VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Compiling Whiley for WebAssembly

Wei Hua

Supervisor: Dr David Pearce

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Whiley is a multi-paradigm programming language which supports Extended Static Checking through formal specification. At compile time, Whiley can identify common errors which are uncaught by a type checker, including division by zero, null reference and array out of bounds errors. WebAssembly is a portable binary code format supported by major browsers. WebAssembly is designed to enable high-performance applications on web pages. This project is about developing a Whiley Compiler back-end plugin which translates Whiley Intermediate Language into WebAssembly. This plugin can support almost all of Whiley syntax. It not only supports basic features like control flow and compound types, but also advanced features like recursive types, templates, lambda expressions. This document gives a general background of this project, the design of the solution, how we implement it. Finally, we compared performance between WebAssembly and JavaScript using WyBench benchmark suite on Node.js, where WebAssembly is 20%-30% faster than JavaScript.

Acknowledgments

I would like to thank Dr David Pearce, for all of the support and mentorship throughout the duration of this project. I would also like to my examiners for their feedback.

Contents

Figures

Chapter 1

Introduction

The Web now becomes a popular application platform across different operating systems and devices. JavaScript dominates web application development. For a long period, JavaScript was the only programming language running in web browsers. There are rational motivations for developing web applications in other programming languages. One could be that there are programs written in the other programming languages. Another reason might be the preference for another language. Hence, people develop compilation tools to translate programs in other languages into JavaScript. For example, `Whiley2JavaScript` which compiles Whiley into JavaScript, and `GHCJS` which compiles Haskell into JavaScript.

Though these solutions can bring new language features to web application development, people still face challenges in developing computationally expensive web applications due to the performance of JavaScript. To address this issue, Mozilla began developing ASM.js [19], which is a strict subset of JavaScript without objects, garbage collection and just-in-time compiler pauses. Hence, people can write web applications in C/C++ and manage the resources by themselves. ASM.js is still JavaScript which means parsing is still a costly task. WebAssembly was created for addressing this problem. WebAssembly is a binary instruction format for a stack-based virtual machine and is designed as a portable target for compilation of high-level languages. Now WebAssembly 1.0 has shipped with four major browsers, Google Chrome, Mozilla Firefox, Apple Safari and Microsoft Edge [3].

Whiley is a multi-paradigm programming language primarily developed by Dr David Pearce at Victoria University of Wellington. Whiley supports Extended static checking through formal specification. At compile time, Whiley can identify common errors which are uncaught by a type checker, including division by zero, null reference and array out of bounds errors [14].

This project aims to implement a back-end plugin of Whiley Compiler, called `Whiley2Wasm`, which can translate the Whiley Intermediate Language into WebAssembly. With this tool, the existing Whiley programs can be compiled into WebAssembly and run in browsers across operating systems and platforms. Programmers can develop new web applications with Whiley. They can employ the Whiley Compiler to verify programs and eliminate many bugs at compile time while the programs have good performance in browsers.

There are two formats of WebAssembly, Text Format and Binary Format. A WebAssembly program in Binary Format can be executed directly in browser engines, and Text Format is a human-readable format of Binary Format. The WebAssembly Binary Toolkit (WABT) is a toolkit that can convert WebAssembly between these two formats. For simplicity, the output of `Whiley2Wasm` will be Text Format.

We have tested our project on a test suite consisting 652 tested cases provided by Dr David Pearce. We have passed 94.1% test while 0.2% failed. We also skipped 5.7% test cases of which most are limitation in current version of Whiley Compiler. We compared

performance between WebAssembly and JavaScript using WyBench benchmark suite on Node.js, where WebAssembly is 20%-30% faster than JavaScript.

1.1 Contributions

The project offers the following contributions:

1. The design and implementation of a Whiley Compiler back-end plugin. Together with the Whiley Compiler front-end, it can compile Whiley programs into WebAssembly which enables Whiley programs to run in browsers.
2. An evaluation of the correctness of the project with a test suite consisting 652 unit testing cases. We additionally evaluate the performance of generated WebAssembly with WyBench benchmark test suite on Node.js.

1.2 Organisation

The remained of this report is structured as follows:

- Chapter 2 provides background information and related work regarding to WebAssembly, Whiley, compiling the other programming languages to WebAssembly.
- Chapter 3 discusses the architecture of Whiley2Wasm and design of implementing Whiley type system and other syntax.
- Chapter 4 provides details of translating Whiley to WebAssembly.
- Chapter 5 provides the results of unit testing and benchmark testing.
- Chapter 6 presents the project conclusion and possible future work.
- Appendix A provides the raw data from benchmark testing.

Chapter 2

Background and Related Work

This chapter provides background information regarding WebAssembly and Whiley. It describes WebAssembly instructions, functions, tables and WebAssembly memory. It introduces Whiley, with particular focus on Whiley's type system.

2.1 A Brief Overview of WebAssembly

In this section, we give a brief overview of WebAssembly. We primarily focus on the WebAssembly syntax [4] used for implementing Whiley in this project.

2.1.1 Stack Machine

WebAssembly is a binary instruction format for a stack-based virtual machine. Instructions pop values from the stack and push values back (similar to Java bytecode). Function invocations have their own stacks, and they have no access to other stacks. This is a safer way than the stack pointer approach used in some other programming languages [9].

2.1.2 Text Format with S-expressions

WebAssembly comes with two formats, Text Format and Binary Format. Text Format, based on S-expressions, is a human-readable representation of Binary Format. Text Format and Binary Format can be converted easily with the WebAssembly Binary Toolkit (WABT). In this project and this report, we primarily adopt the Text Format of WebAssembly for simplicity.

Listing 2.1 illustrates an example program of adding two numbers in WebAssembly. The whole program is an S-Expression where `i32.add` is the operator, and `(i32.const 1)` and `(i32.const 2)` are operands.

Listing 2.1: WebAssembly Text Format Example

```
(i32.add
  (i32.const 1)
  (i32.const 2)
)
```

2.1.3 Value Types and Packed Types

WebAssembly has four value types: 32-bit integer (i32), 64-bit integer (i64), 32-bit floating-point (f32) data and 64-bit floating-point data (f64) under IEEE 754-2008 standard. The instructions' input types and return types will be one of these four types. WebAssembly also

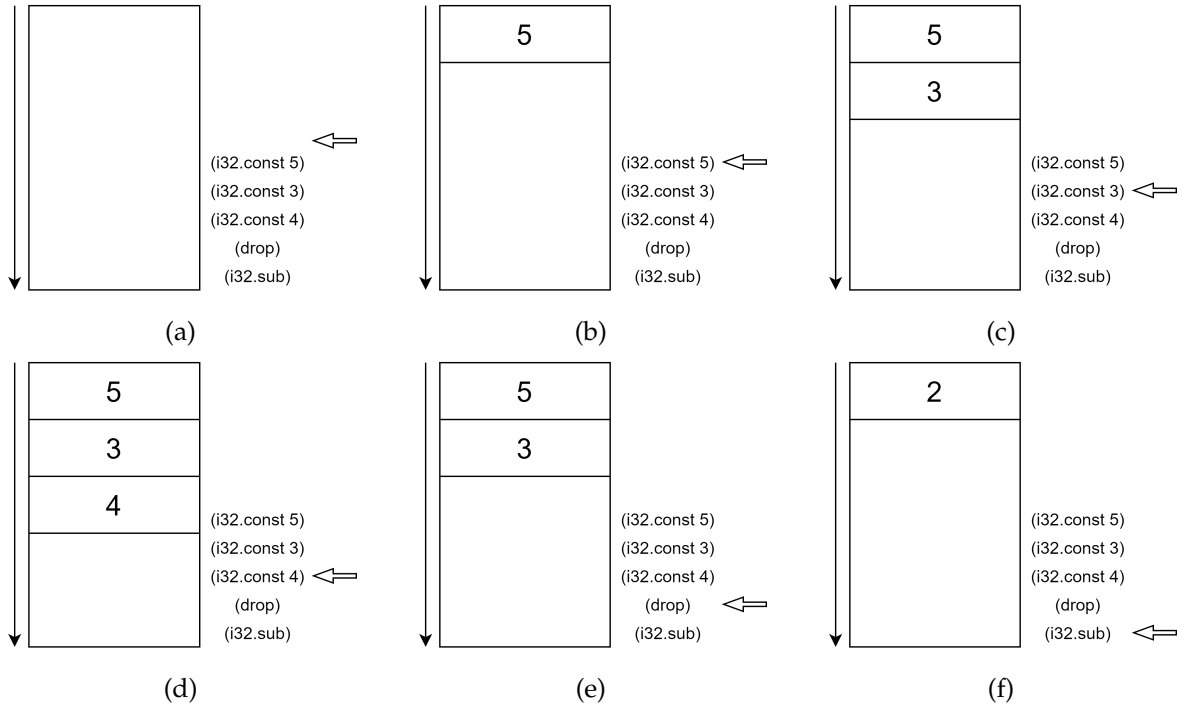

Figure 2.1: Example of WebAssembly Stack Machine

provides three packed types: 8-bit integer (i8), 16-bit integer (i16) and 32-bit integer (i32). WebAssembly can read from or story to memory as one of the three packed types. For example, we can store the low 8 bits of an integer to the memory as an i8 value, or load an i8 value from the memory and extend it into an i32 value.

There are no array types or record types in WebAssembly. Within this project, we use a pointer approach to implement array types and record types by ourselves.

2.1.4 Instructions

WebAssembly provides a wide range of instructions, including arithmetic operations, control flow, variable access, etc. WebAssembly is a typed language. Each instruction takes zero or more values from the stack and leaves zero or one values on the stack. The following explains three basic instructions (const, add and drop).

- `i32.const :: () -> (i32)`. `i32.const` pushes an i32 value onto the stack.
- `i32.sub :: (i32, i32) -> (i32)`. `i32.sub` pops two values from the stack as operands and pushes the result, an i32 value, onto the stack.
- `drop :: ($T) -> ()`. `drop` takes a value with any type from the stack and does not push any value onto the stack. Since this instruction accepts any types, it sometimes called value-polymorphic instruction [4].

Figure 2.1 illustrates how instructions work with the stack. It started with an empty stack. Then it pushed 5, 3, 4 on the stack in order. After that, `drop` instruction dropped the value on the top of the stack, which was 4 in our example. Finally, `i32.sub` instruction took two operands(5 and 3) from the stack and pushed the result back to the stack.

Instruction nesting. Though only control flow, function and module support nested instructions in Binary Format, Text Format supports nested instructions as operands for general Instructions. For example, Listing 2.2 and Listing 2.3 are equivalent. We primarily use the nested instructions in our project as it is close to the original source programs.

Listing 2.2: Add without nesting

```
(i32.const 1)
(i32.const 2)
(i32.add)
```

Listing 2.3: Add with nesting

```
(i32.add
  (i32.const 1)
  (i32.const 2)
)
```

Logical Operations and Comparisons. WebAssembly provides rich instructions for those two operation categories. For example, `i32.and`, `i32.eq` and `i32.or`. There is no Boolean type in WebAssembly and all Boolean values are represented as i32 where zero is for False and non-zero values for True. This convention is similar to the C programming language. However, instruction `and` and instruction `or` do not exhibit short-circuit semantics.

Local Variables and Global Variables. WebAssembly has local variables which are scoped within functions and global variables which are scoped within each module. All variables have to be declared before being used. Local variables have to be declared inside the function declaration but before the function body. Global variables have to be declared in the module. When declaring a global variable, the instruction for initializing the global variable must be provided as well. WebAssembly only accepts a `const` instruction or a `get_*` instruction to initialize the global variable. If a programmer wants to initialize a global variable with a complex expression, the start function (a function called automatically immediately after the module is initialized) is a good option. That is, we firstly initialize the global variable with a constant, and then overwrite it with the value of the complex expression in the start function.
WebAssembly provides four instructions named `get_local`, `get_global`, `set_local` and `set_global` to read from and write to local/global variables. To access a variable, we either use the variable index or the variable name as the immediate value for one of the four instructions. For example, `(get_local 0)` will return the value of the first local variable, and `(get_local $x)` will return the value of the local variable named `$x`.

