A Reference Implementation of ECMAScript Built-in Objects

(extended abstract of the MSc dissertation)

David Manuel Sales Gonçalves
david.s.goncalves@tecnico.ulisboa.pt
Instituto Superior Técnico, Universidade de Lisboa
Portugal

Abstract

ECMAScript (ES), commonly known as JavaScript, is one of the most important programming languages today because it is the *de facto* option when it comes to dynamic front-end web development. Throughout the years, ES has become an increasingly complex language, making it a difficult target for static analyses. This project is part of a wider project that aims to build a trustworthy reference interpreter for ES, which will, among other things, enable the development of precise static analysis tools for modern ES applications through the use of a novel intermediate language called ECMA-SL. This work focuses on implementing three built-in objects of ECMA-262 Edition 5.1: Array (15.4), RegExp (15.10), and JSON (15.12). It also implements a few methods of the String (15.5) built-in object and implements the Promise (25.4) built-in object of the ECMA-262 6th Edition. This implementation scrupulously follows the ES standard's pseudo-code line-by-line, thus ensuring that our interpreter is correct with respect to the standard and allowing us to regard the interpreter as the standard itself in the context of static analysis. To this end, we also extend the ECMA-SL execution engine with new programming constructs, including UTF-8 support. Furthermore, our reference implementation is thoroughly tested against Test262, the official ECMAScript test suite. Finally, in order to assist with the transition from the current HTML representation of the standard to ECMA-SL, we introduce `HTML2ECMA-SL`, a tool that aims to generate the ECMA-SL code for any given function described in the standard.

Keywords: ECMAScript, Specification Language, Reference Interpreters, Dynamic Languages, Test262, OCaml

1 Introduction

JavaScript (JS) is one of the most important programming languages today because it is the *de facto* option when it comes to dynamic front-end web development. Whilst there are viable alternatives (such as TypeScript, CoffeeScript, and so on), the fact is that these end up being transpiled into JavaScript. JavaScript is used by over 95% of websites[1], is the most active language on GitHub[2] and the second most active on StackOverflow[3]. Throughout the years, a rich ecosystem was built on top of JS, from transpilers to frameworks, such that, for a web developer, it is almost impossible to avoid working with JS in some way or another.

JavaScript, whose official name is now ECMAScript, is an ever-evolving programming language that keeps backward compatibility with its earlier versions. It is currently in its 12th edition. However, a large portion of JavaScript's current features were introduced with the release of ECMAScript 6 (ECMAScript 2015), which has added new syntax for writing more complex applications, among many other features that would define the next era of JavaScript.

1.1 ECMA-262's Complexity

ECMA-262 [8] is the specification of the ECMAScript (ES) programming language, standardized by Ecma International. Since 2015, the specification underwent yearly updates, with the largest updates being introduced in ES6 and ES8.

Given the current size and complexity of ECMA-262, introducing new features to the language is a complex and error-prone process. For instance, it must be guaranteed that any change does not break the behaviour of previous features and that it is compatible with the internal invariants maintained by the semantics of the language. For this reason, the ECMAScript committee created Test262 [11], the official ECMAScript Conformance Test Suite, which is known to have significant coverage issues. Even with extensive testing, one might fail to cover an edge case of the language. In contrast to testing, formal methods offer strong correctness guarantees. But, in order to be able to apply formal methods to the ECMA-262 standard, one must first have a formal model of it.

1.2 ECMA-SL Project

The ECMAScript Specification Language (ECMA-SL) is a dedicated intermediate language (IL) for ES analysis and specification that aims to mitigate the problems caused by

[1] Usage statistics of JavaScript as client-side programming language on websites, 31 October 2021, W3Techs.com - https://w3techs.com/technologies/details/cp-javascript

[2] Github most active programming languages based on pull requests, 31 October 2021 - https://madnight.github.io/githut

[3] Stack Overflow Trends over time based on use of their tags, 31 October 2021 - https://insights.stackoverflow.com/trends

the complexity of ECMA-262. One of the goals of the ECMA-SL project is to have a reference interpreter for ES that is tightly linked to the text of ECMA-262, allowing us to regard the interpreter as ECMA-262 itself in the context of formal analysis. We can also generate a textual version of ECMA-262 from the ECMA-SL code, making ECMA-SL a viable alternative for specifying the ES language. As we can test the ECMA-SL interpreter against Test262, we can: make sure that any changes are backward compatible with previous versions; apply test generation techniques [16] to automatically obtain tests for newly introduced features; and measure the coverage of Test262. The ECMA-SL project is currently in the development stage and ships a fully functional reference interpreter for the ECMA-262 Edition 5.1 [10], henceforth referred to as `ECMARef5`, in part contributed to by this work.

1.3 Problem Statement

In this work, we extend `ECMARef5` in order to include the ES5 Array object, the ES5 RegExp object, and the ES5 JSON object, which were yet to be implemented with regards to the pseudo-code of the standard. We also implement a few methods of the ES5 String object, that depend on the ES5 RegExp object and Unicode, and start implementing the ECMA-SL reference interpreter for the ECMA-262 6th Edition, henceforth referred to as `ECMARef6`, by implementing the ES6 Promise object.

The ECMA-SL execution engine is implemented in the OCaml [1] programming language and includes a parser based on *Menhir* [4]. In order to implement the aforementioned built-in objects, we extend the ECMA-SL execution engine with new programming constructs and UTF-8 support. As part of this work, we also create a tool for converting the HTML representation of the ES6 standard directly to ECMA-SL code, which we call `HTML2ECMA-SL`. So far we can only guarantee full support of the ES6 Promise object for `HTML2ECMA-SL`. `HTML2ECMA-SL` has a number of benefits:

1. *It guarantees consistency*: The ECMA-SL instructions generated by this tool are kept consistent across functions. Sometimes, there can be multiple ways to implement a given pseudo-code instruction, and, by adhering to one single way, the inverse process of converting the ECMA-SL code to a textual representation of the standard is facilitated. Moreover, different programmers may have different styles of coding (e.g. placement of braces, spaces around operators, function names, etc.). `HTML2ECMA-SL` helps with style standardization.
2. *It avoids errors*: Manual tasks are known for being error-prone. It is easy for the programmer to make mistakes which can take a long time to debug and detect.
3. *It speeds up the implementation*: Whilst there is an initial overload on implementing this tool alongside the ECMA-SL reference implementation, as more statements and expressions are supported by the tool the easier and quicker it becomes to implement more functions from the ECMA-262's pseudo-code along the line.

We thoroughly evaluate the three main modules of this work: The extensions to the ECMA-SL execution engine, the built-in objects added to the ECMA-SL reference interpreter, and the `HTML2ECMA-SL` conversion tool. The string operators added to the ECMA-SL execution engine concerning UTF-8 were partially tested using the *OUnit* [6] testing framework and all the extensions to the ECMA-SL execution engine were tested together with the ECMA-SL reference interpreter against Test262. Out of a total of 3,440 applicable tests from Test262, we pass 99.8%, thus guaranteeing that our reference implementation is correct with respect to the ECMA-262 standard. `HTML2ECMA-SL` is evaluated using the Jest [12] testing framework, by comparing each generated ECMA-SL function's code to the code that we deem to be correct. `HTML2ECMA-SL` passes 51 out of 51 unit tests and has a test coverage of 97%.

