Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Trusted Infrastructure for JavaScript (ES5) Analysis

Author:
Si Wei Tan

Supervisor:
Prof. Philippa Gardner

June 17, 2019

Abstract

JavaScript is one of the most widely used languages in the world today. Its dynamic nature provides ample flexibility for developers but is often criticised for its complex semantics and strange edge-case behaviours. During the past decade, analysis tools have been deployed to assist developers in verifying JavaScript code to varying success. One such tool is JaVerT, an academic tool developed at Imperial College, which offers formal verification, whole program symbolic testing and bi-abductive compositional testing of JavaScript strict mode programs. However, while strict mode is a recommended-to-use subset of the language with more restrictive and less error-prone semantics, it is still an "opt-in" feature and need not be used by developers in their programs, especially if these programs contain legacy code that depend on non-strict features.

In this project, we have extended JaVerT to support non-strict features of JavaScript (ES5, the fifth edition of the ECMAScript standard. We have implemented the full set of ES5 language features in JS-2-JSIL, the JavaScript compiler of JaVerT. We have validated the new compiler against the official ECMAScript Test262 test suite, achieving 100% coverage for non-strict only test cases and 99.89% coverage across all test cases that are applicable for the compiler. In doing so, we have uncovered bugs in the previous version of JS-2-JSIL that were not spotted due to the incompleteness of the Test262 test suite, as well as bugs in and inconsistent behaviours across modern browsers when it comes to non-strict features of JavaScript. Using the whole-program symbolic testing aspect of JaVerT, we have also created a series of symbolic tests that aim to uncover the intricate details of non-strict features in the language not necessarily tested by Test262.

Acknowledgements

I would like to thank my supervisor, Prof. Philippa Gardner, for her continued support in this project despite her busy schedule. Her encouragement and confidence in the direction of the project is an invaluable source of motivation.

I would also like to thank Petar Maksimović and José Fragoso Santos for their support throughout the project. Petar's 24/7 assistance on my numerous questions and attention to detail was especially helpful in helping me progress quickly in the project. José's detailed and step-by-step explanations of the existing toolchain was also critical in helping me jump-start the initial phases of the project.

Finally, I would like to thank my family, friends and past internship mentors who have supported me throughout my entire degree. Their encouragement was a source of light that helped me pushed forward in my darkest times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

JavaScript is one of the most popular programming languages used for client-side scripting in web applications. It is used by 95.1% of websites [26] and is the most popular language on GitHub [10, 11]. The dynamic nature of JavaScript and high level of control that it presents to the developer enables the rapid development of interactive and responsive web applications.

However, this dynamic nature, coupled with the evolving standards by ECMAScript committee [5] makes JavaScript a difficult language to reason about and ensure functional correctness of. Even within the same standard, different behaviour could be reproduced depending on the *mode* of execution. In particular, JavaScript has two modes: *non-strict*, which is the default, unrestricted mode, featuring the notorious `with` statement and error silencing; and *strict*, which purposefully exhibits better behavioural properties, such as lexicographic scoping and better error reporting.

We illustrate one of the differences between ES5 Strict and ES5 using a simple JavaScript code snippet below (CS 1.1) executed in ECMAScript 5th Edition (ES5) Standard [4]:

```
1  function f() {
2    a = 1;   // Reference error in strict mode
3    var b = 2;
4  }
5  f();
6  a; // Returns 1 in non-strict mode
7  b; // Reference error (b is not defined outside of function f)
```

Code Snippet 1.1: Variable declaration in ES5

In ES5 Strict, references to undeclared variables result in a run-time `ReferenceError`. In non-strict mode, such references result in the creation of global variables. Therefore, in strict mode, the variable `a` in line 2 is undeclared and hence causes a reference error when `f()` is first executed. However, in ES5, the same variable is automatically lifted into a global variable and initialised to 1 in line 2. This difference in behaviour may be spotted by someone well-versed in the ES5 standard, but may also cause hidden bugs with far-reaching consequences in complicated JavaScript programs (imagine if `a` was maliciously redefined by the function `f`).

Given the complexity of JavaScript programs and standards, there is evident great benefit in providing formal methods and tools that offer assurances of code correctness. However, unlike for static languages, such as C and Java, tools for analysing JavaScript code are few and far-between. One such tool is JaVerT [24, 8], developed at Imperial College London. JaVerT is a state-of-the-art analysis tool for JavaScript, which supports symbolic testing, full verification, and automatic compositional testing of ES5 Strict programs.

This project extends the infrastructure of JaVerT to support the full ES5 standard (both strict and non-strict). This is an essential step towards analysing real-world JavaScript code, because: (1) ES5 is supported by all modern browsers,[1] (2) strict mode usage is low amongst JavaScript developers [13], and (3) there are valid use cases of non-strict mode constructs, such as the `with` statement, to emulate block-level scoping in ES5 (cf. §4.2.3).

[1] http://kangax.github.io/compat-table/es5/

In particular, we have extended the compiler of JaVerT to fully support all ES5 strict and non-strict language features. These include dynamic name resolution, the `with` statement, direct and indirect `eval`, together with many other smaller internal methods of ES5. We have benchmarked our implementation against ECMAScript's official test suite, Test262, achieving 100% coverage of all applicable tests for strict- and non-strict-only features, and 99.89% across all applicable tests. Additionally, we have analysed implementations of the ES5 specification in modern programs, such as Microsoft Edge, Chrome, Mozilla, and NodeJS, discovering a bug in Microsoft Edge and an important implementation detail of the NodeJS run-time that could result in unexpected behaviours when running ES5 code. Finally, we have enabled the whole-program symbolic testing aspect of JaVerT for ES5, and demonstrated how it can be used to create symbolic Test-262-like tests, revealing often overlooked or counter-intuitive behaviours of the specification that could cause confusion among developers.

The report is organised as follows. We discuss the differences in behaviour between ES5 Strict and ES5 and present an overview of related work in Chapter 2. In Chapter 3, we present the extended ES5 JS-2-JSIL compiler in detail. In Chapter 4, we demonstrate how JaVerT can be used for whole-program symbolic testing of ES5 programs. We evaluate the project in Chapter 5. We conclude in Chapter 6, outlining possible future extensions that would further improve JaVerT in the analysis and verification of JavaScript programs.

Chapter 2

Background

To aid us in understanding the domain of the problem presented in Chapter 1, we will present the background of three key topics. We start by introducing the JavaScript Language, focusing on key aspects of the language that are relevant, and the differences between ES5 and ES5 Strict mode. Next, we discuss the problem of ensuring program correctness for JavaScript programs, providing a brief overview of existing works on JavaScript verification and validation. Finally, we cover an overview of JavaScript Verification Toolchain 2.0 (JaVerT 2.0), a JavaScript verification and testing framework whose functionality we will extend from ES5 Strict to ES5.

2.1 The JavaScript language

JavaScript is a object-oriented dynamic programming language. It is dynamic in that objects can be extended and variable types can be modified at runtime. It is standardised by the ECMAScript Committee with the latest standard released on June 2018 (9th edition) [5]. The standard defines a strict mode for the language, which provides improved error reporting and restricts some language features available to the developer. These restrictions include removing certain language features that are considered "error-prone" and developers may opt to use strict mode in the interest of security or preference. As mentioned in Chapter 1, this project will work with the 5th edition of the ECMAScript standard - specifically in non-strict mode. For clarity, we will refer to the publication released for the ES5 standard on June 2011 [4].

We will first present the key features of JavaScript, then highlight the difference between strict and non-strict mode in ES5.

2.1.1 Key JavaScript features

2.1.1.1 JavaScript objects

Objects are collections of properties. These properties can include values, functions or other objects. Properties can be classified into two types: *named* or *internal*.

Named properties are further classified into *named data properties* and *named accessor properties*. They are associated with *property descriptors*, which are lists of *attributes*. Each *attribute* describes the way in which a property can be accessed and/or modified. The classification of *named* properties into *named data* and *named accessor* properties is based on their associated list of attributes. Named data properties contain four attributes: Value, Writable, Enumerable and Configurable (denoted by `[V]`, `[W]`, `[E]`, and `[C]`, respectively). Named accessor properties also contain four attributes, but differ from named properties in the first two attributes: Get, Set, Enumerable and Configurable (denoted by `[G]`, `[S]`, `[E]`, and `[C]`, respectively). The attributes have the following semantics: `[V]` holds the property value; `[W]` determines if the value `[V]` can be modified; `[E]` determines if the property is included for the `for-in` enumeration; `[G]` and `[S]` operate similarly to traditional *getters* and *setters* of object-oriented languages such as Java; and `[C]` determines if modification of the other attributes (except for `[V]`) is allowed.

Internal properties are related to the inner workings of JavaScript and are hidden from the user. They exists for specification purposes and are used for critical JavaScript mechanisms such as prototype inheritance.

2.1.1.2 Lexical environments and environment records

A *Lexical Environment* is a specification type that defines the association of ECMAScript identifiers to variables and functions. It is based on the lexical nesting of the ECMAScript code and can be seen informally as a *pair* that contains an *Environment Record* and a (possibly null) reference to an outer Lexical Environment.

An *Environment Record* (ER) records identifier bindings created within the scope of its associated lexical environment. Informally, it can be seen as a lookup table of key-value pairs that maps ECMAScript Identifiers to variable and function references. There are two types of ERs: *Declarative ERs* and *Object ERs*. Intuitively, a declarative ER is associated with language syntactic elements such as function declarations, variable declarations and catch clauses. The associated identifiers of these elements are bound directly to ECMAScript values. Object ERs are associated with other ECMAScript elements such as programs or the `with` statement, which binds the identifiers to the *properties* of some object.

2.1.1.3 Executable code and execution contexts

There are three types of executable code[1]:

1. Global code: Source text that is parsed at the top-level in the JavaScript program. It does not include source text that are part of function bodies.
2. Eval code: Source text passed to the built-in `eval` function.
3. Function code: Source text parsed within a function body. It is **not** recursive with nested functions and their associated function bodies. In addition, source text passed to the built-in `Function` constructor as a function body is also considered function code.

In ES5, the variable state management is emulated via the use of *Execution Contexts*. An execution context consists of three components:

1. LexicalEnvironment[2]: The lexical environment used for resolving identifier references.
2. VariableEnvironment: Lexical environment whose environment record holds the bindings created from variable and function hoisting.
3. ThisBinding: Holds the value associated with the `this` keyword.

An execution context is created whenever control is transferred to an executable code. These execution contexts form a logical stack, with the top of the stack being the current (active) execution context.

2.1.1.4 Scope resolution

Scope resolution is performed by inspecting the identifier bindings present in lexical environments. By construction, the lexical environments of the execution contexts naturally form a linked-list based on the lexical nesting of ECMAScript code. Scope resolution begins by inspecting the LexicalEnvironment of the current execution context, then proceeds "up" the linked-list until the global lexical environment. We represent this global lexical environment as a pair comprising: the global object ER, l_g and a **`null`** outer lexical environment reference. To illustrate the mechanics of scope resolution, we give a simple example of a ES5 code below (Code Snippet 2.1). The code defines a function `f` that takes 2 arguments, `a` and `b`, and returns the second argument, `b`. The LexicalEnvironment formed during the execution of line 5 is illustrated in Figure 2.1.

[1] ES5 Section 10.1
[2] Not to be confused with “Lexical Environment" (note the space) - the type explained in §2.1.1.2

```
1  var f = function (a, b) {
2      return b;
3  }
4  var a = 1;
5  f(3, a); // returns 1
```

Code Snippet 2.1: Scope resolution example

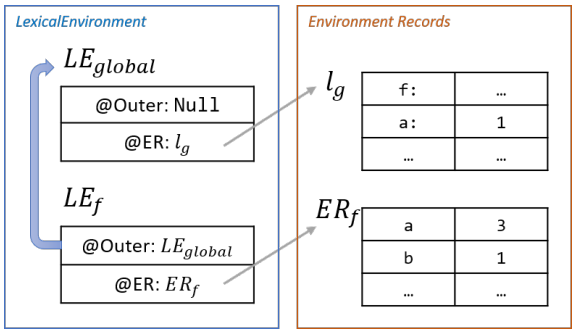


Figure 2.1: Scope resolution example

ER_f is the (declarative) environment record for the function `f`. It contains two properties, corresponding to the two parameters: `a` and `b`. l_g is the (object) environment record for the global environment containing two properties: reference to the function `f` and the variable a. Both ERs contain internal properties used by the inner workings of JavaScript which we are not interested in at the moment. The corresponding lexical environments are LE_f and LE_{global} respectively, and they form the linked-list used for scope resolution. During the execution of line 5, the variable `a` is copied into the parameter `b` of function `f`. Hence, the return value of `f(3, a)` is `1`.

While this example is trivial, we will build on this notion of scope resolution in the next section when discussing about `with` statements.

2.1.2 Strict vs non-strict mode in ES5

In this section, we will highlight the main differences between ES5 Strict and ES5 features.

2.1.2.1 The `with` statement

The most important difference between ES5 Strict and ES5 is the inclusion of the `with` statement in non-strict mode. The syntax of the `with` statement is as follows:

WithStatement :
`with` (*Expression*) *Statement*

The semantics of the `with` statement is described in three steps. First, an object environment record is added to the lexical environment of the current execution context. This object environment record is computed from the given *Expression*. Next, the *Statement* is executed with the modified lexical environment. Finally, the lexical environment is restored to its original state.

We will illustrate the process with the program in Code Snippet 2.2.

Similar to the example in Code Snippet 2.1, we have a function `f`, with two parameters, `a` and `b`. However, instead of simply returning the second parameter `b`, it now returns a **dynamic** value based on the LexicalEnvironment of the current execution context.

With the introduction of the `with` statement in lines 2 to 4, an object ER is added to the LexicalEnvironment by computing the object from the expression `a` (the object computed here is just `a` from the global context). The execution in line 3 will now run with the modified LexicalEnvironment. Variable resolution of `b` in line 3 will first look up the property `b` in `a`, then

up a's prototype chain. If `b` is not found in `a` (and in the properties inherited from its prototype chain), then the lookup will move to the outer lexical environment (ER_f) and attempt to find `b`. The rest of the scope resolution proceeds as mentioned in §2.1.1.4. As such, we obtain `2` from the first execution of `f` on line 7 and `3` from the second execution in line 9. The scope resolution in line 9 is illustrated in Figure 2.2.

```
1  var f = function (a, b) {
2      with (a) {
3          return b;
4      }
5  }
6  var a = 1;
7  f(a, 2); // returns 2
8  var a = {b: 3};
9  f(a, 2); // returns 3
```

Code Snippet 2.2: `with` statement example

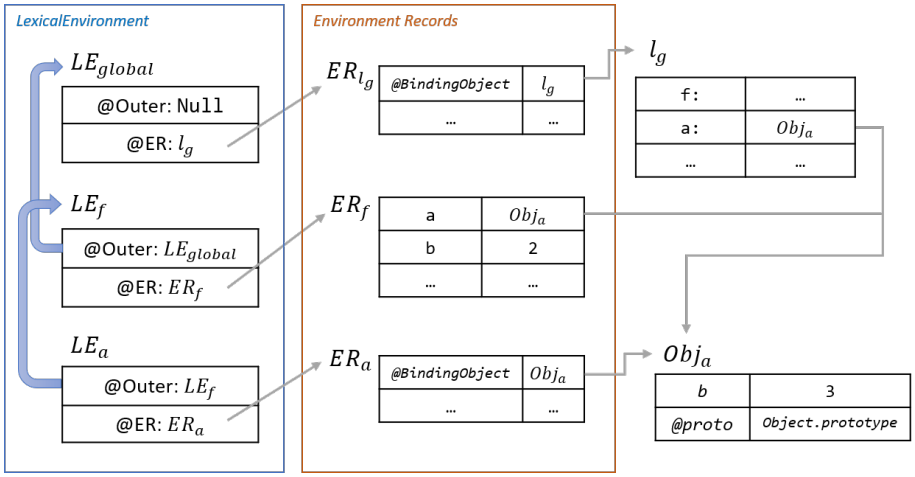


Figure 2.2: Scope resolution in `with` statement

2.1.2.2 `arguments` object

The `arguments` object is an array-like object present by default in function objects. It contains the values of the parameters passed to that function. The use of the arguments object is to typically reference function parameters using array indices, for example:

```
1  var f = function (a, b) {
2     arguments[0]; // a = 1
3     arguments[1]; // b = 2
4     return arguments[0] + arguments[1];
5  }
6  f(1, 2); // returns 3
```

Code Snippet 2.3: `arguments` object example

However, there is a difference in the behaviour of the `arguments` object between ES5 Strict and non-strict mode. In strict mode, the `arguments` object is a *copy* of the function parameters. Modification to the elements in the `arguments` object does **not** modify the function parameters referred to by their named identifier. Additionally, the `arguments` object cannot be declared as an identifier for a function parameter.

For non-strict mode, the `arguments` object (that is created automatically during the execution of the function code) contains objects that share the same bindings as the named parameters of the function. This means that changing the contents of the `arguments` object will change their corresponding named parameters.

A summary of the main differences in behaviour is shown below (Code Snippet 2.4).

```
// compile error: "arguments" used as parameter identifier
var f = function (arguments) {
   "use strict";
}
// compile error: "arguments" used as variable declaration in function body
var g = function (a) {
   "use strict";
   var arguments;
}
// Non-strict mode, arguments[0] shares same binding as a
var h = function (a) {
   arguments[0] = 2;
   return a;
}
h(1); // returns 2
```

Code Snippet 2.4: `arguments` object in strict and non-strict mode

2.1.2.3 Indirect `eval`

The `eval` function is perhaps one of the most notorious features of JavaScript. In a study done in 2011, *The Eval That Men Do* [22], it surveyed the top 10,000 most popular websites and observed that over 50% of these websites use `eval` to varying degrees. Improper use of `eval` could result in bugs and vulnerabilities that are not immediately visible to the developer. It is therefore important to understand the inner mechanics of `eval`. For clarity, we refer a function call to the built-in function `eval` as an "`eval` call" and the string passed to the function as "`eval` code".

There are two types of `eval` calls in general: direct and indirect. A direct `eval` call is a JavaScript statement with a function call that resolves to the built-in `eval` function directly. An indirect `eval` call is any other statements that involve immediate steps before calling the built-in `eval` function.

An important distinction between direct and indirect calls is the *execution context* with which the eval code is executed. A direct `eval` call will be executed in the **current** execution context of the caller while an indirect `eval` call will execute in the **global** execution context. Code Snippet 2.5 shows the difference between direct and indirect `eval` calls, and their corresponding execution context:

```
// Direct calls
eval("1+1;");
(eval)("1+1;");

// Indirect calls
(0, eval)("var str = 'indirect';");
var e = eval;
e("var str = 'this is also indirect';");

// eval code in caller's execution context
var a = 1;
(function() {
    var a = 10;
    // Direct Call, executed in local context
    return eval("a + 1;");
})(); // returns 11

// eval code in global execution context
var a = 1;
(function() {
    var geval = eval
    var a = 10;
    // Indirect Call, executed in global context
    return geval("a + 1;");
})(); // returns 2
```

Code Snippet 2.5: Direct vs indirect `eval` calls

Aside from the execution context of the `eval` code, the type of `eval` call also affects the mode in which the `eval` code in executed. We will term the mode of execution of the caller as *calling mode*.

There are two cases where the `eval` code is executed in strict mode:

1. The `eval` code contains the strict-mode directive, `"use␣strict";`. For example:

```
1 (function() {
2     // Both in strict and non-strict mode
3     var a = 10;
4     // eval code declares "use strict" directive -> execute in strict mode
5     return eval("\"use strict\"; a + 1; b = 1;"); // Reference error for b
6 })();
```

Code Snippet 2.6: Mode of execution in `eval` code

2. In the absence of the strict mode directive, a **direct** `eval` call is made while executing in strict mode.

All other combinations of call type and calling mode will result in the `eval` code executing in non-strict mode. We summarise the combinations in Figure 2.3.

With strict mode directive		Calling Mode	
		Strict	Non-Strict
Call Type	Direct	Strict	
	Indirect		

Without strict mode directive		Calling Mode	
		Strict	Non-Strict
Call Type	Direct	Strict	Non-Strict
	Indirect	Non-Strict	Non-Strict

Figure 2.3: `eval` code mode of execution

2.1.2.4 Other noteworthy differences

Aside from the key differences mentioned above, there are other noteworthy differences that have significant impact on the semantics of the program when compiled in non-strict mode. We will broadly cover some of those mentioned in Annex C of the ES5 Standard [4].

Identifier names. Strict mode restricts the use of certain reserved words for identifiers such as `"implements"`, `"interface"`, `"let"`, etc. These are classified as *FutureReservedWord* and would result in a compile error when used in ES5 Strict. In non-strict mode, the use of these words would behave like any other identifier.

`eval` and `arguments` identifier. These identifiers cannot occur as a *LeftHandSideExpression* in strict mode. This meant that in non-strict mode, it is possible to change the built-in function `eval` and the `arguments` object. It is essential that this difference is made apparent when verifying JavaScript code - distinguish between built-in functions/objects and user-defined functions/objects with the same name.

Assignment to undeclared identifier. In strict mode, an assignment to an undeclared identifier (not found in the LexicalEnvironment of the execution contexts) will result in a ReferenceError. In non-strict mode, the assignment will result in the initiation of the identifier as a global variable. This is best exemplified in the example given in Chapter 1 (Code Snippet 1.1).

2.2 Program correctness for JavaScript

Ensuring the correctness of a program in general is hard. Bugs and vulnerabilities in programs could arise from various reasons such as unchecked user input, vulnerable library dependencies or a mistake on the developers code. To check for correctness in code, developers may employ extensive use of test cases and code reviews. However, these processes are limited in practice. Test cases only prove the correctness of program behaviour in the domain of the test suite, which may not capture the full range of inputs and states the program could execute in. Code reviews are also dependent

on the reviewers' insight on the program and is highly prone to human errors. Program correctness therefore requires more formal verification and validation mechanisms.

In this section, we will discuss the theory and tools related to JavaScript program verification and validation, across three broad categories: formal verification, symbolic execution and compositional testing.

2.2.1 Formal verification based on separation logic (SL)

2.2.1.1 Hoare Logic

Hoare Logic was introduced in 1969 by Tony Hoare [15]. It is a formal verification system based on a set of rules to reason about the correctness of programs. At the center of the reasoning mechanism is the Hoare Triple, $\{P\}C\{Q\}$[3]. It describes the connection between the precondition of the program $\{P\}$, the program C and the result of the execution as the post-condition $\{Q\}$. Informally, it means "if the precondition $\{P\}$ holds before the execution of C and the execution of C successfully terminates, then the post-condition $\{Q\}$ will hold after its execution". $\{P\}$ and $\{Q\}$ are also known as *assertions*. Hoare also developed a set of inference rules that specify the axiomatic behaviour of programs such as the Rule of Consequence [4]:

$$\frac{\vdash P \to P' \quad \vdash \{P'\}\ C\ \{Q'\} \quad \vdash Q' \to Q}{\vdash \{P\}\ C\ \{Q\}}\ \text{Consequence Rule}$$

Figure 2.4: Consequence Rule

However, these rules operate with assertions that are applied on the entire program and hence do not scale well with the size of the program C.

2.2.1.2 Separation logic

To address the scalability issue with Hoare Logic, Separation Logic (SL) [19] was propsed as an extension to Hoare Logic [15]. SL operates by the notion of *local* reasoning, where a program is segmented into local states called *heaps*. Heaps are partial functions that map addresses to values. Local states are deemed to be independent segments of the program and the entire program can be reasoned by composing these local states together.

