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

Whiley to TypeScript Transpiler

Thomas Rainford

Supervisor: David J. Pearce

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

Abstract

Whiley is an imperative and functional programming language that supports Extended Static Checking through formal verification. TypeScript is a gradually typed programming language that is a superset of JavaScript. The main purpose of TypeScript is to add type safety to the JavaScript language. This project designed and implemented a Whiley compiler backend plugin for translating Whiley into TypeScript that can then be compiled into JavaScript to be executed in the browser. This plugin was evaluated using the existing Whiley compiler test suite, the WyBench benchmark suite and on real web applications such as Conway's Game of Life and Minesweeper.

Contents

1 Introduction **1**
1.1 Contributions . . . 2

2 Background **3**
2.1 Whiley . . . 3
2.2 TypeScript . . . 3
2.3 Safe & Efficient Gradual Typing for TypeScript . . . 4
2.4 Refinement Types for TypeScript . . . 5
2.5 Checking correctness of TypeScript interfaces for JavaScript libraries . . . 6

3 Design **7**
3.1 Design of Existing Translator . . . 7
3.2 Modifications to Existing Design . . . 8

4 Implementation **10**
4.1 Base Types . . . 10
4.2 Union Type Translation . . . 12
4.3 Lambda Translation . . . 12
4.4 Generic Translation . . . 13
4.5 Reference Translation . . . 14
4.6 Import Statement Translation . . . 15
4.7 JS Runtime Declaration file . . . 15

5 Evaluation **17**
5.1 Test Suite . . . 17
5.1.1 Results . . . 17
5.2 WyBench Tests . . . 18
5.2.1 Results . . . 19
5.3 Case Studies . . . 20
5.3.1 Results . . . 20

6 Conclusions and Future Work **22**
6.1 Conclusion . . . 22
6.2 Future Work . . . 22

Chapter 1

Introduction

The Whiley programming language has been in active development since 2009 and uses both the imperative and functional programming paradigms [13][15]. Whiley has an aim to solve the verifying compiler challenge proposed by Prof. Sir Tony Hoare by using extended static checking to eliminate errors at compile time [2][8]. This challenge was proposed to minimise the number of disasters caused by software bugs that have claimed lives and caused substantial financial loss. With a verifying compiler, the correctness of a program can be checked, highlighting errors before the program is executed which can help to prevent failures in production. Whiley follows a lineage of similar existing tools which include EC-S/Java [7], Dafny [9], Spec# [1], Frama-C [4] and Why3 [6]. Software verification is having a growing impact in industry. For example, the Microsoft research group developed Spec# as an object-oriented language that includes verification behaviour [1]. Likewise, Amazon Web Services has incorporated writing formal specifications into their daily work-flow [3].

Whiley provides features for verification that are unique to the language. These include the ability to write specifications for methods and functions, and the ability for the compiler to check that a function or method meets its specification. These are achieved in the language using pre/post-conditions, data-type invariants and loop invariants. A precondition is a condition that is checked before a function is executed using the function's parameters. A post-condition is a condition that is checked after a function is executed using the function's return values. A data-type invariant constrains the specified property of a type. Loop invariants are properties that hold on entry of a loop, for each iteration of the body and on exit of the loop [14].

TypeScript is an open-source gradually typed programming language maintained by Microsoft. It is a superset of JavaScript, meaning a valid JavaScript program is also a valid TypeScript program. The main purpose of TypeScript is to add type safety to JavaScript through static typing. TypeScript is a multi-paradigm language employing object-oriented, functional, generic and imperative paradigms [10]. Type checking can be added to existing JavaScript libraries through TypeScript declaration files. TypeScript's popularity has risen dramatically in the past year. Currently attracting 17 million weekly downloads up from 10 million the previous year, from the popular package manager NPM [12]. Many major JavaScript packages provide TypeScript types such as React, Vue and Express. Many large companies have made the transition from JavaScript to TypeScript including Slack, Airbnb and Google [11]. The main advantage of TypeScript over JavaScript is static typing. This allows for greater readability and IDE features such as auto-completion for a better developer experience. TypeScript also provides common constructs found in other languages such as OOP with classes, inheritance, generics, and decorators closely resembling Java code.

Currently, the Whiley compiler has a plugin for transpiling Whiley into JavaScript. This project developed an extension to the existing plugin which allows valid TypeScript to be generated instead of JavaScript. This means Whiley can now inter-operate with TypeScript programs, which as mentioned above, is becoming an increasingly popular alternative language to JavaScript. Creating a transpiler allows a user to write a program in Whiley where they can utilize Whiley's powerful static checking features. They can then use this code in front-end and back-end web development with knowledge of any errors in the transpiled TypeScript program. Also, code from an existing TypeScript codebase can be selectivley reimplemented in Whiley in order to benefit from static verification.

Through development of this project many challenges have been faced. In particular the translation of Whiley union types into TypeScript union types. In terms of syntax, these types are very similar. However, the challenge comes from the difference in the way these types are tested. In Whiley these type tests are very simple, the **`is`** operator in the condition. In TypeScript such an operator does not exist. Instead, a "type guard" function needs to be created. These functions consist of a parameter of some type, logic for checking the type in the body and a type predicate as the return type. This predicate tells the compiler the type that is expected of the variable passed to the function.

1.1 Contributions

1. A Whiley Compiler plugin was developed that translates Whiley into TypeScript, and then into JavaScript to be executed in a browser.
2. This plugin is an extension of the existing Whiley2JavaScript plugin which was used as a starting point for this project.
3. This plugin was initially evaluated with the existing test suite from the Whiley2JavaScript plugin that was modified to support the new architecture. Several case studies were carried out that tested the plugin with real applications that run in the browser. Finally, the plugin was tested using the WyBench test suite.
4. Bugs were found evaluating the plugin on the initial test suite and the WyBench test suite that have been marked for future work.

Chapter 2

Background

This chapter discusses the Whiley and TypeScript languages as well as related academic work. Although these papers are not directly related to this project they do have some relevance and similarities.

2.1 Whiley

The Whiley programming language began development in 2009. The initial goal is to eliminate common errors, such as null dereferences, array-out-of-bounds, divide-by-zero. Whiley also allows the programmer to write function specifications that are then checked by the compiler. Verification in Whiley has two aspects: the ability to write specifications for functions and methods; and the ability to check the body of a function or method meets the required behaviour set by the specification.