Control Flow. WebAssembly provides three control flow instructions: `block`, `loop` and `if else`, alongside with `br` and `br_if` to exit the current flow. `br` and `br_if` cannot jump to arbitrary points in the program, instead, they can only jump to an enclosing control structure. The outer control structures of `br` or `br_if` are referred to indices from 0 to n - 1 where the innermost control structure is 0.

Listing 2.4 is a simple while loop in Whiley and Listing 2.5 is the equivalent translation in WebAssembly. In Listing 2.5, `loop`(line 2) and `block`(line 1) are inner control structure and outer control structure for both `br_if`(line 3) and `br`(line 14). Thus `(br_if 1)` will jump to the `block` and `(br 0)` will jump to the `loop`. Though `block` and `loop` look like control-flow statements in high-level programming languages, they are more like labels – a jump target – in Assembly language. Once the last instruction in the loop body was executed, WebAssembly will execute the instruction next to the `loop` rather than the first instruction in the `loop` body. The difference between `block` and `loop` is that when execution jumps back to `block`, the instruction executed next is the one following the `block`, while when execution jumps back to a `loop`, the instruction executed next is the first instruction from the `loop` body. In other words, `br` behaves like a `break` statement when its jump target is `block`, while it be-

haves like a `continue` statement when its jump target is `loop`.

Listing 2.4: Whiley while loop loop

```
while i < 3:
  i = i + 1
```

Listing 2.5: Whiley while loop loop

```
(block
  (loop
    (br_if 1
      (i32.ge_s
        (get_local $i)
        (i32.const 3)
      )
    )
    (i32.add
      (get_local $i)
      (i32.const 1)
    )
    (set_local $i)
    (br 0)
  )
)
```

2.1.5 Functions

A function consists of a function id, a function type, local variable declarations and a sequences of instructions. A Function is defined in the following format:

$$(func\ id^{?}\ type\ (t : local)^{*}\ (instruction)^{*})$$

A function type is defined by function parameters and a function return type. Currently, only single returns are supported. Listing 2.6 illustrates how functions are declared. The first function has function id `$f` which takes an `i32` value as a parameter and returns an `i32` value. The second function always returns 42. The second function has no function id which is valid in WebAssembly.

Listing 2.6: WebAssembly Function Declaration Example

```
(func $f (param $x i32) (result i32)
  (get_local $x)
  (return)
)

(func (result i32)
  (return (i32.const 42))
)
```

Each function defined in a module has a static index associated with it. Indices are monotonically-increasing values based on the order of functions defined in the module, starting with zero. For the functions shown in Listing 2.6, the index of function `$f` is 0 and the index of the second function is 1.

There are two ways to invoke functions, using the `call` instruction or the `call_indirect` instruction. Instruction `call` can invoke a function through function id or the static index of a function. If the callee has parameters, the caller should firstly push all parameters onto

the stack in order and then push the `call` onto the stack. The `call_indirect` instruction is always paired with a special structure `table` in WebAssembly, so we'll discuss them in a separate section.

Overall, functions in WebAssembly are very similar to other programming languages, but there are some differences referring to the purity of functions and nested functions. Purity for functions is defined as a function that always returns the same output for the same input and performs no side effects. In Whiley, functions refer to pure functions and methods to impure functions. However, functions in WebAssembly are impure, since instructions in function bodies can interact with the states outside the function scope (e.g. read/write values to the global variables). WebAssembly also does not support nested functions. Thus, a function cannot be defined within another function. This brings some challenges of implementing lambda functions in Whiley, especially lambda functions with captured variables.

2.1.6 Table and call_indirect

Tables in WebAssembly are arrays of function references. We can declare at most one table per module with the instruction `table`. To invoke a function whose reference is stored in a table, we can use the instruction `indirect_call` together with table index. Listing 2.7 is an example of using a table and the instruction `indirect_call`. We first create a table with only 1 slot and then we use the instruction `elem` to put the reference of function *f* into the first slot. In the function *main*, we use the instruction `call_indirect` to invoke function *f* indirectly through its reference stored in the table.

Listing 2.7: WebAssembly indirect_call example

```
(table 1 anyfunc)
(elem (i32.const 0) $f)

(func $f (result i32)
  (return
    (i32.const 42)
  )
)

(func $main
  (call_indirect
    (i32.const 0)
  )
)
```

2.1.7 Memory

Memory in WebAssembly is a byte array which can be accessed by memory access instructions together with a byte index. The indices start from 0 and extend up to the allocated memory size per module. Our implementation uses WebAssembly memory to implement `arrays` and `records`. The unit of memory is a WebAssembly page size which is always 64K bytes across all platforms. The instruction `memory.grow` can be used to get more pages.

All memory accesses are bounds checked. Accessing data out of bounds will cause an out of bounds access trap. Finally, memory space is separated from the global/local variables spaces. It is impossible to use memory access instructions to read/write any variables.

2.2 A Brief Overview of Whiley

In this section, we give a brief overview of Whiley, primarily focusing on the syntax implemented in this project [15].

2.2.1 Primitive types

- Whiley uses `int` to denote integers. This differs from other common programming languages, as integers in Whiley are unbounded. In other words, there is no integer overflow in Whiley.
- Whiley uses `bool` to denote boolean values. `bool` only has two values, `true` and `false`.
- Whiley use `byte` to denote 8-bit binary numbers. A sequence of 0 and 1 preceded by `0b` (e.g. `0b00001111`) denotes a binary literal in Whiley programs .

2.2.2 Arrays

Arrays are built-in in Whiley. Array types are declared as follows:

$$ArrayType \ ::== \ Type \ []$$

Arrays can be constructed in two ways:

- Array initialiser: $[e_1, \ e_2, ..., \ e_n]$ where e_i is the ith element of an array
- Array generator: $[e; n]$ which generates an array with length `n` and all elements are initialised with e.

We can access array element by $e_1[e_2]$ and we can also get the length of the array by $|e|$. Listing 2.8 illustrates the array operators.

Listing 2.8: Array example

```
// array initialiser
int[] arr1 = [1, 2, 3]
// array generator
bool[] arr2 = [true; 4]
// array element access
assert arr1[0] == 1
// array length
assert |arr2| == 4
```

Arrays in Whiley have **value semantics**. When an array variable is assigned to another variable, a deep copy of the array is assigned, and the two variables would hold two identical arrays. This differs from other programming languages, like Java or C. In Java, when an array variable is assigned to another, only a copy of the array reference is assigned, and the two variables will hold references to the same object.

2.2.3 Records

Records allow users to define a data type that combines data items of different types. Records are similar to structs in the C programming language, and we can use the `(.)` operator to access record fields. Listing 2.9 is an example of a record representing 3 dimensional vectors.

Listing 2.9: Record example

```
type vec3d is {int x, int y, int z}

method main():
   vec3d v = {x: 1, y: 2, z:1}
   assert v.x == 1
   assert v.y == 2
   assert v.z == 1
```

Whiley employs a **structural type system** where the equivalence of two record types is only determined by the records' actual structure. That is, two record types with different type names but the same structure are equivalent. In Listing 2.10, the type *vec3d* and the type *color* are equivalent, so the variable *c* can be used to initialise the variable *v*. Similar to arrays, records have value semantics as well.

Listing 2.10: Record types example

```
type vec3d is {int x, int y, int z}
type color is {int x, int y, int z}

method main():
   vec3d v = {x: 1, y: 2, z:1}
   color c = v
```

2.2.4 Open-records

Open-records allow users to define records with arbitrary fields. Listing 2.11 illustrates how open-records can be used in Whiley. The record $\{y: 1, x: 3\}$ is a subtype of *OpenRecord*, because it has the *int* type field *x*. We also can use a type test to check whether an open-record has certain type fields at run-time, such as r **is** {**int** y, **int** x}.

Listing 2.11: Open Records example

```
type OpenRecord is {int x, ...}

function getField(OpenRecord r) -> int:
   if r is {int y, int x}:
      return r.x + r.y
   else:
      return -r.x

public export method test() :
   OpenRecord r = {x: 1}
   assert getField(r) == -11
   r = {y: 1, x: 3}
   assert getField(r) == 4
   r = {y: "hello", x: 2}
   assert getField(r) == -2
```

2.2.5 Union types

In Whiley, a union type is defined as a union of component types. A variable defined as a union type can hold any value of one its component types. Listing 2.12 gives an example

of a union type. We first defined a union type called *intnull* which consists of the type `int` and the type `null`. In the *main* function, we declared a variable *x* with the type *intnull*, and initialised with it with an integer 3. After that, we assigned `null` to variable *x*.

Listing 2.12: Union types example

```
type intnull is int | null

method main():
  intnull x = 3
  x = null
```

Union types can be recursive. A typical case is that a component type of a union type refers to the union type itself. Recursive types are very helpful to represent recursive data structures, like binary-trees. Listing 2.13 is an example of recursive types. In the example, the type *btree* is a recursive type, the first component of the type *btree* is a record, and the type of the first field is the union type *btree*, etc.

Listing 2.13: Recursive types example

```
type btree is {btree left, btree right, int val} | leaf
type leaf is int

method main():
  btree x = 1
  btree y = 4
  btree tree = {left: x, right:y, val: 3}
```

2.2.6 Multiple returns and multiple assignments

Functions and methods can return multiple values. It is kind like of returning an anonymous record value. Correspondingly, Whiley provides destructuring syntax to assign multiple returned values to several variables in a single assignment statement. Listing 2.14 illustrates Euclidean division by using multiple returns.

Listing 2.14: Multiple return example

```
function div(int dividend, int divisor) -> (int quotient, int remainder):
  quotient = dividend / divisor
  remainder = dividend % divisor
  return quotient, remainder

method main():
  int quotient
  int remainder
  quotient, remainder = div(5, 2)
  assert quotient  == 2
  assert remainder == 1
```

Whiley also supports multiple assignment statements. Multiple values can assign to multiple variables at the same time. The left and right side must have the same number of elements. Listing 2.15 illustrates how to swap two variables in single multiple assignment statement.

Listing 2.15: Multiple assignment example

```
method main():
   int x = 1
   int y = 2
   x, y = y, x
   assert x == 2
   assert y == 1
```

2.2.7 Lambda expressions

A lambda expression in Whiley defines an anonymous function. The scope of a lambda expression is the scope of the function that defines it. If a lambda expression refers to some variables outside the expression, the value of the variable is captured when the lambda expression is created.

Listing 2.16: Multiple assignment example

```
type fun_t is function(int)->int

function add(int x) -> fun_t:
   fun_t fn = &(int y -> x + y)
   return fn

method main():
   fun_t addOne = add(1)
   fun_t addTwo = add(2)
   assert addOne(1) == 2
   assert addTwo(1) == 3
```

Listing 2.16 is an example of a lambda expression. The function *add* has an integer parameter x and returns an anonymous function. The generated anonymous function simply returns the sum of the value of *x* and its parameter. In the *main* function, *addOne* and *addTwo* are two functions generated by the function *add*. The function *addOne* always adds one to its input and the function *addTwo* always adds two to its input. As we can see, the values of the variable *x* are different within *addOne* and *addTwo*, as the value of x is determined/captured when the lambda is create at run-time.

2.2.8 Preconditions and Postconditions

Preconditions are conditions that must be true before the function is called. Postconditions are conditions that will be true when the function finished. The Whiley Compiler can verify whether the function implementation meets the given preconditions/postconditions at compile-time [15]. Listing 2.17 is a Whiley program with a precondition and a postcondition. The `requires` statement and `ensures` statement define the precondition and the postcondition. Given the precondition (`x >= 0`) and the implementation (`r = x + 1`), the Whiley Compiler can conclude that the postcondition (`r >= 1`) holds.

Listing 2.17: Example Whiley program with a precondition and a postcondition

```
function increment(int x) -> (int r)
// x must be an non-negative integer
requires x >= 0
// return must be a positive integer
ensures r >= 1:
   return x + 1
```

2.3 Related work

There are no existing tools that compile Whiley programs into WebAssembly directly. However, there are some projects related to compiling other programming languages (e.g. C/C++, Rust) to WebAssembly or to compiling Whiley to the other targets [7, 5, 6] (e.g. JVM, JavaScript, C).

