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

Developing a Whiley-to-JavaScript Translator

Cody Slater

Supervisors: David J. Pearce, Roman Klapaukh

October 18, 2015

Submitted in partial fulfilment of the requirements for Bachelor of Engineering with Honours in Software Engineering.

Abstract

Executing Whiley in the web browser requires a server for processing. With many people using Whiley in the web browser simultaneously, this server can become very slow. JavaScript is programming language designed to execute in the web browser without a server. This project designed and implemented a translator for turning Whiley programs into JavaScript. Two web applications, *Minesweeper* and *Conway's Game of Life* have been built in Whiley then executed in the browser using translated JavaScript and HTML5.

Acknowledgments

- David and Roma, for constantly guiding me through this project and providing me with invaluable feedback and support.
- Mike and Trish, for always supporting me
- Rachel, for taking the time to proof read for me
- Emma, for giving me a reason to become a better person
- Maryanne, for helping me through hard times
- Melanie , for being there through the final weeks
- Reddit and LoL for endless distractions and keeping me sane

Contents

Chapter 1

Introduction

JavaScript is a programming language created in 1995 and is most commonly used in the web browser. It acts as a way to interact with the user and change the contents of web pages [3]. No install is required to run JavaScript in a web browser. It executes client side with no need for help from a server. JavaScript is a common programming language designed to make web applications. Some successful implementations of languages being translated to JavaScript include Google's *Dart* [4], Microsoft's *Typescript* [6] which is a typed version of JavaScript and *CoffeeScript* [1] which is a language similar to JavaScript but with simpler syntax.

Whiley is a part functional, part object oriented programming language created by David J. Pearce [9]. Whiley can be executed in the browser with a web page called *Whiley Play* [7], although it requires a server to execute. The Whiley language is one of the core components used in teaching the second year paper `SWEN224`: *Formal Foundations of Programming*. During lab sessions for this class, the number of students using the *Whiley Play* website can reach over 50. This can make the server very slow.

Wyscript is a similar project that was completed as a summer scholarship. It aimed to compile a simplified version of Whiley to JavaScript and was completed successfully. If Whiley programs were translated to equivalent JavaScript, they could have the portability of JavaScript and the language features of Whiley. This would in turn take load off the server that runs *Whiley Play*. These Whiley language features will be discussed in Chapter 2.1.

Whiley is compiled to an intermediate language called the Whiley Intermediate Language (WyIL)[8]. Using Java and the Whiley API these bytecodes can be translated to equivalent JavaScript. However, there are challenges when converting the WyIL to JavaScript. Those that have been identified are:

- WyIL is an unstructured bytecode language, whereas Javascript is a structured language. This makes conversions of control flow statements difficult,
- Whiley is a type rich language which supports union typing, whereas JavaScript is a loosely typed language and variable types do not need to be declared,
- Whiley supports function overloading, whereas JavaScript does not,
- Whiley supports unbounded integers, whereas JavaScript uses 64 bit floating point numbers.

These difficulties needed to be taken into account when completing the project.

This project has be evaluated through four different ways. First, using the existing Whiley Compiler testing suite, we can determine how many of the language features have successfully translated to JavaScript. Secondly, we can also evaluate how many bytecodes

from the WyIL have been successfully converted to JavaScript. Third, there are existing Whiley benchmarks that can be translated and tested on. A subset of these benchmarks have been executed and timed as a performance comparison between Whiley and JavaScript. Finally, two larger Whiley benchmarks, Minesweeper and Conway's Game of Life, have been translated into the web applications that run in the browser. This demonstrates Whiley running in the browser with some graphical output.

1.1 Contributions

This project created a evaluated a Whiley-to-JavaScript translator to support Whiley executing in the browser. The main contributions are:

1. The design and implementation of a Whiley-to-JavaScript translator written in Java
2. An evaluation of the translator done with the existing Whiley test suite and benchmarks
3. A demonstration of a Whiley program translated to JavaScript running in the browser

Chapter 2

Background

This chapter outlines the background knowledge needed to understand Whiley and JavaScript and the differences between the languages. It also provides some research done into other programming languages that translate to JavaScript.

2.1 Whiley

Whiley is an open source programming language that has been under development since 2009 by David J. Pearce at Victoria University of Wellington [9]. It was made with an emphasis on automatically eliminating run time errors like *out-of-bounds errors*, *null pointer exceptions*, *division-by-zero*, etc. It has since been used as a research tool for code verification.

2.1.1 Key Features of Whiley

Whiley has a number of standard types which are *numbers*, *booleans* and *null references* and also provides more complex types like *unions*, *records*, *tuples* and *lists*. Figure 2.1 shows an example of the absolute function implemented in Whiley.

Each variable declared in a Whiley program must have a valid type. This type information can then be used at run time in the form of type tests. Whiley does type tests with the `is` operator, which compares a variable with a type. This statement becomes particularly useful when used in conjunction with union types. Figure 2.2 shows the is operator with a union type.

The Whiley programming language supports unbound integers. This means that numbers with type `int` have no upper or lower limit on their value.

```
0 public function abs(int x) -> (int r)
1 ensures x >= 0 ==> r == x
2 ensures x < 0 ==> r == -x:
3     if x < 0:
4         return -x
5     else:
6         return x
```

Figure 2.1: The absolute function written in Whiley

```
0 function example(int|real x) -> int:
1     if x is int:
2         return x
3     return 0
```

Figure 2.2: An example function that uses the `is` operator and `union` types

2.1.2 Key Features of WyIL

Whiley compiles to a register based intermediate language called the Whiley Intermediate Language (WyIL) [8]. This language is very similar to the Java Bytecode language and is used in the process of verification. The WyIL is a simplified version of the Whiley language. One line of Whiley code could turn into two or more lines of WyIL code. When converting from Whiley to the WyIL, all conditional control flow is converted into unstructured control flow made up of jumps and labels. The previous Whiley example translated to WyIL bitecodes can be seen in Figure 2.3.

Figure 2.3 is the absolute function translated to WyIL bytecodes. Each bytecode has a name to identify it, followed by some operation and sometimes has a type. For example, line 1 is the `const` bytecode which stores a constant into a register. In this case the value 0 is being stored in register %1. An example of a conditional jump operation is on line 2. This instruction, `ifge`, checks the value of the first register is less than the second register, if they are it will jump to label0 on line 6, otherwise it will continue execution on line 3.

The Whiley absolute function in Figure 2.1 uses `ensure` statements to control behavior of its variables. These statements have been omitted from the translated WyIL code in Figure 2.3. This is because it is assumed that when Whiley code is first compiled to the WyIL, the Whiley theorem prover has proved the validity of these statements. Therefore these statements have not been translated by the translator.

WyIL has the same types as the Whiley language and has bytecodes made for runtime type tests. Conditional logic like `if` statements are translated into jumps to a specific label. This label can be in both forward and backward in execution. Like Whiley, the WyIL inherently supports unbound integers.

2.2 JavaScript

JavaScript is an interpreted scripting language with object-oriented capabilities [3]. It syntactically resembles Java, C and C++, but that's where the similarities end. JavaScript is a dynamically typed language, meaning that variable types do not need to be specified when declared. When creating a variable the `var` keyword is all that is needed [2]. Variables take the value of seven different types in JavaScript, these types are: *number*, *string*, *boolean*, *null*, *symbol*, *undefined* and *object*. When a variable hasn't been assigned a value, it is undefined. Figure 2.4 example shows some of these data-types being used.

JavaScript's object-oriented inheritance is prototype base. This is much different from Java or C++ which use explicit class hierarchy. Figure 2.5 shows how JavaScript's prototype inheritance works. Using the prototype system, a user defined object can be created. In this example an `object` type called Integer has been created. These object can hold properties of values and functions. During runtime JavaScript's `instanceof` operator can check if an object is of a certain defined prototype object. This is similar to the Whiley `is` operator.

The JavaScript code in Figure 2.5 code first creates a new function object called `Integer`. An instance object can be created by using the `new` keyword like on line 7. On line 3 a

```
0 body:
1     const %1 = 0
2     ifge %0, %1 goto label0
3     neg %2 = %0
4     return %2
5     goto label5
6 .label0
7     return %0
8 .label1
9     return
```

Figure 2.3: The absolute function in WyIL.

```
0 function test(){
1   var num;
2   num = 10;
3   var x = true;
4   var str = "Hello World";
5   var object = {x: 1, y: 2.3};
6 }
```

Figure 2.4: Some example JavaScript data-types being created.

function called add is added to the `Integer` object. This function returns a new `Integer` object that adds a number to itself. A call to this add function can be seen in on line 8.

