Lua to WASM Compiler - Documentation

ROSS EVANS

1 INTRODUCTION

My project for the term was to compile Lua programs into the WebAssembly format. There were a couple of motivations behind this. For one, I have not yet been fortunate enough to take a graduate level compilers course, so all my knowledge in compilers came from UW's CS 241 course. The CS 842 project seemed like a good opportunity to supplement my knowledge in compiler development. Furthermore, the compiler I had implemented in my undergrad was fairly simplistic. This compiler was statically typed with only two types: integers and closures. This marks a big difference from Lua's support for integers, floats, booleans, tables, function closures, and nil values. I had never implemented a compiler for a dynamically typed language, nor I had ever written a garbage collector. My goal for this project was to gain experience in each of these areas.

In terms of language choice I chose Lua and WebAssembly for a couple of reasons. Lua was chosen because it is a relatively simple language, and one that I have good knowledge of the semantics of thanks to previous experiences using it for game development. WebAssembly was chosen as a learning opportunity for me. Despite working as a web developer professionally and recreationally for over 7 years now, I had never touched WebAssembly before this project. I saw writing a compiler as a good way to dive head first into learning how WebAssembly's virtual machine works, what functionalities it supports, and what considerations need to be made when using WASM as a compilation target.

2 INSTALLATION INSTRUCTIONS

The compiler is written in TypeScript. As a web developer by trade, I am most comfortable with the JavaScript/TypeScript ecosystem, and thought it would be a natural choice for the project's large undertaking.

To compile a module, a number of dependencies are first needed.

(1) Install NodeJS if it is not already installed on your machine
(2) Install the Web Assembly Binary Toolkit. If on Mac with Homebrew installed, `brew install wabt` suffices.
(3) The node package manager `npm` should be included in your NodeJS installation. You can then use `npm install` from the project directory to install some dependencies specified in `package.json` (typescript, antlr, etc.).
(4) Run `npm run build` from the project directory to build the TypeScript into JavaScript
(5) Then, you can use the script `compile.sh` to compile a program. E.g, `./compile.sh testPrograms/primes.lua`. This generates a file called `compiled.wat` in the current directory, which is the output of the compiler. It also automatically uses the WABT's `wat2wasm` utility to convert the outputted text format into a binary, and moves the binary to `web/compiled.wasm`
(6) Finally, one can run the assembly through the included JavaScript runtime. Doing so requires one to host the page included in the `web` directory. An easy way to do this is to navigate into the `web` directory and run `python3 -m http.server` which will launch a local server on `localhost:8000`.
(7) Navigate to `localhost:8000` in Google Chrome or Firefox and click the run program button to execute the WASM code.

Author's address: Ross Evans, rpevans@uwaterloo.ca.

3 ORGANIZATION OF THE COMPILER

The compiler is implemented in essentially four steps.

(1) Parse Tree Construction
(2) AST construction
(3) AST annotation
(4) Code generation

Construction of the parse tree is done by the open source parser generator ANTLR. There is a Github Repository for ANTLR which contains a list of grammars for a variety of different languages. I modified this grammar slightly to provide it with some additional annotations, and to remove some features that I did not intend to support, such as variable attributes, which were a new addition in Lua 5.4. The file `src/antlr/Lua.g4` contains my modified version; the remaining files in `src/antlr` are all automatically generated by the ANTLR executable. Using this open source tool saved me a large amount of time on the project, which I was grateful for, since the focus of the project was on the compilation and the WebAssembly target rather than parsing.

The second step performed by the compiler is AST construction. Parsing via ANTLR is convenient, but working with the parse tree it generates is not. For one, the Lua grammar can have quite complicated derivations for even simple language constructs. The rule for a variable follows below:

```
var
    : (NAME | '(' exp ')' varSuffix) varSuffix*
    ;

varSuffix
    : nameAndArgs* ('[' exp ']' | '.' NAME)
    ;

nameAndArgs
    : (':' NAME)? args
    ;

args
    : '(' explist? ')' | tableconstructor | string
    ;
```

As one can see, a variable is either a simple name, or an expression with a `varSuffix`. The `('[' exp ']' | '.' NAME)` is what allows indexing of Lua tables, but even that might be the result of a function call as a result of `nameAndArgs`. I realized early on that dealing with a parse tree like this would be too unwieldy, and I needed my own AST to simplify things.