The key mechanism to this approach is the introduction of a *separating conjuction* $P * Q$, which takes two assertions P and Q, and asserts the P and Q hold for **disjoint** portions of the addressable storage [19]. This allowed the introduction of the *frame rule*, an important inference rule core to the mechanics of local reasoning:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ \text{Frame Rule}$$

(where no variable occurring free in R is modified by C.)

Figure 2.5: Frame Rule

The use of the frame rule enables the extension of a local specification by involving on parts of the heap that are used by the program C,by adding predicates about parts of the heap that are not modified by C[19].

[3]The notation in the paper [15] uses $P\{Q\}R$ but is semantically the same as the notation we used in this report.
[4]Notation difference from [15]. $\to$ means logically implies (Hoare uses $\supset$).

2.2.1.3 Related works

Separation logic has been shown to be useful in formal verification of programs. In particular, when it comes to JavaScript, there are several significant pieces of work and we briefly cover each of them in this section.

Towards a Program Logic for JavaScript [9] developed separation logic for verifying for a fragment of the ECMAScript 3 (ES3) Standard to reason about the variable store emulated in the JavaScript heap. This enabled reasoning about a substantial subset of ES3 features such as prototype inheritance, the `with` statement, simple functions and simple `eval`. Of interest is the introduction of a new *sepish* connective, $P \bowtie Q$ that allows *partial* separation between heaps. Informally, the sepish connective splits the heap into 3 disjoint heaps, $h_1, h_2, h3$ where P is satisfied in h_1, Q is satisfied in h_2, and h_3 is the "shared" part that satisfies both P and Q. The use of the $\bowtie$ allowed them to reason concisely about prototype inheritance, where objects may share the same prototype. However, as mentioned in [24], the extension of the logic to the full language is intractable. JavaScript operations as standardised in ES5 involves the use of numerous internal functions such as `GetValue`, `Type`, `PutValue`, `GetReferenceName` [4] and complex control flow statements such as `switch` or `try-catch-finally`. To verify programs in JavaScript would require all these complexities to be formalised into proof rules, thus making automation essentially impossible.

Building on the work of [9], *JaVerT: JavaScript Verification Toolchain* [24] was developed as a semi-automatic tool based on separation logic that verifies specifications of JavaScript code based on the ES5 standard in strict mode. JaVerT solves the scalability issue raised above by moving to an intermediate representation of JavaScript, JSIL. They introduced a logic-preserving compiler JS-2-JSIL that compiles JavaScript code into JSIL and a semi-automatic verification tool JSIL Verify that uses sound JSIL separation logic. Their compiler is well tested against the official test suite for JavaScript, Test262 [3], achieving 100% test result for all relevant test cases [5]. This work stops at ES5 Strict, hence does not cover the full ES5 standard.

A Trusted Mechanised JavaScript Specification [2] developed a formalisation of the ES5 standard in the Coq proof assistant JSCert, and a reference interpreter for JavaScript JSRef. It covers a significant portion of the ES5 standard including the syntax and semantics of JavaScript expressions, statements and programs. However, it does not specify the parsing of JavaScript programs, which is an important mechanism required for the `eval` call (§2.1.2.3).

SAFE [18] is a tool developed for formal specification and implementation of JavaScript. It provides three different levels of intermediate representation, all formally defined: Abstract Syntax Tree (AST), IR and Control Flow Graph (CFG). However, this tool is mainly used as parser/compiler for JavaScript and does not actually verify the output of the intermediate representation (as done in [24, 8]). Instead, it provides the framework that other verification components could use for verification purposes. In addition, the robustness of the compiler is not verified against the Test262 test suite. *SAFE 2.0* [21], an extension to *SAFE* added new features such as a HTML Debugger module. While the implementation of *SAFE 2.0* is claimed to be tested against Test262, it does not report the coverage of the test suite.

An interesting work used in [18, 21] is the rewriting of `with` statements as presented in [20]. It classifies `with` statements into 6 types, 5 of which could be rewritten to semantically equivalent statements without the use of `with`. The last category is deemed not rewriteable due to dynamic code generation (a combination of `with` and `eval` calls).

2.2.2 Symbolic execution and testing

Symbolic execution is a testing methodology that involves the use of *symbolic* values to validate the correctness of programs [17]. Symbolic values differ from concrete values in that they represent *classes* of inputs. Each class represents the range of input values that a variable can take, constrained by the conditions of the program executed thus far. As such, symbolic testing provides the benefit of testing the program over a range of inputs without explicitly testing all possible concrete values.

[5] ES5 Strict test cases

2.2.2.1 Related works

A Symbolic Execution Framework for JavaScript [25] developed a symbolic-execution based framework for analysing client-side JavaScript code. Their automated tool, *Kudzu*, provides automatic exploration of the execution space of client-side JavaScript code. The symbolic executor is dynamic, in that the path exploration is *random*, hence does not cover all possible execution paths.

Symbolic Execution for JavaScript [23] developed a symbolic executor *Cosette* for JavaScript programs aimed at providing a general-purpose symbolic analysis and testing tool for developers. It is an extension of the work in [24], borrowing the JS-2-JSIL compiler to compile extended JS programs to extended JSIL programs for reasoning about symbolic values. It supports *whole-program symbolic testing* and *specification-driven bug finding*. While Cosette is based on the ES5 Strict standard, it does not cover the use of `eval`.

2.2.3 Compositional testing

Symbolic execution does not scale well in practice due to the *path explosion problem* [12]. Attempting to perform symbolic execution systematically over all possible program paths in a large, complex program is computationally expensive and can be imprecise due to unbounded number of iterations.

Compositional testing alleviates the path explosion problem by performing symbolic execution *compositionally* [12]. The key idea in compositional symbolic execution is to *summarise* parts of the program into logical units called *functions*[6]. Every function f is given a function summary ϕ_f that describes the preconditions and post-conditions (also known as *constraints*) associated with f. Executing symbolic testing will in turn only inspect program paths that are deemed *feasible* by the constraints [1].

2.2.3.1 Related works

The concept of compositional symbolic execution and testing is not new. [12] used this idea for dynamic test generation and [1] built on the works of [12] to develop a *demand-driven compositional symbolic executor* for .NET applications.

However, to the best of our knowledge, *JaVerT 2.0* [8] is the first to apply this idea to JavaScript programs. We will cover JaVerT 2.0 in more detail in §2.3.

2.3 JaVerT 2.0

The JavaScript Verification Toolchain (JaVerT) 2.0 [8] is an extension to JaVerT [24], a tool built to assist JavaScript (JS) developers in the testing and verification of their programs. JaVerT 2.0 is based on ES5 Strict and supports: whole-program symbolic testing, semi-automatic verification and automatic compositional testing.

The architecture of JaVerT 2.0 is shown in Figure 2.5. It has three main components: (1) JS parser, (2) JS-2-JSIL compiler and a (3) run-time compositional symbolic execution component for JSIL. We will briefly describe the high level overview of each component below.

2.3.1 JS Parser

JaVerT 2.0 uses Esprima [6] as its JavaScript parser. It is a standard-compliant ECMAScript parser that can be used for both lexical analysis (tokenisation) and syntatic analysis (parsing) of JavaScript programs. The output from the parser is a JSON [16] object representing the Abstract Syntax Tree (AST) of the input JS Program. Esprima does distinguish between strict and non-strict mode of ECMAScript [14], but is not clear if it supports a mixture of both in the same program[7].

[6] Not to be confused with `function`s used in programming languages
[7] Inlining `"use␣strict";` in select parts of the program.

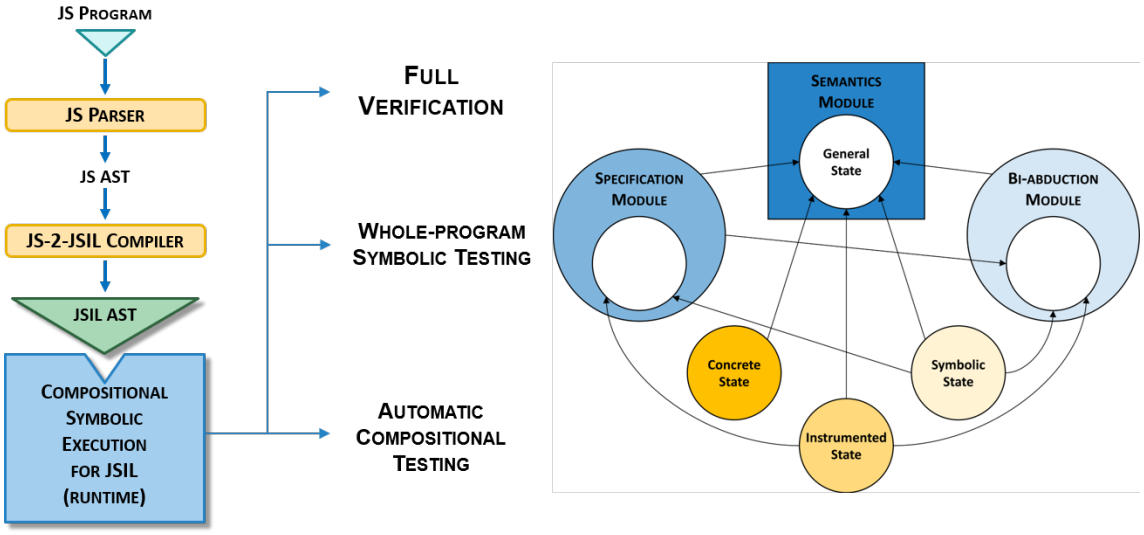


Figure 2.6: JaVerT 2.0 Architecture

2.3.2 JS-2-JSIL Compiler

JS-2-JSIL is a logic preserving compiler that compiles JavaScript code into JSIL, a proprietary intermediate language for JavaScript defined in [24]. The compiler follows line-by-line the ECMAScript English Standard (in strict mode) specified in [4]. ES5 internal functions are translated to JSIL procedural calls to their reference implementation [24] (JSIL internal functions). To demonstrate this correspondence, we provide an example [24] of the *simple assignment* operation in ES5, compiled to JSIL code (Figure 2.7, the ES5 standard on the left, JSIL code on the right).

In this example, we are compiling the statement `contents[k] = v` where `contents` is some arbitrary object with property `k` and `v` is an arbitrary value. The compilation of the 6 step process given by the ES5 standard is as follows:

1. The evaluation of `contents[k]` translates to lines 1-9 of the JSIL code where we obtain `["o"], x_2_v, x_4_n)` as the corresponding reference of property `k` of object `contents`.
2. Next, the evaluation of the variable `v` involves looking up on the scope chain[8] of the current execution context (JSIL line 10) and we obtain `["v", x_7, "v"];` as the reference to `v`.
3. The ES5 internal function `GetValue` is called on reference obtained in step 2, which corresponds to the JSIL internal function `i__getValue` on line 12.
4. This step is the result of strict mode restrictions in ES5, and is represented by the JSIL internal function `i__checkAssignmentErrors`.
5. The assignment operation is then performed by calling JS internal function `PutValue`. This corresponds to the JSIL internal function `i__putValue`.
6. Every JavaScript statement returns a value. The compilation of a JS statement by JS-2-JSIL returns the list of JSIL commands (code presented on the right of Figure 2.7) and the variable (`x_8_v`) that stores the return value.

2.3.3 Compositional symbolic execution for JSIL

The compositional symbolic execution component for JSIL is the runtime component for JaVerT 2.0. It is capable of performing three tasks: full verification, whole-program symbolic testing, and automatic compositional testing (Figure 2.5). To achieve this, the JSIL semantics is separated into two components: a *Semantics Module* and a *state instantiation*.

[8] We will formally introduce the notion of a scope chain in §3.2.1. Intuitively, a scope chain is simply the lexicographic list of environment records that emulates the list of LexicalEnvironments.

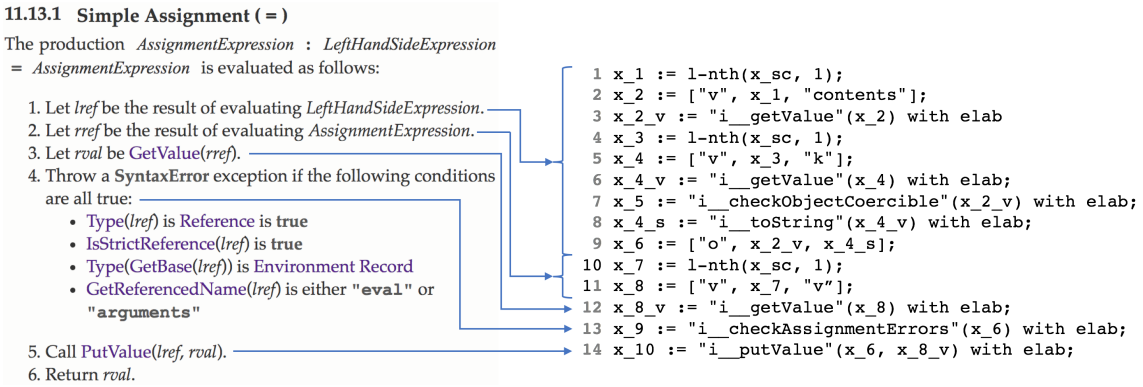


Figure 2.7: Compiling `contents[k] = v` to JSIL by closely following the ES5 Standard.

The **Semantics Module** describes the behaviour of JSIL commands in terms of a *state signature*. This state signature is general, and can be instantiated into three instances to obtain specific JSIL semantics: Concrete, Instrumented and Symbolic. Concrete semantics is used for concrete execution of JSIL programs and to test the infrastructure against Test262, the ECMAScript official test suite [3]. Instrumented semantics is an interim stage between concrete and symbolic semantics. It keeps track of objective properties that are **not** present, in order to exhibit the behaviour of the frame property. This approach resulted in better modular reasoning and simpler proofs than [9]. Symbolic semantics is the core of the compositional symbolic execution component and is obtained by lifting the instrumented state instantiation (above). It is fully formalised and proven sound [8].

Alongside the Semantics Module are the **Specification Module** and **Bi-Abduction Module**. The Specification Module links the JSIL SL assertion language to JSIL states while the Bi-Abduction Module involves the use of bi-abduction for automatic inference of missing resources in specification errors. When used independently, the Specification module provides verification for JSIL programs and the Bi-Abduction Module provides automatic local testing. Automatic compositional testing is achieved by using both in tandem.

Chapter 3

The JS-2-JSIL ES5 Compiler

The JS-2-JSIL compiler is a critical component of JaVerT. Since all of the analyses supported by JaVerT are done on the JSIL intermediate language, it is essential to have a compiler from ES5 programs to JSIL programs that preserves correct program behaviour as per the ES5 specification. In this chapter, we first briefly describe the features of JSIL that are relevant to the project (§3.1), and then present our implementation of the key ES5 features of the extended compiler:

- faithful modelling of the JavaScript (JS) variable state in JSIL (§3.2);
- dynamic name resolution (§3.3);
- compilation of the `with` statement (§3.4);
- hoisting of variable and function declarations (§3.5);
- the `arguments` object (§3.6); and
- direct and indirect `eval` (§3.7),

discussing the contributions, challenges, and implementation choices for each of them in detail.

3.1 Core JSIL constructs

JSIL is a simple object-based intermediate goto language well-suited for JS analysis, introduced in [24, 8]. As such, most of its design decisions behind are in one way or another motivated by the semantics of JS. Here, we focus on the core aspects of JSIL that are relevant for this project.

JSIL Objects and Metadata. JSIL objects contain *fields* and *metadata*. Fields are key-value pairs that represent the properties of the JSIL object. As in JS, field access in JSIL is dynamic and objects are extensible, unlike in, for example, Java, where field access is static and objects are sealed. JSIL objects additionally have metadata, which is meant to hold some extra information about the object. While the metadata can hold any JSIL value, in compiled JS programs it always contains another object with information that is not meant to be accessible by the JS program, but is required by the JS semantics, such as internal object fields (for example, `"@proto"`, which tells us the prototype of the object, or "@extensible", which tells if the object is extensible or not). The creation of a JSIL object takes an optional value as its metadata. For example, `x := new(x_meta)` creates a JSIL object `x` taking `x_meta` as its metadata. The metadata of the created object can then be accessed with: `meta := metadata(x)`. Object metadata will be extensively used in the modelling of JS state (§3.2).

Argument Collection. Within a JSIL procedure, the arguments with which the procedure was called can be accessed via the `args` command, in the style of `x := args`. As `args` is a dynamic component that retrieves parameters passed in at run-time, it allows is possible to supply and access unnamed parameters to the procedure via `args`. It is required for the correct modelling of the JS `arguments` object (§3.6).

JSIL Procedures. JSIL procedures are of the form proc $fid(\overline{x})\{\overline{c}\}$, where *proc* is a keyword for defining a JSIL procedure, fid is the name of the procedure, $\overline{x}$ is an optional list of named parameters and $\overline{c}$ is a list of JSIL commands. A procedure has two modes of completion: normal and error, allowing us to model exceptions in JSIL. Normal completion is performed via the return command while error completion uses the throw command. In both modes, a dedicated return variable `ret` stores the value that is returned.

Procedure calls. JSIL procedure calls have the format: $x :=$ e$(\overline{e})$ with j. They are dynamic, since the procedure identifier is obtained by evaluating the JSIL expression `e`. The parameters to the procedure are given by list of JSIL expressions $\overline{e}$. If the procedure completes normally, execution continues to the next command. Otherwise, we have an error completion and the execution jumps to the j-th command.

External procedure calls. External procedure calls are procedure calls with the extern qualifier: jvar := extern e$(\overline{e})$ with j. These external calls dynamically generate and execute JSIL code at run-time. There are two main use cases for external procedure calls in compiled JS code: `eval` code execution (cf. §3.7) and the `Function` constructor. Both of these cases take a single parameter, which is a string representing a JS program, that needs to be parsed and compiled at run-time.

ϕ-node commands. The ϕ-node command, can intuitively be seen as a conditional assignment based on the (JSIL) code order of the paths through which the command itself can be reached. If we assume that there are n paths through which the ϕ-node command can be reached, the command would have the form `x` $:= \phi(\overline{x})$. Its semantics would then be to assign to `x` the value x_i if and only if the i-th path was used to reach the command. This command is borrowed from the literature on Single-Static-Assignment (SSA), a style of programming that is known to simplify analysis. The JS-2-JSIL compiler produces code that is in SSA.

3.2 The JavaScript variable state in JSIL

Variable state management in JavaScript refers to the management of scope and the resolution of the `this` keyword. We have briefly explained the mechanics of variable state management in JavaScript with the use of execution contexts in §2.1.1.3. In this section, we focus on the differences in variable state management constructs between ES5 JavaScript and JSIL, and how we preserve the semantics despite these differences. In particular, since JaVerT was originally targeting ES5 Strict, which has lexicographic scoping, it had substantial simplifications in place when it comes to variable state management with respect to the standard. As these simplifications could no longer be maintained for ES5, we have fully and faithfully implemented the variable state management mechanism of ES5, line-by-line close to the standard.

Onward, we first examine how variable state management is performed in accordance with ES5 specification (§3.2.1), and then give a high-level explanation of how the JavaScript variable state is modelled in JSIL (§3.2.2). Next, we explain the inner mechanics of JSIL state management in full in §3.2.3 and, finally, conclude by explaining the implementation of environment records in §3.2.4.

We illustrate the differences in variable state management between ES5 and JSIL using the JS program (Code Snippet 3.1) and its corresponding state management constructs (Figure 3.1) below.

```
1  var obj = {
2    a: 1,
3    b: 2
4  }
5  function f(a, b) {
6      with (obj) {
7          var g = function () {
8              "use strict";
9              return a + b;
10         }
11     }
12     return g();
13 }
14 var ret = f(4, 5); // returns 3
```

Code Snippet 3.1: Nested `with` and functions

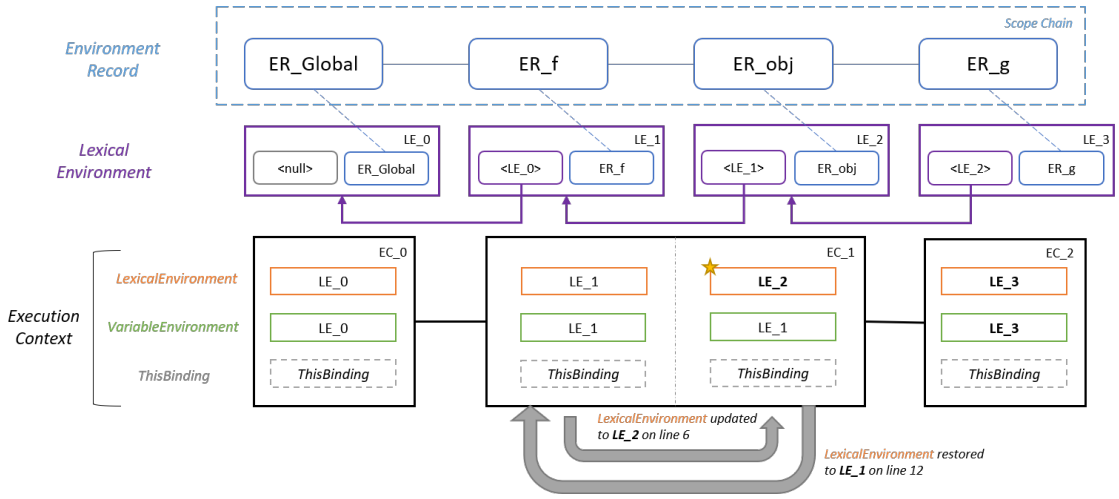


Figure 3.1: Variable State Management of Code Snippet 3.1

3.2.1 Variable state management in ES5

With reference to Code Snippet 3.1, we begin by looking at the environment records created during its execution:

1. `ER_Global`: An object environment record that is created at the start of the program, holding the global object as its binding object.

2. `ER_f`: A declarative environment record created when the execution enters function `f`.

3. `ER_obj`: An object environment record created by the use of the `with` statement in line 7, holding the JS object `obj` as its binding object.

4. `ER_g`: A declarative environment record created when the execution enters function `g`.

In ES5, execution contexts are created whenever control is passed into an ECMAScript executable code[1]. In the execution of Code Snippet 3.1, three execution contexts are created. However, states can change within the same execution context. During the execution of Code Snippet 3.1, modifications to the states are in the following order:

1. The program begins its execution in the global execution context `EC_0`. The LexicalEnvironment and VariableEnvironment initialised with the same environment record `ER_Global`, as `LE_0` and `VE_0` respectively.

2. In line 5, a new function object is created for function `f` with its internal `[[Scope]]` property set to the current **VariableEnvironment** `LE_0`[2].

3. When control is transferred to function `f` in line 14, a new lexical environment is created with the environment record `ER_f` and outer lexical environment specified by the `f` function object's `[[Scope]]` property: `LE_0`. In addition, a new execution context `EC_1` is created with LexicalEnvironment and VariableEnvironment initialised to the newly created lexical environment `LE_1`.

4. During the execution of function `f`, the use of the `with` statement in line 7 creates a new lexical environment `LE_2` with the environment record `ER_obj`. The current LexicalEnvironment is updated to `LE_2`. Note that this does **not** modify the VariableEnvironment and no new execution context is created.

[1] Recall the three types of executable code: global, function, and eval
[2] Function *declarations* create function objects using the current execution context's VariableEnvironment.

5. In line 7, a new function object is created with its internal `[[Scope]]` property set to the current **LexicalEnvironment** `LE_2`[3]. This new function object is assigned to the variable `g` and, for simplicity, we will term this function as `g`.

6. LexicalEnvironment is restored to `LE_1` before the function call to `g` in line 12.

7. The last execution context `EC_2` is created during the execution of function `g` in line 12. Entering the function code results in a creation of a new lexical environment `LE_3` with environment record `ER_g` and outer lexical environment specified by the `g` function object's `[[Scope]]` property: `LE_2`. Both the LexicalEnvironment and VariableEnvironment components of `EC_2` is initialised to `LE_3`.