2 ES Standard

Edition 5.1 [10] of ECMA-262 defines the ECMAScript programming language for version 5.1, hereafter referred to as ES5, and the 6th edition [8] defines version 6 of the language, hereafter referred to as ES6. ECMA-262 defines the types, values, objects, properties, functions, program syntax and semantics that should exist in an ES language implementation. Note that ECMA-262 allows an implementation of the language to provide additional types, values, objects, properties, and functions [4].

ECMAScript defines a collection of built-in objects whose specification comprises almost half of the 6th edition [8] of the ECMA-262 standard (253 out of 545 pages). Built-in objects provide the essential functionality of the language. ES6 is mostly compatible with the previous versions, with a few exceptions [5], so it contains the same built-in objects that are present in ES5. However, it introduces new built-in objects, of which the Promise object is a part of. Of the several built-in objects present in the ECMA-262 standard, this work focuses on the ES5 Array object, part of the ES5 String object, the ES5 RegExp object, the ES5 JSON object, and the ES6 Promise object.

3 Related Work

The literature includes a great many number of works on different types of program analysis techniques for JavaScript, including: type systems [17, 44], points-to analyses [31], control-flow analyses [37], abstract interpretation [20, 22], information-flow analyses [21, 26, 30], and program logics [24, 25, 27]. Here we focus our account of the related work on projects that try to formalise the semantics of JS,

[4] Standard ECMA-262's 6th Edition, Chapter 2, Paragraph 4.
[5] Standard ECMA-262's 6th Edition, Annex D and Annex E.

including reference implementations of some of its built-in objects.

The particularity of our ES5/6 reference interpreter, when compared to previous projects, is that it is designed to be identical to the ECMA-262 specification. In contrast, existing reference interpreters/formalisations differ substantially from the text of the ECMA-262 standard, resulting in two important drawbacks: (1) How can we know that we implemented the intended behaviour as specified in the ECMA-262 standard? Trust in reference implementations is only obtained through testing against Test262, which is known to have coverage issues. (2) How to guarantee that the reference implementation is accessible to a wide audience comprising developers with very different programming backgrounds? Most existing reference implementations were developed in highly technical/mathematical formalisms (such as Coq [14] and K [2]), which are generally out of the reach of non-academic programmers.

The first formal operational semantics of JavaScript. Maffeis et al. design the first operational semantics for the ECMAScript language version 3 (ES3) [34]. This semantics was the first ES semantics that follows the corresponding standard faithfully. The authors used this semantics to reason about various security properties of web applications and mashups [35, 36].

Lambda calculi for reasoning about JavaScript code: λJS and S5. Guha et al. [29] define λJS a core lambda calculus that captures the most fundamental features of ES3. The authors also implement an interpreter for λJS in Racket [13] together with a compiler from ES3 to λJS. This project also comes with a simple type system for checking a security property of λJS expressions.

Later, Politz et al. [40] extend λJS from ES3 to ES5 including a formal semantics of ES property descriptors and a complete treatment of the `eval` statement.

Mechanised semantics of JavaScript: JSCert and KJS. Bodin et al. [18] implement JSCert, a formalisation of the semantics of the ES5 standard written in the interactive proof assistant Coq [14]. Besides an operational semantics, the authors also implement a reference interpreter called JSRef, which they prove correct with respect to the defined operational semantics. Using the Coq-to-OCaml extraction mechanism [1], the authors are able to obtain an OCaml version of the JSRef interpreter, which they use to test JSRef against Test262 [11].

One year later, Gardner et al.[28] extend the JSRef interpreter with support for ES5 Arrays by linking it to the Google's V8 [3] Array built-in object implementation. The authors additionally provide a detailed account of the testing infrastructure used to evaluate the JSCert project, including a detailed breakdown of all the tests passing and failing.

Park et al. present KJS [38], a formal semantics of ES5 written in the K framework [2], a state-of-the-art term-rewriting system with support for several types of program analyses. KJS was tested against Test262 passing 2,782 core language tests. KJS can be executed symbolically and, in fact, it was used to symbolically run small JS programs.

JSExplain. Charguéraud et al. present JSExplain [19], a reference interpreter for the ES5 language that allows programmers to code-step not only their JS code but also the pseudo-code of the standard. To this end, JSExplain produces inspectable execution traces, which allow the programmer to code-step not only his ES5 programs but also the ES5 interpreter itself as it executes the input ES5 programs.

Operational semantics / modules of ES6 Promises. Madsen et al. [33] design λP, a λ-calculus that captures the fundamental behaviour of ES6 Promises. More concretely, λP is an extension of λJS [29] with dedicated constructs for the creation and manipulation of Promises. The authors further introduce the concept of Promise graph and use it to reason about common bug patterns involving Promises.

Later, Alimadadi et al. [15] develop PromiseKeeper, a runtime debugging tool built on top of Jalangi [42] for identifying and explaining Promise-related bugs.

Finally, Sampaio et al. [41] develop JaVerT.Click, an extension of JaVerT with support for event-based programming. More concretely, JaVerT.Click includes JavaScript reference implementations of: the JS Promise built-in object, the DOM Core Level 1 API [45], and the DOM UI Events API [46]. The authors use JaVerT.Click to symbolically test two real world event driven libraries: cash [47] and p-map [43].

JISET. Very recently, the authors of [39] introduce JISET, an instruction set specifically designed to be a compilation target for ES code. They further develop an extraction mechanism that semi-automatically creates an ES to JISET compiler from the text of the ECMA-262 standard. This extraction mechanism generates not only compilation rules for all the constructs of the language, but also an ES parser. The JISET project targets the 10th edition of the ECMA-262 standard. The project comes with an execution engine for JISET which the authors use to test JISET against Test262, passing 18,064 out of 35,990 available tests.

Contrasting with ECMA-SL. Although there have been multiple implementations of the ECMA-262 standard, none of them feature a syntax that closely resembles the pseudo-code of the standard. Also, none of the reference implementations have implemented the ES JSON and RegExp built-in objects, as well as the ES String methods that depend on regular expressions. Additionally, most reference implementations ignore character encoding, unlike our reference implementation. Consequentially, `ECMARef5` is the most complete academic reference implementation of ES5 to date, passing 12,026 tests out of 12,074 filtered tests [32], whilst

JSCert [18] passes 1,796 tests, KJS [38] passes 2,782 tests, JS-Explain [19] passes 5,000 tests, S5 [40] passes 8,157 tests, and JS-2-JSIL [24] passes 8,797 tests. Even though JISET passes 18,064 tests, it only accounts for 62% of its available pool of tests (28,952), which is considerably larger than the pool of tests of `ECMARef5`, due to JISET targeting the 10th edition of the standard. Therefore, the existing reference interpreters prove themselves unsuitable for promoting an executable ES specification, which is the goal of ECMA-SL.

