An interpreter for Lua in Rust

7CCS4PRJ Final Project Report

Author: Robert Bartlensky
Student ID: 1536771
Supervisor: Dr. Laurence Tratt
Programme of study: MSci Computer Science

April 2019

Originality avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Robert Bartlensky
7 April 2019

Abstract

The ecosystem of the Rust programming language does not have all the necessary tools to produce an efficient reimplementation of a dynamic language implemented in C. Despite the large number of Rust libraries available, there are no efficient general-purpose garbage collectors that can be used to build a virtual machine. The aim of this project is to test if Rust is a suitable host language for dynamic language implementations. This project shows that it is not possible to implement an efficient system, without implementing a higly specialised garbage collector.

Acknowledgements

I would like to thank my supervisor, Dr. Laurence Tratt, and his team for providing feedback on my work, and for guiding me throughout the project. I would also like to thank Dr. Edward Barrett for helping me benchmark my project.

Contents

Chapter 1

Introduction

Rust is a high-level systems programming language which focuses on memory-safety, and thread-safety. Rust has been used in a variety of settings with great success, from Command-line interfaces (CLIs) [1, 2], to web frameworks [3]. Rust also aims to be a replacement for C or C++, because it produces very optimised and fast code. One of the aims of this project is to assess whether Rust, in its current state, is a suitable language to build efficient virtual machines in.

The project aims to create an interpreter in Rust in order to prove the following hypothesis:

Hypothesis 1 *The libraries that implement garbage collectors for Rust are not suitable for building efficient virtual machines.*

This project builds a virtual machine (VM) for Lua in Rust, which uses a general purpose garbage collector called `gc`. The VM uses `gc` in order to perform garbage collection of Lua objects. The implementation is benchmarked against different Lua interpreters written in C, and Rust. The results show that the `gc` library slows down the VM considerably, and that other Lua implementations written in Rust, such as Luster, perform better because they implement a custom garbage collector. To better understand the performance implications of using `gc`, it was necessary to analyse the behaviour of the VM on a few benchmarks. The benchmarks revealed that the VM implemented for the purpose of this project is 1288% slower than PUC-Rio Lua. To understand why, a tool called Valgrind was used to determine the most frequently executed functions of the VM. The slowdown incurred from using `gc` is particularly obvious in the benchmark that implements the Sieve of Eratosthenes algorithm for computing prime numbers. This benchmark tests the table array optimisation which most Lua interpreters perform. However, executing the VM built in this project on this benchmark revealed the performance bottleneck of the `gc` library. More details about this are presented in chapter 9.

This project builds a VM for the Lua language, because currently there are no projects that implement Lua in Rust with a general-purpose garbage collector such as `gc`. Efficient garbage collectors are hard to implement. Reusing an existing library enables language designers to implement languages in a shorter amount of time, and also enables early prototyping.

The PUC-Rio Lua implementation provides an interface which can be used by C programs to call functions defined in Lua, and vice-versa. This is why Lua is said to be an embedable language. Writing a VM for Lua in Rust would enable Rust programs to do the same, without having to create bindings to the C API of PUC-Rio Lua. Although this project does not implement such an API, it can be easily extended to allow Rust projects to embed this VM into their system.

Another objective of the project is to illustrate the strengths and weaknesses of Rust when implementing interpreters for dynamic languages. Many components of the interpreter are written in safe Rust which means they benefit from the safety features of the language. However, the type system component is implemented using unsafe operations, due to the way types are represented in the interpreter. This project also shows the downsides of the representation of trait objects chosen by the Rust compiler. In particular, the representation of trait objects creates unnecessary indirection in the value system, which negatively affects the performance of the VM.

Chapter 2

Background

2.1 Stages of an interpreter

Interpreters are pieces of software that execute source files written in a particular programming language. An interpreter defines a virtual machine (VM) in which source code is executed. In order for a source file to be executed, it must first undergo a series of transformations.

2.1.1 Lexers and parsers

The first steps in any language implementation are the lexing and parsing of the source code.

Lexing is the process of splitting an input file (a stream of bytes) into a stream of predefined tokens (i.e. words). Each language defines a set of tokens which the lexer will use to split the input file. Usually, when the lexer finds a token that is not defined, it will stop the process, and report the error to the user.

A parser is a piece of software that assigns a meaning to a stream of tokens based on a grammar. A grammar is a set of rules which specify how tokens can be put together to form expressions. The input of a parser is the output of the lexer, which is used to produce an abstract syntax tree (AST), also called a parse tree.

In order to better understand these two concepts, consider the following example of a simple grammar that only consists of arithmetic expressions. The tokens of this language are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, and -. The grammar of this language is:

$$
\begin{aligned}
Z &::= 0 \\
D &::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
N &::= D \cdot N \mid Z \\
OP &::= OP \cdot + \cdot N \mid OP \cdot - \cdot N \mid N
\end{aligned}
$$

An AST for the input $2 + 3 - 1$ can be found in figure 2.1. Every internal node of the tree structure represents a symbol from the grammar, and every leaf node is token of language. Most of the time internal nodes are called non-terminals, and leaf nodes terminals.

2.1.2 Bytecode generation

The AST produced by the parser is translated into bytecode through a process called compilation. The compiler is a simple piece of software which traverses the parse tree in post-order, and produces bytecode based on the structure of the tree. A post-order traversal of a tree means that the children of a node are processed before the node itself.

The bytecode is a set of instructions which can be executed by an interpreter. There are two main types of bytecode instructions: register-based and stack-based.

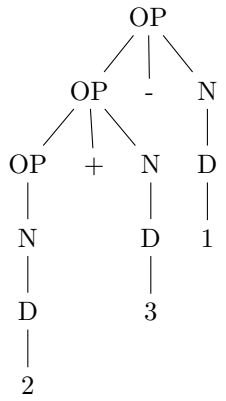

Figure 2.1: The AST for the input $2 + 3 - 1$

Register-based instructions

This type of instruction operates on virtual registers. The operands of an instruction are registers, and the result is usually stored in a register as well. A register can be thought of as a memory cell. The virtual machine provides a finite number of registers. As an example, consider a register-based instruction set for the simple arithmetic language:

$$\langle reg \rangle = add(reg1,\ reg2)$$
$$\langle reg \rangle = sub(reg1,\ reg2)$$

Figure 2.2: A register-based instruction set for the simple arithmetic language.

Listing 2.1: Sequence of register-based instructions for $2 + 3 - 1$

```
reg1 = 2
reg2 = 3
reg3 = add(reg1, reg2)
reg1 = 1
reg2 = sub(reg3, reg1)
```

In figure 2.2, $\langle reg \rangle$ specifies the register in which the result of the operation is stored, and $\langle reg1 \rangle$, $\langle reg2 \rangle$ are the registers which hold the operands of the instruction.

Listing 2.1 shows the bytecode a compiler might emit for the input $2 + 3 - 1$. Note that in this case, the compiler reuses registers 1 and 2, and the final result is stored in register 2.

Stack-based instructions

Stack-based instructions operate on a stack[1] which is stored by the VM. Each operation is performed relative to the top of the stack, which the VM keeps track of.

Figure 2.3 shows a simple stack-based instruction set for the arithmetic language, where $\langle num \rangle$ is a placeholder for a number. The $PUSH$ instruction adds its operand to the top of the stack. The ADD instruction removes the two top-most values from the stack, and pushes their sum back to the stack. The SUB instruction works similarly to the ADD instruction.

Listing 2.2 shows how the compiler might emit the bytecode for $2 + 3 - 1$. After the bytecode is executed, the stack of the VM will have one value, which is the result of the arithmetic expression.

[1] A stack is a simple First-In-Last-Out (FILO) data structure

$PUSH\ \langle num \rangle$
ADD
SUB

Figure 2.3: A stack-based instruction set for the simple arithmetic language.

Listing 2.2: Sequence of stack-based instructions for $2 + 3 - 1$

```
PUSH 2
PUSH 3
ADD
PUSH 1
SUB
```

2.1.3 Bytecode execution

The bytecode generated by the compiler is executed by the VM. VMs follow a fetch-decode-execute cycle in order to execute the instructions of the bytecode, also called the dispatch loop. Interpreters fetch an instruction, decode it in order to check its type and operands, and then execute it. Each operation defined in the bytecode has an associated implementation in the VM.

2.2 Garbage collector (GC)

In programming languages such as C or C++, heap allocated memory must be released manually, or the program leaks memory. Garbage collectors are used to track allocations, and release the memory that is unreachable by the program. Garbage collectors ease the burden of manual memory management. Since garbage collection happens at run-time, the execution of a program can be influenced negatively, depending on how often the GC checks to see if there is any memory to be freed.

2.3 Compilers

The compiler used by an interpreter is usually very simple, and does not perform any major optimisations. However, there is a simple optimisation that most compilers perform, called constant folding.

2.3.1 Constant folding

This is a straightforward optimisation where operations between constant values are performed at compile-time, rather than at run-time. This has the benefit of not emitting unnecessary instructions. The more instructions the interpreter has to execute, the longer the execution of the program will be. If this optimisation is enabled during compilation, then the code from listing 2.1 is modified to:

Listing 2.3: $2 + 3 - 1$ compiled with constant folding enabled

```
1 reg3 = 5
2 reg1 = 1
3 reg2 = sub(reg3, reg1)
```

On line number 1 of listing 2.3 the compiler folded $2 + 3$ into 5, and placed the result into register 3.

2.3.2 Constant propagation

More complex compilers might also perform constant propagation. This optimisation operates on the bytecode, replacing all occurrences of those registers which have a constant value with the value itself. This means that the compiler does not emit unnecessary store instructions. When both constant folding and propagation are enabled, the compiler emits the following bytecode:

Listing 2.4: $2 + 3 - 1$ compiled with constant folding and propagation enabled

```
reg2 = 4
```

Most interpreters choose to omit this optimisation, and only focus on constant folding [4].

2.4 Intermediate representation (IR)

In general, compilers do not translate ASTs to bytecode or machine code directly, but to an intermediate representation. This representation is compiler-specific, and it is used solely for optimisation purposes. Some compilers [5] even use multiple IRs in order to perform more fine-grained optimisations at different levels of abstraction.

2.4.1 Static single assignment form (SSA)

This is a register-based IR used by many compilers [6, 7, 8]. Other intermediate representations include Continuation-passing style (CPS) which is used by the Haskell compiler, and is more suitable for compiling functional languages [9], or a control flow graph (CFG) which is a representation used by compilers to translate imperative languages to machine code. The GNU C Compiler (GCC) moved from CFG to SSA, because it reduced the complexity of the compiler [10].

In SSA each variable, also called register, can only be assigned to once. This restriction allows compilers to perform better optimisations on the code before emitting the final executable. SSA also enables compilers to perform register allocation [11] efficiently, and other optimisation such as constant propagation and folding [12].

Listing 2.5: $2 + 3 - 1$ in SSA form

```
1 reg1 = 2
2 reg2 = 3
3 reg3 = add(reg1, reg2)
4 reg4 = 1
5 reg5 = sub(reg3, reg4)
```

Listing 2.5 shows the bytecode from listing 2.1 in SSA form. Each register is assigned to only once, but the values of registers can be read multiple times.

Basic block

A basic block is an ordered collection of instructions. Each block has a single defined entry point (which is usually the first instruction), and a single exit point. The exit point of a block is marked by a jump or branching instruction. These two instructions define clear parent-child relationships between basic blocks. In order to better illustrate the concept of basic blocks, consider the Lua code from listing 2.6, and its SSA representation in listing 2.7.

In this case, there are 4 basic blocks. BB_1 is the block in which the first `x` is declared, and it also contains a branching instruction to BB_2 and BB_3. Both BB_2 and BB_3 declare their own register for `x` and jump to BB_4. At run-time, the execution starts in BB_1, then moves to either BB_2 or BB_3 depending on whether the branching condition is true or not, and then continues to BB_4. Basic blocks can be thought of as nodes in a control-flow graph. Figure 2.4 shows the parent-child relationships that are formed between basic blocks. This figure is also the control-flow graph of the program from listing 2.6.

Listing 2.6: A simple Lua program

```
local x = 1 -- define a local variable x
if x > 0 then
  local x = 2 -- shadow the outer x, and declare a new variable x
else
  local x = 3 -- same as above, but a different scope
end
```

Listing 2.7: SSA version of the simple Lua program

```
1  BB1: {
2    x1 = 1
3    Jump to BB2 if x1 > 0 else jump to BB3
4  }
5  BB2: {
6    x2 = 2
7    Jump to BB4
8  }
9  BB3: {
10   x3 = 3
11   Jump to BB4
12 }
13 BB4 { }
```

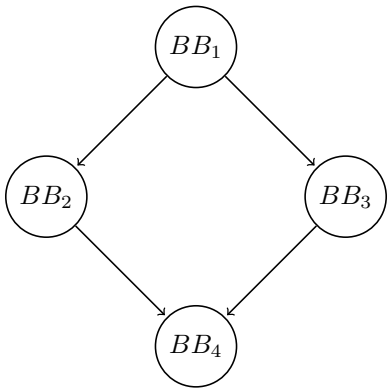

Figure 2.4: Control-graph and the parent-child relationships between basic blocks

Phi (ϕ) instructions

These are special instructions that are used to 'merge' registers defined in different basic blocks. A ϕ instruction is used to choose the value of a particular variable depending on the execution path that the program has taken at run-time. Often, programs change the value of a variable based on a condition. Because in SSA a variable cannot be reassigned multiple times, a ϕ instruction is used instead to express the fact that the value of the variable depends on the execution path.

On line 14 in listing 2.9 a ϕ instruction is used to merge the registers `x2` and `x3` into `x4` . This means that `x4` has the value of either `x2` or `x3` , depending on which execution path was taken at run-time. The ϕ instruction is necessary because BB_2 and BB_3 modify an outer variable, thus it is important for the compiler to preserve the meaning of the original program without breaking the SSA property. If the path of execution goes through `BB2`, then `x4` is assigned the value of `x2`.

ϕ instructions are removed during register allocation, and the operands of the instruction are usually

Listing 2.8: A modified version of the simple Lua program

```
local x = 1 -- define a local variable x
if x > 0 then
  x = 2 -- reassign the outer ‘x‘ to 2
else
  x = 3 -- reassign the outer ‘x‘ to 3
end
```

Listing 2.9: SSA version of listing 2.8

```
1  BB1: {
2    x1 = 1
3    Jump to BB2 if x1 > 0 else jump to BB3
4  }
5  BB2: {
6    x2 = 2
7    Jump to BB4
8  }
9  BB3: {
10   x3 = 3
11   Jump to BB4
12 }
13 BB4 {
14   x4 = ϕ(x2, x3)
15 }
```

allocated to the same physical or virtual register.

Dominators

A block B_2 is dominated by a block B_1 if and only if all paths from B_1 to some other block B_n must pass through B_2. Note that the definition also holds for $B_n == B_2$. A dominator tree is a tree structure which links basic blocks with their dominators. Dominator trees are used in order to check what variables are in scope in a particular basic block. These can be found by traversing the dominator tree upwards towards the root. Figure 2.5 shows the dominator links of the program from listing 2.8.

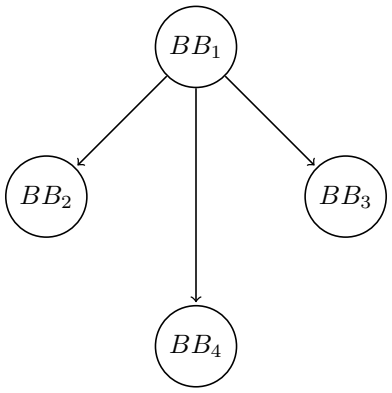

Figure 2.5: Dominator links

Note that in figure 2.5 there are no links between BB_2 and BB_4, or between BB_2 and BB_4, because the

dominator property does not hold. However, there is a link between BB_1 and BB_4 because all paths from BB_1 end in BB_4: $BB_1 \rightarrow BB_2 \rightarrow BB_4$ and $BB_1 \rightarrow BB_3 \rightarrow BB_4$. Also, any variable that is defined in BB_1 is in scope in all three blocks, due to the direct link between them in the dominator tree. However, variables defined in either BB_2 or BB_3 are not in scope in BB_4 because there is no edge between these blocks.

2.5 Related work

Although this project is implementation oriented, there are some articles and books which provided great support during development.

The "Compilers Principles, Techniques, & Tools" book [13] provided a great introduction to compilers. Chapter 6 was the most relevant, as it introduced the concept of intermediate representations, and the different types of IR that a compiler might use.

The design of the interpreter was also influenced by the PUC-Rio Lua implementation [14], which was used as a second reference when the language manual was unclear.

The interpreter built in this project is benchmarked against three different VMs: PUC-Rio Lua, LuaJIT, and Luster.

2.5.1 PUC-Rio Lua

The PUC-Rio Lua VM is the most up to date implementation of the Lua language, and it is written in C. The aim of this interpreter is to provide a small implementation, and a VM which performs well. Other goals include portability, and embeddability [15].

Register-based bytecode

Version 5 of the VM introduced many changes to the bytecode and to the way table types are implemented. In version 5, the VM moved from a stack-based virtual machine to a register-based one. The main benefit of this change was that the emitted bytecode size was reduced. The new bytecode representation also improved the running time of the VM.

Table array optimisation

In version 5, the developers of Lua also introduced an interesting optimisation for tables. Tables are associative arrays, where keys and values can be of any type. Lua does not have arrays, only tables. Arrays can be 'emulated' with integer keys. In earlier versions, tables were implemented as hash tables. This means that even integers were looked-up using the hash table. However, the array optimisation changes the way tables are represented. Tables now have two components: an array used for integer look-ups, and a hash map for all the other types of look-ups. The look-up algorithm is straightforward, as the value of an integer attribute, `i`, is fetched from `array[i]`, if `i` is not out of bounds. In all other cases, including the out-of-bounds case, the look-up is done through the hash map.

