IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Intermediate Language for JavaScript

Author:
Aubrianna Zhu

Supervisor:
Philippa Gardner

Submitted in partial fulfillment of the requirements for the MSc degree in Computing Science of Imperial College London

July 27, 2016

Abstract

TODO

Acknowledgments

TODO

Contents

Chapter 1

Introduction

1.1 What is JavaScript

JavaScript is the unofficial language of the Web—it is implemented in all major browsers, and its main purpose is to add interactivity to web pages by controlling both the style and the content of a browser's document object model. Increasingly, JavaScript is also being used on the server-side, as a means of interacting with databases. It exhibits a number of interesting features; for example, it supports prototypical inheritance rather than class inheritance, which is favored by many other programming languages. It allows for extensible objects—objects that can be altered after creation. Also, as there is virtually no difference between an object and a type, JavaScript does not have static typing or require type declarations [9]. The intricacies of JavaScript will be discussed in chapter 4.

Since its inception, six versions of the standard have been developed, with the 6^{th} having been released in June 2015 [10]. This project will primarily target the 5^{th} edition of the ECMAScript standard, which was published in 2011. Significant changes have occurred in the 5^{th} edition, including the addition of accessor properties and a strict mode [9], both of which will be addressed in this project.

1.2 Verification and its Importance

Computer scientists have created programming languages like JavaScript with the idea of instructing machines to perform actions in a human-like way, but possibilities for mistakes exist even in the creation phase. Indeed, a real life programming language was shown to have failed type safety five years after its creation [8]. It is of little help for correctness that, just like natural languages, programming languages evolve and become increasingly complicated. Moreover, programming language specifications are often written in plain prose that sometimes lacks detail or is simply ambiguous, which can result in diverging interpretations by different readers. When a programmer writes a piece of code that performs a certain task, without proper tools he cannot be entirely sure that his program will perform the task as required without side effects. In order to reduce errors and inconsistencies in different implementations of the same language, a formal verification system is necessary to prove that the programming languages are indeed performing correctly and as expected, with no ambiguity.

1.3 JavaScript Verification

Previous work on JavaScript verification that makes this project possible include formal reasoning about JavaScript using program logics [12]; JSCert and JSRef, which are a mechanized specification of the ECMAScript standard and a certified reference interpreter, both written in the Coq proof assistant [2]; and the JSIL intermediate language, which relies on JSCert and is at the core of this project. All of these works will be discussed in more detail in sections 2.3.1 to 2.3.3.

1.4 Outline of This Report

After introducing programming language verification, this report will proceed to give an overview of the history and landscape of mechanized specifications for a variety of programming languages. Then, it will give a detailed explanation of operational semantics, a formalization method used by many mechanized verification projects, before providing more detail on the topic of JavaScript verification. Next, it will demonstrate existing work on translating ECMAScript standards into operational semantics, and from operational semantics to actual JSIL code. It will then provide detailed discussions on the topics of JavaScript, JSIL as an intermediate language, the JavaScript Array Library, and the Arguments Object, before moving on to discuss the process and results of testing the JSIL implementation. Lastly, a brief conclusion will wrap up the topics studied, and introduce possible future extensions.

Chapter 2

Background

2.1 History of Mechanized Specifications

There has been a long history of mechanized specification of programming languages. Initial formalizations were often simple, as they did not cover all aspects of the languages they targeted. Additionally, they tended to be non-executable and thus the appropriateness of their definitions were at times uncertain [11]. Gradually, formalizations became increasingly mechanized, executable, and testable against standard test suites such as ECMAScript Test262 and GCC torture tests, or specific implementations such as Mozilla's JavaScript in order to ascertain real life correctness of these specifications. The following subsections will describe a number of mechanized specifications for various programming languages.

2.1.1 ML

Lee et al. 2007 This study created an internal language (IL), whose semantics is an explicitly-typed λ-calculus and follows a variation of the Harper and Stone[17] style, for standard ML [20]. This IL was able to capture all of the features in standard ML, and the proof of its type safety was successfully formalized in LF [16] and Twelf [28].

2.1.2 C

There has been many formal semantics written for C, ranging from those using abstract state machines [15], to the HOL theorem proving system [25], in addition to the two described below.

Leroy 2009 CompCert is a certified compiler from Clight to PowerPC assembly code, and it was designed to address the correctness of compilers, as most formal verification is done on source code rather than on code produced by optimized compilation. It used the Coq proof assistant for formal verification, to prove that the semantics of the generated assembly code corresponds exactly to the semantics of the source code [21]. This verification is accomplished using eight intermediate languages from Clight to assembly, with each transformation having a formal proof that it preserves the semantics of the previous language in the translation chain.

Ellison and Rosu 2012 This work employs $\mathbb{K}$ [32], a semantic framework based on rewriting-logic, and Maude [5], a rewriting-logic engine that enables execution and analysis of the mechanized semantics [11]. The $\mathbb{K}$-based semantics for C is executable and was tested against the GCC torture test suite [11].

2.1.3 Java

As for C, there exists a large body of work concerning the formalization of Java.

Drossopoulou and Eisenbach 1998 As early as 1998, a small step operational semantics has been defined for Java [8] [7] using a term rewrite system. This work has covered a substantial subset of the Java language, with a focus on proving that the type system indeed works correctly.

Klein and Nipkow 2006 Another example is work done using the Isabelle theorem prover [27], an environment to formalize and prove the type safety for Jinja [19], a variation of Java. The authors utilized both small step and big step operational semantics—the two styles are used in different aspects of the proof, but they are shown to be equivalent. This work is important because it was the first holistic formal analysis of a Java-like language, its virtual machine, and its compiler [19].

Bogdanas and Rosu 2015 The most recent work done on formalizing Java follows the C example from section 2.1.2, using $\mathbb{K}$ as the development tool for Java 1.4. In this project, all features in the Java specification were covered and tested [3], as $\mathbb{K}$ automatically provides the executable environment for the formal specification, named K-Java.

2.1.4 JavaScript

This section presents four sample studies, with the first being a purely theoretical translation of the ECMAScript standard, whereas the other three also have executable portions.

Maffeis et al. 2008 The authors produced an extensive small step operational semantics for ECMAScript 3, encompassing about 70 pages of rules and definitions [23]. The main goal of this formalization was to analyze the security properties of JavaScript programs running in web browsers, as the operational semantics allowed the authors to understand what can and cannot interact with the heap when a program is run. This operational semantics was not directly executable, but the authors designed experiments to test certain constructs on different browser implementations, and found discrepancies between Firefox, Safari, and the JavaScript standard implementation [23]. This work was important in inspiring other researchers to pursue operational semantics as a way to fully understand JavaScript behavior.

Politz et al. 2012 A formal semantics, S5, for the strict mode of ECMAScript 5.1 was developed following the semantics of λ_{JS} [14]. It covers the core semantics of JavaScript, and all JavaScript features that are not immediately covered can be transformed via a desugaring function. It is executable and tested for conformance with the ECMAScript test suite, Test262. S5 focuses in particular, on the getters and setters (accessor properties), and the `eval` operator [30].

Bodin et al. 2014 In this work, the ECMAScript 5 standard was mechanized using pretty big step semantics (JSCert) and a reference interpreter (JSRef) was implemented using the Coq theorem prover. JSRef was proven correct with respect to JSCert, then extracted to OCaml using Coq's built-in extraction mechanism, and tested against Test262 [2]. The correctness result ensures that both JSCert and JSRef are correct with respect to the JavaScript standard. The current project stems from this work, and JSCert and JSRef will be discussed in more detail in section 2.3.2.

Park et al. 2015 Similar to the study presented in Section 2.1.2, this formal semantics of JavaScript is also based on $\mathbb{K}$ [33]. KJS was tested against Test262, and its language construct closely resembles the ECMAScript standard. In addition, it was able to discover coverage deficiencies in Test262 itself [26].

2.2 Operational Semantics

Like natural languages, programming languages have the distinction between syntax and semantics. Syntax refers to the physical structure and rules of the language, defining valid variable names and correct placement of semicolons; semantics refers to the meaning of the program, specifying the actions and results of program commands. In regards to verification, semantics is what makes a program correct or incorrect. There are three different categories of program semantics: denotational, axiomatic, and operational. Operational semantics has turned out to be the category of choice, since it specifies meaning by giving the steps and behavior of a program during execution [29]. In general, there are two ways of representing the semantics: small step and big step.

Presented below is the syntax of a simple imperative programming language, whose syntax will be followed in the upcoming subsections. The `While` command will serve as a running example throughout this report.

$$\begin{aligned}
B \in \text{Bool} &::= \texttt{true} \mid \texttt{false} \mid E = E \mid E < E \mid B \& B \mid \neg B \mid \ \ldots \\
E \in \text{Exp} &::= \texttt{x} \mid \texttt{n} \mid E + E \mid \ \ldots \\
C \in \text{Com} &::= \texttt{x} := E \mid \texttt{if B then C else C} \mid C; C \mid \texttt{skip} \mid \texttt{while B do C} \mid \ldots
\end{aligned}$$

where we denote variables by `x` and numbers by `n`. Configurations are defined as $\langle E, s\rangle$, $\langle B, s\rangle$ and $\langle C, s\rangle$, meaning expressions, booleans, and commands with respect to state s.