An important thing to note is that JavaScript does not allow `goto` statements or jumps forward in execution, but does allow jumps backwards in execution. This will in turn make translation unstructured control flow difficult. Figure 2.6 shows an example of a jump backwards in execution.

Figure 2.6 is an example of a loop that terminates when `x` equals 10 that is created with a `label` and `continue` statement. On line 2 a `label` called top is created. On line 5 the `continue` statement jumps execution to the label top on line 2.

```
0 Integer = function(val){
1   this.val = val;
2 }
3 Integer.prototype.add = function(other){
4   return new Integer(this.val + other);
5 }
6 function test(){
7   var num1 = new Integer(10);
8   var num2 = num1.add(2);
9   num2.val;//val is 12
10  num1 instanceof Integer;//true
11 }
```

Figure 2.5: Some example JavaScript code showing the prototype mechanism

```
0 function test(){
1   var x = 1;
2   top:
3   if(x < 10){
4     x++
5   continue top;
6   }
7 }
```

Figure 2.6: An while loop that counts to 10 written with JavaScript labels and jumps

2.3 Related Work

This related work section highlights a recent Whiley project and some modern programming languages that compile to JavaScript. The language discussed are *Wyscript*, *Dart*, *TypeScript* and *CoffeeScript*. The target language of these programming languages is the same as the Whiley-to-JavaScript translator. The language that they translate from are much different than the WyIL language. This is because they are all structured languages that can be made by people, whereas WyIL is an unstructured language that is generated from Whiley.

2.3.1 Wyscript

Wyscript was a summer scholarship project completed by Daniel Campbell over the summer of 2014 - 2015. *Wyscript* is a derivative of the Whiley language which has been significantly simplified. The purpose of *Wyscript* is to compile to JavaScript and run client-side in the web browser. An interesting method used in development was the use of a JavaScript run-time file that holds information about Whiley types, binary operations, equals operations and casts. The run-time file uses the prototype inheritance method mentioned above to achieve this. This type of run-time file could also be used in development of Whiley-to-JavaScript. The main difference between the *Wyscript* project and the Whiley-to-JavaScript project is that *Wyscript* does not translate from the WyIL. As mentioned before, *Wyscript* translates a simplified version of the Whiley language straight from the source code. This makes translating much simpler as anything relating to control flow can be easily translated from a structured language to another structured language. This was a major difficulty that was encountered during the Whiley-to-JavaScript project was the translation of unstructured control flow.

2.3.2 Dart

Dart is Google's programming language which started development in 2011 with the purpose of providing a better structured language for creating web applications [4] . Google had two things in mind when creating the *Dart* language; better performance and better productivity [10]. *Dart* was made from the start to compile to JavaScript so it can be run in any modern browser. Each of *Dart's* programming features were made with logical conversion to JavaScript in mind. An important thing to note is that *Dart's* syntax is optionally typed. Optionally typed means that it is up to the user if they want to include type declarations or not. This is an example of a language similar to Whiley that can compile to JavaScript.

2.3.3 TypeScript

TypeScript was created by Microsoft and started development 2012 with the purpose of creating a version of JavaScript that can be used to build and maintain large programming

projects [6]. It is a strict superset of JavaScript which adds in optional static typing and class-based object-oriented programming. *TypeScript* is syntactic sugar for JavaScript, meaning that the extra features that *TypeScript* adds are to achieve better code understandability and quality. This means that every valid JavaScript program is a valid *TypeScript* program, but not necessarily vice versa.

2.3.4 CoffeeScript

CoffeeScript was created by Jeremy Ashkenas and started development in 2009 with the purpose of taking the good parts of JavaScript and redefining them in an easy-to-use way [1]. *CoffeeScript* redefines the syntax and overall feel of JavaScript into a more Python or Ruby style [5]. Some of the changes *CoffeeScript* makes to the JavaScript language is the removal of global variables, replacement of the `function` keyword with `->`, optional use of brackets and the removal of curly braces. *CoffeeScript* is an example of a language that is not typed, but takes the good parts of JavaScript and helps make them better.

Chapter 3

Design

This chapter covers the design process and decisions made for the Whiley-to-JavaScript translator. The goal of this project was to create a piece of software that translates a WyIL file into JavaScript and executes that JavaScript inside a JavaScript environment (ie: in a web browser). This goal was achieved by writing a translator in Java. Java is the target language for the translator because the Whiley compiler and API are written in Java, so it was the obvious choice. The translator iterates through each function and method of a WyIL file and translates them into equivalent functions in JavaScript. Each bytecode found in these functions and methods are translated to valid JavaScript statements. Some bytecodes are harder to implement than others and require a runtime library to provide additional support. These hard to implement bytecodes and runtime library will be discuessed further below.

3.1 Approach

The goal of this project was to execute Whiley programs in the web browser. This project could be completed in two different ways: by translating WyIL bytecodes into a JavaScript file or to interpret the WyIL bytecodes with a JavaScript program. Each method has its own advantages and disadvantages.

To translate WyIL bytecodes to a JavaScript file, an external program written in Java is used. This Java program uses the Whiley API to determine what bytecodes are contained in the WyIL file and in what order. It then writes each bytecode contained in the WyIL file into equivalent JavaScript. This newly created JavaScript file can then be executed in a JavaScript environment. This makes development easier as bytecodes can be added incrementally with the most important bytecodes first. After an initial subset of bytecodes have been successfully translated, the generated JavaScript can be tested to see if it passes the testing harness. Translation from WyIL to JavaScript introduces problems around control flow. WyIL control flow is unstructured, meaning with `goto` statements, execution can jump forward or backward. JavaScript is a structured programming language that does not support goto statements. Therefore the translation of unstructured to structured control flow will be difficult problem that needs to be solved.

An alternative approach to building a translator is to interpret a WyIL file. A JavaScript program would be designed and implemented that can read a WyIL file and interpret what each bytecode in the file does. It would then execute the equivalent action in JavaScript in the web browser. This would make translation of unstructured control flow easier to accomplish as it does not have to be translated into structured JavaScript, it can just execute as it goes. Although interpreting WyIL bytecodes could solve a lot of problems that translating the WyIL has, developing a complex piece of software in JavaScript is challenging. Translat-

ing bytecodes using Java, on the other hand, can fully utilise the Whiley API that can decode a WyIL file easily. This reason made interpreting JavaScript out of scope for this project and a Whiley-to-JavaScript translator written in Java was chosen.

3.2 Overview of Translator

The current process of running a Whiley file is to first compile it to a WyIL file. This WyIL file is then turned into a Java bytecodes in the form of a class file and is executed on the Java virtual machine. The goal of this project is to stop this process before the WyIL file gets turned into Java Bytecodes. Instead the WyIL file gets translated into a JavaScript file which can then be executed in some JavaScript environment. You can see this process in Figure 3.1.

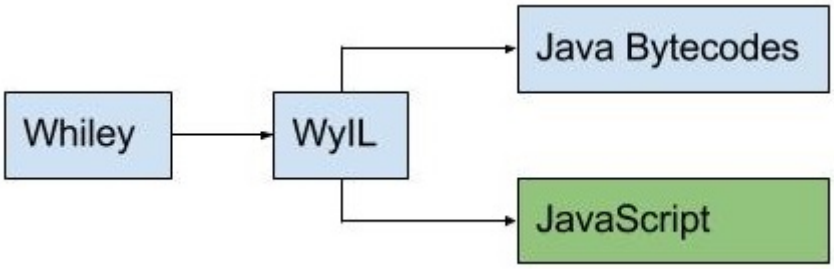

Figure 3.1: The current Whiley process and the new proposed JavaScript process in green

3.3 Whiley Bytecodes

The WyIL language is made up of bytecodes. Each bytecode does a different operation in the Whiley language. There are currently 40 different bytecodes in the WyIL. Translating all WyIL bytecodes is out of scope of this project due to time constraints, so a subset of bytecodes that are the most useful will be translated. The most important bytecodes that have been identified can be seen in Table 3.1. A description of what each bytecode does it also provided.

An important thing to note is that all Whiley tests in the testing suite rely on assert and assume bytecodes. These bytecodes check the value of some variable or function and compare it to the value it should have. If the values are equal execution will continue as normal, but if the values are different a runtime error is thrown with the `fail` bytecode. These bytecodes use an `if` statement when comparing the two values, therefore unstructured control flow is involved. This means that the translation of the WyIL unstructured control flow is crucial to get a testing suite up and running. An example of some WyIL code with an assert statement can be seen in Figure 3.2.