8. Upon completion of the function `g` in line 12, the last execution context is popped from the execution stack and the current execution context becomes `EC_1`.

9. Finally. the completion of function `f` in line 14 restores execution stack back to the initial execution context `EC_0`.

In general, both the LexicalEnvironment and VariableEnvironment of an execution context can be seen as pointers to a specific lexical environment of a linked-list structure. However, they differ in the scenarios in which they are updated: VariableEnvironments are only updated during the creation of a new execution context; LexicalEnvironments are updated by both the creation of a new execution context and when execution enters lexical constructs of the ECMAScript code, such as the `with` or `try-catch-finally` statements. This subtle difference between LexicalEnvironment and VariableEnvironment is crucial as it affects the hoisting of declarations and scope resolution in JS. Failure to recognise this difference would lead to bugs in implementation, one of which we have observed in modern browsers (cf. §5.3.1).

3.2.2 The ES5 variable state in JSIL

Instead of using lexical environments, we model the variable state management of JavaScript in JSIL with the use of *scope chains*. A *scope chain* is a list[4] of environment records, formed by the lexical nesting structure of the ECMAScript code. We observe that the creation of lexical environments in ES5 naturally form a linked-list of lexical environments (§2.1.1.4), with the current LexicalEnvironment as the last element in the list. Since each lexical environment is a container for an environment record, we can deconstruct this linked-list into a list of environment records - the *scope chain* of the running execution context.

We now explain state management in Code Snippet 3.1 in JSIL with respect to scope chains:

1. The program begins with an initial scope chain comprising just the global environment record: `[ ER_Global ]`.

2. When control is transferred to function `f` in line 14, the declarative environment record for function `f` is appended to the scope chain: `[ ER_Global, ER_f ]`.

3. In line 7, the `with` statement introduces a new object environment record to the scope chain: `[ ER_Global, ER_f, ER_obj ]`.

4. Scope chain is restored to `[ ER_Global, ER_f ]` at the end of `with` statement on line 11.

5. A function object is created in line 7 with its `[[Scope]]` internal property set to the modified scope chain. By the same logic in the ES5 explanation above, we will term this function `g`.

6. In line 12, control enters the function `g`. A declarative record `ER_g` is created for function `g`. This declarative ER is appended to the scope chain of the function (previously established in step 5): `[ ER_Global, ER_f, ER_obj, ER_g ]`.

7. As execution exits function `g`, the scope chain is once again restored to `[ ER_Global, ER_f ]`.

[3]Function *expressions* create function objects using the current execution context's LexicalEnvironment
[4]JSIL has native support for lists.

8. Finally, scope chain is restored to its initial state `[ ER_Global ]` as execution exits `f` in line 14.

The primary advantage of using scope chains is simplicity. We only need to manage one construct instead of two pointers to achieve the same state management semantics. For all static code (this excludes the use of `eval`), we do not need to distinguish between LexicalEnvironment and VariableEnvironment, as the last element in the scope chain would be the lexical environment that we would use for compilation.

The disadvantage of this approach is the difficulty of distinguishing between a LexicalEnvironment and a VariableEnvironment, when required. In ES5, this only occurs when we have declarations within a non-strict `eval` code, nested in a `with` statement. This situation is particularly noteworthy and we will revisit this issue and how we resolve it in §3.5.

Although we do not model lexical environments directly in JaVerT, we retain the term for operating with scope chains as "Lexical Environment Operations". In addition, we will simplify our scope chain models used in this report to show only properties in environment records that are relevant to the examples.

3.2.3 The ES5 variable state management in JSIL: Inner Mechanics

As JSIL does not have a global execution context and its variable state management is local to JSIL procedures, information about the ES5 variable state must be passed via the run-time parameters to JSIL procedures. There are two ES5 variable state management constructs passed to JSIL procedures at run-time: the caller's scope chain and a caller provided `thisArg`. We will illustrate how states are passed and updated in JSIL procedures using the partial compilation of a simple JS function: `f(a, b) {}`. Prior to this, however, we need to explain how the `ThisBinding` construct works in ES5.

`ThisBinding`. In ES5, The `ThisBinding` component of execution contexts stores the return value of the `this` keyword. The initial value of the `ThisBinding` component is the global object and it gets updated whenever a new execution context is created. This occurs in two situations:

1. Entering function code. In ES5 strict, entering a function code will set `ThisBinding` to the caller provided `ThisArg`. In non-strict mode, if the caller provided `thisArg` is `undefined` or `null`, it gets updated to the global object. Otherwise, `thisArg` is coerced to type `Object` if needed before assigning to `ThisBinding`.

2. Entering `eval` code. The new execution context's `ThisBinding` is inherited from the caller's execution context.

In JSIL, the `ThisBinding` corresponds to a special JSIL variable `x__this`.

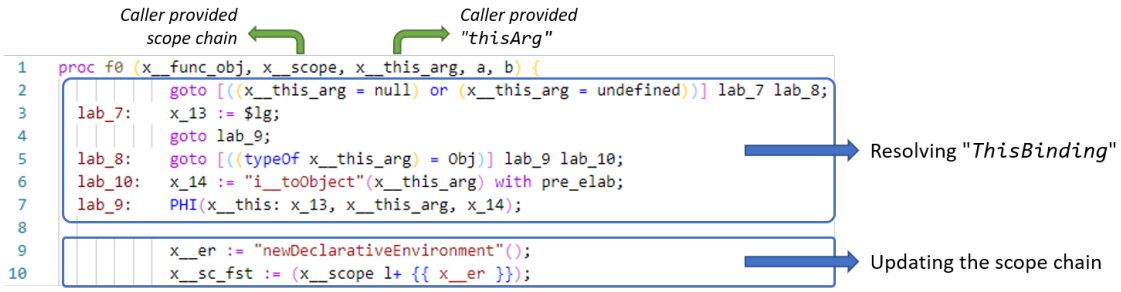


Figure 3.2: Setting up scope chain and `ThisBinding` in compiled JSIL functions

State management in JSIL procedures. We are now ready to examine the state management process in the partial compilation of the JS function `f(a, b) {}`, which is given in Figure 3.2.

1. The compiled procedure receives five parameters. The first three are parameters used internally by the compiled JSIL code, while the last two correspond to the formal parameters from the original JS function:

- `x__func_obj`, the function object that represents the current procedure. This parameter is used in compilation of the `arguments` object (cf. §3.6).
- `x__scope`, the caller's scope chain.
- `x__this_arg`, the caller-provided `thisArg`.
- `a`, the first formal parameter of `f`.
- `b`, the second formal parameter of `f`.

2. From lines 2 to 7, we perform the resolution of the `ThisBinding` construct. These commands follow the non-strict mode behaviour and we determine the final `ThisBinding` on line 7 with a ϕ-node command. If compiled in strict mode, we would directly assign the `ThisBinding` with a single JSIL command `x__this := x__this_arg` instead.

3. In lines 9 and 10, we add a new declarative environment `x__er` with the internal JSIL procedure `"newDeclarativeEnvironment"`. Then, we obtain the scope chain for the procedure `x__sc_fst` by appending `x__scope` with `x__er`. Internally within the compiler, we keep track of the latest variable that holds the scope chain via a *translation context*.

The only exception to the behaviour above is the JSIL procedure main, which is the entry point for a JSIL program. In this case, the scope chain and `ThisBinding` is set directly to the global environment and the global object, respectively.

3.2.4 Implementation of Environment Records (ERs) in JSIL

In §2.1.1.2, we gave a broad overview of the two types of ERs: declarative ERs and object ERs. In this section, we first introduce the broad type that defines generic ERs in JSIL (§3.2.4.1), then dive into how the two specific ER types are implemented in JSIL, along with their design choices (§3.2.4.2 and §3.2.4.3).

The explicit differentiation of ERs into declarative ERs and object ERs, along with the appropriate changes in compilation and the implementation of supporting functions (essentially, the entirety of §3.2.4.2 and §3.2.4.3), is new and is a substantial contribution of this project. Previously, there was no distinction between declarative ERs and object ERs, which was deemed sufficient given that the analysis target of JaVerT was ES5 Strict and that all appropriate Test262 tests were passed. This approach, however, is imprecise, but the Test262 test suite is incomplete and actually does not contain tests that would have revealed the imprecision in this conflation of ERs. We discuss this issue in detail in §5.1.4.

We implemented 11 new internal JSIL procedures to enable this distinction between declarative ERs and object ERs. The 11 procedures are grouped into three categories: (1) environment record operations (ES5 Section 10.2.1), (2) lexical environment operations (ES5 Section 10.2.3), and (3) a reference abstract operation that distinguishes ER references from property references (ES5 Section 8.7). The full list of new internal JSIL procedures can be found in Table A.2.

3.2.4.1 Generic ERs

Environment records are implemented as JSIL objects in JaVerT. Every ER in JSIL is either a declarative ER or object ER, but not both. As there are no special object types in JSIL to differentiate between JS specification constructs and actual JS objects, we have to keep type specific properties in the *metadata* of JSIL objects. We differentiate the type of ER created using the `"@er_type"` property in the metadata: declarative ERs have type `"er_d"`, while object ERs have type `"er_o"`. The metadata property `"@er"` is kept for legacy reasons as a way to distinguish ERs from normal JS Objects in JSIL.

3.2.4.2 Declarative ERs

Declarative ERs can be seen as a dictionary of key-value pairs, where dictionary keys are JS identifiers and dictionary values are JS primitive values (undefined, null, boolean, string and number) or JS objects. These key-value pairs are termed as *bindings* in the ES5 specification.

Bindings are decorated with two flags: immutable and deletable. An *immutable binding* cannot be changed once it has been initialised while a *deletable binding* can have its associated key-value pair removed from the ER in a `delete` operation. Only *mutable bindings* (the opposite of immutable bindings) can be marked as deletable. For brevity, we will term immutable bindings as *immutables*, deletable bindings as *deletables*, and mutable bindings as *mutables*.

Usage of Immutables. Immutables have two uses in ES5: binding of named function expressions to an environment record (in both strict and non-strict modes), and the creation of the arguments object in strict-mode functions. In ES5, the production of a named function expression creates an additional ER known as a *closure*. The closure contains only one binding - the name of the function expression - and it is immutable. This allows the use of recursive calls within the function using its function name but this name cannot be accessed outside of the function expression.

In JSIL, we keep track of immutables of a declarative ER using an *immutable table* stored in the metadata property `"@immutables"` of the ER. The immutable table is a JSIL object where the field names are the string identifiers of the immutable bindings and the field values determine the state of the immutable. For each immutable, it could be in two states: uninitialised or initialised. Newly created immutables are uninitialised and marked with the value `false` in the immutable table while initialised immutables are marked with the value `true`. While the ES5 specification dictates that there are two states for immutables, both usages of immutables initialise the bindings immediately after creation. In fact, the creation of `constants` in ES6+ (third use case of immutables) also exhibit similar behaviour. As such, the state of the immutable after its creation is inconsequential in subsequent JS statements. This allows us to optimise the test for immutability to simply a check for the existence of the field name in the immutable table.

```
1  "use strict";
2  var x = function c() {
3    // Throws TypeError exception
4    c = 50;
5    return typeof c;
6  };
7  x();
```

Code Snippet 3.2: Immutable function expression (ES5 Strict)

```
1  var x = function c() {
2    c = 50;
3    return typeof c;
4  };
5  // returns "function"
6  x();
```

Code Snippet 3.3: Immutable function expression (ES5 Non-strict)

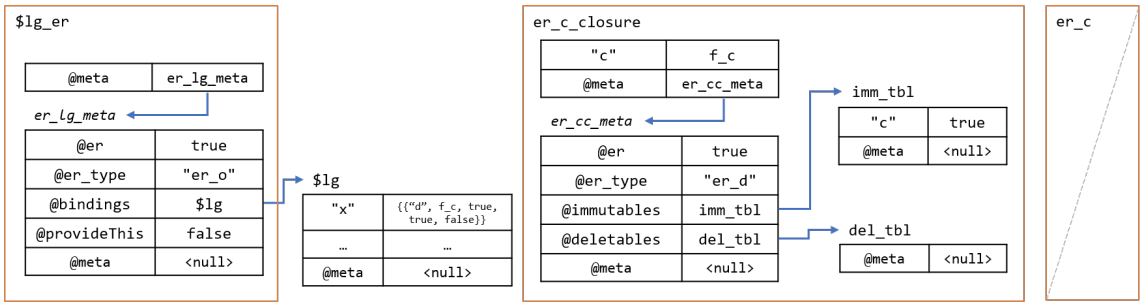


Figure 3.3: Scope chain of Code Snippet 3.2, Code Snippet 3.3

To illustrate the behaviour of immutables, we present an example (in strict and non-strict mode) where an attempt is made to modify the immutable binding of the function identifier in a named function expression. Code Snippet 3.2 represents the instance where the example is executed in strict mode while Code Snippet 3.3 represents the non-strict variant. Both instances make an attempt to modify the immutable binding of the function identifier `c`. They share the same scope chain during the execution of the function `c`, as depicted in Figure 3.3.

For simplicity, we will explain the strict mode case (Code Snippet 3.2) first. The immutable table is denoted by `imm_tbl`, where the identifier `"c"` is record as an initialised immutable. In line 4, an attempt was made to modify the value of `c`. This calls the `setMutableBinding` internal JSIL procedure which checks if the referenced name is an immutable. As a strict mode function, the mutation attempt on the immutable `c` will throw a `TypeError` exception.

The behaviour of the non-strict mode code (Code Snippet 3.3) also prevents the mutation of

immutables. However, instead of throwing a `TypeError` when an attempt was made to mutate `c` in line 2, it simply fails silently and no modification is made to the immutable `c`. As `c` has not been modified, `x()` returns `"function"` in line 6.

Usage of Deletables. Only newly created mutables can be marked as deletable. There is only one such usage in ES5: the dynamic declaration of function and variable identifiers in `eval` code. Bindings created in `eval` code are *configurable*, hence can be subjected to deletion after their creation. A binding is deleted via the `delete` operation, which will call the JS abstract operation on ERs: `DeleteBinding`.

We keep track of deletables with the use of a *deletable table* stored in the metadata property `"@deletables"` of every declarative ER. Similar to how the immutable table work for immutables, the deletable table contains field names for mutable bindings that are marked as deletable. As the table only contain the name of mutable bindings that are deletable, we only have to perform an existence check during `DeleteBinding` operation.

To illustrate the behaviour of deletables, we provide the example Code Snippet 3.4 along with its scope chain Figure 3.4. We run the example in non-strict mode in order for the dynamic declaration of variable `b` to be hoisted to the ER of function `f`. In this example, there two calls to the `delete` operation, one in line 2 and another in line 5. The first `delete` operation is called on the mutable `a` that is not `configurable`, hence the delete operation fails silently (statement returns `false`). In line 3, the execution of the `eval` code creates a new mutable and configurable binding for the variable `b`. This allows the variable to be deleted in line 5 (statement returns `true`).

```
1  function f(a) {
2      delete a; // false
3      eval("var b = 2;");
4      var res = a + b;
5      delete b; // true
6      return res;
7  };
8  f(1); // returns 3
```

Code Snippet 3.4: Deletables in ES5

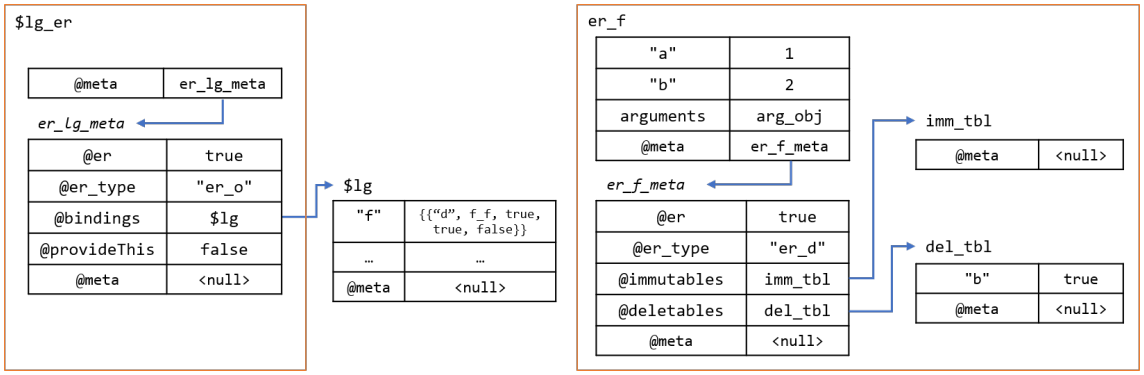


Figure 3.4: Scope chain of Code Snippet 3.4

In strict mode, function and variable identifiers cannot be deleted. Attempts to do so will throw a `SyntaxError` exception if the `delete` operation references the identifier directly (like in line 2 of Code Snippet 3.4), otherwise a `TypeError` is thrown.

Due to the semantics of strict mode code, simplifications could be made if the entire JS program (including any calls to `eval` and the `eval` code itself) is run in strict mode. In this case, all hoisted function and variable declarations are lexically scoped. Any `delete` operation on these hoisted declarations could be caught at compile-time and a `SyntaxError` would be thrown. Additionally, a deletable table would not be required in declarative ERs to keep track of deletables.

3.2.4.3 Object ERs

Object ERs use actual JS objects treated as dictionaries for resolving identifier bindings. These JS objects are known as *binding objects* and every object ER has one such binding object. The property names and descriptors (including all inherited properties) of the binding object are the identifier names and values of the object ER respectively. Intuitively, an object ER encapsulates its binding object to provide an interface for resolution of identifier bindings similar to declarative ERs.

There are three main differences that distinguish object ERs from declarative ERs:

1. All object ER bindings are mutable.

2. Deletion of an object ER binding is governed by the configurable field of its corresponding property descriptor, instead of a deletable flag store in the ER.

3. Object ER can optionally use its binding object as an implicit `this` value in function calls.

The consequence of the differences are twofold. First, we do not need to create two tables to keep track of immutables and deletables. Second, we require a new metadata property `"@provideThis"` to indicate if the object ER should use its binding object as an implicit `this` value.

Usage of the `"@provideThis"` flag. By default, the `"@provideThis"` is set to false. This flag is set to true only when the object ER is created within a `with` statement. Within the `with` statement, if the identifier of the function call resolves to a binding in the object ER, the `this` keyword within the function call will be set to the binding object.

To illustrate this behaviour, we provide the example Code Snippet 3.5 with its scope chain Figure 3.5. In this example, we define two global variables `a` and `obj`. `a` is initialised to the number `10` while `obj` is initialised as a JS object with two fields: `a` with number `1`, and `b` with `undefined`. In line 6, the use of `with` statement creates a new object ER with the `obj` as its binding object. Within the `with` statement, we create a new anonymous (strict) function that returns the sum `this.a + 2`. We assign this function to the property `b` of `obj` in line 7. When we call the function `b` in line 13, the identifier `b` gets resolved to a reference of the binding object `obj`. This causes the `this` keyword within the function to return `obj`. The result of the sum `this.a + 2` is now equivalent to `obj.a + 2`, which returns 3 (in line 13). Intuitively, the function call in line 13 is semantically-equivalent to a property call `obj.b()`.

As a point of comparison, we assign the function to a new variable `c` (which lives in the global ER) in line 12. When we perform a function call in line 13, the resolution of the identifier `c` is the global ER. The global ER is also an object ER, with the global object as its binding object. However, the global ER's `provideThis` flag is `false`, which causes the `this` keyword in the function to return `undefined`. Note that if the function is defined in non-strict mode, the `this` keyword will default to the global object.

```
1  var obj = {
2      a: 1,
3      b: undefined
4  };
5  with (obj) {
6      b = function() {
7          "use strict";
8          if (this === undefined) return -1;
9          return this.a + 2;
10     };
11     // this is equivalent to obj.b();
12     b(); // returns 3
13     c = b;
14     c(); // returns -1
15 }
```

Code Snippet 3.5: Use of `provideThis` flag in Object ER

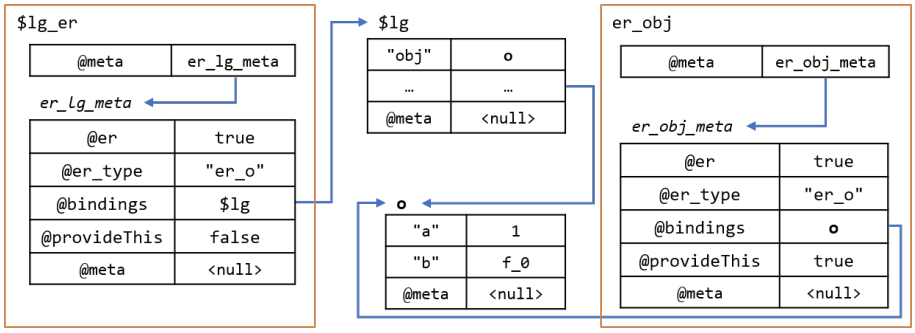


Figure 3.5: Scope chain of Code Snippet 3.5

3.3 Dynamic name resolution

In general, name (identifier) resolution in ES5 is dynamically scoped[5]. We could make simplifications if we restrict the entire JS program to strict mode (including all runtime execution of `eval`). In that situation, name resolution is static and lexically scoped. JaVerT ([8]) makes use of lexical scoping semantics to build a *closure clarification* table at compile time to resolve JS identifiers to their respective ERs in the scope chain. Intuitively, a closure clarification table maps JS identifiers to the environment record that holds its binding (if it exists)[6]. By extending JaVerT to ES5 (non-strict), we can no longer use this simplification, and have to implement internal JSIL procedures to resolve JS identifiers dynamically at runtime.

Dynamic name resolution in ES5 involves over 10 sections of the specification that covers both lexical environment operations and object internal methods. We will not go through all these functions in detail as most of them are purely algorithmic or only require minor modifications from the previous version of JaVerT. For interested readers, we listed the list of lexical environment operations implemented in this project in Table A.2. Instead, we will focus on the main operation that enables dynamic name resolution in ES5: `GetIdentifierReference` (Section 10.2.2.1)

To illustrate our approach, we present the line-by-line compilation of our dynamic name resolution JSIL procedure `"getIdentifierReference"` that is based on the abstract lexical environment operation in ES5 `"GetIdentifierReference"`.

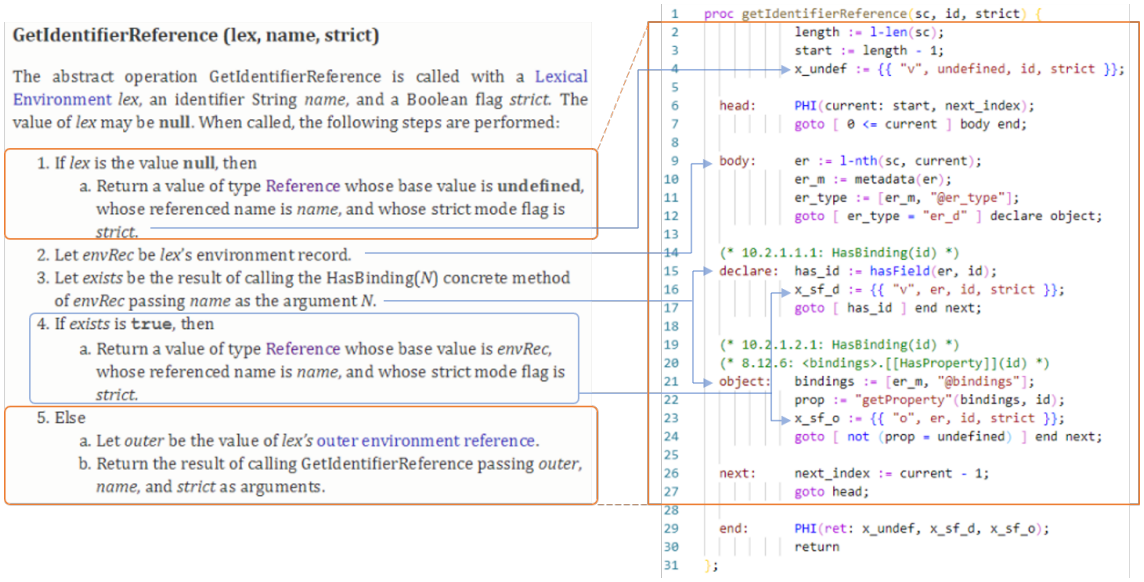


Figure 3.6: `getIdentifierReference` Internal JSIL procedure

[5]For clarity, we refer to scope resolution that has to be done at run-time as dynamic scope resolution and scope resolution that can be computed at compile-time as lexical scope resolution.