Creating my own AST also allows me to perform transformations before annotations or code generation. For example, Lua supports the following syntax for a numeric for loop:

```
for i = 1, 10 do
    print(i)
end
```

This prints out the integers from 1 to 10. The Lua 5.3 Reference Manual details a transformation of this into simpler primitives. When presented with an ANTLR subtree for:

```
for v = e1, e2, e3 do block end
```

I translate this into:

```
do
  local var, limit, step = e1, e2, e3
  var = var - step
  while true do
    var = var + step
    if (step >= 0 and var > limit)
       or (step < 0 and var < limit) then
      break
    end
    local v = var
    block
  end
end
```

I also do similar transformations for Lua's generic for loop syntax (which is used to implement iterators), and Lua's repeat-until statements. This allows all "looping code" to be implemented solely in the form of while loops.

Another example that demonstrates the benefit of creating my own AST from ANTLR's parse tree is de-sugaring function definitions. The following two function definitions are equivalent:

```
function f()
    block
end

f = function()
    block
end
```

However, the derivation for the first is simply `chunk -> block -> statement -> function funcname funcbody`, whereas the second is `chunk -> block -> stat -> varlist = explist`, with `varlist -> var` and `explist -> exp -> functiondef -> 'function' funcbody`. Despite being semantically equivalent, the parse trees are remarkably dissimilar. Creating my own AST allowed me to ensure that both of these definitions had the same canonical representation inside of my AST.

The code to generate my AST based off of ANTLR's parse tree is located inside `src/AntlrVisitor.js`. The compiler is mainly written in TypeScript, but unfortunately ANTLR does not provide type definitions for the parsers that it generates. I isolated the portions of my compiler using ANTLR into `src/AntlrVisitor.js` and `src/main.js` so that the rest of the compiler could fully use TypeScript.

The third step in the compilation process is AST annotation. Lua is weakly and dynamically typed, so there isn't anything to do in terms of static analysis. A typical Lua implementation runs inside of an interpreter, and semantic errors are sent to this interpreter at runtime [Ierusalimschy 2004]. In the case of my compiler, programs with errors simply result in undefined behaviour. This is revisited in the Further Improvements section.

AST annotation occurs in three steps via the `ScopeVisitor`, `StringVisitor`, and `IndexVisitor` classes. Each of these classes inherits from an abstract `AstVisitor` class, and each `ASTNode` contains an `accept` method which in general calls a `visit` method for this particular type of `AstNode` on the `AstVisitor`, recurses on its children, and then calls a `leave` method for this particular type of `AstNode` on the `AstVisitor`. This use of the visitor pattern allows me to write custom behaviours for each type of visitor while reusing the general tree traversal code. Furthermore, visitors can maintain internal state, and the code for a particular type of visitor can be self-contained rather than scattered among `ASTNode` implementations.

The `StringVisitor` and `IndexVisitor` are both fairly simple. The memory layout is discussed further in Memory Layout section of the document, but essentially, we need to create a data segment storing any strings that may be used in the program, since WASM has no native support for strings. The `StringVisitor` simply looks for instances of `StringNode`, which represents a string literal in the program, and for use of global variables, which are implemented in terms of a string lookup in a global table, and creates a set containing all of these strings. The `IndexVisitor` simply assigns each `Function` and `WhileStatement` in the AST an index, and associates `BreakStatements` with the index of the corresponding `WhileStatement` it is breaking out of.

The `ScopeVisitor` does the work of determining whether a variable is in scope or not, as well as which frame it belongs to. An example might look like:

```
local a = 1
local function f()
    print(a)
    print(b)
    local a = 2
    b = 5
    print(a)
    print(b)
end
f()
print(a)
print(b)
```

Variables in Lua are global by default. The first call to `print(a)` should print 1, since `a` is not yet declared in `f`, but there is a definition of `a` in the enclosing lexical scope. The first call to `print(b)` should print `nil`. We search for a global variable in the global table in this case, and find nothing. After the declaration of `local a = 2` and `b = 5`, we should print 2 and 5 respectively. After `f` is done, we should print 1 and 5, since setting `local a = 2` does not affect the `a` in the outer scope, but setting the global does.