2.5.2 LuaJIT

LuaJIT is a tracing Just-in-Time (JIT) compiler for Lua [16]. A tracing JIT is a virtual machine that collects traces of the most executed sequences of bytecode instructions. Once the trace is collected, the tracing compiler translates the trace of instructions into machine code, e.g. x86-64 assembly. If the VM starts executing a path for which the tracing compiler generated machine code, then the machine code is executed instead of the bytecode instructions.

The first stage in the execution of a Lua source file is the compilation to bytecode. LuaJIT compiles Lua into bytecode through the use of a simple compiler, which only performs a few optimisations [17]. This phase is executed quickly, as most of the optimisations are done at run-time. The bytecode produced by LuaJIT is not compatible with the bytecode produced by PUC-Rio Lua.

SSA IR

When the LuaJIT VM starts tracing frequently executed code, it collects bytecode instructions, and at the same time translates them into SSA instructions. The JIT compiler uses this representation to emit machine code. The LuaJIT VM performs many optimisations on the SSA trace [18], to ensure that the machine code output is very efficient. The VM supports many architectures including: ARM, PowerPC, and x86.

Similarities to PUC-Rio Lua

The LuaJIT project reuses some of the data structures implemented by PUC-Rio Lua. Implementing garbage collectors that perform well is a hard task, and the LuaJIT project focuses more on implementing a JIT compiler. Thus, the project borrows the garbage collector implemented by PUC-Rio Lua, as it performs well, and it is more specialised for Lua.

LuaJIT also performs the table array optimisation implemented in PUC-Rio Lua.

2.5.3 Luster

Luster is a Lua VM written entirely in Rust. The aim of the Luster project is to create a garbage collector more suitable for language implementations, and apply it to implement a small dynamic language like Lua. The implementation of Luster is based on the one of PUC-Rio Lua. This means that Luster also performs, for instance, the table array optimisation [19].

'Stackless' style

In Luster, function calls are implemented in 'stackless' style. This means that calls are not implemented recursively. Each `Call` instruction creates a new object, called a `Sequence`, which is passed back to the VM. The `Sequence` structure has a method called `step`, which is used to execute the instructions of the function one-by-one. The VM collects `Sequence`s, and `step`s through them until they are consumed.

Luster performs a garbage collection in between calls to `step`, which the author describes as continuous garbage collection. Other garbage collectors choose to have predefined points where they collect unused objects [20].

Chapter 3

Requirements

3.1 Functional requirements

- The Lua interpreter should support most features of Lua 5.3.
- All features should be implemented as described in the Lua language specification.
- Both the compiler, and the VM must be built in Rust.
- The interpreter should be able to run a few benchmarks provided by the `luajit` testing suite.

3.2 Non-functional requirements

- The virtual machine should perform well, and should execute benchmarks in a reasonable amount of time.
- Both the compiler, and the VM should not use unsafe code, unless it is strictly necessary.
  Using the `unsafe` keyword in Rust means that all the guaratees the language offers are turned off. This means that the programmer must ensure that those safety guaratees are met.

Chapter 4

Design

This chapter describes in detail the main components of the Lua interpreter.

4.1 General architecture

This project follows the general architecture of interpreters described in section 2.1. This means that there are three major components: the lexing and parsing component, the compiler, and the virtual machine. Figure 4.1 shows the high-level process of executing a Lua source file.

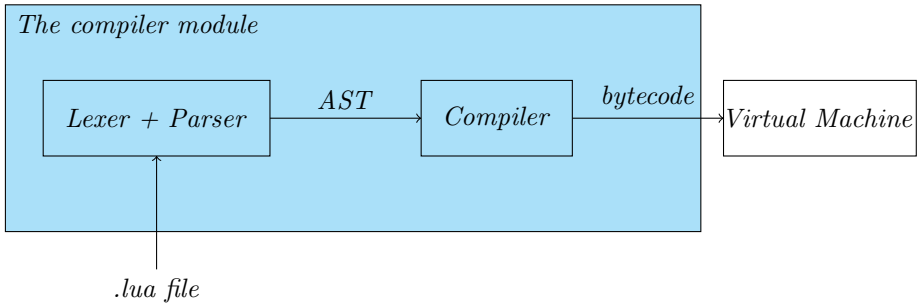

Figure 4.1: Architecture diagram of the process of executing a Lua file.

This project implements the `Compiler` and `Virtual Machine` components from figure 4.1. The lexer and the parser were generated with a parser generator. Parser generators are libraries that take as input a list of tokens and a grammar, and produce code that is able to generate ASTs from source files.

4.2 The compiler component

This component is responsible for transforming the AST into bytecode that can be understood by the virtual machine. This process is shown in figure 4.2.

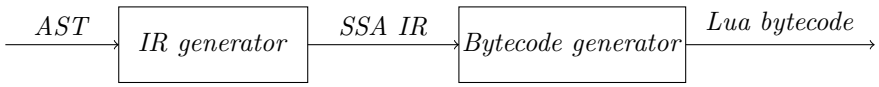

Figure 4.2: Architecture diagram of the process of compiling the AST to bytecode.

Figure 4.2 is a more detailed version of the compiler component from figure 4.1.

4.2.1 IR generator

The IR generator is a sub-component of the compiler module. It takes as input the parse tree of the file that is being compiled, and produces an SSA IR. Figure 4.3 shows a UML diagram of the IR generator. In the first stage, the `LuaParseTree` structure lexes and parses the Lua file, and produces a parse tree, which is supplied to the `IRGen` structure. The responsiblity of the `IRGen` structure is to generate the IR of the input AST.

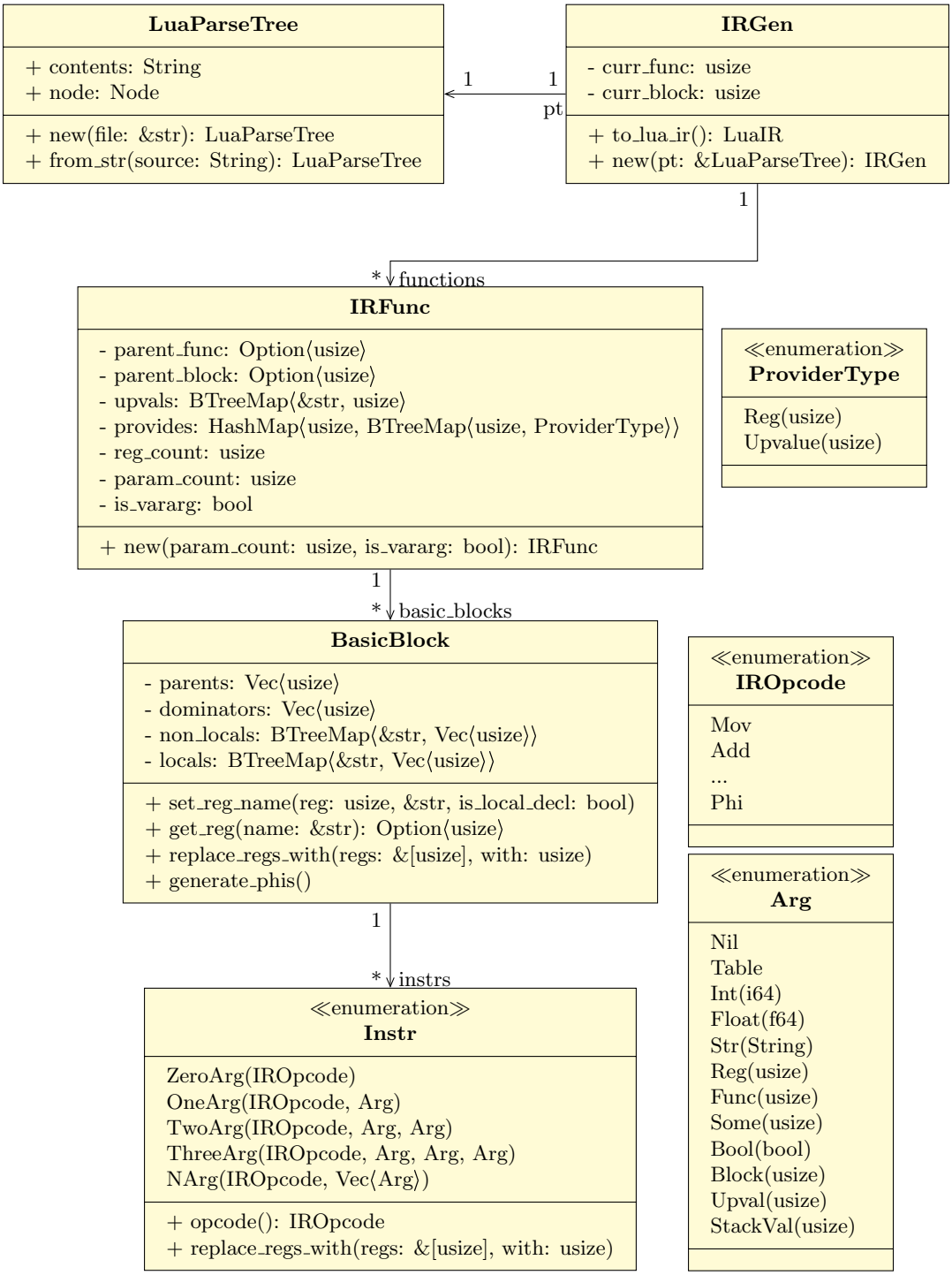

Figure 4.3: UML diagram showing the structure of the *IR generator* component.

4.2.2 The SSA IR of the Lua compiler

Although the compiler does not perform any optimisations, using an SSA IR allows the compiler to be extended in the future with a register allocation algorithm, and optmisations such as constant folding. The intermediate representation of the Lua compiler consists of functions, basic blocks, and instructions. Each of these components is part of the UML diagram from figure 4.3.

Functions

Functions, represented by the `IRFunc` entity, are used to model functions in Lua. A function contains an ordered collection of basic blocks. In order to better understand what the different attributes of `IRFunc` are used for, consider the following Lua snippet.

Listing 4.1: A Lua program which uses the concept of upvalues

```
global_a = 1 -- a global variable
local a = 1 -- a local variable
function global_f(b) -- a global function
  print(c) -- prints nil, and not 2!
  return a + global_a + b
end
local c = 2
print(global_f(3)) -- prints 5
```

The Lua language uses lexical scoping, which means that a variable is in scope if it was defined in the current, or parent scopes. To better illustrate this concept, consider the function `global_f` from listing 4.1. It takes a parameter `b`, and returns the sum of `a`, which is declared in the outer scope *before* the function, `global_a`, and the paramter `b`. An upvalue in Lua is a local variable defined in an outer scope that is read or modified by a function from an inner scope. In the case of `global_f`, `a` is an upvalue because it is local, it is defined in an outer scope, and it is read by `global_f`. However, `c` is not an upvalue, because it is not defined *before* `global_f`. Thus, `c`, in this case, is a global variable. After the definition of `global_f`, there is a declaration of a new local variable `c`, although when processing `global_f` there is no local `c` in scope. The `global_f` function is also a global symbol.

Every function and module in Lua has an upvalue called `_ENV`. This upvalue is a table which stores all global symbols. For instance `x = 1` is equivalent to setting the `"x"` attribute of `_ENV` to 1. Thus, `x = 1` is equivalent to `_ENV["x"] = 1`. In Lua, if a symbol is not declared in an outer scope, then it is fetched from the `_ENV` upvalue.

Listing 4.2: Another Lua program which uses the concept of upvalues.

```
local function f(n)
  if n <= 0 then
    return 0
  else
    return f(n - 1)
  end
end
```

Lua also has the concept of local functions. These are functions which can have themselves as an upvalue. For example, in listing 4.2 the upvalues of `f` are `_ENV` and `f`. A more verbose version of function `f` can be seen in listing 4.3.

The `upvals` attribute of the `IRFunc` entity models the upvalues of a function. It is a mapping from names to indices. For instance in the case of function `f` from listing 4.2, the upvalues map has the values: { ”f” $\rightarrow$ 0

Listing 4.3: An equivalent program to the one in listing 4.2.

```
local f;
f = function(n)
  if n <= 0 then
    return 0
  else
    return f(n - 1)
  end
end
```

}. _ENV is not included in the mapping as an optimization. The index of `0` means that `f` is the first upvalue of the function, after _ENV .

Every function knows what upvalues it uses, however it is also very imporant to record what variables each function 'provides' to other functions as upvalues. More details about the `provides` attribute are given in chapter 5.

A function can have another function as a parent. In listing 4.2, function `f` has the top-level module as a parent. In this project, the top-level module is modelled as a function. In Lua, source files are also called modules or chunks.

Basic block

The `BasicBlock` entity follows the definition of basic blocks given in chapter 2. The methods defined by `BasicBlock`s are described in chapter 5. The `non_locals` and `locals` attributes define mappings between high-level variable names and registers of the IR. The `locals` attribute holds all the variables that have been declared as `local` in the basic block, and the `non-locals` mapping holds the outer *local* variables that the basic block modified.

Instr

Instructions are modelled as 'meta-instructions'. Each type of instruction is defined by an `IROpcode`, and its operands. Instructions are represented this way in order to perform lowering to bytecode more efficiently. The `Arg` enumeration is used to specify the type of an operand. The `Some` variant of `Arg` is a special operand, which does not name any type. The reason for its existence is explained in a later section.

IROpcode

This enumeration defines all the instructions supported by the IR. Table 4.1 gives a detailed description of the meaning of each opcode. $Arg(n)$ refers to the n^{th} operand. Reg_A, Reg_B, Reg_C are placeholders for specific registers that are encoded in an instruction. $\langle OP \rangle$ is a placeholder for a binary operation supported by the compiler, such as: add, sub, equals, less than, etc.

4.2.3 IR generation with `IRGen`

The main task of the `IRGen` structure is to create an intermediate representation of the input Lua program. The lexer and parser generate a parse tree, which is then recursively traversed by `IRGen` in order to create a `LuaIR` structure, which can later be translated into bytecode.

The compiler does not support all the operations defined in the Lua reference manual. However, this section also describes possible solutions for the missing functionality. The following sections describe in detail how different constructs of the Lua language are translated into `LuaIR`.

Mov	$R_A = R_B$
$\langle OP \rangle$	$R_A = R_B \ \langle OP \rangle \ R_C$
GetAttr	$R_A = R_B[R_C]$
SetAttr	$R_A[R_B] = R_C$
Closure	$R_A = Closure(Arg(2))$
Call	Call closure R_A
Push	Push R_A to the stack
VarArg	Copy the $Arg(2)^{th}$ vararg parameter into R_A
MovR	Copy the $Arg(2)^{th}$ return value of the previous call into R_A
Rer	Return to the parent function
SetTop	Mark the stack frame boundary of closure R_A
GetUpAttr	$R_A = Upvals[Arg(2)][R_C]$
SetUpAttr	$Upvals[Arg(1)][R_B] = R_C$
Jmp	Jump to block $Arg(1)$
JmpNe	If $R_A == true$ then jump to block $Arg(2)$ else jump to $Arg(3)$
JmpEq	Opposite of JmpNe
GetUpVal	$R_A = Upvals[Arg(2)]$
SetUpVal	$Upvals[Arg(1)] = R_B$
Umn	$R_A = -R_B$
Phi	A ϕ instruction

Table 4.1: The opcodes of the IR explained in detail.

Lua blocks

Blocks in Lua are sequences of instructions, and can optionally return values. Usually, each Lua block is compiled into a `BasicBlock`. However, there are cases in which multiple `BasicBlock`s are needed to represent a single Lua block. The reason why blocks can optionally return values is because the body of a function is represented as a block.

Variable assignments

In the Lua language there are two types of variables: local, and global. The language also supports multiple assignments in one statement such as `a, b = 1, 2`. In the IR, each variable assignment is stored in a new register.

Listing 4.4: Local assignments in Lua.

```
local a, b = 1, 2
a, b = 3, 4
```

Listing 4.5: The SSA IR of listing 4.4.

```
1 BasicBlock1 {
2   Mov Reg(0), Int(1) # Copy 1 into register 0
3   Mov Reg(1), Int(2) # Copy 2 into register 1
4   Mov Reg(2), Int(3) # Copy 3 into register 2
5   Mov Reg(3), Int(4) # Copy 4 into register 3
6   Phi Reg(0), Reg(2) # merge registers of a
7   Phi Reg(1), Reg(3) # merge registers of b
8 }
```

The `non-locals` attribute of `BasicBlock1` from listing 4.5 contains an empty map because the block does not modify any outer local variable. However, during the compilation of `BasicBlock1`, the `locals` attribute contains the following mappings: "a" $\rightarrow [0, 2]$; "b" $\rightarrow [1, 3]$. When the compiler fully processes a block, it calls the `generate_phis` method of `BasicBlock`. This method is used to merge the different uses of a symbol into a single register. In this case, the compiler merges the registers of 'a' into register 0, and registers of 'b' into 1. The reason why the compiler emits `Phis` to merge local variables from the same block is described in chapter 5.

Tables and _ENV

In order to better understand how global variables work, it is necessary to introduce tables, and how the compiler translates them into IR.