The remaining bytecodes of the WyIL are only translated if they are needed. Most of these bytecodes are not used as often in the Whiley language. These bytecodes include bytes, lambda bytecodes, invariants, and references. The reason invariants will not be translated is because they have no use in the translated JavaScript. When the the original Whiley is translated to WyIL, these invariants are checked with the Whiley Theorem Prover, and will only compile if their invariants are satisfied. Since they have already been satisfied once, it is guaranteed that they will also be satisfied in the translated JavaScript. The other bytecodes mentioned have been left out of the translator because they are relatively unused and meaningful Whiley programs can be made without them.

Bytecode Name	Bytecode Description
Const	Is used to declare a constant value eg: 1 or true
BinaryOperator	Executes binary operations between two variables eg: add
Assign	Assigns a variable from one register to another eh:x = y
Return	Returns from the function with or without a value
Assert and Assume	Used in conjunction with an if statement to check if two variables meet a condition, fails if they do not
Fail	Stops execution and throws a runtime error
If	Compares two variables and jumps to a label if condition is met
Goto	Jumps to a target label
Label	Defines the start of a new block of bytecodes under a label
Invoke	Used to invoke other functions and methods within a WyIL file
Loop	While a condition is true, loop through some bytecodes
NewRecord	Declares a new record object
FieldLoad	Retrieves a certain field of a record object
NewList	Declares a new list object
LengthOf	Calculates the length of a list
Update	Used to update values of composite data-types (lists, records, tuples)

Table 3.1: Important WyIL bytecodes and their descriptions

```
0  public function example() -> void:
1    int x = 1
2    assert x + 1 == 2
```

```
0  public function example():
1  body:
2    const %1 = 1
3    assign %0 = %1
4    assert
5    const %2 = 1
6    add %3 = %0, %2
7    const %4 = 2
8    ifeq %3, %4 goto label0
9    fail
10 .label0
11   return
```

Figure 3.2: **Left**:An example `assert` statement in Whiley, **Right**: the equivalent WyIL `assert` bytecode and `fail` bytecode

3.4 Runtime Library

To fully translate the type-rich language of Whiley and to do complex operations like runtime type checks and casting, a helper library is needed. This library is attached to the translated JavaScript file and from it different Whiley types can be created. This library uses JavaScript's prototype inheritance mentioned in the background chapter. The library also provides methods for complex operations like checking for equality between two objects, doing operations on `lists`, `records` and `tuples`, casting an object to another and runtime type tests.

The main feature of the Runtime Library is to provide a place to define and create Whiley data-types. Because JavaScript objects do not have the same type system of Whiley, a replica of Whiley types were created inside of the library. Each data-type is turned into a JavaScript object with different properties for its type and value. More complex types like `records` have different properties than primitive data-types. `Record` objects hold 3 different properties, 1) an array of values, 2) an array of names and 3) a type. Each type defined in the runtime library follows similar structures.

The runtime file has been attached to the translated JavaScript as an outside library contained within its own file. This file is then linked to the translated source file to provide runtime support. An alternative approach is to attach it to the bottom of each generated JavaScript file. This would make the files very large and harder to understand. Due these reasons it was attached as an external library.

3.5 Unbound Arithmetic

The Whiley language supports unbound arithmetic. This means that numbers have no upper or lower limit. JavaScript does not support unbound arithmetic and has 64 bit floating point numbers. To successfully translate Whiley to JavaScript a method to support unbound integers in Whiley is needed. Two different libraries were found that allow JavaScript to support bigger numbers. These libraries are *Math.js* and *Big.js*. *Math.js* is an extensive math library for JavaScript. This library supports almost every mathematical function as well as very large numbers. To support big numbers it introduces a new data-type called `BigNumbers`. *Math.js* is widely used in the JavaScript community and is being maintained and updated by the creators. The only problem with *Math.js*s is that it is a massive library with many dependencies. The part of code that handles large numbers actually uses a library called *Decimal.js*. *Big.js* is a lightweight JavaScript library for arbitrary precision on decimal numbers. The library introduces a new data-type called `Big`. This data-type is unbounded and its upper or lower limit can be changed. *Big.js* is very similar to the library *Decimal.js* mentioned earlier. *Big.js* is a much smaller more efficient version of *Decimal.js*. *Big.js* has its own functions for binary operators and other mathematics, although it is not as extensive as *Math.js*. Due to these reasons *Big.js* was chosen over *Math.js*.

3.6 Testing

There are 498 tests that are provided by the Whiley compiler. These tests have been written to test certain aspects of Whiley, each one testing a different situation. These tests are made to run in JUnit so they can easily be run inside of eclipse. The tests rely heavily on the `assert` and `assume` bytecodes. Whiley will execute some code then verify the output with an `assert` or `assume` statement. These statements throw a runtime error if their condition is not met. If the test runs silently with no errors thrown it is assumed to pass.

Two different tools have been investigated for use in executing JavaScript for the testing suite. These tools are: *Node.js* and *Rhino.js*. The most commonly used tool is *Node.js*. *Node.js* can execute JavaScript files from the command line. *Node.js* is already installed on Victoria University computers so it was extremely easy to set up. The disadvantages of *Node.js* is, it is difficult to execute JavaScript files that depend on other JavaScript files. This means that if some JavaScript code relies on a runtime library, it needs a statement at the top of the file which imports the library for use. The library itself then needs to add each of its functions to an export array.

Rhino.js is an open source implementation of JavaScript written in Java. Using Java and *Rhino.js*, multiple JavaScript files can be executed together. *Rhino.js* can only be executed in a Java environment which makes it particularity slow. The best thing about *Rhino.js* is that it is easy to set up and easy to use for test cases.

Both of these tools were used for testing. *Node.js* was used for testing before the runtime library was introduced as it was very easy to set up and can be invoked by command line. Once the runtime library was introduced, the testing suite switched to using *Rhino.js* as it was easy to add in files that accompany the translated JavaScript. *Node.js* was still used on occasion for use in debugging and running the benchmarks.

3.7 Evaluation

In addition to the testing suite discussed above, Whiley also includes a separate benchmark suite. To evaluate the translators effectiveness in a more real-world situation, these benchmarks were used. In total there are 37 benchmarks. The benchmarks are split into three categories; micro, small and large. All of the micro benchmarks have been attempted to be translated and run to measure how well the Whiley-to-JavaScript translator works. Two of the small benchmarks that produce GUI output have also be translated. The small benchmarks are more difficult to implement as they require a graphical output which has been made in HTML5. These benchmarks have then been run in the browser using the browsers JavaScript environment. To run the mechanics of each benchmark, they require an adaptor. This adaptor calls the translated functions and links its actions to the HTML5 graphical output. The large benchmarks are considered out of scope for the evaluation.

3.8 Design Process

This project was approached with an agile mindset. There has been weekly meetings with both supervisors to measure weekly progress, discuss any roadblocks that have been found and to set new goals to achieve before the next meeting. Because of this agile process, development has been incremental. It started with the most useful bytecodes first, with more complexity added with each iteration. The produced software has been stored on GitHub. This allows easy tracking of week by week progress by checking the commit log. Using GitHub, it is easy to version control. This means that is is easy to roll back to previous versions of the software if necessary. A feature of GitHub that is being utilised is the issues feature. Issues can be raised about bytecodes that have been implemented and still throw errors. This was a key was to track the progress of the translator and to also track bugs.

Chapter 4

Implementation

This chapter discusses key implementation details of the Whiley to JavaScript translator. The implementation of the translator was done incrementally, starting with the most useful bytecodes first and extending to the less important bytecodes. The implementation process can be broken down into these steps:

1. Test suite set up
2. Translate primitive data-types
3. Translate control flow
4. Translate composite data-types
5. Translate type information of data-types
6. Implement function overloading
7. Implement unbound arithmetic

4.1 Test Suite Set Up

To initially start development a testing environment was created. This testing environment uses the existing Whiley test suite as test cases. As discussed earlier, to execute these tests the `assert` and `assume` bytecodes were needed. These bytecodes inherently use unconditional control flow, so this was the first problem to be solved. The implementation of unstructured control flow will be discussed in the control flow section below. To execute the generated JavaScript file, Node.js was executed from the command line using a Java process. This file was monitored for output and if it ran silently the test passes.

4.2 Translation of Primitive Data-types

The Whiley primitive data-types are *Integers*, *Reals* and *Booleans*. To use one of these data-types it must be declared with the correct type. The declaration of one of these data-types turns into two WyIL bytecodes: `Const` and `Assign`. `Const` declares the value of the data-type being made. `Assign` then assigns the value from the `Const` bytecode to the variable being created. The example below shows the translation of these bytecodes to JavaScript.