`Emscripten` [20] is a well-known toolchain compiling C/C++ to WebAssembly. Emscripten was originally designed for ASM.js. It takes LLVM bytecodes as input and compiles them into JavaScript. More recently Emscripten started to support WebAssembly as well.

`Binaryen` [1] is compiler and toolchain infrastructure library for WebAssembly. What Binaryen mainly does is to compile its internal IR called `Binaryen IR` to WebAssembly in a fast, effective way. There are some projects using Binaryen to build compilers for programming languages like TypeScript and Haskell.

Speedy.js [12] is a cross-compiler that translates JavaScript/TypeScript to WebAssembly. Instead of translating whole programs like Emscripten, Speedy.js only translates performance-critical JavaScript/TypeScript code to WebAssembly and generates glue code to integrate the WebAssembly code and JavaScript code. Speedy.js uses the LLVM and Binaryen toolchain to translate performance-critical code. Speedy.js firstly uses the TypeScript compiler to generate TypeScript AST. Then it traverses the parsed TypeScript AST and translates it into LLVM IR. In the next step, LLVM WebAssembly backend translates LLVM IR into WebAssembly text format. Finally, Binaryen takes the generated WebAssembly text format, performs WebAssembly Specific optimizations and outputs WebAssembly binary code.

Slater [18] designed and implemented a Whiley-to-JavaScript translator. He used JavaScript Object to implement all Whiley types, and implemented a helper run-time library in JavaScript for compound data related operations, like equality test. Slater also successfully implemented unbound integers by using the JavaScript library `Big.js`.

Weng [13] built a compiler translating Whiley programs into C programs. The generated C programs were orders of magnitude more efficient than the same Whiley programs translated into naive Java code. Because arrays and records have value semantics, the translated C programs copy arrays and records when they are assigned. Weng used backward live variable analysis to find dead variables of arrays or records, and copies of these dead variables could be safely removed. Weng also did function read-write analysis to find read-only parameters. Copies of arrays/records which are passed to read-only function parameters can be eliminated as well. Since Whiley is designed for managing resource automatically (usually achieved by Garbage Collection) and there is no explicate statement to free memory, the translated C programs face the issue of memory leak. Weng used live variable analysis determine to which memory can be deallocated at a certain point and avoided most memory leaks. As copies of dead variables are eliminated by letting the variables pointing to the dead variables, sometimes more than one variables point to the same array/record. A boolean flag is assigned to each heap variable to denote which variable is responsible for deallocating the memory.

Similar to Whiley, arrays in MATLAB have value semantics as well. The current MATLAB implementation employs a reference-counting approach. It only increases the reference-count when an array is assigned, passed as a parameter or returned. When the array is up-

dated and the reference-count is larger than one, it makes a copy of the array. The reference-counting approach reduces unnecessary array copies but introduces extra work at run-time. Lameed [11] proposed a two-stage static analysis technique to eliminate unnecessary array copies in a MATLAB JIT compiler without reference-counting. The first stage is to detect read-only parameters and apply the *copy replacement* algorithm. The second stage consists of a forward analysis and a backward analysis. The forward phase detects which array update requires a copy, and the backward phase moves the copies to the best locations which may reduce necessary copies.

Herrera [8] examined the numerical program performance of JavaScript and WebAssembly on a wide range of devices including workstations, laptops, mobile devices and the Raspberry Pi. JavaScript's performance is 2X slower than native C. The authors not only tested JavaScript on the latest version of browsers but also tested on old versions from 2014. The numerical program performance of different versions of browsers is very close. The numerical program performance of WebAssembly are only 10% to 20% slower than native C. Firefox browser has the best WebAssembly performance of all browsers, which even beat the server-side Node.js. The reason might be Firefox browser firstly introduced ASM.js, and have the most experience on related optimizations.

Abhinav [10] built Browsix-Wasm, a Browsix [17] extension, which allows WebAssembly-compiled Unix applications to run in browsers. Abhinav used Browsix-Wasm and Browsix-SPEC to evaluate the performance of WebAssembly and native code by SPEC CPU 2006 and 2017 Benchmarks. Compared with native code, WebAssembly is 1.55 times slower in Chrome and 1.45 times slower in Firefox on average. Abhinav identified the causes of the performance gap between WebAssembly and native code. WebAssembly implementations use register allocators and code generators that perform worse than Clang. WebAssembly implementations must choose fast algorithms to allocate registers, while Clang can choose expensive algorithms for a better result. Since browsers reserves registers (e.g. Chrome reserves `r13` for Garbage Collection), it brings challenges to allocate registers. WebAssembly's safety design (stack overflow checks, indirect call checks, memory access checks) also slows it down.

Chapter 3

Design

This section discusses the design of a back-end plugin of Whiley Compiler, called `Whiley2Wasm`. It illustrates the basic pipeline of the Whiley Compiler and how `Whiley2Wasm` works together with other parts of the Whiley Compiler toolchain. It details each component of `Whiley2Wasm` and the assumptions made for it. This project aims to compile Whiley programs into WebAssembly so that they can be executed in browsers on different platforms with a good performance.

3.1 The Whiley compiler

Figure 3.1 illustrates the compilation pipeline of a Whiley source file to the different targets. The Whiley Compiler [7] is the front-end part of the compiler used to parse the source file, checking syntax, typing and verification. The output of the Whiley Compiler is the Whiley Intermediate Language (WyIL). Back ends or other tools can then parse WyIL files and generate target codes, or access the in-memory representation (a.k.a the Abstract Syntax Tree) of it for further processing. All the Whiley compiler backend plugins use the latter approach.

- `Whiley2Wasm`, the plugin developed during this project, converts the AST into WebAssembly text format (the .wat files). The final stage is to use the WebAssembly Binary Toolkit(WABT) [2] to translate the WebAssembly text format to the WebAssembly binary format(the .wasm files).

- `Whiley2JavaCompiler` [5] converts the AST into Java bytecode files. `Whiley2JavaCompiler` enables Whiley programs to run on Java Virtual Machine across operating systems.

- `Whiley2JavaScript` [6] converts the AST into JavaScript files. `Whiley2JavaScript` enables Whiley programs to run in browsers.

3.2 Whiley2Wasm

Figure 3.2 illustrates the pipeline of how `Whiley2Wasm` converts an instance of `WhileyFile` (the class name of Whiley Abstract Syntax Tree) to WebAssembly text format. `WasmFile` is a WebAssembly Abstract Syntax Tree defined in this project. The components of `WasmFile` are illustrated in the following sections. `WhileyFile2WasmFile` takes Whiley AST as input and output a `WasmFile` instance. `WasmFilePrinter` traverses the generated `WasmFile` instance and prints it to a file in WebAssembly `Text Format`. Another potential solution could be to

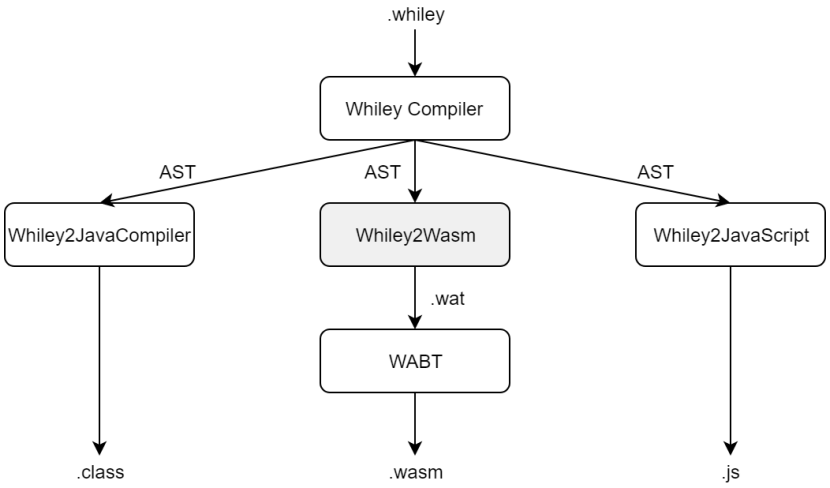

Figure 3.1: Whiley compilation pipeline

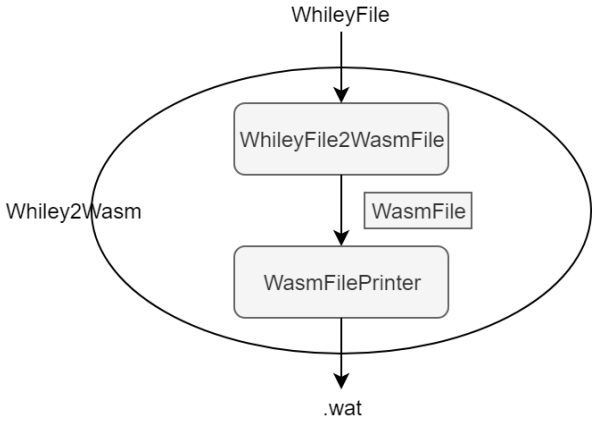

Figure 3.2: Whiley2Wasm

let `WhileyFile2WasmFile` directly output WebAssembly text format to a file without converting to an intermediate AST and the `WasmFilePrinter` would not be needed. We choose to use the intermediate representation (a.k.a `WhileyFile`) for several reasons:

- Having the WebAssembly AST is a good way to reuse code. The WebAssembly syntax and semantics are much simpler than Whiley's. Sometimes we cannot use a single WebAssembly Instruction to represent a Whiley statement/expression. For example, WebAssembly does not have Record, or logical negation. Thus, we have to reuse the existing WebAssembly instructions to implement these bits of Whiley syntax. For example, we use the WebAssembly `if` instruction to implement Whiley `if` statements, and we also use `if` instruction to implement Whiley `switch` statements.

- Since the WebAssembly AST is written in Java, we can employ the Java Compiler to ensure the generated WebAssembly AST is syntactically correct. For example, an `add` instruction takes two operands, and it would be a compile-time error if we try to create an `add` by providing only one operand.

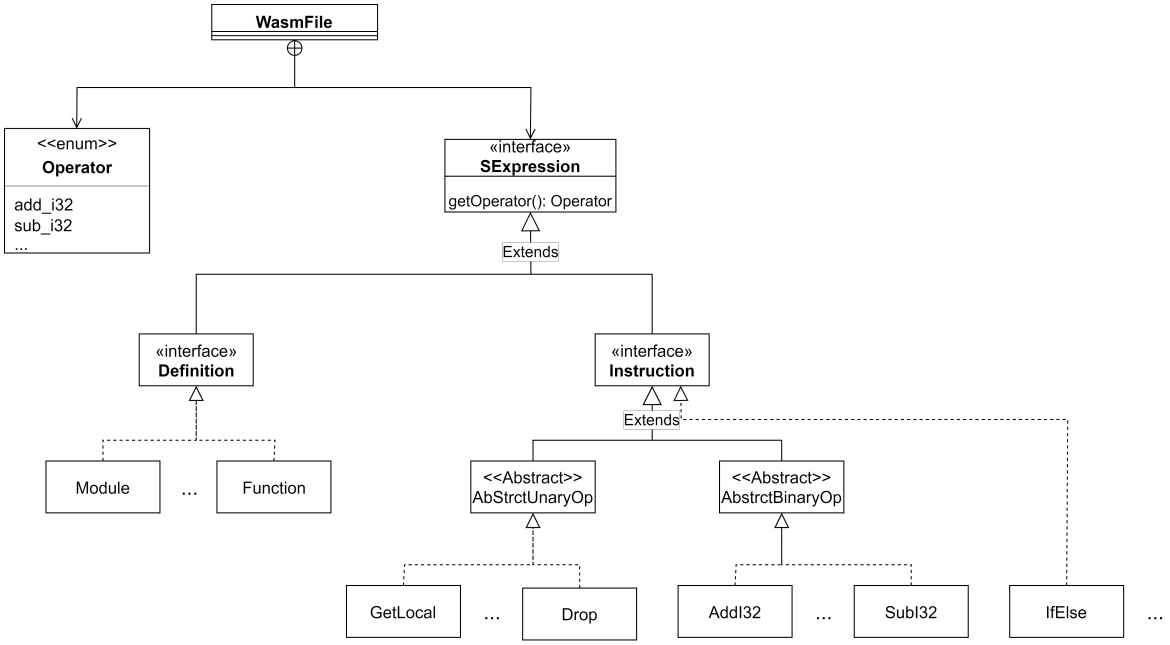

Figure 3.3: WasmFile Structure

3.3 WasmFile

`WasmFile` defines an AST mapping to WebAssembly, where almost every element has a corresponding WebAssembly instruction. Figure 3.3 illustrates the high-level structure.