2.2.1 Small Step

Small step operational semantics is also called structural operational semantics, and it evaluates expressions one step at a time [13]. Due to its fine granularity, small step semantics is often used to model type systems and concurrency [4], as well as mechanized programming language definitions. The relation between two states is represented by the $\rightarrow$ symbol, and an example of this semantics is given below. In S-LEFT, an expression is evaluated in the order from left to right, thus $\mathbf{E}_1$ is evaluated before $\mathbf{E}_2$. If the left hand side is already a numeral, then the expression on the right hand side, **E**, will be evaluated in S-RIGHT. Once all expressions are evaluated, addition will occur as in S-ADD.

$$\text{S-LEFT}\ \frac{\mathbf{E}_1 \rightarrow \mathbf{E}_1'}{\mathbf{E}_1 + \mathbf{E}_2 \rightarrow \mathbf{E}_1' + \mathbf{E}_2} \qquad \text{S-RIGHT}\ \frac{\mathbf{E} \rightarrow \mathbf{E}'}{n + \mathbf{E} \rightarrow n + \mathbf{E}'} \qquad \text{S-ADD}\ \frac{\underline{\mathtt{n}_1} + \underline{\mathtt{n}_2} = \underline{\mathtt{n}_3}}{\mathtt{n}_1 + \mathtt{n}_2 \rightarrow \mathtt{n}_3}$$

A more concrete example, $3 + (2 + 1) \rightarrow 3 + 3 \rightarrow 6$, would require the S-RIGHT rule to be applied to (2 + 1) and the S-ADD rule to produce 3 to give the intermediate stage 3 + 3, then another S-ADD to give the final result 6 [13].

As for a `While` command, only one rule is necessary as the semantics takes care of the recursive looping on its own. Only the initial step needs to be defined and the rule will carry on until it reaches a `skip` condition. Details for the small step evaluation rules for sequential composition and conditionals have been omitted, but they both contribute towards defining the `While` semantics.

The following small step semantics for `While` is presented in the Models of Computation course at Imperial for second-year Computing students. [1]

$$\text{SS-WHILE}\quad \frac{}{\langle \texttt{while B do C, s}\rangle \to_c \langle \texttt{if B then (C; while B do C) else skip, s}\rangle}$$

2.2.2 Big Step

Big step operational semantics is also called natural operation semantics, and in contrast to small step, it evaluates the entire expression in one step and returns the final result immediately [13], and the behavior of one term can be seen as the composition of the behaviors of the sub-terms [4]. The relation between two states is represented by the $\Downarrow$ symbol, and the same example is given below. In B-NUM, a number is evaluated to a number. The equivalent was not in the small step semantics because numbers are considered to be fully evaluated. In B-ADD, $\mathbf{E}_1$ and $\mathbf{E}_2$ are evaluated separately to two numbers, $\mathtt{n}_1$ and $\mathtt{n}_2$, respectively, while applying actual addition in the same rule to arrive at the final answer $\mathtt{n}_3$.

$$\text{B-NUM}\quad \frac{}{\mathtt{n} \Downarrow \mathtt{n}} \qquad\qquad \text{B-ADD}\quad \frac{\mathbf{E}_1 \Downarrow \mathtt{n}_1 \qquad \mathbf{E}_2 \Downarrow \mathtt{n}_2 \qquad \underline{\mathtt{n}_1} + \underline{\mathtt{n}_2} = \underline{\mathtt{n}_3}}{\mathbf{E}_1 + \mathbf{E}_2 \Downarrow \mathtt{n}_3}$$

The same example yields $3 + (2 + 1) \Downarrow 6$ for big step semantics because the final result is produced immediately [13].

The `While` command for big step semantics requires two different rules [31], one for when the boolean condition is `true`, another for `false`. In BS-WHILE-FALSE, the boolean condition *B* evaluates to `false`, so the command *C* is never executed, and the program state *s* stays the same after evaluation. In BS-WHILE-TRUE, *B* evaluates to `true`, so the while loop is unrolled once, command C is executed, which alters the state from s to s', and the while loop is called recursively with the state s'.

$$\text{BS-WHILE-FALSE}\quad \frac{\langle \texttt{B, s}\rangle \Downarrow \langle \texttt{false}\rangle}{\langle \texttt{while B do C, s}\rangle \Downarrow \langle \texttt{s}\rangle} \qquad \text{BS-WHILE-TRUE}\quad \frac{\langle \texttt{B, s}\rangle \Downarrow \langle \texttt{true}\rangle \quad \langle \texttt{C, s}\rangle \Downarrow \langle \texttt{s'}\rangle \quad \langle \texttt{while B do C, s'}\rangle \Downarrow \langle \texttt{s''}\rangle}{\langle \texttt{while B do C, s}\rangle \Downarrow \langle \texttt{s''}\rangle}$$

2.2.3 Pretty Big Step

A variation of big step semantics has been used in describing JavaScript program logic [2] developed by Charguraud [4]. While big step semantics has the advantage of immediacy and hence intuitively makes more sense, it does have several disadvantages, the biggest of which is the redundancy of premises for many different evaluation rules when exceptions and/or divergence occur [4]. Thus, for programs with no rule repetitions, big step semantics and pretty big step semantics can be quite indistinguishable, but if an entire programming language must be mechanized, big step semantics simply does not scale. Indeed, the work done so far with JavaScript verification follows the style of pretty big step semantics since it addresses the redundancy problem. The running example is given below.

[1] It was necessary to self-study a portion of the materials presented in the second-year Models of Computation course to get up to speed with operational semantics.

In $\mathbf{E}_1 + \mathbf{E}_2$, to avoid redundancy as in big step semantics, one expression is evaluated at a time and in the order from left to right. So PB-ADD evaluates the first expression, $\mathbf{E}_1$, and utilizes an operator $+_1$ for when the first expression is already evaluated. In PB-$+_1$, the $+_1$ rule is defined by evaluating the second expression and utilizing a $+_2$ operator for when both expressions are numbers. PB-$+_2$ then performs the actual addition between two numbers using $+_2$, propagating the result of $+_2$ back to PB-$+_1$, which then propagates its result to PB-ADD to yield the final result, and the cycle is complete.

$$\text{PB-ADD}\ \frac{\mathbf{E}_1 \Downarrow \mathtt{n}_1 \qquad \mathtt{n}_1 +_1 \mathbf{E} \Downarrow \mathtt{n}}{\mathbf{E}_1 + \mathbf{E}_2 \Downarrow \mathtt{n}} \qquad \text{PB-}+_1\ \frac{\mathbf{E} \Downarrow \mathtt{n}_2 \qquad \mathtt{n}_1 +_2 \mathtt{n}_2 \Downarrow \mathtt{n}_3}{\mathtt{n} +_1 \mathbf{E} \Downarrow \mathtt{n}_3} \qquad \text{PB-}+_2\ \frac{}{\mathtt{n}_1 +_2 \mathtt{n}_2 \Downarrow \underline{\mathtt{n}_1} + \underline{\mathtt{n}_2}}$$

This would also yield $3 + (2 + 1) \Downarrow 6$, but the difference here is that two different versions of the + operator were defined.

2.3 JavaScript Specifications

2.3.1 JS Logic

Gardner et al. [12] have demonstrated that a program logic based on separation logic can be used to reason about JavaScript, since separation logic is useful to reason about programs that manipulate the heap, and JavaScript stores the entire program state in the object heap. A concept of weak locality was employed to establish the soundness of reasoning, since program commands are required to be local but JavaScript commands are not [12]. The actual soundness proof was done using big step operational semantics that was defined for most of the ECMAScript 3 standard. This big step semantics supplements previous effort to develop a small step operational semantics for core ECMAScript 3 functionality by Maffeis et al. [23]. An explanation of operational semantics can be seen in Section 2.2.

2.3.2 JSCert and JSRef

JSCert is a Coq specification of JavaScript [2]. JSCert rules are in the form of $t/S/C \Downarrow_s o$, where t is a statement, S is the object heap and environment record, C is the execution context, and o is the output, meaning that if we execute the statement t starting from the object heap S and execution context C, we will obtain the output o. The output consists of the final state of the heap and a completion triple indicating the type of the return, the type of return value, and the return label [2]. JSCert can be used to reason only about the scenarios where the program terminates, and not those in which it diverges.

JSRef, on the other hand, although also written in Coq, is a reference interpreter rather than a specification. JSRef contains records of functions that evaluate various parts of JavaScript source code, and recursively returns the result of program execution [2]. JSRef functions are extracted automatically into OCaml code thanks to Coq, which enables JSRef functions to be run directly on Test262 to establish the correctness of both JSRef and JSCert with respect to the ECMAScript standard.

2.3.3 JSVerify

For languages such as C and Java, efforts to verify program correctness have well been underway, as described in Section 2.1, yielding many integrated development environments (IDEs) that perform dynamic and static analyses of code and help programmers debug their programs. Current

JavaScript IDEs provide little more than simple syntax highlighting and do not adequately address program correctness. Coupled with its dynamic nature, JavaScript code is often buggy [12] and even vulnerable to security attacks [20].

The ideas presented in sections 2.3.1 and 2.3.2 culminated into a verification project called JSVerify, which is currently ongoing. Its aims are to use separation logic to reason about JavaScript programs and provide proofs of their correctness. This project ties in with JSVerify in that JSVerify will actually analyze code written in the intermediate language JSIL from this project, in order to determine the corresponding JavaScript program's correctness. This can be accomplished because there exists a proof of correctness of the translation from JavaScript to JSIL.

