1

Preparation

Word budget: ~2500–3000 words

Describe the work undertaken before code was written.
Talk about revision control strategy, licensing of any libraries I used

In this chapter I describe my research into the WebAssembly architecture and the Relooper algorithm. I highlight the main features of Rust, the programming language I learned for this project, and my rationale for using it.

I also discuss the requirements of the project, and describe the software engineering approach I employed. Finally I outline my starting point and previous experience.

1.1 WebAssembly

Before starting to write the compiler, I extensively researched the WebAssembly specification [1], to gain a deep understanding of the binary code the project aims to generate. The following sections provide a high-level overview of the instruction set architecture.

1.1.1 Primitive values

The primitive value types supported by WebAssembly are outlined in Table 1.1, and described in more detail below.

Integers can be interpreted as either signed or unsigned, depending on the operations applied to them. They will also be used to store data such as booleans and memory addresses[1] Integer literals are encoded in the program using the LEB128 variable-length encoding scheme [2]. Listing 1.1 shows how to encode an unsigned integer. The algorithm is almost the same for signed integers (encoded in two's complement); the only difference is that we sign-extend to a multiple of 7 bits, rather than zero-extend. When decoding, the decoder needs to know if the number is signed or unsigned, to know whether to decode it as a two's complement number or not. This is why WebAssembly has signed and unsigned variants of some instructions.

Floating-point literals are encoded using the IEEE 754-2019 standard [3]. This is the same standard used by many programming languages, including Rust, so the byte representation will not need to be converted between the compiler and the WebAssembly binary.

[1]I.e. addresses within WebAssembly's sandboxed linear memory space, rather than function addresses etc.

Type	Constructor	Bit width
Integer	`i32`	32-bit
	`i64`	64-bit
Float	`f32`	32-bit
	`f64`	64-bit
Vector	`v128`	128-bit
Reference	`funcref`	Opaque
	`externref`	

Table 1.1: WebAssembly primitive types.

```
function LEB128EncodeUnsigned($n$)
    Zero-extend $n$ to a multiple of 7 bits
    Split $n$ into groups of 7 bits
    Add a 0 bit to the front of the most significant group
    Add a 1 bit to the front of every other group
    return bytes in little-endian order
end function
```

Listing 1.1: Pseudocode for the LEB128 encoding scheme (for unsigned integers).

WebAssembly also provides vector types and reference types. Vectors can store either integers or floats: in either case, the vector is split into a number of evenly-sized numbers. Vector instructions exist to operate on these values. `funcref` values are pointers to WebAssembly functions, and `externref` values are pointers to other types of object that can be passed into WebAssembly. These would be used for indirect function calls, for example. This project does not have a need for using either of these types; the C language doesn't have vector types, and although a compiler could create them, mine will not. Additionally I do not support function references in the scope of this project. I will only use the four main integer and float types.

1.1.2 Instructions

WebAssembly is a stack-based architecture; all instructions operate on the stack. For binary instructions that take their operands from the stack, the first operand is the one that was pushed to the stack first, and the second operand is the one pushed to the stack most recently (see Listing 1.2).

All arithmetic instructions specify the type of value that they expect. In Listing 1.2, we put two `i32` values on the stack, and use the `i32` variant of the `sub` instruction. The module would fail to instantiate if the types did not match. Some arithmetic instructions have signed and unsigned variants, such as the less-than instructions `i32.lt_u` and `i32.lt_s`. This is the case for all instructions where the signedness of a number would make a difference to the result.

WebAssembly only supports structured control flow, in contrast to the unstructured control flow found in most instruction sets that feature arbitrary jump instructions. There are three types of block: **`block`**, **`loop`**, and **`if`**. The only difference between **`block`** and **`loop`** is the semantics of branch instructions. When referring to a **`block`**, `br` will jump to the end of it, and when referring to a

```
i32.const 10 ;; first operand
i32.const 2  ;; second operand
i32.sub
```

Listing 1.2: WebAssembly instructions to calculate `10 - 2`.

`loop`, `br` will jump back to the start. This is analogous to `break` and `continue` in C, respectively. It is worth noting that `loop` doesn't loop back to the start implicitly; an explicit `br` instruction is required. `if` blocks, which may optionally have an `else` block, conditionally execute depending on the value on top of the stack. A `br` instruction inside an `if` block behaves in the same way as inside a `block`; it will jump to the end of it.