- `Operator` is defined as an enum which is a collection of immutable constants used to identify the type of an AST element.

- Interface `SExpression` represents an arbitrary S-expression. It defines one method called "getOperator" which returns an enum value from `Operator` class.

- The `Definition` interface represents the WebAssembly definition instruction. For example, module declaration and function declaration belong to this category. `Instruction` interface represents WebAssembly operations. `Add` and `sub` belong to `Instruction`. `Definition` and `Instruction` both are an empty interface. The purpose of this is to help us to group classes.

- We defined two abstract class `AbstractBinaryOp` and `AbstractUnaryOp` to represent WebAssembly binary operators and WebAssembly unary operators correspondingly. Since there are so many similarities between binary operators, we abstract the common parts and put them into `AbstractBinaryOp` class. Similarly for `AbstractUnaryOp`.

- The operators which are neither binary operators nor unary operators implement Instruction directly. For example "drop" instruction and "unreachable" instruction.

- We categorise control flow statements as `Instruction` as well. This differs from Whiley where statement and expressions are defined separately. But in WebAssembly, there is no fundamental difference between expressions and statements. They both are executable instructions.

3.4 Translating Whiley to WebAssembly

This part discusses the design for translating Whiley AST to WebAssembly. We discuss the assumptions we made for this project. We also discuss our approach, particularly focusing on the Whiley syntax and semantics which cannot directly be presented by WebAssembly primitives, like union types, compound types and lambda expressions.

3.4.1 Assumptions

Given the limited scope of this project, we made the following assumptions for simplicity.

- **Assumption 1: all Whiley integers are i32**. Integers in Whiley are unbounded, which means any operations on integers won't cause integer overflow. With this assumption, we simply use `i32` to encode Whiley Integers which means they can overflow.
- **Assumption 2: memory leaks are allowed.** In Whiley, compound type data should be deallocated implicitly when they are not referenced by others. This can be achieved by reference counting or Garbage Collection (GC) provided by the run-time environment. Since WebAssembly doesn't support GC and implementing reference counting is beyond the scope of this project, we choose to allow memory leak and do not deallocate any data on the WebAssembly memory.
- **Assumption 3: open-records won't be supported.** To implement open-record, all fields of a record need to be stored in memory. This requirement will be quite different from the way we choose to implement records (which will be discussed very soon), so we decide to not implement open-records within this project.

3.4.2 Primitive types

Whiley has several primitive types like `int`, `bool`, `bytes`, `null`, etc., while WebAssembly only has four value types, i32, i64, f32 and f64. Therefore, we use the following rules to represent Whiley primary types:

- i32 represents `int`. The i32 value is the integer value in Whiley.
- i32 represents `bool`. 0 is `false` value and any non-zero value are `true` value.
- i32 represents `null`. We always use 0 to represent `null`.

Though we use i32 to represent many different types, the actual meaning of the i32 values are determined by their types and all types are checked by the Whiley Compiler.

3.4.3 Arrays

WebAssembly does not have arrays as a primitive type, so we need to implement array by ourselves. We use pointers to implement arrays which is similar to the C programming language. For each array, we allocate contiguous bytes in memory and store the address of the first array element to the pointer. Thus, array indices are zero-based offsets from the box pointer and headers are negative offset. We choose this design mainly because JavaScript uses zero-based offsets for arrays and the generated WebAssembly programs may be invoked by JavaScript programs. The layout of arrays of non-union type elements in memory is illustrated in Figure 3.4. The first eight bytes are the header of an array. The first four bytes store a pointer of array elements' tag values. In this example, it is always

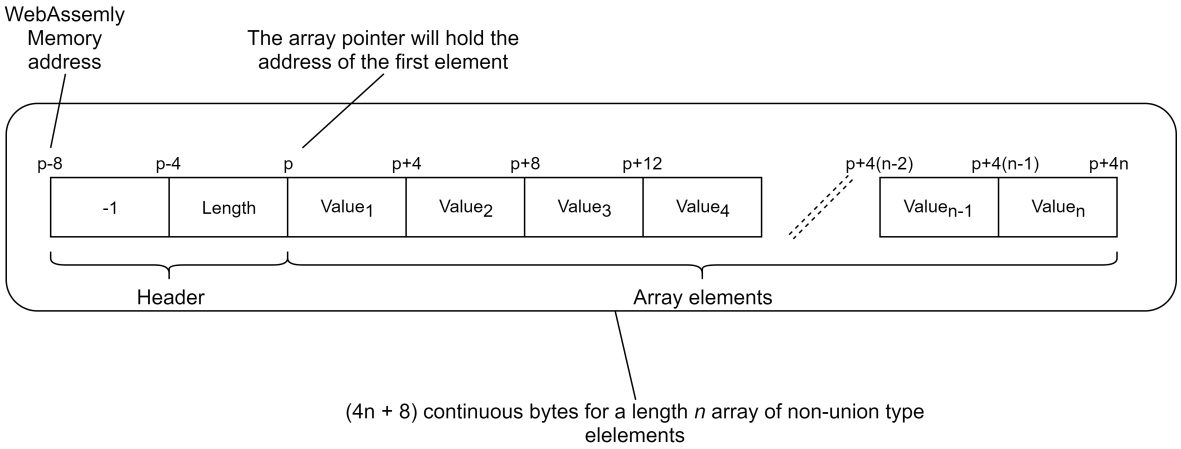

Figure 3.4: Layout of arrays

the integer -1 as we only store tag values in memory for union type elements. The next following 4 bytes in the header are for the length of the array. Starting from the 9th byte, every 4 bytes represents a value stored in the array. The size of the array is calculated with this formula:

$$bytes\_needed = n * 4 + 8$$

where n is the size of the array. Since we use i32 to represent all primitive types and pointers, for a size n array, we need 4 * n bytes to stored all array elements. We need extra 4 bytes for storing the length of the array and another 4 bytes for the pointer of the tag array. For example, an integer array with length 10 needs 48 bytes.

Multi-dimensional arrays under this design are arrays of pointers to lower-dimensional arrays. Figure 3.5 illustrates the memory layout of a two-dimensional array with values `[[10, 20], [30, 40]]`. The first and second element in the array are both pointers, which point to first elements from two one dimensional arrays.

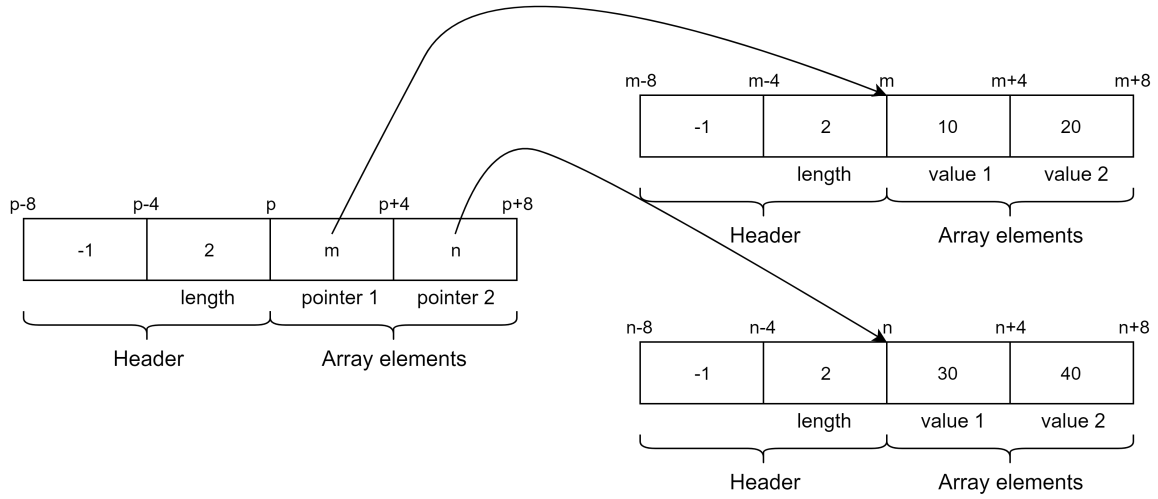

Figure 3.5: Illustration of a two dimensional array

3.4.4 Records

In Whiley, records' fields can be any type, but in this section, we only discuss records whose fields are non-union types. We also use pointers to implement records. This is similar to ar-

rays. We allocate continuous space in memory and convert record field accesses to memory access. Figure 3.6 shows the memory layout of records. We still have the -1 at the beginning (to reserve spaces for the pointer of tag values), then the total number of fields, followed by each field's value.

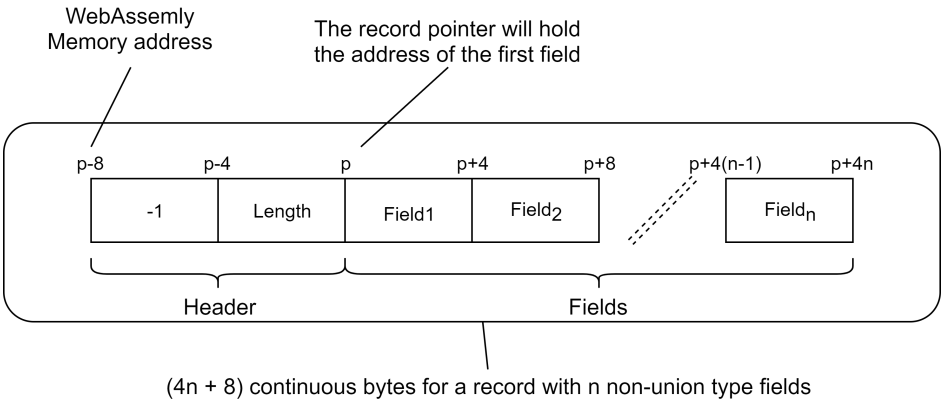

Figure 3.6: Memory layout of records

3.4.5 Union types

In Whiley, a union type is a combination of two or more component types. A union type can be defined in the following way: **type** nint **is null** | **int**, where *nint* is a union type, and *null* and *int* are the component types. Variables with union types can hold values of any component type. In many cases, we want to know the exact type of the value of a union type, but we cannot derive this information from the encoded byte values. For example, based on our encoding, a WebAssembly i32 type variable with 0 value can be interpreted as *null*, integer *0* or the boolean value *false* in Whiley. To resolve this issue, we introduce a technique called **tagged unions** to capture run-time type information.

A tag union is a combination of a tag and a union. The tag is an integer which denotes the exact component type a union type holds. For a union type with n component types, $UnionType ::= T_0|T_1..|T_{n-1}$, the tag value for T_i is i. Figure 3.7 illustrates how to tag the type *nint* defined by: **type** nint **is null** | **int**. The tag value of the variable x is 0 which denotes x is *null*. The tag values of the variable y and z are 1 which denotes they are integers.

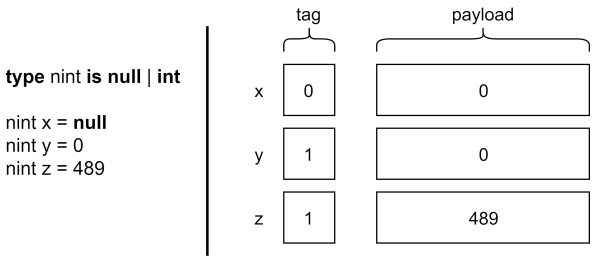

Figure 3.7: Tagged unions example

Union types can be nested, which means a component type of a union can be another union type. In the code section of Figure 3.8, the *nboolint* type is a nested union type. Based on the previous tagging approach, the variable x and the variable y may both have tag value 1, which is obviously not enough to tell us the exact variable type of the value at run-time. To address this, we extend the tagging approach to tag each level of the union types and use

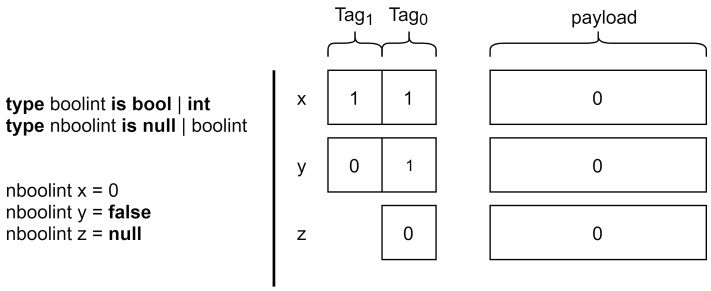

Figure 3.8: Tagged nested unions example

the combination of tags at each level as the final tag. The variable y's tag has two parts: the right part denotes y is *boolint* type and the left part further denotes y is *int* type.