2.3.4 Intermediate Languages for JavaScript Verification

Most real world programming languages are far too complex for reasonable analysis, and simpler forms of these languages are necessary for verification. Intermediate representations provide that simplicity. Historically, intermediate representations have been used in compilers, translating the source language into an intermediate form before translating into assembly code [24]. As the need for language verification arose, a form of intermediate representation, called intermediate language, found its usefulness in helping analyze the source program in addition to being used in compilers.

Currently, many intermediate languages exist, and one of the most prominent is that for C, called CIL. Unlike C, CIL has only one looping construct, it makes the return statement explicit for all functions, and it simplifies the type system [24].

JavaScript Intermediate Languages

JSIR One intermediate language is JSIR [22], which aims to make different analysis tools more unified and less platform-dependent. Much of its grammar is defined, and can be found in the Github as cited [22]. However, up to this point, JSIR has not been implemented and there is no compilation provided from any higher-level programming language to JSIR.

WALA Another interesting intermediate language, initially developed by IBM, is WALA [6]. Unlike most other intermediate languages which are developed and used in academia, WALA is a set of full-fledged Java libraries used in production code for static and dynamic analyses. Although the bytecode is written in Java, it also supports simple program analyses for JavaScript in the form of JS-WALA [1].

Lambda JS / S5 λ_{JS} [14] is a small step operational semantics that claims to capture the essential features of JavaScript, except for the `eval` functionality. S5 extends λ_{JS} to the ECMAScript 5 standard and it focuses on JavaScript's strict mode [30]. Additional features of S5 include accessors and covering the `eval` function. JavaScript is translated to the core λ_{JS} language via a desugaring function, and both λ_{JS} and $S5$ are executable and tested against a Mozilla test suite and the Test262 suite, respectively.

JSIL Like other intermediate languages, JSIL does not have many of the high-level language constructs. Instead, it has simple `goto` commands with labels and actions that resemble assembly code. JSIL has a memory model that is similar to that of JavaScript, and since JavaScript heaps are paramount in determining the outcome of running programs, this similarity helps relate JSIL programs with JavaScript programs.[2] As seen in the heap models below, JavaScript

[2]Active work is in progress for JSIL and pending publication.

heaps map locations, $\mathcal{L}$, and JavaScript variables, $\mathcal{X}_{JS}$, to JavaScript values, $\mathcal{V}_{JS}$, and scope chains, $\mathcal{S}ch_{JS}$, whereas JSIL heaps map locations, $\mathcal{L}$, and JavaScript variables, $\mathcal{X}_{JS}$, to JSIL values, $\mathcal{V}_{JSIL}$.

JavaScript Heap Model
$\mathtt{h} \in \mathcal{H}_{JS} : \mathcal{L} \times \mathcal{X}_{JS} \rightharpoonup (\mathcal{V}_{JS} \cup \mathcal{S}ch_{JS})$

JSIL Heap Model
$\mathtt{h} \in \mathcal{H}_{JSIL} : \mathcal{L} \times \mathcal{X}_{JS} \rightharpoonup \mathcal{V}_{JSIL}$

The syntax of JSIL will be discussed in detail in Chapter 5.

High-Level Languages Lastly, there are many studies focusing on high level constructs of JavaScript rather than using intermediate representations for JavaScript verification. For example, Jensen et al. [18] describes a way to transform JavaScript programs to eliminate the use of `eval` functions to improve static analysis of JavaScript source code. But as these methods only verify the source programs, and not the programming language itself, they are outside the scope of this project and will not be discussed in further detail.

2.4 Operational Semantics and Specification in Action

2.4.1 While Loop

A simple `While` command in operational semantics was demonstrated for both small step and big step semantics in section 2.2. In JavaScript, however, the definition for `While` is much more complicated. The following subsections will describe the `While` command as defined in the JavaScript standard, in operational semantics, and finally in JSIL code.

ECMAScript 5 Standard

The while statement is given as the following in chapter 12.6.2 [9]. Its definition makes use of various JavaScript constructs such as references, predefined JavaScript values, and label sets. It also depends on other built-in functions such as `ToBoolean` and `GetValue`. Thus to completely understand how `While` functions, one must understand these other constructs as well.

The production *IterationStatement* : while *(Expression) Statement* is evaluated as follows:

1. Let V = `empty`.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (`normal, V, empty`).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not `empty`, let V = *stmt*.value.
   e. If *stmt*.type is not `continue` || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is `break` and *stmt*.target is in the current label set, then
         A. Return (`normal, V, empty`).
      ii. If *stmt* is an abrupt completion, return *stmt*.

ES5 Strict Operational Semantics

Based on the ECMAScript standard definition, operational semantics for the `While` command in JSIL is much more complicated since it needs to faithfully represent the standard. Each bullet in the standard's text is transcribed into a rule in the operational semantics, and the repeat statement is taken care of by loop unrolling of the semantics.

In the style of pretty big step semantics, the relation takes the form of $L, l \vdash \langle h, s \rangle \Downarrow \langle h', o \rangle$. L is the scope chain, l is the location of *this*, h and h' are the initial and final heaps, s is the statement to be evaluated, and o is the outcome. In the rules below, s is defined to be $while(e)s$, and $\Downarrow^{\gamma}$ represents the actual evaluation of expressions or statements.

$$\textsc{While}\quad \frac{L, l \vdash \langle h, \text{while}_1(e)\{s, \text{empty}\}\rangle \Downarrow \langle h', o' \rangle}{L, l \vdash \langle h, \text{while}(e)\{s\}\rangle \Downarrow \langle h', o' \rangle}$$

$$\textsc{While-1}\quad \frac{L, l \vdash \langle h, e \rangle \Downarrow^{\gamma} \langle h_1, o_1 \rangle \qquad L, l \vdash \langle h_1, \text{while}_2(o_1, e)\{s, o\}\rangle \Downarrow \langle h_2, o_2 \rangle}{L, l \vdash \langle h, \text{while}_1(e)\{s, o\}\rangle \Downarrow \langle h_2, o_2 \rangle}$$

$$\textsc{While-2(true)}\quad \frac{\neg\text{False}(v) \qquad L, l \vdash \langle h, s \rangle \Downarrow^{\gamma} \langle h_1, o_1 \rangle \qquad L, l \vdash \langle h, \text{while}_3(e)\{s, o, o_1\}\rangle \Downarrow \langle h_2, o_2 \rangle}{L, l \vdash \langle h, \text{while}_2(v, e)\{s, o\}\rangle \Downarrow \langle h_2, o_2 \rangle}$$

$$\textsc{While-2(false)}\quad \frac{\text{False}(v)}{L, l \vdash \langle h, \text{while}_2(v, e)\{s, o\}\rangle \Downarrow \langle h, o \rangle}$$

$$\textsc{While-3(value)}\quad \frac{L, l \vdash \langle h, \text{while}_1(e)\{s, v\}\rangle \Downarrow \langle h_1, o_1 \rangle}{L, l \vdash \langle h, \text{while}_3(e)\{s, o, v\}\rangle \Downarrow \langle h_1, o_1 \rangle}$$

$$\textsc{While-3(return)}\quad L, l \vdash \langle h, \text{while}_3(e)\{s, o, \text{ret } v\}\rangle \Downarrow \langle h, \text{ret } v \rangle$$

$$\textsc{While-3(empty)}\quad \frac{L, l \vdash \langle h, \text{while}_1(e)\{s, o\}\rangle \Downarrow \langle h_1, o_1 \rangle}{L, l \vdash \langle h, \text{while}_3(e)\{s, o, \text{empty}\}\rangle \Downarrow \langle h_1, o_1 \rangle}$$

JSIL Code

Compared to the JSIL operational semantics, there are a few constructs here not represented above, such as the *fresh*() command, which creates a fresh JSIL variable to hold a return value.

The `::` symbol represents concatenation of commands, and the `goto` command has two different forms—`goto` *i* directs the flow of control directly to line *i*, and `goto [e]` *i, j* first evaluates e, and if the evaluation result is `true`, the flow of control will go to line *i*, otherwise to line *j*. The `+` and `-` notations in the `goto` commands represent whether the program flow should go down or go up a certain number of lines, respectively, and the `#` notation represents the number of lines that a certain command occupies, so $+\#cl_s$ means to go down the program the number of lines in cl_s. The JSIL program terminates when it reaches a `skip` command.

let $x_e = fresh();\ x_s = fresh();$

$cl_e = \mathcal{C}_m(e, x_e);\ cl'_e = \mathcal{W}_\gamma(x_e);$

$cl_s = \mathcal{C}_m(s, x_s);\ cl'_s = \mathcal{W}_\gamma(x_s);$

$e = (x_e = \text{false}) \mid\mid (x_e = \text{undefined}) \mid\mid (x_e = \text{null}) \mid\mid (x_e = \text{""})$

in

$x := \text{empty}$

$cl_e :: cl'_e$

`goto` $[e]\ [+\#cl_s + \#cl'_s + 4], [+1]$

$cl_s :: cl'_s$

`goto` $[x_s\ != \text{empty}]\ [+1], [+2]$

$x := X_s$

`goto` $[-\#cl_e - \#cl'_e - \#cl_s - \#cl'_s - 3]$

`skip`

2.4.2 Internal JavaScript Function: Get

An example of a JavaScript internal function that the `While` command uses is [[Get]], because during execution of `While`, variables must be dereferenced to obtain their values. Indeed, the *GetValue* command mentioned in step 2.b of `While` makes use of [[Get]] in its definition. [[Get]] itself depends on another internal function [[GetProperty]], which returns the property descriptor of the named property of this object or undefined [9]. This section follows the structure of the previous section, first describing the JavaScript standard, then the JSIL operational semantics, and finally JSIL code.