[6]To be more precise, the closure clarification table maps JS identifiers to a specific index of the current scope chain, which holds the environment record with its binding (if it exists).

With reference to Figure 3.6, specification for `"GetIdentifierReference"` is shown on the left while our implementation is shown on the right. Our implementation has significant differences from the specification which we will explain based on the algorithmic flow of the specification:

1. The first difference is in our use of scope chains instead of lexical environments. As such, we pass in the current scope chain (`sc`) as a run-time argument instead of a lexical environment (`lex`). A scope chain is a simply a list, hence we transform the recursive calls in the specification (orange boxes on the left) to the iterative loop in our implementation (orange box on the right).

2. Since environment records in the scope chain are ordered by their creation time, we iterate on the scope chain backwards, starting with the last environment record in the scope chain.

3. In each iteration, we check if a binding exists for the identifier `id`. In the specification, an abstract operation `HasBinding` of the environment record is called, passing the parameter *name* (equivalent to `id` in our JSIL procedure). However, we noticed that we can make a small optimisation by *inlining* the operation call into the main body of the loop (lines 14 to 24 on the right). There are three benefits to this approach:

   (a) We reduce a function call for every identifier resolution at run-time, potentially improving run-time performance significantly

   (b) With one less function call, we could obtain a more simplified call stack during debugging.

   (c) We require the procedure to return JSIL references that have types based on the environment record in which the first binding for the identifier is found. A similar distinction between ER types is also made in our implementation of `hasBinding`. By inlining the procedure call, we also reduce the duplicated ER type check.

4. If the binding exists in a particular iteration, we exit the loop and return the appropriate JSIL reference of the following form:

   (a) (1st element) Type `"v"` if found in declarative ER, type `"o"` if found in object ER.

   (b) (2nd element) Reference base, which is the ER where the binding is first found.

   (c) (3rd element) Reference name, which is the identifier `id`.

   (d) (4th element) A flag to indicate if the reference obtained in strict mode (inherited from the parameter `strict`).

At this juncture, it is important to note that we **cannot** switch between lexical and dynamic scoping based on the mode of execution of individual executable code components of a JS program. Once an executable code of a JS program is run in non-strict mode, we lose lexical scoping for the **entire program**. To help the reader understand the consequence of intermixing execution mode within a JS program, we provide a simple JS program below:

```
1  "use strict";
2  typeof a; // "undefined"
3  (0, eval)("a = 2;");
4  typeof a; // "number"
```

Code Snippet 3.6: Mixed execution mode

With reference to Code Snippet 3.6, let us explain the program step-by-step:

1. In line 1, we have a strict mode directive (`"use␣strict;"` at the top level (global). This implies that all global and function code components of the program must be executed in strict mode.

2. In line 2, as the identifier `a` does not exist in the global scope, the type of `a` is `undefined`.

3. In line 3, we **inject non-strict behaviour at runtime** with the use of an **indirect** `eval` (recall the execution modes of `eval` in §2.1.2.3). The execution of the `eval` code `"a␣=␣2;"` creates a new global variable `a` initialised with the number 2.

4. In line 4, due to the creation of `a` in line 3, we obtain a different result with the same statement in line 2, obtaining the type of `a` as `number`.

If we compile Code Snippet 3.6 by switching between lexical and dynamic scoping, we would be using the closure clarification tables to resolve the identifier `a` in **both** line 2 and 4 since both lines are executed in strict mode. Since the closure clarification table is built at compile-time, it would not contain a valid mapping for `a` that is dynamically created by the `eval` call in line 3. As such, line 4 would return the incorrect type - `undefined` - with the use of the closure clarification table.

It should also be noted that if we were to modify line 3 of Code Snippet 3.6 into a direct `eval` call or use a strict mode directive in its `eval` code: `"use␣strict;␣var␣a␣=␣2;"`, the entire program would be in strict mode, hence preserving lexical scoping.

3.4 Compilation of the `with` statement

Having explained the state management and dynamic name resolution implemented in JSIL, we now have the necessary constructs to begin the compilation of the `with` statement.

In §2.1.2.1, we explained the semantics of the `with` statement in ES5. It is a non-strict-mode-only feature and is the first language feature we had to implement when extending the compiler. We present in this section a line-by-line compilation from the ES5 specification into JSIL using a generalised `with` statement of the following form: `with (e) {s}`, where `e` is a JS expression and `s` is a JS statement.

With reference to Figure 3.7, the semantics of the `with` as per ES5 specification is shown on the left, and its correponding compiled JSIL commands are shown on the right. Following the steps of the ES5 specification, the compilation is as follows:

1. We start by recursively calling our compiler on the expression `e` to generate a list of commands `cmds1` that evaluates it and store the result in a fresh variable `x1`.
2. Next, we coerce the result of the evaluated expression to an object in a two-step process: (a) First, call `i__getValue` (JSIL procedure for abstract operation `GetValue`), passing `x1` as the parameter and storing the result in a fresh variable `x_v`; (b) Next, call `i__toObject` (JSIL procedure that implements JS abstract operation `ToObject`) passing `x_v` as the parameter and storing the result in a fresh variable `x_o_1`.
3. The current LexicalEnvironment is given by the current scope chain: `x_sc_0`.
4. Using `x_o_1` as its binding object, a new object environment record is created in the call to JSIL procedure `newObjectEnvironment` (implementation of the JS abstract operation `NewObjectEnvironment`). The newly created object ER is stored in `x_er`. Note that we do not require a reference to the current execution context's LexicalEnvironment for the construction of a new object environment record due to our choice of using a scope chain for state management.
5. We set the `provideThis` flag of the newly created ER to true by retrieving its metadata and setting the field `"@provideThis"` to true.
6. Next, we update the state of execution by appending `x_er` to the current scope-chain, `x_sc_0`, storing the result to a fresh variable `x_sc_1`. Internally within the compiler, we also update a compile-time state management construct known as a *translation context* with the new scope chain reference[7].
7. With the updated scope chain, we call the compiler on the JS statement `s` to generate a list of commands `cmds2`, which evaluates `s` and stores the result in a fresh variable `x2`.
8. As our compiler maintains the reference to the variable that holds the original scope chain (`x_sc_0`) with which the `with` statement is called on, we do not need to explicitly restore the scope chain at the end of the compilation.

[7] More precisely, a new translation context is created with updated states every time a change is made to the execution state.

9. By construction, J2-2-JSIL compiler will return the the list of JSIL commands shown in the figure along with the variable that stores the return value of the `with` statement. In this case, the return value will be stored in variable `x2`.

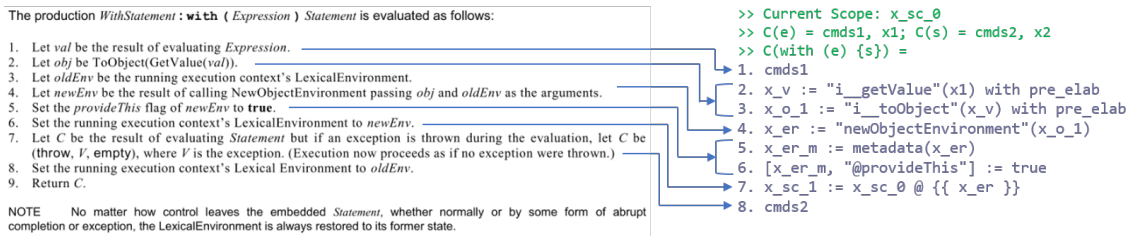


Figure 3.7: Line-by-line compilation of `with (e) {s}`

3.5 Hoisting of variable and function declarations

Hoisting is a mechanism used to forward declare variables and functions. One of the primary benefits of hoisting function declarations is the ability to perform mutual recursions. In ES5, variable and function declarations are hoisted to the VariableEnvironment of the current execution context. In all statically compiled JS code, at the moment of hoisting, the VariableEnvironment and LexicalEnvironment point to the same lexical environment. This implies that we could hoist declarations directly to the last environment record in the scope chain in JSIL.

However, this assumption is broken when dealing with dynamically compiled code. In particular, we could construct a situation where the LexicalEnvironment and VariableEnvironment point to different lexical environments with the use of the `with` statement. Additionally, we can force the creation of a new execution context that inherits the LexicalEnvironment and VariableEnvironment with a direct and non-strict `eval` call. In this case, the last environment of the scope chain is no longer the correct environment record that should receive the hoisted declarations contained with the `eval` code.

Let us illustrate this behaviour with the example below (Code Snippet 3.7):

```
1  var obj = {
2      inner: 1
3  };
4  function outer() {
5      with (obj) {
6          eval("function inner(a) { return inner + a; };");
7          if (typeof inner === "function") {
8              // Never reached
9              return inner(2);
10         }
11     }
12     return inner(3);
13 }
14 outer(); // returns 4
```

Code Snippet 3.7: Dynamic hoisting

In this example, the key modifications to the scope chain and hoisted declarations are as follows:

1. In line 5, we extend the scope chain of the function `outer` with an object ER using `obj` as its binding object.

2. In line 6, we make a direct and non-strict `eval` call. Within the `eval`, it declares a new function `inner` that takes a single formal parameter `a`. As the `eval` call inherits the current state of execution, we would require the newly declared function to be hoisted to the declarative ER of function `outer` instead of the object ER of the `with` statement created in line 5. This implies that the `inner` property of `obj` is **not** updated to the new function.

3. Additionally in line 6, notice that there is an reference to the identifier `inner` within the function body of the function `inner`. Due to the hoisting mechanism above, this `inner` reference within the function body resolves to the property `obj.inner` instead of the newly declared `inner` function.

4. We can do a check to ascertain that the type of `obj.inner` has not changed and the dynamically declared `inner` function can only be called outside the `with` statement in line 11.

The difficulty in implementing hoisting lies in the 2nd step. Specifically, we require an internal JSIL procedure to obtain the equivalent of the VariableEnvironment from the current scope chain. To do so, we need to analyze the situations where the VariableEnvironment is updated and used for hoisting. There are three situations corresponding to the three types of executable code:

1. Global code. Declarations are hoisted to the global ER, or more precisely, as properties of the global object.

2. Function code. Declarations are always hoisted to the new declarative ER that is created for the function.

3. `eval` code. There are three sub-cases for `eval`:

   (a) Strict mode. An additional declarative ER is created and used for hoisting.

   (b) Non-strict and direct call. Inherits the caller's VariableEnvironment and uses it for hoisting. Notice that this VariableEnvironment will always be a declarative ER even if the caller is within a `with` statement.

   (c) Non-strict and indirect call. Inherits the global environment and hoists declarations to the global ER.

In all of these cases, we observe that declarations are always hoisted to either the global ER (case (1) and (3)(c)) or to the latest declarative ER (all other cases). As such, we can do an iterative search on the current scope chain to obtain the last declarative ER or the global object (whichever is found first) as the appropriate ER for hoisting declarations. This is essentially the implementation of the internal JSIL procedure `getVariableEnvironment`.

The subtlety in the choice of the lexical environment that receive hoisted declarations creates counter-intuitive behaviours in dynamically generated code. Specifically, care should be taken when executing `eval` code in non-strict mode.

3.6 The `arguments` object

We briefly explained the semantics of the `arguments` object between the strict and non-strict mode in §2.1.2.2. The primary difficulty of implementing the `arguments` object in non-strict mode is the two-way binding between the JS function's formal parameters (also known as named parameters) and the elements within the `arguments` object itself. This binding only exists in non-strict mode and is broken in two situations: (1) the property in the `arguments` object is deleted, or (2) the property is redefined into an accessor property. To maintain the mapping between formal parameters and properties of the `arguments` object, we require two constructs to assist us:

1. A `map` object that keeps track of properties in the `arguments` object that are mapped to formal parameters. This object is stored in the metadata property `"@parameterMap"` of the `arguments` object. Each property of the `map` object represents a mapped formal parameter, and stores an accessor descriptor.

2. Each accessor property of the `map` field above require an abstract argument getter and setter to retrieve and modify the actual properties of the `arguments` object. As such, we implement JSIL procedures to dynamically create functions for both the getters and setters.

The core component that enables the above two constructs to work together are the modified `Get`, `GetOwnProperty`, `DefineOwnProperty` and `Delete` internal methods for the `arguments` object.

These internal methods are only used in non-strict mode, hence we set a metadata property `"@argInternals"` of `arguments` object as a flag that will indicate to the run-time if these methods should be used. We will not explain line-by-line the details of all these functions as they are purely algorithmic, but will provide a general intuition on how these are used to provide the dynamic binding behaviour between formal parameters and fields of the `arguments` object.

All modified internal methods take at least one parameter - a property name in the `arguments` object. If this property name references a mapped property within the `arguments` object, the call is redirected into internal method calls of the `map` object. Otherwise, it redirects the method call to the default object internal methods of the `arguments` object. A special case exists in the modified `DefineOwnProperty` which deletes a mapped property if the new descriptor definition is an accessor descriptor. This removes the two-way binding for that property and its corresponding formal parameter.

3.6.1 Two-way binding example

Let us illustrate the behaviour of the mapping with the example Code Snippet 3.8. In this example, we have a simple function `f(a, b)` that takes in two (formal) parameters `a` and b. At run-time, we call the function with **three** parameters: `f(3, 4, 5)`. This results in a creation of an `arguments` object, `arg_obj` as shown in Figure 3.8.

```
1  function f(a, b) {
2    arguments[0] += arguments[2];
3    var del = delete arguments[0];
4    if (del) return a;
5    return b;
6  }
7  f(3, 4, 5); // 8
```

Code Snippet 3.8: `arguments` object behaviour

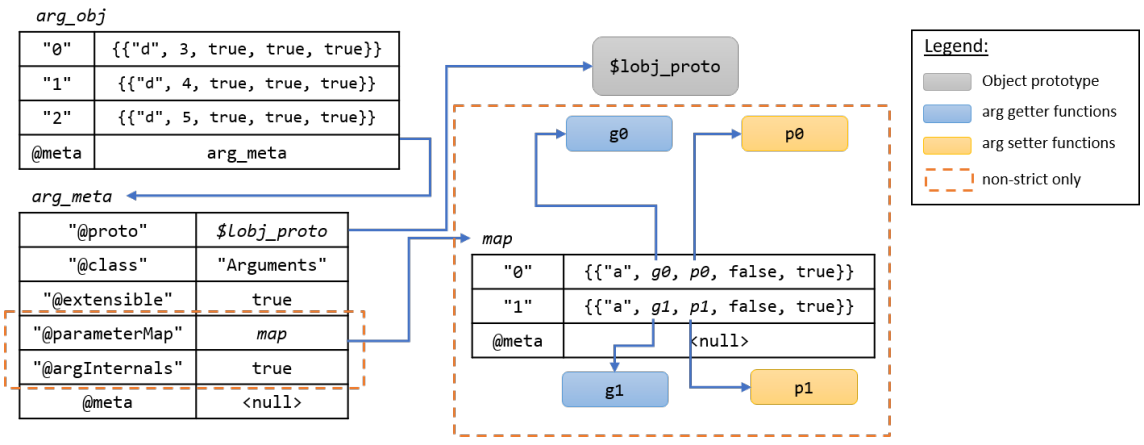


Figure 3.8: `arguments` object example

In both strict and non-strict modes, `arg_obj` will be initialised with 3 fields. The field names are the index of the run-time parameters: `"0"`, `"1"`, `"2"`. Their corresponding field values are data descriptors containing the value of the parameter, along with all remaining three attributes (writable, enumerable and configurable) set to **`true`**. As there are only two formal parameters, the `map` object contains only two fields, one for each of the formal parameter. The third run-time parameter is only accessible via the `arguments` object and is not mapped to the formal parameters.

Within the function body of `f`, we perform a compound assignment to `arguments[0]` by adding `arguments[2]` to `arguments[0]`, giving `8`. Due to the two-way binding, the formal parameter `a` also gets updated. Properties of `arg_obj` created by default in the compilation process are configurable, hence can be deleted. We delete the property `"0"` of `arg_obj` in line 3 which causes the function to return `a` (which was updated in line 2), with the value `8`.

3.6.2 Non-strict `callee` property

In addition to the two-way binding explained above, the semantics of the `"callee"` property of the `arguments` object is modified in non-strict mode to return the current function object. While this change in semantics is trivial to understand, we face a dilemma when implementing it in JSIL. To explain this dilemma, we need to first introduce the previous implementation of the compilation process that translates JS programs into JSIL procedures.

Prior to this project, translated JSIL procedures are of the following format: `proc fid (x__sc, x__this_arg, <formal_params>)`, where `x__sc` is the scope chain of the caller, `x__this_arg` is the `thisArg` passed to the function for resolving `ThisBinding`, and `<formal_params>` is a placeholder for the translated formal parameters of the original JS function. Function objects are created during function calls (which reside outside of the translated JSIL procedure) while creation of the `arguments` object is done within the JSIL procedure. This meant that we do not have access to the function object during the creation of `arguments` object.

To resolve this issue, we thought of three approaches:

1. Add the function object as a new first parameter for translated JSIL procedures, while keeping the rest of the parameters.
   (a) This is a breaking change for any JSIL procedures that depend on the procedure parameter format and/or depend on run-time optional parameters. Specifically, these include most of the JSIL implementation of built-in JS functions. These procedures depend on implicit assumptions on the number of run-time parameters and obtain their parameters via the `args` JSIL command. As access to this parameters is done via array indices, introducing a new first parameter would effectively create a "right shift" (increase index by 1) for all subsequent parameters. We have to modify all of these functions to take into account this change.
   (b) The primary advantage of this approach is for any other JSIL procedure that references the caller scope chain `x__sc` directly. This change would preserve direct references to the scope chain without resorting to additional calls into the function object to obtain it.
   (c) Additionally, the semantics of the specification assumes that we have access to the function object within the function body during compilation. This change would preserve the semantics of the specification.
2. Replace the scope chain (first) parameter of translated JSIL procedures with the function object. The scope chain of the caller is then referenced from the function object's `"@scope"` metadata property.
   (a) This likely to be the most intrusive change as it could potentially affect most of the JSIL procedures. Any JSIL procedure that directly references the caller scope chain must be modified with additional steps to obtain the scope chain from the function object. Additionally, JSIL procedures that references run-time parameters via the `args` command could also be affected if they expect the first parameter to be the caller scope chain.
   (b) The main benefit of this approach is that it preserves the number of parameters in all existing JSIL procedures. Additionally, we do not duplicate the scope chain data that could be referenced from the function object anyway.
   (c) This approach also has the benefit of preserving the semantics of the specification (as with approach (1)).
3. Add a global JSIL variable that stores the latest function object. Since we know that the `arguments` object is created only in the body of the JSIL procedure, we can update the global reference with the function object used for each function call before entering the JSIL procedure. This allows us to safely assume that we would be able to obtain the reference correct function object during the creation of the `arguments` object within the procedure.
   (a) This approach is likely to be the least intrusive as no major changes are required for any JSIL procedure.

(b) The main disadvantage of this approach is that we pollute the heap with a temporary variable in order to keep track of a local state. This does not conform to the semantics of the specification and has no guarantees that our assumption would continue to hold in future specification. Intuitively, this feels like a “hack".

The considerations for selecting our approach are two-fold. First, we want to preserve the semantics of the specification. Second, we want to introduce as minimal changes to the code base as possible. Approach (3) does not fulfil the first criterion and is immediately removed from consideration. Between approaches (1) and (2), (1) is ultimately chosen as it is less intrusive than (2).

The implementation of approach (1) results in the updated JSIL procedure format: `proc fid (x__func_obj, x__sc, x__this_arg, <formal_params>)`. This updated format allows us to directly pass in the function object `x__func_obj` during the creation of the `arguments` object for the `callee` property.

3.7 Direct and indirect `eval`

In §2.1.2.3, we explained the difference between direct and indirect `eval` calls. In short, a direct `eval` call has two conditions:

1. The reference obtained from evaluating the *MemberExpression* (left side of the expression before the parameters) has an environment record as its base (i.e. binding must be obtained from an ER), and the reference name must be ``eval".
2. The result of calling `GetValue` abstract operation on the referenced obtained from (1) must be the built-in `eval` function.

To distinguish between these two types, we implemented the internal JSIL procedure `isDirectEval(ref, ref_v)` as shown in Figure 3.9.

```
1   proc isDirectEval(ref, ref_v) {
2       (* Check if ref is a reference type *)
3       goto [typeOf ref = List] test no;
4   test:   goto [(l-nth (ref, 0) = "v") or (l-nth (ref, 0) = "o")] rcand no;
5
6   rcand:  is_prop := "i__isPropertyReference" (ref);
7       is_er := not(is_prop);
8       rfield := l-nth (ref, 2);
9       (* (1) ref has ER as base value, "eval" as referenced name *)
10      goto [is_er and (rfield = "eval")] check no;
11      (* (2) ref_v is the standard built-in eval function $lg_eval *)
12  check:  goto [ref_v = $lg_eval] yes no;
13
14  yes:    ret_yes := true;
15      goto rlab;
16
17  no:     ret_no := false;
18
19  rlab:   PHI(ret: ret_yes, ret_no);
20      return
21  };
```

Figure 3.9: `isDirectEval` implementation

The procedure is called when a function call expressions evaluates to a call to the built-in `eval` function. It takes two parameters: `ref` and `ref_v`. `ref` is the reference obtained from evaluating the *MemberExpression* in the function call while `ref_v` is the result of calling `GetValue` on `ref`.

We make a preliminary check in lines 2 and 3 to ensure that `ref` is a JSIL reference. If this check fails, we return false. Next, we check if `ref` fulfills the first condition in lines 6 to 10. JS

references can be classified into *property references* and *ER references*. We perform this check in line 6 by calling the internal JSIL procedure `i__isPropertyReference`, which returns `false` if it is an ER reference. The referenced name can be obtained from the 2-th element in the list `ref` in line 8. We then perform the actual check for the first condition in line 10. If this check passes, we proceed to check for the second condition in line 12 using our global JSIL location reference to the built-in `eval` function (`$lg_eval`). If any of the two checks fails, we return `false`. Otherwise, we return `true`.

At first glance, there appears to be a duplication of parameters as we could have obtain `ref_v` from `ref` by calling `GetValue` within the procedure. However, we observe that the compilation of a function call would always make a `GetValue` call on the reference obtained from evaluating *MemberExpression*. This meant that we would already have obtained the results to check for the second condition prior to the check for direct `eval`. As such, we could perform an optimisation in the run-time by directly passing the result of the `GetValue` call directly into `isDirectEval` as its second parameter.

Chapter 4

Analysing ES5 Programs

In this chapter, we take a deeper dive into the intricate details of ES5. We will base most of our discussions around symbolic programs as examples that reveal the complexity of the language. Specifically, we showcase how small details of ES5 can have big impact on the entire program.