Listing 4.6: Local table in Lua.

```
local t = {}
t[1] = 2
t["string"] = 3
```

Listing 4.7: The SSA IR of listing 4.6.

```
1 BasicBlock1 {
2   Mov Reg(0), Table # Create the table t in register 0
3   Mov Reg(1), Int(2) # process the value
4   Mov Reg(2), Int(1) # process the key
5   SetAttr Reg(0), Reg(2), Reg(1) # perform t[1] = 2
6   Mov Reg(3), Int(3)
7   Mov Reg(4), Str(string)
8   SetAttr Reg(0), Reg(4), Reg(3) # perform t["string"] = 3
9 }
```

Listings 4.6 and 4.7 show how tables are compiled to IR. Note that both `1` and `"string"` are keys of the table `t`. In order to compile table look-ups, `IRGen` first recursively compiles the right-hand-side of the expression, and then recursively compiles the left-hand-side. Having compiled both sides, the compiler can now emit the correct instructions to assign the right-hand-side to the left-hand-side.

As mentioned earlier, the `_ENV` variable is a table which is always the first upvalue of any function. The compiler considers a Lua module as a function.

Listing 4.8: Global assignment in Lua.

```
x = 1 -- _ENV["x"] = 1
```

In listing 4.8, `x` is a global variable, thus the attribute `"x"` of `_ENV` is assigned the value of `1`. Listing 4.9 shows the main use case of the `Upvalue` variant of the `Arg` enumeration. The compiler emits a `SetUpAttr` instruction for upvalue `0`, which is `_ENV`, in order to store the new value of `x`.

Binary expressions

Each binary operation of the Lua language has an associated `IROpcode`. The compiler handles most binary expressions in the same way, except for `and` and `or`. Binary expressions are translated recursively. First the compiler generates code for the left operand, and stores it into a register, then it does the same for the right

Listing 4.9: The SSA IR of listing 4.8.

```
IRFunction1 {
  BasicBlock1 {
    Mov Reg(0), Int(1)
    SetUpAttr Upvalue(0), Str("x"), Reg(0) # _ENV["x"] = 1
  }
}
```

operand. Finally, the compiler emits an instruction, which represents the binary operation. The arguments of this instruction are the two registers that contain the values of the two operands. To illustrate this, consider the program from listing 4.10, and its associated IR from listing 4.11.

Listing 4.10: A binary operation example.

```
local x = 1 + 2
```

Listing 4.11: The SSA IR of listing 4.10

```
BasicBlock1 {
  Mov Reg(0), Int(1) # left operand
  Mov Reg(1), Int(2) # right operand
  Add Reg(2), Reg(0), Reg(1) # Reg(2) = Reg(0) + Reg(1)
}
```

It is also worth mentioning that the `locals` attribute of `BasicBlock1` from listing 4.11 contains a mapping from `x` to register `2`. This is a small optimisation done by the compiler in order to avoid emitting an extra `Mov` instruction.

OR short-circuiting

The Lua compiler performs `or` short-circuiting, which means that given an `or` expression, if the left operand is true, then the right operand is not evaluated. The compiler performs this optimisation by emitting multiple basic blocks, and branching instructions.

Listing 4.12: An `or` operation example.

```
local x = 1 or 2
```

Each operand of the `or` operation is compiled in its own basic block as shown in listing 4.13. The compiler emits a branching instruction, `JmpEq`, in `BasicBlock1` in order to check the truth value of the left operand expression. If it is true, then the VM is instructed to jump to `BasicBlock3`. If is it false, then the VM jumps to `BasicBlock2`, where it computes the truth value of the right operand. After the `or` expression is executed, the VM resumes the execution in `BasicBlock3`. Note that the `locals` attribute in both `BasicBlock1` and `BasicBlock3` contains an entry for `x`, while `BasicBlock2` does not.

`AND` short-circuiting

The `and` operator is compiled very similarly to the `or` operator. The main difference between the two is that if the left operand of the `and` operation evaluates to a false value, then the right operand is not evaluated.

Listing 4.13: The SSA IR of listing 4.12.

```
# locals = { 'x' -> [0] }
BasicBlock1 {
  Mov Reg(0), Int(1)
  JmpEq Reg(0), Block(2), Block(3)
}
# non-locals = { 'x' -> [1] }
BasicBlock2 {
  Mov Reg(1), Int(2)
  Jmp Block(3)
}
# locals = { 'x' -> [0] }
BasicBlock3 {
  Phi Reg(0), Reg(1)
}
```

Listing 4.14: An `and` operation example.

```
local x = 1 and 2
```

Listing 4.15: The SSA IR of listing 4.14.

```
# locals = { 'x' -> [0] }
BasicBlock1 {
  Mov Reg(0), Int(1)
  JmpNe Reg(0), Block(2), Block(3)
}
# non-locals = { 'x' -> [1] }
BasicBlock2 {
  Mov Reg(1), Int(2)
  Jmp Block(3)
}
# locals = { 'x' -> [0] }
BasicBlock3 {
  Phi Reg(0), Reg(1)
}
```

Another notable difference between the two operations, is that the compiler emits a `JmpNe` in the case of `and` instead of a `JmpEq` instruction as seen in listing 4.15.

Unary operators

The compiler only supports the `-` unary operator. Because `-` only has one argument, the compiler simply evaluates it, and stores the negation of its value in a new register. Other unary operators work similarly, and the compiler can be easily extend to support them.

If statements

Conditionals are an essential construct in any language. The compiler, as with many other constructs, emits several basic blocks in order to translate an if statement into `LuaIR`.

In listing 4.17, `BasicBlock1` is the block in which `if cond_a` from listing 4.16 is compiled. If the condition is true, then the VM jumps to `BasicBlock2` where `a` is set to `2`. In `BasicBlock2`, the `Jmp` instruction is used

Listing 4.16: An if statement in Lua.

```
local cond_a = true
local cond_b = false
local a = 1
if cond_a then
  a = 2
elseif cond_b then
  a = 3
else
  a = 4
end
```

Listing 4.17: The SSA IR of listing 4.16.

```
# locals: { 'a' -> [2] }
BasicBlock1 {
  Mov Reg(0), Bool(true) # cond_a = true
  Mov Reg(1), Bool(false) # cond_b = false
  Mov Reg(2), Int(1) # local a = 1
  JmpNe Reg(0), Block(2), Block(3) # if cond_a ...
}
# non-locals: { 'a' -> [3] }
BasicBlock2 {
  Mov Reg(3), Int(2) # a = 2
  Jmp Block(6)
}
BasicBlock3 {
  JmpNe Reg(1), Block(4), Block(5) # if cond_b ...
}
# non-locals: { 'a' -> [4] }
BasicBlock4 {
  Mov Reg(4), Int(3) # a = 3
  Jmp Block(6)
}
# non-locals: { 'a' -> [5] }
BasicBlock5 {
  Mov Reg(5), Int(4) # a = 4
  Jmp Block(6)
}
# locals: { 'a' -> [2] }
BasicBlock6 {
  Phi Reg(2), Reg(3), Reg(4), Reg(5)
}
```

to jump over the basic blocks which represent the other branches of the if statement. `BasicBlock3` contains the instructions which check whether `cond_b` is true or not. `BasicBlock6` contains a `Phi` instruction which merges all the uses of `a` into a single register. This is necessary because all the blocks modify the value of `a`. Listing 4.17 also specifies which basic blocks have their `non-locals` attribute set. In the case of `BasicBlock` 2, 4, and 5, `a` is a non-local because `a` was declared in an outer block, but reassigned in the current block. This attribute helps the compiler emit `Phi` instructions. Also note that that `a` is local in `BasicBlock6`. The reasoning behind it is that `BasicBlock6` is the continuation of `BasicBlock1`, in which `a` is a local, therefore `a` should remain a local variable even in `BasicBlock6`. It is also worth mentioning that the parent basic blocks of `BasicBlock6` are `BasicBlock` 2, 4, and 5. This is because the `Jmp` instruction defines a parent-child

relationship between two basic blocks. `BasicBlock1` is also the dominator of `BasicBlock6`, because all paths that start in `BasicBlock1` end in `BasicBlock6`.

While loops

While loops are compiled similarly to if statements. This is the first construct which introduces cycles in the control-flow graph.

Listing 4.18: A while statement in Lua.

```
local i = 0
while i < 10 do
  i = i + 1
end
```

Listing 4.19: The SSA IR of listing 4.18.

```
# locals: { 'i' -> [0] }
BasicBlock1 {
  Mov Reg(0), Int(0) # local i = 0
  Jmp Block(2)
}
BasicBlock2 {
  Mov Reg(1), Int(10)
  Lt Reg(2), Reg(0), Reg(1) # R(2) = i < 10
  JmpNe, Reg(2), Block(3), Block(4)
}
# non-locals: { 'i' -> [4] }
BasicBlock3 {
  Mov Reg(3), Int(1)
  Add Reg(4), Reg(0), Reg(3) # i = i + 1
  Jmp Block(2) # jump back to recheck the while condition
}
# locals: { 'i' -> [0] }
# dominators: [1]
# parents: [2]
BasicBlock4 {
  Phi Reg(0), Reg(4)
}
```

While loops have a condition, and execute the body until the condition becomes false. In order to model this, the compiler emits 3 different basic blocks as shown in listing 4.19. The first basic block, `BasicBlock2`, is the basic block in which the truth value of the condition is checked. If the condition is true, then execution moves to `BasicBlock3`, or to `BasicBlock4` otherwise. `BasicBlock3` is the body of the while-loop. This contains a `Jmp` instruction to `BasicBlock2` which rechecks the truth value of the condition. `BasicBlock4` contains a `Phi` instruction which merges the two uses of `a` into register 0. Just like in the case of the `if` construct, `BasicBlock4` has `BasicBlock1` as a dominator for the same reasoning: all paths from `BasicBlock1` end up in `BasicBlock4`. `BasicBlock2` is a dominator of `BasicBlock4`, however the compiler does not create a dominator link for optimisation purposes. This is because `BasicBlock2` does not modify any variables, as such there is no need to link it to `BasicBlock4`.

Do blocks

The `do..end` construct is used to create a new scope in Lua.

Listing 4.20: A do block in Lua.

```
local i = 0
do
  i = 1
  local i = 2
end
-- i == 1
```

Listing 4.21: The SSA IR of listing 4.20.

```
# locals = { 'i' -> [0] }
BasicBlock1 {
  Mov Reg(0), Int(0)
}
# locals = { 'i' -> [2] }
# non-locals = { 'i' -> [1] }
BasicBlock2 {
  Mov Reg(1), Int(1) # i = 1
  Mov Reg(2), Int(2) # local i = 2
}
# locals = { 'i' -> [0] }
# dominators: [1]
# parents: [2]
BasicBlock3 {
  Phi Reg(0), Reg(1)
}
```

In listing 4.21, the compiler emits a new basic block, where it adds the instructions of the `do` block. `BasicBlock2` modifies both local variables and non-local ones, thus the `locals` and `non-locals` attributes both contain an entry for `i`. Note that if a block does not end with a jump instruction, then it is assumed that it jumps to the next basic block. In this case, `BasicBlock1` jumps to `BasicBlock2`.

For loops

There are two types of for loops in Lua. The first kind, the counter based for loop, is a loop which executes its body a number of times.

Listing 4.22: A for statement in Lua.

```
1 local j = 0
2 for i=0,10,1 do -- equivalent to 'for i=0,10 do'
3   j = j + 1
4 end
5 -- j has the value of 11
```

The `for` loop from listing 4.22 executes 11 times. The `i` variable is the counter, and has an initial value of 0. The third integer from line 2 represents the step of the loop. At each iteration, the counter is incremented step times. The loop stops when the value of `i` is greater than 10, which is called the `bound`. The step must be an integer or float value, and can also be negative. When the step has a negative value, the bound is treated as a lower bound. It is also important to note that `i` is a local variable, which goes out of scope after the `for` loop. This means that only the body of the loop is able to use the value of `i`.

Listing 4.23: A `for` statement with a negative step.

```
local j = 0
for i=0,-10,-1 do
  j = j + 1
end
-- j has the value of 11
```

The compiler handles for loops by translating them into while loops. The program from listing 4.22 is translated into the program from listing 4.24. The IR for listing 4.24 is omitted for the sake of brevity. The main point is that the compiler reuses existing functionality in order to generate `for` loops.

Listing 4.24: A for statement in Lua

```
1  local j = 0
2  do
3    local i, bound, step = 0, 10, 1
4    while (step >=0 and i <= bound) or (step < 0 and i >= bound) do
5      j = j + 1
6      i = i + step
7    end
8  end
9  -- j has the value of 11
```

The other type of `for` loop is the iterator `for` loop. The compiler does not handle these, but they can also be translated into a while-based construct such as the counter-based type.

Function definitions

Function definitions are also considered expressions in Lua. This means that definitions can be assigned to variables, and also returned from functions.

The compiler creates a `IRFunc` for each function definition it encounters. The `parent_block` and `parent_func` attributes are used to denote the parent function and basic block in which the definition was compiled in. This piece of information is important for upvalues generation, which is described in chapter 5.

Listing 4.25: A simple function with two return values.

```
local function f(a, b)
  return a + b, a - b
end
```

Function `f` from listing 4.25 is a local function, which takes two parameters, and returns two values. The compiler places the arguments of a function into the first N registers, where N is the number of parameters. In this case, `a` is stored in register 0, and `b` in register 1. In this case, `f` returns two values. The VM expects return values to be placed on the stack when the function returns. The compiler generates IR show in listing 4.26 for all values that are returned, and pushes them to the stack. The second operand of the `Push` instruction is a special operand which is described in chapter 5. The `Ret` instruction marks the end of a function call, and instructs the VM to continue the execution of the caller (the parent function). The `parent_func` attribute of `IRFunction2` is set to `1`, and `parent_block` also to `1`. This is because the definition of `f` is compiled in `BasicBlock1` of `IRFunction1`. The `Closure` instruction is used to create a `closure` value in the VM out of a function index. The `closure` is stored in register 1, and its associated function is `IRFunction2`. The parameters of a function are always considered local variables.

Listing 4.26: The SSA IR of listing 4.25.

```
IRFunction1 {
  BasicBlock1 {
    Mov Reg(0), Nil
    Closure Reg(1), Func(2)
    Phi Reg(0), Reg(1)
  }
}
IRFunction2 {
  # locals = { 'a' -> [0], 'b' -> [1] }
  BasicBlock1 {
    Add Reg(2), Reg(0), Reg(1)
    Push Reg(2), Some(1)
    Sub Reg(3), Reg(0), Reg(1)
    Push Reg(3), Some(1)
    Ret
  }
}
```

Function calls

In Lua, functions can be called in a similar way as it is done in other programming languages.

Listing 4.27: Calling a simple function with two return values.

```
local function f(a, b)
  return a + b, a - b
end
local x, y = f(1, 2)
```

Listing 4.28: The SSA IR of listing 4.27.

```
1  BasicBlock1 {
2    Mov Reg(0), Nil # local f = nil
3    Closure Reg(1), Func(1) # f = function(a,b) ...
4    SetTop Reg(1)
5    Mov Reg(2), Int(1)
6    Push Reg(2)
7    Mov Reg(3), Int(2)
8    Push Reg(3)
9    Call Reg(1) # f(1, 2)
10   MovR Reg(4), StackVal(0) # copy first return value from the stack into register 4
11   MovR Reg(5), StackVal(1)
12   Phi Reg(0), Reg(1)
13 }
```

For the sake of brevity, listing 4.28 only shows the basic blocks of the module. The code for function `f` can be found in listing 4.26. When the compiler translates a function call, first it compiles each argument of the call (lines 5, 7) and pushes them to the stack (lines 6, 8), then it emits a `Call` instruction to the function, and copies the return values of the call (lines 10-11). The compiler performs an unpacking operation on lines 10-11, which means that the values returned by `f` are copied into `x` and `y`. The `SetTop` instruction from

line 4 marks the start of a function call. This instruction is used to solve the issue of 'nested' stack frames. However, the details of this issue, and about unpacking are described in chapter 5.

4.2.4 Bytecode generation

Once the compiler generates the IR, it moves to bytecode generation. This is done through the use of the `BcGen` structure, which generates bytecode from Lua IR.