Figure 2.1 illustrates a specification for a function that returns the largest of two integers. The "**ensures**" keyword is used to declare a post-condition on this function. The first post-condition says that at least one of the arguments must equal the return value "z". Therefore, this function must return either the value of "x" or the value of "y". The second post-condition says that the return value can be no larger than either of the arguments.

```
function max(int x, int y) -> (int z)
ensures x == z || y == z
ensures x <= z && y <= z:
    if x > y:
        return x
    else:
        return y
```

Figure 2.1: Whiley function

2.2 TypeScript

TypeScript is a programming language that is a super-set of JavaScript. TypeScript improves the type safety of JavaScript by adding type annotations and static type checking. The Type-

Script Compiler (tsc) generates JavaScript code as its target which can then execute in the browser. This compiler also includes features for type-checking and narrowing.

Figure 2.2 shows an example function in TypeScript. The argument "`arr`" includes a type annotation of "`number[]`". This tells the compiler that this function will only accept an argument that is a number array. Furthermore, the function includes a type annotation for the return type. This tells the compiler that this function must return a value of the type "`number`".

```
function lastIndexOf(arr: number[]): number {
    const lastIndex: number = arr.length - 1
    return arr[lastIndex]
}
```

Figure 2.2: A simple TypeScript function that demonstrates type annotations

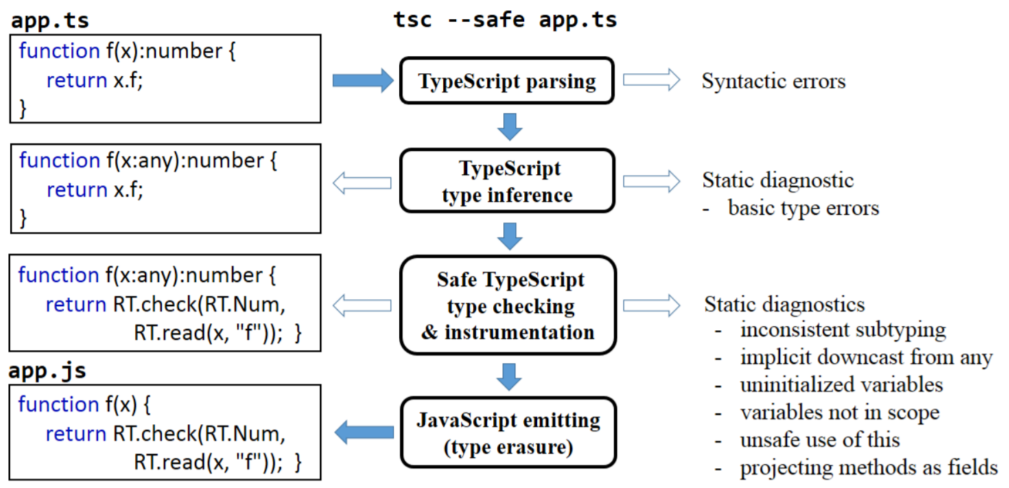
TypeScript gives the programmer the ability to create custom types called type declarations that describe the shape of data. TypeScript includes a feature called type manipulation for creating complex types. Figure 2.3 shows a basic example of type manipulation in TypeScript. The type "`Pick`" creates a type with all specified keys from the given type "`T`". The value "`pickUser`" uses this type "`Pick`" to create a type that contains the keys "`name`" and "`age`" from the type "`User`".

```
type Pick<T, K extends keyof T> = { [P in K]: T[P] }
type User = {
    name: string
    address: string
    age: number
}
const pickUser: Pick<User, 'name' | 'age'> = {
    name: 'John Smith',
    age: 30
}
```

Figure 2.3: A basic TypeScript type manipulation example

2.3 Safe & Efficient Gradual Typing for TypeScript

Existing gradually-typed languages such as Dart and TypeScript don't deal with issues of scale, code reuse and popular programming design patterns. Gradual typing is a type system were type annotations can be omitted and the type of a value is inferred by the compiler. Gradually-typed languages are easier to write and maintain, but do not prevent type errors at runtime. A solution to this called Safe TypeScript, is a compiler that achieves soundness by enforcing stricter static checking and adding runtime checks into the compiled TypeScript code [16]. This code will then be emitted to a JavaScript file. Safe Typescript is a type-checker and code generator for a subset of TypeScript such as JavaScript that guarantees type safety through static and dynamic checks. Safe TypeScript can be used as an extension to the TypeScript compiler by adding a command-line flag that will enable the Safe TypeScript feature. The inserted code adds stricter type checking than that found in the TypeScript compiler.



app.ts

```
function f(x):number {
    return x.f;
}
```

```
function f(x:any):number {
    return x.f;
}
```

```
function f(x:any):number {
    return RT.check(RT.Num,
                RT.read(x, "f")); }
```

app.js

```
function f(x) {
    return RT.check(RT.Num,
                RT.read(x, "f")); }
```

```
tsc --safe app.ts
```

TypeScript parsing
TypeScript type inference
Safe TypeScript type checking & instrumentation
JavaScript emitting (type erasure)
Syntactic errors
Static diagnostic
- basic type errors
Static diagnostics
- inconsistent subtyping
- implicit downcast from any
- uninitialized variables
- variables not in scope
- unsafe use of this
- projecting methods as fields

Figure 2.4: Safe TypeScript Architecture [16]

Figure 2.4 is an example of the TypeScript and Safe TypeScript compilation process. The process consists of three phases with the third phase being the Safe TypeScript step. This is essentially the second phase of type checking after the initial type check from the TypeScript compiler. The third step also illustrates the runtime check code being inserted into the TypeScript file. This inserted code then remains in the emitted JavaScript code.

This paper's focus is not on code transpilling, however, it does use translation for translating TypeScript into Safe TypeScript. This is achieved through the use of an external module called "RT" as seen in the above diagram. The Whiley2TypeScript transpiler uses an external module for various Whiley functions. These functions are then used in the translated TypeScript code. Although the transpiler is not between two different languages the transpiler is used to improve the TypeScript language.

2.4 Refinement Types for TypeScript

Another type-checking extension called Refined TypeScript (RSC) adds a refinement type-system to TypeScript. RSC enables static verification of higher-order, imperative programs. Higher-order programs use functions, objects and modules as values [18]. An imperative program uses statements that change a program's state. The RSC system enables flow-sensitive typing by translating input programs to an intermediate language in static single assignment (SSA) form. Flow-sensitive typing improves type safety and is used, for example, by the Whiley type system. An intermediate language is used by a compiler to represent source code. SSA is an intermediate representation that requires that all variables are assigned once and not defined before their use. RSC supports imperative and dynamic features of TypeScript including overloading and type reflection [18].