We organised this chapter into two sections. The first section (§4.1) presents a concrete JS program with an intentional bug that is difficult to detect at first sight. It highlights a small feature of JS and serves as a motivation for moving towards symbolic testing. We then proceed to analyse JS programs using the whole-program symbolic testing aspect of JaVerT, which we enabled for ES5, in §4.2. We present these symbolic programs in the style of Test262 (ECMAScript's official test suite) tests. Our main contributions in these symbolic tests are two-fold: (1) the generalisation of concrete tests, and (2) providing new tests that check for behaviour previously not found in the official test suite.

4.1 Motivating Example

Concrete tests check the behaviour of a program based on concrete values and are usually the primary form of testing for many programs. The official test suite for ECMAScript, Test262, is also based on concrete tests. As developers, we use concrete tests to provide an indication of correctness. However, it may not always be feasible to generate tests for an entire range of inputs. This problem is further compounded when dealing with complex languages such as JavaScript. Its many subtle semantics as a dynamic language may confuse developers who are not familiar with the language.

We present a small JS program (Code Snippet 4.1) as a motivating example in this section. It highlights limitation of concrete tests and a feature of ES5 JS that is often overlooked by developers from other languages such as C or Java - `hoisting`.

```
1  function isPrime(x) {
2      var prime = false;
3      if (x < 2) return prime;
4      if (x === 2) return !prime;
5
6      for (var i = 2; i <= Math.sqrt(x); i += 2) {
7          var prime;
8          if ((x % i) === 0) { prime = false; }
9          if (prime !== undefined) return prime;
10     }
11     return true;
12 }
13
14 var collect = [];
15 for (var i = 0; i < 10; ++i) {
16     if (isPrime(i)) { collect.push(i) };
17 }
18 collect; // [2, 3, 5, 7]
```

Code Snippet 4.1: Non-block-level scope declarations in ES5

With reference to Code Snippet 4.1, we have a simple JS program with a single function `isPrime`. Given an integer `x`, `isPrime` returns **`true`** if `x` is prime, **`false`** otherwise. At first glance, from the standpoint of, for example, a C developer, this function appears to behave correctly. We can also test it against the first 10 integers (lines 15 to 19) and check that the result is indeed accurate. However, if we check the result of `isPrime(11)`, we would get **`false`**! `11` is clearly prime, which means there is a bug in `isPrime`.

This bug actually stems from *variable hoisting* in ES5 JS. In line 2, we declared a function-level variable `prime`. We re-declare a variable with the same name `prime` in line 8. In ES5, variables are always hoisted to the current execution context, which means the `prime` in line 8 is also hoisted to the function-level. Re-declared variables without initialisation in ES5 are essentially ignored. Hence, the declaration in line 8 is equivalent to an empty statement. This means that `prime` is always defined, and will return **`false`** whenever control enters the loop. We intentionally crafted the program such that control will only enter the `for` loop for values of `x` greater than 10, as square root of `10` is greater than 3, which is the starting value of `i`. This is also why we successfully generate the first 4 primes between 0 and 10.

Contrast this to block-level scoped languages like C, where a similar program would create a local variable `prime` in the block scope of the `for` statement in lines 7 to 11. This local variable would *shadow* the outer variable `prime` at the function-level. The `prime` in line 8 would be `undefined` until we find an `i` that divides `x`, which will set `prime` and return `prime` in lines 9 and 10 respectively.

As a point of interest, we could actually emulate block-level scoping using the `with` statement in non-strict JS programs. A rewriting of Code Snippet 4.1 into block-level scoped variables is presented in Code Snippet 4.2. In this program, we create a new lexical environment with a **new** object `{prime: undefined}` in every iteration of the `for` loop. References to the identifier `prime` in lines 10 and 11 would point to property of the object instead of the function-level `prime`.

```
1  function isPrimeWith(x) {
2      var prime = false;
3      if (x < 2) return prime;
4      if (x === 2) return !prime;
5
6      for (var i = 2; i <= Math.sqrt(x); i += 2) {
7          with ({prime: undefined}) {
8              // prime contained in block due to with statement
9              if ((x % i) === 0) { prime = false; }
10             if (prime !== undefined) return prime;
11         }
12     }
13     return true;
14 }
15
16 var collect = [];
17 for (var i = 0; i < 10; ++i) {
18     if (isPrimeWith(i)) { collect.push(i) };
19 }
20 collect; // [2, 3, 5, 7]
21 isPrimeWith(11); // true
```

Code Snippet 4.2: Emulation of block-level scoping using `with` statement in ES5

If we examine the versions of the standard beyond ES5, we will notice that ES6 introduced the `let` construct, which declares lexically-scoped variables. An equivalent rewriting of the block-scoped ES5 program Code Snippet 4.2 would be the program in Code Snippet 4.3.

Of course, one could argue that the example we provided in Code Snippet 4.1 is clearly contrived. Reusing variable names within the same function in nested blocks is generally not an ideal programming style. The variable declaration in line 2 is also clearly unnecessary; the `prime`s in lines 3 to 5 can be replaced with primitive booleans directly. However, we could generalise this behaviour into larger programs with more lengthy and complex functions, and similar unintentional reuse of variable names. In these situations, proof-reading entire programs to avoid such situations may become tedious or infeasible. As such, we move towards whole-program symbolic testing, where we test JS programs using symbolic values that represents **abstract concepts**, such as numbers, strings, or general JavaScript values.

```
1  function isPrimeES6(x) {
2      let prime = false;
3      if (x < 2) return prime;
4      if (x === 2) return !prime;
5
6      for (let i = 2; i <= Math.sqrt(x); i += 2) {
7          // Block scoped variable
8          let prime;
9          if ((x % i) === 0) {
10             prime = false;
11         }
12         if (prime !== undefined) return prime;
13     }
14     return true;
15 }
16
17 var collect = [];
18 for (var i = 0; i < 10; ++i) {
19     if (isPrimeES6(i)) { collect.push(i) };
20 }
21 collect; // [2, 3, 5, 7]
22 isPrimeES6(11); // true
```

Code Snippet 4.3: Equivalent block-level scoping in ES6

4.2 Whole-Program Symbolic Testing

Whole-program symbolic testing is a feature of JaVerT that enables the use of symbolic constructs to test entire JS programs. The primary benefit of symbolic tests is the generalisation of concrete tests. These provide stronger guarantees in the correctness of the program as we test against an abstract range of inputs instead of concrete instances.

This section is organised into three logical parts:

1. We start by introducing the core symbolic execution constructs of JaVerT that we will be using throughout the rest of the chapter in §4.2.1.

2. Subsequently, we present key symbolic tests that illustrate different ES5 behaviours from section §4.2.2 to §4.2.7 (one section per example). We will also highlight symbolic tests that check for ES5 behaviour previously not found in the Test262 test suite.

3. Finally, we summarise our main contributions to these symbolic tests in §4.2.8.

4.2.1 Core symbolic execution constructs

In this section, we introduce the core symbolic constructs used in this chapter. The full mechanics of the symbolic execution engine and its formalisation is beyond the scope of this project. Interested readers are directed to the following papers for more details: [24, 8].

There are three main constructs that we employ in our symbolic tests:

Symbolic values. During whole-program symbolic testing, there are four types of symbolic values that are directly available to the developer: numbers, booleans, strings, and general values. Symbolic numbers, booleans and strings are simply symbolic variants of the intuitive definition of concrete numbers, booleans and strings in programming. We can initialise a JS variable with these types of symbolic values by using the functions `symb_number`, `symb_boolean`, `symb_string`, and `symb`. For example, a JS statement `var x = symb_number(x)` informs the symbolic execution engine that `x` is symbolic, and contains a symbolic number.

Assumptions. We can place constraints on symbolic values with the use of *assumptions* in the symbolic engine. For instance, we can inform the engine that a symbolic number `x` can only contain even numbers with the following JS statement: `Assume((x % 2) = 0)`. Note that the `Assume`

function acts like a directive for the symbolic engine. The argument to the function, an expression written in JSIL, is the assumption the engine must make in the execution of the rest of the program.

Assertions. Assertions are the tests for the program. Similarly to the mechanics of `Assume`, described above, an `Assert` function takes a JSIL expression that when evaluated, must return `true`. For instance, `Assert(not ((x % 3) = 0))` checks that `x` must not be divisible by 3. As these tests are symbolic, the symbolic engine will attempt to create a *model* (a set of inputs) in which the assertions might fail.

4.2.2 Declaration hoisting order

We will start with a simple symbolic test in Figure 4.1 (on the left). For clarity we will refer to the program on the left of the figure as (a) and the program on the right as (b) This test illustrate the implicit *order of hoisting* between variable and function declarations.

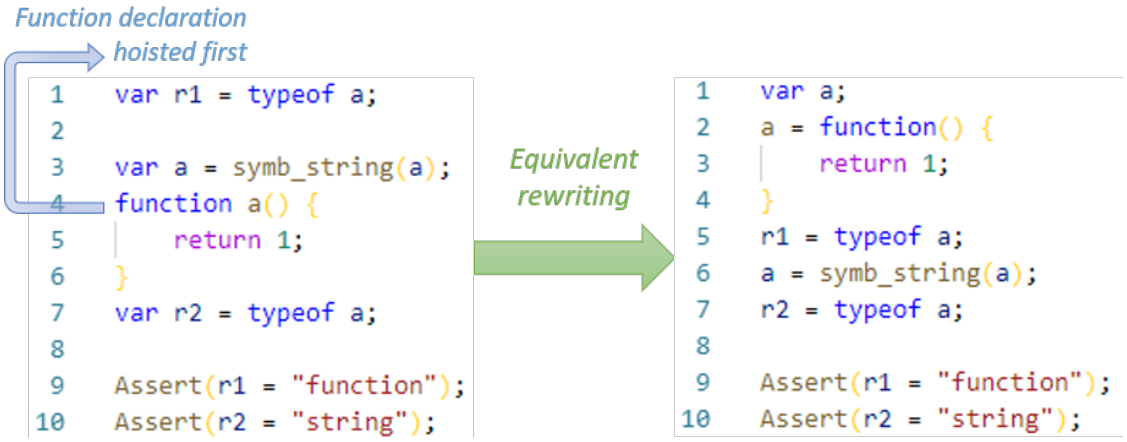


Figure 4.1: Declaration hoisting order: (a) original symbolic test (left); (b) rewritten test (right)

In program (a) of Figure 4.1, we make two type checks on the identifier `a` (line 1 and line 7). There are two declarations of `a`: first in line 3, assigned with a symbolic string; second in line 4, assigned with a function object. If we were to interpret the program sequentially, we would assume that the first type check in line 1 would give us a string, and the second in line 7 to give us a function.

However, in ES5, function declarations are always hoisted before variable declarations. As such, the function declaration in line 4 is hoisted **before** the variable declaration of the same identifier `a` in line 3. This results in the first type check to return `"function"` and the second to return `"string"`. An equivalently rewriting of the program to the semantics of the hoisting behaviour in ES5 is given on the right of Figure 4.1.

We can check for the correct hoisting behaviour with the use of the `Asserts` in lines 9 and 10 of (a). As this tests for the type of the return values instead of over a range of outputs, the value in using symbolic tests is not very high. In fact, one could argue that this example is almost equivalent to a concrete test with the exception of the single use of symbolic string in line 3 (which is also arguably superfluous). To better illustrate the value of symbolic testing, we will proceed with more complicated examples in the subsequent sections.

4.2.3 Non-block-level declarations

In this section, we will revisit the `isPrime` example we gave in §4.1, rewritten into a symbolic test. We made a few tiny adjustments to make it easier to check for incorrect behaviour.

With reference to Figure 4.2, we first explain the differences between the `isPrime` function here and in §4.1:

1. We changed the initialisation of variable `prime` from `false` to `true`. We made this change

to allow the function to incorrectly return certain non-primes as primes, instead of always rejecting every number greater than 9.

2. We shifted the even number check `x % 2 === 0` to be part of the divisibility check in the `for` loop in line 11.

3. Due to a limitation of the symbolic engine, the `Math.sqrt` function has not been implemented symbolically. As such, we regress to a more inefficient algorithm that checks for divisibility from 2 to one less than `x`.

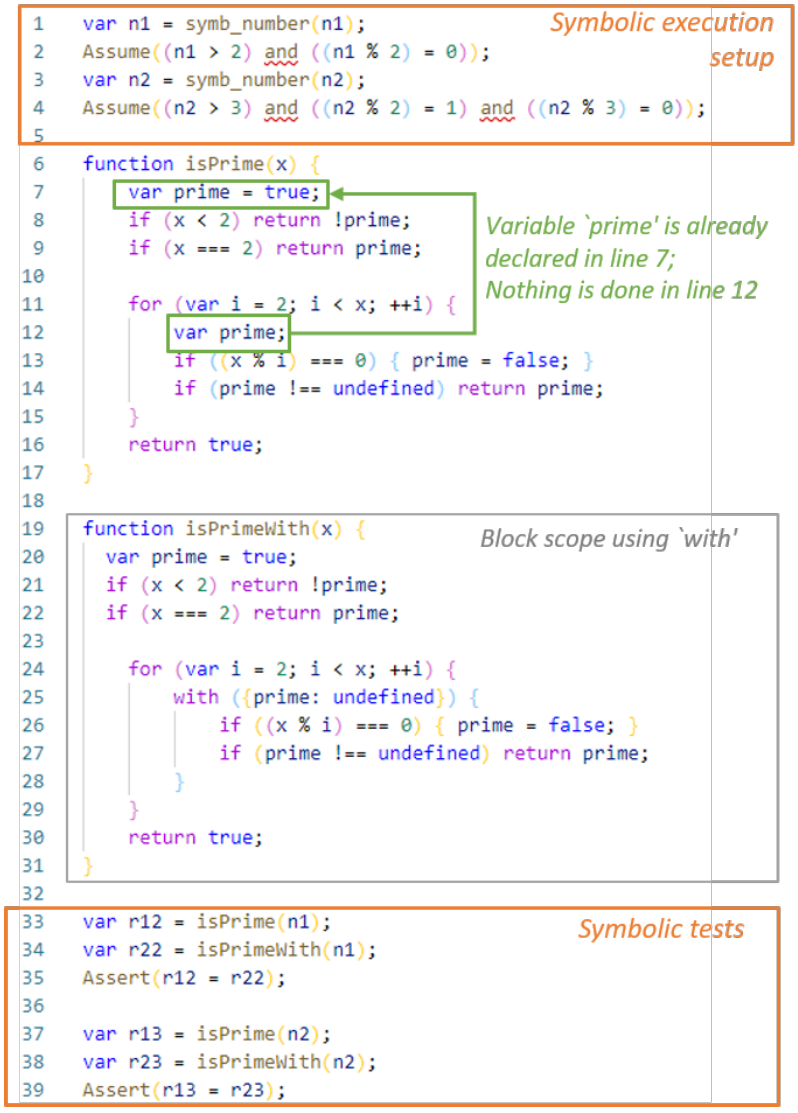


Figure 4.2: Variable declaration within `for` statement is ignored

The changes above allowed us to demonstrate the value of symbolic testing. `isPrime` takes a number `x`, which we pass in symbolically. We initialised two symbolic variables, `n1` and `n2`. `n1` is assumed to be an even number greater than 2, while `n2` is assumed to be an odd number greater than 3, but also divisible by 3. In our test, we use `isPrimeWith` as an "oracle" to indicate correctness of behaviour.

We first test the behaviour using symbolic variable `n1`. As `n1` is assumed to be even, it will always pass the divisibility check in line 13, hence returning **`false`** (value of `prime`) in line 14. This passes the assertion in line 35. However, in our second symbolic check with symbolic variable `n2`, we assumed `n2` to be an odd number greater than 3, but divisible by 3. Concretely, `n2` can be numbers $9, 15, 21, 27$, *etc.* In this case, `isPrime` will pass these numbers as "prime" because of the function-level hoisting behaviour explained in §4.1. Our symbolic engine will find a counter-model

for the assertion in line 39 `r13 = r23` when `n2 = 9`. We have managed to detect a bug in the program without iteratively passing a range of concrete values to the program in concrete tests.

4.2.4 Modifying properties of the `arguments` object

In §3.6, we explained the mechanics of the `arguments` object in functions. An often confused mechanic is the two-way binding between formal parameters of a function and its run-time `arguments` object in non-strict mode. It is further complicated by the different deletable semantics of formal parameters and mapped properties of the `arguments` object. We will illustrate these behaviours with a symbolic test shown in Figure 4.3.

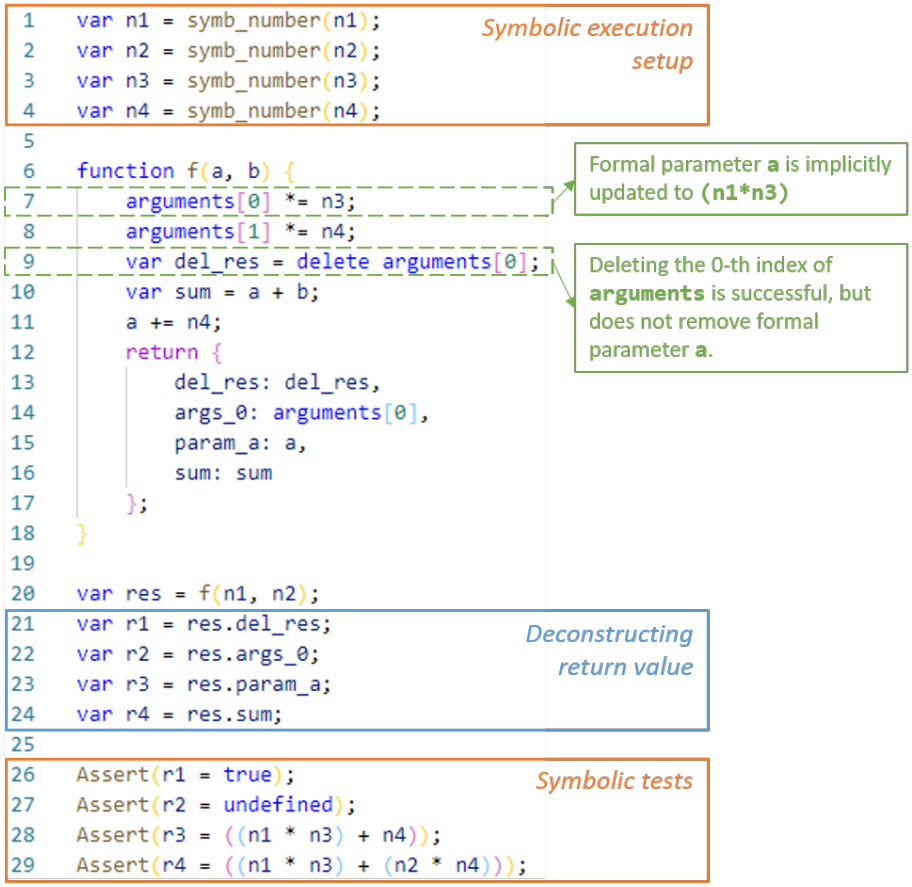


Figure 4.3: Symbolic test for `arguments` object

We define a function `f` that has two formal parameters `a` and `b`. This creates two mapped properties in the `arguments` object at index `"0"` and `"1"` respectively. If we were to modify these mapped properties directly, such as in lines 7 and 8, the corresponding formal parameters will also be updated. This behaviour is tested when we check for the sum `a + b` in line 29 `Assert(r1 = ((n1 * n3) + (n2 * n4)))`. Formal parameters cannot be deleted, but their corresponding mapped properties in the `arguments` object can. We test this behaviour by deleting the `"0"`-th property in line 9, which succeeds (returns `true`). Deletion of the property simply breaks the two-way binding between `arguments[0]` and `a`, but the formal parameter `a` retains the value (`n1 * n3`) before the deletion. We can continue to modify the formal parameter `a` directly in line 11 even if its mapped property has been deleted.

For clarity, we summarise the behaviour checks in the symbolic test below:

1. `Assert(r1 = true)` checks that the deletion of `arguments[0]` in line 9 is successful.
2. `Assert(r2 = undefined)` checks that the deleted property is indeed `undefined`.
3. `Assert(r3 = ((n1 * n3) + n4))` checks that we can continue to modify the formal parameter `a` even after its mapped property in the `arguments` object has been deleted.

4. `Assert(r4 = ((n1 * n3) + (n2 * n4)))` checks that both formal parameters are changed due to the two-way mapping when assignments are made in lines 7 and 8.

It should be noted that there exists test cases in Test262 that tests some of these behaviour individually. For instance, `language/statements/function/S13_A13_T2.js` tests for the deletion behaviour of mapped properties in the `arguments` object. We can view the symbolic test presented above as a generalisation of these tests into symbolic variants.

4.2.5 Indirect, non-strict `eval` call

One of the most confusing elements of JavaScript is the `eval` function. `eval` calls can be executed in either strict or non-strict mode, and can be called directly or indirectly. By simple combinatorics, this should result in a total of 4 different combinations. However, direct `eval` calls can also implicitly inherit the mode of execution while indirect `eval` calls do not. As such, we have a total of **5 combinations** of execution (previously shown in Figure 2.3). In addition, the context with which `eval` runs in depends on whether the `eval` call is a `direct` or `indirect` call.

We will illustrate the two most confusing aspects of `eval`: (1) non-strict execution in indirect `eval`, and (2) inheritance of the global execution environment in indirect `eval`.

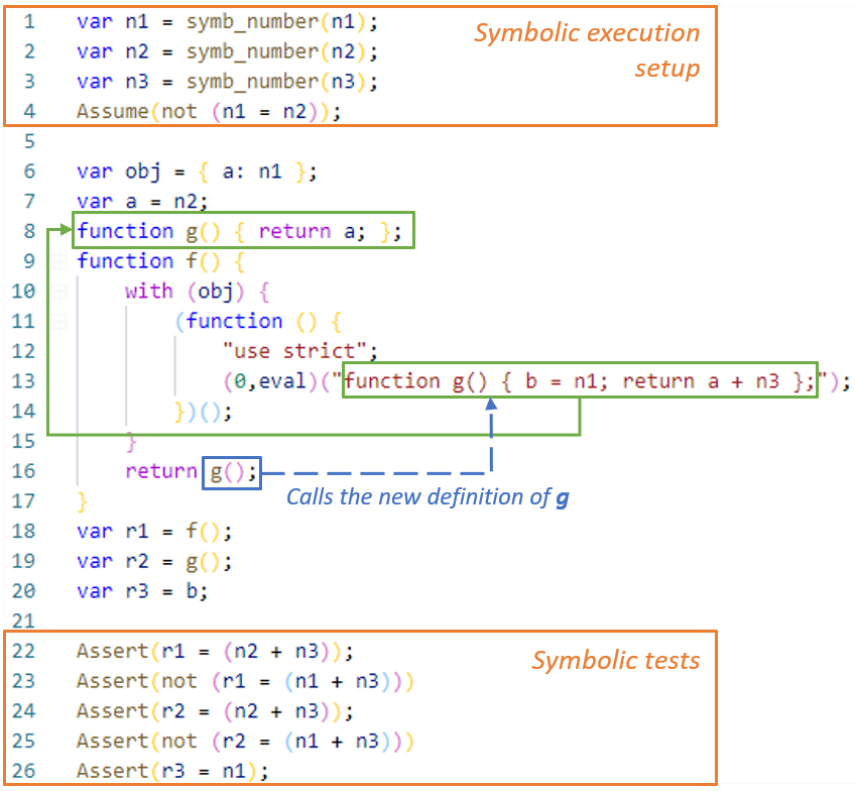


Figure 4.4: Dynamic redefinition with indirect non-strict `eval`

In Figure 4.4, we have two functions declared in the global level: `f` and `g`. Within `f`, we extend the scope chain using the `with` statement and `obj` as the binding object. We make an immediate function call on an anonymous function expression in `strict mode`. In line 13, an indirect `eval` call is made. This causes a **redefinition** of the function `g` in the global scope. Additionally, this new function `g` is compiled in non-strict mode due to the indirect call. We further test the non-strict behaviour of the new function `g` by assigning to a previously undeclared variable `b`, which will create a new global variable in non-strict mode.

