Introduction

Background and Motivation

Increasingly in modern society, more and more applications are shifting to cloud computing as one of the primary ways of interacting with computers. We are experiencing a transition away from traditional native applications and towards performing the same tasks in an online environment. However, the standard approach of building web apps with JavaScript fails to deliver the performance necessary for many intensive applications.

WebAssembly aims to solve this problem by bringing near-native performance to the browser space. It is a virtual instruction set architecture, which executes on a stack-based virtual machine. Per the Introduction section of the WebAssembly specification [1], it is designed to have "fast, safe, and portable semantics" and an "efficient and portable representation". The next two paragraphs expand on what this entails.

The semantics are designed to be able to be executed efficiently across different hardware, be memory safe (with respect to the surrounding execution environment), and to be portable across source languages, target architectures, and platforms.

The representation is designed with the primary target of the web in mind. It is designed to be compact and modular, allowing it to be efficiently transmitted over the Internet without slowing down page loads. This also includes being streamable and parallelisable, which means it can be decoded while still being received.

Survey of Related Work

Preparation

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.

Primitive values

The primitive value types supported by WebAssembly are outlined in Table 1, and described in more detail below.

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: WebAssembly primitive types.

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]. ?? 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.

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.

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 ??).

All arithmetic instructions specify the type of value that they expect. In ??, 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 `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.

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.

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.

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

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

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.

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 (??). There can be no labels or branch instructions other than at the start or end of the block. The 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 (??). *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.

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:

1. Calculate the reachability of each label.
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.
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.

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 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.

5 If Step 4 fails, create a loop block in the same way as Step 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 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 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.

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.

Project Strategy

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.

Software Engineering Methodology

Testing

Starting Point

Knowledge and experience

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.

Tools Used

The code for the body of the compiler was written in Rust (see ??). I wrote the runtime environment in JavaScript [10], using Node.js [11], which is necessary to interface with the WebAssembly binary. Additional development scripts, for automated testing and profiling, were written in Python [12], because of the ease of use, and my familiarity with libraries such as Matplotlib [13]. I used the LALRPOP parser generator library [14] to generate parsing code from the abstract grammar of C.

I used Git [15] for version control, regularly pushing the repository to GitHub [16] as an off-site backup. I used the CLion IDE as my development environment, using the Intellij Rust plugin [17]. CLion has excellent support for C, and the plugin enables native Rust support, including code analysis and debugging.

Implementation

Repository Overview

I developed my project in a GitHub repository, ensuring to regularly push to the cloud for backup purposes. This repository is a monorepo containing both my research and documentation along with my source code.

The high-level structure of the codebase is shown below. All the code for the compiler is in the **src/** directory. The other directories contain the runtime environment code, skeleton standard library implementation, and tests and other tools.

src/	Compiler source code.
program_config/	Compiler constants and runtime options data structures.
front_end/	Lexer, parser grammar, AST data structure.
middle_end/	IR data structures, definition of intermediate instructions. Converting AST to IR.
middle_end_optimiser/	Tail-call optimisation and unreachable procedure elimination.
relooper/	Relooper algorithm.
back_end/	Target code generation stage.
wasm_module/	Data structures to represent a WebAssembly module.
dataflow_analysis/	Flowgraph generation, dead code analysis, live variable analysis, clash graph.
stack_allocation/	Different stack allocation policies.
data_structures/	Interval tree implementation that I ended up not using.
preprocessor.rs	C preprocessor.
id.rs	Trait for generating IDs used across the compiler.
lib.rs	Contains the main run function.
runtime/	NodeJS runtime environment.
headers/	Header files for the parts of the standard library I implemented.
tools/	
profiler.py	Plot stack usage profiles.
testsuite.py	Test runner script.
tests/	Automated test specifications.

System Architecture

?? describes the high-level structure of the project. The front end, middle end, and back end are denoted by colour.

Each solid box represents a module of the project, transforming the input data representation into the output representation. The data representations are shown as dashed boxes.

I created my own AST representation and IR, which are used as the main data representations in the compiler.

Front End

The front end of the compiler consists of the lexer and the parser; it takes C source code as input and outputs an abstract syntax tree. I wrote a custom lexer, because this is necessary to support `typedef` definitions in C. I used the LALRPOP parser generator [14] to convert the tokens emitted by the lexer into an AST.