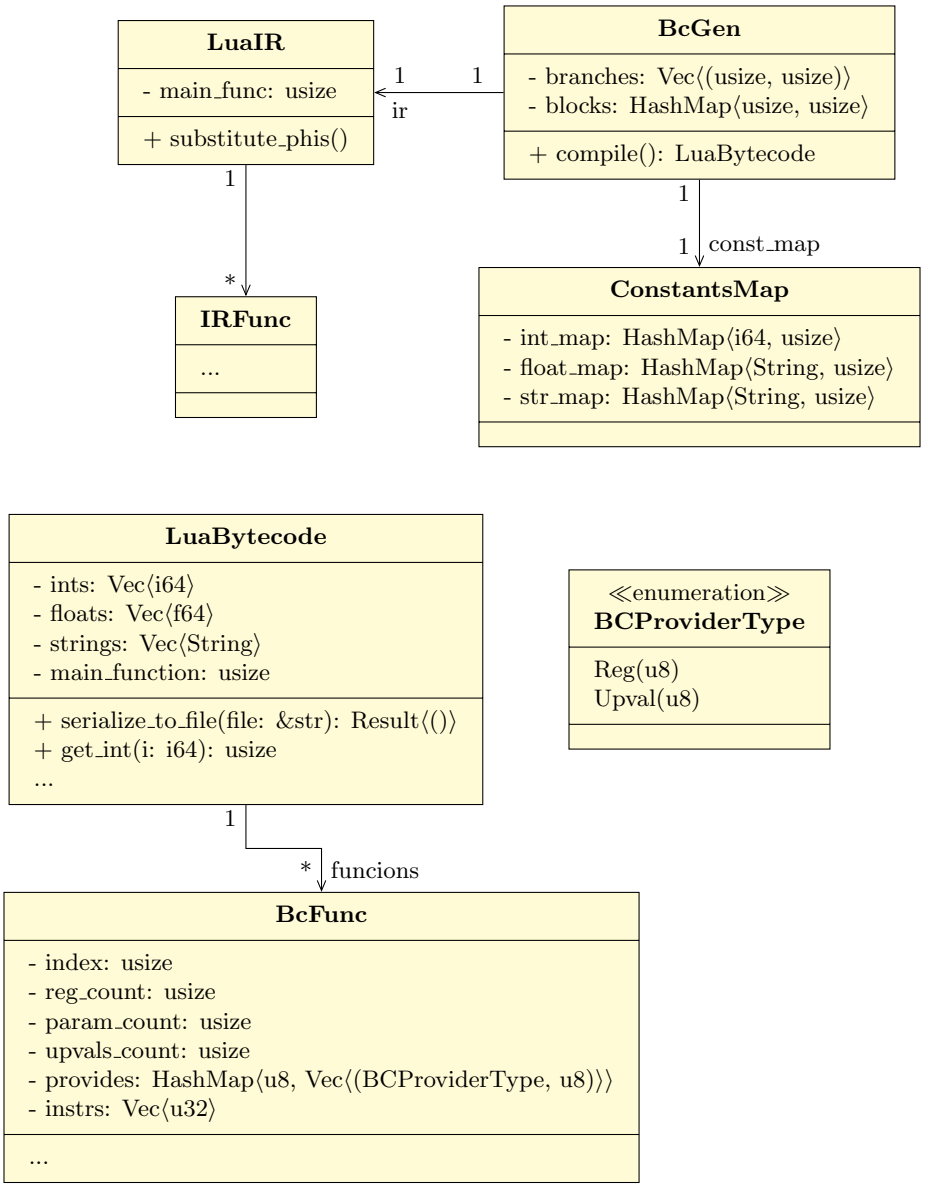

Figure 4.4: UML diagram showing the structure of the *bytecode generator* component.

The `ConstantsMap` entity, from figure 4.4 is a collection of all the constant values that are found in a source file. The `BcGen` structure compiles each `IRFunc` into a `BcFunc`. `BcFunc`s are the bytecode representation of `IRFunc`s. The other entities will be described in more detail in chapter 5.

4.2.5 Bytecode format

The instruction set of the bytecode produced by the compiler is different than the one produced by the PUC-Rio Lua compiler.

The format of the bytecode can be thought of as a compact representation of the IR, although bytecode instructions do not follow SSA rules. The `ints`, `floats`, and `strings` attributes are extracted from the `ConstantsMap`. Their main use will be highlighted in chapter 5. The `main_function` attribute represents the function which denotes the entry point of the program.

The `instrs` attribute of `BcFunc` represents all the instructions of the `BcFunc`. Note that each instruction is a `u32`, which means that an instruction is 32 bits long. The other attributes of `BcFunc` are described in chapter 5.

Instruction encoding

This encoding was inspired by the PUC-Rio Lua implementation [14]. An instruction has 4 main parts: the opcode, and 3 operands. Figure 4.5 shows a visual representation of an instruction.

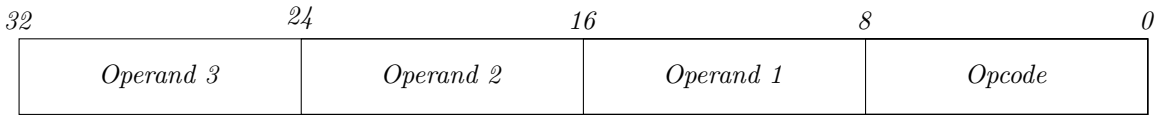

Figure 4.5: The format of a bytecode instruction.

Each operand of an instruction is 8 bits long, which means there are certain restrictions which the VM imposes. For instance, if one of the operands is a register number, then it means that the operand can encode at most 256 registers. The compiler must translate programs which can have an unlimited number of variables into a representation which can only address 256 'variables'.

4.3 The virtual machine

The virtual machine is the last component of the Lua interpreter. It takes as input the bytecode produced by the compiler component, and interprets its instructions.

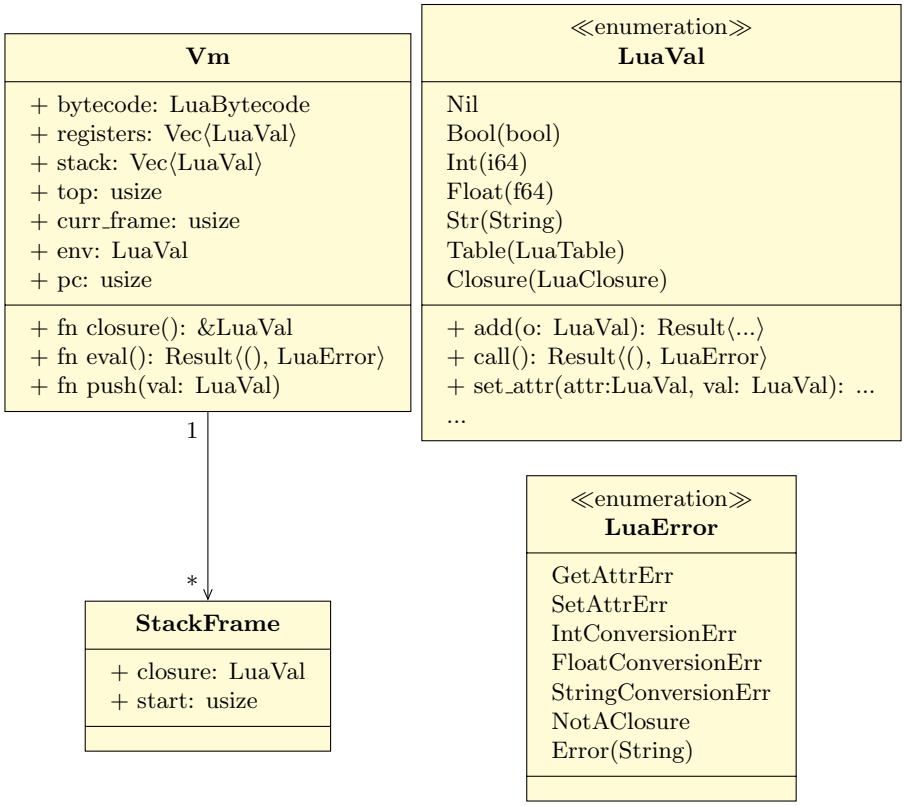

Figure 4.6: UML diagram of the virtual machine component.

The `Vm` entity, from figure 4.6 is the main entry point of the interpreter. When it is created, the VM initialises its environment and attributes in order to be able to execute the instructions of the bytecode. The `eval` function starts the interpreter. Other methods, such as `closure` and `push` are solely used for convenience.

The `LuaVal` entity represents the type system of the virtual machine. Note that this entity is an abstract representation of the type system. The type system defines many operations between `LuaVal`s. However, for the sake of brevity, not all of them are included in the diagram. If at any point, there is a mismatch between the expected type of a `LuaVal`, and its actual type, then the system raises a `LuaError`.

The `LuaError` entity models the different types of errors that can occur during interpretation. The `Vm` checks if each instruction executes successfully. If at any point an error occurs, the `Vm` stops the execution of the bytecode, and reports the error to the user.

The `StackFrame` entity models the call-stack of the virtual machine. The details of this entity will be described in more detail in chapter 5.

Chapter 5

Implementation

This chapter describes in detail how the design from chapter 4 is implemented in Rust. Section 5.1 describes the implementation of the Lua compiler crate[1], and section 5.2 the details of the virtual machine.

5.1 The `luacompiler` crate

The `luacompiler` crate is a library which can generate Lua bytecode[2] from a Lua source file.

As described in chapter 4, the first stage of compilation involves lexing and parsing the input file. In this project, the lexer is generated using the `lrlex`[3] crate, and the parser using `lrpar`[4]. The compiler uses the aforementioned crates in order to generate an efficient parser for the Lua language. The implementation of the `LuaParseTree` structure (figure 4.3) is straightforward. The `new` method loads the source file into memory, and generates a parse tree using the `lrlex` and `lrpar` crates. Once the AST is created, its root node is saved into `LuaParseTree` along with the contents of the source file.

5.1.1 The `irgen` module

This module facilitates the creation of the Lua IR. The `compile_to_ir` function is the main entry point of this module. It takes a `LuaParseTree` as input, and returns a `LuaIR` structure. This function creates a new `IRGen` compiler, which receives as input the parse tree. As soon as the compilation is done, the function retrieves the `IR` from the compiler, and returns it to the caller.

The `IRGen` compiler has a simple design. Each of its methods generates code for a symbol from the grammar. For example, `compile_funcbody` is called on non-terminals which represent a function definition.

When a new `IRGen` structure is created, it immediately creates an `IRFunc` for the module. This can be seen in listing 5.1. The module does not take any parameters, and is not a vararg function, thus `(0, false)` are passed as arguments to the `new` function of `IRFunc`.

The compiler defines a few useful methods such as `curr_func` and `curr_block` which return a reference to the function, or block, that is being compiled. Whenever the compiler emits instructions, it adds them to the current basic block of the current function. In order to emit instructions in a different block, the compiler must change its `curr_block` attribute to another value. When compiling branching statements, this attribute is changed in order for the compiler to emit instructions in the correct blocks.

Indices instead of references

The `parents`, `dominators`, `functions` attributes all use indices instead of references. This is because Rust imposes a few restrictions on references, which are hard to meet in the current design. For example a `BasicBlock` knows the blocks that dominate it, but cannot access their internals, because the structure only

[1] A `crate` is a Rust project.
[2] Incompatible with PUC-Rio Lua bytecode.
[3] `https://crates.io/crates/lrlex`
[4] `https://crates.io/crates/lrpar`

Listing 5.1: `IRGen::new`, and `IRFunc::new`.

```
impl IRFunc {
    pub fn new(param_count: usize, is_vararg: bool) -> IRFunc {
        IRFunc {
            parent_func: None, parent_block: None,
            upvals: BTreeMap::new(), provides: HashMap::new(),
            reg_count: 0, param_count,
            basic_blocks: vec![], is_vararg,
        }
    }
    ...
}

impl IRGen {
    fn new(pt: &LuaParseTree) -> IRGen {
        let functions = vec![IRFunc::new(0, false)];
        IRGen {
            pt, functions, curr_func: 0, curr_block: 0,
        }
    }
    ...
}
```

stores the index of the dominators, and not a reference to them. The compiler has access to all blocks, and can look up a basic block in constant time using an index, just like with references.

The `compile_expr` method

This method is used to compile an expression in Lua. This is the most important method as it is used by nearly every other method. It takes as input a node which represents an expression, and returns the number of the virtual register in which the result is stored.

Literals, such as numbers, or strings are considered expressions in Lua. Listing 5.2 shows the details of how literals are compiled into Lua IR. All literals are compiled as follows: the compiler determines the type of the literal, generates a new register in the current function with `get_new_reg()`, and it creates a `Mov` instruction, which copies the literal into the newly generated register.

Unary and binary expressions are compiled as described in chapter 4, and are not explained in more detail as their implementation is straightforward.

Expression can also contain variables. This case is handled by the `find_name` method, which is shown in listing 5.3.

There are three types of symbols: locals, upvalues, and globals. The `find_name` method first checks to see if `name` is a local variable. This is done by calling the `get_reg` method (line 3), which optionally returns the register which contains the current value of a symbol. Listing 5.4 shows the definition of the `get_reg` method.

The `get_reg` algorithm first checks if the symbol is defined in the current block (line 22). If it is, then the register of the symbol is returned (line 24). In SSA, in order to find a local variable, it is necessary to traverse the dominator tree upwards towards the root. If the variable is not found in the current block, then the algorithm recursively searches the dominators of the first block in order to find the symbol (lines 27-32).

If `get_reg` returns `None`, then the variable must be either an upvalue, or a global symbol. An upvalue is a local variable that is not defined in the current function, but in an enclosing function. This search is done by the `get_upval` method of `IRGen`.

The `get_upval` method, shown in listing 5.5 returns the index of the symbol in the `upvalues` attribute of a `IRFunc`. The upvalue algorithm first checks to see if the current function already defines the symbol as an upvalue (lines 4-6). If the current function does not define it, then the algorithm moves to the parent function

Listing 5.2: A snippet from `compile_expr` for literals.

```
impl IRGen {
    fn compile_expr(&mut self, node: &Node<u8>) -> usize {
        // what type of node is it?
        match *node {
            ...
            Term { lexeme } => {
                // get the value of the node: "x", "1", etc.
                let value = self.pt
                    .get_string(lexeme.start(), lexeme.end().unwrap_or(lexeme.start()));
                // what type of terminal is it?
                match lexeme.tok_id() {
                    lua5_3_l::T_NUMERAL => {
                        let new_reg = self.curr_func().get_new_reg();
                        let arg = if value.contains(".") {
                            Arg::Float(value.parse().unwrap())
                        } else {
                            Arg::Int(value.parse().unwrap())
                        };
                        self.instrs().push(Instr::TwoArg(Mov, Arg::Reg(new_reg), arg));
                        new_reg
                    }
                    lua5_3_l::T_SHORT_STR => {
                        let new_reg = self.curr_func().get_new_reg();
                        self.instrs().push(Instr::TwoArg(Mov, Arg::Reg(new_reg),
                            Arg::Str(value[1..(value.len() - 1)].to_string()),
                        ));
                        new_reg
                    }
                    lua5_3_l::T_FALSE => {
                        let new_reg = self.curr_func().get_new_reg();
                        self.instrs().push(Instr::TwoArg(Mov, Arg::Reg(new_reg), Arg::Bool(false)));
                        new_reg
                    }
                    ...
                }
            }
        }
    }
}
```

Listing 5.3: The `find_name` method.

```
impl IRGen {
    fn find_name(&mut self, name: &str) -> usize {
        match self.get_reg(name) {
            // does the current function define a mapping for 'name'?
            Some(reg) => reg,
            // check to see if any parent functions or blocks contain this variable
            None => {
                let reg = self.curr_func().get_new_reg();
                if let Some(upval_idx) = self.get_upval(name) {
                    self.instrs().push(Instr::TwoArg(GetUpVal,
                        // + 1 because _ENV is the first upvalue
                        Arg::Reg(reg), Arg::Upval(upval_idx + 1)))
                } else {
                    self.instrs().push(Instr::ThreeArg(GetUpAttr,
                        Arg::Reg(reg), Arg::Upval(0), // Upval 0 is _ENV
                        Arg::Str(name.to_string())));
                }
                reg
            }
        }
    }
}
```

(line 7). Note that the search in the parent function starts from the block in which the current function is defined, i.e. `parent_block`. The while loop (lines 10-26) tries to find the upvalue in all of the ancestors of the current function. If the symbol is not found, then the algorithm returns `None`. When the algorithm finds the upvalue in a parent function, `P`, it marks the symbol as an upvalue for the current function (line 13), and pushes a provider to `P`. On lines 14-18, the `reg` register from function `P` is marked as the value that is passed as upvalue with index `upvalue_idx` to the current function.

In listing 5.3, if `get_upval` returns an index, then a `GetUpVal` instruction is emitted in order to load that particular upvalue into a register. The register in which the upvalue is loaded is returned to the caller of `find_name`.

If the symbol is not an upvalue, then the compiler emits a `GetUpAttr` instruction in order to load the symbol from `_ENV`, which is always the first upvalue of any function.

The `compile_funcbody` method

Function definitions are also considered expressions in Lua. When `compile_expr` needs to compile a function definition, it calls the `compile_funcbody` method.

Listing 5.6 shows how the compiler translates function definitions. First, the algorithm creates a new `IRFunc` in the current block (lines 4-7), and populates the new function with a new empty basic block (line 9). As described in chapter 4, the parameters of a function are placed in the first N registers. The `compile_param_list` method (line 11) loops through each parameter of the function, assigns a register to it, and declares it as `local` in the first basic block of the new function. After that, on line 12, the body of the function is compiled. Once the compilation finishes, the compiler restarts the compilation process of the function in which the definition was found. Last, but not least, the algorithm creates a new register, and creates a `Closure` out of the index of the child function. The `Closure` is placed into the newly generated register.

Before returning the closure, the algorithm performs an upvalue propagation. Consider the Lua code from listing 5.7.

It demonstrates an important edge case of upvalues. The only upvalue of `g` is `a`[5]. Function `f` does not

[5]Not considering `_ENV` for simplicity.