For clarity, we summarise the symbolic tests given in this example below:

1. `Assert(r1 = (n2 + n3))` and `Assert(not (r1 = (n1 + n3)))` checks that the call to function `g` in line 16 is the dynamically defined function `g` (in line 13).

2. `Assert(r2 = (n2 + n3))` and `Assert(not (r2 = (n1 + n3)))` checks that the global function `g` has indeed been redefined, and the correct assertions preceding these is not due to hoisting of `g` to the scope of `f`.

3. `Assert(r3 = n1)` checks that the dynamically defined function `g` is indeed run in non-strict mode and the reference on an undeclared variable `b` creates a new global variable with the same name.

4.2.6 Immutable function identifiers

An often overlooked semantic in ES5 is the immutability of function identifiers for named function expressions. We explained its behaviour in §3.2.4.2 while introducing the notion of immutability in declarative environment records. In this section, we present a symbolic test that checks for this immutability.

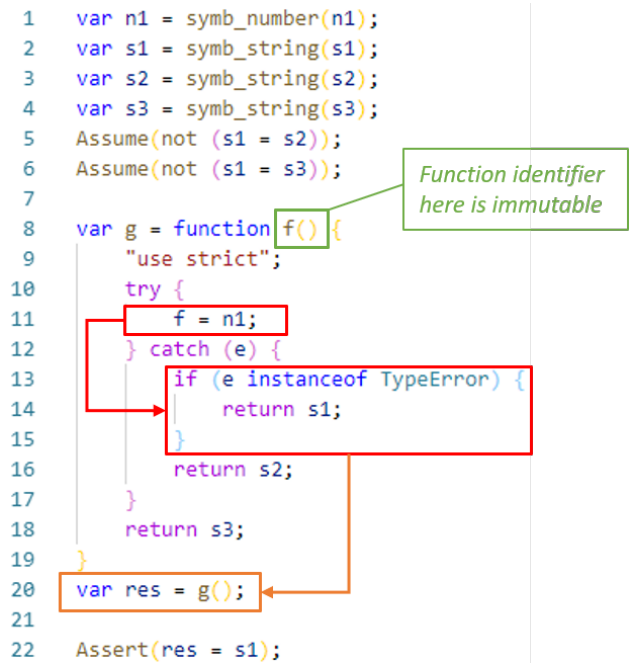


Figure 4.5: Function identifier is immutable in named function expressions

With reference to Figure 4.5, we attempt to change the definition of `f` (in line 11) within the closure of its function. As `f` is an immutable binding, this mutation will throw a `TypeError` exception in strict mode. The test expects this specific type of error to be thrown and passes the test.

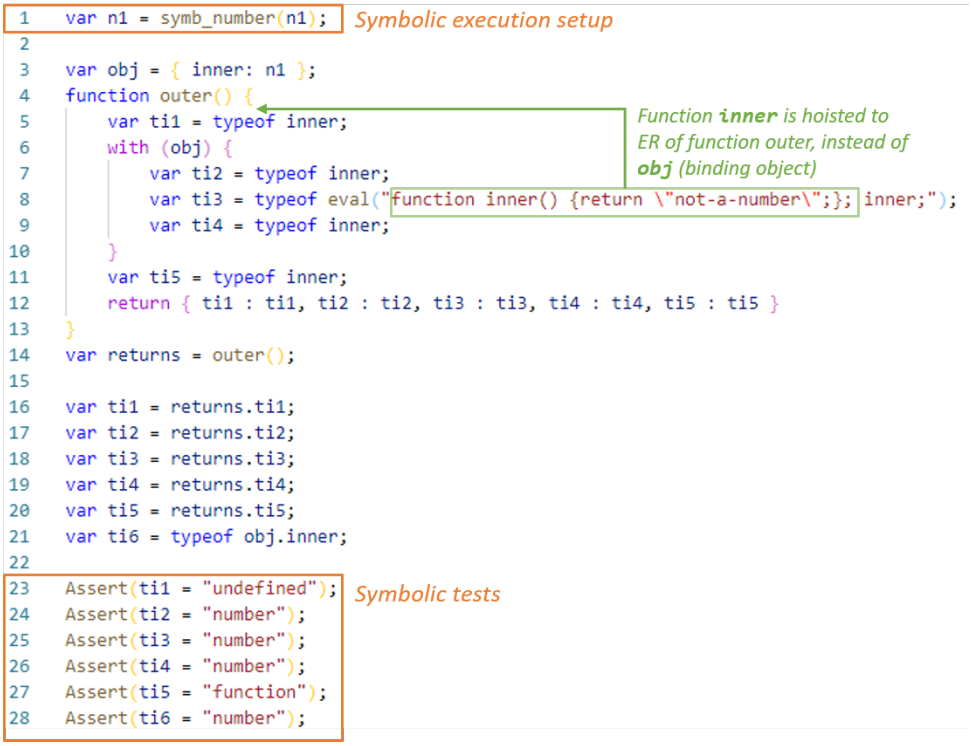
We intentionally run the function `f` in strict mode as a prelude to a later discussion about the mutation behaviour in previous versions of JaVerT. This discussion (§5.1.4) will highlight a bug previously not caught by the Test262 test suite, indicating that there may be a missing behaviour check in the test suite.

4.2.7 Dynamic hoisting

In this section, we present yet another dynamic hoisting example. Different from the previous symbolic tests regarding hoisting, this is dynamic hoisting using direct `eval` calls in non-strict mode. The main motivation for this test case is a real-world implementation bug that we discovered in a modern browser - Microsoft Edge - which we will discuss in depth in §5.3.1.

With reference to Figure 4.6, the program starts with only one declared function `outer`. A new function `inner` is dynamically declared in line 8 with the direct `eval` call. We intentionally wrap this dynamic function creation within a `with` statement (in line 6) that takes `obj` as its binding object. Note that `obj` also has a property named `inner`, that is initialised to a symbolic number

`n1`. As function `inner` is dynamically hoisted to the scope of function `outer`, all type checks on `inner` within the `with` statement (lines 7 and 9) should return `"number"`; `obj.inner` *shadows* the function `inner` present in the scope of `outer`. We can verify that function `inner` does exist in scope of `outer` with the type check in line 11. Additionally, we also verify the property `obj.inner` has not be modified with the type check in line 21.




```
1   var n1 = symb_number(n1);
2
3   var obj = { inner: n1 };
4   function outer() {
5       var ti1 = typeof inner;
6       with (obj) {
7           var ti2 = typeof inner;
8           var ti3 = typeof eval("function inner() {return \"not-a-number\";}; inner;");
9           var ti4 = typeof inner;
10      }
11      var ti5 = typeof inner;
12      return { ti1 : ti1, ti2 : ti2, ti3 : ti3, ti4 : ti4, ti5 : ti5 }
13  }
14  var returns = outer();
15
16  var ti1 = returns.ti1;
17  var ti2 = returns.ti2;
18  var ti3 = returns.ti3;
19  var ti4 = returns.ti4;
20  var ti5 = returns.ti5;
21  var ti6 = typeof obj.inner;
22
23  Assert(ti1 = "undefined");
24  Assert(ti2 = "number");
25  Assert(ti3 = "number");
26  Assert(ti4 = "number");
27  Assert(ti5 = "function");
28  Assert(ti6 = "number");
```

Figure 4.6: Dynamic hoisting behaviour

For clarity, we summarise the assertions in this example below:

1. `Assert(ti1 = "undefined")` checks that `inner` was not hoisted prior to the execution of `eval` in line 8.
2. `Assert(ti2 = "number")`, `Assert(ti3 = "number")` and `Assert(ti4 = "number")` checks that the dynamic hoisting is beyond the scope of the `with` statement. As such, all references to `inner`, include the reference in the last statement of the `eval` code, point to the property `obj.inner`.
3. `Assert(ti5 = "function")` checks that function `inner` is indeed hoisted to the scope of function `outer`.
4. `Assert(ti6 = "number")` does a final check to ensure `obj.inner` has not been modified.

4.2.8 Summary of symbolic examples

In this chapter, we gave a total of six symbolic JS programs written in the style of Test262 test cases. By using the symbolic execution engine of JaVerT, we are able to examine these tests in detail to reveal the complexities of the language. An interesting observation is that four of these six symbolic tests involve some form of hoisting (static or dynamic). Given the confusing nature of hoisting, intermixed with complex dynamic options (such as the example in §4.2.5), we strongly believe that whole-program symbolic testing provides stronger guarantees to implementation correctness than concrete testing.

Additionally, we identified two potential edge cases (§4.2.6 and §4.2.7) not covered in Test262, that could lead to bugs in implementation. These will be covered in greater detail in Chapter 5.

Chapter 5

Evaluation

The primary objective of this project is to extend JaVerT to support ES5 non-strict. We evaluate our progress towards that goal in two aspects:

1. Implementation correctness of the JS-2-JSIL compiler (§5.1).
2. Usefulness of the toolchain in analysing ES5 programs (§5.2).

Of the two evaluation indicators, our priority is on the implementation correctness of the compiler, which is where the majority of our work resides. An incorrect compiler cannot serve as a basis for formal verification and future extensions. As such, we test our compiler thoroughly against ECMAScript's official test suite, Test262. We believe that this work has fully achieved its goal when it comes to implementing the ES5 language features and that it represents a significant step towards a complete implementation of the entire ES5 standard.

We organised our evaluation as follows:

- We start by evaluating the implementation correctness of our compiler in §5.1. Using Test262 as the benchmark, we provide the test coverage of our compiler, along with the explanations of the failed and aborted test cases.
- Next, we evaluate the usefulness of JaVerT in analysing non-strict JS programs in §5.2. We will focus on the use of the symbolic execution engine in JaVerT to perform whole-program symbolic analysis, and its current limitations.
- We then further our evaluation by investigating real-world implementations of ES5+ in §5.3, specifically covering the run-time behaviour of Microsoft Edge and NodeJS. These case studies shed light on both the complexity of the language and subtle implementation decisions that have huge impact on the run-time behaviour of JS programs.
- As our source of reference throughout the project, we will also evaluate the clarity of the ES5 specification in §5.4. In particular, we will highlight the difficulties we faced in this project when trying to understand it.
- Finally, we discuss the main limitations of our work in §5.5, making references to the earlier evaluations in §5.1 and §5.2.

5.1 Implementation correctness

Our benchmark for implementation correctness is the official ECMAScript's test suite, Test262. Test262 has 11,062 ES5 test cases which provide extensive coverage across most of ES5 features. We target the latest standard of Test262 and filter for tests that have "es5id" within the test descriptions, meaning that they are intended/originate from the previous versions of the test suite for ES5. However, there are some mislabelled tests where the test behaviour has been updated with a later standard (such as ES6), but the tests still retain their "es5id" labels. In these cases,

since the goal of JaVerT is to move on to later versions of the standard, and given that all browsers already implement the behaviours of the newer standards, we also implement the behaviour of the newer standards. For clarity, when we qualify a standard (for example, ES5) with the '+' symbol, we meant "(ES5) and later standards".

Due to complexities of the language, there are multiple ways to group these tests into logical categories. In this section, we will evaluate the correctness of our compiler across the following categories:

1. **Mode of execution**
   (a) **Non-strict only tests.** These involve non-strict features of the language, such as the `with` statement.
   (b) **Strict only tests.** These tests check for specific strict mode behaviour, such as strict mode restrictions and semantic differences from non-strict mode.
2. **Feature type**
   (a) **Language tests.** These tests features of the JavaScript language such as expressions and statements. They cover most of the ES5 specification except Section 15 (Standard Built-in ECMASCript Objects).
   (b) **Built-in tests.** These tests the standard built-in objects (Section 15) assumed to be present in every JavaScript program. They are not tied to any language feature, but do depend on the correct implementation of the language features.

We present the overview of our compiler's testing results in Table 5.1 and Table 5.2 for tests in "mode of execution" and "feature type" respectively. For clarity, we define "applicable" test cases as tests which use ECMAScript built-in objects with corresponding JaVerT implementations. For example, this excludes tests for regular expressions or tests for other features of the language which use regular expressions, as regular expressions have not yet been implemented in JaVerT.

Category	Passing	Failing	Aborting	Total	% Passing
Strict only tests	296	0	0	296	100.00%
Non-strict only tests	715	0	4	719	99.30%
Applicable strict only tests	296	0	0	296	100.00%
Applicable non-strict only tests	715	0	0	715	100.00%

Table 5.1: Test results categorised by mode of execution

Category	Passing	Failing	Aborting	Total	% Passing
Language tests	3152	8	42	3202	98.44%
Built-in tests	6652	3	1205	7860	84.63%
All ES5+ tests	**9804**	**11**	**1247**	**11062**	**88.63%**
Applicable language tests	3152	8	0	3160	99.75%
Applicable built-in tests	6652	3	0	6655	99.95%
All applicable tests	**9804**	**11**	**0**	**9815**	**99.89%**

Table 5.2: Test results categorised by feature type

Within the scope of the project, we have achieved tremendous progress towards supporting non-strict features of ES5 in JaVerT. In particular, we achieved 100% coverage for all applicable

strict and non-strict only tests, and 99.89% coverage for all applicable tests. Of the 11 applicable test cases that failed, 8 of them are due to limitations of our parser in handling Unicode strings within the source text. The remaining 3 applicable failing tests are related to the changes in ES6+ syntax and semantics.

In the following sections, we will analyse the test results from "language tests" and "built-in tests". In our analysis, we examine why these tests fail or abort, and their impact on the correctness of our compiler. We skip the categories for "modes of execution", as they are contained within the "feature type" tests. The interested readers can find the full tests breakdown, grouped by feature type, in Appendix A.

5.1.1 Language tests

There are 3202 language tests spanning over 84 language features. The summary of our compiler's language test result is shown in Table 5.3. We will examine the aborted tests first (§5.1.1.1), then the failed tests (§5.1.1.2). Finally, we summarise our contribution in language tests in §5.1.1.3.

Category	Breakdown	ES5+	Applicable ES5+	Remarks
Language	Passing	3152	3152	
	Failing	8	8	ES5+ failures (5) due to parser not parsing Unicode characters correctly.
				ES6+ failures (2) due to change in change in ***if*** statement behaviour.
				ES6+ failure (1) due to change in grammar for function declarations.
	Aborting	42	0	ES5+ aborts (42) due to not implemented built-ins.
	Total	**3202**	**3160**	
	% Passing	**98.44%**	**99.75%**	

Table 5.3: Language test overview

For interested readers, the full breakdown grouped by language feature is given in Table B.1. The list of failed and aborted tests is given in Table B.2.

5.1.1.1 Aborted tests

All 42 aborted tests are due to ECMAScript built-in objects that we have not yet implemented in JaVerT. Aborts of this type are not directly related to any language feature and they are not indicative of incorrectness in our compiler's language features. Hence, we immediately exclude these tests from our applicable test list.

5.1.1.2 Failed tests

We now investigate the failing language tests summarised in Table 5.4. These 8 failed tests can be classified into 3 groups:

1. Limitations of the parser in parsing Unicode strings. There are 5 such cases, labelled 1 through 5 in Table 5.4.
2. Semantic changes between ES5 and ES6+. They correspond to the last 3 cases in Table 5.4, with the first 2 being due to the change in the semantics of the `if` statement, and the last one being due to a change in permitted positioning of functions in the code.

The limitations of the parser in parsing Unicode strings stem from the difference in representation of these strings. We use OCaml to parse and compile JS programs, which transforms JS strings into OCaml strings. Our OCaml representation of Unicode strings are in UTF-8, which are interpreted

No.	Name	Type	Status	Reason
1	line-terminators/7.3-15.js	line-terminators	Failing	Parser (unicode)
2	line-terminators/7.3-5.js	line-terminators	Failing	Parser (unicode)
3	line-terminators/7.3-6.js	line-terminators	Failing	Parser (unicode)
4	line-terminators/invalid-string-cr.js	line-terminators	Failing	Parser (unicode)
5	source-text/6.1.js	source-text	Failing	Parser (unicode)
6	statements/for/head-init-expr-check-empty-inc-empty-completion.js	for-statement	Failing	"if" completion updates empty return to undefined (ES6+)
7	statements/for/head-init-var-check-empty-inc-empty-completion.js	for-statement	Failing	"if" completion updates empty return to undefined (ES6+)
8	statements/break/S12.8_A3.js	break	Failing	Function declaration in statement position (ES6+)

Table 5.4: Language test failures

at the byte level. In constrast, ECMAScript Unicode strings are represented in UTF-16. As such, a single Unicode character, such as U+104A0, could be represented as 4 bytes in UTF-8 but takes 2 UTF-16 code units. This Unicode character is precisely the reason why the fifth test case failed.

Additionally, there are several semantic differences in how ES5 treats certain Unicode characters with special meaning (white space, carriage returns, terminating string) compared to OCaml Unicode characters. These are not recognised by our parser, hence the failures of tests 1 to 4.

Failed tests due to limitation of the parser are still relevant to ES5 since they are part of the lexical grammar. However, we place a lower priority on these, as they are not tied to any higher-order language feature.

The next two failed test cases deal with the change in semantics of the `if` statement in later standards. In ES5, the evaluation of the `if` statement returns the result of evaluating the appropriate branch statement (determined by the boolean expression). However, in ES6+, an additional check is placed on the return value of each statement. If the return value is `empty`, it gets updated to `undefined` before the statement completes its evaluation. We illustrate with the first failed test case in this group in Code Snippet 5.1. The intuition for the second test case is similar.

```
1  // Copyright 2009 the Sputnik authors.  All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3
4  /*---
5  info: |
6      The result of evaluating "for( ExpNoIn;Exp;Exp)" loop is returning
7      (normal, evalValue, empty)
8  es5id: 12.6.3_A9.1
9  description: Using eval
10 ---*/
11
12 var supreme, count;
13 supreme=5;
14
15 var __evaluated =  eval("for(count=0;;) {if (count===supreme)break;else count
       ++; }");
16
17 assert.sameValue(__evaluated, void 0, '#1: __evaluated === 4. Actual:
       __evaluated ==='+ __evaluated);
```

Code Snippet 5.1: Test case: `head-init-expr-check-empty-inc-empty-completion.js`

In this example, line 15 calls `eval` with a `for` loop that does a conditional `break` statement. In ES5, the evaluation would return `empty` into the variable `__evaluated`. However, ES6 would return `undefined` instead. The test case expects the ES6 behaviour, as seen by the `void 0` in the assertion (`void 0` evaluates to `undefined`). Due to time constraints, we were not able to correct this failure in time for the report. The correction requires non-trivial changes in the compilation of the `if` statement.

The last failed test, presented in Code Snippet 5.2, is meant to test the behaviour of breaks

within a labeled loop. However, there exists a function declaration `IN_DO_FUNC` in line 17 which is within the block statement of the outer **`do-while`** loop (lines 12 to 18). In ES5, the grammar does not permit the presence of function declaration in block statements. In fact, function declaration can only occur at the top level of a program or a function declaration as *SourceElements*. Our compiler will catch this function declaration and throw an early error after parsing.

However, in ES6+, function declarations within block statements are allowed. It is part of the rule of *HoistableDeclaration* - a type of statement. The semantics of a function declaration within a block statement is more akin to the forward declaration of a variable whose name is the identifier of the declared function, and is initialised to a function expression whose body is the same as the declared function. In other words, an equivalent rewriting into ES5 would transform line 17 of Code Snippet 5.2 into **`var IN_DO_FUNC = function() {};`** and inserted between lines 12 and 13.

```
1  // Copyright 2009 the Sputnik authors.  All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3
4  /*---
5  info: When "break" is evaluated, (break, empty, empty) is returned
6  es5id: 12.8_A3
7  description: Using "break" without Identifier within labeled loop
8  ---*/
9
10 LABEL_OUT : var x=0, y=0;
11
12 LABEL_DO_LOOP : do {
13     LABEL_IN : x=2;
14     break ;
15     LABEL_IN_2 : var y=2;
16
17     function IN_DO_FUNC(){}
18 } while(0);
19
20 LABEL_ANOTHER_LOOP : do {
21     ;
22 } while(0);
23
24 function OUT_FUNC(){}
25
26 ////////////////////////////////////////////////////////////////////////////
27 //CHECK#1
28 if ((x!==2)&&(y!==0)) {
29     $ERROR('#1: x === 2 and y === 0. Actual:  x ==='+x+' and y ==='+y);
30 }
31 //
32 ////////////////////////////////////////////////////////////////////////////
```

Code Snippet 5.2: Test case: `statements/break/12.8_A3.js`

There are two solutions to this test case:

1. The first is to follow the rewriting intuition we gave above. We could modify the AST generated by the parser such that we reorder all function declarations to the first statement of each block, then perform the equivalent rewriting into a variable declaration initialised with a function expression. The main disadvantage of this approach is that we are effectively changing the structure of the original JS program to avoid dealing with the actual problem. This has no guarantees that it would work in all future versions of ECMAScript.

2. The second solution is to remove the early error and allow function declarations within block statements. We would have to further modify the compiler to correctly traverse the AST and perform the appropriate hoisting. The main complication with this approach is that the semantics of function declarations have also changed in ES6+. Specifically, the scope of the function inherits the *LexicalEnvironment* of the running execution context in ES6, instead of *VariableEnvironment* as in ES5. We have already discussed at length the differences between LexicalEnvironments and VariableEnvironments within execution contexts in §3.5. In short, this change would impact the compilation of function declarations within `with`

statements, where the LexicalEnvironment and VariableEnvironment point to different lexical environments.

Given these solutions and further constraints by time, we chose to push this test case to the future work of extending JaVerT to ES6.

5.1.1.3 Summary of language tests

Overall, we view the language test results to be a complete success. We achieved 99.75% coverage for all applicable language tests and a high coverage of 98.44% across all language tests. While there is room for improvement in the parser and the ES6+ tests, we do know how to fix those failures and were only limited by time in the end.

5.1.2 Built-in tests

There are 7860 built-in tests across 12 types of ECMAScript built-in objects. The table Table 5.5 summarises our compiler's built-in test results. Similar to our discussion of the tests in language tests (§5.1.1), we will first examine the aborted tests (§5.1.2.1), then the failed tests §5.1.2.2. Finally, we summarise our contributions in §5.1.2.3.

Category	Breakdown	ES5+	Applicable ES5+	Remarks
Built-ins	Passing	6652	6652	
	Failing	3	3	ES5+ failures (3) due to Unicode passed into Number constructor.
	Aborting	1205	0	All aborts are due to non-implementation.
	Total	**7860**	**6655**	
	% Passing	**84.63%**	**99.95%**	

Table 5.5: Built-in test overview

Interested readers can refer to Table B.3 for the breakdown by built-in object type.

5.1.2.1 Aborted tests

We have 1205 aborted built-in tests, all of which are due to unimplemented built-in functions in JSIL. They range from partial implementations of the `Global` and `String` objects, to entire missing objects, such as `RegExp` and `JSON`. We determine that these aborted tests are not indicative of incorrectness in our compiler and exclude them from our list of applicable tests. It remains part of the broader goal of JaVerT to fully implement the entire specification, including the missing built-ins.