1.1.3 Modules

A module is a unit of compilation and loading for a WebAssembly program. There is no distinction between a 'library' and a 'program' such as there is in other languages; there are only modules that export functions to the instantiator (i.e. the runtime environment that instantiates the WebAssembly module). Modules are split up into different sections. Each section starts with a section ID and the size of the section in bytes, followed by the body of the section. Each of the sections is described in turn below.

The *type section* defines any function types used in the module, including imported functions. Each type has a type index, allowing the same type to be used by multiple functions.

The *import section* defines everything that is imported to the module from the runtime environment. This includes imported functions as well as memories, tables, and globals.

The *function section* is a map from function indexes to type indexes. This comes before the actual body code of the functions, because this allows the module to be decoded in a single pass. The type signature of all functions will be known before any of the code is read, so all function calls will be able to be properly typed without needing multiple passes.

A table is an array of function pointers, which can be called indirectly with `call_indirect`. It is necessary for WebAssembly to have a separate data structure for this, because functions live outside of the memory visible to the program; tables keep the function addresses opaque to the program, keeping the execution sandboxed. The *table section* stores the size limits of each table; any elements to initialise the tables with are stored in the element section.

The *memory section* is similar to the table section; is specifies the size limits of each linear memory of the program[1], in units of the WebAssembly page size (defined as 64 KiB).

The *global section* defines any global variables used in the program, including an expression to initialise them. Global variables can be either mutable or immutable.

The *export section* defines everything exported from the module to the runtime environment. Normally this is mainly function exports, however it can also include tables, memories, and global variables.

[1]In the current WebAssembly version, only one memory is supported, and is implicitly referenced by memory instructions.

The *start section* optionally specifies a function that should automatically be run when the module is initialised. This could be used to initialise a global variable to a non-constant expression.

The *element section* is used to statically initialise the contents of tables with function addresses. Segments can be active or passive. An active segment will be automatically initialise the table when the module is instantiated, whereas a passive segment needs to be explicitly loaded into a table with a `table.init` instruction.

The *data count section* is an optional additional section used to allow validators to use only a single pass. It specifies how many data segments are in the data section. If the data count section were not present, a validator would not be able to check the validity of instructions that reference data indexes until after reading the data section; but because this comes after the code section, it would require multiple passes over the module. This section has no effect on the actual execution.

The *code section* contains the instructions for each function body. All code in a WebAssembly program is contained in a function. Each function body begins by declaring any local variables, followed by the code for that function.

The *data section* is similar to the element section; it is used to statically initialise the contents of memory. This can be used, for example, to load string literals into memory.

1.2 The Relooper Algorithm

C allows arbitrary control flow, because it has **`goto`** statements. However, WebAssembly only has structured control flow, using **`block`**, **`loop`**, and **`if`** constructs as described above. Therefore, once the intermediate code has been generated, it needs to be transformed to only have structured control flow.

There are several algorithms that achieve this. The most naive solution would be to use a label variable and one big **`switch`** statement containing all the basic blocks of the program; the label variable is set at the end of each block, and determines which block to switch to next. However, this is very inefficient.

The first algorithm that solved this problem in the context of compiling to WebAssembly (and JavaScript, before that) was the Relooper algorithm, introduced by Emscripten in their paper on compiling LLVM to JavaScript [4]. In today's WebAssembly compilers, there are three general methods used to convert to structured control flow [5, 6]: Emscripten/Binaryen's Relooper algorithm, LLVM's CFGStackify, and Cheerp's Stackifier. All of these implementations, including the modern implementation of Relooper, are more optimised than the original Relooper algorithm, however therefore also more complex. The original Relooper algorithm is a greedy algorithm, and is well described in the paper, and is the one I decided to implement.

1.2.1 Input and Output

The Relooper algorithm takes a so-called 'soup of blocks' as input. Each block is a basic block of the flowgraph, that begins at an instruction label and ends with a branch instruction (Figure 1.1). There can be no labels or branch instructions other than at the start or end of the block. The

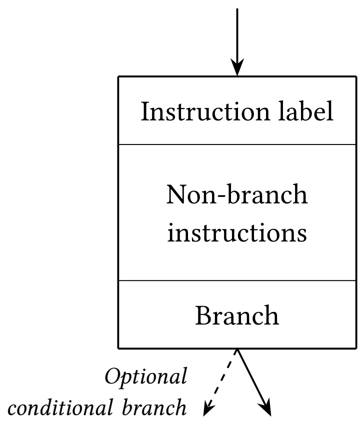

Figure 1.1: Structure of labels (Relooper algorithm input blocks).