Listing 5.4: The `get_reg` method.

```
impl BasicBlock {
    pub fn get_reg(&self, name: &'a str) -> Option<usize> {
        // check to see if `name` is a local variable
        let res = self.locals.get(name);
        if res.is_some() {
            return res.map(|v| *v.last().unwrap());
        }
        // check to see if the block modified `name` or not
        self.non_locals.get(name).map(|v| *v.last().unwrap())
    }
}

impl IRGen {
    fn get_reg(&self, name: &str) -> Option<usize> {
        self.get_reg_from_block(name, self.curr_func, self.curr_block)
    }

    fn get_reg_from_block(&self, name: &str, func: usize, bb: usize) -> Option<usize> {
        let curr_func = &self.functions[func];
        let curr_block = curr_func.get_block(bb);
        // is `name` defined in the current block?
        let res = curr_block.get_reg(name);
        if res.is_some() {
            return res;
        }
        // if `name` wasn't found, then try to find it in the dominators
        for &d in curr_block.dominators() {
            let res = self.get_reg_from_block(name, func, d);
            if res.is_some() {
                return res;
            }
        }
        // this function does not have a register for `name`
        None
    }
}
```

Listing 5.5: The `get_upval` method.

```
impl IRGen {
    fn get_upval(&mut self, name: &str) -> Option<usize> {
        // does the current function define the upvalue already?
        if let Some(&upval_idx) = self.curr_func().upvals().get(name) {
            return Some(upval_idx);
        }
        let mut func = self.curr_func().parent_func();
        let mut block = self.curr_func().parent_block();
        // go through all the parents, and check if we can find <name>
        while func.is_some() {
            // found <name>, so we can create a new upvalue
            if let Some(reg) = self.get_reg_from_block(name, func.unwrap(), block.unwrap()) {
                let upval_idx = self.curr_func().push_upval(name);
                // func provides an upvalue for curr_func
                self.functions[func.unwrap()].push_provider(
                    self.curr_func,
                    upval_idx + 1, // which upvalue is provided
                    ProviderType::Reg(reg),
                );
                return Some(upval_idx);
            } else {
                // go to the parent function
                let p_func = &self.functions[func.unwrap()];
                block = p_func.parent_block();
                func = p_func.parent_func();
            }
        }
        None
    }
}
```

Listing 5.6: The `compile_funcbody` method.

```
impl IRGen {
    fn compile_funcbody(&mut self, nodes: &Vec<Node<u8>>) -> usize {
        ... // elided for clarity
        let mut new_func = IRFunc::new(0, is_vararg);
        new_func.set_parent_func(self.curr_func);
        new_func.set_parent_block(self.curr_block);
        self.functions.push(new_func);
        self.curr_func = new_func_id;
        let new_basic_block = self.curr_func().create_block();
        self.curr_block = new_basic_block;
        self.compile_param_list(&nodes[1]);
        self.compile_block_in_basic_block(&nodes[3], new_basic_block);
        self.curr_func = old_curr_func;
        self.curr_block = old_curr_block;
        let reg = self.curr_func().get_new_reg();
        self.instrs().push(Instr::TwoArg(
            Closure,
            Arg::Reg(reg),
            Arg::Func(new_func_id),
        ));
        ... // elided for clarity
        reg
    }
}
```

Listing 5.7: A Lua program which demonstrates upvalue propagation.

```
local a = 1
function f()
  function g()
    return a
  end
  g()
end
```

use `a`, however, it has `a` as an upvalue. This is because `g` uses `a` which is defined in the module, but `g` is a child of `f`. Thus, `f` needs to provide `a` as an upvalue to `g`, if `g` is called from `f`. This means that the upvalue of `g` is 'propagated' to all the ancestors that do not define `a` as a local variable. As such, function `f` provides its `a` upvalue as an upvalue to `g`. This is where the `ProviderType::Upvalue` variant is used. The exact implementation of this feature is not shown for the sake of brevity, and can be found in the appendix.

The `compile_call` method

Function calls can return values, as such they can be used in expressions. The `compile_expr` method returns a register in which the result of an expression is stored. In the case of function calls, the method invokes `compile_call` in order to generate a `Call` instruction. A `Call` instruction does not return a value, it simply instructs the VM to call a given function. Because `compile_expr` must return the result of an expression, it emits a `MovR` instruction. The `MovR` instruction emitted instructs the VM to copy the first return value of the function into a newly generated register. The `compile_expr` method returns this new register, as it stores the result of the call. The `compile_call` method works very similarly to the algorithm described in section 4.2.3.

The `unpack_to_stack` method

Function calls can return multiple values, and in certain circumstances, it is important to copy all the returned values from the stack, not only the first one.

Listing 5.8: A Lua program which demonstrates stack unpacking.

```
1  function f()
2    return 1, 2, 3
3  end
4  function g(a, b, c)
5    print(a, b, c)
6  end
7  g(f()) -- prints 1, 2, 3
8  g(f(), f(), f()) -- print 1, 1, 1
```

In listing 5.8 the interpreter performs an unpacking operation on line 7. This means that all values returned by `f()` are passed as arguments to `g`. In Lua, only the last expression of an expression list must be unpacked. There are two types of expressions that need to be unpacked: function calls, and variable arguments (`varargs`). One of the biggest problems is that the compiler does not know statically how many return values a function has.

Listing 5.9: A function which returns a different number of values.

```
1  function f()
2    if some_condition then
3      return 1, 2, 3
4    else
5      return 1
6    end
7  end
```

In listing 5.9, `f` might return three values, or only one. The exact number of returned values is known at run-time, as such, the compiler emits a special instruction which instructs the VM to unpack *all* return values.

The `unpack_to_stack` method from listing 5.10 is used to unpack an expression to the stack. The compiler uses this method when compiling function calls, or return statements. As described in an earlier section, the `MovR` instruction is used to copy a return value of a function from the stack into a register. However, the `MovR` instruction can also be used to push return values back to the stack. If the last operand of `MovR` is `1`, or `2`, then the VM pushes *all* return values of the previously called function to the stack. The details

Listing 5.10: The `unpack_to_stack` method.

```
impl IRGen {
    fn unpack_to_stack(&mut self, last_expr: &Node<u8>, increment_ret_vals: bool) {
        let reg = self.compile_expr(last_expr);
        if self.is_unpackable(last_expr) {
            {
                let len = self.curr_block().instrs().len();
                // this is either a VarArg instr, or a MovR
                let last_instr = self.curr_block().get_mut(len - 1);
                debug_assert!(last_instr.opcode() == MovR || last_instr.opcode() == VarArg);
                *last_instr = Instr::OneArg(
                    last_instr.opcode(),
                    Arg::Some(1 + increment_ret_vals as usize),
                );
            }
            // compile_expr generates (VarArg/MovR <reg> <op2> <op3>)
            // but because we are modifying the last instruction, there is
            // no need to keep the previously allocated register
            self.curr_func().pop_last_reg();
        } else {
            ... // emit a PUSH instruction for <reg>
        }
    }
}
```

of the `MovR` instruction are given in section 5.2.3. The algorithm of the `unpack_to_stack` method is rather straightforward. The implementation of `unpack_to_stack` first checks if the given expression is unpackable or not (line 4). In the case it is not, then the compiler simply pushes the register in which the result of the expression is stored to the stack (not shown in the listing for the sake of brevity). If the expression is unpackable, then it means that the expression is either a function call or a vararg expression. This section will only focus on the function call case, because the vararg case is very similar to it. The compiler knows that the last instruction is a `MovR` instruction (lines 6-9). Because the expression is being unpacked to the stack, the compiler modifies the operands of the `MovR` instruction (lines 10-13) in order to instruct the VM to unpack all return values instead of only the first one. The `increment_ret_vals` parameter only tells the compiler if the unpacking is done in a return statement, or in a function call. On line 12, the compiler sets the special argument of the `MovR` to either 1, in the case `increment_ret_vals` is false, or to 2 otherwise.

Method calls

In Lua, a method call is another type of function call. Methods are functions which take the table on which they are called on as an argument.

Listing 5.11: An example of a method.

```
1  local function g(self, a) print(self.i + a) end
2  x = {}
3  x.i = 1
4  x.g = g
5  x:g(2) -- prints 3
6  g(x, 2) -- also prints 3
```

Listing 5.11 shows an example of method calls. Lines 5 and 6 are equivalent, which means that compiling a method call is very similar to compiling a function call. The `compile_method` method of `IRGen` works similar to `compile_call` except that the compiler makes sure to push the table before the other arguments.

Listing 5.12: The `compile_assignment` method.

```
impl IRGen {
    fn compile_assignment(
        &mut self,
        var: VarType,
        right: &Node<u8>,
        action: AssignmentType,
    ) -> ResultType {
        match var {
            VarType::Name(name) => {
                let mut value = self.compile_expr(right);
                ...
                if action == AssignmentType::LocalDecl || self.get_reg(name).is_some() {
                    ...
                    self.set_reg_name(value, name, action == AssignmentType::LocalDecl);
                    ResultType::Local(value)
                } else {
                    if action != AssignmentType::Postponed {
                        self.set_upval(name, value);
                    }
                    ResultType::Global(value)
                }
            }
            VarType::Table(from, attr) => {
                let reg = self.compile_expr(right);
                if action != AssignmentType::Postponed {
                    self.instrs().push(Instr::ThreeArg(
                        SetAttr,
                        Arg::Reg(from),
                        Arg::Reg(attr),
                        Arg::Reg(reg),
                    ));
                }
                ResultType::Table(reg)
            }
        }
    }
}
```

The `compile_assignment` method

In Lua, there are two main types of assignments:

1. to a variable: `(local) name = expression`,
2. to a table element: `t[attr1][attr2] = expression`

The `compile_assignment` method from listing 5.12 takes three parameters, and returns a result. The first parameter is the left-hand-side of the assignment, also called lvalue. In the compiler, an lvalue is represented by a `VarType`. This enumeration represents the two assignment cases described above. The second parameter of `compile_assignment` is the expression to be stored in the variable, called rvalue, and the final argument is an `AssignmentType`. This enumeration instructs the compiler how to deal with a particular assignment.

If the lvalue is a `VarType::Name`, i.e. a variable name, then the compiler generates code for the rvalue, and checks if the symbol is a local variable or not (line 12). The `get_reg` method, which was described in an earlier section, returns the register from the current function, which contains the value of a given variable. If the assignment type is `LocalDecl`, then this means that the source program declares a new local variable,

thus there is no need to check if it is local or not. If the variable is a local one, then the compiler calls `set_reg_name` on the register in which the result of the rvalue is stored. Note that `compile_expr` generates a new register in which the result of the expression is stored. This register does not have an associated name, thus the compiler can assign a name to it, namely the name of the symbol. The `set_reg_name` method is described in the next section, for now it is important to know that this method assigns a name to a register. If the variable is not a local one, then it must be a global one, or an upvalue. In this case, the compiler calls the `set_upval` method. The `set_upval` algorithm first checks whether the given symbol is an upvalue. If it is, then the compiler emits a `SetUpVal` instruction. However, if the symbol is a global, the compiler emits a `SetUpAttr` instruction to store the symbol in `_ENV` (the first upvalue).

The `Postponed` variant of `AssignmentType` means that the compiler should not emit any instructions to store the rvalue into the lvalue. However, the caller must emit the store at some point after calling `compile_assignment`. In the case of a local variable, this method returns a `ResultType::Local`, letting the caller know that the expression has been stored in a variable local to the function. If, however, the symbol is a global value, then the compiler returns a `ResultType::Global`. If the assignment type is `Postponed`, and the result is a `Global`, then the caller must emit an instruction which stores the result into `_ENV`.

The lvalue can also be a table assignment. The `VarType::Table(from, attr)` variant means that the rvalue should be stored in the `from` table, at key `attr`. The caller can also postpone the store of the rvalue using `Postponed`. In this case, the compiler returns a `ResultType::Table`, meaning that the rvalue is stored in a table.

The `set_reg_name` method

This method calls `set_reg_name` on the current basic block. The method takes 3 parameters, the register, the new name of the register, and a boolean value (`is_local_decl`) which describes the type of variable: local or non-local. If the value is a local one, then the compiler creates a mapping between the name and the register in the `locals` attribute of the basic block. If the value is not a local one, then the `non-locals` attribute is updated instead.

Listing 5.13: Scoped assignments.

```
1  local a = 1
2  do
3    a = 2
4    a = 3
5    local a = 4
6    a = 5
7    local a = 6
8  end
```

Each assignment from listing 5.13 triggers a call to the `set_reg_name` method. The comments from listing 5.14 show how the `locals` and `non-locals` attributes of `BasicBlock2` are modified by the `set_reg_name` method. Each time this method is called with `is_local_decl` set to true, the block emits a `Phi` instruction. When `a` is declared on line 5 (line 8 in IR), the basic block emits a `Phi` which merges all the values stored in `non-locals` for the entry `'a'`. A similar instruction is emitted when the second local `'a'` is declared on line 7 of the Lua source file. However in this case the `Phi` merges the `locals` entry for `'a'` instead. The `Phi` instruction forces the registers it takes as an operand into the same VM register. In order to further illustrate why this is necessary consider the following example:

Because the compiler generates an SSA IR, the `a` declared on line 1 in listing 5.15 is placed into `reg1`, but the second `a` (line 4) is placed into `reg2`. However, `a` is an upvalue of `g`, but the compiler only records `reg1` as being an upvalue of `g`. This means that the compiler must associate the two registers with a `Phi` so that when register allocation is done, the two registers are allocated to the same bytecode register, in order to preserve the meaning of the original program.

Listing 5.14: The SSA IR of listing 5.13

```
# locals = { 'a': [0] }
BasicBlock1 {
  Mov Reg(0), Int(1)
}
BasicBlock2 {
  Mov Reg(1), Int(2) # non-locals = { 'a': [1] }
  Mov Reg(2), Int(3) # non-locals = { 'a': [1, 2] }
  Mov Reg(3), Int(4) # locals = { 'a': [3] }
  Phi Reg(1), Reg(2) # non-locals = { 'a': [1] }
  Mov Reg(4), Int(5) # locals = { 'a': [3, 4] }
  Mov, Reg(5), Int(6) # locals = { 'a': [5] }
  Phi Reg(3), Reg(4)
}
...
```

Listing 5.15: An example which shows why the extra `Phi` instructions are necessary.

```
local a = 1
local function g() print(a) end
g() -- prints 1
a = 2
g() -- prints 2!
```

The `compile_assignments` method

This method is used to compile a multi-assignment statement. There are two types of multi-assignments: local and non-local. The `compile_assignments` method takes three parameters, a list of lvalues $(l_0, l_1, ..., l_n)$, a list of rvalues $(r_0, r_1, ..., r_m)$, and a boolean value which denotes whether the assignment is local or not. The algorithm begins by creating assignments between $(l_0, ..., l_k)$ and $(r_0, ..., r_k)$ where $k = min(m, n)$. If the number of expressions is greater than the number of lvalues $(m > n)$, then the compiler must also evaluate $(r_{k+1}, ..., r_m)$. This is necessary as it is part of the language reference manual. In the other case $(n > m)$, the compiler needs to perform an unpacking operation. This means that the last expression r_m is unpacked into the lvalues $(l_m, ..., l_n)$. If the last expression is unpackable, then the unpacking is performed by the `unpack` method. This method takes two parameters: the registers in which the compiler must perform the unpacking, and the last expression. In the case of a function call, the compiler emits `MovR` instructions which copy return values one by one into the given registers. However, if the last expression is not unpackable, then the compiler must set the extra lvalues to nil.

If an lvalue is a table, a global symbol, or an upvalue, then the compiler will emit instructions (for example `SetAttr`, or `SetUpVal`) in order to store the rvalue into the lvalue. However, if the last expression is unpackable, the compiler will emit `MovR` instructions in order to perform the unpacking. The VM expects `MovR` instructions to be consecutive, which means that the compiler must postpone emitting the store instructions until after the sequence of `MovR`s.

Vararg functions

In Lua, a function takes a variable number of arguments if the last parameter is the `...` symbol. When the compiler encounters a function definition that has such a parameter, it sets the `is_vararg` attribute of the `IRFunc` to `true`.

All parameters of a function are stored on the stack, before the return values. The `VarArg` instruction is used to copy a vararg from the stack into a register. It works similarly to `MovR`, because both instructions copy a certain value from the stack into a specfied register. The main difference between the two is how they

are implemented in the VM. The `...` parameter can also be used in expressions. The compiler ensures that if this parameter is used in an expression, it is unpacked if necessary. The unpacking logic for varargs is the same as the unpacking logic for function calls, thus it will not be described for the sake of brevity.

On-the-fly dominator trees

If statements, for loops, while loops, etc. are implemented as described in chapter 4. However it is worth mentioning that the compiler also generates the dominator tree of a function while generating branching statements.

For instance, when the compiler encounters an if statement, it creates a dominator link between the block in which the if is encountered, and the block to which all branches jump to. A similar approach is taken when compiling loops and `do..end` blocks. Using such an approach results in a block only knowing its immediate dominators, not all of them. This is not a problem, as a dominator of an immediate dominator is also a dominator of the current block, and therefore can be easily computed.

5.1.2 The `bytecodegen` module

This module is used to generate bytecode from `LuaIR`, and its main entry point is the `compile_to_bytecode` function. This function creates a new `BcGen` compiler, and calls its `compile` method in order to generate bytecode.

The first operation that the bytecode compiler performs is a `Phi` substitution.

The `substitute_phis` method

This is a method defined in `LuaIR`. Its main purpose is to substitute all uses of registers based on `Phi` instructions. To illustrate this, consider the following example:

Listing 5.16: An example of a program which uses `Phi`s.

```
1  BasicBlock1 {
2    Mov Reg(0), Int(0)
3    Mov Reg(1), Int(1)
4    Mov Reg(2), Int(2)
5    Mov Reg(3), Int(3)
6    Add Reg(4) Reg(0) Reg(1)
7  }
8  BasicBlock2 {
9    Phi Reg(0), Reg(4)
10   Phi Reg(2), Reg(3)
11 }
```

The algorithm performs a `Phi` substitution on the IR from listing 5.16. This means that all uses of `Reg(4)` are substituted with `Reg(0)`, and `Reg(2)` with `Reg(3)`. These two substitution are performed due to the `Phi` instructions from lines 9 and 10 of listing 5.16. Listing 5.17 shows the IR after `Phi`s are removed.

