use this approach again, as it ensured that I was always on tasking, and doing something that was working towards the goal of my project. Be that as it may, I would still experiment with alternative approaches as I believe that there is always room for improvement when it comes to project management.

Every time I started a new task on my todo list, I created a simple markdown notes file. In this file I noted down any research I conducted, changes I made to my code, alongside any other observation I made or useful thought I had. These notes were an invaluable source of knowledge throughout the project, and ensured that I always had a record that I could look back on at any point. These notes were combined with a work log I kept, noting down the work I did every day, which provided a more detailed account of my progress than my Gantt chart alone. This log was particularly useful for recalling the progress I had made between supervisor meetings, which were held approximately every three weeks. This schedule worked well as it gave me enough time to make significant progress between meetings while still allowing me to receive regular feedback on the progress I was making.

5.2 Reflection

Overall I enjoyed this project as I was able to learn lots about constructing a compiler and implementing type inference, which are two areas I had no experience with coming into this project. There were many times where I struggled to implement something due to a lack of experience, but I was able to persevere and successfully fulfil all the aims and objectives I set out for myself. I was able to develop an AOT compiler for a statically typed variant of Lua. This compiler is generally interoperable with existing Lua implementations when programs just use the language features supported. When this is the case, my compiler has been shown to have faster execution times than other existing Lua implementations, when just considering the run time of the executable.

On a more general note, this project has shown that it is possible to produce an AOT compiler for a static variant of an originally dynamic language that provides improved performance. Additionally, it has shown that this can be done through building a new compiler from the ground up, without needing to use advanced techniques to convert an existing interpreter to a compiler. Furthermore, my benchmarking results for LuaAOT show that using these techniques can result in worse performance. Despite this, the lack of certain language features means that my compiler's practical use in its current state is limited, especially in relation to running existing programs. Significant improvement is needed for it to compete with LuaJIT and LuaAOT in a meaningful way. Overall, my proudest achievement is simply being able to write a program in Lua, and have it compile to an equivalent LLVM IR program.

That being said, there are definitely things I would do differently if I were to do this project again. Firstly, I would initially spend more time doing research before diving into the implementation, especially since I was new to compilers and type systems when starting this project. While this approach let me get basic things up and running quickly, it meant I had to go back to make changes at later points in the project in response to figuring new things out. Doing more initial research would've prevented this, alongside providing a better direction for my project. When deciding on the purpose and motivation behind this project, I figured a lot of it out as I went along, meaning more initial research would've provided a stronger foundation to base these decisions on. In addition, while I enjoyed generating LLVM IR code "by hand", I encountered lots of issues when trying to generate correct code. I would've had an easier time if constructed the LLVM IR code using a library like llvm-tf [32], with using a library being almost essential in implementing more sophisticated language features. Despite this I don't regret generating LLVM IR code this way as it allowed me to gain a much better deeper understanding of how the language works. While I had some background in Haskell coming into this project, I had never undertaken a project of this scale using Haskell. Therefore, while I was able to get a lot more comfortable working with Haskell over the course of the project, my inexperience has definitely left room for improving the codebase. I perhaps could've achieved more if I used a language I was more comfortable with, but then I wouldn't have been able to improve my Haskell skills.

In relation to LSEPI issues, the most relevant issue to my project is what I plan to do with my source code. I plan to make the source code open source, and available under the MIT license, allowing anyone to read, download, run, and freely expand upon my code. The key benefit of this is that it will allow people to draw inspiration from this project, perhaps applying the idea to a different language such as Python or Ruby. Additionally, it will allow people to independently verify the results of my testing. While the exact times will be different due to using different hardware, you would be able to see if the results are the same in relation to each other. A possible unintended consequence of making my code open source is that people might not realise the limitations of my compiler compared to other implementations. Therefore, I will need to ensure that I add good documentation to my repository that clearly and accurately describes what my compiler is able to do, alongside its specific limitations compared to other implementations.