To effectively test these data-types, some simple binary operators were implemented. To start these operators were simply converted to native JavaScript binary operators. This

```
0 private method example()
1 body:
2   const %2 = 1
3   assign %1 = %2
4   assign %0 = %1
5   return
```

```
0 function example(){
1   var r2 = 1;
2   var r1 = r2;
3   var r0 = r1;
4   return;
5 }
```

Figure 4.1: **Left**: Some example WyIL code, **Right**: The equivalent JavaScript code

included operations such as plus, minus, time, divide, equal, greater than and less than. Although the implementation of these operators started simple, to capture the full language of Whiley type information was needed. To incorporate type information into these operators they were added into the runtime library, this will be discussed in Runime Library section below. JavaScript uses fixed width variables meaning numbers have an upper limit. Whiley has unbound integers, meaning the currently translated JavaScript does not execute like Whiley. This problem will be discussed further in the Unbound Arithmetic section below.

4.3 Control Flow

Control flow refers to changing the flow of execution at run time. In the Whiley language control flow happens with `if`, `loop` and `switch` statements. The nature of the WyIL made this particularly challenging, especially with `loop` statements. When Whiley compiles to the WyIL, its control flow becomes WyIL unstructured control flow. This means that during execution, jumps to other parts of the code, either forward or backward can be made. These jumps happen in the form of a conditional or unconditional `goto` statements. Conditional `goto` statements are jumps that happen when some condition is met. This condition is in the form of an `if` bytecode. Unconditional jumps always happen. In the WyIL file, jumps are always made to some specific label elsewhere in the code. Figure 4.3 shows a Whiley method with control flow and the equivalent unstructured WyIL method.

JavaScript does not support `goto` statements natively. This meant that to implement unstructured control flow, a solution using the JavaScript language would have to be found. To try and solve this problem a JavaScript library called *Goto.js* was found. This library introduces new syntax into JavaScript to define `labels` and `goto` statements. The library comes with a Parser library that translates the new JavaScript syntax into normal JavaScript syntax and executes it. There were two limitations found when using this library: it did not support jumps forward in execution and it relied heavily on the Parser library. The Parser library was reverse engineered to figure out how exactly the `goto` jumps worked. It was found the library converted its own implementation of `labels` to JavaScript native `labels` and its `goto` jumps to JavaScript `continue` and `break` statements. Figure 4.2 shows a simple `loop` made with with `labels` and `continue` statements.

An important thing to note is that JavaScript can only continue and break to a specific label if the label is backwards in execution. After further research it was found that combining these methods with a `switch` statement inside of a `while loop` it was possible to simulate unstructured control flow. This will be discussed in the next section.

4.3.1 Implementation of Unstructured Control Flow

By using a `switch` statement inside of a `while` statement and a backward jump in execution it is possible to translate the unstructured control flow of the WyIL into JavaScript. Fig-

```
function example(){
 var x = 0;
 top://label top
 if(x < 10){
  x++;
  continue top;//jump to label top
 }
}
```

Figure 4.2: An example while `loop` that counts to 10 written in JavaScript with labels and jumps

```
method f(int x) -> int:
  if x > 10:
    return x-10
  else:
    return x
```

```
private method f(int) -> int:
body:
  const %1 = 10
  ifle %0, %1 goto label0
  const %2 = 10
  sub %3 = %0, %2
  return %3
  goto label1
.label0
  return %0
.label1
  return
```

Figure 4.3: **Left**: Some example Whiley conrtol flow, **Right**: The translated WyIL bytecodes

ure 4.4 shows a translated WyIL method using the `if` bytecode and the translated JavaScript. Every method and function in a WyIL file will have its own `while-switch` statement. The way the control flow works is with the variable called *pc* and the `label` called *outer*. The *pc* variable controls which `case` of the `switch` statement should be executed. The label *outer* is made so it is possible to jump backwards to it from inside the switch statement. Whenever execution needs to jump to a new `switch case`, the `label` *outer* will be jumped to. Each `label` found in a WyIL method becomes a `case` in the `switch` statement. The body of the WyIL method will become the default `case` which is executed first.

To replicate the conditional `goto` bytecode of the WyIL, a JavaScript `if` statement is created that is equivalent to the condition on the WyIL jump. If the condition is true, the *pc* variable is changed to the number relating to the label that is being jumped to. The `continue` statement then jumps backward to the *outer* `label`. Execution continues from the *outer* `label` and the new case that relates to the value *pc* was assigned is taken. If the original condition is not true, execution will continue passed the `if` statement as normal. Unconditional jumps do not need an if statement to be translated. The *pc* variable can be changed to a new value and jump to the *outer* `label`. Once type information was added the `ifis` bytecode was implemented in the same way as this. Instead of comparing some condition between variables, the `ifis` bytecode compares a variable against a type. For this to happen the a runtime library function was needed. This function will be discussed in the Runtime Library Chapter.

```
0 private method example(int):
1 body:
2   const %2 = 1
3   ifle %0, %2 goto label0
4   const %4 = 0
5   assign %3 = %4
6   assign %1 = %3
7 .label0
8   return
```

```
0  function example(r0){
1   var pc = -1;
2   outer:
3   while(true){
4    switch(pc){
5     case -1 :
6      var r1 = 1;
7      if(r0 < r1){
8       pc = 0;
9       continue outer;
10     }
11     var r2 = 2;
12     var r0 = r2;
13     pc = 1;
14     continue outer;
15    case 0:
16     var r3 = 3;
17     var r0 = r3;
18    case 1:
19     return;
20   }
21  }
22 }
```

Figure 4.4: **Left**: An example of unstructured control flow in WyIL, **Right**: The translated JavaScript

4.3.2 Implementation of the Loop Bytecode

Replicating a `loop` bytecode compared to the `if` bytecode was more difficult. The first implementation of the `loop` bytecode was attempted with a JavaScript `while` statement that was set to `true`. Inside of the `loop` would be a termination condition that would escape the `loop`. This termination condition is in the form of a JavaScript `continue` statement, like discussed in the previous section. With the current implementation of control flow logic, this caused problems when leaving the `loop` on some jump and not being able to get back in at the same state. Using the Whiley API it was possible to generate a lable map of bytecodes found inside of a `loop` bytecode. When translating the `loop` bytecode, each jump encountered is checked where it was jumping to using the label map. If it was jumping to a point that was inside the `loop`, the code that needed to be executed from the jump was brought inside the JavaScript `while` statement. This technique using the label map solved the problem with conditional jumps inside a `loop` bytecode.

With the current implementation of the `loop` bytecode, embedded loops produced some unexpected behavior. An embedded `loop` is one or more loops inside of each other. Embedded loops are used extensively when running the two Whiley benchmarks used for the evaluation, so they are crucial to get right. The problem arose when the second `loop` had finished executing its `loop` and was terminiating. It then has to jump back to the `loop` outside of it. It had no way to continue executing the outer `loop` in the state that it was left in.

To solve this problem it was found that using the already set up `switch` and `while` statements, it is possible to simulate the `loop` bytecode accurately. When the `loop` bytecode is encountered during the translation of a WyIL file, a new `case` is created in the already made `switch` statement. Execution jumps to the case that was just created, this case will

```
private method example():
body:
 ...
 loop (%0,%3,%4,%5,%6,%7)
  const %3 = 10
  ifge %0, %3 goto label0
  const %4 = 1
  add %5 = %0, %4
  assign %0 = %5
  const %6 = 8
  ifne %0, %6 goto label1
  const %7 = 9
  assign %0 = %7
 .label1
.label0
  return
```

```
function example(){
 var pc = -1;
 outer:
 while(true){
  switch(pc){
   case -1 :
    ...
    pc = -2;//OTO LOOP TOP
    break;
   case -2:
    var r3 = 10;
    if(r0 > r3){
     pc = 0;//LEAVE THE LOOP
     continue outer;
    }
    ...
    if(WyJS.equals(r0, r6, true)){
     pc = 1;
     break;
    } else{
     pc = -3;
     break;
    }
   case -3:
    ...
    pc = -2;
    break;
   case 1:
    pc = -2;
    break;
   case 0:
    return;
  }
 }
}
```

Figure 4.5: **Left**: An example of a `loop` in the WyIL, **Right**: The equivalent JavaScript

become the top of the `loop` and makes sure no variables get reinitialised to their previous values. The `loop` is then written as normal making sure to check for any conditional or unconditional jumps inside. If any jumps are found, using the previously made label map, they are checked to see if they are jumping within or outside of the `loop`. If the jump is within the `loop` the translator remembers to always jump back to the `loop` once the `case` has been executed. Using this technique, no code needs to be written specially inside of the `while loop` like the previous solution. If there an embedded `loop`, both loops are written in separate cases of the switch statement, so jumps between the two loops can happen. An example of a translated `loop` with control flow inside of it is show in Figure 4.5.