ECMAScript 5 Standard

When the [[Get]] internal method of *O* is called with property name *P*, the following steps are taken:

1. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, return **undefined**.
3. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
4. Otherwise, IsAccessorDescriptor(*desc*) must be true so, let *getter* be *desc*.[[Get]].
5. If *getter* is **undefined**, return **undefined**.
6. Return the result calling the [[Call]] internal method of *getter* providing *O* as the **this** value and providing no arguments.

ES5 Strict Operational Semantics

The formalization of [[Get]] also closely follows the ECMAScript standard. Here $_{\text{g}}$ refers to [[Get]] and $_{\text{gp}}$ refers to [[GetProperty]]. In G-getProp, the result of calling [[GetProperty]] with x is stored as o, which is then used as the input for [[Get]]. In G-propUndef, if the input for [[Get]] is undefined, then the returned result will be undefined. In G-propDefData, d is a data descriptor, and the return value v is the *Value* attribute of d. in G-propDefAccGetUndef, d is an accessor descriptor, and d's *Getter* attribute is undefined, so undefined is returned. And lastly in G-propDefAccGetDef, d is an accessor descriptor, its *Getter* attribute is defined and assigned as a_g, and a_g is called to return the final outcome o_f.

$$\textsc{G-getProp}\quad \frac{L, v_t \vdash \langle h, \mathcal{I}^o_{gp}(x)\rangle \Downarrow \langle h, o\rangle \qquad L, v_t \vdash \langle h, \mathcal{I}^o_{g}(o)_1\rangle \Downarrow \langle h_f, o_f\rangle}{L, v_t \vdash \langle h, \mathcal{I}^o_{g}(x)\rangle \Downarrow \langle h_f, o_f\rangle}$$

$$\textsc{G-propUndef}\quad L, v_t \vdash \langle h, \mathcal{I}^o_{g}(\text{desc undefined}_1)\rangle \Downarrow \langle h, \text{desc undefined}\rangle$$

$$\textsc{G-propDefData}\quad \frac{\mathcal{P}_{(dd)}(d) \qquad v = d.[[V]]}{L, v_t \vdash \langle h, \mathcal{I}^o_{g}(\text{desc } d)_1\rangle \Downarrow \langle h, v\rangle}$$

$$\textsc{G-propDefAccGetUndef}\quad \frac{\mathcal{P}_{(ad)}(d) \qquad d.[[G]] = \text{undefined}}{L, v_t \vdash \langle h, \mathcal{I}^o_{g}(\text{desc } d)_1\rangle \Downarrow \langle h, \text{undefined}\rangle}$$

$$\textsc{G-propDefAccGetDef}\quad \frac{\mathcal{P}_{(ad)}(d) \qquad a_g = d.[[G]] \neq \text{undefined} \qquad L, v_t \vdash \langle h, a_g()\rangle \Downarrow \langle h_f, o_f\rangle}{L, v_t \vdash \langle h, \mathcal{I}^o_{g}(\text{desc } d)_1\rangle \Downarrow \langle h_f, o_f\rangle}$$

JSIL Code

The JSIL translation faithfully follows the ECMAScript standard. The procedure *o__get* takes two parameters, the object itself and a property name. The [[GetProperty]] internal function is first located, then called. Since the property name may not exist, [[GetProperty]] is called with an error label, *elab*, to throw errors as necessary. Depending on whether [[GetProperty]] has returned undefined, *o__get* either returns or determines the descriptor type of the result. Descriptors are modeled as a list of elements, with the zeroth element being a character that signals the type of the descriptor. For instance, data descriptors have ‘d’ as the first element, accessor descriptors have ‘a’, and general descriptors have ‘g’. For data descriptors, the first element is the *Value* attribute, and for accessor descriptors, it is the *Getter* attribute. The `nth` function indexes the n^{th} element in a list. If the descriptor is a data descriptor, its value is returned; if it is an accessor descriptor and its *Getter* is undefined, undefined is returned. Otherwise, the program will find the scope of the descriptor’s *Getter*, and the location of the *Getter*’s [[Call]] method, before actually calling the *Getter* function and returning.

```
proc o__get (l, prop) {
      gp := [l, ‘‘@getProperty”];
      xret := gp (l, prop) with elab;
      goto [xret = $$undefined] rlab def;

  def:  d := nth (xret, 0);
      xret := nth (xret, 1);
      goto [d = ”d”] rlab acc;

  acc:  goto [xret = $$undefined] rlab get;
  get:  xsc := [xret, ‘‘@scope”];
      fun := [xret, ‘‘@call”];
      xret := fun (xsc, l) with elab;

  rlab: skip;
  elab: skip
}
with
{
    ret: xret, rlab;
    err: xret, elab;
};
```

Chapter 3

This Project and Concurrent Progress

Previous work has already been done on JSIL and the compiler, and this project seeks to expand JSIL to include descriptors, arguments object, and other library functions. Concurrently, tests against ECMAScript Test262 will be run to prove correctness for the implementation. For example, the memory model for both JavaScript and JSIL representations has already been changed to incorporate JavaScript descriptors, $\mathcal{D}_{JS}$, and the new JavaScript heap model will look like below, rather than the one presented in section 2.3.4.

JavaScript Heap Model
$\mathtt{h} \in \mathcal{H}_{JS} : \mathcal{L} \times \mathcal{X}_{JS} \rightharpoonup (\mathcal{V}_{JS} \cup \mathcal{S}ch_{JS} \cup \mathcal{D}_{JS})$
JSIL Heap Model
$\mathtt{h} \in \mathcal{H}_{JSIL} : \mathcal{L} \times \mathcal{S}tr \rightharpoonup \mathcal{V}_{JSIL}$

3.1 ES5 Strict and Core ES5 Strict

Strict mode was a concept introduced in the fifth version of the ECMAScript standard [9], and it presents slightly different semantics compared to the non-strict version in the following significant ways: in strict mode, the only object environment record is the global object, variables cannot get assignment without first being declared, and errors will always be thrown when needed. The scope of strict mode is within the specific code unit that imposes strict mode, so JavaScript programs could include both strict and non-strict code components [9].

Strict mode influences the implementation of the language in multiple ways. One example would be in resolving the naming reference of variables. An abstract operation, IsStrictReference(V) (V is a Reference), returns the strict reference component of V [9], and it is used in internal functions such as [[getValue]] and [[putValue]], like the [[Get]] internal function discussed in section 2.4.2.

The initial phase of this project seeks to complete the implementation of core strict mode, with core defined as the native language constructs and library functions that tap into these constructs. All other library functions that could be defined using what is already in core are not included in this initial stage but will be incorporated later on in the project.

3.2 Contrast JSIL Without and With Descriptors

Work by my supervisors to improve JSIL has been concurrently underway since I started this project, and most notably is the recent inclusion of descriptors into the memory model.

The following code snippets demonstrate the difference that descriptors can make in JSIL. On the left is JSIL code for [[Get]] without descriptors, and which will be referred to as *LHS-without*, and on the right, JSIL with descriptors as presented before, and will be referred to as *RHS-with*. Many internal functions make use of descriptors, thus without which the JSIL representation of JavaScript would simply be incorrect.

```
proc o__get(l, prop) {
        gp := [l, "@getProperty"];
    rlab: xret := gp (l, prop);
}
with
{
    ret: xret, rlab;
}
```

```
proc o__get (l, prop) {
      gp := [l, "@getProperty"];
      xret := gp (l, prop) with elab;
      goto [xret = $$undefined] rlab def;

  def:  d := nth (xret, 0);
      xret := nth (xret, 1);
      goto [d = "d"] rlab acc;

  acc:  goto [xret = $$undefined] rlab get;
  get:  xsc := [xret, "@scope"];
      fun := [xret, "@call"];
      xret := fun (xsc, l) with elab;

  rlab: skip;
  elab: skip
}
with
{
    ret: xret, rlab;
    err: xret, elab;
};
```

Recall section 2.4.2 presented the ECMAScript standard for the [[Get]] internal function. The JSIL implementation without descriptors is much shorter because all the steps in the standard that deal with descriptors are inapplicable. Starting from step 3 which states ”If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]]”, *LHS-without* simply cannot address the concept of descriptors, whereas *RHS-with* implements step 3 in the *def* code block. The three subsequent steps, ”Otherwise, IsAccessorDescriptor(*desc*) must be true so, let *getter* be *desc*.[[Get]]”, ”If *getter* is **undefined**, return **undefined**”, and ”Return the result calling the [[Call]] internal method of *getter* providing *O* as the **this** value and providing no arguments” also went unaddressed in *LHS-without*, but were implemented as the *acc* and *get* code blocks in *RHS-with*.

3.3 Scope and Aims

This project aims to complete the Object.prototype and Array libraries in their JSIL implementation. The Object.prototype library encompasses seven built-in functions for a total of two pages in the ECMAScript standard, while the Array library includes 28 built-in functions for a total of 20 pages. All of the functions in these libraries will be implemented on the JSIL level, and tested against the standard test suite, Test262, on a continually integrated basis.