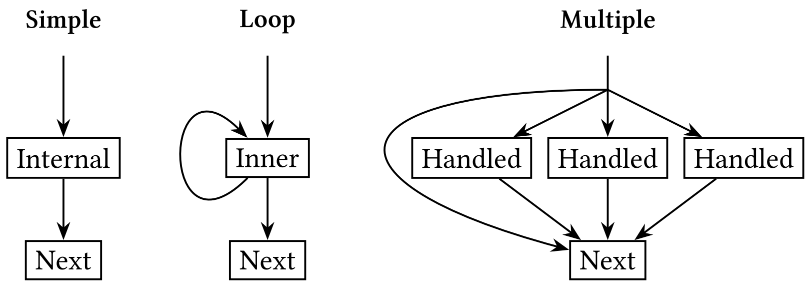

Figure 1.2: Structured blocks, generated by the Relooper algorithm.

branch instruction may either be a conditional branch (i.e. `if (...) goto x else goto y`), or an unconditional branch.

To avoid overloading the term *block*, these input blocks are referred to as *labels*.

The algorithm generates a set of structured blocks, recursively nested to represent the control flow (Figure 1.2). *Simple* blocks represent linear control flow; they contain an *internal* label, which contains the actual program instructions. *Loop* blocks contain an *inner* block, which contains all the labels that can possibly loop back to the start of the loop, along some execution path. *Multiple* blocks represent conditional execution, where execution can flow along one of several paths. They contain *handled* blocks to which execution can pass when we enter the block. All three blocks have a *next* block, where execution will continue.

1.2.2 Algorithm

Before describing the algorithm, I define the terms used. *Entries* are the subset of labels that can be immediately reached when a block is entered. Each label has a set of *possible branch targets*, which are the labels it directly branches to. It also has a set of labels it can *reach*, known as the *reachability* of the label. This is the transitive closure of the possible branch targets; i.e. all labels that can be reached along some execution path. Labels can always reach themselves.

Given a set of labels, and set of entries, the algorithm to create a block is as follows:

$\boxed{1}$ Calculate the reachability of each label.

$\boxed{2}$ If there is only one entry, and execution cannot return to it, create a simple block with the entry as the internal label.

- Construct the next block from the remaining labels; the entries for the next block are the possible branch targets of the internal label.

$\boxed{3}$ If execution can return to every entry along some path, create a loop block.

- Construct the inner block from all labels that can reach at least one of the entries.
- Construct the next block from the remaining labels; the entries for the next block are all the labels in the next block that are possible branch targets of labels in the inner block.

$\boxed{4}$ If there is more than one entry, attempt to create a multiple block. (This may not be possible.)

- For each entry, find any labels that it reaches that no other entry reaches. If any entry has such labels, it is possible to create a multiple block. If not, go to Step $\boxed{5}$.
- Create a handled block for each entry that uniquely reaches labels, containing all those labels.
- Construct the next block from the remaining labels. The entries are the remaining entries we didn't create handled blocks from, plus any other possible branch targets out of the handled blocks.

$\boxed{5}$ If Step $\boxed{4}$ fails, create a loop block in the same way as Step $\boxed{3}$.

The Emscripten paper outlines a proof that this greedy approach will always succeed [4, p. 10]. The core idea is that we can show that (1) whenever the algorithm terminates, the block it outputs is correct with respect to the original program semantics; (2) the problem is strictly simplified every time a block is created, therefore the algorithm must terminate; and (3) the algorithm is always able to create a block from the input labels. Point (3) lies in the observation that if we reach Step $\boxed{5}$, we must be able to create a loop block. This holds because if we could not create a loop block, we would not be able to return to any of the entries; however, in that case it would be possible to create a multiple block in Step $\boxed{4}$, because the entries would uniquely reach themselves.

The algorithm replaces the branch instructions in the labels as it processes them. When creating a loop block, branch instructions to the start of the loop are replaced with a **`continue`**, and any branches to outside the loop are replaced with a **`break`** (all the branch targets outside the loop will be in the next block). When creating a handled block, branch instructions into the next block are replaced with an **`endHandled`** instruction. Each of these instructions is annotated with the ID of the block they act on, because blocks may be nested. Note that functionally, an **`endHandled`** instruction is equivalent to a **`break`**, but I chose to keep the two distinct because they store IDs of different block types.

To direct control flow when execution enters a multiple block, Relooper makes use of a label variable. Whenever a branch instruction is replaced, an instruction is inserted to set the label variable to the label ID of the branch target. Handled blocks are executed conditional on the value of the label variable. This will generate some overhead of unnecessary instructions, since most of the time the label variable is set, it will not be checked. However, later stages of the compiler pipeline are able to optimise this away.