Preprocessor

I used the GNU C preprocessor (`cpp`) [18] to handle any preprocessor directives in the source code, for example macro definitions. However, since I do not support linking, I removed any `#include` directives before running the preprocessor and handled them myself.

For each `#include <name.h>` directive that is removed, if it is one of the standard library headers that I support, the appropriate library code is copied into the source code from `headers/<name>.h`. One exception is when the name of the header file matches the name of the source program, in which case the contents of the program's header file are inserted into the source code, rather than finding a matching library.

After processing `#include` directives, the compiler spawns `cpp` as a child process, writes the source code to its stdin, and reads the processed source code back from its stdout.

Lexer

The grammar of the C language is mostly context-free, however the presence of `typedef` definitions makes it context-sensitive [19, Section 5.10.3]. For example, the statement `foo (bar);` can be interpreted in two ways:

- As a variable declaration, if `foo` has previously been defined as a type name[1]; or
- As a function call, if `foo` is the name of a function.

This ambiguity is impossible to resolve with the language grammar. The solution is to preprocess the `typedef` names during the lexing stage, and emit distinct type name and identifier tokens to the

[1] The brackets will be ignored.

parser. Therefore, I implemented a custom lexer that is aware of the type names that have already been defined at the current point in the program.

The lexer is implemented as a finite state machine. ???? highlight portions of the machine; the remaining state transition diagrams can be found in ??. The diagrams show the input character as a regular expression along each transition arrow. (Note: in a slight abuse of regular expression notation, the dot character '`.`' represents a literal full stop character and the backslash character '\' represents a literal backslash.) It is assumed that when no state transition is shown for a particular input, the end of the current token has been reached. Transition arrows without a prior state are the initial transitions for the first input character. Node labels represent the token that will be emitted when we finish in that state.

The finite state machine consumes the source code one character at a time, until the end of the token is reached (i.e. there is no transition for the next input character). Then, the token corresponding to the current state is emitted to the parser. Some states have no corresponding token to emit because they occur part-way through parsing a token; if the machine finishes in one of these states, this raises a lex error. (In other words, every state labelled with a token is an accepting state of the machine.) For tokens such as number literals and identifiers, the lexer appends the input character to a string buffer on each transition and when the token is complete, the string is stored inside the emitted token. This gives the parser access to the necessary data about the token, for example the name of the literal.

If, when starting to lex a new token, there is no initial transition corresponding to the input character, then there is no valid token for this input. This raises a lex error and the compiler will exit.

?? shows the finite state machine for lexing number literals. This handles all the different forms of number that C supports: decimal, binary, octal, hexadecimal, and floating point. (Note: the states leading to the ellipsis token are shown for completeness, even though the token is not a number literal, since they share the starting dot state.)

?? shows the finite state machine for lexing identifiers and **`typedef`** names. This is where we handle the ambiguity introduced into the language. Every time we consume another character of an identifier, we check whether the current name (which we have stored in the string buffer) matches either a keyword of the language or a **`typedef`** name we have encountered so far. (Keywords are given a higher priority of matching.) If a match is found, we move to the corresponding state, represented by the ϵ transitions (since no input is consumed along these transitions). When we reach the end of the token, the three states will emit the corresponding token, either an identifier, keyword, or **`typedef`** name token respectively. The lexer stores the **`typedef`** names that have been declared so far so that it can emit the correct type of token for future names.

Parser

I used the LALRPOP parser generator [14] to generate parsing code from the input grammar I wrote. It generates an AST that I defined the structure of. Microsoft's C Language Syntax Summary [20] and C: A Reference Manual [19] were very useful references to ensure I captured the subtleties of C's syntax when writing my grammar. My grammar is able to parse all of the core features of the C language, omitting some of the recent language additions. I chose to make my parser handle a larger subset of C than the compiler as a whole supports; the middle end rejects or ignores nodes of

the AST that it doesn't handle. For example, the parser handles storage class specifiers (e.g. **`static`**) and type qualifiers (e.g. **`const`**), and the middle end simply ignores them.

Dangling Else Ambiguity

A naive grammar for C (??) contains an ambiguity around **`if`**/**`else`** statements [19, sec. 8.5.2]. C permits the bodies of conditional statements to be written without curly brackets if the body is a single statement. If we have nested **`if`**/**`else`** statements that do not use curly brackets, it can be ambiguous which **`if`** an **`else`** belongs to. This is known as the dangling else problem [21].