The `ScopeVisitor` walks through the parse tree, maintaining a stack of functions, and of blocks. Functions are assigned a nesting depth as they are traversed. When a local assignment is encountered, we register the variable name with its surrounding block and its surrounding function, so long as it's not a redeclaration. When encountering the use of a variable, either in a non-local assignment, or an expression, we iterate through the stack of blocks to determine whether it's a local or global variable. If it doesn't exist in the stack of blocks, then we know it's a global access, and we can transform this at the AST level to a field access in the global table. If it does appear in the stack of blocks, then we can note the use of the variable as local, and also compute which function the variable was initially declared in. If the variable was used in a nested function, but defined in the enclosing function, then during code generation the variable won't be found in the current stack frame, it'll be found in the statically enclosing stack frame.

Having annotated the AST with all the pertinent information, we can now perform the code generation step. The `CodeGeneration` class handles the layout and formatting of the Web Assembly Text (WAT) file. It includes a number of WASM functions to do things like hashing, memory allocation, and checking variable equality. Finally, it uses the `CodeVisitor` class, to traverse the tree and do the majority of the brute work. The `CodeVisitor` maintains an array `functionWasms` which, after having traversed the AST, stores the instructions in the WAT format for each function found inside the program. These instructions are inserted into the WAT output by the `CodeGeneration` class, the `CodeGeneration` class then returns a fully formed `WAT` program to main, which is printed to `stdout`.

4 WASM EXECUTION

To discuss what the compiled code does at run-time, we have to consider what functionality WASM supports. Unlike the architectures I had previously studied, which were all types of register machines, WASM is executed via a stack machine [Contributors 2022]. The types that can be used in this stack machine are very limited however; any value on the stack must be a 32 or 64 bit integer or floating point number. Of course, this is insufficient for more complex data types, so there is also the option to use WebAssembly Memory: which is just a block of linear memory (my runtime ended up initializing just 1 64KB page to ensure that I could easily trigger garbage collection).

There is no inherent structure enforced within this linear memory, it is entirely up to the WASM being executed to ensure any invariants. In addition to the stack machine and linear memory, WASM also supports mutable globals that can be fetched or set at any point throughout execution. My solution to the memory layout problem was therefore to mimic other architectures: my memory is divided into a data segment, a stack, and a heap. The data segment, as previously mentioned, is never written to and stores strings that are known to be used within the program. The stack is used for intermediate computations with complex Lua values. For example, a call like `f(x)` would first evaluate `f` for the closure value, push this onto the stack, evaluate `x`, which could be of any type, push it onto the stack, call the associated function, and then pop the two values. WASM's stack is unfortunately insufficient for dealing with these more complex value types. Finally, the heap is used for dynamic memory: I store some information in the heap for all Lua types apart from `int`, `boolean` and `nil`. We then use the mutable globals to keep track of a stack pointer, a heap pointer, and a frame pointer.

4.1 Lua Types

This section will discuss the implementation of the various types defined in Lua. Since Lua is dynamically typed, one must be able to reassign any variable to one of a completely different type. This poses a potential challenge if the variables are of different sizes. My solution for this was to ensure that all variables could be stored using an 8 byte representation. The first 4 bytes always stores the type of the variable in a 32 bit integer. The latter 4 bytes then stores the "value" of the variable. In the case of Lua's integer type, it is the integer. For booleans, it's 0 for false, 1 for true. For the nil type, 0 is always stored in the value. And finally, for all other types, a pointer is stored to some location in the heap containing further information. This means that assigning one variable to another is as simple as overwriting 8 bytes. Furthermore, this is aligned with the description given in the Lua 5.3 Reference Manual: it notes that "Tables and functions are objects: variables do not actually contain these values, only references to them. Assignment, parameter passing, and function returns always manipulate references to such values; these operations do not imply any kind of copy." [Roberto Ierusalimschy 2015] Strings, being immutable in Lua, also can simply be assigned by overwriting the 8 byte variable.

Given that the types of my variables can only take on 9 possible values, I will acknowledge that it is somewhat wasteful to use an entire 32 bit integer on them. For example, the nil type could easily be represented with just a 4 byte sequences of all zeros; the associated value type for nil doesn't really achieve anything. I did this for memory alignment reasons however. Initially I did attempt to store some types, like nil, and an internal pointer type using just 4 bytes. However, upon evaluating some number of expressions on the stack, it became difficult to determine what number of bytes to be popped if it was a mixture of 4 byte and 8 byte values. Further improvements to this compiler could investigate packing the type of a variable in a more condensed manner.