1.3 Rust

I learned the Rust programming language [7] for this project. I chose Rust because it is performant and memory efficient. It has a rich type system with pattern matching, which is well suited to writing a compiler. It uses the concept of a borrow checker rather than a garbage collector or other memory management system, which eliminates runtime overhead while guaranteeing memory safety at compile time.

I made use of Rust's excellent online documentation [8, 9], using it to become familiar with the language. The borrow checker was the main new feature of the language to learn. At its core are the concepts of *ownership* and *borrowing*. The main rule is that every value has exactly one owner at any time. When the owner goes out of scope, the value is automatically deallocated; this takes the place of the garbage collector found in other languages. When using a value, it can either be *moved* or *borrowed*. A move will make the new variable the owner of the value, and invalidate the old variable. Borrowing, on the other hand, allows a value to be used without taking ownership; it creates a reference that points to the owned value. There can either be many immutable borrows of a value, or a single mutable borrow. When the borrower is done with the value, it is given back to the owner.

1.3.1 Requirements Analysis

The requirements for this project were clear from the project description. A C to WebAssembly compiler takes C source code as input, and generates WebAssembly binary code that can be instantiated and executed from JavaScript.

The compiler consists of a pipeline of stages: a front end, middle end, and back end. The front end takes C source code, and outputs an abstract syntax tree (AST). The middle end takes the AST and transforms it to an intermediate representation (IR), in this case three-address code. The back end takes the IR and generates WebAssembly instructions, writing them to a binary file. Each stage of the compiler pipeline is distinct, with defined data models connecting them.

The stages described are the core requirements of the project. Optimisations to the compiler were set as extensions to the project, once the core objectives had been completed. These fall within the middle and back ends, transforming the IR and the target code.

1.4 Project Strategy

TODO put short para here as section overview

1.4.1 Software Engineering Methodology

I structured my project using the incremental model [10, sec. 3.3.1], which builds upon the waterfall model [10, sec. 3.2]. The project is broken down into 'increments', which are individual sections of functionality. Each increment is developed consecutively, using the waterfall model to design, implement, and test each section before moving on to the next one.

Each stage of the compiler pipeline was a separate increment. Subsequent stages of the project each depended on the deliverable from the previous one, making this model a natural fit, because it ensured a stage was working correctly before building upon it.

1.4.2 Testing

My testing strategy in the incremental model was to verify the correct functioning of each section, as much as possible, before moving onto the next section. I wrote a suite of test programs designed to use different language constructs. After completing a stage of the compiler pipeline, I carefully examined the output of compiling each test program to check its correctness.

Once the compiler could generate an AST, I wrote a function to reconstruct the source code from the AST. I compared this output against the original source to check that the program semantics had been maintained.

Once the entire pipeline was complete, I used GCC [11] as a reference compiler; I compiled my test programs with both my compiler and GCC, and verified that the resulting binaries produced the same output.

1.5 Starting Point

The *Compiler Construction* course of the Tripos was my starting point for my knowledge of compilers. The Part II course *Optimising Compilers* was useful in extending my project with optimisations, though the optimisations I implemented were not all included in the course.

I had familiarity with the C language from the *Programming in C and C++* course, as well as from previous personal projects. Additionally, I had experience writing JavaScript and Python from personal projects. I gained all my knowledge of WebAssembly and Rust through independent research.

1.6 Tools Used

The code for the body of the compiler was written in Rust (see Section 1.3). I wrote the runtime environment in JavaScript [12], using Node.js [13], which is necessary to interface with the WebAssembly binary. Additional development scripts, for automated testing and profiling, were written in Python [14], because of the ease of use, and my familiarity with libraries such as Matplotlib [15].

I used the LALRPOP parser generator library [16] to generate parsing code from the abstract grammar of C. Other parser generator libraries are available for Rust: notably, `nom` and `pest`. `nom` is more suited to parsing binary file formats rather than source code. `pest` is another popular choice, however it separates the formal grammar and AST generation, making the code more complicated. LALRPOP provides an intuitive and powerful approach, where each grammar rule contains both the grammar specification and code to generate the corresponding AST node.

I used Git [17] for version control, regularly pushing the repository to GitHub [18] as an off-site backup. I used the CLion IDE [19] as my development environment, using the Intellij Rust plugin [20]. CLion has excellent support for C, and the plugin enables native Rust support, including code analysis and debugging.