Another topic of interest is the arguments object, which is initially described in section 10.6 and mentioned throughout section 15. Implementing the arguments object entails altering the compiler that translates ES5 strict code into JSIL, as well as testing against Test262 to ensure the correctness of the compiler.

3.4 Testing

The current testing framework includes the entire ECMAScript 6 test suite (ES6 Test262) which consists of 14998 test cases. Previous testing has applied several rounds of filtering to discard irrelevant test cases, including ES6-only features, non-strict functionality, non-core functionality, and so on, after which 4599 tests remain. Of these 4599, current implementation of JSIL covers 2388 test cases, and the rest of the 2211 tests need to be worked on. The exact number of test cases that this project will cover remains to be defined.

The interactive testing framework is available online from inside the Imperial network, but soon it will be integrated into a publicly accessible web address and will become continuously integrated into the JSIL project. At that point, each JSIL commit can be viewed from the testing portal, and the results can be compared with previous commits to determine whether the commit improved or worsened test coverage. Thus, JSIL development and testing will be done concurrently, and mistakes will be spotted and addressed as soon as they appear.

Chapter 4

JavaScript

4.1 Definition of Objects

JavaScript is a language that relies heavily and primarily on Objects, which are collections of **properties**, each with zero or more **attributes** that define the property [9]. Properties can hold the following types: Objects, primitive values, or functions, which are a type of special Objects. The JavaScript values and boolean values associated with each property are the attributes of the property. Each attribute has its own name and value. Essentially, properties can be seen as name-value pairs, with the names generally being primitive strings, and the values being one of the three types previously mentioned. Objects can be created via literal notation or via Object constructors that create and initialize the Objects, and then establishes the prototype chain for the Object [9]. Within an object, there are two types of properties - own properties and inherited properties. Own properties are those defined within the current object, and inherited properties are those defined in the object's prototype chain.

4.2 Categories of Properties

There are two types of properties in JavaScript - named properties and internal properties [9]. Named properties, by definition, associate user-defined names with JavaScript constructs, called descriptors, whereas internal properties do not have user-defined names. Data properties associate a name with a JavaScript value, as well as a set of boolean values that specify whether this property's value can be altered, whether the property is used in for-in enumeration, and whether it can be deleted or changed into an accessor property. Accessor properties associate a name with two JavaScript functions, a getter and a setters, in addition to a similar set of boolean values that specify whether the property is used in for-in enumeration and whether it can be deleted or changed into a data property. The getter and setter are used to retrieve or store, respectively, the JavaScript value associated with the property. All objects share a set of twelve internal properties that get defined when the objects are created, including the prototype property mentioned above, and the object's class, whether more properties can be added to the ojbect, the default value of the object, and so on. There are an additional twelve internal properties that only apply to certain types of objects, for example, Function Objects alone always have the [[Call]] internal method.

4.2.1 Descriptors

4.2.2 Attributes

4.3 Prototype Inheritance

As mentioned in the background section, rather than using class-based inheritance, JavaScript utilizes prototype inheritance. Each object has a property named **prototype** in order to implement prototype-based inheritance [9]. When an object is created, the new object's prototype property is implicitly set to the value of the object constructor's prototype. Alternatively, an object can also be created with an explicitly stated prototype. In turn, the new object may become the prototype of another object, and thus resulting in many layers of prototype referencing, called the prototype chain [9]. When a property name is referenced, if the current object cannot resolve the name reference, then the program will find the object's prototype, and attempt to resolve the name reference in the prototype. The process will continue until the property name reference is resolved, or until the prototype points to null. If two objects share the same prototype with a certain property that neither object possesses, then the two share this property through prototype inheritance. This method of inheritance makes JavaScript objects more flexible than traditional class-based objects, some of which can only implement one level of inheritance.

4.4 Initial Heap

4.5 Function Objects

What does it mean to call a function? How to construct a function?

Chapter 5

JSIL

5.1 Starting JSIL Syntax

The syntax of JSIL at the beginning of this project is as follows:

$$
\begin{array}{rlrl}
Variables: & \mathtt{x} \in \mathcal{X}_{JSIL} & & \\
Types: & \mathtt{t} \in \mathtt{Type} & ::= & \mathtt{Num} \mid \mathtt{Bool} \mid \mathtt{Str} \mid \mathtt{Undef} \mid \mathtt{Ref}_o \mid \mathtt{Ref}_v \mid \mathtt{Obj} \\
Values: & \mathtt{v} \in \mathcal{V}_{JSIL} & ::= & \mathit{lit} \mid \mathit{l} \mid \mathit{r} \mid \mathtt{empty} \mid \mathtt{error} \mid \mathtt{t} \\
Expressions: & \mathtt{e} \in \mathcal{E}_{JSIL} & ::= & \mathtt{v} \mid \mathtt{x} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{typeof}(\mathtt{e}) \mid \mathtt{ref_o}(\mathtt{e},\mathtt{e}) \mid \\
& & & \mathtt{ref_v}(\mathtt{e},\mathtt{e}) \mid \mathtt{base}(\mathtt{e}) \mid \mathtt{field}(\mathtt{e}) \\
Commands: & \mathtt{c} \in \mathtt{Cmd} & ::= & \mathtt{skip} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{x} := \mathtt{new}() \mid \mathtt{x} := [\mathtt{e},\mathtt{e}] \mid [\mathtt{e}_1,\mathtt{e}_2] := \mathtt{e}_3 \mid \\
& & & \mathtt{delete}(\mathtt{e},\mathtt{e}) \mid \mathtt{x} := \mathtt{hasField}(\mathtt{e},\mathtt{e}) \mid \mathtt{goto}\ \mathit{i} \mid \\
& & & \mathtt{goto}\ [\mathtt{e}]\ \mathit{i}, \mathit{j} \mid \mathtt{x} := \mathtt{protoField}(\mathtt{e},\mathtt{e}) \mid \\
& & & \mathtt{x} := \mathtt{protoObj}(\mathtt{e},\mathtt{e}) \mid \mathtt{x} := \mathtt{e}(\overline{e})\ \mathtt{with}\ \mathit{j} \\
Procedures: & \mathtt{proc} \in \mathtt{Proc} & ::= & \mathtt{proc}\ \mathit{m}(\overline{x})\{\overline{c}\}
\end{array}
$$

where lit denotes JavaScript literals (undefined, null, numbers, booleans, and strings), l denotes locations, and r denotes references. `typeof` returns the type of its argument; $\mathtt{ref_o}$ and $\mathtt{ref_v}$ are used to construct object and variable references, respectively; `base` and `field` return, respectively, the first and second component of the reference that is passed to them as a parameter. $\mathtt{x} := \mathtt{protoField}(\mathtt{e}_1, \mathtt{e}_2)$ assigns object $\mathtt{e}_1$'s field $\mathtt{e}_2$ to `x` and $\mathtt{x} := \mathtt{protoObj}(\mathtt{e}_1, \mathtt{e}_2)$ assigns `true` to `x` if object $\mathtt{e}_2$ is in the prototype chain of object $\mathtt{e}_1$ and `false` otherwise.

$\mathtt{x} := \mathtt{e}(\overline{e})\ \mathtt{with}\ j$ is a dynamic procedure call. When executed, the procedure will evaluate the expression `e` with the arguments $\overline{e}$ as provided, and assign the resulting value to `x`. If evaluating the procedure results in an error, then flow control will proceed to the code block labeled by *j*, otherwise, flow control will continue to the next line in the program. This dynamic procedure call syntax is commonly used in the implementation component in this project, as it is necessary to evaluate results for when internal functions are called.

A top level JSIL program follows the syntax of proc m$(\overline{x})(\overline{c})$, which executes procedure `m` with $\overline{x}$ as formal parameters and $\overline{c}$ as a list of JSIL commands. The implementation stage of this project defines a number of JavaScript library functions, and each function follows this syntax of top level program. JSIL programs do not have explicit return statements, rather, they contain a return label and optionally an error label, usually denoted as *rlab* and *elab*, respectively. When the program flow of control reaches one of these labels, the corresponding return variable, a normal value x_{ret} or an error value x_{err}, will be returned by the procedure.

5.2 Final JSIL Syntax

Components of this syntax have evolved since the start of the project, and the differences are as follows:

5.3 Semantics

Since JSIL procedures are top level,

5.4 Heap Model

The JSIL Heap Model is designed to correspond as closely as possible to the JavaScript Heap Model (beta indistinguishability relation)

JavaScript Heap Model
$\mathtt{h} \in \mathcal{H}_{JS} : \mathcal{L} \times \mathcal{X}_{JS} \rightharpoonup (\mathcal{V}_{JS} \cup \mathcal{S}ch_{JS} \cup \mathcal{D}_{JS})$
JSIL Heap Model
$\mathtt{h} \in \mathcal{H}_{JSIL} : \mathcal{L} \times \mathcal{S}tr \rightharpoonup \mathcal{V}_{JSIL}$

Chapter 6

Array Library

6.1 Arrays in JavaScript

Array Objects are a type of built-in Object in the JavaScript language, and they have predefined methods and special properties.

6.1.1 Indices and Length

An Array in JavaScript is a special type of Object, referred to as Array Objects. Array Objects differ from normal Objects in the treatment of property names. Given a property name P, ToUint32(P) converts P into an unsigned 32-bit integer, and ToString(ToUint32(P)) converts this integer back to a string value. If the unsigned integer does not equal $2^{32\text{-}1}$, which is the largest value an unsigned 32-bit integer can be, and if this converted string value is equal to the original string, P, then this property name is considered to be an index in the Array Object [9]. Thus, P has to be a string value containing non-negative integer values, otherwise performing the two operations would not result in the same string value.