4.1.1 Strings. Strings in Lua are read-only sequences of bytes. The value inside of a string variable is a pointer to either the heap or the data segment. Stored at this memory address is a 32 bit integer representing the number of characters in the string, which is immediately followed by a sequence of 1 byte characters. When a new string is allocated on the heap, for example, when two strings are concatenated together, we ensure that the heap pointer is adjusted not only by the length of this string object, but also by any extra space needed to ensure the heap pointer is 4 byte-aligned.

4.1.2 Functions. Functions in Lua are first class values. As such, we have to be able to store functions inside of variables, and freely reassign them to one another. The value inside of a function is a pointer to the heap. Located in the heap is another 8 byte structure. The first 4 bytes is a 32 bit integer representing the index of the function. To allow for first class function values, WASM supports a `call_indirect` instruction. To use `call_indirect` requires that

a WASM table is defined, which just lists whichever WASM functions you may wish to dynamically call in a canonical order. The `call_indirect` instruction then takes in an integer, which corresponds to the function at that given index in the WASM table. The second 4 bytes is a 32 bit integer representing the environment of the closure. For example:

```
function addBy(x)
    return function(n)
        return x + n
    end
end
addBy5 = addBy(5)
print(addBy5(10))
```

In this case, our function variable `addBy5` would have a function index of 2 (since it is the second function encountered in the program). The 32 bit integer representing the environment would be a pointer to `addBy`'s function frame. This allows `addBy5(10)` to search for the parameter `x` even once `addBy` has finished execution.

4.1.3 Tables. Lua's only type of data-structuring mechanism is the table type, which functions as a map from any type (excluding nil) to any other type. Before constructing my compiler I was unsure of the actual mechanisms in which tables were implemented in Lua. Lua 4.0 simply used a hash table for all values, however Lua 5.0 onwards separates the data structure into two parts: an array for keys with integer values, and a hash table for all other keys. There is also an intelligent algorithm for resizing to ensure that the array portion is reasonably dense, i.e `a[1000000] = 1` should not immediately allocate an array of size `1000000`. Due to time constraints, I ended up implementing solely a hash table, akin to Lua 4.0.

The value associated with a hash table is a pointer to a 12 byte collection stored in the heap. The first 4 bytes stores how many key value pairs are stored inside the table. The second 4 bytes store the total capacity of the table currently. These two values combined allow us to determine when to resize our table. Finally, the last 4 bytes are another pointer, to a chunk of memory of size (`capacity` times 16 bytes). Each 16 byte block stores an 8 byte key variable, followed by an 8 byte value variable.

My hash function is simple: for all keys apart from strings, I simply take the latter 4 bytes of the variable storing the value (for integers and booleans), or the pointer (tables, functions) modulo the capacity of the table. For strings I used a polynomial hash function, multiplying each character of the string interpreted as an integer between 0 and 255 by the prime 257 raised to the power of the index of the character. Once again, we taking the value modulo the capacity of the table. I compute this polynomial efficiently using Horner's method.

Having evaluated the hash function to get an index within the hash table's array, I attempt to store the key and value at that particular index. When collision occurs, I simply use linear probing to find the next available slot. When the hash table array exceeds the threshold of 1/2 full, I double the size of the array, round up to the nearest prime, and rehash all the elements from the old array into the new array. Ensuring that the capacity of the array is always a prime is to try and decrease collisions - since our key for the table and function type is a pointer, which is always divisible by 4, we may have clustering if we used a composite capacity.

Further work would be valuable to determine the efficiency of this implementation over different types of data. Given the time constraints I was only able to check for correctness, not efficiency. It would also be valuable to employ some strategy like Lua 5.0 to ensure that the table is fast even if only used with dense integer keys.

4.2 Functions and Scoping

Functions may need to access variables outside of their immediate scope in Lua: this is a result of Lua supporting closures and nested functions. I implemented functions using frames: when a function is called, it allocates space for a function frame, which is 12 bytes, followed by 8 bytes times the number of local variables declared inside the function. The first 4 bytes of the function frame is the static link, it points to the frame of the enclosing function (code written outside of any function is permitted in Lua, in which case the static link is set to -1). The second 4 bytes is the dynamic link, which is what allows us to reset the frame pointer FP to the function caller upon return. Finally, the last 4 bytes contain the number of local variables, which allows us to determine the size of the frame during garbage collection.