In conclusion, I was able to achieve the aims and objectives I set out for myself, while expanding my knowledge of compilers and type systems.

Bibliography

[1] Abo, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, Boston, MA, USA, 2006.

[2] Bart, E. Nelua Overview. https://nelua.io/overview/, Retrieved 6th December, 2023.

[3] Burge, W. H. *Recursive Programming Techniques*. Addison-Wesley, Reading, MA, USA, 1975.

[4] Capretta, V., Kraus, N., and de Jong, T. Lecture 10: Extending our Language: Towards a Simple Imperative Language. COMP3012 Compilers, University of Nottingham, UK. Delivered on 14th November 2023.

[5] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd ed. MIT Press, Cambridge, Massachusetts, USA, 2009.

[6] Free Software Foundation. Gnu time. https://www.gnu.org/software/time, Retrieved 5th April, 2024.

[7] Gualandi, H. M. LuaAOT. https://github.com/hugomg/lua-aot-5.4, Retrieved 31st March, 2024.

[8] Gualandi, H. M., and Ierusalimschy, R. A Surprisingly Simple Lua Compiler. In *Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP '21)* (New York, NY, USA, 2021), ACM Inc., pp. 1–8.

[9] Haskell.org. Applications and libraries/Compilers and interpreters. https://wiki.haskell.org/Applications_and_libraries/Compilers_and_interpreters, Retrieved 8th December, 2023.

[10] Herington, D., and Hengel, S. HUnit: A unit testing framework for Haskell. https://hackage.haskell.org/package/HUnit, Retrieved 6th April, 2024.

[11] Hutton, G. Higher-Order Functions for Parsing. *Journal of Functional Programming 2*, 3 (July 1992), 323–343.

[12] Hutton, G. *Programming in Haskell*, 2nd ed. Cambridge University Press, Cambridge, UK, 2016.

[13] Hutton, G., and Meijer, E. Monadic Parser Combinators. Department of Computer Science, University of Nottingham, Nottingham, UK. https://www.cs.nott.ac.uk/ pszgmh/monparsing.pdf, 1996.

[14] Ierusalimschy, R. *Programming in Lua*. Lua.org, 2003.

[15] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. The Evolution of Lua. In *Proceedings of the 3rd ACM SIGPLAN History of Programming Languages Conference (HOPL III)* (San Diego, California, USA, June 2007), ACM, pp. 2:1–2:26.

[16] Lerner, B. Lecture 11: Type Inference.
https://course.ccs.neu.edu/cs4410sp19/lec_type-inference_notes.html, Retrieved 31st March, 2024.

[17] LLVM Project. Clang: a C language family frontend for LLVM.
https://clang.llvm.org/, Retrieved 4th April, 2024.

[18] LLVM Project. llc - LLVM static compiler.
https://llvm.org/docs/CommandGuide/llc.html, Retrieved 4th April, 2024.

[19] LLVM Project. LLVM Language Reference Manual.
https://llvm.org/docs/LangRef.html, Retrieved 8th December, 2023.

[20] LLVM Project. The LLVM Compiler Infrastructure. https://llvm.org/, Retrieved 8th December, 2023.

[21] lua-users.org. Lua Uses. http://lua-users.org/wiki/LuaUses, Retrieved 10th October, 2023.

[22] Lua.org, PUC-RIO, Majumdar, D., and Ueyama, R.
New Compiler Framework in Ravi.
https://github.com/dibyendumajumdar/ravi/blob/master/readthedocs/ravi-compiler.rst, Retrieved 8th December, 2023.

[23] Lua.org, PUC-RIO, Majumdar, D., and Ueyama, R. Ravi Programming Language. https://github.com/dibyendumajumdar/ravi, Retrieved 8th December, 2023.

[24] Lua.org AND PUC-Rio. Lua 5.4 Reference Manual.
https://www.lua.org/manual/5.4/manual.html, Retrieved 8th December, 2023.

