MetaData262: Computing and Visualizing Test262 MetaData

Diogo Costa Reis
ist187526
diogo.costa.reis@tecnico.ulisboa.pt

Instituto Superior Técnico
Av. Rovisco Pais, 1
1049-001 Lisboa
Tel: +351 218 417 000
mail@tecnico.ulisboa.pt

Abstract. The ECMAScript standard is the specification of the JavaScript language which is one of the most popular programming languages in the world. Implementing the ECMAScript standard is challenging in various ways, both due to its complexity and its scale. Test262 was created to alleviate this difficulty by providing a comprehensive test suite that allows developers to test their implementations. However, Test262 does not come to a structured description of its tests, making it impossible to filter tests by the JavaScript features they target. The MetaData262 v0 system was created to help with this issue, providing a database containing the metadata of Test262 tests and a collection of scripts to compute that metadata. With this thesis, we will improve MetaData262 v0 in two separate ways. First, we will re-engineer the system for metadata computation, making it more precise and performant and enriching the computed metadata with additional information. Second, we will develop a web interface for the visualization of Test262, allowing developers to easily filter tests by feature.

Keywords: ECMAScript · Specification Language · Reference Interpreters · Test262

Table of Contents

1 Introduction

JavaScript (JS) is a programming language mainly used in the development of client-side web applications, also being one of the most popular programming languages. According to both GitHub and StakeOverflow statistics, JavaScript finished 2021 as the second most active language on GitHub[1] as well as on StackOverflow.[2]

JavaScript is defined in the `ECMAScript` standard[3], a document written in English. The standard has been growing with each new version, both in size and complexity. The growth of the standard is illustrated in Figure 1. This continuous growth in complexity makes the JavaScript a language very hard to implement with many corner cases. `Test262`[14] was created to aid the development of JavaScript engines by providing a conformance test suite that developers can use to test their implementations. As the standard is extensive and complex, a conformance test suite for JavaScript must necessarily be very large given that it must cover all the corner cases of the standard. The standard being complex and extensive also leads to many of its implementations being only partial, particularly those made in academic contexts, for example JSRef[17] and KJS[34].

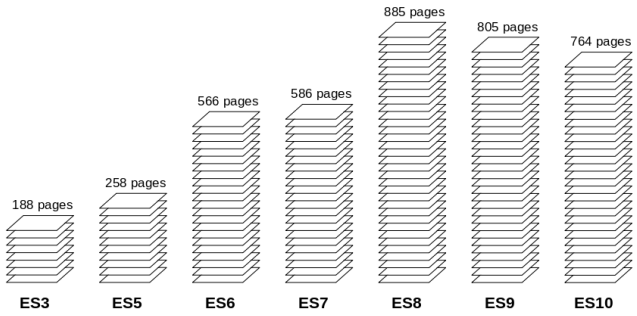

Fig. 1: Evolution of ECMAScript standard's number of pages

When implementing a JavaScript engine or a JavaScript analysis, it is very important to know the tests that should pass for the specific feature being implemented. The selection of applicable tests for partial implementations is particularly difficult since tests for one feature may or may not use other features that are not implemented. For instance, a test targeting the Math library might use features of the array library. `MetaData262 v0` is a project developed at IST with the goal of facilitating this problem. In particular, MetaData262 v0 processes the

[1] Second most utilized language based GitHub pull requests - https://madnight.github.io/githut/

[2] Tendencies based on the Tags used - https://insights.stackoverflow.com/trends

`Test262` test suite and calculates metadata about the tests, storing that metadata in a NoSQL database that can be easily inspected. `MetaData262 v0` stores for each test various properties, such as the JavaScript version to which the test belongs, the syntactic constructs that are used in the test, and the built-ins that are used by the test.

In this thesis we will improve `MetaData262 v0` in two ways. First, we will develop a `MetaData262 v1`, a new version of MetaData262 with an improved infrastructure for computing the metadata of Test262 tests, making it more precise and extending it with additional information. Second, we will build `MetaData262Viz`, a web application for the visualization of the metadata that will allow the developers to easily select tests per feature and obtain statistics regarding the selected testes.