An example of the dangling else problem is shown in ??. According to the grammar in ??, there are two possible parses of this program. Either the **`else`** belongs to the inner or the outer **`if`** (??).

C resolves the ambiguity by always associating the **`else`** with the closest possible **`if`**. We can encode this into the grammar with the concept of 'open' and 'closed' statements [22]. ?? shows how I introduce this into my grammar for **`if`**/**`else`** statements. All other forms of statement must also be converted to the new structure. Any basic statements, i.e. statements that have no sub-statements, are added to **`closed-stmt`**. All other statements that have sub-statements, such as **`while`**, **`for`**, and **`switch`** statements, must be duplicated to both **`open-stmt`** and **`closed-stmt`**.

A closed statement always has the same number of **`if`** and **`else`** keywords (excluding anything between a pair of brackets, because bracket matching is unambiguous). Thus, in the second alternative of an **`open-stmt`**, the **`else`** terminal can be found by counting the number of **`if`**s and **`else`**s we encounter since the start of the **`closed-stmt`**; the **`else`** belonging to the **`open-stmt`** is the first **`else`** once we have more **`else`**s than **`if`**s.

If we allowed open statements inside the **`if`** clause of an **`open-stmt`**, then **`open-stmt`** and **`closed-stmt`** would no longer be disjoint, and the grammar would be ambiguous. This is because we wouldn't be able to use the above method for finding the **`else`** that belongs to the outer **`open-stmt`**.

LALRPOP Parser Generator

I chose the LALRPOP parser generator because it builds up the AST as it parses the grammar. This is in contrast to some of the other available libraries, which separate the grammar code and the code that generates the AST. LALRPOP provides an intuitive and powerful approach. Each grammar rule contains both the grammar specification and code to generate the corresponding AST node.

?? is an example of the LALRPOP syntax for addition expressions. The left-hand side of the `=>` describes the grammar rule, and the right-hand side is the code to generate an `Expression` node. Terminals are represented in double quotes; these are defined to map to the tokens emitted by the lexer. Non-terminals are represented inside angle brackets, with their type and a variable name to use in the AST generation code.

LALRPOP also allows macros to be defined, which allow the grammar to be written in a more intuitive way. For example, I defined a macro to represent a comma-separated list of non-terminals (??). The macro has a generic type `T`, and automatically collects the list items into a `Vec<T>`, which can be used by the rules that use the macro.

String Escape Sequences

The parser had to handle escape sequences in strings. I implemented this by first creating an iterator over the characters of a string, which replaces escape sequences by the character they represent as it emits each character. When the current character is a backslash, instead of emitting it straight away, the iterator consumes the next character and emits the character corresponding to the escape sequence. I wrapped this in an `interpret_string` function that internally creates an instance of the iterator and collects the emitted characters back to a string.

Parsing Type Specifiers

Another feature of the C language is that type specifiers (`int`, `signed`, etc.) can appear in any order before a declaration. For example, `signed int x` and `int signed x` are equivalent declarations. To handle this, my parser first consumes all type specifier tokens of a declaration, then constructs an `ArithmeticType` AST node from them. It uses a bitfield where each bit represents the presence of one of the type specifiers in the type. The bitfield is the normalised representation of a type; every possible declaration that is equivalent to a type will have the same bitfield. The declarations above would construct the bitfield `0b00010100`, where the two bits set represent `signed` and `int` respectively. For each type specifier, the corresponding bit is set. Then, the bitfield is matched against the possible valid types, to assign the type to the AST node.

Middle End

The middle end takes an AST as input, and transforms it to intermediate code. It also runs the Relooper algorithm on the IR. Optimisations are run on the IR in this stage; they are described in ?? .

Intermediate Code Generation

I defined a custom three-address code IR (the instructions are listed in ??). The IR contains both the program instructions and necessary metadata, such as variable type information, the mapping of variable and function names to their IDs, etc. The instructions and metadata are contained in separate sub-structs within the main IR struct, which enables the metadata to be carried forwards through stages of the compiler pipeline while the instructions are transformed at each stage. In the stage of converting the AST to IR code, the instructions and metadata are encapsulated in the `Program` struct.

Many objects in the IR require unique IDs, such as variables and labels. I created a `Id` trait to abstract this concept, together with a generic `IdGenerator` struct (??). The ID generator internally tracks the highest ID that has been generated so far, so that the IR can create as many IDs as necessary without needing to know anything about their implementation. IDs are generated inductively: each ID knows how to generate the next one.