To implement the above design, we need to encode each part of tags into single number. Our approach is to recursively calculate the tag value of the nested union types based on the formula:

$$tag = i + n * c \tag{3.1}$$

Here i is the index of the component type, n is the number of components in this union, c is the tag value from the sub-union.

If we use codes in Figure 3.8 as am example, the tag of the variable x would be 3, and the tag of the variable y would be 1. For the variable x, since int is second component of *boolint*, so $i = 1, n = 2, c = 1$ and $tag = 1 + 1 * 2 = 3$. For the variable y, the type *bool* is the first component of the type *boolint*, so $i = 1, n = 2, c = 0$ and $tag = 1 + 2 * 0 = 1$.

3.4.6 Arrays with union type elements

Since we need extra spaces to store the tag values, we allocate spaces in memory to store tag values for each union type array element. Figure 3.9 illustrates the layout of arrays of union type elements. It can be viewed as two arrays, one array for array elements values and the other for array tag values. Compared with the layout of the arrays of non-union elements, the first 4 bytes are not -1 any more. Instead, it stores the address the tag value of the first array element(a.k.a the base address of the array of tags).

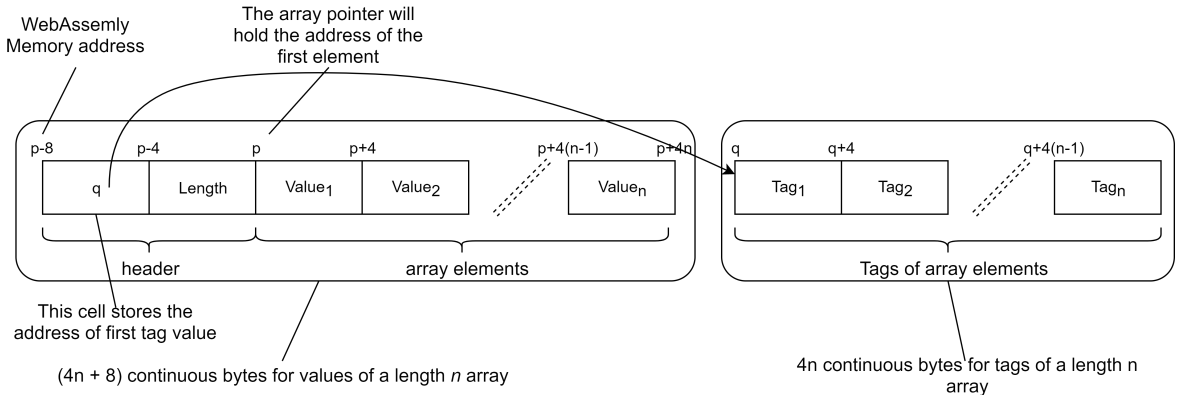

Figure 3.9: Layout of arrays of union type elements

3.4.7 Records have union type fields

Similarly, for records which have union type fields, we also allocate continuous spaces in memory for tag values. For simplicity, we allocate space for the tags of all fields, no matter how many fields of the record are union types. By this design, we can reuse the code built for implementing arrays but it is inefficient and wastes some bytes. This is because the types of a record's fields may be a mixture of union types and non-union types (e.g. **type** t **is** { **int** x, **int**|**bool** y}), and it is unnecessary to allocate space for the tag of a field with non-union types.

3.5 Function Type Variables, Function References and Lambda Expressions

In Whiley, function type variables are variables which hold function references or lambda function references. We can make indirect invocation through function type variables. To implement indirect invocation in WebAssembly, we can employ WebAssembly tables. We first put the referred function into the WebAssembly table, and let the function type variable hold or indirectly hold the table index of the referred function. Then each Whiley indirect invocation can be translated into a WebAssembly indirect invocation.

WebAssembly does not allow nested functions. Thus, all Whiley lambda functions must be translated to regular WebAssembly functions in a module. For lambda functions without captured variables, that is easy. We simply translate lambda functions as regular functions but with random names. Then assigning a lambda function to a variable becomes function reference assignment, and invoking lambda function becomes indirect invocation.

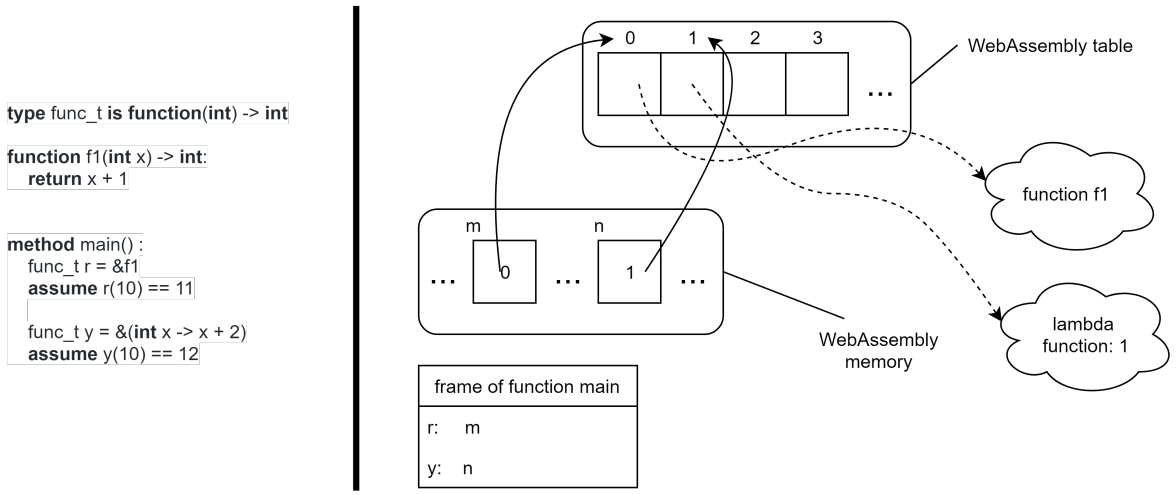

Figure 3.10: An example of function references and lambda functions without captured variables.

Figure 3.10 visualises the layout of a program at run-time. The left-hand-side is a program in Whiley. In the *main* method, the variable *r* is a function reference and the variable *y* is a lambda function reference. The right-hand-side is the WebAssembly layout when running this program. In the WebAssembly table, the first element points to the function *f1*, and the second element points to the lambda function created in the *main* method. In memory, at location m there is a length 1 array whose first element is the index of the function *f1*; at location n there is also a length 1 array whose first element is the index of the lambda function. In the frame table, the variable *r* and the variable *y* holds the base addresses of the two

length 1 arrays, m and n, correspondingly. The reader might wonder, in this design, whether the length one array is redundant, as the function type variable can hold WebAssembly table index directly. We choose to use arrays mainly for implementing lambda functions with captured variables.

Implementing lambda expressions with captured variables is harder because the values of the captured variables are determined at the run time and we have to create functions at compile-time in WebAssembly. Our solution to achieve this is passing an array as extra parameter. The array represents all the captured variables. That is, every operation related to captured variables will be converted to operations on the array's elements. At run-time, when the lambda expression should be executed, we make a copy of all the captured variables and store the copy in the array. Thus, a function type variable need to hold two things, one is the index of the function on the WebAssembly table, and the other one is an array for the captured variables. Since a variable can only hold one value, we then use an array to hold both of them. Thus, the first element in the array is the index of the function, and the rest would be the copy of captured variables.

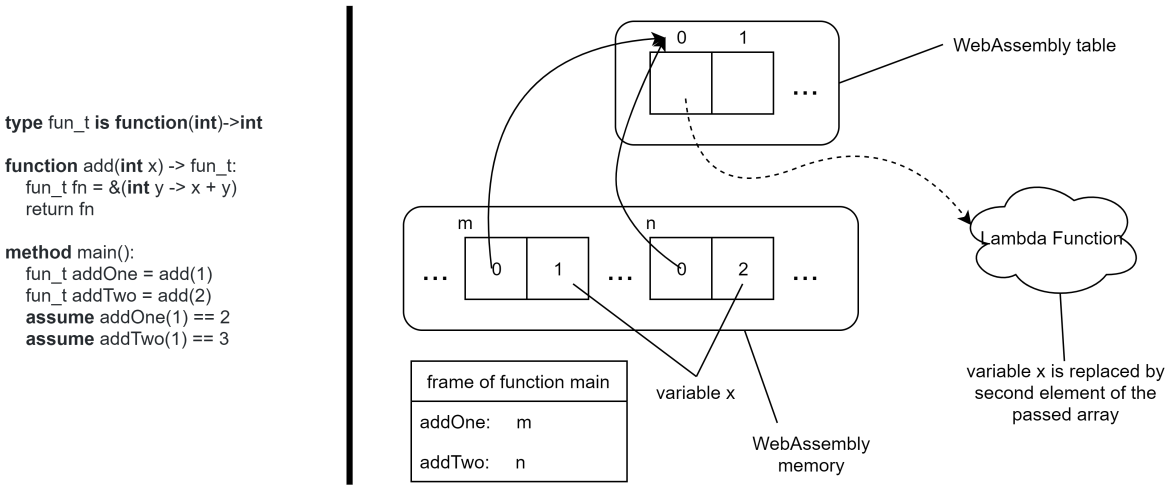

Figure 3.11: An example lambda functions with captured variables.

Figure 3.11 illustrates how the captured variables are implemented by an example. In the code section, the function *add* returns a lambda function and the variable x is a captured variable of the lambda function. In the *main* method, the function *add* is invoked twice, and results are stored in two function type variables. However, as the WebAssembly layout shows, on the right-hand side of Figure 3.11, we only generate one lambda function. When the function *add* is invoked, in WebAssembly, it only generates the arrays which contains the lambda function index (0 in this case)and the a copy of the scope variable x.

Chapter 4

Implementation

The implementation of `whiley2wasm` involves translating Whiley expressions, statements and implementing Whiley type systems in WebAssembly. This section provides an overview of these parts. We begin by examining some of the more straightforward cases and then consider more complex cases.

4.1 Expressions and Control Flow Statements

We translate the Whiley AST using bottom-up style. We will use a concrete example to illustrate how we translate expressions, control flow statement and functions from Whiley to WebAssembly. Listing 4.1 shows a Whiley function called *getFactor*. If the input of the function is a composite number, *getFactor* returns the input's largest factor. If the input is a prime number, *getFactor* returns `-1`. We used the `Whiley2Wasm` compile the program in Listing 4.1, and the generated WebAssembly program is shown in Listing 4.2.

Most Whiley operators can be converted into WebAssembly instructions directly. For example, the subtract operator in Whiley can be translated into **i32.sub**. However, some Whiley operators have no directly mapping into WebAssembly operators. For example, WebAssembly has no negation operator. To address this, we usually use a combination of several WebAssembly operators instead. In our code example, the negation of variable *i* (Listing 4.1, line 13) is converted to zero minus *i* (Listing 4.2, line 32).

Listing 4.1 shows two `if` instruction examples. You may notice that the second instruction (line 27) has a flag (**result i32**) because its true branch contains a `return` statement. This indicates that after executing this `if` instruction, a value will be left on the stack. To guarantee the stack height is always deterministic, WebAssembly requires the stack height of both branches from a `if` instruction must be the same. This brings some challenge because it is very common that one branch contains `return` statement and the other branch does not (e.g. Listing 4.1, line 10-13). To address such unbalanced branches, we first insert a dummy value to the branch which contains no `return` statement. Then, we add a `drop` instruction immediately after the `if` instruction to discard the dummy value (e.g. Listing 4.2, line 26-37).