The `Phi` substitution algorithm processes the IR in two passes. The first pass collects all substitutions, and the second pass performs the substitutions. Each `Phi` instruction is converted into a pair. The first element is the first operand of the `Phi`. It represents the register that the other registers will be substituted by. The second element of the pair is a list of the remaining operands of the `Phi` instruction. These registers are going to be replaced by the first register of the pair. Each of these pairs is saved into a map. The second pass iterates over each instruction of the current function and applies the substitutions gathered in the first pass. The compiler also makes these substitutions in the `provides` attribute of the current function, in order to ensure that the correct registers are passed as upvalues. Although this method destroys the SSA property of the IR, it is only called during bytecode generation.

The next stage of bytecode generation is the translation of the IR into bytecode. This stage is carried out by the bytecode compiler, represented by the `BcGen` structure. The following sections describe the attributes of the `BcGen` structure, as they are used extensively during code generation.

Listing 5.17: Listing 5.16 where `Phi` substitution was performed.

```
BasicBlock1 {
  Mov Reg(0), Int(0)
  Mov Reg(1), Int(1)
  Mov Reg(2), Int(2)
  Mov Reg(2), Int(3)
  Add Reg(0) Reg(0) Reg(1)
}
BasicBlock2 {
  Phi # consumed by substitute_phis
  Phi # consumed by substitute_phis
}
```

The `const_map` attribute

An important attribute of the bytecode compiler is `const_map`. The `ConstantsMap` structure creates three constant tables, one for ints, one for floats, and one for strings. Whenever the compiler encounters an argument which represents a constant, its value is mapped to its index in the constant table. The `get_int` method returns the index of an integer in the constant table. If the integer has not been seen yet, the map creates a new index for it, and returns it. The `get_str` and `get_float` methods work similarly to `get_int`. They are used to perform the same operation for constant strings, and floats respectively.

The bytecode instruction set provides a few extra opcodes in addition to those defined by the IR. For instance, `Ldi` is used to load an integer into a register. This second operand of `Ldi` is a number which represents the index of an integer in the constant table. The constant tables of `ConstantsMap` are also stored in the bytecode. This is necessary because bytecode instructions, such as `Ldi`, have two operands. The first operand represents the register in which an integer is loaded, and the second operand encodes the index of the integer in the constant table. The VM uses this index, and the constant table in order to get the actual value that is loaded into the register. The `Ldf`, and `Lds` instructions are used to copy values from the floats and strings constant tables.

The `blocks` attribute

The bytecode compiler generates bytecode for each basic block. The bytecode of a basic block consists of the bytecode instructions that resulted from compiling the IR instructions from the basic block. Each compiled basic block is appended to the final bytecode. The bytecode does not have the concept of basic blocks, as it only consists of bytecode instructions. However, the compiler can calculate where the instructions of a basic block begin in the bytecode. This is called the starting position of the block. The `blocks` attribute maps each `BasicBlock` to its starting position in the bytecode. This information is used later in the compilation process in order to patch jump instructions.

The `branches` attribute

This attribute represents a list of pairs, where the first element is the position of a jump instruction, and the second element the basic block index the instruction jumps to. The bytecode versions of `Jmp`, `JmpNe`, and `JmpEq` have a different meaning than their IR counterparts. Because the bytecode does not have basic blocks, jump instructions have an integer operand which represents the number of instructions that the VM needs to jump over. For instance an instruction such as `JmpNe Reg(0), Block(1), Block(2)` is converted to `JmpNe Reg(0), N`, where N is the number of instructions that the VM needs to skip in order to start the execution of the instructions which were part of `BasicBlock2`.

The lowering of the IR to bytecode is a straightforward process. Most IR instructions have a one-to-one mapping to bytecode instructions. Recall that bytecode instructions are 32-bits in size, and that each operand of the instruction has a size of 8 bits. This means that the compiler must make sure that, for instance, the

Listing 5.18: A snippet from the `compile_instr` method.

```
impl BcGen {
    fn compile_instr(&mut self, f: usize, bb: usize, i: usize, instrs: &mut Vec<u32>) {
        ...
        match opcode {
            ...
            Mov => {
                if let Instr::TwoArg(_, ref arg1, ref arg2) = instr {
                    let (opcode, arg2) = match *arg2 {
                        Arg::Reg(reg) => (Opcode::Mov, reg),
                        Arg::Int(i) => (Opcode::LDI, self.const_map.get_int(i)),
                        ...
                        Arg::Bool(b) => (Opcode::LDB, b as usize),
                        _ => panic!("Mov shouldn't have {:?} as an argument.", arg2),
                    };
                    instrs.push(make_instr(opcode, arg1.get_reg() as u8, arg2 as u8, 0))
                }
            }
            Jmp => {
                let len = instrs.len();
                instrs.push(if let Instr::OneArg(_, arg1) = instr {
                    self.branches.push((len, arg1.get_block()));
                    make_instr(opcode.to_opcode(), 0, 0, 0)
                } else {
                    panic!("Not enough arguments for {:?}!", opcode)
                })
            }
            ...
        }
    }
}
```

IR does not use registers whose number is larger than 255. There are, however, some instructions that do not have one-to-one mappings to bytecode instructions such as: `Mov`, `Jmp`, `JmpNe`, and `JmpEq`.

The compiler generates bytecode one function at a time. For each function, it generates the instructions of each basic block, and creates a `BcFunc` structure which represents a function in the bytecode.

Listing 5.18 shows how `Mov` and `Jmp` are lowered to bytecode. The `Mov` instruction is used with many different arguments. In listing 5.18, `arg2` represents the value that is loaded into the register stored in `arg1`. For example, if the second argument is an `Arg::Int`, the compiler creates an `Ldi` instruction, and converts the number into an index using the `get_int` method of `ConstantsMap`. Similarly, the bytecode provides `Ldt` for tables, `Ldb` for bools, etc. The `make_instr` function from line 18 takes four `u8` arguments and generates a 32 bit instruction which the VM can decode.

Jump instructions are also handled differently. Line 23 creates a jump instruction, and adds it to the bytecode instructions of the function. On line 24, the compiler creates an entry in the `branches` attribute, where the first element of the pair is `len`, i.e. the position of the jump instruction in the bytecode, and the second element is `arg1.get_block()`, which is the index of the `BasicBlock` to which the VM jumps. On line 25, the compiler creates an empty jump instruction which is later patched by the compiler. Once the instructions of the IR are all converted into bytecode instructions, the compiler loops through all the `branches`, and changes the operands of jump instructions based on how far the instructions are from the basic blocks they jump to. This process uses the `blocks` attribute to find out where each basic block starts in the bytecode.

Listing 5.19 contains the IR of a program that uses jump instructions. Listing 5.20 shows the bytecode produced by the `BcGen` compiler from this IR. As described above, most IR instructions have a corresponding bytecode instruction. For example, the `Gt` instruction which checks if an expression is greater than the value

Listing 5.19: IR

```
1  BasicBlock1 {
2    Mov Reg(0), Int(2)
3    Mov Reg(1), Int(2)
4    Gt Reg(2), Reg(0), Reg(1)
5    JmpNe Reg(2), Block(2), Block(3)
6  }
7  BasicBlock2 {
8    Mov Reg(3), Int(3)
9    Jmp Block(3)
10 }
11 BasicBlock3 {
12   Phi Reg(0), Reg(3)
13 }
```

Listing 5.20: Bytecode

```
Ldi 0 0
Ldi 1 0
Gt 2 0 1
JmpNe 2 2 0

Ldi 0 1
Jmp 0 0 0
```

of another expression has the same semantics both in the IR and in the bytecode. The first `Mov` instruction is converted into an `Ldi` because the second argument of the IR instruction is an `Int`. `Ldi 0 0` means 'load the integer from index 0 of the integer constant table, and store it into register 0'. The number `2` is associated to index `0` because it is the first constant integer which is processed by the compiler. The `Mov` instruction from line 3 also takes `2` as an argument, therefore the `Ldi` instruction will also load the first integer from the constant table. The bytecode version of the `JmpNe` instruction is different than its IR counterpart. The bytecode instruction reads: 'if the value in register 2 is not true, then jump over the next two instructions'. Moreover, the `blocks` attribute has the following entries: {`1 -> 0, 2 -> 4, 3 -> 6`}. As seen in the figure, `BasicBlock1` starts at the first instruction of the bytecode, then `BasicBlock2` at the fifth instruction, etc. The `branches` attribute contains two entries: `[(3, 3), (5, 3)]`. The first entry means that the fourth instruction of the bytecode jumps to the third basic block, and the second entry means that the sixth instruction can jump to the third basic block.

The second operand of the `JmpNe` instruction represents the number of instructions the VM needs to jump over if the first operand is false. This is calculated by subtracting the position of the instruction, `3`, from the index of the destination block in the bytecode (`blocks[3]` - 1 = 5). Thus, `5 - 3 = 2`, becomes the second operand of `JmpNe` during the patching process.

5.2 The `luavm` crate

This crate represents the virtual machine, which takes as input Lua bytecode and executes it. The virtual machine has three main components: the VM itself, the value system, and the opcode implementations.

5.2.1 The `LuaVal` system

The value system of the VM is implemented by the `lua_val` module. The VM supports bools, integers, floats, strings, tables, and closures. Closures are represented by a bytecode function and its upvalues.

The `LuaVal` structure represents a tagged pointer, where the last 3 bits of the `val` attribute indicate the type of the value.

The `val` attribute, shown in listing 5.21, is a 64-bit unsigned integer which stores two pieces of information:

1. The first 63 bits store the actual data (an int, float, etc.), or a pointer to where the data is located
2. The lowest-order 3 bits store a `LuaValKind`, which is used at run-time to determine the type of data that is stored in the first 63 bits

Pointers on 64-bit machines are always 8-bytes aligned. This means that the last 3 bits of a pointer are always set to 0. The last 3 bits can be used to store information. In this case, the last 3 bits will be used to encode one of the variants of `LuaValKind`. Using 3 bits, it is possible to encode $2^3 = 8$ different values.

Listing 5.21: The `LuaVal` and `LuaValKind` types

```
pub enum LuaValKind {
    BOXED = 0,
    INT = 1,
    FLOAT = 2,
    Gc = 3,
    GcRoot = 4,
    BOOL = 5,
    NIL = 6,
}

pub struct LuaVal {
    val: Cell<usize>,
}
```

`LuaValKind` has 6 variants, which means that the integer representation of a variant can be safely stored in the last 3 bits of a pointer.

`LuaVal`s implement various types of methods such as arithmetic operations, relational operations, and table look-ups shown in listing 5.22. For instance the `add` method adds a `LuaVal` to another `LuaVal`. The Lua reference specifies that if any of the operands of an arithmetic operation is a float, then both values must be converted to floats, before performing the arithmetic operation. The implementation of the VM follows the language reference as shown on line 12: the system checks if any of the operands is a float, and on line 13 both operands are converted to float and summed. If at any point a conversion fails, the `?` operator ensures that the method returns a `LuaError`, which is propagated to the VM.

`LuaVal`s also implement methods which allow base types such as `i64` or `f64` to be converted into `LuaVal`s.

The `INT`, and `FLOAT` tags

These tags are used to represent 'raw' integers and floats. If any of these tags are used, then the first 63 bits of a `LuaVal` represent an integer or a float, and not a pointer. The explanations that follow will use the `INT` tag as an example, but the `FLOAT` tag is implemented similarly to `INT`.

Because the `LuaVal` can only store 63 bits of data, this means that the encoded integer must not have a value that uses more than 63 bits. The value system handles two types of integers:

1. If none of the 3 highest-order bits of the integer are set, then it means that the value can be represented with 63 bits. In this case, the value system shifts the integer 3 places to the left, setting the 3 lowest-order bits to 0. This means that the system can safely encode the `INT` tag into the 3 lowest-order bits.

2. If any of 3 highest-order bits of the integer are set, then the system cannot encode the tag without losing information. In this case, the integer is converted into a `LuaObj`. The system creates a pointer to this `LuaObj`, and stores its address in the 63 bits of data of a `LuaVal`.

Floats are handled very similarly to integers, and will not be described for the sake of brevity.

The `LuaObj` trait

Traits in Rust are similar to Java interfaces. Rust provides trait objects, which are used to achieve dynamic typing. For example `Box<LuaObj>` is a trait object. A `Box` in Rust is a pointer to a type that is allocated on the heap. `Box<LuaObj>` represents a pointer to an object that implements the `LuaObj` trait. Trait objects are also called 'fat pointers' in Rust, because they store two different pointers: a pointer to the data (`LuaInt`, `LuaFloat`, etc.) and a pointer to the virtual functions table. This means that the size of a `Box<LuaObj>` is not 64 bits, but 128 bits, and that it is not possible to store a `Box<LuaObj>` into `LuaVal`. A way around this

Listing 5.22: Some of the methods of `LuaVal`

```
impl LuaVal {
    ...
    pub fn to_float(&self) -> Result<f64, LuaError> {
        ...
    }
    ...
    pub fn set_attr(&self, attr: LuaVal, val: LuaVal) -> Result<(), LuaError> {
        ...
    }

    pub fn add(&self, other: &LuaVal) -> Result<LuaVal, LuaError> {
        Ok(if self.is_aop_float() || other.is_aop_float() {
            LuaVal::from(self.to_float()? + other.to_float()?)
        } else {
            LuaVal::from(self.to_int()? + other.to_int()?)
        })
    }
    ...
    pub fn set_upval(&self, i: usize, value: LuaVal) -> Result<(), LuaError> {
        ...
    }
}
```

problem is to create a `Box<Box<LuaObj>`, i.e. a pointer to a trait object. This means that in order to get the underlying `LuaInt` for instance, the value system must follow two pointers instead of one.

The `LuaObj` trait defines all the operations that can be performed on integers, floats, and strings. The `LuaInt`, `LuaFloat`, and `LuaString` structures are wrappers around the `i64`, `f64`, and `String` types of Rust, and implement this trait.

When the value system creates a LuaString, or LuaInt (which stores an integer whose value does not fit in 63 bits), it allocates a `Box<Box<LuaObj>>` and stores it in the `val` attribute together with the `LuaValKind::BOXED` tag.

The `UserTable` type

This type implements the `table` type of Lua. This structure contains one field, `v`, which is a `HashMap` from `LuaVal`s to `LuaVal`s. `LuaVal`s can be compared, and hashed, thus they can be used with Rust's `HashMap` type.

The `UserBcFunc` type

`UserBcFunc`s represent functions that are defined in the bytecode. The attributes of this structure include:

1. `upvals` - the upvalues of the function, represented as a list of `LuaVal`s.
2. `reg_count` - the number of virtual registers that the function uses. This information is extracted from the bytecode.
3. `param_count` - the number of formal parameters of the function. This information is also extracted from the bytecode.
4. `ret_vals` - the number of values pushed to the stack by the function as return values.
5. `index` - the index of the `BcFunc` from the bytecode to which the VM 'jumps' when the `UserBcFunc` is called.

Listing 5.23: `Vm::new` method

```
impl Vm {
    pub fn new(bytecode: LuaBytecode, script_args: Vec<&str>) -> Vm {
        let mut registers: Vec<LuaVal> = Vec::new();
        registers.resize(REG_NUM, LuaVal::new());
        let mut env = LuaVal::from(UserTable::new(HashMap::new()));
        Vm::init_stdlib_and_args(&script_args, &mut env);
        let closure = {
            let index = bytecode.get_main_function();
            let main = bytecode.get_function(index);
            let func = UserBcFunc::with_upvals(index, 0, main.param_count(), vec![env.clone()]);
            LuaVal::from(func)
        };
        let mut stack_frames = Vec::with_capacity(255);
        stack_frames.push(StackFrame { closure, start: 0 });
        Vm {
            bytecode, registers, stack: vec![], top: 0,
            stack_frames, curr_frame: 0, env, pc: 0,
        }
    }
}
```

The `BuiltinBcFunc` type

`BuiltinBcFunc`s are pointers to functions defined in the VM. For instance `print` is a builtin function, the implementation of which is provided by the VM. This structure has two attributes:

1. `handler` - a pointer to a function from the VM which implements a particular operation. When the VM calls a `BuiltinBcFunc`, it extracts the `handler` pointer and invokes it.
2. `ret_vals` - the number of values pushed to the stack by the function as return values

The `GcVal` trait

In the `LuaVal` system, `closures` (`UserBcFunc`, `BuiltinBcFunc`), and `tables` are garbage collected. These structures implement a trait called `GcVal`, which defines operations such as: `set_upval`, `set_attr`, etc. For instance, if the underlying object is a `UserTable`, then `set_attr` can be used to store a value at a particular attribute. Other methods, such as `set_upval`, can only be used if the underlying object is a closure.

The objects implementing this trait, are garbage collected by the value system. For the purposes of this project, the VM uses an external crate, called `gc`, which implements a garbage collector for Rust. `LuaVals` which are garbage collected are all stored with the `Gc` or `GcRoot` tags. As a disclaimer, the implementation of the `Gc`, and `GcRoot` tags was heavily inspired by the implementation of the `Gc` type from the `gc` crate.

5.2.2 The VM

The `new` method of the VM sets up a new instance, and prepares the environment in which the bytecode will execute. The execution starts in the function which is returned by the `get_main_function` method of `LuaBytecode`. The `functions` attribute of `LuaBytecode` stores a list of `BcFuncs`. The `get_main_function` method returns the index of the entry function.

On lines 3-4 of listing 5.23 the VM sets up the 256 registers that the instructions of the bytecode can use to perform operations. Line 5 sets up the `_ENV` table, which is going to be stored as an attribute of `Vm`, and passed as the first upvalue to all functions.