To convert the AST to IR code, the compiler recursively traverses the tree, generating three-address code instructions and metadata as it does so. At the highest level, the AST contains a list of statements. For each of these, the compiler calls `convert_statement_to_ir(stmt, program, context)`.

`program` is the mutable intermediate representation, to which instructions and metadata are added as the AST is traversed. `context` is the context object described in ?? , which passes relevant contextual information through the functions recursively.

The core of converting a statement or expressions to IR code is pattern matching the AST node, and generating IR instructions according to its structure, recursing into sub-statements and expressions. The case for a **`while`** statement is shown in ??; the labels and branches to execute a while loop are added, and the instructions to evaluate the condition and body are generated recursively. Other control-flow structures are similar.

Some statements took a little more care to ensure the semantic meaning of the program is preserved. For example, **`switch`** statements can contain **`case`** blocks in any order. Some blocks may fall-through to the next block, and there may be a **`default`** block. I handled this by first generating instructions for each case block, and storing them in a switch context until all blocks have been seen. We then put conditional branches to each case block at the start of the switch statement, followed by each of the block instructions. By doing this, there is no direct fall-through between any blocks; instead, a block that falls through to the next block will end in a branch instruction to the start of that block. This also allows **`default`** blocks to be easily handled; we just add an unconditional branch to the default block after all the conditional branches. If there is no **`default`** block, the conditional branches are followed by an unconditional branch to the end of the **`switch`** statement. ?? shows the structure of the generated instructions for **`switch`** statements.

Declaration statements have a lot of different cases, each of which I had to handle separately. For example, variables may be declared without being initialised with a value.

Function declarations only differ syntactically from other declarations by the type of the identifier; so we have to handle them in the same place. For function definitions (i.e. declarations plus body), we transform the body code and add a new function to the IR.

Arrays are complicated because there are multiple ways their length can be specified. It can be given explicitly in the declaration, or implicitly inferred from the length of the initialiser. To further complicate them, an explicit size can either be a static value or a variable that is only known at runtime (creating a variable-length array). To handle variable length arrays, an instruction is inserted to allocate space on the stack for it at runtime. Array and struct initialisers are handled by first allocating memory for the variable, then storing the value of each of the inner members.

Some expressions can be evaluated by the compiler ahead-of-time, for example array length expressions. I implemented a compile-time expression evaluation function that can handle arithmetic expressions and ternaries. Expressions are converted according to their structure, recursing into sub-expressions.

Context Object Design Pattern

Throughout the middle and back ends, I used a design pattern of passing a context object through all the function calls. For example, when traversing the AST to generate IR code, the `Context` struct in ?? is used to track information about the current context we are in with respect to the source program. For example, it tracks the stack of nested loops and switch statements, so that when we convert a **`break`** or **`continue`** statement, we know where to branch to.

In an object-oriented language, this would often be achieved by encapsulating the methods in an object and using private state inside the object. Rust, however, is not object oriented, and I found this approach to offer more modularity and flexibility. Firstly, the context information itself is encapsulated inside its own data structure, allowing methods to be implemented on it that gives calling functions access to exactly the context information they need. It also allows creating different context objects for different purposes. In the target code generation stage, the `ModuleContext` stores information about the entire module being generated, whereas the `FunctionContext` is used for each individual function being converted. The `FunctionContext` lives for a shorter lifetime than the `ModuleContext`, so being able to separate the data structures is ideal.

Types

The following types are supported by the IR, mirroring the types supported by the C language [19, ch. 5]. **`Ux`** and **`Ix`** types represent unsigned and signed x-bit integers, respectively. Enumeration types (enums) are supported; their values are encoded as **`U64`**s. I followed the standard implementation convention for the bit size of **`char`**s, **`short`**s, **`int`**s, and **`long`**s; 8, 16, 32, and 64 bits respectively[1].

$$
\begin{aligned}
T = \ & \texttt{I8} \mid \texttt{U8} \mid \texttt{I16} \mid \texttt{U16} \mid \texttt{I32} \mid \texttt{U32} \mid \texttt{I64} \mid \texttt{U64} \mid \texttt{F32} \mid \texttt{F64} \mid \texttt{Void} \\
& \mid \texttt{Struct}(T[]) \mid \texttt{Union}(T[]) \\
& \mid \texttt{Pointer}(T) \mid \texttt{Array}(T, size) \\
& \mid \texttt{Function}(T, T[], is\_variadic)
\end{aligned}
$$

I created a data structure to represent these types, along with methods for operations on those types.