4 Extending ECMA-SL

The ECMAScript Specification Language (ECMA-SL) is a simple imperative language with top-level functions and extensible objects. It was created with the purpose of serving as a dedicated intermediate language (IL) for ES analysis and specification. For this reason, ECMA-SL contains all the control-flow constructs used by the pseudo-code of ECMA-262; these include, for instance, a `return` statement, a `do-while` loop, a `while` loop, and an `if` statement. As in ECMAScript, we can dynamically add / remove properties from objects. Finally, the primitive data types of ECMA-SL mostly coincide with those of ES, with the most notable difference being that ECMA-SL also includes the integer type.

ECMA-SL Project Architecture. The ECMA-SL project contains four main components that together make up the ECMA-SL reference interpreter.

- `JS2ECMA-SL` - a tool written in *Node.js* that parses a given JavaScript program, using the *Esprima* parser [5], and then creates an ECMA-SL function, called `build-AST`, that builds the abstract syntax tree (AST) of the given program in memory, returning the ECMA-SL object corresponding to the root of the AST.
- `ECMARef5` - the ES5 interpreter written in ECMA-SL, which also contains the implementation of the ES5 built-in objects.
- `ECMARef6` - the ES6 interpreter written in ECMA-SL, which also contains the implementation of the ES6 built-in objects.
- ECMA-SL Execution Engine - the interpreter of ECMA-SL, written in the OCaml [1] programming language, which receives and executes an ECMA-SL program. The output of the program and execution trace are printed to the console; furthermore, the final heap resulting from the program's execution can also be serialised to a file.

String encoding in OCaml. According to Chapter 6 of the ES5 standard [10], the source text of an ES program is to be interpreted using the UTF-16 encoding. If the source text happens to be saved in a different encoding, then it must first be converted to the UTF-16 encoding. This is due to the fact that string literals in ES are to be encoded in UTF-16. However, this detail had been overlooked in the beginning of the ECMA-SL project's development. In addition, the *ocamllex* [6] lexical analyzer, which had been chosen for the lexical analysis of ECMA-SL source code, has no support for Unicode.

Tackling the problem using UTF-8. In an attempt to address the problem of Unicode escape sequences occurring in string literals of some of the tests of Test262 [11], our first approach was to replace the existing string-related operators in ECMA-SL, which were unaware of the string encoding (s_len, s_nth, s_subst, from_char_code, and to_char_code), with operators that assume the encoding to be UTF-8.

Besides implementing new operators, we have also added support for Unicode escape sequences in ECMA-SL, which are converted to UTF-8 encoded code points during lexical analysis. An observation that we have made regarding Unicode escape sequences, is that until ES6, a single Unicode escape sequence could only represent one code point in the Basic Multilingual Plane (BMP), more specifically, a single Unicode escape sequence would require exactly four hexadecimal digits (e.g. `\uhhhh`). This means that code points outside the BMP would have to be represented with surrogate pairs. From ES6 onwards, Unicode escape sequences support a new syntax that allows a variable number of hexadecimal digits, by enclosing them in curly braces (e.g. `\u{10FFFF}`, which is equivalent to `\uDBFF\uDFFF` in UTF-16).

In order to demonstrate how we deal with UTF-8 encoded code points, Listing 1 presents the source code of our OCaml function that receives a string and returns the number of UTF-8 encoded code points that it contains.

```
1  let s_len_u = fun (s : string) : int ->
2    let rec loop s cur_i_u cur_i =
3      if cur_i >= (String.length s) then (cur_i_u) else
4      let c = Char.code (s.[cur_i]) in
5        if (c <= 0x7f) then
6          loop s (cur_i_u + 1) (cur_i + 1)
7        else if c <= 0xdf then
8          loop s (cur_i_u + 1) (cur_i + 2)
9        else if c <= 0xef then
10         loop s (cur_i_u + 1) (cur_i + 3)
11       else
12         loop s (cur_i_u + 1) (cur_i + 4)
13   in loop s 0 0
```

Listing 1. Implementation of the ECMA-SL `s_len_u` operator in OCaml

In line 1 of Listing 1 we declare our function, `s_len_u`, which calls the recursive function `loop` in line 13. The recursive function `loop` (line 2) receives the current index of the UTF-8 encoded code point, `cur_i_u`, and the current index of the byte, `cur_i`, that we visit on each iteration. In UTF-8, the number of leading 1's of a leading byte tells us how many bytes are used to encode a code point. With this in mind, we increase `cur_i_u` and `cur_i` accordingly. For instance, if the current byte has a value greater than $7F_{16}$ (01111111_2)

[6] Lexer and parser generators (ocamllex, ocamlyacc), 31 October 2021 - https://ocaml.org/manual/lexyacc.html

but less than or equal to DF_{16} (11011111_2), then our current byte, which is always the first of a UTF-8 encoded code point, must have two leading 1's, and thus we increase `cur_i` by two in order to get to the first byte of the next UTF-8 encoded code unit. Once there are no more bytes to visit in the string, `cur_i_u`, which now represents the number of UTF-8 encoded code points in the string, is returned. Notice that we do not check for invalid UTF-8 representations, because *Node.js*, which is used by `JS2ECMA-SL`, ensures that the ECMA-SL source text has a valid UTF-8 representation.

We have also implemented operator `utf8_decode`, which receives a Unicode escape sequence and returns the corresponding UTF-8 encoded code point, and operator `hex_decode`, which receives an hexadecimal escape sequence (e.g. `\xhh`) and returns the corresponding UTF-8 encoded code point.

UTF-8 is a temporary solution. The operators that we implemented deal with UTF-8 encoded code points. However, operations on ES strings should deal with 16-bit code units instead, which is not possible with the current UTF-8 encoded strings, due to the incompatibility between UTF-8 and UTF-16. For instance, the length of a UTF-16 encoded string should equal the number of 16-bit code units that it contains, which may or may not equal the number of UTF-8 encoded code points in a UTF-8 encoded string. For code points in the BMP, a 16-bit code unit of a UTF-16 encoded string is equal to the code point, which means that for UTF-8 encoded strings containing only code points in the BMP, the number of UTF-8 encoded code points will be the same as the number of 16-bit code units if the string were to be encoded in UTF-16. But for code points in other planes, the operators that we implemented will not produce the desired results. Because all of the tests of Test262 [11] that target ES5 do not contain Unicode escape sequences outside the BMP, with the exception of one test for the JSON's `stringify` method, this solution works as a temporary work-around.