The improvements to `MetaData262 v0` will make the calculations more precise and performant. The precision improvements we are planning on will affect the version calculations and the built-in calculations. The improvements to the version will come from adding a parser that supports a later version of the `ECMAScript` standard and more JavaScript engines. The built-in will need a newer JavaScript parser as well and a revision of the *dynamic approach*. The performance improvements will come from parallelizing the docker containers in which the JavaScript engines used to compute the version metadata will run. The docker containers used will make the calculation easier to replicate at the same time. We will also enrich the metadata `harness` metadata. In particular, we will compute the `includes` that will hold the `harness` files the test uses and the `harness-functions` that will store the exact functions of the `harness` files are used.

The web interface we will build, also known as `MetaData262 Visualization System`, has the objective of making the interaction with the results of the `MetaData262 v0` calculations simpler and more accessible. The web interface will allow the user to filter the `Test262` metadata by its versions and built-ins. The output will display the metadata of the tests and statistics of the metadata.

This document is structured in the following way. Section 2 has the objectives of this thesis. Section 3 gives an overview of the field of the thesis, describing the `ECMAScript` standard, the `Test262` test suite, and the project `MetaData262 v0`. Next, Section 4 presents the scientific work done that relates to this thesis topic. Section 5 presents the design and methodology that will be be followed in this thesis. As for Section 6, it describes how we plan to evaluate the results of this thesis as well as its timeline. Finally, Section 7 presents the main conclusions and a summary of this proposal.

2 Goals

This thesis has two main goals. The first is to improve `MetaData262 v0` and the second is to create a web visualization system to make the metadata more accessible. In the end, we hope that developers of JS engines and analyses will

use the MetaData262 Visualization System for filtering Test262 tests per feature, making the test selection process more efficient, effective, and reliable.

MetaData262 v0 improvements This goal is to improve the precision and performance of MetaData v0 making it more complete, scalable, and easier to replicate.

MetaData262 Visualization System This goal aims to create a web platform to easily visualize the metadata calculated by `MetaData262 v0`. The system will provide filters with which the developer will be able to filter Test262 tests by their features. Then, the system will output the filtered tests in various formats as well as useful statistics about them.

3 Background

This chapter provides an overview on the ECMAScript standard, the Test262 that are used to test the correct implementation of the ECMAScript standard, and finally an outline of the new metadata generated.

3.1 ECMAScript

ECMAScript standard is the official document, written in the English language, in which the JavaScript language is defined. This document is in constant evolution, being updated by the ECMA Technical Committee 39 (TC39), which is responsible for maintaining the standard. The standard is currently in its twelfth version. The standard specifies the `JavaScript` language, to ensure its multiple compilers and interpreters implementations are coherent. Some of the `JavaScript` compilers are the Hop [5] and the JSC [6] compilers, the most popular interpreters are Node.js [9] and SpiderMonkey [13]. These are only four implementations among many others, which come along with the many use cases that `JavaScript` has. `JavaScript` is mostly used in the web context, both client-side within browsers and server-side, but also in embedded devices. Since `JavaScript` is used in so many scenarios and across so many different contexts, it is highly important that ECMAScript standard is defined in great detail to ensure consistency. Browsers, for example, need to run `JavaScript` implementations that coincide so that websites are correctly rendered and exhibit the same behavior. In order to achieve coherent implementations, the standard defines the types, values, objects, properties, syntax, and semantics of `JavaScript` that must be the same in every `JavaScript` compiler and interpreter, while allowing `JavaScript` implementations to define additional types, values, object, properties, and functions.

The `JavaScript` language can be divided into three major components, those being expressions and commands, built-in libraries, and finally internal functions.

– Expressions and commands describe the behavior of static constructions, detailing the semantics of the diverse expressions (e.g., assignment expressions,

built-in operators, etc.), commands (e.g., loop commands, conditions command, etc.), and built-in types (Undefined, Null, Boolean, Number, String and Object).
- The internal functions of the language are used to define the semantics for both expressions and commands, as well as the built-in libraries. Internal functions are not exposed beyond the internal context of the language. In other words, no JavaScript program uses internal functions directly.
- Finally, built-in libraries encompass all the internal objects available when a JavaScript program is executed. Internal objects expose many functions implemented by the language itself, including functions to manipulate numbers, text, arrays, objects, among other things.

The remaining subsection provides a description of the three types of artifact described in the standard.