Figure 2.5 shows examples of basic refinement types of RSC. These types closely resemble TypeScript types but with an extra condition added after the property declaration. The condition is used for static verification so the value set to the property must return the condition as true. For example, the first type, `nat`, has a property `v` and a condition `0 <= v`.

```
type nat      = {v:number | 0 ≤ v}
type pos      = {v:number | 0 < v}
type natN<n> = {v:nat     | v = n}
type idx<a>  = {v:nat     | v < len(a)}
```

Figure 2.5: TypeScript Refinement Types [18]

This says that v must be a value that is greater than or equal to zero. In other words, the value must be a positive or natural number. This condition would be checked at compile time by the RSC compiler. These types also closely resemble Whiley types as Whiley also makes use of type conditions, known in the Whiley language as data-type invariants.

Much like the previous paper, this paper does not focus on code translation but instead creates an extension to TypeScript to improve the language which requires a translation method.

2.5 Checking correctness of TypeScript interfaces for JavaScript libraries

TypeScript offers support for interacting with existing JavaScript libraries through interface declarations. This interaction allows for static type-checking of the JavaScript code and auto-completion. These declarations are contained in a TypeScript declaration file ".d.ts". This paper presents a way of checking the correctness of declaration files by using JavaScript library code [5].

This declaration file correctness checker operates in three phases

1. Execute the libraries initialization code and extract a snapshot of its state.
2. Check the global object and all reachable objects in the snapshot match the types from the declaration file using a structural type checking algorithm.
3. Carry out a lightweight static analysis of each library function to check that it matches its signature in the declaration file.

This paper does not discuss code translational methods, but it does discuss the use of declaration files and JavaScript libraries. The Whiley2TypeScript transpiler requires a JavaScript module for executing Whiley functions. A declaration file could be created to allow for type-checking and auto-completion of these functions in the generated TypeScript code.

Chapter 3

Design

This chapter discusses the design of the Whiley to TypeScript transpiler. The goal of this project was to translate Whiley into TypeScript that can be compiled into JavaScript to be executed in a JavaScript environment. This transpiler is an extension of the existing Whiley to JavaScript transpiler [17]. This means that an existing architecture is already in place but needed modifying to support the TypeScript language. These modifications extended the existing transpiler to output a TypeScript file and, in addition, compile this file into JavaScript using the command-line TypeScript Compiler (tsc). The main difference from JavaScript is that TypeScript includes static type-checking. The existing transpiler needs extending to translate type information between Whiley and TypeScript.

3.1 Design of Existing Translator

The existing Whiley2JavaScript transpiler is a backend plugin for the Whiley compiler. The outputted IL from the Whiley compiler is passed to the Whiley2JavaScript plugin were it is transilted into a JavaScript IL. This is then translated into JavaScript code and printed to a file. This file can then be executed in a JavaScript runtime environment. The Whiley to JavaScript plugin architecture is illustrated in Figure 3.1

The JavaScript IL is represented as an AST (Abstract Syntax Tree). Figure 3.2 shows an example AST of a variable declaration where the variable "`arr`" is being assigned to an array. When translated into JavaScript code, this IL will look like this; "`let arr = [1,2,3]`".

An alternative design to the architecture of the existing Whiley2JavaScript plugin was considered. This alternative design translated Whiley IL directly into JavaScript code. The current design translates Whiley IL into a JavaScript IL, and then into JavaScript code. The reason for choosing the existing design is because it is easier to extend and modify. Furthermore, because TypeScript can be thought of as an extension of JavaScript, the existing design is an ideal starting place.

The chosen translation method employs an intermediate object structure called an Abstract Syntax Tree (AST). ASTs are common data structures in compilers used to represent the program code. There are some benefits of using an AST. An AST can store extra information about the program, for example, the locations in the code of each element. This allows for more helpful error messages as they can contain the exact location of the error. In this transpiler, the AST is a tree-like structure of objects. These objects represent elements of the TypeScript code that have been translated from Whiley code. This AST is then converted into a string as TypeScript code. Much of the logic for this was already implemented in the existing Whiley2JavaScript plugin. However, the TypeScript plugin extends this by adding type information to the appropriate objects in the AST. Such as functions and variable declarations.

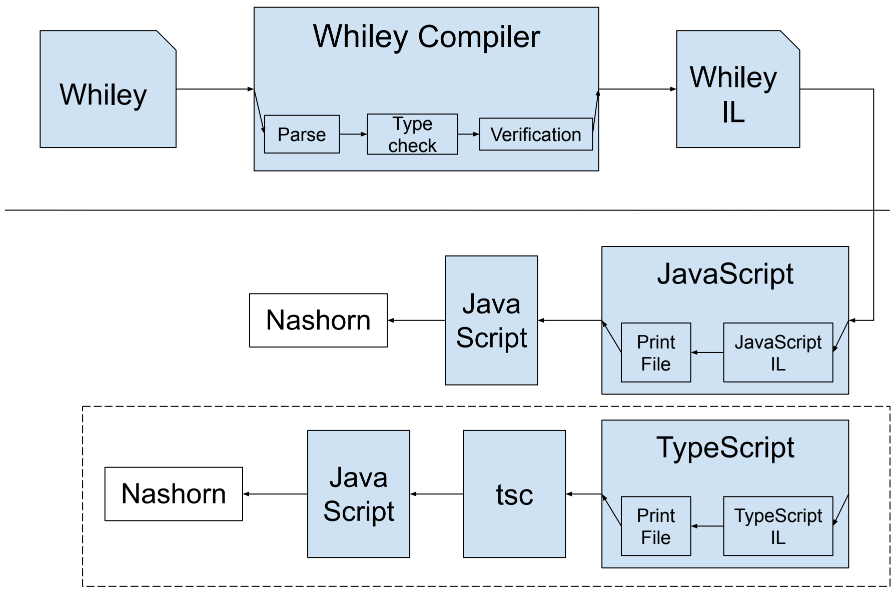


Figure 3.1: This diagram is split into two sections. The top section is the basic architecture of the Whiley compiler. The bottom section is the architecture of the Whlíey2JavaScript plugin and the Whiley2TypeScript plugin highlighted with a dotted box.

3.2 Modifications to Existing Design

A number of modifications to the existing architecture are required. The JavaScript IL needed extending to include type information to the necessary elements such as functions, variables etc. The transpilation process and the file printer were modified to accommodate the changes to the JavaScript IL. The transpilation process needs to translate Whiley types into TypeScript types, and the file printer needs to be able to read the TypeScript IL and print the new type information. The modified architecture is illustrated in Figure 3.1

Figure 3.2 demonstrates an AST of the TypeScript IL. This example shows an extension of a JavaScript AST. This extension is illustrated as a type property on the ”Variable Declaration”. This type property contains the type information of this variable. The type for this variable is an array type. This array type has a property that describes the type of the array elements that this array stores. In this case the array stores elements of type `"number"`.