Proposed solution. Due to time constraints, we were unable to implement an ideal solution in time for the Unicode problem. We considered making ECMA-SL's string-related operators aware of code points that would require two 16-bit code units in a UTF-16 encoding, but the fact is that iterating over code points is already detrimental to performance on its own. In ES and in most programming languages, string-related operations are only aware of code units, not code points. Thus, the length of a string is to be counted in code units and the *nth* "character" of a string is to be the *nth* code unit. Furthermore, the ECMA-SL source code of the `buildAST` function, that is constructed by `JS2ECMA-SL`, is encoded in UTF-8 in order to be compatible with *ocamllex*, and because `JS2ECMA-SL` is written in *Node.js*, isolated surrogate values are replaced with the $FFFD_{16}$ code point when written to a file, which is the replacement character recommended by the standard for invalid UTF representations [7].

In order to solve the Unicode problem, we propose the replacement of *ocamllex* with *sedlex* [8], a lexer generator for OCaml [1], similar to *ocamllex*, that features support for UTF-16 inputs.

4.1 Other Extensions

Besides the string-related operators, we have also added other operators to ECMA-SL that were deemed necessary during the implementation of the ES5 built-in objects covered in this work: `int_to_string`, `int_of_string`, `floor`, `l_prepend`, `l_reverse`, `l_remove_last`, `octal_to_decimal`, `int_to_four_hex`, `parse_number`, `parse_string`.

5 Reference Implementation

We have implemented the following ES built-in objects: ES5 Array, ES5 RegExp, ES5 String (partially), ES5 JSON, and ES6 Promise. Here we elaborate on their internal representation and some of the challenges that we faced.

5.1 Internal representation

Figure 1 represents the ES5 Array object's graph implementation in ECMA-SL. Because ECMA-SL objects do not make a distinction between internal and named properties, the latter are stored in a property named `JSProperties`, while the former are stored as regular properties. This approach avoids name collisions between these two types of properties.

ES5 Array. Internal methods of an ES built-in object are stored as properties of its corresponding ECMA-SL object, with the key being the method's name, as described in ECMA-262, and the value being the corresponding ECMA-SL function. Naturally, should there be variations of an internal method, the name given to its corresponding function can differ, as is the case of the `DefineOwnProperty` internal method of the ES Array object, whose corresponding ECMA-SL function we have decided to call `DefineOwnPropertyArray`. Built-in methods of an ES built-in object are treated differently. First, an ES Function object is created [9] that stores, among other data, the name of the ECMA-SL function for the corresponding built-in method. Then, a data property descriptor is created to store this Function object. Finally, the data property descriptor is assigned to a named property of the built-in object, whose key is, as expected, the name of the built-in method.

[7] Are there any byte sequences that are not generated by a UTF? How should I interpret them?, 31 October 2021 - https://www.unicode.org/faq/utf_bom.html#gen8

[8] Sedlex, an OCaml lexer generator for Unicode, 31 October 2021 - https://github.com/ocaml-community/sedlex

[9] Creating Function Objects in ES5: http://es5.github.io/#x13.2

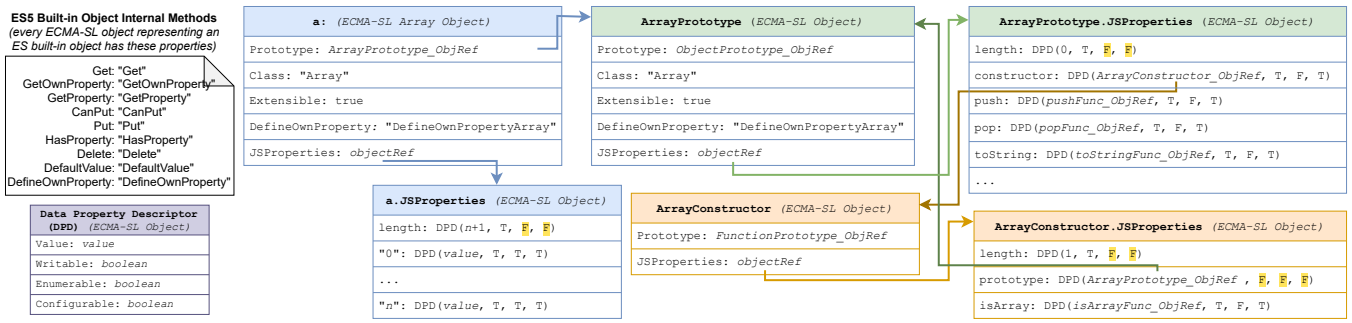

Figure 1. ES5 Array Object Graph in ECMA-SL.

ES5 String. For the String object, we actually store each character of the string in a named property of the object, despite the fact that the `[[GetOwnProperty]]` internal method creates and returns a new data property descriptor for a given index. We take this approach because the auxiliary functions in ECMA-SL that had been created to get the named properties of an object will look for them in the object's `JSProperties` property. Therefore, statements such as the *for-in* loop, that iterate over the enumerable properties of an object, can only be aware of them if they exist in the `JSProperties` property of the object being iterated over.

ES5 RegExp. In ECMA-SL, ES5 RegExp instances are represented as ECMA-SL objects. The methods shared by all RegExp instances are stored in the ECMA-SL object corresponding to the `RegExp.[[Prototype]]` object. The named properties of a RegExp instance, `re`, are stored in the object `re.JSProperties`. Particularly, RegExp instances store their corresponding matchers in the internal property `[[Match]]`. In ECMA-SL, matchers are modelled as lambda functions that recognize expansions of the corresponding regular expression. Our implementation includes a regular expression interpreter that, given the AST of a regular expression, generates the lambda function corresponding to its matcher. The implemented interpreter follows the algorithms proposed by Fragoso Santos et al. in [23]. An account of these algorithms is out of the scope of this thesis. The AST of the regular expression to be interpreted is computed by `JS2ECMA-SL` using the `regexp-tree` [10] regular expression parser.

ES5 JSON. Our ECMA-SL implementation of the JSON object graph coincides almost exactly with the ES representation, except that the named properties of the JSON object are stored in a separate object whose location is stored in its property `JSProperties`. Hence, the methods exposed by the JSON object, `stringify` and `parse`, are stored as properties of the object `JSON.JSProperties`.

ES6 Promise. Our implementation of the ES6 Promise built-in object was made on top of `ECMARef5`, in parallel with the transitioning of `ECMARef5` to `ECMARef6`. For this reason, the ES5 built-in object internal methods were kept, along with the `[[Class]]` internal property. The Symbol keyed properties were not implemented.