4.3.3 Implementation of the Switch Bytecode

The `switch` bytecode was the easier to implement control flow related bytecode. The `switch` bytecode is used in the implementation of the Whiley benchmark *Conway's Game of Life*. To implement the `switch` bytecode, JavaScript `if` statements were used. Each `case`

```
method f(int) -> int:
body:
 switch %0 1->label2,
             -1->label3,
             *->label1
.label2
 const %1 = 1
 sub %2 = %0, %1
 return %2
 goto label1
.label3
 const %3 = 1
 add %4 = %0, %3
 return %4
 goto label1
.label1
 const %5 = 1
 return %5
 return
```

```
function f(r0){
 pc = -1;
 outer:
 while(true){
  switch(pc){
   case -1 :
    if(WyJS.equals(r0 == 1){
     pc = 2;
     continue outer;
    } else if(r0 == -1){
     pc = 3;
     continue outer;
    } else{
     pc = 1;
     continue outer;
    }
   case 2:
    ...
    return r2;
    pc = 1;
    continue outer;
   case 3:
     ...
    return r4;
    pc = 1;
    continue outer;
   case 1:
    var r5 = new WyJS.Integer(1);
    return r5;
  }
 }
}
```

Figure 4.6: **Left**: An example of a switch bytecode in the WyIL, **Right**: the translated JavaScript

inside of the Whiley `switch` statement would become a new `if` statement in a chain of `if else` statements. Since the WyIL already separated the different `cases` in the `switch` statement to different `labels`, each `if` statement just has to jump to its assignment `case`. Figure 4.6 shows the `switch` statement in the WyIL and the translated JavaScript.

4.4 Runtime Library

The runtime library is a helper library that holds information about Whiley data-types and operations on those data-types. It helps with anything that can happen at runtime. The library can be split into four different parts: constructors for specific data-types, functions on those data-types, type information for those data-types and helper runtime functions

4.4.1 Data-Types

The data-type and data-type functions section of the library defines constructor functions for different Whiley data-types and functions for those data-types. All of Whiley's data-types were turned into JavaScript objects. These objects hold a number of properties. Each

object that is created always has a value and type property. Some more complex data-types have more properties. Different functions have been defined that can be called on these objects. For primitive datatypes these functions are operations like add, subtract and cast. For composite data-types these functions are operations for adding and extracting values from their properties. Figure 4.7 shows the JavaScript code creating an Integer object and calling the `add` function. The corresponding runtime library code that defines this object and function is also shown.

Whiley's primitive data-types have been turned into their corresponding JavaScript objects. These data-types only need to be told their value when they are created. An example of a variable creation can be seen in Figure 4.7. Primitive data-types have a number of functions attached to them. These functions are created for binary operators and for casting. With this implementation of the runtime library, only an `integer` can be cast to a `real`.

Whiley's more complex data-type are called composite data-types. Composite data-types include arrays, records and tuples. Each of these data-types have accompanying bytecodes that perform actions like retrieving and changing variables. These data-types and their functions are defined in the runtime library. Like primitive data-types, they are created from a constructor method. The constructor method is different for composite data-types as they have to define their value and their type. This is because the runtime library has to know exactly what type an object is at runtime. This type information is then used for operations for like runtime type tests and casting.

Each composite data-type holds some properties that define it. A `list` object has two properties, its value, which is a JavaScript `array` and a type. A `record` object has three properties, a JavaScript `array` for its values, an `array` for its names and a type. A `tuple` object has two properties, a JavaScript `array` for its values and a type. None of these properties are accessed straight from the translated JavaScript code, they are only manipulated from within the runtime library. To manipulate a composite data-type in JavaScript, the appropriate function must be called.

4.4.2 Data-type functions

Each composite data-type has its own set of functions. Figure 4.8 shows an example of an `array`, `record` and `tuple` being created, and a record value being extracted from a record. All composite data-types have their own equals method. This is because JavaScript cannot determine equality between two `objects`. Instead, these functions define their own equals methods. The *equals* helper method calls these functions which checking for equality. For equality between two composite datatypes to be true, the ordering and value of all of its members must be equal.

When translated to JavaScript, composite data-types are all treated as arrays. Due to Whileys language feature of preserving value semantics, each data-type must have its own clone method. This clone method makes a new object that is equal to the object that is being cloned. Whenever composite data-types are assigned to new values to clone method must be called. This is also true when these data-type care passed between functions. This is to limit the amount of unusal behavior that can happen when two objects share the same data.

The composite data-types have their own specific functions for manipulating their data. These functions correspond to bytecodes in the WyIL. The `list` type has functions to get and set a value at a given index. It has a function to retrieve its length. The `record` type has functions to load and set values depending on which key is given. They also have a function for checking if a key is contained in a `record`. The `tuple` type has one function to load a specific value at a given index.

```
function f(){
  var r0 = new Integer(1);
  var r2 = r0.add(2);
}
```

```
Integer = function(val){
 this.val = val;
}

Integer.add = function(num){
 return new Integer(this.val + num);
}
```

Figure 4.7: **Left**: An example JavaScript function that creates an Integer object and calls the add function, **Right**: The runtime library code that defines the Integer object and add function

```
function example(){
  var r0 = new WyJS.List([1], new WyJS.Type.Int());
  var vals = [0,0];
  var names = ["x", "y"];
  var r1 = new WyJS.Record(vals, names, new WyJS.Type.Record(names,
                [new WyJS.Type.Int(),new WyJS.Type.Int()]));
  var r3 = r1.fieldLoad("x");
}
```

Figure 4.8: An example of how to create composite data-types with the runtime library in JavaScript

4.4.3 Runtime Types

As mentioned earlier, Whiley is a typed programming language and JavaScript is not. To simulate types in JavaScript, each object that is created has a type variable. This variable is an object that is defined in the runtime library called a Type object. Each different Whiley object has its own type object. Primitive types have a type declared for them inherently. This is because they cannot be more than one type. `Lists`, `records` and `tuples` need to have their type declared in the constructor method. This is because they can hold any type of object. Union types also have their own type object. This type object holds which other types it is union with. Union types are defined as a type object that can be many different types at once.

Each type object has a function called subtype. This function takes a Type object as a parameter and returns true if the parameter is the same type as the object. The union type is different from other types and has two type checking functions. One function is for checking supertypes and returns true if at least one of its union types match with the parameter type. The other functions is for subtypes which returns true if all of its union types match with the parameter type. Whenever an object's type is checked against another, the type must be checked if it is a union type. These subtype functions are used in the helper function *is*, which will be discussed below.

4.4.4 Helper Functions

The runtime library has a number of runtime helper functions. These functions range from checking from equality to doing runtime type tests. The most important helper function is the *equals* function. All of the translated Whiley data-types have been turned into JavaScript objects. This means that JavaScript has no way to know two objects are equal or not. The runtime library defines an *equals* function that takes two objects as parameters and returns

```
0  private method f(int) -> int:
1  body:
2   const %1 = 10
3   ifle %0, %1 goto label0
4   const %2 = 10
5   sub %3 = %0, %2
6   return %3
7   goto label1
8  .label0
9   return %0
10 .label1
11     return
```

```
0  function f(r0){
1   var pc = -1;
2   outer:
3   while(true){
4    switch(pc){
5     case -1 :
6      var r1 = new WyJS.Integer(10);
7      if(WyJS.lt(r0, r1, true)){
8       pc = 0;
9       continue outer;
10     }
11     var r2 = new WyJS.Integer(10);
12     var r3 = r0.sub(r2);
13     return r3;
14    case 0:
15     return r0;
16    case 1:
17     return;
18   }
19  }
20 }
```

Figure 4.9: **Left**: Some example WyIL control flow **Right**: The translated JavaScript using the runtime library

if they are equal to each other. A third `boolean` parameter is added to the function change the equals function to not equals. The equals function uses the JavaScript `instanceof` operator to find the type of the objects being compared. Their types are then checked to be equal using the *is* helper function. If they are equal their values are then checked to be equal. For primitive types their values are extracted and compared. For composite data-type, their own equals method is called and the result of that is returned.

The *is* helper function is used to determine if an object is of a certain type. This is a direct translation of the WyIL `ifis` bytecode. The *is* function works like the *equals* function except it compares types instead of values. It uses the JavaScript `instanceof` operator to first determine the type that is being compared to. If the type is a primitive type, the object is checked for that type and result is returned. If the type is a composite type, the composites types subtype method is called and the retsult of that is returned.

Primitive data-types can be compared by using *less than*, *less than equal to*, *greater than* and *greater than equal to* operators. Helper functions were created to implement these functions. These functions can only be called on primitive data-types. The functions take two object parameters which are the obects being compared. A third `boolean` parameter is added to check for equals or not. For example if the *greater than* function is called with its third parameter being true, it will check for greater than and equals to. Figure 4.9 shows how translated control looks when using the runtime library.