As seen earlier in the case of `addBy`, if we were to allocate `addBy`'s frame on the stack, then when we call `addBy5(10)`, the variable `x` stored in `addBy`'s frame would have already been popped. Functions therefore cannot be implemented on the stack for variable lifetime reasons. I allocate all function frames on the heap; frames which are not referenced in any form are cleaned up during garbage collection. A more optimized compiler could likely do some static analysis to determine when closures could be created, and only allocate frames on the stack if necessary. For the sake of time and simplicity, I simply maintain that all function frames are on the heap.

We annotate in the AST whether a variable is local or global. If a variable is local, we also know which function it is used in, and which function it was declared in. We can compute via the difference in nesting depths how many times to follow the static link. If it was declared and used in the same function, the nesting depth is zero, so the variable can be found in this frame. If the it was declared in a higher function, then we follow the static link the number of times corresponding to the difference between the functions nesting depths, this gives us the frame where the variable was declared, and we can then access the variable as its predefined offset.

The dynamic link is simple to compute, it is simply the current frame pointer. We compute the static link once again depending on the nesting depths. If we are calling a nested function, then we are the enclosing function, so we pass the caller's frame pointer as the static link. If we are calling a sibling function, then we share an enclosing function, so we forward our static link to our sibling. Finally, if we are calling out to an enclosing function, we calculate the difference in the nesting depths (call it n), and forward our static link after following it n times.

4.3 Return Statements

One of Lua's nice features is that it has native support for multiple return values - rather than needing to package multiple return values into an array or a dictionary like structure, a function can simply do something like:

```
function f()
    return 1, 2
end
```

I have often wondered why it is that other languages do not support this sort of construction, which can be very convenient as a programmer. I know now that it is because it is a *pain* to implement. Functions returning multiple values has a large impact on the semantics of the language. For example:

```
function f()
    return 1, 2
end
function g(x, y)
    print(x)
    print(y)
end
g(f())
```

Lua specifies that in this case, the function `g` should be called with the arguments 1, 2. However, in another case:

```
function f()
    return 1, 2
end
function g(x, y, z)
    print(x)
    print(y)
    print(z)
end
g(f(), 3)
```

This prints out 1, 3, nil. The Lua standard states that if a function call is the last thing in a list of expressions, then all of its return values should be used - otherwise, only the first return value is used and the other is discarded. However, in the construction of tables, one can do something like this:

```
myTable = {-100, x = "Hello", true, f(), y = nil}
```

In which case, we should have that `myTable[1] = -100`, `myTable.x = "Hello"`, `myTable[2] = true`, `myTable.y = nil`, and the values `myTable[3], myTable[4]`, and onwards are filled with the multiple return values of `f`.

To implement this, I created a type used internally, called `ReturnArray`. A `ReturnArray`'s value is a pointer to the heap, which contains a 4 byte capacity, followed by 8 byte variables for each value being returned. When a function is called, it always pushes a ReturnArray on the stack. For each function call in the AST, I annotate it with how many return values are needed. If the entire statement is just a function call, we generate code to immediately pop the ReturnArray from the stack. If it's used in an expression context, or in an expression list where it isn't the final element, then we fetch the first value inside of the ReturnArray and replace the ReturnArray with this value on the stack. Finally, if the function call is the last expression in an expression list, then the ReturnArray is left on the stack, and it is assumed that the ASTNode that evaluated the expression list can handle the ReturnArray type.

Looking at another example:

```
function g()
return 5, 6
end
function f(x, y, z)
end
f(1, 2, 3, 4, g())
```

It is completely legal for functions to be called with extra arguments in Lua, those arguments are simply discarded. This poses a challenge for allocating the frame of a function: we need to be inside the function declaration to know how much space to allocate for the frame, but once we're in the function, we no longer know the number of arguments it was called with. The solution is that function calls not only push each argument onto the stack, but also an integer for the number of arguments in total. The function call prologue then has to be able to look through each argument given, copy it into it's own frame, while also handling that the last argument given might be a 'ReturnArray', in which case it will need to start copying values from the heap. If at any point we run out of arguments in the frame to copy into, then the rest should be discarded, and if it any point we run out of arguments provided to the function, then we have to set the remaining arguments in the frame to nil.