[25] Mol, M. Rosetta code. https://rosettacode.org/wiki/Rosetta_Code/, Retrieved 15th April, 2024.

[26] Muhammad, H. Teal. https://github.com/teal-language/tl/tree/master, Retrieved 8th December, 2023.

[27] Pall, M. LuaJIT. https://luajit.org/luajit.html, Retrieved 10th October, 2023.

[28] Pall, M. LuaJIT Performance.
https://staff.fnwi.uva.nl/h.vandermeer/docs/lua/luajit/luajit_performance.html, Retrieved 10th October, 2023.

[29] Pattis, R. E. EBNF: A Notation to Describe Syntax.
https://ics.uci.edu/ pattis/misc/ebnf2.pdf, Retrieved 2nd April, 2024.

[30] Pierce, B. C. *Types and Programming Languages*, 1st ed. MIT Press, Cambridge, Massachusetts, USA, 2002.

[31] The Titan Project Developers. Titan. https://github.com/titan-lang/titan, Retrieved 8th December, 2023.

[32] Thielemann, H., O'Sullivan, B., and Augustsson, L. llvm-tf: Bindings to the llvm compiler toolkit using type families.
https://hackage.haskell.org/package/llvm-tf, Retrieved 8th April, 2024.

A Benchmarking Results

A.1 Fibonacci Sequence

Fibonacci Sequence	My Compiler	Lua	LuaAOT	LuaJIT
Compile 1 (ms)	87	N/A	255	N/A
Compile 2 (ms)	70	N/A	263	N/A
Compile 3 (ms)	71	N/A	248	N/A
Compile 4 (ms)	67	N/A	249	N/A
Compile 5 (ms)	71	N/A	244	N/A
Compile Avg. (ms)	59	N/A	203	N/A
Run 1 (ms)	59	975	1307	118
Run 2 (ms)	60	972	1311	137
Run 3 (ms)	57	982	1303	114
Run 4 (ms)	58	973	1252	140
Run 5 (ms)	56	975	1249	101
Run Avg. (ms)	58	975.4	1284.4	122
Compile & Run Avg. (ms)	117	975.4	1487.4	122

A.2 Rod-Cutting Problem

Rod-Cutting Problem	My Compiler	Lua	LuaAOT	LuaJIT
Compile 1 (ms)	73	N/A	282	N/A
Compile 2 (ms)	66	N/A	285	N/A
Compile 3 (ms)	66	N/A	279	N/A
Compile 4 (ms)	67	N/A	280	N/A
Compile 5 (ms)	67	N/A	280	N/A
Compile Avg. (ms)	67.8	N/A	281.2	N/A
Run 1 (ms)	27	86	135	18
Run 2 (ms)	27	88	140	14
Run 3 (ms)	26	86	138	14
Run 4 (ms)	26	83	132	17
Run 5 (ms)	26	87	135	17
Run Avg. (ms)	26.4	86	136	16
Compile & Run Avg. (ms)	94.2	86	417.2	16

A.3 Floyd-Warshall Algorithm

Floyd-Warshall Algorithm	My Compiler	Lua	LuaAOT	LuaJIT
Compile 1 (ms)	363	N/A	501	N/A
Compile 2 (ms)	343	N/A	504	N/A
Compile 3 (ms)	351	N/A	492	N/A
Compile 4 (ms)	342	N/A	494	N/A
Compile 5 (ms)	346	N/A	502	N/A
Compile Avg. (ms)	349	N/A	498.6	N/A
Run 1 (ms)	23	314	513	8
Run 2 (ms)	23	300	500	7
Run 3 (ms)	23	288	506	7
Run 4 (ms)	23	286	519	7
Run 5 (ms)	22	291	546	7
Run Avg. (ms)	22.8	295.8	516.8	7.2
Compile & Run Avg. (ms)	371.8	295.8	1015.4	7.2