If a property's name is an Array index, then this property is commonly referred to as an element. An element can be Objects, primitive values, or functions, and the elements within an Array Object do not have to be of the same type. By convention, JavaScript Arrays are zero-indexed, meaning that the first element in an Array Object has index equals zero, the second equals one, and the last equals the length of the Array Object minus one. The length of an Array Object is stored as a special property of the Array Object, with its name as "length", and its value as an integer that measures the number of elements, including empty elements, inside the Array. This integer must be non-negative and less than 2^{32}, which restricts the size of the maximum possible Array Object.

An interesting feature of JavaScript Array length that possibly sets it apart from Array implementations in other languages, however, is that Array elements can contain absolutely nothing, and hence be completely empty, so the length value technically does not reflect the number of actual elements in the Array. Rather, the length value is simply the integer that is 1 greater than the largest property name in the Array Object. When altering the Array Object, such as adding or deleting elements, the length property's value must change accordingly. When an element is added, length will be changed to the element's array index plus 1, and when an element is deleted, length will be changed to the maximum array index plus 1. If the length property itself is changed, then all elements whose array indices are larger than the length value minus 1 will be deleted. Hence, the length invariant will be maintained in all Array operations [9].

6.1.2 DefineOwnProperty

Array Objects also differ from other Objects in their implementation of the [[DefineOwnProperty]] internal function. [[DefineOwnProperty]] takes in the following parameters - a property name P, a property descriptor Desc, and a boolean value Throw for whether an error would be thrown given TypeError exceptions. The purpose of this internal function is to "create or alter the named own property to have the state described by a Property Descriptor" and to "test various fields of the Property Descriptor Desc for specific values" If the field is "absent from Desc then its value is considered to be false" [9].

[[DefineOwnProperty]] allows the Array Object to change the value of a particular element and to add new elements. Many Array Library functions make use of the internal function [[Put]], which in turn calls [[DefineOwnProperty]]. [[DefineOwnProperty]] works by getting the original length property's value of the Array Object, then performs different actions based on whether the property P argument is the **length** property or an array index.

Length If property P is the length property, one of the following scenarios will occur.

- The descriptor Desc's value attribute is absent, and the default [[DefineOwnProperty]] internal function will be called with "length" as the property.
- The value attribute of the descriptor Desc is not a number, so an error will be thrown.
- The value attribute of the descriptor Desc is larger than the length value of the Array Object, and the length property can be altered, the default [[DefineOwnProperty]] internal function will be called with "length" as the property.
- The value attribute of the descriptor Desc is larger than the length value of the Array Object, and the length property can not be altered, and an error will be thrown.
- Calling the default [[DefineOwnProperty]] internal function will be called with "length" as the property returns false, so the function returns false.
- The length property's new value is less than the old value, all elements in the Array Object with an index greater than the new length value will be deleted. If the deletion function encounters any errors, the default [[DefineOwnProperty]] internal function will be called with "length" as the property, and an error will be thrown.
- The descriptor Desc can not be altered, and the default [[DefineOwnProperty]] internal function will be called with "length" as the property, [[Writable]]:false as Desc, and false for Throw.
- Calling the default [[DefineOwnProperty]] internal function will be called with "length" as the property returns true, the descriptor Desc can be altered, and the length property's new value is greater than or equal to the old value, the value true is returned.

Index On the other hand, if property P is an array index, the function proceeds to determine whether P's unsigned 32-bit numerical value is greater than or equal to the value of the Array's length property, and whether the length property can not be altered. In the case that both conditions hold, [[DefineOwnProperty]] will fail and throw an error because no new element can be added to the Array Object. If one of the conditions does not hold, the function will call the default [[DefineOwnProperty]] internal function using the same arguments. If the default internal function does not succeed, the function will return false. If the first condition holds, the length value will be increased by one after the function call, and it will return true.

Default In the event that [[DefineOwnProperty]] has not yet produced a return value, the default [[DefineOwnProperty]] internal function for general Objects will be called with the same arguments provided.

6.1.3 Array Creation

An Array Object can be created in one of three ways - using an Array literal, using the Array Object constructor, and using the constructor as a function. Array literal notation utilizes square brackets to enclose the elements, and is used in the following manner:

```
var arr = [’a’, ’b’, 1, 2, true, false];
```

The array indices do not have to be specified because they start at zero by default, and increases by 1 for all subsequent elements. The Array Object constructor can take one of the following forms:

```
var arr1 = new Array([item0 [, item1 [, ...]]]);

var arr2 = new Array(len);
```

The first version of the constructor can be used with zero, one, or many arguments. If no argument is provided, it will create an empty Array Object and initialize the Object with default properties such as ”class” and ”length”. If more than one argument is provided, the Array Object will add each argument as an element in the order they are given. All the elements are treated as data properties, with the value attribute as the items provided, and the boolean value **true** for all the other attributes. The length property’s value will be set to the number of arguments given [9].

The second version of the constructor works when only one argument is given. If the argument is a JavaScript Number, and its unsigned 32-bit version is equal to itself, then the length property’s value will be set to the argument provided. If the argument is a negative JavaScript Number, an error will be thrown. If one argument is provided, and the argument is not a Number, then an Array Object with one element will be created, and default properties will be initialized as well. As the previous constructor, the element is added as a data property, with the value attribute as the item provided, and the boolean value **true** for the other attributes.

Both constructors will set the following internal properties of the Array Object

- Its [[Prototype]] will be the original Array prototype object
- Its [[Class]] will be the JavaScript String ”Array”
- Its [[Extensible]] will be the boolean value **true**

The Array Object constructor itself is a JavaScript Object, and its prototype property is set to the Function Object prototype, which will not be discussed in further detail here.

6.1.4 A Working Example

```
var arr = new Array(’a’,’b’);
arr[2] = ’c’;
arr[5] = ’f’;
var len = arr.length;
Object.preventExtensions(arr);
arr[6] = ’g’;
len = arr.length;
```

Here, a new Array Object named arr with elements ”a” and ”b” is created on line 1, a new element ”c” is added as the third element on line 2, and a new element ”f” is added as the sixth element on line 3. Line 4 gets the value of the length property of Array arr and assigns it to a variable named len. At this point, the value of len is 6, even though only four elements have been added to Array arr, and the fourth and fifth elements of Array arr are currently undefined but could be altered at a future stage.

Line 5 changes the [[Extensible]] internal property of Array arr to **false**, so no new elements can be added to arr. As a result, if line 6 tries to add a seventh element to Array arr, the length value of arr stays at 6.

6.2 Array Library Functions

Below is an outline of all of the functions in the ECMAScript 5 Array Library. Each function’s name will be listed then followed by a brief explanation of the intended usage of the function.

6.2.1 Array Object Constructor Functions

The following functions belong to the Array Object constructor, and the first two have been described in detail in 6.1.3.

1. new Array([item0 [, item1 [, ...]]])
2. new Array(len)
3. Array([item0 [, item1 [, ...]]])
   This is using the Array constructor as a function, i.e., without using the **new** keyword, but it functions the same way as the constructor when used with **new**.
4. Array.prototype
   This property’s initial value is Array Prototype Object, and this property’s value cannot be altered, it cannot be used in for-in enumeration, and its attributes cannot be changed. The [[prototype]] internal property is the default Object Prototype.
5. Array.isArray(arg)
   This method checks to see if the [[class]] internal property of arg is equal to the String ”Array”.

6.2.2 Array Object Prototype Functions

The following functions belong to the Array Object prototype:

1. Array.prototype.constructor
   This is the constructor previously explained.
2. Array.prototype.toString()
   This method utilizes the Array.prototype.join(separator) function below, to produce a String representation of all the elements inside the Array.
3. Array.prototype.toLocaleString()
   This method also returns a String representation of all the elements inside the Array, as Array.prototype.toString(). The main differences however, are the locale specific conversion

of each element into Strings using the element's own toLocaleString method and the use of a locale specific and implementation-defined separator between the element strings. This difference becomes especially relevant when the Array Object holds Date Objects, since the date concept is represented differently in different countries.

4. Array.prototype.concat([item1 [, item2 [, ...]]])
This method creates a new Array Object that holds all the elements of the original Array Object in the original order, then appends the optional arguments at the end and returns the new Array.

5. Array.prototype.join(separator)
This method converts all the elements into Strings and concatenates all the Strings together using the separator. If no separator is provided, the default separator is the comma.

6. Array.prototype.pop()
This method removes the last element of the Array and returns it. The original Array Object is modified, and its length property deceases in value by one.

7. Array.prototype.push([item1 [, item2 [, ...]]])
This method modifies the original Array by appending the optional arguments at the end in the order given, and calculates the new length property value of the Array.

8. Array.prototype.reverse()
This method reverses the original order of the Array. No new Array Object is created and the original Array is modified from this method. The algorithm used is swapping the elements at each end until meeting in the middle.

9. Array.prototype.shift()
This method is the opposite of the Array.prototype.pop() method, in that it removes the first element of the Array, returns it, and decreases the length by one. This method, however, requires all subsequent elements in the Array to be shifted up by one, since Arrays need to start at index zero. Each element is shifted by getting the current element value and associating it with the previous index value until reaching the last index, which is then deleted from the Array.