Semantics of IF statement Figure 2 shows a snippet of the ECMAScript standard description of the `IF` command. In order to evaluate `IF` commands with the shape:

```
if (Expression) Statement1 else Statement2
```

the language begins by evaluating the `Expression` storing the result in the variable `exprRef` (step 1). The previous step will be used as Boolean, therefore, the result of the previous step will be converted to a Boolean using the internal functions `ToBoolean` and `GetValue`, and having the result stored in the variable `exprValue` (step 2).A different `Statement` will be followed depending on `exprValue`. If `exprValue` has the value `true` the variable `stmtCompletion` will have the evaluation of the first `Statement` (step 3). Otherwise, the variable `stmtCompletion` will store the result of evaluating the second `Statement` (step 4). Finally, a `Completion` will be returned, if the `stmtCompletion` has non empty value then it will be returned, however, when the value is empty it will be replaced with undefined (step 5).

IfStatement : **if (** *Expression* **)** *Statement* **else** *Statement*

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *exprValue* be ! ToBoolean(? GetValue(*exprRef*)).
3. If *exprValue* is **true**, then
    a. Let *stmtCompletion* be the result of evaluating the first *Statement*.
4. Else,
    a. Let *stmtCompletion* be the result of evaluating the second *Statement*.
5. Return Completion(UpdateEmpty(*stmtCompletion*, **undefined**)).

Fig. 2: ECMAScript definition of an if-else statement

Semantics of the Pop function The Array built-in is an object as any other in JavaScript. The main difference is in its properties. Array Objects have a property `length` that contains the size of the array, as well as a property for each element of the array (from zero to `length` minus 1).

Figure 3 shows a simplified version of an array performing the pop function, where `(a)` and `(b)` are the before and after respectively. Before performing `pop (a)`, the array has three properties `length`, `0`, and `1`. Property `length` represents the size of the array that has value `2`, while the properties `0` and `1` store the first (`banana`) and second (`kiwi`) elements of the array respectively. After `pop` is performed `(b)`, the last element is of the array is removed (highlighted in red at `(a)`) and the `length` property (highlighted in green) is decremented by one since the size of the array changes to one.

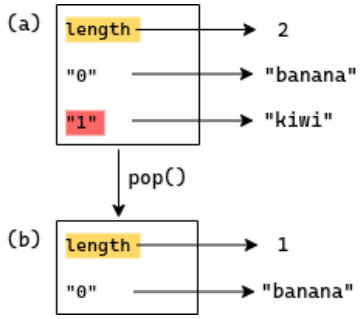

Fig. 3: Example Array.pop

Figure 4 shows a snippet of the ECMAScript standard description of the pop function in the Array Built-in. To begin with, the array will be converted to and Object using the `ToObject` function, and stored in the `O` variable (step 1). Afterwards, the array length of the previously calculated variable will be calculated with the `LengthOfArrayLike` internal function, and storing the result in the `len` variable (step 2). At this point there are to ways to proceed depending on the value of `len`. If the value is zero, the Array is empty, then the property `length` of `O` is set to zero and `undefined` is returned (step 3). Otherwise, when `len` is different from zero, meaning that the Array is not empty, the Array's last element will be removed (described in Figure 3) and returned (step 4). To begin with, the language will assert that `len` is positive (step 4.a). Afterwards, the `newLen` variable will store the value of `len` decremented by 1 (step 4.b). The variable `index` will store the variable calculated in the previous step represented as a String converted with the `toString` function (step 4.c). Then, stores the value of the `O` variable at the property corresponding to `index` in the `element` variable using the `Get` function (step 4.d). Subsequently, deletes the previously mentioned property of the `O` variable with the `DeletePropertyOrThrow` function (step 4.e). In addition, sets the `length` property of the `O` variable to the `newLen`

using the `Set` function (step 4.f). Finally, returning the value of the variable `element` (step 4.g).

1. Let *O* be ? ToObject(**this** value).
2. Let *len* be ? LengthOfArrayLike(*O*).
3. If *len* = 0, then
    a. Perform ? Set(*O*, **"length"**, **+0**$_{\mathbb{F}}$, **true**).
    b. Return **undefined**.
4. Else,
    a. Assert: *len* > 0.
    b. Let *newLen* be F(*len* - 1).
    c. Let *index* be ! ToString(*newLen*).
    d. Let *element* be ? Get(*O*, *index*).
    e. Perform ? DeletePropertyOrThrow(*O*, *index*).
    f. Perform ? Set(*O*, **"length"**, *newLen*, **true**).
    g. Return *element*.