The *cast* helper function is used to translate the `convert` bytecode. This function casts a Whiley object to a certain type. In the case of composite data-types, the *cast* function is recursively called on its values.

```
Integer = function(i) {
 this.val = i;
 this.type = new Type.Int();
}
Integer.prototype.add =
        function(o) {
 return new Integer(this.val
                    + o.val);
}
gt = function(l, r, isEq) {
 if (isEq) {
  return l.val >= r.val;
 }
 return l.val > r.val;
}
```

```
Integer = function(i) {
 this.val = new Big(i);
 this.type = new Type.Int();
}
Integer.prototype.add =
         function(o) {
 return new Integer(this.val.add(
                      o.val));
}
gt = function(l, r, isEq) {
 if (isEq) {
  return l.val.gteq(r.val);
 }
 return l.val.gt(r.val);
}
```

Figure 4.10: **Left**: An example JavaScript Implementation of an Integer object, an add function and a greater than function without unbound integers, **Right**: The runtime library for the same functions that uses unbound integers

4.5 Function Overloading

Whiley supports functions overloading. This means functions can have the same name as long as they have different parameters. JavaScript does not support function overloading so an immediate problem arose. A technique called name mangling was used to simulate function overloading in JavaScript. During the translation from the WyIL to JavaScript, each function and method declared without the `export` modifier its name changed in some unique way. The unique identifier used in this case was a generated binary string of the functions types. The reason functions and methods with the `export` modifier are not name mangled is because the `export` modifier means they are going to be used by a foreign function interface. Simply this means that the methods may be called from outside the Whiley file, so if their name was mangled it would be impossible to reference it. The export modifier is used a lot in the Whiley benchmarks.

4.6 Unbound Arithmetic

To implement unbound arithmetic a library called Big.js was used. This library introduces a new data-type called `Big`. The `Big` data-type is unbound and has no upper or lower limit. The Big.js library was implemented inside of the already created runtime library. To do this, the implementations of Integers and Reals were changed to use the `Big` data-type. This meant that functions that use these data-types need to be changed too.

To implement the `big` data-type into the runtime library, the primitive data-type had its value property changed. Instead of being a native JavaScript number, it is now a `big` data-type from the *Big.js* library. This means that anywhere that primitive data-types have their properties used or changed, the implementation had to change to use the new *big.js* library. Specifically this meant the primitive binary functions and helper functions like *equals* or *greater than* needed to be updated. Figure 4.10 shows how the runtime library changed when the *big.js* library was implemented.

Chapter 5

Evaluation

This chapter outlines the evaluation methods used for the Whiley-to-JavaScript translator. The main aim of the translator is to generate JavaScript files that are equivalent to Whiley files and to execute these files in a web browser. The translator has been evaluated in four different ways. First the number of bytecodes succesfully translated by the Whiley-to-JavaScript translator has been discussed. Second, the number of existing Whiley JUnit tests that pass has been discussed. Third, a subset of the micro benchmarks has been translated and its execution time has been measured while executing in Node.js and Rhino. Fourth, two small benchmarks, *Minesweeper* and *Conway's Game of Life*, have been translated and made to execute in the web browser using HTML5 and a canvas to display graphical output.

5.1 Bytecodes

The number of bytecodes in the WyIL is 40. The Whiley-to-JavaScript translator can successfully translate 26 of these bytecodes. Before implementation of the translator started, the WyIL bytecodes were evaluated for their usefulness in the Whiley language. The bytecodes that were considered most useful were listed in Figure 3.1 of the Design section. All of these bytecodes have been translated to JavaScript successfully. Some extra bytecodes were implemented like `convert`, `switch`, `ifis` and `tuples`. These bytecodes were implemented because they were found to help pass more Whiley tests. Some of these bytecodes were also used in the Whiley benchmarks, for example *Conway's Game of Life* needed the `switch` bytecode to run.

The bytecodes that were not implemented are bytecodes that were found to be not as useful as the implemented bytecodes. These bytecodes were not as prominent in the testing suite and were not used in most benchmarks. Some bytecodes that have not been implemented are the `lambda` and `invariant` bytecodes. Some bytecode types not implemented are the `byte` and `reference` bytecode types.

5.2 Tests

There are 498 Whiley tests that have been written for the Whiley compiler. These tests vary in what they do: they all test some aspect of the Whiley language. The tests are being used as an evaluation metric for how much of the Whiley language has been covered in the translator. The tests run in Eclipse using JUnit. The tests will execute some code, and then use `assert` and `assume` statements to check if the code executed correctly. If the tests pass with no running output, it is assumed that the code ran correctly. However if one or

more of the assertions fail, an error is thrown and the program stops running. The way the tests work is by first compiling the Whiley test file into a WyIL file. The WyIL file then gets translated through the Whiley-to-JavaScript translator and a JavaScript output file is generated. The JavaScript file is then run with Rhino.js. If there is no output of running the file (ie: no exceptions are thrown) the test is assumed to pass.

5.2.1 Results

Out of the 498 Whiley tests 383 pass, 102 are errors and 15 are failures. The difference between errors and failures is that errors fail during the translation process and failures fail during the execution process. The errors happen when a specific bytecode that has not been implemented is being tested. The translator will throw a runtime error with a message showing which bytecode failed.

The bytecodes that made the most tests throw errors is the `reference` and `byte` types and the `lambda` bytecode. These bytecode could be implemented using the runtime library. Due to time constraints around this project these bytecodes could not be implemented. Another subset of tests that throws errors is because of recursive types. To solve this error it would take a lot of time changing the existing implemented type system in the translator.

The failing tests are because of translated bytecodes that do not execute as expected in JavaScript. This is due to unusual behavior happening which makes an assert statement fail during execution. Each failing test would need to be looked at and evaluated individually. Due to time constraints this was not possible and effort was put into getting the biggest subset of the tests to pass.

5.3 Benchmarks

There are a number of Whiley benchmarks that are available for use. Each one of them tests more realistic applications of Whiley. There are varying levels of complexity within the benchmarks. They have been classed into three sections: micro, small and large. The micro benchmarks are small Whiley programs that test some algorithm and only uses console statements for output. The small benchmarks become more complex and use a Java interface to show graphical output and sometimes interacts with the user. The large benchmarks have not been used for evaluation for this project due to the time required to implement them in the browser.

5.3.1 Micro Benchmarks

There are 25 micro benchmarks that are part of the Whiley benchmark suite. From these 25, 12 have been successfully translated to JavaScript. Timing harnesses for Node.js and Rhino were created to record the time they each took to execute. The original benchmarks were also run in Whiley using the JVM, these times were compared to the JavaScript running times. The benchmarks were each run 5 times initially as a ramp up. They were times running for an additional 10 times and an average of the execution time was recorded.

The reason the other 13 benchmarks were not used in the evaluation is because they rely on parts of the Whiley language that have not been added to the translator. For example a number of benchmarks used `Strings`, which are currently not supported. A few of the benchmarks are also incomplete and do not contain valid test methods, so they were removed. The benchmarks that were used are shown in Table 5.1 with an explanation of what they do. Note benchmarks marked with a * require an input file to run.

Benchmark Name	Benchmark Description
Average*	Computers the average of a list of integers.
Fib	Computes the first 41 numbers in the Fibonacci sequence.
GCD*	Implements Euclid's algorithm for finding the Greatest Common Divisor of two numbers.
Matrix*	Implements the classical Matrix multiplication algorithm.
Queens	Implements a solution for the N-Queens problem.
Scc*	Implements a variant of Tarjan's algorithm for finding strongly connected components.
Sort*	Implements a simple merge sort.
Lights	A simple implementation of the (British) traffic lights sequence.
Date	A simple Gregorian Calendar implementation.
Bits	Implements an unsigned bit sequence that can be incremented and converted to an Integer.
Reverse	Reverses the contents of an array.
Flag	Constructs the Dutch national flag from an array of colours.

Table 5.1: Micro Benchmarks and their descriptions. Benchmarks marked with a * require an input file to execute

Many of the micro benchmarks read an external file that hold data that is going to be manipulated. These files are classified as *small*, *medium* or *large*. Each data-set file holds 100, 1000 and 10000 data instances respectively. These files are parsed through the Whiley parser library. When translated to JavaScript, these parser libraries cannot be used. Instead the input files are added to the source of the translated JavaScript files in the form of JavaScript data-types. During execution these objects are turned into valid Whiley objects using the runtime library.

5.3.2 Results

The results of running the micro benchmarks in original Whiley and translated JavaScript running in Node.js and Rhino can be seen in Figures 5.1 and 5.2. The figures have been split into two graphs, one for benchmarks that need an input file to be executed and one for benchmarks that do not. Note that the Y axis is on a logarithmic scale. This is because the Whiley benchmarks executing in the JVM finished very fast and benchmarks executing in Rhino finished drastically slower.