10. Array.prototype.slice(start, end)
This method creates a new Array Object, and adds into it the original Array elements beginning at index **start** until the element before **end**. If the end argument is undefined, it will default to through the end of the Array Object, and if the start index is larger than the end index, the new Array will have no elements. Both arguments provided can be negative, and if so, the length of the original Array will be added to the argument. This method does not alter the original Array.

11. Array.prototype.sort(comparefn)
This method will be explained in detail in 6.2.4.

12. Array.prototype.splice(start, deleteCount, [item1 [, item2 [, ...]]])
This method, similar to Array.prototype.slice(start, end), will create a new Array Object with **deleteCount** number of elements beginning at index **start**. The major differences, however, are that splice actually deletes those elements from the original Array whereas slice does not, and that splice takes optional arguments that could be added in the place of the deleted elements in the order they are given. The **start** argument can be negative, and if so, the

length of the original Array will be added. If the **deleteCount** argument is negative, it will default to zero, and if it is too large, it will delete all the Array elements after index start.

13. Array.prototype.unshift([item1 [, item2 [, ...]]])
    This method is the opposite of the Array.prototype.push([item1 [, item2 [, ...]]]) method, in that it adds Array elements at the start of the Array rather than at the end. The original elements' indexes will be shifted backwards by the number of arguments, then the elements will be added in the front of the Array in the order they are given in the arguments list,

14. Array.prototype.indexOf(searchElement [, fromIndex])
    This method compares the **searchElement** argument to each of the elements in the Array in ascending order, either from the zero index, or the **fromIndex** if it is given as an argument. If searchElement is found, the index of the first instance is returned, otherwise -1 is returned. If **fromIndex** is provided and is greater than or equal to the length of the Array, the Array will not be searched; if it is negative, the length of the Array will be added to it to get the actual fromIndex, and if it is still negative, the entire Array will be searched.

    This method makes use of the internal Strict Equality Comparison Algorithm to determine whether searchElement and an Array element are the same.

15. Array.prototype.lastIndexOf(searchElement [, fromIndex])
    This method is similar to Array.prototype.indexOf(searchElement [, fromIndex]), except it starts the search from the end of the Array, rather than the beginning. If **fromIndex** is provided and greater than or equal to the length of the Array, the whole Array will be searched; if it is negative, length of the Array will be added to it to get the actual fromIndex, and if it is still negative, the entire Array will not be searched.

16. Array.prototype.every(callbackfn [, thisArg])
    This method requires a callback function to be provided as an argument, and an optional argument to be used for the **this** value of the callback function. The method calls the callback function for each of the elements in the Array in ascending order, and will return either when the callback function returns **false** or when the entire Array is traversed.

    The callback function must take three arguments, namely the value and index of each Array element, and the Array Object that calls it. Its return value must be coercible to a boolean value, to be passed back to the Array method.

    This method is used to determine whether every single element in an Array satisfies certain criteria laid out in the callback function.

17. Array.prototype.some(callbackfn [, thisArg])
    This method is similar to Array.prototype.every(callbackfn [, thisArg]), but rather than returning upon a **false** return value from the callback function, it will return upon a **true** return value, or when the entire Array is traversed. This method is used to determine whether at least one element in the Array satisfies certain criteria laid out in the callback function.

18. Array.prototype.forEach(callbackfn [, thisArg])
    This method requires a callback function to be provided as an argument, and an optional argument to be used for the **this** value of the callback function. The method calls the callback function for each of the elements in the Array in ascending order, and will return only when the entire Array is traversed.

    The callback function must take the same three arguments as outlined previously, but its return value does not need to be coercible to a boolean value.

This method is used to apply a callback function to each of the elements in the Array, but it should only be used when no return value is required from this operation.

19. Array.prototype.map(callbackfn [, thisArg])
    This method is similar to Array.prototype.forEach(callbackfn [, thisArg]) in that it traverses the entire Array and calls the callback function on each element in the Array. The main difference is that this method creates a new Array Object, and stores in it the return values of the callback function. This method will return the new Array when it completes, so it should be used when a return Array is required from this operation.

20. Array.prototype.filter(callbackfn [, thisArg])
    This method will be explained in detail in 6.2.3.

21. Array.prototype.reduce(callbackfn [, thisArg])
    This method requires a callback function to be provided as an argument, and an optional argument to be used for the initial value of the callback function. The method calls the callback function for each of the elements in the Array in ascending order, and will return a single value when the entire Array is traversed.

    The callback function must take four arguments, namely the value from the previous call to callbackfn (or the initial value if provided as an argument), the value and index of each current Array element, and the Array Object that calls it.

22. Array.prototype.reduceRight(callbackfn [, thisArg])
    This method is similar to Array.prototype.reduce(callbackfn [, thisArg]), but rather than traversing through the Array in ascending order, it traverses in descending order.

6.2.3 Filter In Detail

This method requires a callback function to be provided as an argument, and an optional argument to be used for the **this** value of the callback function. The method creates a new Array Object, calls the callback function for each of the elements in the original Array in ascending order and stores the element for which the callback function has returned **true** in the new Array, and lastly returns after the entire original Array is traversed.

The callback function must take three arguments, namely the value and index of each Array element, and the Array Object that calls it. Its return value must be coercible to a boolean value, to be passed back to the Array method.

This method is used to filter out all elements in an Array that satisfy certain criteria laid out in the callback function. It is similar to both Array.prototype.every(callbackfn [, thisArg]) and Array.prototype.some(callbackfn [, thisArg]), but rather than only providing a boolean return value, it actually shows all the successful elements.

ECMAScript Standard

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.

2. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **"length"**.

3. Let *len* be ToUint32(*lenValue*).

4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.

5. If *thisArg* was supplied, let T be *thisArg*; else let T be **undefined**.
6. Let A be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
7. Let k be 0.
8. Let to be 0.
9. Repeat, while k ¡ *len*
   a. Let *Pk* be ToString(k).
   b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
   c. If *kPresent* is **true**, then
      i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
      ii. Let *selected* be the result of calling the [[Call]] internal method of *callbackfn* with T as the **this** value and argument list containing *kValue*, k, and *O*.
      iii. If ToBoolean(*selected*) is **true**, then
         A. Call the [[DefineOwnProperty]] internal method of A with arguments ToString(*to*), Property Descriptor [[Value]]: *kValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**, and **false**.
         B. Increase *to* by 1.
   d. Increase k by 1.
10. Return A.

Pretty Big Step Operational Semantics

JSIL Code

```
proc AP_filter() {
      arguments := args;
      vthis  := nth (arguments, 1);
      cbf := nth (arguments, 2);

      num := length (arguments);

      xret := "i__toObject" (vthis) with elab;
      vthis := xret;

      g := [vthis, "@get"];
      xret := g (vthis, "length") with elab;
      xret := "i__toUint32" (xret) with elab;
      len := xret;

      xret := "i__isCallable" (cbf);
      goto [xret] cont throw;

  cont: goto [num <= 3] undef def;

  def:  t := nth (arguments, 3);
```

```
      goto seta;
  undef:  t := $$undefined;

  seta: xret := "Array_construct" () with elab;
      A := xret;
      hp := [vthis, "@hasProperty"];
      k := 0;
      to := 0;
  loop: goto [k < len] next end;

  next: xret := "i__toString" (k) with elab;
      pk := xret;
      xret := hp (vthis, pk) with elab;
      kpres := xret;
      goto [kpres] tt ff;

  tt:   xret := g (vthis, pk) with elab;
      kval := xret;
      scp := [cbf, "@scope"];
      fun := [cbf, "@call"];
      xret := fun (scp, t, kval, k, vthis) with elab;
      sel := xret;
      xret := "i__toBoolean" (sel) with elab;
      goto [xret] rett ff;

  rett: xret := "i__toString" (to) with elab;
      strt := xret;
      xret := "a__defineOwnProperty" (A, strt, {{ "d", kval, $$t,
    $$t, $$t }}, $$f) with elab;
      to := to + 1;

  ff:   k := k + 1;
      goto loop;

  end:  xret := A;
  rlab: skip;

  throw:  xret := "TypeError" ();
  elab: skip
}
with
{
  ret:  xret, rlab;
  err:  xret, elab;
};
```

6.2.4 Sort In Detail

The Array.prototype.sort(comparefn) method is undoubtedly the most complicated in the Array Library. If the Array Object falls into any of the four predefined scenarios, then an implementation defined algorithm will be used to sort the Array. If an Array is not a special Array, then the method will use the [[Get]], [[Put]], and [[Delete]] internal methods and the **sortCompare** method in an implementation defined manner to sort the Array. JSIL, like all other JavaScript implementations, has a choice of which sorting algorithm to utilize.

Sorting Algorithms

Sorting a list of elements has long been a favorite topic of study for computer scientists, and many sorting algorithms currently exist. Some algorithms prefer memory efficiency, while others prefer runtime efficiency.

One of the most widely used sorting algorithms is Quicksort. This algorithm takes a list, and randomly chooses a pivot element. It then finds the correct list index for the pivot and proceeds to sort all the elements below the pivot and above the pivot independently and recursively.