We always use a `block` instruction and a `loop` instruction to translate `while` statements. With this setup, `break` statements and `continue` statements will be compiled as `br` instructions targeting the `block` and the `loop` correspondingly. Though it seems we've resolved translating control flow statements, the real challenge is to determine indices of the targets. As we discussed in Section 2.1.4, the outer control structures (e.g. loop or block) of `br/br_if` are referred to indices from 0 to n - 1 where the innermost control structure is 0. In Listing 4.2, (**br_if** 1) at line 8 and (**br** 2) at line 18 have the same jump target but different indices!

Listing 4.1: An Whiley Function of calculating largest factor

```
function getFactor(int item) -> (int r):
    int i = item - 1

    while i > 1:
        if item % i == 0:
            break
        else:
            i = i - 1

    if i > 1:
        return i
    else:
        i = -i

    return i
```

This is because `if` instruction in WebAssembly is actually a special form of `block`, and we have to include `if` instruction when we calculate indices. By this reason, we have to traverse the AST to to calculate indices of the jump targets for every `br` instruction at compile-time.

4.2 Switch Statement

In many programmings, like Java or C , `switch` statements allows execution to fall through from one case to the following cases util a `break` statement is reached. However, in Whiley, `switch` statements do not fall through. That is, after executing the last statement in a `switch case`, the flow of control jumps to the next statement following the `switch` statement. Listing 4.3 gives an example of Whiley `switch` statements. Because `swtich` statements do not fall through, in Listing 4.3, the function *f* and *g* are equivalent.

In this project, we use nested `if` instructions to implement Whiley `switch` statements. Listing 4.4 illustrates how the function *f* in Listing 4.3 is translated into WebAssembly with this approach. WebAssembly provides an instruction called `br_table`, which might be used to translate Whiley `switch` statements. `br_table` uses the value on top the stack as an index to jump to certain target, which is similar to `TableSwitch` in Java bytecode. Listing 4.6 illustrates how the function *s* in Listing 4.5 is translated into WebAssembly using `br_table`. The reason we choose `if` instruction over `br_table` is that, `br_table` can only use an integer as a jump index and we cannot use `br_table` to translate the `switch` statements with arrays or records as switch cases (e.g. the function *g* in Listing 4.6).

Listing 4.2: Generated getFactor in WebAssembly

```
(func $getFactor (param $item i32)(result i32)(local $r i32)(local $i i32)

  (i32.sub (get_local $item) (i32.const 1))
  (set_local $i)

  (block
    (loop
      (br_if 1                          (* conditionally jump to block *)
        (i32.le_s (get_local $i) (i32.const 1)))
      (i32.eq                             (* condition of if statement *)
        (i32.rem_s
          (get_local $item)
          (get_local $i)
        )
        (i32.const 0)
      )
      (if
        (then (br 2))                                 (* jump to block *)
        (else
          (i32.sub (get_local $i) (i32.const 1))
          (set_local $i))
      )
      (br 0)                                          (* jump to loop  *)
    )
  )
  (i32.gt_s (get_local $i) (i32.const 1))
  (if (result i32)
  (then
    (return (get_local $i))
  )
  (else
    (i32.sub (i32.const 0) (get_local $i))
    (set_local $i)
    (i32.const 0)                 (* put a dummy value to the stack *)
  )
  )
  (drop)                                      (* drop the dummy value *)
  (return (get_local $i))
)
```

Listing 4.3: : Whiley switch statements example

```
function f(int x) -> int:
  switch x:
    case 1:
      x = x + 1
    case 100:
      x = x + 2
    default:
      x = -1
  return x

function g(int x) -> int:
  if x == 1:
    x = x + 1
  else if x == 100:
    x = x + 2
  else:
    x = -1
  return x

method main():
  assume f(1) == 2
  assume f(100) == 102
  assume f(3) == -1
```

Listing 4.4: : An example of implementing Whiley statements in WebAssembly

```
func $f (param $x i32)(result i32)
  (i32.eq (get_local $x) (i32.const 1))
  (if
  (then
    (*first swith case*)
    (i32.add (get_local $x)(i32.const 1))
    (set_local $x)
  )
  (else
    (i32.eq (get_local $x)(i32.const 100))
    (if
    (then
      (*second swtich case*)
      (i32.add (get_local $x) (i32.const 2))
      (set_local $x)
    )
    (else
      (*default case*)
      (i32.sub (i32.const 0) (i32.const 1))
      (set_local $x)
    )))
  )
  (return (get_local $x)))
```

Listing 4.5: : Whiley switch statements example

```
function s(int x) -> int:
  switch x:
    case 0:
      return 99
    case 1:
      return 100
    case 2:
      return 101
    default:
      return 102

function g(int[] x) -> int:
  switch x:
    case [0;0]:
      return 0
    case [1]:
      return -1
  return 10
```

Listing 4.6: : An example of translating Whiley switch statements with br_table

```
(func $s (param i32) (result i32)
  (block
    (block
      (block
        (block
          (br_table 0 1 2 3 (local.get 0))
        )
        (return (i32.const 100))
      )
      (return (i32.const 101))
    )
    (return (i32.const 102))
  )
  (return (i32.const 103))
)
```

4.3 Function Overloading

Function overloading refers to the ability to have multiple functions have the same function name but different parameters. Whiley supports function overloading but WebAssembly

Listing 4.7: Whiley function overloading example

```
function add(int x, int y) -> int:
  return x + y

function add(int x, int y, int z) -> int:
  return x + y + z
```

does not. Listing 4.7 illustrates two overloaded functions in Whiley. The two functions has the same function name but different numbers of parameters. To support Whiley function overloading we use the approach of *name mangling*. Here we encode each function names with the original function name plus a string representing function parameters (called the mangle). For example:

- the encoded name of function `add(int, int)` will be "add_II"
- the encoded name of function `add(int, int, int)` will be "add_III"
- the encoded name of function `g(int[])` will be "g_aI"
- the encoded name of function `g(bool[])` will be "g_aB"

Sometimes name mangling brings some challenges to link modules. A typical use case for WebAssembly is to write a core part of a program in one programming language, compile them into WebAssembly and write some JavaScript codes to invoke the compiled code. For a programmer, it is natural to invoke a function by the declared function name rather than a mangled name. WebAssembly has a nice feature to solve this: each exported function has a exported function name which can be different from the function name. So in our implementation, we use the function's name as the exported function name and use the mangled function names within the module. Because we only export Whiley functions with the `export` modifier, and because Whiley force the exported function names to be unique, we needn't worry about whether exported functions are overloaded.

4.4 Types

Whiley has a powerful type system. It not only has common types like primitive types, arrays and records, but also `union types`. Whiley employs structure typing to bring flexibility to the types system. In this section, we will discuss the implementation of Whiley type system in WebAssembly, primarily focusing on union types and compound types' value semantics.

4.4.1 Value Semantics

Since arrays and records in Whiley have value semantics, we usually should make copy of arrays/records when assigning them to variables or passing them as parameters. This differs from Java in that, Java only assigns copy references to variables or passes references as parameters. We use a pointer approach to implement array, and a high dimensional array is an array of pointers pointing to lower dimensional arrays. It is impossible to implement a universal array copy function, because we cannot tell an array element is a pointer or a primitive value based on its byte values. The approach we used is to generate `copy` functions per array/record type. Figure 4.1 illustrates how a two dimensional array copy function

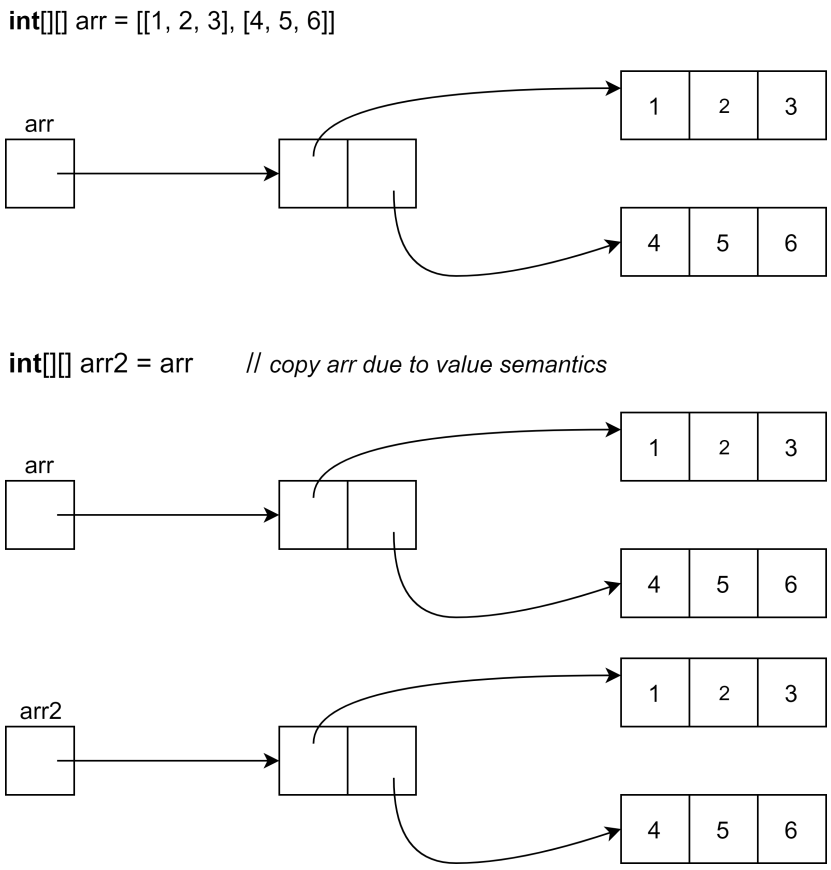

Figure 4.1: Deep copy a two dimensional array

works. Since each two dimensional array in our implementation is an array of pointers to one dimensional arrays, the `copy` function need to copy the original array deeply. That is, it copies the content of each one-dimensional array rather than the pointers of them. Listing 4.8 pseodocode of the `copy` functions of copying two-dimensional int arrays.

Listing 4.8: Pseudocode of two-dimensional int array copy function

```
Function ArrayCopy1D(int[]a) -> int[]:
  Create an empty array b with the same length of a
  for i in 0..|a|:
    b[i] = a[i]
  return b

Function ArrayCopy2D(int[][] a) -> int[][]:
  Create an empty array b with the same length of a
  for i in 0..|a|:
    b[i] = ArrayCopy1D(a[i])
  return b
```

4.4.2 Union Types

Ghost variables

Recall discussion from Section 3.4.5, we use tagged unions to implement union types and we need extra spaces to store tag values. For records and arrays, we allocate extra spaces

in memory for tags. For variables, we choose to add a ghost variable to each union type variable to store tags values, because the byte sizes of variables are fixed in WebAssembly. For example, if we have a union type variable `x`, we will create a ghost variable named "__x__T" for `x`'s tag.

Updating tags

Because we use tags to determine the exact value types of unions at run-time, we have to guarantee tags are up-to-date after unions change their values. For example, we need to update the ghost variable when we assign a new value to a union type variable. Listing 4.9 gives some examples of assignments involve updating tags. Updating a tag has four scenarios: `copy`, `enter`, `retag` and `leave`.

- `Copy` refers to assignment between same the union type. In Listing 4.9, `v = x` is an assignment between the same type. In this case, we can assign `x`'s tag value to `v`'s tag directly.
- `Enter` refers to assign a non-union type value to a union. In Listing 4.9, `v = y` is an assignment between a primitive type and a union type. Since we know `v`'s tag value should be 1 after this assignment at compile-time, we can generate an instruction to update `v`'s tag to 1.
- `Retag` refers to assignment between different the union type. In Listing 4.9, `v = z` is an assignment between different union types. In this case, we need to convert `z`'s tag value from type `nintbool` to type `intbool`. Specifically, if `z`'s tag value is 0 or 1, we should assign 1 or 2 to `v`'s tag correspondingly. Such tag value conversion is done by invoking a `convert` function which is generated at compile-time. Listing 4.10 shows the pseudocode of this function.
- `Leave` refers to assign a union to a non-union. In Listing 4.9, `y = x` assigns a union to a primitive variable. In this case, we simply discard the tag of `x`, and only assign the value of `x` to `y`.