[10] `regexp-tree`, 31 October 2021 - https://github.com/DmitrySoshnikov/regexp-tree

5.2 Line-by-line closeness

In order to demonstrate the line-by-line closeness of our reference implementation with respect to the text of ECMA-262, Listing 2 presents part of our ECMA-SL code for the `exec` method of the RegExp prototype object. Each instruction is accompanied by a comment with its corresponding pseudo-code instruction, as described in the ES5 standard. We can see that the ECMA-SL constructs are descriptive and very similar to the pseudo-code of the standard.

```
1  /* 9. Repeat, while matchSucceeded is false */
2  while (matchSucceeded = false) {
3    /* a. If i < 0 or i > length, then */
4    if ((i < 0) || (i > length)) {
5      /* i. Call the [[Put]] internal method of R with
6          arguments "lastIndex", 0, and true. */
7      {R.Put}(R, "lastIndex", 0., true);
8      /* ii. Return null. */
9      return 'null
10   };
11   /* b. Call the [[Match]] internal method of R with
12       arguments S and i. */
13   ret := {R.Match}(R, S, i);
14   /* c. If [[Match]] returned failure, then */
15   if (isFailure(ret)) {
16     /* i. Let i = i+1. */
17     i := i + 1
18   }
19   /* d. else */
20   else {
21     /* i. Let r be the State result of the call to [[
       Match]]. */
22     r := ret;
23     /* ii. Set matchSucceeded to true. */
24     matchSucceeded := true
25   }
26 };
```

Listing 2. A snippet of the ECMA-SL implementation of the RegExp prototype's `exec` method.

Exceptions. Some instructions of the pseudo-code leave the implementation algorithm to the discretion of the programmer. These instructions cannot therefore follow a line-by-line closeness approach. For the `JSON.parse` method, the standard simply says that the provided JSON text is to be parsed according to a given set of grammar rules (Section 15.12.2 of the ECMA-262 Edition 5.1 [10], pseudo-code line 2). Hence, we were free to implement this function as we deemed appropriate. We considered two main options: implement the function in OCaml and create an ECMA-SL operator exposing the OCaml function to the ECMA-SL code, or implement the function directly in ECMA-SL. We chose to go for the second option as it is much easier to manipulate and interact with ECMA-SL objects in ECMA-SL code than at the OCaml level. In order to parse the JSON text, we made use of several auxiliary functions. Importantly, pseudo-code instructions that leave the implementation to the discretion of the programmer raise a problem when it comes to the generation of the English HTML description of ECMA-262, as we cannot generate the original text from it. In order to solve this problem, ECMA-SL uses annotations. However, due to time constraints, we have left this topic outside the scope of this thesis.

Automatic generation of ECMA-SL code. The implementation of the ES6 Promise built-in object, unlike the implementation of the ES5 built-in objects presented heretofore, was generated entirely by the `HTML2ECMA-SL` tool, which will be introduced in Section 6, with the exception of the ECMA-SL functions responsible for creating the Promise object's graph and other auxiliary functions. In order to implement the ES6 Promise built-in object, we also had to make a temporary implementation of Jobs and Job Queues, specified in Section 8.4 of the ECMA-262 6th Edition [8], so that the Promise object could be tested against Test262 [11]. Other ES6 internal functions that are used by the Promise object, such as operations on iterator objects, were implemented by other contributors to the ECMA-SL project.

5.3 Implementation-defined behaviour

Some methods, or part of their behaviour, can be implementation-defined. Implementation-defined behaviour is outside the scope of the goals of ECMA-SL, thus, we approach these cases in a minimalistic way.

ES5 Array. For instance, the `toLocaleString` method of the Array prototype does not specify which list-separator String to use for separating the array elements. We use the comma character as list-separator for simplicity. The `sort` method of the Array prototype does not specify if elements that compare equal should keep their order, nor the sorting algorithm to use. We chose to use Quicksort for its simplicity, which does not keep equal elements in their original order.

ES5 String. The String built-in object has several functions whose summary, or pseudo-code, does not rigorously specify how to implement. Such is the case for the following casing operations: `toLowerCase`, `toLocaleLowerCase`, `toUpperCase`, and `toLocaleUpperCase`. For these methods, the specification tells us to use the case mappings in the Unicode character database [7], that is, the `UnicodeData.txt` and `SpecialCasings.txt` files included in this database. However, this mapping implementation will produce several lines of code due to the conditional mappings that can be found in the `SpecialCasings.txt` file. As an example, when converting the upper-case Σ character ($03A3_{16}$) to lower-case, it will generally be mapped to σ ($03C3_{16}$), unless the character appears at the end of a string, in which case it will be mapped to ς ($03C2_{16}$). This was the only conditional casing that was implemented in this work, required for passing all tests of Test262 [11] that cover the ES5 standard [10].

An observation that we have made while implementing the aforementioned casing methods, is that there is an additional mapping possible, besides the lower-case and upper-case mappings, and that is the title-case mapping. For instance, the lower-case ß character ($00DF_{16}$) is mapped to the upper-case `SS` ($0053_{16}0053_{16}$) and to the title-case `Ss` ($0053_{16}0073_{16}$). This leads us to suggest the addition of a `toTitleCase` method to the String prototype object.

6 HTML2ECMA-SL

During the implementation of our first ES built-in object, the ES5 Array, we came to the realization that this task was considerably repetitive. For each method, we akways copy its pseudo-code to our ECMA-SL source file, with each instruction in its own comment, and implement the instructions below their respective comments. This process can become quite dull and error-prone, which led us to develop `HTML2ECMA-SL` for the purpose of automating it. `HTML2ECMA-SL` searches for a function in the HTML version of the ECMA-262 6th Edition [9] and parses the HTML description of the function's pseudo-code into functional ECMA-SL code, by taking advantage of regular expressions.

`HTML2ECMA-SL` allows us to output the generated ECMA-SL code of a function to: the console, with or without syntax highlighting, which is useful for debugging and copy/pasting; an ECMA-SL source file, which will become useful when `HTML2ECMA-SL` is capable of generating a whole section of the standard; and an HTML file with syntax highlighting, which may be useful for embedding the generated ECMA-SL code in a web application or PDF document. On top of this, `HTML2ECMA-SL` also allows us to specify the number of spaces or tabs to use for the indentation, the maximum line width of the comments, and whether to generate the comments or not.

We have fully implemented the automatic generation of ECMA-SL code for the ES6 Promise and ES6 Proxy built-in

objects. Because many pseudo-code instructions have recurring patterns, many other methods are already partially or fully implemented, including those belonging to newer versions of ECMA-262, although these require a few modifications to our existing regular expressions. This means that, even if we cannot generate ECMA-SL code for the entirety of ECMA-262, we can already automate many of the repetitive tasks, namely, the generation of comments and the generation of ECMA-SL instructions whose patterns have already been implemented in `HTML2ECMA-SL`. At this stage, six other contributors to the ECMA-SL project are already enjoying the benefits of this tool.