Algorithm 1 Non-Recursive Quicksort Algorithm

Require: a is an Array, cmp is a compare function
  if $cmp == undefined$ **then**
    use $sortCompare$ algorithm
  end if
  if length of $a < 2$ **then**
    $return$
  end if
  Create a new Array named $stack$
  Push 0 onto $stack$
  Push (length of a - 1) onto $stack$
  Let $size = 1$
  while $size > 0$ **do**
    Pop from $stack$ and assign to end
    Pop from $stack$ and assign to $start$
    $--size$
    Let el equal $a[end]$
    Let l equal $start$
    Let $current$ equal $start$
    while $current < end$ **do**
      Assign $a[current]$ to tmp
      if $cmp(current, end) < 0$ **then**
        Assign $a[l]$ to $a[current]$
        Assign tmp to $a[l]$
        $++l$
      end if
      $++current$
    end while
    Assign $a[l]$ to $a[end]$
    Assign el to $a[l]$
    if $end > l + 1$ **then**
      Push $(l + 1)$ onto $stack$
      Push end onto $stack$
      $++size$
    end if
    $--l$
    if $l > start$ **then**
      Push $start$ onto $stack$
      Push l onto $stack$
      $++size$
    end if
  end while

Chapter 7

Arguments Object

Like the Array Object, the Arguments Object is also a special type of JavaScript Object whose prototype is is the standard built-in Object prototype and whose [[Class]] internal property's value is "Arguments". It is used to capture all the arguments provided to a function call, and associates the actual arguments provided to the function to named variable references.

The *CreateArgumentObject* operation, called when the Function Objects execute, is what creates the Arguments Object, and it differs quite drastically in its treatment of code written in strict mode versus non-strict mode.

7.1 Full ES5

7.1.1 Internal Methods

The non-strict version of *CreateArgumentObject* makes use of the following abstract operations. Both take a String $name$ and an environment record env as input arguments. The exact manner in which Function Objects are created will not be discussed in further detail here.

$MakeArgGetter$ Creates a Function Object that returns the value bound for $name$ in env [9].

$MakeArgSetter$ Creates a Function Object that sets the value bound for $name$ in env [9].

In addition, the [[Get]], [[GetOwnProperty]], [[DefineOwnProperty]], and [[Delete]] internal methods in the Arguments Object are also overridden with special versions as compared to the default Object.

7.1.2 *CreateArgumentObject* Algorithm

JavaScript distinguishes between a Function Object's formal parameters and the actual arguments passed into a function call. When *CreateArgumentObject* is first called, it stores the number of actual arguments passed into *len*, creates the new Arguments Object, and sets its "length" property to have a value of *len*. It then creates a new Object named *map* and an empty list named *mappedNames*. It then starts at the end of the arguments list provided to examine one element at a time. It will call [[DefineOwnProperty]] on the Arguments Object and create one new property for each argument - the property name will be the index of the argument in the list, and the property value will be the argument provided.

len may be different from the number of formal parameters that the function takes, in which case the following set of commands would only execute when the index of the provided argument is less than the number of formal parameters. The element at the current index of the formal parameters list will be added as an element in *mappedNames*, and this element and the current environment record are provided as input to *MakeArgGetter* and *MakeArgSetter*. The results of these internal operation calls will be stored as the getter and setter attributes of the new property added to the *map* object with the current index as the property name.

If any arguments corresponding to a formal parameter is added in *mappedNames*, then the *map* Object will be added as the value of the [[ParameterMap]] internal property of the Arguments Object, and the internal methods in 7.1.1 will be redefined to the overridden versions.

Lastly, a new property named "callee" will be added to the Arguments Object, with the value being the Function Object instance that was called originally, and the Arguments Object itself is returned when the internal operation finishes.

7.2 ES5 Strict

7.2.1 *CreateArgumentObject* Algorithm

The algorithm of *CreateArgumentObject* starts off the same way for strict code as for non-strict code. They begin to differ, however, with regards to adding elements into the *mappedNames* list. In strict mode, no elements would be added to *mappedNames*, and thus as *mappedNames* is always empty, the [[ParameterMap]] internal property of the Arguments Object will not be set to the *map* Object, and the internal methods in 7.1.1 will not be redefined to the overridden versions.

Then, instead of only defining a new "callee" property in the Arguments Object, strict code will perform the following actions. Two separate properties, "caller" and "callee", will be defined in the Arguments Object, with the getter and setter attributes set to the built-in [[ThrowTypeError]] Function Object. This means that in strict mode, no getter and setter could be defined given a formal parameter element and the current environment record. This is reasonable because no formal parameters are associated with elements in the *mappedNames* list.

7.2.2 JSIL Implementation

Since this project works exclusively in strict mode, the JSIL implemented corresponds to the strict mode algorithm, and is hence relatively simple.

```
proc create_arguments_object (argList) {

    len := length (argList);

    (* Create the arguments object *)
    xret := new ();
    xret := "create_default_object" (xret, $lobj_proto, "Arguments", $$t);
    obj  := xret;

    (* Define length *)
    dop := [obj, "@defineOwnProperty"];
    xret := dop (obj, "length", {{ "d", len, $$t, $$f, $$t }}, $$f) with elab;

    (* Loop through values *)
    indx := len - 1;

  loop: goto [0 <= indx] head call;
  head: xret := "i__toString" (indx) with elab;
```

```
    xret := dop (obj, xret, {{ "d", nth (argList, indx), $$t,
  $$t, $$t }}, $$f) with elab;
    indx := indx - 1;
    goto loop;

    (* Set caller and callee *)
  call: [obj, "caller"] := {{ "a", $lthrow_type_error,
  $lthrow_type_error, $$f, $$f }};
    [obj, "callee"] := {{ "a", $lthrow_type_error,
  $lthrow_type_error, $$f, $$f }};

  rlab: xret := obj;
  elab: skip
}
with
{
  ret:  xret, rlab;
  err:  xret, elab;
}
```

7.3 A Working Example

A function, *myFunc* is defined as below. *myFunc* takes three formal parameters, named x, y, and z, and it returns the sum of the three parameters.

```
var myFunc = function(x, y, z) {
  return x + y + z;
}
```

In JavaScript, however, *myFunc* can be called with three arguments, less than three arguments, or more than three arguments, with each scenario outlined below.

```
\\ Scenario A
var a = myFunc(1, 2, 3);

\\ Scenario B
var b = myFunc(1, 2);

\\ Scenario C
var c = myFunc(1, 2, 3, 4, 5);
```

In non-strict code, the Arguments Object would have the following new properties with "name": value - "4": 5, "3": 4, "2": 3, "1": 2, and "0": 1. *mappedNames* list would contain ["z", "y", "x"]. The *map* Object would have the following properties with "name": getter: setter - "2": getz: setz, "1": gety: sety, and "0": getx: setx. In strict code, *mappedNames* and *map* would be empty.

Chapter 8

String Library

Chapter 9

Testing

9.1 Array Library

9.2 Arguments Object

9.3 String Library

Chapter 10

Conclusion

Bibliography

[1] Wala - t.j. watson libraries for analysis. pages

[2] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javascript specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 87–100, New York, NY, USA, 2014. ACM. pages

[3] Denis Bogdanas and Grigore Roşu. K-java: A complete semantics of java. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, New York, NY, USA, 2015. ACM. pages

[4] Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, March 2013. pages

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007. pages

[6] Julian Dolby and Manu Sridharan. Static and dynamic program analysis using wala. online, 2010. pages

[7] S Drossopoulou and S Eisenbach. *Towards an Operational Semantics and Proof of Type Soundness for Java*. Springer-Verlag, 1999. pages

[8] Sophia Drossopoulou and Susan Eisenbach. Towards an operational semantics and proof of type soundness for java. *Formal Syntax and Semantics of Java*, 1523, 1998. pages

[9] ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011. pages

[10] ECMA International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, 2015. pages

[11] Chucky Ellison and Grigore Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, New York, NY, USA, 2012. ACM. pages

[12] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 31–44, New York, NY, USA, 2012. ACM. pages

[13] Phillipa Gardner. Models of computation, lecture 2. University Lecture, 2010. pages

[14] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag. pages

[15] Yuri Gurevich and James K. Huggins. The semantics of the c programming language. In *Selected Papers from the Workshop on Computer Science Logic*, CSL '92, pages 274–308, London, UK, UK, 1993. Springer-Verlag. pages

[16] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993. pages

[17] Robert Harper and Christopher Stone. Proof, language, and interaction. chapter A Type-theoretic Interpretation of Standard ML, pages 341–387. MIT Press, Cambridge, MA, USA, 2000. pages

[18] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proc. 21st International Symposium on Software Testing and Analysis (ISSTA)*, July 2012. pages

[19] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006. pages

[20] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of standard ml. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 173–184, New York, NY, USA, 2007. ACM. pages

[21] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. pages

[22] Ben Livshits. JSIR – an intermediate representation for JavaScript analysis, May 2016. pages

[23] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 307–325, Berlin, Heidelberg, 2008. Springer-Verlag. pages

[24] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. *Compiler Construction: 11th International Conference, CC 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings*, chapter CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, pages 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. pages

[25] Michael Norrish. C formalised in hol. Technical Report UCAM-CL-TR-453, University of Cambridge - Computer Laboratory, December 1998. pages

[26] Daejun Park, Andrei Stefănescu, and Grigore Roşu. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 346–356, New York, NY, USA, 2015. ACM. pages

[27] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989. pages

[28] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University. pages

[29] A. M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000. pages

[30] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 1–16, New York, NY, USA, 2012. ACM. pages

[31] Grigore Rosu. Big-step structural operational semantics (big-step sos). University Lecture, 2011. pages

[32] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. pages

[33] Traian Florin Şerbănuţa, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The K primer (version 3.2). Technical report. pages