In the VM, a `closure` is an object which represents a function, and its upvalues. Lines 7-12 set up a `closure` which represents the module of the source file. The `index` variable, on line 8, is the index of the

Listing 5.24: `Vm::eval` and `Vm::closure` methods.

```
1  impl Vm {
2      pub fn closure(&self) -> &LuaVal {
3          &self.stack_frames[self.curr_frame].closure
4      }
5
6      pub fn eval(&mut self) -> Result<(), LuaError> {
7          self.pc = 0;
8          let index = self.closure().index()?;
9          let len = self.bytecode.get_function(index).instrs_len();
10         while self.pc < len {
11             let instr = self.bytecode.get_function(index).get_instr(self.pc);
12             (OPCODE_HANDLER[opcode(instr) as usize])(self, instr)?;
13             self.pc += 1;
14         }
15         Ok(())
16     }
17 }
```

main function in the bytecode. The `main` variable represents a reference to the actual `BcFunc` object. Line 10 creates a new `UserBcFunc` which represents the main function of the bytecode. Note that `env` is passed as an upvalue, as specified in the reference manual. On lines 13-14 the VM sets up the `stack_frames` attribute, and pushes a new frame which contains the newly created `UserBcFunc`. Stack frames are described in the following sections.

The `eval` method

This is the method that starts the execution of the current stack frame, represented by the `curr_frame` attribute.

Listing 5.24 shows the implementation of `eval`. The `Vm` sets its `pc` attribute to 0 (line 3), in order to start the execution from the first instruction. The `closure` method returns the `closure` object of the current stack frame. In the beginning, this is set to the `closure` which represents the module of the source file. Lines 6-10 implement the dispatch loop. If at any point an instruction fails, the method returns a `LuaError`.

The dispatch loop is implemented with the help of an array of function pointers. The `OPCODE_HANDLER`, from line 12, is an array that has the size of the number of operations supported by the VM. At each index of the array, the VM stores a function pointer. For example, at index 0, the VM stores the function pointer which can handle the `Mov` operation. During instruction dispatching, the VM extracts the opcode of the current instruction, and uses it as an index into the `OPCODE_HANDLER` array in order to find the function which handles that particular opcode. All of these function pointers return a `Result`, and take two arguments: the state of the VM, and the instruction being executed. The functions are defined in the `instructions` module of the `luavm` crate.

5.2.3 The `instructions` module

This module contains the implementations of all the instructions that are defined in the bytecode instruction set. Binary and unary instructions such as `add`, `equals`, `unary minus` are already implemented by the value system. In such cases, the VM delegates the operation to the value system. For example, listing 5.25 shows the implementation of the `Umn` opcode, which stands for unary minus.

The `umn` function extracts the second operand of the instruction (line 3) which represents operand of the unary operation, and negates it using the `LuaVal` API. The result is stored into the register defined by the first operand of the instruction (line 4).

Most opcode implementations follow the same basic structure, although there are a few instructions, such as `Call`, `MovR`, `VarArg`, `Closure`, that cannot simply delegate to `LuaVal`.

Listing 5.25: The implementation of unary minus.

```
// R(1) = -R(2)
pub fn umn(vm: &mut Vm, instr: u32) -> Result<(), LuaError> {
    let val = vm.registers[second_arg(instr) as usize].negate_number()?;
    vm.registers[first_arg(instr) as usize] = val;
    Ok(())
}
```

Listing 5.26: The implementation of the `Closure` opcode.

```
1  pub fn closure(vm: &mut Vm, instr: u32) -> Result<(), LuaError> {
2      let func = vm.bytecode.get_function(second_arg(instr) as usize);
3      let upvals_len = func.upvals_count();
4      let mut upvals = Vec::with_capacity(upvals_len + 1);
5      upvals.push(vm.env.clone());
6      for _ in 0..upvals_len { upvals.push(LuaVal::new()); }
7      let ufunc = UserBcFunc::new(func.index(), func.reg_count(), func.param_count());
8      vm.registers[first_arg(instr) as usize] = LuaVal::from(ufunc);
9      let caller_index = vm.stack_frames[vm.curr_frame].closure.index()?;
10     if let Some(provides) = vm.bytecode.get_function(caller_index)
11         .provides().get(&(func.index() as u8)) {
12         for (provider, upval) in provides.iter() {
13             ...
14         }
15     }
16     vm.registers[first_arg(instr) as usize].set_upvals(upvals)?;
17     Ok(())
18 }
```

The `closure` function

The `closure` function must create a new `LuaVal` of type `UserBcFunc`, initialise its upvalues, and store it into a register.

Listing 5.26 shows the implementation of the `Closure` opcode. The second operand of the instruction gives the index of the function in the bytecode. Lines 2-6 set up the upvalues of the closure, and on line 5 the first upvalue is set to `_ENV`. On lines 7-8 the VM creates a `UserBcFunc`, and stores the corresponding `LuaVal` into the register denoted by the first operand of `instr`. The `caller_index` variable represents the index of the function in which the `Closure` instruction is executed. On lines 10-15, the VM checks if the `caller_index` function provides any upvalues to the newly defined closure. Once the upvalues are updated, line 16 sets the upvalues of the `closure`.

The anatomy of a function call

In section 4.2.3 it was described how the compiler emits instructions for a function call. It is important to describe what a stack frame looks like from the perspective of the VM.

Lines 2-3 of listing 5.28 set up the local function `f`. Every function call starts with a `SetTop` instruction, whose first argument is the register which contains the closure that is about to be called. In this case, on line 4, the compiler emits a `SetTop` instruction for register 0, which contains the closure. The `SetTop` instruction creates a new `StackFrame`, and assigns to its `closure` attribute the `LuaVal` from register 0, and to `start` the current top of the stack, which is 0 in this case. On lines 5-10, the VM evaluates the arguments of the call, and pushes them to the stack. Finally, the VM executes the `Call` instruction which jumps to the definition of `BcFunc1` from listing 5.28, because the closure from register 0 points to it.

Listing 5.27: An example of a function that uses varargs.

```
local function f(a, ...)
  return ...
end
local a, b = f(1, 2, 3)
```

Listing 5.28: Bytecode for listing 5.27

```
BcFunc 0 {
  Ldn 0
  Closure 0 1
  SetTop 0
  Ldi 2 0
  Push 2
  Ldi 3 1
  Push 3
  Ldi 4 2
  Push 4
  Call 0
  MovR 5 0
  MovR 6 1
}
BcFunc 1 {
  VarArg 0 0 2
  Ret
}
```

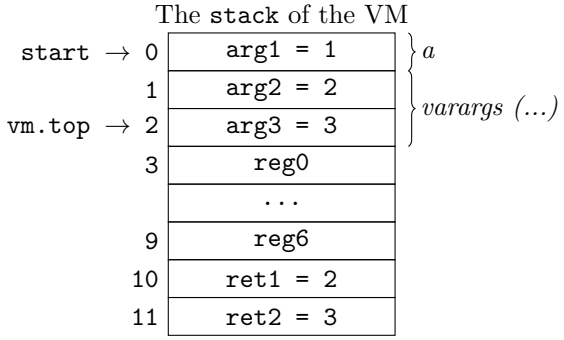

Figure 5.1: The stack of `f` before it returns to the caller.

When the VM starts executing the `Call` instruction, the top of the stack is at index 2 shown in figure 5.1. The `start` from the figure is the top of the stack recorded by the newly created `StackFrame`, i.e. the index at which the arguments of the call start.

The `call` function implements the `Call` opcode. This function first sets up the upvalues of the closure that is called using the `set_upval` method defined on `LuaVal`s. In this case the closure does not have any upvalues, other than `_ENV`, which was already set by the `closure` function. Then, the VM saves the registers of the caller, by pushing them to the stack. These can be seen on figure 5.1 between the indices 3 and 9. This is necessary because the called function might modify some of the registers of the parent. Now the registers can be modified without overwriting any of the values computer by the parent function. Function `f` expects its first argument to be in register 0, thus the VM copies `arg1` from the stack into register 0, *before* the VM jumps to the function. Finally, the VM sets its `curr_frame` to the next stack frame (the frame of `f`), and calls `eval` recursively.

Function `f` returns all of its variable arguments, as seen on line 2 of listing 5.27. This is done through the `VarArg` instruction from line 16 of listing 5.28. The last argument of this instruction is `2`, which means that the VM should push all the varargs of the current function to the stack, and increment the return values of the function. In this case, the VM calculates that the varargs of `f` start at index 1 (figure 5.1), and end at index 2. Thus, the VM pushes two return values to the stack (index 10 and 11), and sets the `ret_vals` of the current closure, which is the closure which represents function `f`, to 2. Once the recursive call of `eval` exits, the VM continues the `call` algorithm. First, the VM restores the registers of the caller. If the function modified any of its upvalues, the state of the caller must be updated as well, thus the VM does an inverse operation, where it copies the upvalues of the called function back into the registers where they came from.

The compiler always emits `MovR` instructions after a `Call` in order to save the return values of a function. The VM executes these `MovR` instructions, and copies the return values from the stack to the desired registers. Consider the instruction from line 13 of listing 5.28. The VM copies the second return value from the stack (index 11) into register 6 of the caller. Once the VM returns all the values, the stack frame is removed. This is done by simply moving the stack pointer back to the `start` attribute of the current `StackFrame`. Once the `call` algorithm finishes, the `curr_frame` is set back to the `StackFrame` of the caller.

The special `MovR` and `VarArg` operands

In the previous section, and in chapter 2, it was mentioned that both `MovR`, and `VarArg` can have a special third argument. Both of these instructions copy values from the stack stored by the VM, as such, their implementations are very similar.

There are three main uses of `MovR`:

1. If the last operand of a `MovR` is 0, then the first two operands are used by the VM to copy the specified return value from the sack into a specified register. This is used, for instance, when a function call is part of an assignment statement.

2. If the last operand is 1, it means that the VM should remove the stack frame of the previous call, and keep its return values on the stack. The main use of this is when the results of a function call are

unpacked as arguments to another function call. The VM pushes all the return values of previous call to the stack as arguments to the new call.

3. If the last operand is 2, then the VM pushes all return values of a function call to the stack, and increments the `ret_vals` attribute of the parent function. This is mainly used when a function call is unpacked in a return statement.

The nested stack frames problem

The `SetTop` instruction is a solution to the problem of nested stack frames. This occurs when the argument of a function call is another function call. To illustrate this problem consider the following Lua snippet:

Listing 5.29: Nested stack frames.

```
local function f(a, b) end -- left empty for the sake of brevity
local function g(a) return a end
f(1, g(2))
```

Listing 5.30: Bytecode for listing 5.29 with `SetTop` instructions removed.

```
1  BcFunc 0 {
2    Ldn 0
3    Closure 0 1 # load function f
4    Ldn 2
5    Closure 2 2 0 # function g
6    Ldi 4 0
7    Push 4
8    Ldi 5 1
9    Push 5
10   Call 2
11   MovR 0 0 1
12   Call 0 # call f
13 }
```

If the `SetTop` instructions are removed, the VM cannot tell which locations from the stack are meant as arguments to a call. For instance, when the VM executes the instruction from line 10 (listing 5.30), it does not know if all the arguments from the stack (lines 6-9) are meant as arguments to the closure of `g`. In Lua, it is possible to call functions with any number of arguments. Thus, the VM cannot use the number of formal parameters of a function to guess where its arguments start on the stack. `SetTop` solves this issue as it saves the top of the stack before the actual arguments of the call are pushed. The VM can now correctly identify which ranges from the stack correspond to the arguments of the current call.

5.2.4 The standard library

The VM defines a standard library which is loaded when the VM is created. The VM only supports a few functions such as `print`, `assert`, and `io.write`. When the VM is created, the implementation creates `BuiltinBcFuncs` which wrap these functions, and adds the corresponding `LuaVals` into the `env` attribute. When a user program calls `print`, the compiler emits a `Call` to `_ENV["print"]`. The VM stores the `BuiltinBcFunc` of `print` in its `env` attribute, thus the VM will call the `print` function, even though the user did not define it.

Chapter 6

Testing

6.1 Unit tests

Both the `luacompiler`, and `luavm` crates have unit tests for different components.

6.1.1 The `luacompiler` crate

The compiler has a comprehensive set of unit tests which check if the compiler emits code correct Lua IR from a given parse tree. Every feature supported by the compiler has an associated test. For instance, there are four different tests for `if` statements. This is because there are many cases that need to be handled by the compiler such as an `if` without an `else`, multiple `elseif`s, nested `if`s, etc. The tests also check if the dominator tree links, and parent-child relationships are correct, and if the compiler produces the correct `upvals`, and `provides` attributes. There are also tests for nested loops, function calls, and argument unpacking. Tests provide a good overview of how the compiler emits IR for certain sequences of instructions, thus they can be used for documentation purposes.

There are also unit tests for the `substitute_phis` algorithm of `LuaIR`. This is an important phase in bytecode generation, and the algorithm must be correct, otherwise it would not be possible to know if the VM executes bytecode correctly.

The `ConstantsMap` structure is also unit tested to check whether the constant tables are generated correctly.

The functions from the `bytecode` module are tested, in order to ensure that the VM can correctly decode instructions, and their operands.

6.1.2 The `luavm` crate

The `LuaVal` structure has many different types of tests, as it represents the main component of the VM. The test suite aims to test the interaction between all the possible types of `LuaVal`s. For instance, one of the tests checks that all integer operations work correctly on `LuaVal`s tagged with either `LuaValKind::INT` or `LuaValKind::BOXED`. Similar tests are defined for `floats`, `String`s, `UserTable`s, and `UserBcFunc`s. There are also tests which check if the binary operators are implemented correctly. For instance, the test for addition checks whether the result of adding two integers also yields an integer, or if adding a float to a string results in a float. All combinations are checked in order to ensure that the language reference is followed correctly.

The `Vm` structure is another important component that is tested. The Lua code being tested always stored result into `_ENV`. Because the VM has direct access to the `env` attribute, the tests check whether certain attributes of `env` are set, and contain the correct results.

6.2 Black-box testing

Both the compiler and the VM are black-box tested in order to ensure that Lua programs can be correctly compiled, and produce the correct results.

Black-box testing the compiler is a challenge because there is no correct answer to how a source file should be translated to bytecode. The current approach involves compiling a set of source files, and saving their bytecode to disk. The test involves compiling the source files to bytecode again, and checking if the resulting bytecode is the same as the one originally generated. The saved bytecode was manually verified to be correct. If the implementation of the compiler is changed frequently, then this is not a feasible testing approach. However, considering the small scale of the project, this kind of test will suffice as the compiler is not changed very often.

The most important tests of this project test the VM. The test suite consists of a directory which contains a number of Lua source files, ranging from small programs such as a simple function calls, to more complex functions such as Fibonacci function. In order to create such tests, it was necessary to implement the `assert` function from the Lua standard library. A test passes if the VM does not report a `LuaError`. If an assertion fails, then the VM will output a `LuaError::Error("Assertion Failed")`. For instance, the Fibonacci test defines the Fibonacci function, and asserts whether different inputs produce correct results. The VM also has a special test function which downloads the `luajit` test-suite and runs a subset of the tests. The `luajit` test-suite provides many test cases. However, the VM cannot run some of them either due to missing functionality, or because it takes to long to execute.

Chapter 7

Professional issues

This project reuses the `gc` library which is an open source project available on GitHub. It also borrows a few implementation details which are necessary to speed up the VM. While developing this project, I followed the British Computer Society (BCS) code of conduct, especially the clause about third party rights. In the source code, I made it clear if any code was borrowed from other projects by linking to the original repository of the project, and I made sure to respect the licenses of those projects.

A lot of care has been taken to ensure that the project does not have have any security vulnerabilities. Because this project builds a compiler and a VM it is important to ensure that attackers cannot exploit the implementation in any way. All the limitations of the project have been disclosed in this report.

Chapter 8

Challenges

One of the biggest challenges of this project was to design a bytecode representation of Lua. The opcodes used in the bytecode were added incrementally, and some of their semantics were also changed during development. For instance, in the beginning, it was not clear if there was a need for stack instructions, such as `Push`. However, due to how function calls work, it was necessary to add instructions that modify the stack.

Another difficult problem was the upvalue generation. To implement this, it was necessary to create dominator trees, and to come up with an algorithm that searches for symbols in order to decide whether they are global symbols or upvalues.

A lot of effort was invested into the `LuaVal` system. The current system is not optimal, and its performance is shown in chapter 9. The representation of the types that a `LuaVal` wraps was the most challenging part. This is because it was necessary to understand how Rust represents different constructs, such as `enums`, or fat pointers, in memory, and how different representations influence performance.

Tables are an essential part of the Lua language, and one of the main challenges was to design a data structure which performs efficient string look-ups. However, it turned out that the algorithm developed in the early stages of development did not work in certain cases. This section briefly describes the intuition of the algorithm, as it shows how challenging it is to consider all corner cases.

Most Lua programs load global symbols from the `_ENV` table. However, the compiler knows at compile-time which symbols are loaded from `_ENV`. To illustrate the following optimisation, consider the following Lua program, and its bytecode representation.

```
x = 1
local y = x
```

```
Function 0 {
  Ldi 0 0 0 # load 1 into register 0
  Lds 2 0 0 # load string "x" into register 2
  SetUpAttr 0 2 0 # _ENV["x"] = 1
  Lds 1 0 0 # load string "x" into register 1
  GetUpAttr 1 0 1 # y = _ENV["x"]
}
```

Figure 8.1: A simple Lua program, and its associated bytecode.