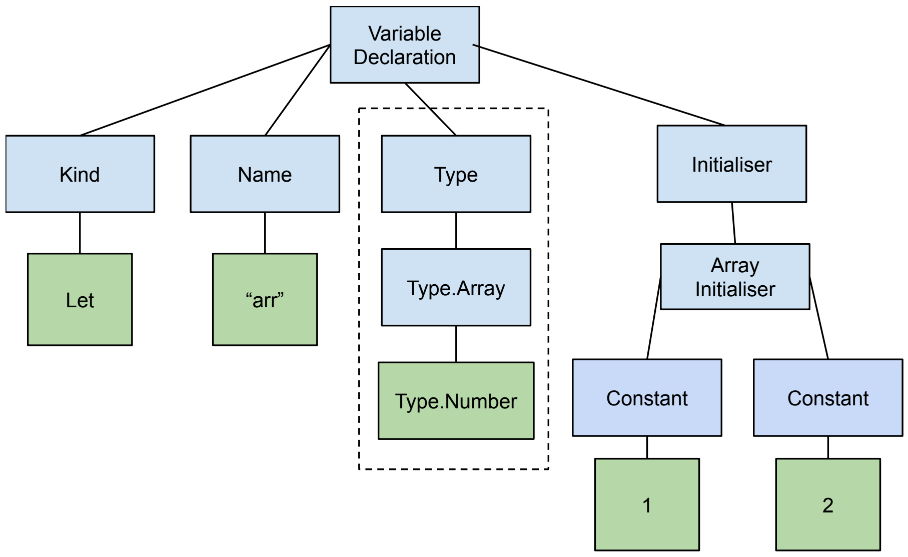


Figure 3.2: This is an example AST for a variable declaration. The dotted box highlights the extension to the JavaScript AST that adds type information to the variable declaration.

Chapter 4

Implementation

This chapter discusses the implementation of the Whiley2TypeScript translator. As the implementation of the translator consists of extending an existing translator, the key extensions to the translator are discussed.

4.1 Base Types

The aim initially was to translate the basic primitive types first, then continue with translating more complex types. These primitive types in TypeScript are used for variable, parameter and return types. First the Whiley type is received and translated into a TypeScript type. For example, the Whiley type **`int`** could be translated into the TypeScript type `number`. Figure 4.1 and 4.2 demonstrates one possible translation of a Whiley function.

```
function sum(int a, int b) -> (int r):
    return a + b
```

Figure 4.1: Whiley function

```
function sum(a: number, b: number): number {
    return a + b
}
```

Figure 4.2: TypeScript function

Whiley integers are larger in size compared to the TypeScript `number` type as Whiley integer types are unbounded. Using the `number` type would limit the integer size you could use in Whiley if you aim to translate that Whiley code into TypeScript as `number` is 64-bit floating point. The solution to this is to use a the JavaScript's `bigint` type as this is unbounded. One problem here is that this type is only supported in ECMAScript 2020 or later. To handle this when a Whiley integer is encountered, the TypeScript compiler version will be checked. If the version supports `bigint` then use it in place of Whiley integer. Otherwise using the `number` type with an appropriate warning being emitted.

Once primitive types were being correctly translated for variable, parameter and return types, more complex translations can be considered. Firstly, this includes the translation of Whiley records to TypeScript type declarations. The syntax between them are conveniently similar, with only minor differences between them. In Whiley this operator is the **`is`** operator and in TypeScript it is `=`. In both languages these declarations can be written in two different ways. One way is to assign a type to another type. For example in TypeScript, **`type`** `A = number`. With this the type alias `A`, can now be used as a `number` type. The second way is to assign a type to an object structure. For example in TypeScript, **`type`** `B = {x: number, y: number}`. The type alias `B` now references an object with two properties `x` and `y`.

```
type A = number
type B = {x: number, y: number}

function sum(b: B): A {
    return b.x + b.y
}
```

Figure 4.3: TypeScript type declarations example

The translation of array types can also be considered. These types are also similar in Whiley and TypeScript which makes the translation fairly simple. The translator will first get the array element type, this type could be another array type of a certain number of dimensions or a single type such as `number` or `boolean`. If the element type is an array type, that means it's a multidimensional array.

Another type translation is the Whiley **`byte`** type. This type is not supported in TypeScript, however the current JavaScript backend contains methods for creating a byte using the `number` type. This makes the translation very simple, because when the **`byte`** type is encountered in Whiley, the type to use is simply `number` and the existing backend code will take care of the logic.

The final basic types to be translated are related to Whiley string types. In Whiley strings are represented as integer arrays. The current JavaScript backend handles this similarly to the byte type. This means the translation for this is very simple as it can just be translated to the `number` type.

Whiley also supports JavaScript strings which are imported into a Whiley program as shown in Figure 4.4. This is also a simple translation as TypeScript has a **`string`** type.

```
import std::ascii
import js::core

ascii::string asciiString = "Hello World"
core::string jsString = asciiString
```

Figure 4.4: Whiley JavaScript string example

4.2 Union Type Translation

The translation of union types was relatively challenging. Adding the union type object to the translator was simple as Whiley and TypeScript represent union type in a similar way. The challenge came from the difference in how Whiley and TypeScript handle type checking of union types. As a union type can be any type given in the union, the value assigned to a variable with a union type must be checked to ensure there are no type mismatch errors. In Whiley this is achieved with a condition that uses the **`is`** operator. For example, `if (a is int)`. TypeScript has a similar operator, `typeof`. However, this is much less powerful and does not operate in the same manner. In TypeScript a function has to be created, called a ”type guard”, were the return type is a predicate that asserts that a given variable has a given type. Figures 4.6 and 4.5 demonstrates a translation of a Whiley union type with a type checking condition. The main difference is the TypeScript program contains a type guard function for checking if a given value is of type `"number"`.

```
type MyUnion is int[] | int | bool;

MyUnion value = 123;
if (value is int):
    value += 1; // Type of value is int
```

Figure 4.5: TypeScript type guard example

```
type MyUnion = string | number | boolean

// Type guard function
function isNumber(x: string | number | boolean): x is number {
    if (typeof x === "number") { return true }
        return false
}

let value: MyUnion = 123
if (isNumber(value)) { value += 1 } // Type of value is number
```

Figure 4.6: TypeScript type guard example. When the function ”isNumber” returns true, the TypeScript assumes that ”x is number” holds.

4.3 Lambda Translation

The translation of Whiley lambda expressions is already implemented in the Whiley2JavaScript transpiler. For the translation of Whiley lambda expressions to TypeScript, the necessary type information needed to be added. Whiley and TypeScript both provide the ability to use function types. Figures 4.7 and 4.8 demonstrate the translation of Whiley lambda expressions to TypeScript functions. Figure 4.8 shows the translation of a Whiley function `type function(int) -> int` to a TypeScript function type `(a: number) => number`.