All of the Whiley benchmarks that require files to execute ran in less than 0.01 milliseconds, which is very fast compared to JavaScript. This is because the JVM is very optimised at executing code. The translated benchmarks executed in Node.js executed a lot slower than Whiley. This is because JavaScript is interpreted and not as optimised as Whiley running on the JVM. The translated benchmarks executing in Rhino were typically slower than both Whiley and Node.js. This is because Rhino is even less optimised than Node.js. Rhino must run at a less optimised speed when the size of the JavaScript files being executed reaches over a size threshold. When the data sets from the input files were added to the source code this meant Rhino had to execute slower than usual because file sizes grew too large.

It was found that benchmarks that require a lot of function calls to the runtime library ran slower. This can be seen with the *GCD* and *Sort* benchmarks. If the runtime library was added to the source JavaScript, execution time could possibly be reduced in these cases. The large data sets for the *Matrix*, *Scc* and *Sort* benchmarks were left out because their data sizes made the JavaScript file too big.

It was found that Whiley and Node.js have similar execution times on the benchmarks that not not need files to execute. Rhino is the slowest to run on average, taking a long time to complete the *Fib* and *Queens* benchmarks. This is because of the high dependence on the runtime library and also the fact that these programs are very recursive.

Whiley, Node.js and Rhino were found to perform similarly on the benchmarks that test function calls and handling of data-types. This can be seen by the low execution times on the *Lights*, *Date*, *Bits*, *Reverse* and *Flag* benchmarks.

5.3.3 Minesweeper

The first small benchmarks that was made to run in the browser was Minesweeper. This is a simple game that was popularised with Windows. The game consists of a 2D board of squares. Each square is either a bomb, a number that shows how many bombs are touching that square or a blank square. The goal of the game is to reveal every square that is not a bomb. Squares can be flagged to show that a square might hold a bomb.

The original Whiley benchmark runs from a main Whiley file that controls game logic. Game logic is functions like revealing a square, checking how many bombs surround a square and cascading the squares that do not have any bombs surrounding them. The game runs with the help of a Java file called a board adaptor. The board adaptor acts as the bridging point between the Whiley file and Java GUI. This file can call the Whiley files methods and keeps track of the board state. Java Swing is used to display the output of the game and check for user input.

The benchmarks main game logic file was translated to JavaScript with the Whiley-to-JavaScript translator. The Java board adaptor was translated by hand into a similar JavaScript board adaptor. This board adaptor file calls the Whiley methods from the translated game logic file and also keeps track of the current board state. The board adaptor also communicates with the HTML5 web page and changes output depending on the board state. The board adaptor is also responsible for handling user input.

Minesweeper was the first small benchmark that was implemented. While developing this benchmark, some problems in the translators current implementation were exposed. The first problem encountered was the embedded `while` loop bug discussed in the Implementation Chapter. Because the board is 2D array of squares, to iterate through each square the code must loop through height then width. Sometimes these embedded loops would have complex control flow inside of them.

Another problem encountered was the ordering of record values when translated from Whiley to WyIL. When records are translated to WyIL, their names would be alphabetised but the ordering of their values would stay the same. This caused problems when initialising record data-types. The way this problem was solved is to sort the array of members before they are used.

A screen-shot of the finished game running in the browser can be seen in Figure 5.3

5.3.4 Conway's Game of Life

Conway's game of life is a popular game based on evolution. The game is made up of a 2D board of squares. Each square can either be live which is shown by the square being black, or dead which is shown by the square being white. The game follows these basic rules: any live square with less than two neighbours dies from under population, any live square with 2 or 3 neighbours continues living, any live cell with more than three neighbours dies from overcrowding, and any dead square with 3 neighbours becomes a live square by reproduction.

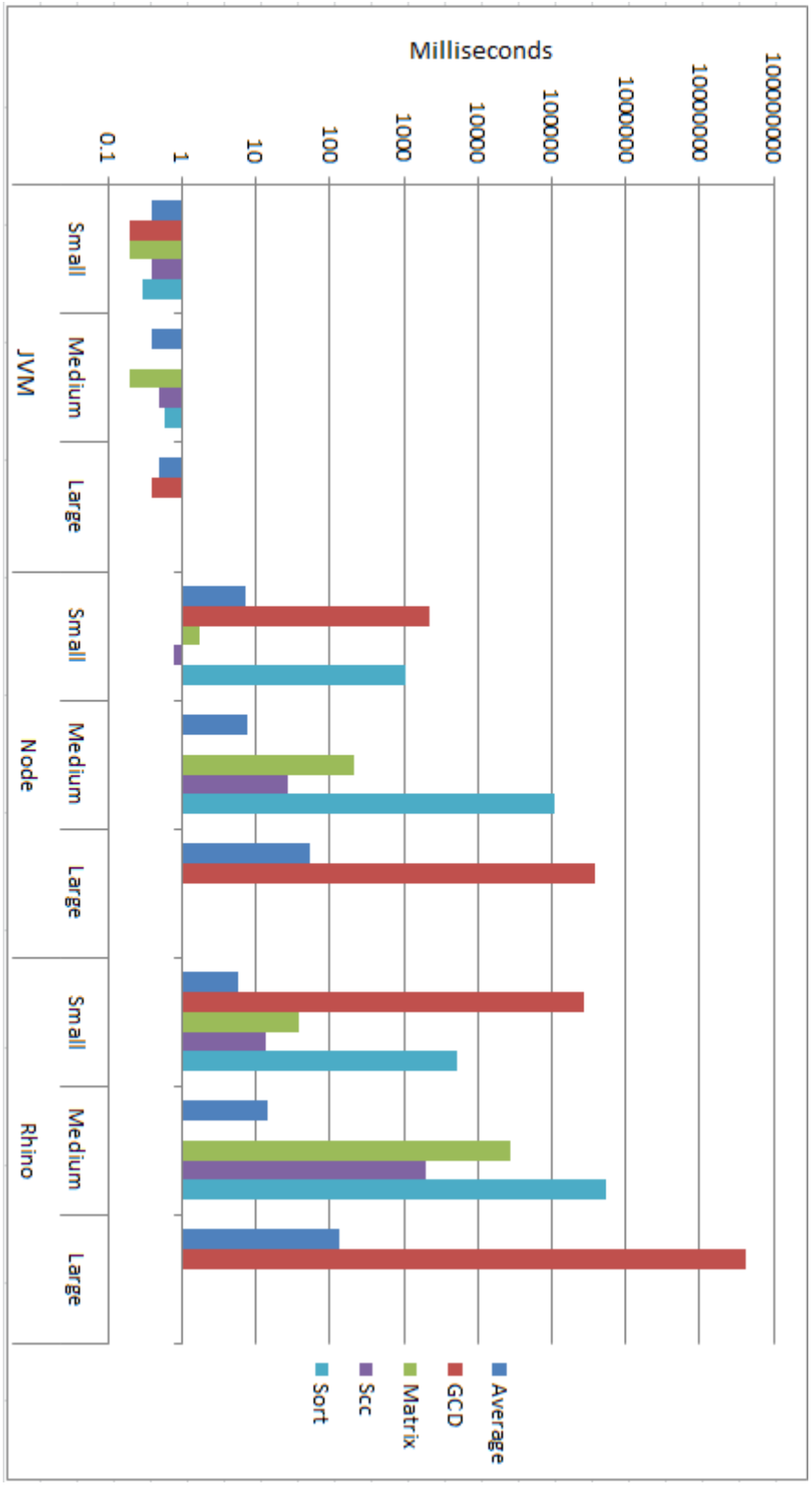

Figure 5.1: The timings of benchmarks that require input files to run.