Listing 4.9: An example of updating tags

```
type intbool is int|bool
type nintbool is null|int|bool


method main():
   intbool  v = false
   intbool  x = −1
   int      y = 2
   nintbool z = true

   v = x   // copy
   v = y   // enter
   if z is intbool:
      v = z   // retag
   if x is int:
      y = x   // leave
```

Listing 4.10: Pseudocode of generated tag value converter

```
//from type intbool to type nintbool
Function convert(int tag) −> int:
   If tag == 0
      Return 1

   If tag == 1
      Return 2

   Return −1
```

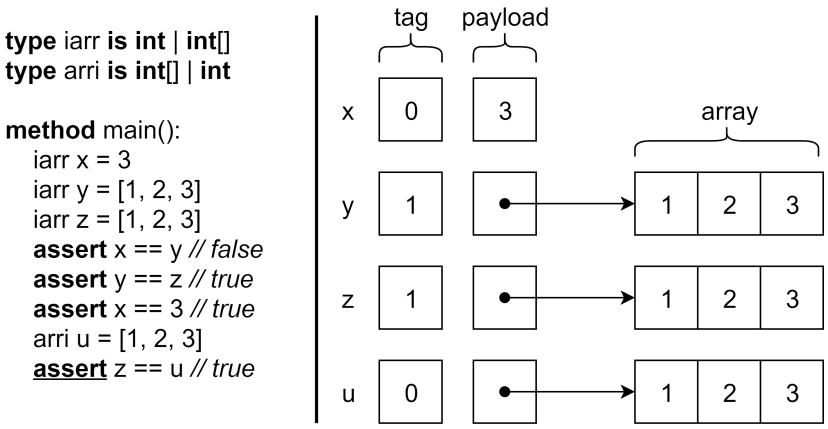

Figure 4.2: An example of testing quality between unions

Equality test

Testing equality between two unions involving comparing actual values and tag values. Similar to assignment statements, to do the proper equality test also needs to consider testing between two different union types, and testing between a union type and a non-union type. Figure 4.2 gives examples of these scenarios. `x` isn't equal to `y` because `x` and `y` are the same union type but have different tag values, which indicates the types of their payload are different. `y` is equal to `z`, because their tags indicate their payloads are pointers of int arrays and the pointed arrays are equal. `x` is equal to 3, because its tag indicates its payload is integer and the integer equals 3. Testing equality between `u` and `z` is more complicated than others, because the `u` and `z` are different union types. Though the tag values of `z` and `u` are different, they both indicate the payload of `z` and `u` are pointers of int arrays. Since the pointed arrays are equal, we can finally claim `u` and `z` are equal.

As for cloning, we generate `equals` functions for each union type at compile-time. Listing 4.11 gives pseudocode of the `equals` function of type int | int []. It first check whether the tag values are equal. If tags are not equal, it returns `false` directly. After that, based on the tag value, we use different strategies to compare values. When the tag is 0, we compare two values directly because 0 indicates they are both integer. When the tag is 1, we will invoke `ArrayEquals` function to compare two arrays.

Listing 4.11: Pseudocode of an `equals` function

```
type intarr is int|int[]
Function Equals(intarr one, int oneTag, intarr other, int otherTag) -> boolean:
  If oneTag != otherTag
  Then
    Return False

  Switch oneTag:
  Case 0:
    Return one == other
  Case 1:
    Return ArrayEquals(one, other)
  Default:
    Return false
```

To handle equality test between two different union types, we first cast the right-hand

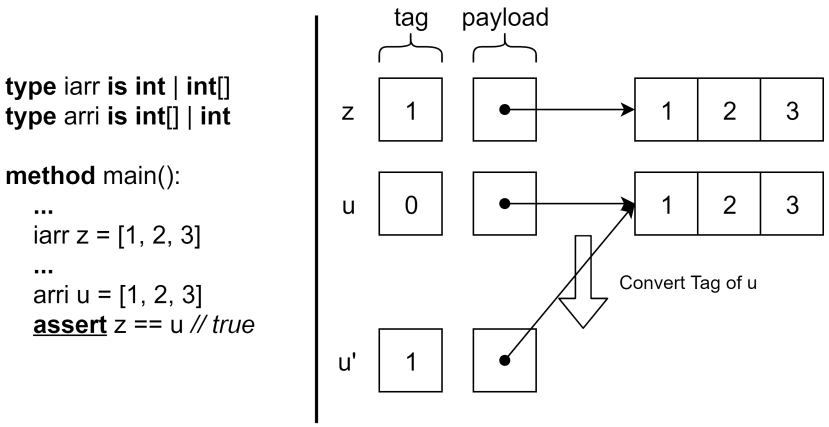

Figure 4.3: An example of using convert function for testing quality between unions

side to the type of the left-hand side union. Then, we can reuse the `equals` functions for the same union types we discussed before. Do the casting is straightforward, because we just need to invoke the tag `convert` function. Figure 4.3 illustrates how to use this approach in an example. We first invoke the `convert` function (shown in Listing 4.12) to cast `u` from type *arri* to *iarr*. The casted `u` are denoted as `u'` in Figure 4.3. As we see, only the tag value of `u` changed and the payload is the same. After converting the tag, we can use the `equals` function shown in 4.11 to test equality between `u'` and `z`.

Listing 4.12: Pseudocode of the convert function used in Figure 4.3

```
// convert from arri to iarr
Function convert(int tag) -> int:
  Switch tag:
  Case 0:
    Return 1
  Case 1:
    Return 0
  Default:
    Return -1
```

Chapter 5

Evaluation

We evaluated both the correctness and performance of our implementation. We use the existing unit testing suite that comes with the Whiley Compiler to evaluate the implementation's correctness. To evaluate the performance, we used a benchmark suite called WyBench [16]. The test cases and the WyBench are both provided by Dr David Pearce.

5.1 Unit testing

The unit testing suite consists of 652 test cases which are all written in Whiley. Each test case focus on one particular syntax. Listing 5.1 is a test case of multiplication. The `assert` statement is evaluated at run-time. If x is not 7, it will throw a run-time exception indicating the test case is failed.

Listing 5.1: example test case of multiplication

```
public export method test() :
   int x = (2 * 3) + 1
   assert x == 7
```

Testing mainly has two steps. Run the compiler to compile the test cases into WebAssembly and save the result on disk. Then run the compiled WebAssembly program on `Node.js` and track whether the program throws run-time exceptions. If it throws any exception, then we will mark the test case failed. Otherwise, we will mark the test case passed. We use JUnit, a Java unit testing framework, to perform these two steps together. For the second step, we use the Java `Runtime` class to invoke Node.js and to extract run-time information.

Table 5.1 shows the test statistics. We passed 614/652 test cases while only had 1 test case failed. The remaining 37 test cases were skipped. Table 5.1 also shows the categories of the 37 skipped test cases belong to. Around two thirds of them (24/37) are skipped due to some existing WyC issues. WyC is the compiler front-end we used to generate Whiley AST. For various reasons, the WyC cannot parse the 24 cases correctly. We skipped the remainning 13 test cases based on the assumptions we made before, that we assume all integers are `int32`, and we won't support open records. This indicates we successfully implement almost all Whiley syntax and semantics.

5.2 Performance evaluation

WyBench is a benchmark programs suite for testing Whiley. We selected 10 out of 33 tests from WyBench to test the speed and run-time memory usage of WebAssembly. Since WebAssembly does not support disk I/O so far, we inline the test data into the source codes.

Total	**652(100%)**
Passed	614(94.1%)
Failed	1(0.2%)
Skipped: WyC issues	24(3.7%)
Skipped: Unbound Integers	4(0.6%)
Skipped: Open Records	9(1.4%)

Table 5.1: Unit test statistics

Listing 5.3 and Listing 5.2 illustrate how we inline the data to the `003_gcd` program from the WyBench. We also use the existing Whiley Compiler JavaScript back-end to compile the same tests into JavaScript, and use JavaScript's performance on Node.js as the baseline. We choose JavaScript rather than the other languages, like Java, for several reasons. Firstly, JavaScript and WebAssembly both are mainly used for writing Web applications, and WebAssembly is designed to be a complement to JavaScript. Secondly, the JavaScript back-end uses the same assumption that integers in Whiley are i32. Thirdly, in the experiment, JavaScript programs and WebAssembly programs can run on the same platform, Node.js.

Listing 5.2: : The original 003_gcd program from Wybench

```
...
method main(ascii::string[] args):
   if |args| == 0:
      io::println("usage: gcd <input−file>")
   else:
      // First, parse input
      filesystem::File file =
        filesystem::open(args[0],filesystem::READONLY)
      ascii::string input = ascii::from_bytes(file.read_all())
      int[]|null data = parser::parseInts(input)
      // Second, compute gcds
      if data is null:
         io::println("error parsing input")
      else:
         nat i = 0
         while i < |data|:
            nat j = i+1
            while j < |data|:
               if(data[i] is nat && data[j] is nat):
                  io::println(gcd(data[i],data[j]))
               j = j + 1
            i = i + 1
```

Listing 5.3: : the 003_gcd program with inline data

```
...
method main():


   // First, parse input
   int[] data = [6779,9795,3006,....,]


   // Second, compute gcds
   nat i = 0
   nat result = 0
   while i < |data|:
      nat j = i+1
      while j < |data|:
         if(data[i] is nat
                  && data[j] is nat):
            result = gcd(data[i],data[j])
         j = j + 1
      i = i + 1
```

In addition to run the compiled JavaScript Benchmark programs on Node.js, we also run them on the Oracle Nashorn which is a JavaScript Engine for the JVM shipped with Java SE8. Oracle Nashorn dose not aggressively optimise JavaScript, and, for example, no support for Just-in-time compilation (JIT). In contrast, V8, the JavaScript engine inside of Node.js, employs many optimization techniques, like JIT and inlining, to improve its performance. Including Oracle Nashorn as a running environment could give us some sense of how optimization would impact the performance.

5.2.1 Methodology

All the tests were run an Ubuntu based desktop. The execution engine is Node.js version v10.15.3 and Oracle Nashorn from Java SE 1.8.0_201.

To get accurate execution times for each benchmark, we write a simple framework which imports the Benchmark programs and invokes the main function of the programs ten times in a loop. We start and end the timer right before and after the invocation statement. Finally, we drop the first five results and report the average execution time of the last five runs. The first five runs are treated as warm-up run will give VM time to do optimisation.

We use the Resident Set Size(RSS) of Node.js as the measurement of memory usage for both JavaScript and WebAssembly benchmark programs. Resident Set Size is the amount of space occupied in the main memory device for the process, which includes the heap, code segment and stack. We record RSS for each run and report the maximum RSS. However, we did not measure benchmark programs' memory usage on Oracle Nashorn, because we can only get the JVM memory usage but this number includes the other process's memory usage like JUnit.

Table 5.2 gives details of the ten benchmark programs. For the last benchmark program, `Conway`, we give two equivalent implementations. The first one use arrays to represent the board. Since arrays have value semantics, every time we pass an array to the other method, it will create a deep copy of the array. Because we did not deallocate memory, this WebAssembly version reaches the 1GB memory limit after five iterations in the game. To avoid unnecessary copying arrays, we convert this program into a version which use Whiley references to represent the board. In this version, we can run the game for 10000 iterations safely.

Benchmark program name	**Lines of code**	**Details**
002_fib	14	Computes Finonacci numbers
003_gcd	25	Computes the Greastest Comman Divisor of two numbers
004_matrix	313	Matrix multiplication
006_queens	124	Finds solutions for the N-queens problem
007_regex	70	Implements a simple regular expression matcher
009_lz77	236	Implements LZ77 compression and decompression
010_sort	73	Applies merge sort to an integer array
014_lights	39	Traffic lights simulation
017_math	192	Simple math functions: abs, max, min, etc.
102_conway	123	Implements Conway's Game of Life. In the implementation, the board is represented by a two-dimensional array. Run the game for *4* iterations.
102_conwayptr	122	Implements Conway's Game of Life. In the implementation, the board is represented by a Whiley reference to two-dimensional array. Run the game for *10000* iterations.