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

function g() -> func:
    return &(int x -> x + 1)

func f = g()
assume f(1) == 2
```

Figure 4.7: Whiley lambda expression example

```
type func = (a: number) => number

function g(): func {
    return function(): (a: number) => number {
        return function(x: number): number {
                return x + 1;
        };
    }();
}

let f: func = g();
Wy.assert(f(1) === 2);
```

Figure 4.8: TypeScript lambda expression translation

4.4 Generic Translation

Generic types are a common feature in statically typed languages that allow types to be parameterized. Generics in Whiley and TypeScript behave identically and use a very similar syntax. This made implementing the translation relatively straightforward. Figures 4.9 and 4.10 demonstrate the translation of generic types between Whiley and TypeScript. These examples implement a linked list where the link data property is a generic type. A new `LinkedList` record is created in `g` with the generic type assigned to **`int`**. This is translated into `number` in the TypeScript code. Figure 4.10 that uses the `Wy.copy` function. This is explained in section 4.7.

```
type Link<T> is { LinkedList<T> next, T data }
type LinkedList<T> is null | Link<T>

function g():
    LinkedList<int> l1 = null
    Link<int> l2 = { next: l1, data: 0 }
    Link<int> l3 = { next: l2, data: 1 }
```

Figure 4.9: Whiley generics example

```
type Link<T> = {
    next: LinkedList<T>
    data: T
}
type LinkedList<T> = null|Link<T>
function g(): void {
    let l1: LinkedList<number> = null;
    let l2: Link<number> = new Wy.Record(
        { next: Wy.copy(l1), data: 0 }
    )
    let l3: Link<number> = new Wy.Record(
        { next: Wy.copy(l2), data: 1 }
    )
}
```

Figure 4.10: TypeScript generics translation

4.5 Reference Translation

The translation of Whiley references has been implemented in the existing Whiley2JavaScript plugin. As both JavaScript and TypeScript don't implement any pointer to some type `T`, the behaviour can be replicated using objects. The JavaScript runtime contains a constructor function for creating a property `"$ref"` that is assigned to a given value. This property acts as a reference to the given value. For example, `let y = new Wy.Ref(2)` will create a `$ref` property on `y` which is assigned a value of 2. This means that this value can be retrieved by accessing the `$ref` property, `y.$ref == 2`. Accessing the `$ref` property acts as a dereference as seen in Whiley. Figures 4.11 and 4.12 demonstrate the translation of Whiley references to TypeScript. As TypeScript requires type information, a type declaration must be used when creating a reference property. This can be achieved with `type Refer<T> = { $ref: T`.

```
method swap(&int x_ptr, &int y_ptr):
int tmp = *x_ptr
*x_ptr = *y_ptr
*y_ptr = tmp

public export method test():
    &int x = new 1
    &int y = new 2
    swap(x,y)
```

Figure 4.11: Whiley reference example

```
type Refer<T> = { $ref: T }
function swap(
    x_ptr: Refer<number>,
    y_ptr: Refer<number>
): void {
    tmp: number = x_ptr.$ref;
    {
        let $0: number = y_ptr.$ref;
        x_ptr.$ref = $0;
    }
    {
        let $1: number = tmp;
        y_ptr.$ref = $1;
    }
}

function test(): void {
    let x: Refer<number> = new Wy.Ref(1);
    let y: Refer<number> = new Wy.Ref(2);
    swap(x, y);
}
```

Figure 4.12: TypeScript reference translation

4.6 Import Statement Translation

The translation of import statements is an important aspect of the Whiley2TypeScript transpiler as, unlike JavaScript code, TypeScript requires modules to be imported to be considered correct by the compiler. The Whiley language currently has three distinct import statement types. The "`import package.File`" example, imports the "File" compilation unit from the package "`package`". This then allows named entities within the compilation unit to be referenced from a partially qualified name that omits the package component, "`File.Entity`". The "`from`" import statements ("`import Entity from package.File`") import the given named entities from the given compilation unit only. They do not import the entire compilation unit. The "`with`" import statements ("`import Entity from package.File`") import the given named entities from the given compilation unit, and they import the entire compilation unit as well.

TypeScript also includes import statements for importing named entities and the entire module. Figures 4.13 and 4.14 demonstrate the translation of the three Whiley import statement types into TypeScript. You will notice that import translation "3" includes two import statements in the TypeScript code. This is because no import statement that matches the functionality of the "`with`" import in Whiley in TypeScript. However, a combination of two import statements does provide the correct functionality.

4.7 JS Runtime Declaration file

The JS runtime is a JavaScript library that contains functions that imitate Whiley language behaviour. These functions are then used in the translation process to support common

```
import package.File //1
import Entity from package.File //2
import Entity with package.File //3
```

Figure 4.13: Whiley import statements

```
import * as package from './file' //1
import { Entity } from './file' //2
import * as package from './file'; //3
import { Entity } from './file'
```

Figure 4.14: TypeScript import statements translation

Whiley language features. Some examples that are included in this library are a `copy` function that provides a deep clone of values or objects, an **`assert`** function that provides a mechanism for raising assertions and an `equals` function for checking the equality of objects. This library also includes constructor functions to create Whiley record and reference objects.

The JavaScript runtime library is currently used in the existing Whiley2JavaScript transpiler. Using this library as is with the Whiley2TypeScript transpiler would cause some problems. Firstly, referencing a JavaScript library from a TypeScript file would cause type-checking errors in strict compiler modes. As JavaScript is dynamically typed, the TypeScript compiler will not be able to guarantee these types. Secondly, because the compiler will not be able to guarantee the correctness of the JavaScript runtime types, there is no way of checking whether the translation of the Whiley types into TypeScript is correct.

The solution to these problems is to create a TypeScript declaration file. A declaration file is used to give JavaScript libraries type information. This was achieved by creating a type declaration for all properties within the JavaScript library. Figure 4.15 demonstrates an example of the `"equals"` function declaration within the `"Wy"` object which is part of the JavaScript runtime. The `"equals"` function is a property on the `"Wy"` object, so the name of the property is included in the declaration as well as the type information. A problem with declaration files is they are prone to mistakes. For example, if incorrect type information is given to a property within the declaration file, then a programmer could give the correct type but will receive a compiler error. Furthermore, the programmer could give an incorrect type and the compiler will not return an error. Fortunately, the TypeScript compiler contains a feature for generating declaration files from a given JavaScript file.