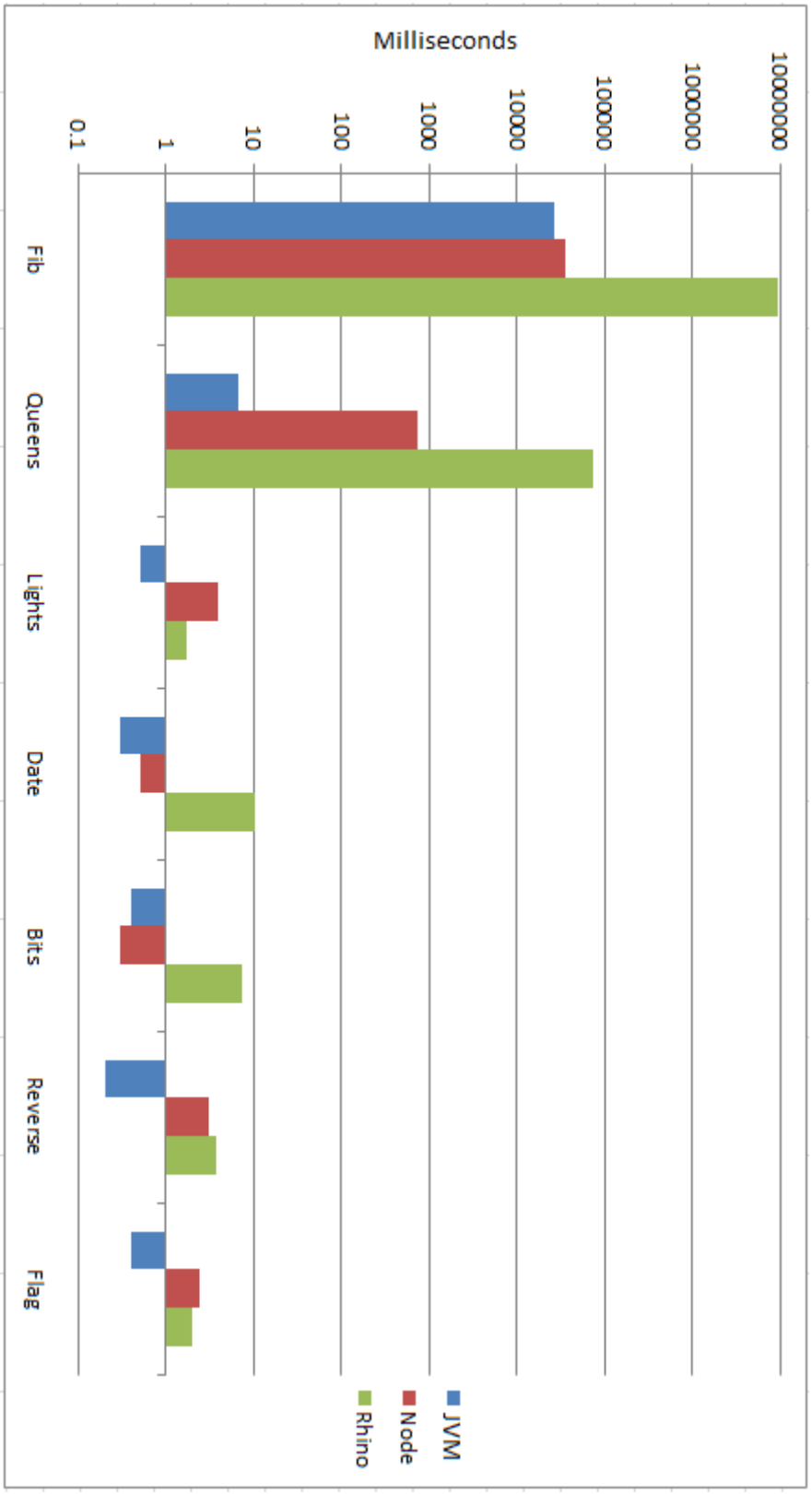

Figure 5.2: The timings of benchmarks that do not require an input file to run

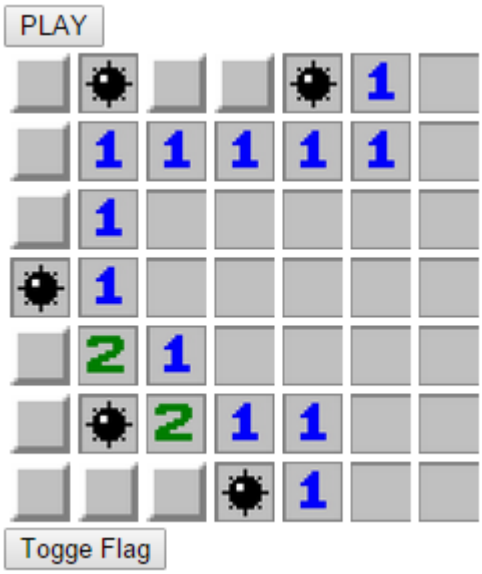

Figure 5.3: A screen-shot of the Minesweeper game running in the web browser

This game had previously been implemented as a benchmark for the Whiley compiler. The benchmark would read an input file that would be the starting state of the board. From then the game would loop and update the board as it goes. To translate this to run in the browser, JavaScript and the HTML canvas was used. Like Minesweeper, the game was made with an adaptor JavaScript file. This file acted as the bridging point between the translated JavaScript and the GUI. The user can start the game by pressing the start button. The game has a start state and then updates the board every .25 seconds. A screen-shot of Conway can be seen in figure 5.2.

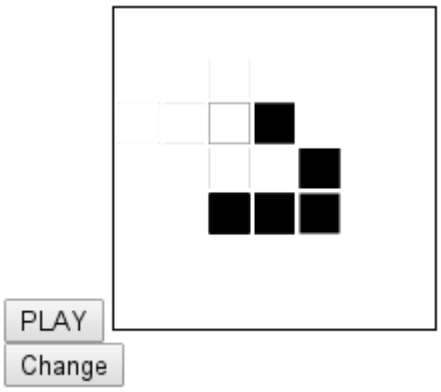

Figure 5.4: A screen-shot of Conway: The Game of Life running in the web browser

Chapter 6

Conclusions

This chapter outlines the future work beyond this project and some conclusions drawn from the project as a whole. This project was set out with the goal of creating a translator that can translate a Whiley program into a JavaScript program. This program should then be able to execute in the web browser.

6.1 Future Work

The completed project opens up avenues for more projects that build on the Whiley-to-JavaScript translator. The main useful bytecodes have been translated from the WyIL to JavaScript, but there are still further bytecodes remaining. The current translation method is not optimised. The current way to run Whiley programs with graphical output in the web browser required manual set up. Taking these considerations into account the following areas have been identified for future work:

- **Whiley library for JavaScript graphical output**. A standard library for linking Whiley and the HTML5 web browser could be created. The library could define Whiley functions that translate to valid JavaScript which can change the contents of a HTML5 web page.
- **Additional Whiley language features**. The remaining bytecodes of the WyIL can be translated into JavaScript. The already implemented bytecodes can be restructured to increase the number of passing tests.
- **Optimisation of current implementation**. The current implementation of the Whiley-to-JavaScript translator is not optimised. Using knowledge of way the WyIL bytecodes are designed and knowledge of the JavaScript language, it is possible to create less JavaScript code that runs more efficiently.

6.2 Conclusions

To recap from Chapter 1.1 the main contributions of this project were:

1. The design and implementation of a Whiley-to-JavaScript translator written in Java
2. An evaluation of the translator done with the existing Whiley test suite and benchmarks
3. A demonstration of a Whiley program translated to JavaScript running in the browser

The design and implementation of the Whiley-to-JavaScript translator was successful. Out of the 40 WyIL bytecodes a total of 26 were successfully translated. The 26 bytecodes have been identified as the most useful subset of WyIL bytecodes to implement. With these bytecodes 383 tests from the Whiley compiler test suite pass.

The design of the translator relies heavily on a helper runtime library. All of Whileys data-type have been created as JavaScript objects inside of this library. A number of helper functions have also been created inside of the library that can help with any type tests at runtime. Using the JavaScript library *Big.js*, unbound Integers were successfully implemented within the runtime library.

A number of Whiley benchmarks have been implemented from the micro and small benchmark sections. These benchmarks run as expected, although can become slow when the runtime library is used a lot. The small benchmarks have been successfully executed in the web browser with graphical output being displayed using HTML5. These benchmarks are *Minesweeper* abd *Conway's Game of Life*.

Bibliography

[1] ASHKENAS, J. Coffeescript. http://coffeescript.org/. Accessed on 07/07/2015.

[2] CROCKFORD, D. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[3] FLANAGAN, D. *JavaScript: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, Incorporated, 2006.

[4] GOOGLE. Dartlang. https://www.dartlang.org/. Accessed on 07/07/2015.

[5] MACCAW, A. *The Little Book on CoffeeScript*. O'Reilly and Associate Series. O'Reilly Media, 2012.

[6] MICROSOFT. Typescript. http://www.typescriptlang.org/. Accessed on 07/07/2015.

[7] PEARCE, D. Whiley play. http://whiley.org/play/. Accessed on 12/10/2015.

[8] PEARCE, D. J. Practical verification condition generation for a bytecode language. Tech. rep., Victoria University of Wellington, 2014.

[9] PEARCE, D. J., AND GROVES, L. Whiley: a platform for research in software verification. In *Springer* (2013), pp. 238–248.

[10] WALRATH, K., AND LADD, S. *Dart: Up and Running*. O'Reilly Media, 2012.