In short, having dynamic number of arguments and dynamic number of return values is complicated.

4.4 Output

Lua provides the `print` function as a primitive to print to `stdout`. WASM does not have any native support for I/O, so this was handled by using `imports`. A WASM module can specify a list of import functions which must be provided by the client who is executing the program. All programs that I compile must import a function called `print`. WASM requires type signatures on each of its functions, so I specify that the `print` function takes in a 32 bit integer, which I use to pass in the memory location of the variable to be printed. The WebAssembly memory can be used not only from within the WASM module, but also within the JavaScript which instantiates and runs the module. This allows the print function to fetch the variable given its memory location, it can inspect the type of the variable, from the type determine the expected formatting, and finally, append the output to the DOM. The behaviour of Lua's `print` function can be completely customized by a client to suit the needs of their web application.

4.5 Garbage Collection

Garbage collection was one of the things that I hoped to implement for the compiler: it becomes especially necessary with all of my function frames being placed in the heap. I was able to write an implementation for garbage collection, but I ended up doing it in the JavaScript runtime rather than as part of the compiled code.

Whenever a request is made for memory by the compiler, it calls an `alloc` function with the number of bytes requested. This `alloc`

function always increments the heap pointer by said number of bytes, but before doing so, it calls to an imported function `gc`.

My basic function `gc` implements a stop and copy garbage collection method: it divides the heap into two halves, the from-space and the to-space. If the request for the number of bytes would cause the heap pointer to write outside of a half-space, then the garbage collection runs. The live set in this case can be determined recursively from the frame pointer. The frame pointer gives us the frame of the current executing function, and we can then determine the entire call stack by following the dynamic link. We then begin copying frames into the to-space from the bottom of the call stack upwards. We maintain a list of pointer intervals, which describe how pointers in the from-space map to memory locations in the to-space. Whenever we copy heap memory over, we update the list of pointer intervals accordingly. If a variable is stored inside of a function frame, it is considered live. For simple types like integers, booleans, and nil, we can simply copy the variable from the old frame to the new frame. For strings, if the string is not located in the data segment, and its pointer isn't located in any of the stored pointer intervals, then we must copy the string's data into the to-space. Tables have to have their data copied into the to-space, and we recursively copy each key value pair. Finally, a function closure might have an environment which is no longer part of the call stack, in which case the frame given by this environment pointer should be copied, which will then recursively copy its call stack as well.

Having copied over the live portions of the heap, we then iterate through all variables on the stack, and update any pointers accordingly from the pointer interval map.

Time constraints were a key factor in implementing `gc` in the JavaScript runtime; I didn't think I had enough time to implement it directly in the WebAssembly module. However, it also affords some niceties. For one, it can be seen as a feature of the compiler itself: just as with the imported `print` function, if you don't like the garbage collection given to you, you are free to implement your own. Lua is often used in game development, an example where one might desire this functionality to customize the garbage collection is if stop and copy is too inefficient; with garbage collection being a somewhat slow process, we would not want it to cause a frame drop in the game being played inside the browser. Furthermore, implementing garbage collection in the runtime is faster and easier which can allow one to easily prototype and determine which garbage collection algorithm works best for a particular use case.

A further improvement to this compiler might be including a flag to toggle between garbage collection through the JavaScript runtime and some garbage collection written directly into the compiled WASM module.

5 FEATURES SUPPORTED

My initial project proposal was that I would compile a subset of Lua to WebAssembly, so this section intends to detail exactly the features I support, with the Further Improvements section detailing what would be necessary to implement the remaining features.

- Integer, boolean, string, nil, function, and table values
- If statements, While loops, Repeat-Until loops, Numerical For loops, Generic For loops over iterators
- Tables as maps from any type of non-nil key to any type of value
- String concatenation
- Assignment between variables of different types
- Well-defined equality checks between any types
- Functions as first class values with lexical scoping
- Functions returning multiple values
- Forwarding of multiple return values in return statements and function arguments
- Boolean operators `and` and `or` with short-circuit evaluation, as well as `not`
- Boolean operators over integers: addition, subtraction, multiplication, division, modulo, less than, greater than, less than or equal to, greater than or equal to
- Garbage collection
- Primitive functions `print` and `ipairs`

6 FURTHER IMPROVEMENTS

There are a number of remaining Lua language features that I unfortunately did not have time to implement. Some of these include:

6.1 Floats

Lua's number type is generic, the language supports integers and floating point numbers with conversions happening between the two. I do not think this would be difficult to implement within the compiler, it would just require run-time type checking during binary operations, which is already done for things like equality.