I implemented the ISO C unary and binary conversions for types [19, pp. 174–176]. They are applied before unary and binary operations respectively. Unary conversion reduces the number of types an operand can be. Smaller integer types are promoted to **`I32`**/**`U32`** appropriately, and $\texttt{Array}(T, \_)$ types are converted to $\texttt{Pointer}(T)$. Binary conversions make sure that both operands to a binary operation are of the same type. First, the unary conversions are applied to each operand individually. Then, if both operands are an arithmetic type, and one operand is a smaller type than the other, the smaller type is converted to the larger type. This includes integer types being promoted to float types.

When transforming each AST node to intermediate code, the compiler checks that the types of the operands are valid (after unary/binary conversion) and stores the corresponding type of the result variable. For example, before generating a `t = a < b` instruction, the compiler checks whether `a` and `b` are comparable arithmetic types—if not, a compile error is thrown—and sets the type of `t` to **`I32`**[2]. This ensures that the IR passed to the back end is type safe.

The Relooper Algorithm

The Relooper algorithm is described in ?? . In this section I describe my implementation. The Relooper stage takes the IR as input, and transforming the instructions to a structure of Relooper blocks, but preserving the program metadata.

[1] The C specification doesn't specify the exact bit widths, only the minimum size.
[2] **`I32`** is used to represent booleans.

First, the intermediate code is 'soupified'. This is the process of taking the linear sequence of instructions and producing a set of basic blocks, which are the input to the Relooper algorithm itself. Each label starts with a `label` instruction and ends in a `branch`. The process of soupifying is as follows:

1. Remove any label fall-through. By this I mean any label that is not preceded by a branch, where control will flow to directly from the previous instruction. This is removed by inserting a `branch` instruction before each label, if one does not already exist, branching to that label.

   This will generate many redundant branch instructions, however they get removed when the Relooper blocks are transformed to target code.

2. Insert an unconditional branch after each conditional branch, to ensure that conditional branches do not occur in the middle of a block.

   ```
   br <l1> if ...    ▷ Original conditional branch
   br <l2>           ▷ New unconditional branch
   label <l2>        ▷ New label
   ...
   ```

3. If there is no label at the very start of the instructions, add one.

4. Merge any consecutive label instructions into a single label, updating branches to the labels accordingly.

5. Finally, divide the instructions into blocks by passing through the instructions sequentially and starting a new block at each label instruction.

Steps 1 to 4 transform the instruction sequence until it can be directly split into basic blocks. All control flow between basic blocks is made explicit.

For every function in the IR, the instructions are soupified and then the algorithm described in ?? is used to create a Relooper block. The `IdGenerator` struct described above is used to assign each loop and multiple block a unique ID.

As it creates blocks, the algorithm replaces branch instructions. In simple blocks, branches are replaced with an instruction to set the label variable to the ID of the branch target. For the inner block of a loop, all branches to the start of the loop are replaced with a `continue` instruction and all branches out of the loop are replaced with a `break` instruction.

Back End: Target Code Generation

Runtime Environment

Optimisations

Unreachable Procedure Elimination

Tail-Call Optimisation

Summary

Evaluation

In this chapter, I evaluate my project against my success criteria, showing that all the success criteria were achieved. In ?? I demonstrate that the compiler is correct using a variety of test programs. In ?? I evaluate the impact of the optimisations I implemented, particularly showing significant improvements in memory usage.

Success Criteria

The success criteria for my project, as defined in my project proposal, are:

- The program generates an abstract syntax tree from C source code.
- The program transforms the abstract syntax tree into an intermediate representation.
- The program uses the Relooper algorithm to transform unstructured to structured control flow.
- The program generates WebAssembly binary code from the intermediate representation.
- The compiler generates binary code that produces the same output as the source program.

All of my success criteria have been met. The first four criteria correspond to the main stages of the compiler pipeline, respectively. The correctness of this pipeline is verified by the correctness of the generated binary code. The generated binary would not be correct if any of the stages had not been successful.

I used a variety of test programs to verify the success of the fifth criteria. This is described in ?? .

Correctness Testing

I wrote a suite of test programs in C to evaluate the correctness and performance of my compiler. There were two types of program:

- 18 ‘unit test’ programs, which test a specific construct in the C language; and
- 11 ‘full programs’, which represent real workloads that the compiler would be used for.