Project structure. We decided to develop `HTML2ECMA-SL` in TypeScript (TS), a superset of JS that adds a type system to the language. For the runtime we use Node.js [11]. Because the focus of ECMA-SL is on ES, this can help us further understand modern ES from a user's perspective. The most relevant source code of `HTML2ECMA-SL` is divided into the following ES modules [12]:

- `main.ts` - This is the entry point of the application, responsible for: parsing the given command line arguments, using the Yargs [13] Node.js library for this purpose, which allows us to specify various options on how to output the generated ECMA-SL code; fetching the HTML version of the ECMA-262 6th Edition [9] and extracting the HTML portion of a given function's specification from it, using the `ecma262.ts` module; and parsing the HTML of the function's specification and generating the corresponding ECMA-SL code, using the `parser.ts` module.
- `ecma262.ts` - Responsible for: fetching the HTML of the ECMA-262 6th Edition [9] from its URL, caching it to a local file for future uses; fetching the HTML portion of a function's specification from the HTML of ECMA-262; and verifying if a given function is a constructor function by searching for its name in Section 18.3 of the ECMA-262 6th Edition.
- `parser.ts` - Responsible for parsing the HTML portion of a function's specification and producing the corresponding ECMA-SL source code, taking into account the indentation, line-width, and comment generation preferences.
- `syntax_highlighting.ts` - Adds syntax highlighting to the ECMA-SL source code by using the Prism [14] syntax highlighting library, which generates an HTML output that can then be converted to Linux console or Windows Terminal [15] syntax-highlighted text by using the Chalk [16] Node.js library.

[11] Node.js, 31 October 2021 - https://nodejs.org/en/
[12] ES modules were introduced in the ECMA-262 6th Edition [9] and allow us to split our code in multiple files.
[13] Yargs, 31 October 2021 - https://www.npmjs.com/package/yargs
[14] Prism, 31 October 2021 - https://www.npmjs.com/package/prismjs
[15] Windows Terminal, 31 October 2021 - https://github.com/microsoft/terminal
[16] Chalk, 31 October 2021 - https://www.npmjs.com/package/chalk

Aside from the aforementioned modules, we also have a test directory containing TS source files for unit tests, in which we use the Jest [12] testing framework. In these tests, we verify if the text of a generated ECMA-SL function, whose instructions we have implemented, is as expected. Currently, `HTML2ECMA-SL` consists of 1,472 LOC for the source code and 1,188 LOC for the unit tests.

Parsing instructions. The structure of the HTML `SECTION` tag of a function's specification is divided in three parts: the header, which contains the function's section, name, parameters, and summary; the pseudo-code, which is identified by an `OL` tag with class `"proc"`; and the footer, which contains notes and/or miscellaneous information about the function. Some function (or macro) specifications may have more than one main block of pseudo-code, as is the case of the `IfAbruptRejectPromise` macro (Section 25.4.1.1.1 of ES6).

In order to extract the necessary information from the HTML, we make extensive usage of the RegExp object. This is possible because the patterns of function specifications are generally the same, as well as the patterns of recurring pseudo-code instructions. Listing 3 demonstrates the regular expression pattern that we use for extracting the information corresponding to a function's signature. We obtain the function's section identifier with the first capturing group, highlighted in red, the function's name with the second capturing group, highlighted in green, and the function's parameters with the third capturing group, highlighted in blue. The third group requires further text processing in order to extract the individual parameters from the captured string. From the second capturing group, we can not only extract the function's name, but we can also infer if the function is internal or built-in, by verifying if the name contains the dot character, which is used to access properties of an object.

```
1 const re = /<h1><span[^>]*><a[^>]*>([^<]*)<\/a><\/span>\s
    *([^(]+) (\([^)]*\))?\s*<\/h1>/;
```

Listing 3. RegExp for extracting a function's signature from the HTML.

For parsing the pseudo-code instructions, we parse each HTML `LI` tag, that is, each instruction, individually. Each instruction is matched against a list of possible regular expression patterns until a match is found. Because instructions may contain several expressions, this process can repeat for some of the capturing groups. As we have not written regular expressions to cover all the possible patterns of pseudo-code instructions in the standard, it is often the case that `HTML2ECMA-SL` is not able to create an ECMA-SL statement corresponding to the instruction to be parsed. In such cases, the tool simply outputs `/* TODO: Instruction not yet implemented. */` and continues with the generation process.

7 Evaluation

Here we present the evaluation results for the main topics of this work: the implementation of the ES5 Array, part of the ES5 String, ES5 RegExp, and ES5 JSON built-in objects in `ECMARef5`; the implementation of the ES6 Promise built-in object in `ECMARef6`; and `HTML2ECMA-SL`.

7.1 Reference Implementations

At this stage, our evaluation is primarily done with Test262 [11], the official ECMAScript Conformance Test Suite. Despite its known coverage issues, Test262 is the most comprehensive ES test suite to date. At a later stage in the ECMA-SL project, it will be possible to measure the coverage of Test262, with respect to the code of the `ECMARef5` interpreter, in order to know which features have not yet been tested.

Test filtering. The evolution of Test262 is consistent with that of ECMA-262. Thus, Test262 contains tests that target the latest edition of the standard—currently ES12—which our ES5 and ES6 reference implementations cannot handle. Furthermore, as of October 2021, the number of tests from Test262 covering ES built-in objects is over 19k. Fortunately, many of the Test262 test files targeting ES5 contain the "es5id" key in the meta-data of the test, known as the frontmatter [17]. This simplifies some of the filtering process required in our project.

Unfortunately, both the "es5id" and "es6id" keys have been deprecated [18]. New tests contain a "esid" key, without the version number, whose value is the hash ID of the HTML anchor of a section of the latest ECMA-262's HTML version. For this reason, many tests with the "esid" key may be supported by ES5 and/or ES6. We filter these tests manually as we investigate the causes for the failing tests during development. In some cases, there are small incompatibilities between different versions of ES, such as the "length" property of Function objects being configurable from ES6 onwards. For these cases, we have chosen to adapt the behaviour of our reference interpreter to the latest version of ECMA-262.

Testing pipeline. Test262 test files make use of auxiliary functions for run-time assertions. The set of auxiliary functions used in test files comes from a collection of helper files called the *harness* of Test262. In order to run a test file with the ECMA-SL project we must first include the harness, which we do by concatenating it with the test file prior to running it.

In order to automate the evaluation of our reference implementation we make use of a shell script that automates the testing process. This process works as follows: (1) we check if the frontmatter of the test includes the boolean value `onlyStrict` in the list of boolean values referenced by the key `flags`, in which case we prepend the directive `"use strict"` directive to the code of the test; (2) we prepend the code of the Test262 harness to the code of the test; (3) we compile the test to ECMA-SL and execute it using the ECMA-SL engine; (4) we analyse the result of executing the test. Negative tests are expected to throw an exception of a certain type, while positive tests are expected to execute normally.

After executing the pipeline for a given list of tests, we are given the information of which tests have passed and which tests have failed, as well as the total passing and total failing tests.

Test results. The test results of our reference implementation for the `ECMARef5` interpreter are presented in Table 1. In total, our ES5 reference implementation passes 3,433 out of the 3,440 filtered tests, with 7 tests failing. Many dependencies of the ES6 Promise built-in object, such as syntactic elements and built-in functions, were not implemented before the conclusion of this work. Thus, only 171 of a total of 613 filtered tests for the ES6 Promise object are currently passing.