In figure 8.1, `"x"` is located at index `0` in the constant strings table. This means that there is a mapping between index `0` and `"x"`. The VM can associate to every `LuaString` an optional index, which represents the index of the string in the constant table. When the `Lds` instruction is executed, the VM fetches the $\texttt{i}^{th}$ string from the constant table, , where `i` is the value of the second operand of the `Lds` instruction, and stores it into a register. Thus, the VM has access to both the string and the index, and can store them into a `LuaString`. For the purpose of this example, consider a new type of table, `StrTable`, which has two members: an array `str_attrs` which has the size of the constant strings array, and a hashmap from `LuaVal` to `LuaVal`, called `attrs`. The way the VM can perform faster string look-ups is to check if the underlying `LuaString` has an

index or not. In the case it does not have an associated index, then the look-up is done via the `attrs` map. However, if it does have an associated index, it means that the look-up can be done via `str_attrs` array. The reasoning behind this is that the index of a string in the constant table could be used to determine the slot of its associated value. Moreover, the index can be thought of as the hash of the string. Although this seems like a good optimisation, it does not work in some cases. Consider the following Lua listing:

Listing 8.1: Program which breaks the optimisation

```
1  t["ab"] = 1 -- stores ‘1‘ at index 0 in the ‘str_attrs‘ array
2  t["ab"] -- fetches ‘1‘ from index 0 of the ‘str_attrs‘ array
3  t["a".."b"] -- returns nil, and not 1
```

On line 1 of listing 8.1, `"ab"` is the first string in the constant table, thus has index `0` associated to it. Line 1 is translated by the VM to `t.str_attrs[0] = 1`, based on the optimisation described above. However, on line 3, `"ab"` is built from two different strings, thus it will not have an associated index even though it is part of the constants table. Because the VM cannot find the associated index, it performs a look-up using `attrs`, which doesn't have an entry for `"ab"`.

Chapter 9

Critical evaluation and benchmarking

The project meets all of its function and non-functional requirements. The VM implements most features of Lua 5.3, and can run a few benchmarks provided by the LuaJIT project.

9.1 The shortcomings of `luacompiler`

The `luacompiler` project approaches bytecode generation in a different way than other implementations. PUC-Rio Lua, LuaJIT, and Luster have simple compilers which translate the parse tree into bytecode directly, and only perform constant folding. These projects mostly focus on improving the execution speed of the VM rather than on emitting the most efficient bytecode. The approach of `luacompiler` is different from the other three implementations, as it compiles parse trees to bytecode in two phases.

The SSA form of the Lua IR makes it easier to perform optimisations such as constant folding and constant propagation. Also, it is relatively easy to implement the Linear Scan [21] register allocation algorithm for an SSA IR. However, the Lua IR has a few shortcomings which were also mentioned in chapter 5. Because of how upvalues need to be handled at run-time, the compiler must emit additional `Phi` instructions to ensure that different uses of the same variable are lowered to the same virtual register. An alternative intermediate representation would make the implementation of the compiler easier to understand and to extend, as well as more efficient.

Another shortcoming of the compiler is that it does not generate debugging symbols, which means run-time errors can be difficult to debug. These have been omitted, because they were not included in the functional requirements of the project.

Compile-time errors are not user-friendly. Most of the time, the compiler panics[1] instead of returning an error to the user. For this reason, compile-time errors are often difficult to fix.

9.2 The shortcomings of `luavm`

One of the main differences between `luavm`, and other Lua VMs is the way it implements function calls. In Luster function calls are 'stackless', as described in chapter 2. The `luavm` crate handles `Call` instructions recursively, which can impact the performance of recursive programs. Performance is influenced by the fact that each recursive call creates a stack frame in the host language as well. Repeatedly creating and destroying stack frames theoretically slows down the VM, as more CPU time will be dedicated to setting up frames, rather than executing bytecode.

The `LuaVal` system has a few shortcomings as well. One of the main problems with the system is that it requires two heap allocations to allocate a `LuaObj`, due to how Rust represents trait objects, and because the data stored in `LuaVal` is constrained to 64 bits. A different approach would be to implement `LuaVal` as an `enum` as shown in listing 9.1.

This approach removes the need for `LuaObj` and `GcVal` as the enumeration can have as many variants as the VM needs. However, `enum`s are not as space efficient because the Rust compiler allocates an additional

[1] A panic in Rust is similar to an abort.

Listing 9.1: An alternative `LuaVal` representation.

```
enum LuaVal {
  Int(i64),
  LString(LuaString),
  Table(UserTable),
  ...
}
```

machine word (64 bits) along with the data contained in the `enum`. This means that a `LuaVal`'s size would become larger than 64 bits. A larger size means that fewer `LuaVal`s are going to fit into the cache of the CPU. This can lead to an increased number of cache misses, which negatively affects the performance of the VM. Another shortcoming of the `LuaVal` system is its dependency on the `gc` crate. PUC-Rio Lua, LuaJIT, and Luster implement their own specialised garbage collector. The `gc` crate does not allow objects that are garbage collected to be mutated. As such, the user has to wrap objects into a structure called `GcCell` in order to mutate their underlying data. This structure, as it is shown in section 9.3, decreases the performance of the VM considerably. The VM could potentially be made faster by implementing its own GC, similar to Luster.

Unlike PUC-Rio Lua, LuaJIT, and Luster, the `UserTable` does not implement the table array optimisation, which means that integer look-ups are much slower than in other VMs.

9.3 Benchmarks

The following section is going to focus on showing the performance of the `luavm` compared to that of PUC-Rio Lua, LuaJIT, and Luster. The benchmarks were run on `bencher10` which is a server provided by the `Software development team`[2]. This machine can be used to benchmark more accurately the performance of programs. The server runs a minimal Debian GNU/Linux instalation. Its hardware consists of an Intel Xeon 3.7 GHz quad core processor with overclocking, and hyper-threading disabled, and 32 GB of RAM. Every benchmark is run 30 times with each VM.

	luavm	PUC-Rio Lua	LuaJIT	Luster
fib(30)	$1.2580_{\pm 0.0056}$	$0.0960_{\pm 0.0028}$	$0.0140_{\pm 0.0014}$	$0.3680_{\pm 0.0028}$
fib_iter(60)	$0.0060_{\pm 0.0005}$	$0.0010_{\pm 0.0000}$	$0.0010_{\pm 0.0000}$	$0.0010_{\pm 0.0000}$
bin-trees	$0.0230_{\pm 0.0028}$	$0.0040_{\pm 0.0000}$	$0.0030_{\pm 0.0000}$	–
nsieve	$141m\ 54.334s$	$0.0360_{\pm 0.0038}$	$0.0090_{\pm 0.0005}$	$0.1680_{\pm 0.0033}$

Table 9.1: Execution times (in s) of the VMs on four different benchmarks (reported with 99% confidence intervals). Keys: '-': the benchmark cannot be run with a particular VM. Note that `luavm` executes the `nsieve` benchmark very slowly, thus its entry is measured in minutes.

As it can be seen in table 9.1, the `luavm` does not execute the benchmarks as fast as the other VMs. As expected, LuaJIT outperforms every other VM on almost all benchmarks, expect on `fib_iter` where it performs just as well as PUC-Rio Lua, and Luster.

There are a couple of factors which can influence the performance of the whole system:

1. Redundant instructions in the bytecode
2. The recursive implementation of `Call`
3. The naïve implementation of table look-ups

[2] `https://soft-dev.org/`

4. The garbage collector

5. The double pointer implementation of `LuaObj` and `GcVal`

All of these are equally likely to affect the performance, as such the following paragraphs describe in detail which factors influence the performance of the VM on every benchmark. In order to better understand which functions of the VM are executed the most, the benchmarks are run with valgrind, in particular with the *cachegrind* tool. This tool gathers information about which functions were executed the most, thus giving an idea of where the VM could be optimised.

9.3.1 Recursive fibonacci

This benchmark could be potentially influenced by all the factors mentioned above. The benchmark calls the fibonacci function recursively, which means that there are a lot of `Call` instructions executed. Also, because the function is declared as a global symbol, it means that before a `Call` instruction, the symbol is loaded from `_ENV`. Thus table look-ups, and table accesses might also slow down the VM. Valgrind is run on a modified version of the benchmark, where the fibonacci function is called with the argument `10` instead of `30`. The modification is necessary because programs running in valgrind are slower due to the profiling done by the tool.

15.39	_int_malloc	malloc.c
9.15	_int_free	malloc.c
4.36	malloc	malloc.c
3.80	__memcpy_avx_unaligned_erms	memmove-vec-unaligned-erms.S
2.86	malloc_consolidate	malloc.c
2.76	_LTstd..collections..hash..map..HashMapLTK$C$$u20$V...	(unknown)
2.35	regex::re_unicode::Regex::find_at::h98f394a553919467	(unknown)
2.25	_$LT$$RF$$u27au20mutu20$bincode..de..Deserializer$...	(unknown)
1.73	_LTserde..de..impls..LTimpl$u20$serde..de..Deserialize$...	(unknown)
1.58	regex::compile::Compiler::compile_finish::h1aba0b3b07078a91	(unknown)
1.51	free	malloc.c
1.34	regex::compile::Compiler::c_class::hf8049f1eabf5af01	(unknown)
1.31	regex::compile::Compiler::fill::h592cf38290f51e2e	(unknown)
1.30	_LTstd..collections..hash..map..HashMapLTK$C$$u20$V...	(unknown)
1.27	realloc	malloc.c
1.26	std::collections::hash::table::make_hash::h51a12bc7619b8fdd	(unknown)
1.21	__memset_sse2_unaligned_erms	memset-vec-unaligned-erms.S
1.21	_int_realloc	malloc.c

Figure 9.1: A visualisation of *cachegrind*'s output for fibonacci.

Figure 9.2 shows the output of *cachegrind* for the modified fibonacci benchmark. The VM spends most of its time allocating and freeing heap memory: `_int_malloc`, `malloc`, and `_int_free`. The most heap-heavy operations that the VM does are: string, closure, and table creation, and string copies.

Listing 9.2 shows a subset of the instructions produced by `luacompiler` from the fibonacci source file. It can be seen that the compiler emits an `Lds` of the same string multiple times per function. This means that during every recursive call, the VM will create two `Box<Box<LuaString>>`, due to how `Lds` is implemented. `Box` allocates memory on the heap, thus it is very likely that its implementation calls `malloc` [3]. A nesting of boxes means that there are at least two calls to `malloc`. Also, the implementation of `Lds` copies a string from the constant string table, which also reallocates memory on the heap. This means that each `Lds` operation produces 3 heap allocations, thus 6 heap allocations per function call. The running time can be improved in two ways. First, the compiler could reuse the register which stores the string, instead of reloading it again into another register. Manually removing the extra `Lds` in `BcFunc 1`, and running the benchmark again yields a better result: 1.184 s. Emitting better bytecode improves the running time by 5%. Rerunning *cachegrind* gives the same result, that there are a large number of heap allocations and deallocations. A second approach to improving the running time is to try and remove the nested `Box`es, which means that `LuaVal` needs to be reworked.

[3] The `malloc` function allocates heap memory, and is part of `libc`.

Listing 9.2: The bytecode of the fibonacci function.

```
BcFunc 0 {
  Closure 0 1
  Lds 3 0
  SetUpAttr 0 3 0 # _ENV["fib"] = R(0)
  Lds 1 0
  GetUpAttr 1 0 1 # R(1) = _ENV["fib"]
  ...
}
BcFunc 1 {
  ...
  Lds 7 0
  GetUpAttr 7 0 7
  ...
  Lds 11 0
  GetUpAttr 11 0 11
  ...
}
```

9.3.2 Nsieve

The nsieve benchmark calculates the number of prime numbers up to a certain value, and stores them into a dictionary. On this benchmark, the `luavm` performs very poorly. This benchmark was designed to test whether integer look-ups are implemented efficiently. Because PUC-Rio Lua, LuaJIT, and Luster implement the table array optimisation presented in chapter 2, they all execute the benchmark rapidly. However, theoretically, not performing this optimisation should not hinder the ability of the VM to execute the benchmark in a timely manner.

30.02	_LTluavm..lua_values..lua_table..UserTable$u20$as$u20$luavm..lua_values..gc_val.....	(unknown)
28.44	core::ptr::real_drop_in_place::hcc68c788ef9a8f0d	(unknown)
5.17	_int_malloc	malloc.c
3.01	_int_free	malloc.c
2.38	regex::re_unicode::Regex::find_at::h98f394a553919467	(unknown)
1.43	malloc	malloc.c

Figure 9.2: A visualisation of *cachegrind*'s output for nsieve.

Figure 9.2 shows the output of *cachegrind* for a modified nsieve benchmark. This time the VM seems to spend most of its time in `UserTable::set_attr`, and in `drop_in_place`. It is expected that `set_attr` is executed many times, as the benchmark is supposed to test attribute stores.

In Rust, when an object goes out of scope, the language inserts a call to `obj.drop()`. The `drop` method is similar to a destructor in C++, and it is used to clean up the object.

In order to better understand why the `drop` method is called so often, consider the implementation of `UserTable::set_attr`.

Line 4 of listing 9.3 hides a couple of important operations. The `borrow_mut` method is used to access the underlying data of a `GcCell`, and it returns a `GcCellMutRef` structure. This structure is a wrapper around a reference to the hash table of the `UserTable`. The code then uses the `insert` method to update the hash map. Once the new value is inserted into the map, the algorithm executes a `GcCellMutRef.drop()`. This is where the implementation of the external `gc` crate influences negatively the performance of the VM. If the hash map is not wrapped into a `GcCell`, then the benchmark runs in 0.872 s, however the VM now leaks memory, and crashes upon exit.

Listing 9.3: The implementation of `set_attr` for `UserTable`.

```
impl GcVal for UserTable {
    ...
    fn set_attr(&self, attr: LuaVal, val: LuaVal) -> Result<(), LuaError> {
        self.v.borrow_mut().insert(attr, val);
        Ok(())
    }
    ...
}
```

9.3.3 Iterative fibonacci, and binary trees

In both of these cases, `luavm` is a few times slower than its counterparts. These two benchmarks are more general benchmarks, and they test the overall performance of the VM. Note that Luster does not implement a language feature used in `bin_trees`, thus its entry is missing. The performance of `luavm` could be improved by emitting better bytecode, and by optimising different components of the VM, such as the type system.

Chapter 10

Conclusion

This project shows that is is possible to create a virtual machine for a dynamic language such as Lua in Rust. In addition, this project proves the hypothesis that it is not possible to create an efficient interpreter without a specialised garbage collector. Luster is able to run faster because it created its own garbage collector. Chapter 9 showed how the `gc` crate negatively influences the run-time of the VM, by comparing the execution time of `luavm` with PUC-Rio Lua, LuaJIT, and Luster, and by analysing which components of the system are slowing the VM down the most.

10.1 Future work

The `luavm` crate can be extended in a few ways to improve its performance. First, the system needs a new garbage collector, either one similar to the one developed by Luster, or a reimplementation of an Immix garbage collector [22]. Other ways in which the VM could be improved include: adding support for the table array optimisation, and removing extra heap allocations that are generated by the value system.

The `luacompiler` can be extended in many ways. Currently the compiler lacks a register allocation algorihm, and also peforms lowering to bytecode very naïvely. Improving these two aspects about the compiler should enable better bytecode emission, thus an improvement in the overall performance of the interpreter.

Bibliography

[1] ripgrep. `https://github.com/BurntSushi/ripgrep`. Last accessed: 2019-03-10.

[2] fd. `https://github.com/sharkdp/fd`. Last accessed: 2019-03-10.

[3] Rocket. `https://github.com/SergioBenitez/Rocket`. Last accessed: 2019-03-10.

[4] The CPython Bytecode Compiler is Dumb. `https://nullprogram.com/blog/2019/02/24/`. Last accessed: 2019-03-10.

[5] Rust IRs. `https://blog.rust-lang.org/2016/04/19/MIR.html`. Last accessed: 2019-03-12.

[6] Go language uses SSA. `https://golang.org/doc/go1.8#compiler`. Last accessed: 2019-03-10.

[7] SSA in LLVM. `http://llvm.org/docs/LangRef.html`. Last accessed: 2019-03-10.

[8] Visual C++ SSA. `https://devblogs.microsoft.com/cppblog/new-code-optimizer/`. Last accessed: 2019-03-10.

[9] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[10] Jeremy Singer, Christos Tjortjis, and Martin Ward. Using software metrics to evaluate static single assignment form in gcc. 07 2010.

[11] Hanspeter Mssenbck and Michael Pfeiffer. Linear scan register allocation in the context of ssa form and register constraints. volume 2304, pages 229–246, 04 2002.

[12] Ron Cytron, Barry K. Rosen Jeanne Ferrante, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programmmg Languages and Systems*, 13:451–490, 1991.

[13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson, 2007.

[14] PUC-Rio Lua. `https://github.com/lua/lua`. Last accessed: 2019-03-17.

[15] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The implementation of lua 5.0. *Universal Computer Science*, 11:1159–1176, 2005.

[16] LuaJIT VM. `http://luajit.org/index.html`. Last accessed: 2019-03-17.

[17] LuaJIT compiler optimisations. `http://wiki.luajit.org/Optimizations`. Last accessed: 2019-03-17.

[18] LuaJIT SSA IR. `http://wiki.luajit.org/SSA-IR-2.0`. Last accessed: 2019-03-17.

[19] Luster implementation. `https://github.com/kyren/luster`. Last accessed: 2019-04-03.

[20] Luster description. `https://www.reddit.com/r/rust/comments/awx9cy/github_kyrenluster_an_experimental_lua_vm/`. Last accessed: 2019-04-03.

[21] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21:895–913, 1999.

[22] Yi Lin, Stephen M. Blackurn, Antony L. Hosking, and Michael Norrish. Rust as a language for high performance gc implementation. *ACM SIGPLAN Notices - ISMM '16*, 51:89–98, 2016.