6.2 Metatables

If an operation between two types is not defined, one can define a function inside a metatable. For example, adding two tables together would check to see if the `__add` field exists and is defined to be a function inside of an associated metatable, calling it if it exists, and throwing a runtime error otherwise. This would require some further implementation though

6.3 VarArgs

Lua supports a syntax like the following:

```
function f(x, y, ...)
    local vars = {...}
end
```

The function f can take any number of arguments, and any excess arguments are placed inside of the array `vars`. I currently do not have support for this. It would some re-engineering to do, since my compiler currently assumes all local parameters are stored inside a function's stack frame. To support this I would likely need an extra parameter pointing somewhere in the heap that contains the excess variables, and the syntax ... would need to evaluate to an expression list containing each element in the array. This is doable, but would require some time to implement.

6.4 Standard Library and Primitive Functions

There are a number of primitive functions and standard library functions that are left unimplemented. This was due to time constraints over anything else. My compiler supports functions to have bodies

that are defined with custom assembly, this is how I implemented the `print` primitive. One could easily implement these functions using custom bodies, calling out to JavaScript if necessary.

6.5 Coroutines

One of Lua's basic types is thread, which is used to implement coroutines. I am unsure of how I would begin to implement coroutines. The intent of WASM is that it is safety, and as such, it has no support for GOTO statements. All control flow in WASM is controlled by functions, blocks (which can be jumped to the end of), and loops (which can be jumped to the beginning of). Hence, any use of coroutines would need to be translated at the AST level to something describable by these primitives; I am currently unaware of any algorithms to do this.

6.6 Error Checking

At the moment, any runtime errors simply result in undefined behaviour. The compiler could instead have the client provide an imported `error` function, which would then be called whenever a runtime error occurs. Most of the runtime errors occur due to invalid typing: for example, a typo in the name of a variable could cause it to be nil, which would then error if used inside of a binary operation, function call, or a table index, etc. We would simply require code generation to use type checks before any type of operation occurs, calling the imported `error` function if the types are invalid. This would not be difficult to support.

7 TAKEAWAYS

WebAssembly is an interesting target to compile for. I am as of yet not convinced of the benefit of the stack machine architecture compared to register architectures. After all, WASM's support for globals essentially allows one to define as many pseudo-registers as one might want. One of WASM's goals, it seems, is to be well-typed and safe: functions have to specify their return types, the `wat2wasm` tool gives errors if the number of arguments or types of arguments to an instruction are incorrect, and there is no jumping to arbitrary locations. From a compiler perspective though, I did not really see these benefits. With WASM only supporting 32/64 bit integers and floating point numbers as types, I ended up doing most of my computation inside of linear memory rather than on the stack machine. This resulted in a larger number of instructions as I was frequently loading and storing. Furthermore, I didn't receive any type safety benefits given that most of the manipulations I was doing were on 32 bit integers that I was interpreting as pointers to any one of my own types. One could argue that if my compiler made better use of locals I could decrease the number of instructions, which is definitely true. However, this would complicate garbage collection since local variables (unlike globals) cannot be accessed by the JavaScript runtime, so there's greater risk of storing an old reference to a pointer that has been moved. Given that the entire point of WASM modules is that they run over the web, and hence bandwidth is a primary concern, it seems strange that they would choose an architecture that seems to naturally lead itself to a larger number of instructions.

8 CONCLUSION

This has been an enjoyable project to work on. I have learned a lot on the internal workings of both Lua and WebAssembly by building this compiler, and am proud of what I managed to achieve over this term.

REFERENCES

MDN Contributors. 2022. *Understanding WebAssembly text format.* https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

Roberto Ierusalimschy. 2004. *8.5 – Error Messages and Tracebacks.* https://www.lua.org/pil/8.5.html

Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. 2015. *Lua 5.3 Reference Manual.* https://www.lua.org/manual/5.3/manual.html