With respect to the failing tests, we currently have three tests failing for the methods that we implemented for the ES5 String object, which are caused by a faulty conversion of large numbers to string values in ECMA-SL. This is because the current implementation of the `ToString` method applied to the Number type, as specified in Section 9.8.1 of the ECMA-262 Edition 5.1 [10], is incomplete in `ECMARef5`. The test that currently fails for the JSON object has to do with the attempt of stringifying Unicode surrogate values. A Unicode surrogate value is currently replaced with the $FFFD_{16}$ code point by `JS2ECMA-SL`, as long as we are using UTF-8, which causes the stringify operation to generate a wrong result. Finally, for the RegExp object, we have two tests failing due to backreferences appearing before their capture groups being treated as decimal escapes by the `regexp-tree` library [19], which we use for parsing the string representation of an ES regular expression pattern to its corresponding AST. The remaining test that is currently failing for the RegExp object is due to an implementation error in the handling of backslash characters in the string representation of RegExp patterns.

7.2 HTML2ECMASL

The evaluation of `HTML2ECMA-SL` is a challenging task, since this application is not expected to produce fully functional ECMA-SL code right out-of-the-box, considering that the

[17] Test262 frontmatter, 31 October 2021 - https://github.com/tc39/test262/blob/main/CONTRIBUTING.md#frontmatter

[18] Test262 Technical Rationale Report, 31 October 2021 - https://github.com/tc39/test262/wiki/Test262-Technical-Rationale-Report,-October-2017#specification-reference-ids

[19] Backreferences appearing before their capture groups incorrectly treated as decimal escapes, 31 October 2021 - https://github.com/DmitrySoshnikov/regexp-tree/issues/69

Section	Description	#T	Passed	Failed
15.4	Array	117	117	0
15.4	Array prototype	2150	2150	0
15.5.4	String prototype	538	535	3
15.12	JSON	116	115	1
15.10	RegExp	109	109	0
15.10	RegExp grammar	292	289	3
15.10	RegExp prototype	118	118	0
Total		3,440	3,433	7

Table 1. Test262 test results for our ES5 reference implementation.

ECMA-SL project is still under development. It will often be necessary to review the generated code, write auxiliary functions, and, in some cases, extend ECMA-SL with new operators and syntax. Hence, we cannot simply evaluate HTML2ECMA-SL by having it generate several sections of the ECMA-262 standard and then check if the generated code passes the corresponding Test262 tests.

In order to evaluate HTML2ECMA-SL, we apply it to Sections 25 and 26 of the ECMA-262 6th Edition [9], and check whether the generated ECMA-SL code is syntactically valid or not. These sections correspond to the Promise and Proxy built-in objects, respectively. To automate the testing process, we make use of unit tests, resorting to the Jest [12] testing framework. Each unit test compares the generated ECMA-SL code for a given function described in the ECMA-262 standard with the code that we expect to be generated, which is susceptible to change over time as we adjust it to work with ECMARef6.

As previously mentioned, we could not have fully functional code for the ES6 Promise built-in object due to the lack of syntactic and built-in function elements in ECMARef6, which is currently under development. We also did not test the code generated for the ES6 Proxy built-in object in ECMARef6. Despite this, testing that the ECMA-SL code is generated as expected is important in order to ensure that future changes to HTML2ECMA-SL do not introduce silent bugs and corrupt the behaviour of the application. HTML2ECMA-SL currently passes 51 out of 51 unit tests, with a test coverage of 97%, measured by Codecov [20].

8 Conclusion

With the steady evolution of the ECMA-262 standard, it becomes increasingly difficult to maintain and extend this language specification. For this reason, we contribute to the ECMA-SL project by implementing the following built-in objects in its ES5 reference interpreter: Array, String (partial), RegExp, and JSON. This work paves the way for the implementation of these built-in objects in future ES reference interpreters, written in ECMA-SL, that target later versions of ECMA-262. Additionally, we further contribute to the ECMA-SL project by devising a tool, which we have called HTML2ECMA-SL, that considerably simplifies the implementation process by automatically generating ECMA-SL code from the pseudo-code of the ECMA-262 6th Edition [9]. We were able to fully generate the ECMA-SL code for the ES6 Promise and Proxy objects, which we used to initiate the development of an ES6 reference interpreter written in ECMA-SL.

The three main topics of this thesis were thoroughly evaluated. Our reference implementation of the ES5 Array, String, RegExp, and JSON built-in objects was tested against Test262 [11], passing 3,433 out of the 3,440 filtered tests. At this stage, we do not yet measure the coverage of Test262, nor apply formal methods to the language. Our reference implementation of the ES6 Promise built-in object was also tested against Test262, passing 171 out of 613 filtered tests. And, finally, HTML2ECMA-SL was tested using a custom made test suite consisting of 51 unit tests, each targeting a specific algorithm / function of the ECMA-262 standard. The proposed test suite has a 97% code coverage of HTML2ECMA-SL, giving us a strong guarantee of its correctness.

Our contribution to the ECMARef5 interpreter has a total of 3,184 LOC, our contribution to the ECMARef6 interpreter has a total of 542 LOC, and HTML2ECMA-SL has a total of 2,660 LOC. The ECMA-SL project, as well as HTML2ECMA-SL, will become open-source in the near future.

Future work. We categorize the future work in two types: immediate and long-term. Due to the time constraints of this project, we were unable to apply HTML2ECMA-SL to the entirety of the ECMA-262 6th Edition [9] and to generate a given built-in object's graph in ECMA-SL from the specification of the standard. Hence, our immediate future work would be to extend HTML2ECMA-SL in order to recognize more patterns of pseudo-code instructions occurring in the ES6 specification, as well as to generate a given built-in object's graph in ECMA-SL from its specification in the standard. Our implementation of the ES6 Promise object in the ECMARef6 interpreter is also failing a large percentage of tests, which we would seek to correct by extending the ECMA-SL language and the ECMARef6 interpreter as necessary. In the long-term, we would like to adapt the patterns of pseudo-code instructions that occur in the ECMA-262 6th Edition [9] to the later editions.

[20] Codecov is a dedicated code coverage solution, which we use to measure the coverage of our tests, 31 October 2021: https://about.codecov.io

References

[1] 1 [n.d.]. OCaml - General-purpose, multi-paradigm programming language. https://ocaml.org/. Accessed on 2021-10-31.

[2] 10 [n.d.]. K - Rewrite-based executable semantic framework. http://www.kframework.org/index.php/Main_Page. Accessed on 2021-10-31.

[3] 11 [n.d.]. V8 - Google's open source high-performance JavaScript and WebAssembly engine, written in C++. https://v8.dev/. Accessed on 2021-10-31.

[4] 12 [n.d.]. Menhir - LR(1) parser generator for the OCaml programming language. http://gallium.inria.fr/~fpottier/menhir/. Accessed on 2021-10-31.

[5] 13 [n.d.]. Esprima - ECMAScript parsing infrastructure for multipurpose analysis. https://esprima.org/. Accessed on 2021-10-31.

[6] 2 [n.d.]. OUnit - a unit test framework for OCaml. https://github.com/gildor478/ounit/. Accessed on 2021-10-31.