5.1.2.2 Failed tests

No.	Name	Type	Status	Reason
1	Number/S9.3.1_A2.js	Number	Failing	Parser (unicode)
2	Number/S9.3.1_A3_T1.js	Number	Failing	Parser (unicode)
3	Number/S9.3.1_A3_T2.js	Number	Failing	Parser (unicode)

Table 5.6: Built-in test failures

Of the built-ins that we have implemented, there are only 3 failing tests and they belong to the same type of failure: limitations of the parser in parsing Unicode strings.

Similar to the failing tests due to Unicode parsing in §5.1.1.2, the first group of failing built-in tests face the same constraint when dealing with Unicode strings. In particular, these three test

cases attempt to construct `Number` objects with Unicode strings containing Unicode characters classified as “white-space" in ECMAScript. Our compiler does not distinguish between these characters from any (Unicode) character and passes them unaltered to the run-time construction of a number in OCaml. This calls a string to float conversion function (`Float.of_string`) which fails and we return `NaN` (Not-a-Number) as the constructed number instead.

There are two possible methods to resolve this issue:

1. Pre-process Unicode characters passed to the `Number` constructor at compile-time. This would allow us to replace these Unicode (white-space) characters into equivalent ASCII white-space or simply an empty string (since they have equivalent semantics in `Number` construction). However, we will fail in the dynamic case if the string used for construction cannot be determined at compile time.

2. Parse the Unicode strings at run-time, which will enable us to handle dynamic strings. However, this will add significant complexity in the verification of JS programs, in addition to the adverse impact it would have on the performance of the run-time.

Both solutions are non-trivial to implement and we are constrained by time. As these tests are not critical in the analysis of ES5 (non-strict) behaviour - the focus of this project - we chose to lower the priority of these tests and push them as future improvements.

5.1.2.3 Summary of built-in tests

Overall, we view the built-in test results to be a significant success. Of all the implemented built-ins, we have achieved near 100% coverage, limited only by the parsing of Unicode strings. Echoing the same sentiments from our conclusion of the language tests, we would have liked to fix the test cases that fail due to the parser had we had more time. A significant amount of work remains to implement the entire built-in objects in JaVerT.

5.1.3 Evaluation of the Test262 test suite

While passing the great majority of the applicable Test262 tests gives us confidence in the correctness of our compiler, we should keep in mind that Test262 is not a conformance test suite and that it does not cover all of the possible paths through the JavaScript semantics. This could result in JavaScript implementation having errors that are very difficult to detect. In fact, we have discovered two edge cases that are not covered in the Test262 test suite, which have resulted in bugs in JaVerT and real-world browsers. We will cover these two cases in §5.1.4 and §5.3.1 respectively.

```
1  // Copyright (c) 2012 Ecma International.  All rights reserved.
2  // This code is governed by the BSD license found in the LICENSE file.
3
4  /*---
5  es5id: 10.4.3-1-101-s
6  description: >
7      Strict Mode - checking 'this' (non-strict function passed as arg
8      to String.prototype.replace from strict context)
9  flags: [noStrict]
10 ---*/
11
12 var x = 3;
13
14 function f() {
15     x = this;
16     return "a";
17 }
18
19 assert.sameValue(function() {"use strict"; return "ab".replace("b", f);}(), "aa
       ");
20 assert.sameValue(x, this, 'x');
```

Code Snippet 5.3: Use of built-in `String.replace` in language test

In addition, the test cases in Test262 often include built-ins in their language tests, which introduce unnecessary dependencies. Take the test case `language/function-code/10.4.3-1-101-s.js`, for instance (Code Snippet 5.3).

This is a language test for function-code that checks for the `ThisBinding` value for non-strict functions. However, due to the use of the built-in `String.prototype` function `String.replace`, there is now a dependency on the built-in function to be implemented before this test case can be executed correctly. This, in particular, was a problem for us as we do not yet have the implementation of `String.replace` in JaVerT[1].

A solution to the problem is to re-implement the test case with similar semantics but without the use of the built-in function, shown in Code Snippet 5.4. We replace the call to `String.replace` to a function `callback` that accepts a callback function `cbf`, and executes it within the body. While it does not behaviour exactly the same as `String.replace`, it achieves the same purpose of testing the behaviour of `ThisBinding` in a non-strict callback function within a strict function.

```
1  /*---
2  es5id: 10.4.3-1-101-s (modified)
3  description: >
4      Strict Mode - checking 'this' (non-strict function passed as arg
5      to a function that accepts a callback function)
6  flags: [noStrict]
7  ---*/
8  var x = 3;
9
10 function f() {
11     x = this;
12     return "a";
13 }
14
15 function callfunc(dum, cbf) {
16     if (dum === "b") {
17         return cbf();
18     }
19     return dum;
20 }
21
22 assert.sameValue(function() {"use strict"; return callfunc("b", f);}(), "a");
23 assert.sameValue(x, this, 'x');
```

Code Snippet 5.4: Function code test case without built-ins

On a broader level, this example also highlights the importance of designing test suites. Ideally, unit test cases should be self-contained, testing only the aspect that is required without depending on external constructs. If a unit test fails, it should fail because it does not meet the expected behaviour within the scope of the test, and not because of a dependency that failed. In the situation where *integration* testing is required (two or more modules are involved), it should be clearly labelled and grouped separately from unit test cases.

5.1.4 Bug in JaVerT 2.0 (ES5 Strict)

We discovered a bug in the previous version of JaVerT that is not caught by the Test262 test suite. This occurs due to a mutability bug of named function expressions.

In ES5, named function expressions create an additional declarative ER (before the ER created for the function itself), known as a *closure*. This ER houses only the function identifier for the named function expression and it is an **immutable binding**. In the past implementation of the JS-2-JSIL compiler, we do not have constructs to distinguish between mutables and immutables. Unless specific exceptions are made (such as `eval` and `arguments`), bindings are always implicitly mutable. As such, we can mutate the binding of the function identifier as shown in Code Snippet 5.5. The immutability behaviour is consistent between strict and non-strict mode of ES5. We chose to use a strict mode example to further illustrate the point that this is indeed a bug in the previous version of JaVerT that only runs ES5 strict mode programs.

[1]The implementation of several `String.prototype` functions is part of another project.

In the execution of Code Snippet 5.5, we create a function object from a named function expression of `f`. Within the body of `f` in line 15, we try to modify the binding of the identifier `f` from a function object to the number `10`. In the previous version of JaVerT, this mutation would succeed, and all future references of `f` would be the number `10`. In the example program, it would throw the error `"Expected␣immutable␣f"` as an indication of the incorrect behaviour.

```
1  /**
2   * [commit 52619c50] ES5 "Strict" JS-2-JSIL
3   * Run-time throws the Error: Expected immutable f
4   * This means that the older version of JaVerT does not check for
5   * the immutable binding of named function expressions.
6   *
7   * [commit ecc07138] ES5 JS-2-JSIL
8   * Run-time returns "okay"
9   */
10
11 var g = function f() {
12     "use strict";
13     try {
14         // This mutation succeeds in old version of JaVerT
15         f = 10;
16     } catch (e) {
17         if (e instanceof TypeError) {
18             return "okay";
19         }
20         throw Error("Expected TypeError");
21     }
22     throw Error("Expected immutable f");
23 }
24 g();
```

Code Snippet 5.5: Mutability bug in named function expressions

We fixed the incorrect behaviour in the new version of the compiler with the use of immutables as discussed in §3.2.4.2. The same mutation in line 15 will now correctly throw a `TypeError`.

Given the 100% test coverage of Test262 ES5 strict in the previous version of JaVerT [24], it is likely the mutation behaviour demonstrated above was not part of the test suite. From this, we can clearly see that test coverage is not always the best measure of implementation correctness. On a broader level, if we consider test-driven development in software development, we can only guarantee that the software is only as correct as the test cases it passes.

5.2 Usefulness in analysing ES5 programs

In Chapter 4, we presented several symbolic tests that could serve as the basis of analysing the behaviour of JS programs symbolically. For the use cases given, the whole-program symbolic testing mechanism has worked relatively well. We were able to execute symbolic programs with minimal changes to the symbolic engine after our compiler has been modified to support ES5 non-strict features. This gives us confidence in the long-term sustainability of supporting whole-program symbolic testing in JaVerT as we move progressively towards the full ES5 specification or later standards.

However, there is a major limitation to the run-time of the symbolic engine - symbolic implementation of some of the operators used by JavaScript and, therefore, JSIL. Previously, in our evaluation of the implementation correctness of our compiler, we are less concerned about the non-implementation of built-ins compared to the language feature tests. This is due to the fact that built-ins largely do not affect the correctness of the compiler, but rather limit the variety of programs that could be executed in the run-time. However, in analysis of real-world programs, having a symbolic implementation of language operators that are supported concretely is especially important in providing a realistic symbolic testing environment.

We briefly mentioned one instance of this limitation when we were rewriting the `isPrime` example in §4.2.3. We were forced to replace the more efficient loop condition `i < Math.sqrt(x)` to an inefficient variant `i < x` as we do not have the symbolic implementation of the square root in JSIL.

In this case, we were able to make a simple modification as: (1) the substitution is straight-forward, and (2) we own the code. If we were to extend our analysis to more complex real-world programs, the lack of built-ins would severely impact the feasibility of performing whole-program symbolic testing. As such, more work is required in this area before JaVerT can become a top-tier symbolic analysis tool.

5.3 Real-world implementations of ES5(+)

In this section, we will discuss two real-world implementations of ES5. We start by investigating a peculiar bug in Microsoft Edge (§5.3.1). This case is particularly interesting as it highlights the complexity of the language and how a small deviation could have a large impact on JS programs as a whole. We also discuss an interesting design choice of NodeJS where JS programs may exhibit slightly different semantics due to the packaging of modules (§5.3.2).

5.3.1 Bug in Microsoft Edge

In §3.2.1, we examined the mechanics of state management in ES5. Specifically, we mentioned the subtle differences between LexicalEnvironments and VariableEnvironments, and how this difference could affect hoisting of declarations and scope resolution. During our implementation of function hoisting, we investigated its implementation behaviour in modern browsers which revealed an implementation bug in Microsoft Edge. Specifically, this bug occurs during the case of **dynamic** function declaration hoisting through the combination of `with` and direct `eval`.

To assist us in understanding how the bug occurs, we provide the following example in Code Snippet 5.6 with the expected print results from `console.log` in the comments. We first explain the expected behaviour of the example, then compare the correct behaviour with the incorrect behaviour displayed in Microsoft Edge. As a point of comparison, both the behaviour of Google Chrome (version 74.0.3729.169) and Microsoft Edge (version 17.17134) are shown in Figure 5.1.

```
1  var obj = {
2      inner: 1
3  };
4  function outer() {
5      console.log(typeof inner); // undefined
6      with (obj) {
7          console.log(typeof inner); // number
8          console.log(typeof eval("function inner() {}; inner;")); // number
9          console.log(typeof inner); // number
10     }
11     console.log(typeof inner); // function
12 }
13 outer();
14 console.log(typeof obj.inner); // number
```

Code Snippet 5.6: Dynamic function hoisting in ES5

In Code Snippet 5.6, two function declarations are made: (1) A statically declared function `outer`. Statically declared functions are hoisted to the parent ER at compile time, and (2) a dynamically declared function `inner` that is contained in the string passed to the direct `eval` call. All function declarations (static or dynamic) are hoisted to the VariableEnvironment of the current execution context. In the static case, analysis is simple as hoisting occurs at compile time. For the function `outer`, the declaration occurs in global code, hence hoisted to the global environment record. However, complication arises in the dynamic case.

Recall that **direct non-strict** `eval` calls inherits the parent execution context as its running execution context. During the execution of `outer` in line 6, the LexicalEnvironment has been modified by the `with` statement, adding a new environment record to the scope chain. However, the VariableEnvironment **remains unchanged**. The `eval` call inherits the modified LexicalEnvironment and the unmodified VariableEnvironment, hence hoists the function `inner` to the environment record of the function `outer`. This implies that there exists two distinct bindings for the string identifier `"inner"` in the scope chain: one in the object environment record for the `with (obj)`

statement and the other in the environment record for `outer`. In other words, the `inner` property of object `obj` *shadows* the function `inner` in the execution of the `eval` statement. As such, the return value of the `eval` call[2] is the property `inner` of `obj` (type is `"number"`). By the same logic, `inner` in line 9 refers to the same property in `obj`.

Exiting the `with` statement in line 11 restores the LexcialEnvironment to the lexical environment of the function `outer`. The identifier `inner` now references the dynamically declared function `inner` (type is now `"function"`). Finally, we make a sanity check in line 14 to ensure that the `outer.inner` property has not been modified (type is `"number"`).

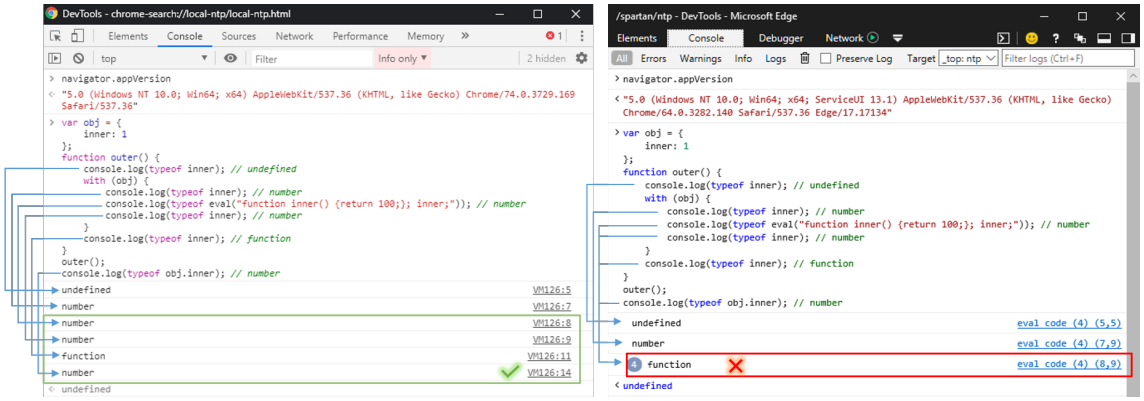


Figure 5.1: Correct function hoisting in Chrome (left) vs Incorrect function hoisting in Edge (right)

The behaviour described above is reflected in the execution behaviour in Chrome. In Edge however, the bug occurs during the dynamic hoisting of the `inner` function within the `eval` code. Instead of hoisting the function to the inherited VariableEnvironment (scope of function `outer`), Edge hoists it to **both** the inherited VariableEnvironment and LexicalEnvironment (scope of the `with` statement). This causes the `obj.inner` property to be overwritten by the function `inner`, and a new function `inner` to be added to scope of `outer`. As such, all subsequent calls to `typeof inner` now returns `"function"` (4 times in total).

This behaviour in Edge is especially bizarre as the function is hoisted to two locations: (1) in the object of the `with` statement, and (2) in the outer function `outer`. If the bug is caused by a lack of distinction between LexicalEnvironment and VariableEnvironment, then the function should be wrongly hoisted to the object in the `with` statement only. However, the additional hoisting to the outer function seems to indicate that Edge does recognise the difference between the lexical environments. We hypothesize that it could be due to a misinterpretation of later standards (ES6+) where the dynamic function declaration is treated additionally as a static variable declaration initialised with an anonymous function expression within the `with`. Edge could be performing both the static hoisting and dynamic hoisting in the same instance despite it clearly being a dynamic-only hoisting case.

5.3.2 NodeJS script execution behaviour

NodeJS is a JS runtime built on the V8 JavaScript engine [7]. As NodeJS uses the same JavaScript engine as Google Chrome, the execution of JS programs in NodeJS usually display the same behaviour as Google Chrome. However, in our early testing of dynamic hoisting using indirect and non-strict `eval`, we discovered a case where the execution of the exact same program produces different results in NodeJS and Google Chrome.

We present this case in Code Snippet 5.7. Notice that this is actually a concrete case of the symbolic test that we gave in §4.2.5. Due to its similarly with the symbolic test, we will skip most of the description of the program and focus only on the parts that are significant to our discussion.

The execution of Code Snippet 5.7 by ES5 specification should return the following order of `console.log` prints: `"2"`, `"12"`, `"12"`, `"3"`. If we were to run this program in Google Chrome, we would indeed get these results. However, when we execute this program in NodeJS, we get a

[2]Recall that `eval` calls return the last computed statement that is not `empty`.

different output: `"2"`, `"2"`, `"2"`, `"not found"`. It appears as if the function `g` was never redefined after the execution of the `eval` call in line 9. Hence, the variable `b` was never implicitly declared, and we get `"not found"` in the last `console.log` as well. This seem to indicate that NodeJS is incorrectly hoisting the function `g` in line 9.

```
1  var obj = { a: 1 };
2  var a = 2;
3  function g() { return a; }
4  function f() {
5      console.log(g()); // "2"
6      with (obj) {
7          (function () {
8              "use strict";
9              (0,eval)("function g() { b = 3; return a + 10 };");
10         })();
11     }
12     return g();
13 }
14 console.log(f()); // "12"
15 console.log(g()); // "12"
16 try {
17     console.log(b); // "3"
18 } catch (e) {
19     console.log("not found");
20 }
21
22 // Chrome returns: 2, 12, 12, 3
23 // Node returns: 2, 2, 2, not found
24 // Edge returns: 2, 2, 2, not found
```

Code Snippet 5.7: Dynamic hoisting in indirect non-strict `eval`

However, what actually happened is that NodeJS encapsulates every JS script it runs with a wrapper function. This is part of a design decision made by NodeJS to *modularise* the loading of JS scripts in isolated containers. We can more accurately represent Code Snippet 5.7 as the program in Code Snippet 5.8. As such, the **`var`** declarations in a JS script run by the NodeJS run-time are actually hoisted to function-level variables. Taking into account the wrapper, the behaviour of the original program in NodeJS is actually an accurate implementation of the ES5 standard. In fact, if we were to modify the code in Code Snippet 5.8 such that line 4 and 6 become *implicit* globals (`a = 2` and `g = function(){return a;}`), we would get the "correct" behaviour given in Chrome.

```
1  (function (exports, require, module, __filename, __dirname) {
2      // Locally scoped variable declarations
3      var obj = { a: 1 };
4      var a = 2;
5      // Locally scoped function g
6      function g() { return a; }
7      function f() {
8          console.log(g()); // "2"
9          with (obj) {
10             (function () {
11                 "use strict";
12                 // `g' hoisted to global level
13                 (0,eval)("function g() { b = 3; return a + 10 };");
14             })();
15         }
16         return g();
17     }
18     console.log(f()); // "2"
19     // References g in the local function scope
20     console.log(g()); // "2"
21     try {
22         console.log(b);
23     } catch (e) {
24         console.log("not found");
25     }
26 });
```

Code Snippet 5.8: Module wrapping in NodeJS

As an additional point of interest, executing Code Snippet 5.7 in Microsoft Edge returns the same result seen in NodeJS. This is another bizarre behaviour that Edge does and we do not have a concrete explanation of what actually happened in this case.

5.4 Evaluation of the ES5 specification

One of the main challenges we faced in this project is the clarity of the ES5 specification. There are two main issues we faced when reading the specification: (1) extensive redirections, and (2) questionable names.

5.4.1 Extensive redirections

We spent a considerable amount of time going through the extensive redirections that take us from one section to many others. For example, the compilation of a simple variable reference `v` involves over 10 sections of the specification. We present this traversal in full for the reader's enjoyment.

The compilation starts in Section 11.1.2 (Identifier Reference), which has two sentences, the first of which redirects us to Section 10.3.1. Section 10.3.1 (Identifier Resolution) contain a 3 line algorithm, with the third line calling the abstract lexical environment operation `GetIdentifierReference` defined in Section 10.2.2.1. Within Section 10.2.2.1, we have additional redirections to the `HasBinding` concrete operations of both types of environment records (Section 10.2.1.1.1 and Section 10.2.1.2.1). The object ER's `HasBinding` concrete method makes additional calls to `HasProperty` (Section 8.12.6), `GetProperty` (Section 8.12.2) and `GetOwnProperty` (Section 8.12.1) internal methods of the binding object. We have already covered 8 sections thus far. If we also include string object's own `GetOwnProperty` internal method (Section 10.5.5.2), which is a special case of `GetOwnProperty`, we would need to add another 3 more sections to the list.

5.4.2 Questionable naming

Aside from redirections, the choice of names is also questionable. In particular, the use of *lexical environment* (two words with a space) and *LexicalEnvironment* (joined as one word) was a major source of confusion. Despite how similar their names are, they are in no way referring to the same construct in ES5. A *lexical environemnt* is a wrapper of an environment record, and can be seen as a node in a linked-list of lexical environments. A *LexicalEnvironment* is a *pointer* to a lexical environment. One is a specification construct, one is a pointer to another. However, if one was not careful and misinterpret *lexical environment* and *LexicalEnvironment* as the same construct, several parts of the specification may become semantically different.

For instance, the definition of the *VariableEnvironment* in an execution context is also pointer to a *lexical environment*. If we were to assume that *lexical environment* and *LexicalEnvironment* are the same construct, this effectively means that the *VariableEnvironment* is now the same as *LexicalEnvironment*. This could result in incorrect hoisting of dynamically declared functions and variables when the two pointers should point to different lexical environments, such as in a `with` statement. This could be a possible explanation of the incorrect behaviour in Microsoft Edge (§5.3.1) where the dynamically function is hoisted to the binding object of the `with` statement instead of the outer scope.

5.5 Known limitations

Despite the success we have in implementation correctness of our compiler, there are several limitations of our current work. In this section, we will discuss these limitations.

Unicode parsing The primary reason for the failed tests in our list of applicable Test262 tests is the parsing of Unicode strings. We discussed the impact of incorrect interpretation of Unicode strings with significant depth when we examined the failed test cases in both §5.1.1.2 and §5.1.2.2.

In short, we require a parser that can correctly recognise certain Unicode characters as special characters (white-space, carriage return, termination, etc.) as per the ES5 lexical grammar.

Non-implementation of ECMAScript built-in objects We have a significant lack of implementation of ECMAScript built-in objects. This limitation is seen in both the concrete and symbolic execution engines of JaVerT. In the concrete case, we have over 16% of all test cases abort due to non-implementation of the built-in objects. While the built-in objects is not critical when analysing JS programs, it does limit the variety of JS programs JaVerT can analyse. This is particularly limiting in the symbolic case, as there is an additional requirement for equivalent implementations with symbolic constructs.

Formal verification of ES5 programs The main limitation of this project lies in the lack of formal verification for ES5 programs. In fact, due to the numerous changes to the compiler, run-time and scope resolution, we have effectively disabled the ability of JaVerT to formally verify ES5 programs. The changes to the scope resolution would be a significant problem to overcome as we migrated from lexical scoping to dynamic scoping when we introduced the `with` statement. In addition, we have introduced and made significant modifications to over 60 internal JSIL functions (Appendix A). We will have to write new specifications or rewrite existing ones to fit these JSIL procedures. As formal verification provides the strongest guarantees in program correctness, the lack of it in the current state of JaVerT does limit its usefulness.

5.6 Lessons learnt

5.6.1 Compilers and language specifications

Prior to this project, my experience in compilers and language specifications was limited to small toy languages taught during my undergraduate studies.

This project is my first experience working with a large and contemporary language specification. Due to the algorithmic nature of the ES5 specification, it was certainly tempting to "code blindly" to the standard without understanding the semantics. In fact, I was able to complete the implementation of all lexical environment operations (§3.2.2) without understanding the significance of LexicalEnvironments and VariableEnvironments within execution contexts. However, this lack of understanding of the underlying semantics proved to be detrimental when I met edge case behaviours such as the bug in Microsoft Edge (§5.3.1). In fact, it is this bug that also revealed my prior misunderstanding of the hoisting behaviour when I hoisted all declarations to the last environment record in the scope chain. As Test262 does not have test cases that reveal this edge case behaviour, I was able to successfully pass all scope related tests despite the (now apparent) wrong implementation.