```
declare Wy: {
    . . .
    equals: <T, U>(o1: T, o2: U) => boolean;
    . . .
}
```

Figure 4.15: TypeScript JS runtime declaration example

Chapter 5

Evaluation

This chapter describes the evaluation methods used for the Whiley2TypeScript transpiler. The aim of the transpiler is to take a Whiley program and generate the equivalent TypeScript program. For a given Whiley program the equivalent TypeScript program should produce the same output. The TypeScript code can then be compiled down to JavaScript and executed in the browser. Three different evaluation methods have been used. The first method used a test suite from the existing Whiley2JavaScript plugin which are originally from the Whiley compiler itself. This tests the translation of specific aspects of the Whiley language into TypeScript. The TypeScript is then compiled into JavaScript to be executed. The second method used WyBench which is a suite of benchmark programs used for testing and demonstrating the Whiley language. Some examples of these benchmark programs include, LZ77 compression/decompression, Matrix Multiplication and N Queens problem. The final method aimed to test the translation of web applications written in Whiley. This posed a number of challenges that simple test cases do not, such as linking dependencies and managing modules.

5.1 Test Suite

The Whiley2TypeScript plugin is tested using 733 test cases from the Whiley compiler test suite. Figure 5.1 shows an example test case. These tests are used to validate specific translation components from the Whiley language. These tests consist of a Whiley file that contains the program that will be tested. Included in the program are one or more **`assume`** statements that are used to test whether a given condition is true. This file is passed through the Whiley2TypeScript plugin and the corresponding JavaScript file is outputted. The **`assume`** and **`assert`** statements from the Whiley file are translated into the **`assert`** function from the JavaScript runtime which takes a boolean value. If this boolean value is false then an Error is thrown. This indicates that the test has failed. The JavaScript file is executed using a JavaScript runtime environment called Nashorn. If any errors are thrown from the executed JavaScript, then the test has failed. Before the JavaScript can be executed, the TypeScript code needs to be compiled. If any errors are returned then the test fails. This ensures that the TypeScript code translated is correct. Correctness here means the code is free of syntax errors, runtime errors and type-checks without error.

5.1.1 Results

Out of the 733 test cases, 46 are ignored as these tests contain Whiley features that are not yet supported by this plugin. **Out of the supported test cases, all but one compiles and executes successfully using the Whiley2TypeScript plugin developed in this project**. The

```
type list is int[]

function index(list l, int index) -> int
requires index >= 0 && index < |l|:
    return l[index]

public export method test() :
    int[] l = [1, 2, 3]
    assume index(l,0) == 1
    assume index(l,1) == 2
    assume index(l,2) == 3
```

Figure 5.1: An example test case from the Whiley compiler test suite that was used to test the Whiley2TypseScript transpiler. This test case tests array access.

test case that fails does so because of a TypeScript compiler error. This test case aims to test recursive types. A recursive type is a data type that contains values of the same type. Figure 5.2 demonstrates a recursive type in TypeScript. The type `Node<T>` contains two properties, `value` and `nodes`. The `nodes` property is an array of the current type. This type describes a tree data structure where a node may contain a collection of other nodes. A null reference to the `nodes` property would indicate a leaf node. The failing test case does not fail because of the in-correctness of the recursive type translation, but instead fails because of the way conditional type checking is translated. A condition is created that is always true which creates unreachable code.