Programs of the first type are not strictly unit tests by the standard definition. Unit tests verify the functionality of individual units of source code, in isolation from the rest of the application. Each test should test one particular behaviour of that unit, and should be independent from the rest of the program's functionality [23]. My test programs don't test an isolated behaviour of the compiler's source code. Instead, they test a single behaviour of the C source code that is being compiled (for example, dereferencing a pointer), verifying that the compiler pipeline maintains the correct behaviour. This allowed me to trace bugs to the units of code that transformed that particular construct.

Examples of 'full programs' include Conway's Game of Life [24], calculating the Fibonacci numbers (recursively), and finding occurrences of a substring in a string. Six of the test programs I used were sourced from Clib [25], which contains many small utility packages for C. Those that I used had no external dependencies, and were useful in verifying that my compiler worked for other people's code as well as my own. Clib is licensed under the MIT license, which permits use of the software "without restriction". The remaining five full programs and all 18 unit test programs were written by myself.

For all my tests, I used GCC[1] as the reference for correctness. I deemed a program to be correct if it produced the same output as when compiled with GCC. To facilitate this, I made liberal use of `printf`, to output the results of computations.

I wrote a test runner script to ensure that I maintained correctness as I continued to develop the compiler, fix bugs, and implement optimisations. This script read a directory of `.yaml` files, which described the path to each test program, and the arguments to run it with. It then compiled the program with both my compiler and with GCC, and compared the outputs. A test passed if the outputs were identical, and failed otherwise. The script reported which tests, if any, failed.

I also implemented some convenience features into the test script. The command-line interface takes an optional 'filter' argument, which can be used to run a subset of the tests whose name matches the filter. The script can also be used to run one of the test programs without comparing to GCC, printing to the standard output. This allows easier manual testing.

To prevent bugs from accidentally being introduced into my compiler, I set up the test suite to run automatically as a commit hook whenever I committed changes to the Git repository. This would prevent a commit from succeeding if any of the tests failed, allowing me to make corrections first. This ensured that the version of my project in source control was always correct and functioning.

Impacts of Optimisations

In the following sections, I will evaluate the impacts that the optimisations I implemented had. For unreachable procedure elimination, I will evaluate the reduction in code size. For tail-call optimisation and stack allocation optimisation, I will evaluate the effectiveness at reducing memory usage.

To evaluate the memory usage optimisations, I inserted profiling code when compiling the programs, to measure the size of the stack throughout program execution. When generating instructions that move the stack pointer, the compiler additionally inserts a call to `log_stack_ptr`, a func-

[1] GCC version 11.3.1.

tion imported from the JavaScript runtime. `log_stack_ptr` reads the current stack pointer value from memory and appends it to a log file. I wrote a Python script to visualise the resulting data. The plots show how the size of the stack grows and shrinks throughout the execution of the program. ???????? are generated using this method.

Unreachable Procedure Elimination

In the context of this project, unreachable procedure elimination mainly has benefits in removing unused standard library functions from the compiled binary. When a standard library header is included in the program, the preprocessing stage inserts the entire code of that library[1]. If the program only uses one or two of the functions, most of them will be redundant. Unreachable procedure elimination is able to safely remove these, resulting in a smaller binary.

I only implemented enough of the standard library to allow my test programs to run, so the impact of this optimisation is limited by the number of functions imported. If I were to implement more of the standard library, this optimisation would become more important.

The standard library header with the most functions that I implemented was `ctype.h`. This header contains 13 functions, of which a program might normally use two or three. This is where I saw the biggest improvement from the optimisation. Programs that used the `ctype` library saw an average file-size reduction of 4.7 kB.

The other standard library headers I implemented only contained a few functions, so the impact of this optimisation was much more limited. However, as mentioned above, if I implemented more of the standard library, I would see much more of an improvement.

Due to this difference in the standard library header files, and also to the fact that source programs can arbitrarily contain functions that are never used, it is not meaningful to calculate aggregate metrics across all my test programs. However, testing each program individually does verify that any functions that are unused are removed from the compiled binary.

Tail-Call Optimisation

My implementation of tail-call optimisation was successful in reusing the existing function stack frame for tail-recursive calls. The recursion is converted into iteration within the function, eliminating the need for new stack frame allocations. Therefore, the stack memory usage remains constant rather than growing linearly with the number of recursive calls.