Table 5.2: Details of the benchmark programs

5.2.2 Results

Figure 5.1 shows the normalized execution time. (The raw results can be found at appendix A.) In most cases, WebAssembly benchmark programs run faster than JavaScript benchmark programs on Node.js. On average, WebAssembly is 20% to 30% faster. In two programs (`017_math` and `102_conway`), WebAssembly is slower than JavaScript on Node.js. In the Conway benchmark, WebAssembly is 150 times slower which is mainly caused by the program having many array copy operations. The pointer version Conway proves this explanation, WebAssembly is 25% faster. Surprisingly, WebAssembly is 2 times slower in the *math* program. This is because Node.js did optimization for JavaScript, for example, inlining.

Running the same JavaScript programs on Node.js and Oracle Nashorn have significant differences. Node.js on average are 4 5 times faster than Oracle Nashorn. Clearly, V8 engine did a lot of optimization on JavaScript. In spite of this, WebAssembly is still faster than the optimized JavaScript.

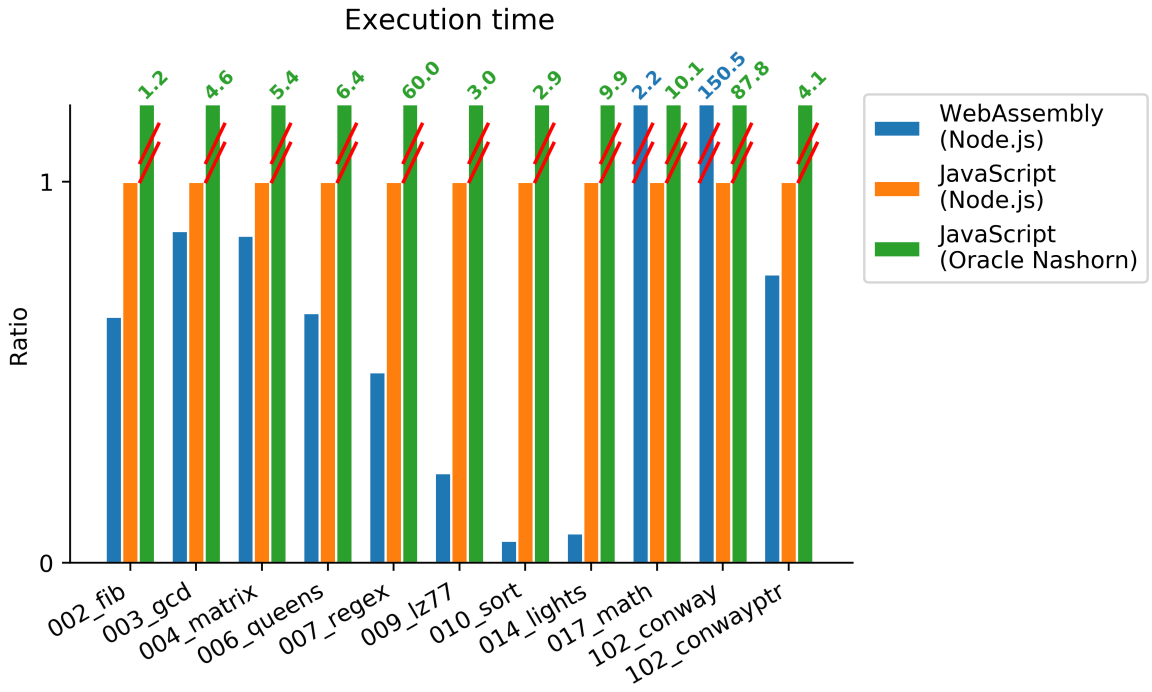

Figure 5.1: Execution time comparison

Figure 5.2 compares the execution memory usage between WebAssembly and JavaScript. (The raw results can be found at appendix A.) Since in our implementation, the WebAssembly programs do not deallocate spaces, WebAssembly uses more memory as expected. Memory leaking is a big issue for the programs involves many array copy. In the Conway program, it not only causes WebAssembly uses 88 times more memory but also slow the program down a lot as the WebAssembly program needs to keep request more WebAssembly pages.

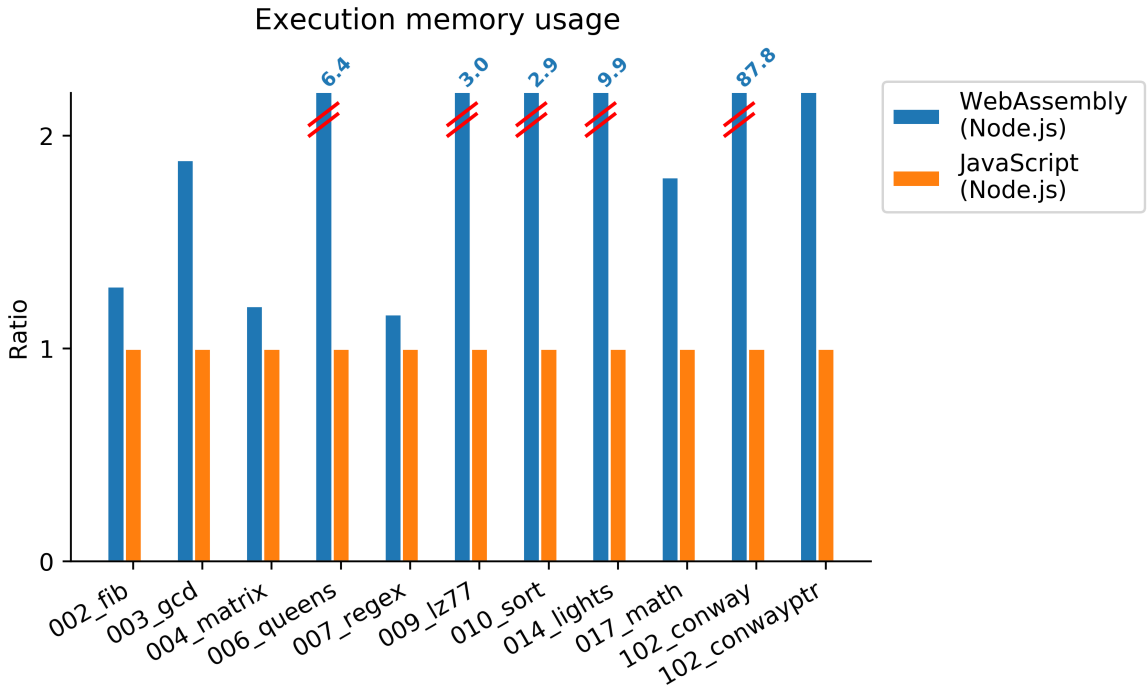

Figure 5.2: Execution memory usage comparison

Chapter 6

Conclusions and Future Plan

6.1 Conclusion

In this project, we design and implemented a Whiley Compiler back-end plugin called `Whiley2Wasm`, which can translate Whiley intermediate representation into WebAssembly Text Format. With `Whiley2Wasm`, Whiley programs can be compiled into WebAssembly and run in browsers across operating systems and platforms with a good performance.

`Whiley2Wasm` supports almost all Whiley syntax and semantics. It supports all the Whiley expressions like arithmetic expressions, array/record expressions, lambda expression except open-records. Since WebAssembly does not have built-in arrays and records, we use WebAssembly memory and related instructions to implement all the array/record expressions. `Whiley2Wasm` also supports all the Whiley statements like assert statement, assignment statement, flow control statements. Whiley has a powerful type system. It not only integrates some uncommon types like union types, recursive types, but also employ flow typing and structural typing. `Whiley2Wasm` fully supports Whiley type system by utilizing tagged union to determine the type of a union at run-time.

We have tested our project on a test suite consisting 652 tested cases provided by Dr David Pearce. We have passed 94.1% test while 0.2% failed. We also skipped 5.7% test cases of which most are limitation in current version of Whiley Compiler. We compared performance between WebAssembly and JavaScript using WyBench benchmark suite on Node.js, where WebAssembly is 20%-30% faster than JavaScript.

6.2 Future Work

Whiley2Wasm can be taken further in many ways:

- Resolving the memory leaking issue. For simplicity, we make an assumption that memory leaking is allowed. From the benchmark testing we can see, in most case, the programs performs well. However, in the Conway program, the WebAssembly program is 150 times slower and use 88 times more memory. A potential solution could be implementing reference counting. Since Whiley Compiler prevents reference cycles, implementing reference counting could work well.
- Implementing unbound integers. In this project, we assume Whiley integers are all i32. The future work could implement an unbound integer library like Java `BigInteger` class and use that to replace all current integer operations.
- Optimization. The current implementation does not do any optimization to the generated code. An extra phase can be added to `Whiley2wasm` to perform optimization on

the intermediate AST.

- Generating glue code to integrate with JavaScript. WebAssembly programs cannot run directly in browsers and we need to write glue code to invoke WebAssembly programs explicitly. `Whiley2wasm` can be extended to generated such glue code directly and make the generated programs easy to use.

Bibliography

[1] Binaryen. https://github.com/WebAssembly/binaryen.

[2] Wabt: The webassembly binary toolkit. https://github.com/WebAssembly/wabt.

[3] Webassembly. http://webassembly.org.

[4] Webassembly specification. https://webassembly.github.io/spec/core/index.html.

[5] Whiley2javacompiler. https://github.com/Whiley/Whiley2JavaCompiler.

[6] Whiley2javascript. https://github.com/Whiley/Whiley2JavaScript.

[7] Whileycompiler. https://github.com/Whiley/WhileyCompiler.

[8] DAVID HERRERA, HANFENG CHEN, E. L. L. H. Webassembly and javascript challenge: Numerical program performance using modern browser technologies and devices.

[9] HAAS, A., ROSSBERG, A., SCHUFF, D. L., TITZER, B. L., HOLMAN, M., GOHMAN, D., WAGNER, L., ZAKAI, A., AND BASTIEN, J. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 185–200.

[10] JANGDA, A., POWERS, B., BERGER, E., AND GUHA, A. Not so fast: Analyzing the performance of webassembly vs. native code, 2019.

[11] LAMEED, N., AND HENDREN, L. Staged static techniques to efficiently implement array copy semantics in a matlab jit compiler. In *Compiler Construction* (Berlin, Heidelberg, 2011), J. Knoop, Ed., Springer Berlin Heidelberg, pp. 22–41.

[12] MICHA REISER, L. B. Accelerate javascript applications by cross-compiling to webassembly.

[13] MIN-HSIEN WENG, BERNHARD PFAHRINGER, M. U. Static techniques for reducing memory usage in the c implementation of whiley programs.

[14] PEARCE, D. The whiley language specification.

[15] PEARCE, D. Getting started with whiley.

[16] PEARCE, D. Wybench. https://github.com/Whiley/WyBench, 2019.

[17] POWERS, B. Browsix - bringing unix to the browser. https://https://github.com/plasma-umass/browsix, 2019.

[18] SLATER, C. Developing a whiley-to-javascript translator.

[19] ZAKAI, A. asm.js. `http://asmjs.org/`. Accessed: 2019-09-30.

[20] ZAKAI, A. Emscripten: An llvm-to-javascript compiler, 2011.

Appendix A

Raw Benchmark Testing Data

A.1 Execution Time

	nashorn	Node.js	Wasm
002_fib	4426	3601	2325.35
003_gcd	31540	6786	5911
004_matrix	2471	456.11	391.628
006_queens	12537	1960	1285.726
007_regex	36	0.6	0.3
009_lz77	8435	2844.93	669.295
010_sort	15235	5315	310.32
014_lights	27363	2773	216.49
017_math	3314	327	723.79
102_conway	260	2.96	445.48
102_conwayptr	9337	2275	1722

Table A.1: Execution time (ms)

A.2 Execution Memory Usage

	Node.js	Wasm
002_fib	33.22656	42.94531
003_gcd	31.55469	59.50781
004_matrix	67.49609	81.00391
006_queens	105.1484	1512.574
007_regex	40.01172	46.46094
009_lz77	68.93359	1672.035
010_sort	94.09375	813.2188
014_lights	36.16797	652.75
017_math	33.51563	60.49609
102_conway	36.07422	964.1719
102_conwayptr	64.32813	145.484375

Table A.2: Execution memory usage (MB)