```
type Node<T> = {
    value: T
    nodes: Node<T>[]
}
```

Figure 5.2: TypeScript recursive type.

5.2 WyBench Tests

WyBench is a collection of Whiley program that are representative how Whiley might be used. These programs can be used with different Whiley backend tools for the purpose of testing their correctness. The tests in WyBench start with trivial calculations such as the Fibonacci sequence and increase in complexity. The Whiley2TypeScript transpiler was tested using the WyBench test suite. Each Whiley project was built using the Whiley2TypeScript backend. This generated a TypeScript file which is the translation of the Whiley file included in the project. As TypeScript is just syntactic sugar over JavaScript, the TypeScript file will need compiling down into JavaScript. The result will be a project that now contains a Whiley file, TypeScript files and JavaScript files. To test the output, the code must be executed in a browser. Ideally, the code would be executed using a JavaScript runtime tool such as NodeJS. However, the transpiler is set up to output code that is set up for running in a browser.

5.2.1 Results

The WyBench test suite contains groups of tests varying in size. The smallest group, ”micro”, contains 24 programs that can be used to test the Whiley2TypeScript transpiler. **Out of the 24 tests, 19 passed**. Table 5.1 shows the details of the failing tests. Three of the programs that failed involved testing the behaviour of arrays. This means that the translation of the array functionality is either incorrect or could contain a bug. This also means that the 687 tests from the test suite do not adequately test the translation of Whiley arrays into TypeScript arrays.

Name	Description	Result
001_average	Average over integer array.	Pass
002_fib	Recursive Fibonacci generator.	Pass
003_gcd	Classical GCD algorithm.	Pass
004_matrix	Straightforward matrix multiplication.	Fail: Invalid Output.
006_queens	Classical N-Queens problem.	Pass
007_regex	Regular expression matching.	Pass
008_scc	Tarjan’s algorithm for finding strongly connected components	Pass
009_lz77	LZ77 compression / decompression.	Pass
010_sort	Merge Sort.	Pass
011_codejam	Solution for Google CodeJam problem.	Pass
012_cyclic	Cyclic buffer	Pass
013_btree	Binary search tree with insertion / lookup.	Fail: Whiley compiler error.
014_lights	Traffic lights sequence generator.	Pass
015_cashtill	Simple change determination algorithm.	Fail: Invalid Output.
016_date	Gregorian dates.	Pass
017_math	Simple math algorithms	Pass
022_cars	Controlling cars on bridge problem.	Pass
023_microwave	Classical microwave state machine.	Pass
024_bits	Algorithms for bit arrays.	Pass
025_tries	String trie with lookup / insertion.	Pass
026_reverse	Reversing an array.	Pass
027_c_string	Model of C strings.	Pass
028_flag	Dutch national flag Problem	Pass
029_bipmatch	Perfect matching for bipartite graphs.	Fail: Java Exception in transpiler.
030_fraction	Big rationals.	Pass
032_arrlist	ArrayList implementation.	Fail: Invalid Output.

Table 5.1: All benchmarks that were translated into TypeScript. The name, description and result are included. The WyBench test suite can be found at https://github.com/Whiley/WyBench

5.3 Case Studies

Once the test suite had a sufficient pass rate, more complex programs were tested. These programs are web-based applications written in Whiley using two Whiley dependencies, Web.wy and Dom.wy. The Web.wy dependency is responsible for the state of the DOM which also includes updating and manipulating the DOM. Dom.wy holds Whiley bindings for the W3C Document Object Model. The transpiler should be able to transpile the Whiley application into TypeScript. This includes transpiling the dependencies the application requires. Each Whiley file that the application requires is passed through the Whiley2TypeScript transpiler. The main Whiley file and each dependency will have its corresponding TypeScript file. This TypeScript code should then be able to be compiled into JavaScript to be executed in the browser. The correctness of the translation is tested by checking the behaviour of the application is as intended. This is achieved through a series of inputs specific to the application being tested. The following list explains each of the programs that were tested.

WebCalc https://github.com/DavePearce/WebCalc.wy
: The Whiley WebCalc web application is a very simple calculator that can compute simple mathematics operations. Testing this application is very easy. Once the cualte button is clicked, the number output to the screen should be as expected and not errors should be displayed in the console.

Conway's Game of Life https://github.com/DavePearce/Conway.wy
: This program is a web application that implements Conway's Game of Life. This program is more difficult to test than the WebCalc program as the outcome is often unpredictable. However, this program can be tested against another implementation of Conway's Game of Life by comparing the outcome of a given input. Also, there are certain predictable starting patterns that can be used. One of the most common patterns is called the "glider". This pattern is set-up in a way were the state changes of the grid positions moves the pattern across the board in a single direction. The program is correct if this pattern is able to achieve this movement and not errors are displayed in the console.

Minesweeper https://github.com/DavePearce/Minesweeper.wy
: The Minesweeper game is a step up in complexity from Conway's Game of Life as it involves more game states. Furthermore, the games are randomly generated so a given input from two different Minesweeper implementations will not produce the same output. However, given inputs can be verified for correctness very easily as the output is produced with very view operations when compared to Conway's Game of Life.

5.3.1 Results

The three Whiley web applications were successfully translated into TypeScript then compiled into JavaScript and executed in the browser with the intended functionality. Figure 5.3 and 5.4 demonstrate the Conway's Game of Life and Minesweeper applications running in a browser. Although each case study was successful, the translation of these application was not straight forward. It required reasonable effort to generate executable JavaScript with the intended functionality.

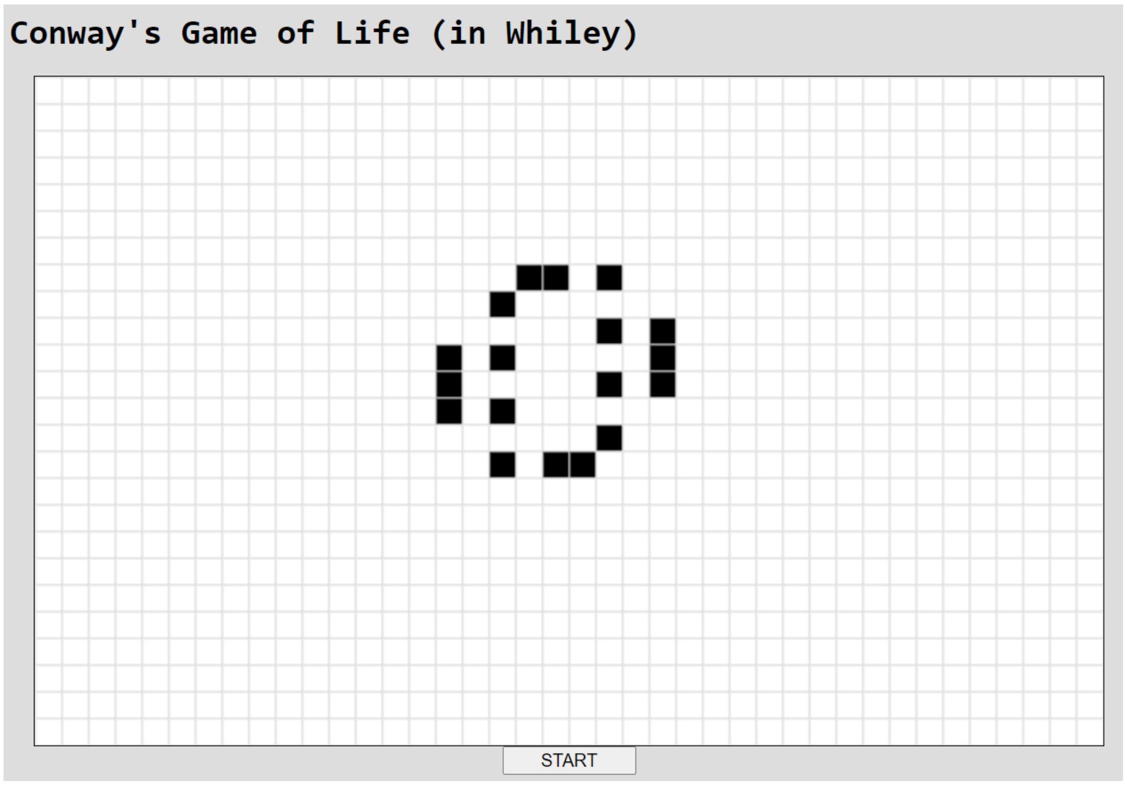


Figure 5.3: Conway's Game of Life web application.

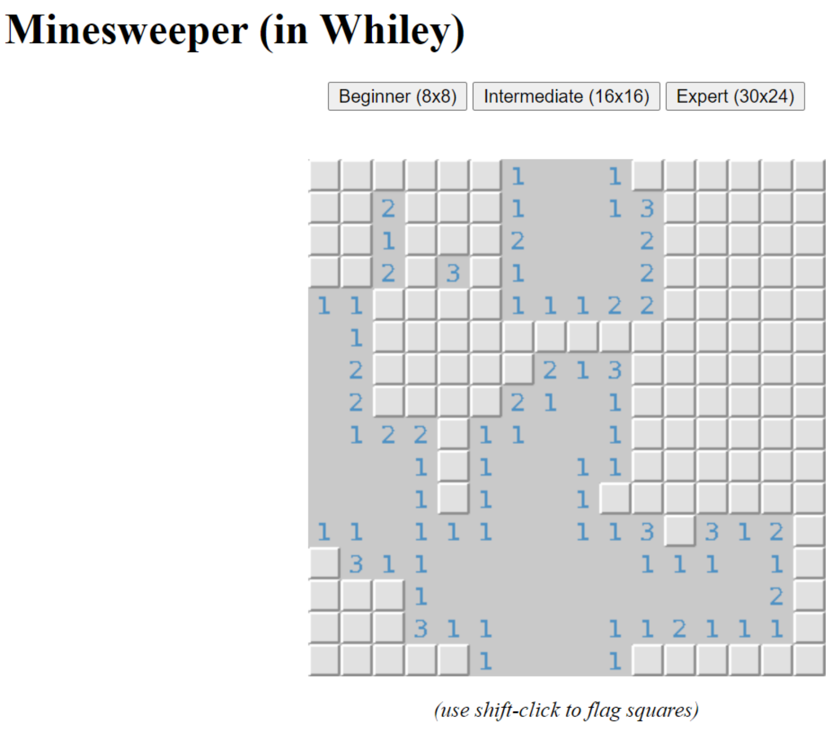