[7] 21 [n.d.]. The Unicode Character Database. https://www.unicode.org/ucd/. Accessed on 2021-10-31.

[8] 3 [n.d.]. ECMAScript® Language Specification, 6^{th} Edition / June 2015. https://www.ecma-international.org/ecma-262/6.0. Accessed on 2021-10-31.

[9] 4 [n.d.]. HTML rendering of ECMA-262 6^{th} Edition, The ECMAScript 2015 Language Specification. https://www.ecma-international.org/ecma-262/6.0/. Accessed on 2021-10-31.

[10] 5 [n.d.]. Annotated, hyperlinked, HTML version of Edition 5.1 of the ECMAScript Specification. https://es5.github.io/. Accessed on 2021-10-31.

[11] 6 [n.d.]. Test262 - Official ECMAScript Conformance Test Suite. https://github.com/tc39/test262/. Accessed on 2021-10-31.

[12] 7 [n.d.]. Jest - a JavaScript testing framework designed to ensure correctness of any JavaScript codebase. https://jestjs.io/. Accessed on 2021-10-31.

[13] 8 [n.d.]. Racket - General-purpose programming language. https://racket-lang.org/. Accessed on 2021-10-31.

[14] 9 [n.d.]. Coq - Interactive formal proof management system. https://coq.inria.fr/. Accessed on 2021-10-31.

[15] Saba Alimadadi, Magnus Madsen, Di Zhong, and Frank Tip. 2018. Finding broken promises in asynchronous JavaScript programs. *Proceedings of the ACM on Programming Languages* 2 (10 2018), 1–26. https://doi.org/10.1145/3276532

[16] Saswat Anand, E. Burke, T. Chen, John A. Clark, Myra B. Cohen, W. Grieskamp, M. J. Harrold, A. Bertolino, Juan Li, and H. Zhu. 2013. An Orchestrated Survey on Automated Software Test Case Generation I.

[17] Christopher Anderson, Sophia Drossopoulou, and Paola Giannini. 2005. Towards Type Inference for JavaScript. *Lecture Notes in Computer Science* 3586 (07 2005). https://doi.org/10.1007/11531142_19

[18] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages* 49 (01 2014), 87–100. https://doi.org/10.1145/2578855.2535876

[19] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. *WWW '18: Companion Proceedings of the The Web Conference 2018* (04 2018), 691–699. https://doi.org/10.1145/3184558.3185969

[20] Avik Chaudhuri. 2016. Flow: Abstract Interpretation of JavaScript for Type Checking and Beyond. (10 2016). https://doi.org/10.1145/2993600.2996280

[21] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. (10 2015). https://doi.org/10.1145/2810103.2813684

[22] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A parallel abstract interpreter for JavaScript. (02 2015), 34–45. https://doi.org/10.1109/CGO.2015.7054185

[23] Jose Fragoso Santos, Luís Almeida, and Rui Abreu. submitted. RexStepper: a Reference Debugger for JavaScript Regular Expressions. (submitted).

[24] Jose Fragoso Santos, Petar Maksimović, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. 2017. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM on Programming Languages* 2 (12 2017), 1–33. https://doi.org/10.1145/3158138

[25] Jose Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. 2019. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages* 3 (01 2019), 1–31. https://doi.org/10.1145/3290379

[26] Jose Fragoso Santos, Thomas P. Jensen, Tamara Rezk, and Alan Schmitt. 2016. Hybrid Typing of Secure Information Flow in a JavaScript-Like Language. (01 2016). https://doi.org/10.1007/978-3-319-28766-9_5

[27] Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a Program Logic for JavaScript. *Sigplan Notices - SIGPLAN* 47 (01 2012), 31–44. https://doi.org/10.1145/2103621.2103663

[28] Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood. 2015. A Trusted Mechanised Specification of JavaScript: One Year On, Vol. 9206. 3–10. https://doi.org/10.1007/978-3-319-21690-4_1

[29] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. *ECOOP, Lec- Ture Notes in Computer Science* (06 2010), 126–150. https://doi.org/10.1007/978-3-642-14107-2_7

[30] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. (03 2014). https://doi.org/10.1145/2554850.2554909

[31] Dongseok Jang and Kwang-Moo Choe. 2009. Points-to analysis for JavaScript. *Proceedings of the ACM Symposium on Applied Computing* (01 2009), 1930–1937. https://doi.org/10.1145/1529282.1529711

[32] Luís Loureiro. 2021. *ECMA-SL - A Platform for Specifying and Running the ECMAScript Standard.* Master's thesis. Instituto Superior Técnico.

[33] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proceedings of the ACM on Programming Languages* 1 (10 2017), 1–24. https://doi.org/10.1145/3133910

[34] Sergio Maffeis, John Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. 307–325. https://doi.org/10.1007/978-3-540-89330-1_22

[35] Sergio Maffeis, John Mitchell, and Ankur Taly. 2009. Isolating JavaScript with Filters, Rewriting, and Wrappers, Vol. 5789. 505–522. https://doi.org/10.1007/978-3-642-04444-1_31

[36] Sergio Maffeis and Ankur Taly. 2009. Language-Based Isolation of Untrusted JavaScript. *Proceedings - IEEE Computer Security Foundations Symposium*, 77–91. https://doi.org/10.1109/CSF.2009.11

[37] Changhee Park and Sukyoung Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. (01 2015). https://doi.org/10.4230/DARTS.1.1.12

[38] Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. *ACM SIGPLAN Notices* 50 (06 2015), 346–356. https://doi.org/10.1145/2813885.2737991

[39] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based Semantics Extraction Toolchain. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* 647–658. https://doi.org/10.1145/3324884.3416632

[40] Joe Politz, Matthew Carroll, Benjamin Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. *ACM SIGPLAN Notices* 48 (10 2012), 1–16. https://doi.org/10.1145/2384577.2384579

[41] Gabriela Sampaio, José Fragoso Santos, Petar Maksimović, and Philippa Gardner. 2020. A Trusted Infrastructure for Symbolic Analysis of Event-Driven Web Applications. (11 2020), 15–16.

[42] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (08 2013). https://doi.org/10.1145/2491411.2491447

[43] Sorhus, Sindre. [n.d.]. p-map (GitHub). https://github.com/sindresorhus/p-map. Accessed on 2021-10-31.

[44] Peter Thiemann. 2005. Towards a Type System for Analyzing JavaScript Programs. *Lecture Notes in Computer Science* 3444 (04 2005). https://doi.org/10.1007/978-3-540-31987-0_28

[45] W3C. [n.d.]. DOM Core Level 1 Specification. https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/level-one-core.html. Accessed on 2021-10-31.

[46] W3C. [n.d.]. UI Events. https://www.w3.org/TR/uievents/. Accessed on 2021-10-31.

[47] Wheeler, Ken and Spampinato, Fabio. [n.d.]. cash (GitHub). https://github.com/kenwheeler/cash. Accessed on 2021-10-31.