One of the functions I used to evaluate this optimisation was the function in ?? below, that uses tail-recursion to compute the sum of the first n integers.

?? compares the stack memory usage with tail-call optimisation disabled and enabled. Without the optimisation, the stack size clearly grows linearly with n. When running the program with $n = 500$, a stack size of 46.3 kB is reached. When the same program is compiled with tail-call optimisation enabled, only 298 bytes of stack space are used; a 99.36 % reduction in memory usage. Of course, the reduction depends on how many iterations of the function are run. In the non-optimised case, the stack usage is $\mathcal{O}(n)$ in the number of iterations, whereas in the optimised case it is $\mathcal{O}(1)$.

[1] That is, the entirety of my skeleton implementation of that library, rather than the actual standard library code.

When testing with large n, the non-optimised version quickly runs out of memory space, and throws an exception. In contrast, the optimised version has no memory constraint on how many iterations can be run. It successfully runs 1 000 000 levels of recursion without using any more memory than for small n; even GCC fails to run that many.

Optimised Stack Allocation Policy

The stack allocation policy that I implemented was successful in reducing the amount of stack memory used.

From my test programs, the largest reduction in memory use was 69.73 % compared to the unoptimised program. The average improvement was 50.28 %.

?? shows the impact that this optimisation had on the different test programs. The full-height bars represent the stack usage of the unoptimised program, which we measure the optimised program against. The darker bars show the stack usage of the optimised program as a percentage of the original stack usage. Shorter bars represent a greater improvement (less memory is being used).

For programs that benefit from tail-call optimisation, I measured the effect of this optimisation on both the optimised and unoptimised versions. I did this because tail-call optimisation also affects how much memory is used, so it may have an impact on the effectiveness of this optimisation.

One of the main factors influencing the amount of improvement is the number of temporary variables generated. The more temporary variables generated, the larger each stack frame will be in the unoptimised version, and the more scope there is for the compiler to find non-clashing variables to overlap. Because temporary variables are generated locally for each instruction, the majority of them only have short-range dependencies. Only the variables that correspond to user variables have longer-range dependencies. Therefore the temporary variables offer the compiler more options of independent variables.

The result of this is that as a function increases in its number of operations, the number of temporary variables increases, and so does the scope for optimisation that the compiler is able to exploit.

We can see the effect of this directly when we compare the stack usage of the unoptimised and optimised versions of the same program. ?? shows the size of the stack over the execution of a test program that converts strings to upper, lower, or camel case. Since the main stack allocations and deallocations occur on function calls and returns respectively, each spike on the plot corresponds to a function call. We can use this to figure out which parts of the plot correspond to which part of the source program.

The program in turn calls `case_upper()`, `case_lower()`, and `case_camel()`, which corresponds to the three distinct sections of the plot.

`case_upper()` and `case_lower()` each make repeated calls to `toupper()` and `tolower()`, which corresponds to the many short spikes on the plot (one for each character in the string). Other than a **`for`** loop, they do not contain many operations, and therefore not many temporary variables are generated.

In contrast, `case_camel()` performs many more operations iteratively in the body of the function. ?? shows an extract of its body code. Even in this short section, more temporary variables are created than in the entire body of `case_upper()`. This results in the large spike at the end of ??.

Due to the differences in temporary variables described above, the compiler is able to optimise `case_camel()` much more than the other functions. This parallels the fact that `case_camel()` had the largest stack frame initially.

Another area that this optimisation has a large impact is for recursive functions. Since this optimisation reduces the size of each stack frame, we will see a large improvement when we have lots of recursive stack frames. ?? shows the size of the stack over the execution of calculating the Fibonacci numbers recursively. In this instance, the optimised stack allocation policy reduced the stack size of the program by 67.20 %.

Summary

The main objective of this project was to produce a compiler that generated correct WebAssembly binary code. Through the range of testing described above, I have shown that this objective was achieved, with the definition of correctness being that the generated program behaves in the same was as when compiled with GCC.

The objective of adding optimisations is to improve the performance of the compiled programs, while maintaining the semantic meaning of the program. The correctness of my optimisations was verified with the same test suite as was used to test the unoptimised compiler's correctness. In the previous sections, we have seen measurable evidence that the optimisations did improve performance. Therefore the optimisations were a success.

Conclusions

Project Summary

Lessons Learned

Further Work