Figure 5.4: Minesweeper web application.

Chapter 6

Conclusions and Future Work

6.1 Conclusion

In this project, a Whiley compiler plugin called Whiley2TypeScript was designed and implemented with the purpose of translating Whiley into TypeScript. Generated TypeScript code can then be compiled down into JavaScript to be executed in a JavaScript environment. It was decided that the existing Whiley2JavaScript plugin was a natural starting point as it has a favourable architecture and can be extended to support the translation of Whiley to TypeScript. The Whiley2TypeScript plugin successfully translates Whiley into TypeScript with many of the Whiley features supported in the Whiley2JavaScript transpiler are also supported in the Whiley2TypeScript transpiler. This results in all but one passing test case from the Whiley Compiler test suite. Furthermore, a 19 out of 24 pass rate for benchmarks from the WyBench benchmark suite and the successful translation of Whiley web applications including WebCalc, Conway's Game of Life and Minesweeper.

6.2 Future Work

The completion of this project opens up avenues for future work.

- **One failing test from the Whiley Compiler test suite.** One of the 687 tests from the test suite is currently failing. This test is failing due to a condition that aims to type check an argument of a function at runtime. This is necessary in JavaScript to retain the behaviour of the Whiley code. However, because TypeScript function parameter types are checked at compile time, it is possible to create unreachable code. This failing test generates a condition that is always true so generates unreachable code which returns an error from the TypeScript compiler.

- **Failing WyBench tests.** A small portion of tests in the WyBench benchmark suite that involve testing array functionality fail. This is interesting as the array tests from the main test suite all pass. This uncovers two problems; Firstly, the main test suite does not sufficiently test array translation. Secondly, there is a bug in the array translation.

- **Compiling TypeScript to JavaScript correctly.** As browsers don't yet have support for modules, the generated TypeScript code must be compiled into JavaScript and bundled in a way where the functionality of the program is maintained and the browser can execute this program successfully. Progress was made with bundling the TypeScript code into a single JavaScript file. However, certain TypeScript compiler errors prevented the JavaScript output from executing without error. This is because the bundler requires correct module behaviour to successfully bundle all TypeScript files. The module behaviour is mostly correct, the problem arises when JavaScript code is appended to dependency files. This JavaScript code has properties that are referenced from other TypeScript files but this JavaScript code does not export them. Another problem is the import names and the references to the import names do not match.

- **Test larger programs.** The evaluation of the Whiley2TypeScript transpiler involved testing the translation of web applications. All web applications tested were successfully translated and executed in the browser. Further testing of larger applications can be carried out.

- **Generate bindings from existing TypeScript definition files for Whiley.** Whiley language bindings for JavaScript and the W3C Document Object Model have been manually created in Whiley. Automatically generating these language bindings would give greater accuracy, easily support further language bindings for other JavaScript libraries and handle any changes that may occur to JavaScript libraries that already have language bindings.

Bibliography

[1] BARNETT, M., FÄHNDRICH, M., LEINO, K. R. M., MÜLLER, P., SCHULTE, W., AND VENTER, H. Specification and verification: The spec experience. *Commun. ACM 54*, 6 (June 2011), 81–91.

[2] BICARREGUI, J. C., HOARE, C. A. R., AND WOODCOCK, J. C. P. The verified software repository: a step towards the verifying compiler. *Formal Aspects of Computing 18*, 2 (Jun 2006), 143–151.

[3] CHONG, N., COOK, B., KALLAS, K., KHAZEM, K., MONTEIRO, F. R., SCHWARTZ-NARBONNE, D., TASIRAN, S., TAUTSCHNIG, M., AND TUTTLE, M. R. Code-level model checking in the software development workflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (New York, NY, USA, 2020), ICSE-SEIP '20, Association for Computing Machinery, p. 11–20.

[4] CUOQ, P., KIRCHNER, F., KOSMATOV, N., PREVOSTO, V., SIGNOLES, J., AND YAKOBOWSKI, B. Frama-c. In *Software Engineering and Formal Methods* (Berlin, Heidelberg, 2012), G. Eleftherakis, M. Hinchey, and M. Holcombe, Eds., Springer Berlin Heidelberg, pp. 233–247.

[5] FELDTHAUS, A., AND MØLLER, A. Checking correctness of typescript interfaces for javascript libraries. *SIGPLAN Not. 49*, 10 (Oct. 2014), 1–16.

[6] FILLIÂTRE, J.-C., AND PASKEVICH, A. Why3 — where programs meet provers. In *Programming Languages and Systems* (Berlin, Heidelberg, 2013), M. Felleisen and P. Gardner, Eds., Springer Berlin Heidelberg, pp. 125–128.

[7] FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. Extended static checking for java. *SIGPLAN Not. 37*, 5 (May 2002), 234–245.

[8] HOARE, T. The verifying compiler: A grand challenge for computing research. In *Compiler Construction* (Berlin, Heidelberg, 2003), G. Hedin, Ed., Springer Berlin Heidelberg, pp. 262–272.

[9] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning* (Berlin, Heidelberg, 2010), E. M. Clarke and A. Voronkov, Eds., Springer Berlin Heidelberg, pp. 348–370.

[10] MICROSOFT. typescriptlang (home page), 2021.

[11] MICROSOFT. typescriptlang (migration stories), 2021.

[12] NPM, I. Npm (typescript package), 2021.

[13] PEARCE, D. J. Whiley.org.

[14] PEARCE, D. J. Getting started with whiley. Tech. rep., School of Engineering and Computer Science, Victoria University of Wellington, 2018.

[15] PEARCE, D. J., AND GROVES, L. Designing a verifying compiler: Lessons learned from developing whiley. *Science of Computer Programming 113* (2015), 191–220. Formal Techniques for Safety-Critical Systems.

[16] RASTOGI, A., SWAMY, N., FOURNET, C., BIERMAN, G., AND VEKRIS, P. Safe efficient gradual typing for typescript. *SIGPLAN Not. 50*, 1 (Jan. 2015), 167–180.

[17] SLATER, C. Developing a whiley-to-javascript-translator. Tech. rep., School of Engineering and Computer Science, Victoria University of Wellington, 2015.

[18] VEKRIS, P., COSMAN, B., AND JHALA, R. Refinement types for typescript. *SIGPLAN Not. 51*, 6 (June 2016), 310–325.