Learning from this mistake, I would always ensure I understand the semantics of the specifications before implementing new language features to the compiler. Additionally, revisiting previously implemented language features (in older versions of JaVerT) such as the creation of function objects in function expressions, also enabled me to identify the mutability bug in JaVerT (§5.1.4) that was not discovered until this project.

5.6.2 Working with JaVerT 2.0

JaVerT is a relatively mature toolchain with a significantly large code base. Having worked with it over the past few months, I have gathered some of my thoughts and lessons learnt below.

5.6.2.1 Design decisions

Early decision decisions have long-term impact. In my opinion, JaVerT generally has well designed components within the compiler and run-time. Its modularity was also a major benefit when

implementing new features. However, the decision to deviate from the ES5 specification for simplifications in strict mode was a challenging problem to resolve in this project.

Compiler design. The compiler is split into several modules and my work was generally focused on only a few files. In the area of state management, the decision to create new translation contexts in keep track of updated compile-time states (such as the appropriate scope chain reference) was an excellent choice. It helped simplify the complex control flow logic in the run-time that could arbitrary return or jump to different parts of the JS program. This has been particularly useful during the compilation of the `with` statement where the scope chain must be restored even in the event of abrupt completions (such as breaks, continues or returns). Since the compiler keeps track of the appropriate scope chain at each lexical level of the code, we basically get scope restoration for "free"" without having to manually update the translation context when control flow exits the `with` statement.

Early simplifications. The main challenges I faced when extending JaVerT was in the modification of existing behaviour and breaking previous assumptions. Prior to this project, JaVerT was made to target the strict mode of ES5. Due to many of the simplified semantics in strict mode, the compiler was developed with many simplified assumptions in mind. These simplifications has brought about many benefit such as ease of control flow and optimisations to the specification algorithms. For instance, in ES5 strict, we could resolve references lexically by building a closure clarification table. With the benefit of resolving references lexically, there was no need to implement all the lexical environment and environment record operations that are capable of dynamic resolution. In fact, the redirection mess mentioned in §5.4 (over 10 sections worth of specifications) for variable references was entirely avoided and replaced with a simple list element retrieval at run-time. This was not only easier to reason formally about, but also a lot more efficient than dynamic referencing.

However, simplifications can create technical debt at times. The non-implementation of the dynamic referencing operations results in a significantly longer implementation time when we move towards ES5 non-strict. This is further amplified when significant deviation from the specification has been made - such as using the global object directly in the scope chain instead of creating an object ER to encapsulate it.

In a certain way, early simplifications can be viewed more like the style of agile development, where we only develop features as and when we need it. The goal is more short-term, and hence, ensuring short-term correctness and efficiency is more important than implementing all requirements. However, this style of development may create additional challenges for longer-term projects.

5.6.2.2 Ease of use

One of the limiting factors in development productivity for this project was the lack of proper debugging tools. There was no method to follow the steps of the JSIL program during its runtime and the only way to check the outcome of the execution was through plain text logs, which could run on for several hundreds of megabytes. Despite its shortcomings, these text logs are surprisingly well structured. With the aid of a good text editor, it is possible to emulate the use of "step-through" debugger by searching for command keywords and function names. In addition, the ability to see entire program state per execution of JSIL command was also a great help when debugging.

Chapter 6

Conclusion

In this project, we have successfully extended JaVerT to the full ES5 standard. In particular, the JS-2-JSIL compiler of JaVerT now supports the compilation of all non-strict ES5 language features while preserving the existing correct behaviour of ES5 strict features, and the whole-program symbolic testing aspect of JaVerT is now operational on full ES5. Additionally, we have further improved on JaVerT's implementation correctness in strict mode by fixing a bug not covered by ECMAScript's official test suite, Test262.

In parallel, we dived into the intricacies of the ES5 specification to discover subtle semantics and run-time behaviour not covered in Test262. Our detailed analysis also showed the advantage of using whole-program symbolic testing over concrete tests in detecting bugs in JavaScript program.

Finally, we obtained the most interesting results during our investigation of real-word implementations of ES5 in modern browsers. We managed to discover a bug in Microsoft Edge, a modern browser that is shipped by default with Windows 10. Additionally, we have learnt an important implementation detail of NodeJS that alters how scripts are interpreted, giving rise to potentially unexpected run-time behaviours.

6.1 Future work

While the body of work presented here is substantial, we acknowledge that our work is not yet complete. There are limitations previously mentioned in §5.5 that could be improved and future extensions that could be implemented in subsequent works:

1. **Proper Unicode parsing.** Currently, 8 of the 10 failed Test262 test cases is due to the limitation of the parser in handling Unicode strings within the JS program. While these tests do not affect the overall correctness of language features and JaVerT's ability to analyse most JS programs, we recognise that there might be use cases where Unicode strings are used in real-world programs. For completeness, a parser that correctly parses all source text (in both Unicode and ASCII) correctly would be a great addition.

2. **Complete implementation of ECMAScript Built-in Objects.** The main limitation that we face in whole-program symbolic analysis is the lack of symbolic implementation of certain built-ins such as the `Math` object functions. With complete implementation of the missing built-in objects, we could vastly expand the range of ES5 programs we could analyze and start to dive into real-world JS libraries written in ES5. Complete implementation of concrete built-in objects would also enable us to cover the entire Test262 test suite, providing us greater confidence in the implementation correctness of our compiler.

3. **Formal verification for ES5 programs.** One of potential direction of this project was to provide formalisation to ES5 non-strict features such as rewriting of the `scope` predicate of previous works [24, 8]. However, due to the scope and complexity of the ES5 non-strict features, we made the decision to focus on getting the correct implementation of the compiler. Currently, the specification engine (the module that provides formal verification) of JaVerT

does not work with the new compiler in ES5 non-strict mode. This is due to the numerous changes made to both the compiler, internal JSIL procedures and scope resolution. Future works could revisit this aspect and provide support for formal verification to ES5 non-strict programs. As formal verification provides a stronger guarantee for program correctness compared to concrete and symbolic tests, this extension would greatly increase the usefulness of JaVerT as a tool-chain for analysing ES5 programs.

4. **Extension to ES6+ standards.** As an evolving language, newer standards of JavaScript are always in the works by ECMAScript committee. At the time of writing this report, the latest standard is ES9 (June 2018) [5], a full 4 standards newer than the target of our project (ES5). Of course, most commercial implementations of JavaScript are nowhere near the latest standard. In fact, most modern browsers' support are approximately between ES6 and ES7[1]. By progressively moving towards later ES standards, we enable the use of JaVerT to verify more modern features such as arrow functions and promises.

[1] Compatibility table can be found on https://kangax.github.io/compat-table/es2016plus/

Appendix A

Contributions to JS-2-JSIL Compiler

No.	Function / translation case	Category	Lines of code
1	generate_main	Global code	92
2	generate_proc	Function code	215
3	generate_proc_eval	Eval code	137
4	compile_hoist_var_decls	Declaration hoisting	42
5	compile_hoist_fun_decls	Declaration hoisting	152
6	compile_var_assign	Variable assignment	57
7	make_create_function_object_call	Function object creation	11
8	Variable expression	JS expression compilation	65
9	Function call	JS expression compilation	232
10	Delete expression	JS expression compilation	121
11	Named function expressions	JS expression compilation	42
12	while statement	JS statement compilation	66
13	with statement	JS statement compilation	50
		Subtotal	**1282**

Table A.1: Compiler changes

No.	Procedure Identifier	Category	Lines of code
1	hasBinding	Environment Records	17
2	createMutableBinding	Environment Records	23
3	setMutableBinding	Environment Records	26
4	getBindingValue	Environment Records	35
5	deleteBinding	Environment Records	30
6	implicitThisValue	Environment Records	14
7	createImmutableBinding	Environment Records	8
8	initializeImmutableBinding	Environment Records	8
9	getIdentifierReference	Lexical Environments	31
10	newDeclarativeEnvironment	Lexical Environments	13
11	newObjectEnvironment	Lexical Environments	11
12	getVariableEnvironment	Lexical Environments	21
13	isDirectEval	Eval code	21
14	makeArgGetter	arguments object	15
15	makeArgSetter	arguments object	16
16	arg__get	arguments object	19
17	arg__getOwnProperty	arguments object	21
18	arg__defineOwnProperty	arguments object	52
19	arg__delete	arguments object	16
20	isStrictFunction	Function code	16
21	isPropertyReference	Reference abstract operations	19
		Subtotal	**432**

Table A.2: New internal JSIL procedures

No.	Procedure Identifier	Category	Lines of code
1	getOwnProperty	Internal functions	21
2	get	Internal functions	15
3	deleteProperty	Internal functions	15
4	o__delete	Internal functions	19
5	defineOwnProperty	Internal functions	21
6	i__getValue	Reference abstract operations	57
7	i__putValue	Reference abstract operations	59
8	Function_construct	Function constructor	41
9	FP_call	Function prototype	35
10	FP_apply	Function prototype	50
11	Array_construct	Array constructor	44
12	AP_toString	Array prototype	24
13	AP_pop	Array prototype	23
14	AP_push	Array prototype	34
15	AP_every	Array prototype	47
16	AP_some	Array prototype	47
17	AP_forEach	Array prototype	41
18	AP_map	Array prototype	47
19	AP_filter	Array prototype	58
20	AP_reduce	Array prototype	58
21	AP_reduceRight	Array prototype	60
22	Number_construct	Number constructor	16
23	Number_call	Number prototype	15
24	Boolean_construct	Boolean constructor	9
25	Boolean_call	Boolean prototype	4
26	BP_toString	Boolean prototype	25
27	BP_valueOf	Boolean prototype	16
28	String_call	String prototype	18
		Subtotal	**919**

Table A.3: Existing JSIL procedures with significant changes[†]

Summary

Functions created/modified: 62

Lines of code involved: (approximately) 2633

Note: Lines of code are approximations as they may include comments within the functions (comments outside of the function body are excluded).

[†]These include extensions made to ES5 non-strict behaviour and rewriting of existing internal JSIL procedures.

Appendix B

Breakdown of Test262 results

B.1 Language Tests

Core	Total	Passing	Failing	Aborting
arguments-object	46	46	0	0
asi	101	101	0	0
comments	18	16	0	2
directive-prologue	62	62	0	0
eval-code	58	58	0	0
function-code	212	206	0	6
future-reserved-words	55	55	0	0
global-code	3	3	0	0
identifier-resolution	11	11	0	0
identifiers	49	49	0	0
keywords	25	25	0	0
line-terminators	41	37	4	0
literals	145	115	0	30
punctuators	11	11	0	0
reserved-words	13	13	0	0
source-text	1	0	1	0
types	109	109	0	0
white-space	40	40	0	0
Total	1000	959	5	38

(a) Language tests: Core

Statements	Total	Passing	Failing	Aborting
block	11	11	0	0
break	19	18	1	0
continue	15	15	0	0
do-while	24	23	0	1
empty	1	1	0	0
expression	3	3	0	0
for	64	62	2	0
for-in	16	16	0	0
function	214	214	0	0
if	22	22	0	0
labeled	1	1	0	0
return	15	15	0	0
switch	11	11	0	0
throw	14	14	0	0
try	70	70	0	0
variable	64	64	0	0
while	23	22	0	1
with	146	146	0	0
Total	733	728	3	2

(b) Language tests: Statements

Expressions	Total	Passing	Failing	Aborting
addition	35	34	0	1
array	11	11	0	0
assignment	42	42	0	0
bitwise-and	23	23	0	0
bitwise-not	14	14	0	0
bitwise-or	23	23	0	0
bitwise-xor	23	23	0	0
call	30	30	0	0
comma	5	5	0	0
compound-assignment	353	353	0	0
concatenation	5	5	0	0
conditional	15	15	0	0
delete	56	56	0	0
division	35	35	0	0
does-not-equals	30	30	0	0
equals	31	31	0	0
function	19	19	0	0
greater-than-or-equal	37	37	0	0
greater-than	41	41	0	0
grouping	9	9	0	0
in	14	14	0	0
instanceof	34	34	0	0
left-shift	38	38	0	0
less-than-or-equal	41	41	0	0
less-than	37	37	0	0
logical-and	16	16	0	0
logical-not	17	17	0	0
logical-or	16	16	0	0
modulus	32	32	0	0
multiplication	33	33	0	0
new	13	13	0	0
object	38	38	0	0
postfix-decrement	25	25	0	0
postfix-increment	26	26	0	0
prefix-decrement	22	22	0	0
prefix-increment	21	21	0	0
property-accessors	20	19	0	1
relational	1	1	0	0
right-shift	30	30	0	0
strict-does-not-equals	22	22	0	0
strict-equals	22	22	0	0
subtraction	32	32	0	0
this	6	6	0	0
typeof	1	1	0	0
unary-minus	12	12	0	0
unary-plus	16	16	0	0
unsigned-right-shift	38	38	0	0
void	9	9	0	0
Total	1469	1467	0	2

(c) Language tests: Expressions

Total Language	Total	Passing	Failing	Aborting
	3202	3152	8	42
		98.44%	0.25%	1.31%

(d) Language tests: Summary

Table B.1: Language test results

No.	Name	Type	Status	Reason
1	comments/S7.4_A5.js	comments	Abort	String.fromCharCode (not impl)
2	comments/S7.4_A6.js	comments	Abort	String.fromCharCode (not impl)
3	expressions/addition/S11.6.1_A2.2_T2.js	addition	Abort	Date.prototype.toString (not impl)
4	expressions/delete/11.4.1-5-a-28-s.js	delete	Abort	RegExp (not impl)
5	expressions/property-accessors/S11.2.1_A3_T2.js	property-accessors	Abort	Number.prototype.toFixed (not impl)
6	function-code/10.4.3-1-100-s.js	function-code	Abort	String.prototype.replace (not impl)
7	function-code/10.4.3-1-100gs.js	function-code	Abort	String.prototype.replace (not impl)
8	function-code/10.4.3-1-101-s.js	function-code	Abort	String.prototype.replace (not impl)
9	function-code/10.4.3-1-101gs.js	function-code	Abort	String.prototype.replace (not impl)
10	function-code/10.4.3-1-102-s.js	function-code	Abort	String.prototype.replace (not impl)
11	function-code/10.4.3-1-102gs.js	function-code	Abort	String.prototype.replace (not impl)
12	literals/null/S7.8.1_A1_T2.js	literals	Abort	RegExp (not impl)
13	literals/regexp/7.8.5-2gs.js	literals	Abort	RegExp (not impl)
14	literals/regexp/S7.8.5_A1.1_T1.js	literals	Abort	RegExp (not impl)
15	literals/regexp/S7.8.5_A1.1_T2.js	literals	Abort	RegExp (not impl)
16	literals/regexp/S7.8.5_A1.4_T1.js	literals	Abort	RegExp (not impl)
17	literals/regexp/S7.8.5_A1.4_T2.js	literals	Abort	RegExp (not impl)
18	literals/regexp/S7.8.5_A2.1_T1.js	literals	Abort	RegExp (not impl)
19	literals/regexp/S7.8.5_A2.1_T2.js	literals	Abort	RegExp (not impl)
20	literals/regexp/S7.8.5_A2.4_T1.js	literals	Abort	RegExp (not impl)
21	literals/regexp/S7.8.5_A2.4_T2.js	literals	Abort	RegExp (not impl)
22	literals/regexp/S7.8.5_A3.1_T1.js	literals	Abort	RegExp (not impl)
23	literals/regexp/S7.8.5_A3.1_T2.js	literals	Abort	RegExp (not impl)
24	literals/regexp/S7.8.5_A3.1_T3.js	literals	Abort	RegExp (not impl)
25	literals/regexp/S7.8.5_A3.1_T4.js	literals	Abort	RegExp (not impl)
26	literals/regexp/S7.8.5_A3.1_T5.js	literals	Abort	RegExp (not impl)
27	literals/regexp/S7.8.5_A3.1_T6.js	literals	Abort	RegExp (not impl)
28	literals/regexp/S7.8.5_A4.1.js	literals	Abort	RegExp (not impl)
29	literals/regexp/S7.8.5_A4.2.js	literals	Abort	RegExp (not impl)
30	literals/string/S7.8.4_A4.1_T1.js	literals	Abort	String.fromCharCode (not impl)
31	literals/string/S7.8.4_A4.1_T2.js	literals	Abort	String.fromCharCode (not impl)
32	literals/string/S7.8.4_A4.2_T1.js	literals	Abort	String.fromCharCode (not impl)
33	literals/string/S7.8.4_A4.2_T3.js	literals	Abort	String.fromCharCode (not impl)
34	literals/string/S7.8.4_A4.2_T5.js	literals	Abort	String.fromCharCode (not impl)
35	literals/string/S7.8.4_A4.2_T7.js	literals	Abort	String.fromCharCode (not impl)
36	literals/string/S7.8.4_A5.1_T1.js	literals	Abort	String.fromCharCode (not impl)
37	literals/string/S7.8.4_A6.1_T1.js	literals	Abort	String.fromCharCode (not impl)
38	literals/string/S7.8.4_A6.3_T1.js	literals	Abort	String.fromCharCode (not impl)
39	literals/string/S7.8.4_A7.1_T1.js	literals	Abort	String.fromCharCode (not impl)
40	literals/string/S7.8.4_A7.3_T1.js	literals	Abort	String.fromCharCode (not impl)
41	statements/do-while/S12.6.1_A8.js	do-while	Abort	String.prototype.split (not impl)
42	statements/while/S12.6.2_A8.js	while	Abort	String.prototype.split (not impl)
43	line-terminators/7.3-15.js	line-terminators	Failing	Parser (unicode)
44	line-terminators/7.3-5.js	line-terminators	Failing	Parser (unicode)
45	line-terminators/7.3-6.js	line-terminators	Failing	Parser (unicode)
46	line-terminators/invalid-string-cr.js	line-terminators	Failing	Parser (unicode)
47	source-text/6.1.js	source-text	Failing	Parser (unicode)
48	statements/for/head-init-expr-check-empty-inc-empty-completion.js	for-statement	Failing	"if" completion updates empty return to undefined (ES6+)
49	statements/for/head-init-var-check-empty-inc-empty-completion.js	for-statement	Failing	"if" completion updates empty return to undefined (ES6+)
50	statements/break/S12.8_A3.js	break	Failing	Function declaration in statement position (ES6+)

Table B.2: Breakdown of all language failures and aborts

B.2 Built-in Tests

Global	Total	Passing	Failing	Aborting
Infinity	7	7	0	0
NaN	7	7	0	0
decodeURI	52	7	0	45
decodeURIComponent	52	7	0	45
encodeURI	28	7	0	21
eval	7	7	0	0
encodeURIComponent	28	7	0	21
global	31	31	0	0
isFinite	2	2	0	0
isNaN	2	2	0	0
parseFloat	40	7	0	33
parseInt	57	7	0	50
undefined	8	8	0	0
Total	321	106	0	215

(a) Built-in tests: Global Object

JSON	Total
Total	90
Passing	9
Failing	0
Aborting	81

RegExp	Total
Total	501
Passing	39
Failing	0
Aborting	462

Math	Total
Total	81
Passing	81
Failing	0
Aborting	0

Date	Total
Total	430
Passing	413
Failing	0
Aborting	17

Error	Total
Total	33
Passing	33
Failing	0
Aborting	0

Object	Total
Total	2892
Passing	2886
Failing	0
Aborting	6

Function	Total
Total	398
Passing	394
Failing	0
Aborting	4

Array	Total
Total	2171
Passing	2168
Failing	0
Aborting	3

String	Total
Total	749
Passing	337
Failing	0
Aborting	412

Number	Total
Total	152
Passing	144
Failing	3
Aborting	5

Boolean	Total
Total	42
Passing	42
Failing	0
Aborting	0

(b) Built-in tests: Other built-in objects

BUILT-INS	Total	Passing	Failing	Aborting
	7860	6652	3	1205
		84.63%	0.04%	15.33%

(c) Built-in tests: Summary

Table B.3: Built-in test results

Bibliography

[1] Anand, S., Godefroid, P., and Tillmann, N. Demand-driven compositional symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 367–381.

[2] Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., and Smith, G. A trusted mechanised javascript specification. In *ACM SIGPLAN Notices* (2014), vol. 49, ACM, pp. 87–100.

[3] ECMA TC39. Github - tc39/test262: Official ecmascript conformance test suite. https://github.com/tc39/test262. (Accessed on 01/22/2019).

[4] ECMA TC39. The 5th Edition of the ECMAScript Language Specification. Tech. rep., ECMA, 2011.

[5] ECMA TC39. The 9th edition of the ECMAScript Language Specification. Tech. rep., ECMA, 2018.

[6] Esprima. Esprima. http://esprima.org/. (Accessed on 01/23/2019).

[7] Foundation, N. Node.js. https://nodejs.org/en/. (Accessed on 06/15/2019).

[8] Fragoso Santos, J., Maksimović, P., Sampaio, G., and Gardner, P. Javert 2.0: compositional symbolic execution for javascript. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 66.

[9] Gardner, P., Maffeis, S., and Smith, G. Towards a program logic for javascript. In *POPL* (2012), pp. 31–44.

[10] GitHub. Projects | the state of the octoverse. https://octoverse.github.com/projects#languages. (Accessed on 01/17/2019).

[11] GitHut. Githut - programming languages and github. https://githut.info/. (Accessed on 01/17/2019).

[12] Godefroid, P. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2007), POPL '07, ACM, pp. 47–54.

[13] Hafiz, M., Hasan, S., King, Z., and Wirfs-Brock, A. Growing a language: An empirical study on how (and why) developers use some recently-introduced and/or recently-evolving javascript features. *Journal of Systems and Software 121* (2016), 191–208.

[14] Hidayat, A. Validating strict mode. https://ariya.io/2012/10/validating-strict-mode. (Accessed on 01/23/2019).

[15] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (1969), 576–580.

[16] International, E. ECMA-404—The JSON Data Interchange Format. Tech. rep., ECMA, 2011.

[17] King, J. C. Symbolic execution and program testing. *Communications of the ACM 19*, 7 (1976), 385–394.

[18] Lee, H., Won, S., Jin, J., Cho, J., and Ryu, S. Safe: Formal specification and implementation of a scalable analysis framework for ecmascript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages* (2012), Citeseer, p. 96.

[19] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic* (2001), Springer, pp. 1–19.

[20] Park, C., Lee, H., and Ryu, S. All about the with statement in javascript: Removing with statements in javascript applications. *ACM SIGPLAN Notices 49*, 2 (2014), 73–84.

[21] Park, J., Ryou, Y., Park, J., and Ryu, S. Analysis of javascript web applications using safe 2.0. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on* (2017), IEEE, pp. 59–62.

[22] Richards, G., Hammer, C., Burg, B., and Vitek, J. The eval that men do. In *European Conference on Object-Oriented Programming* (2011), Springer, pp. 52–78.

[23] Santos, J. F., Maksimović, P., Grohens, T., Dolby, J., and Gardner, P. Symbolic execution for javascript. In *PPDP* (2018), vol. 11, pp. 1–11.

[24] Santos, J. F., Maksimovic, P., Naudziuniene, D., Wood, T., and Gardner, P. Javert: Javascript verification toolchain. *PACMPL 2*, POPL (2018), 50–1.

[25] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., and Song, D. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 513–528.

[26] w3techs. Usage statistics of javascript for websites, january 2019. https://w3techs.com/technologies/details/cp-javascript/all/all. (Accessed on 01/17/2019).