Fig. 4: ECMAScript definition of Array.pop

LengthOfArrayLike internal function Figure 5 shows a snippet of the ECMAScript standard description of the `LengthOfArrayLike` internal function, that evaluates the function:

```
LengthOfArrayLike (obj)
```

The language starts by asserting that `obj` is an `Object` (step 1). Afterwards, gets the value of the property `length` from `obj` using the function `Get`. Then, converts the previously mentioned value to an Integer that represents the length with the `ToLength` function, and finally returns said Integer (step 2).

1. Assert: Type(*obj*) is Object.
2. Return ℝ(? ToLength(? Get(*obj*, **"length"**))).

Fig. 5: ECMAScript definition of the LengthOfArrayLike

3.2 Test262

Implementing a `JavaScript` engine is particularly difficult since it involves dealing with the many corner cases that exist in the language. To test that corner cases are correctly dealt with there is `Test262`, the ECMAScript standard

test battery. Although, `Test262` is vital to the `JavaScript` engines, it is very hard to maintain due it's complexity, the total number of tests is around 40000 divided into 87 subfolders, each correspond to roughly one section of the standard. `Test262` complexity grows with changes to the standard since in most cases backward compatible is maintained except for a few select cases.

Due to the ECMAScript standard being so extensive most implementations are only partial, especially implementations and analysis developed in academic contexts. In order to test partial implementations, one must be able to obtain the applicable set of tests from all the tests contained in Test262. Selecting the applicable tests is not a trivial matter because there are too many tests and too many features. The current methodology is that each development team manually selects the tests that are applicable to their corresponding implementation. This raises the problem that there is not standard and precise way of picking the all the right tests from the almost 40000 tests in `Test262`, making the possibility of human error when selecting the applicable tests likely.

Figure 6 shows a test from `Test262`. Every test of `Test262` has 3 parts: first is the copyright section represented with the comment `//` (lines 1 and 2), second is the `Frontmatter` section between `/*---` and `---*/` with some metadata about the test (lines 4 to 7), and finally is the `Body` section with the code of the test (lines 9 to 13). The copyright section has information about the owner and license of the test. The `Frontmatter` section has the test id (15.4.5-1) and a description of the test. Finally, the `Body`'s code tests the correct implementation of the standard.

```
1   // Copyright (c) 2012 Ecma International.  All rights reserved.
2   // This code is governed by the BSD license found in the LICENSE file.
3
4   /*---
5   es5id: 15.4.5-1
6   description: Array instances have [[Class]] set to 'Array'
7   ---*/
8
9   var a = [];
10  var s = Object.prototype.toString.call(a);
11
12  assert.sameValue(s, '[object Array]',
13      'The value of s is expected to be "[object Array]"');
```

Fig. 6: Test262 es5id: 15.4.5-1

The `Frontmatter` has keywords to hold metadata of the test. These keywords are associated with specific elements of metadata concerning the test. Bellow is the list of possible keywords and their meaning:

– `description` - contains a short description about what will be tested;

- `esid` - contains the hash identifier of the ECMAScript portion associated with the feature that will be tested (the identifier references the most recent version of ECMAScript when the test is created);
- `info` - contains a deeper explanation of the test behavior, frequently includes a direct citation of the standard;
- `negative` - indicates that the test throws an error; associated to the keyword will be the type of error the test is supposed to be thrown (e.g. `TypeError`, `ReferenceError`) as well as the phase in which the error is expected to be thrown (e.g. `parse` vs `resolution` vs `runtime`);
- `includes` - contains the list of `harness` files that should be included in the execution of the tests (`Test262` makes use of a large number of auxiliary function defined in a dedicated library referred to as the `Test262 harness` described later in this section);
- `author` - contains the identification of the author of the test;
- `flags` - contains a list of booleans for each test property, the properties being: (1) `onlyStrict`, the test is only executed in strict mode; (2) noStrict, the test will only be executed in mode *sloppy*; (3) `module`, the test must be integrated as a `JavaScript` module; (4) `raw`, executes the test without any modification, which implies running as `noStrict`; (5) `async`, the test is contains asynchronous functions; (6) `generated`, the test generates the files specified by the property; (7) `CanBlockIsFalse` and (8) `CanBlockIsTrue`, the test will run if the property `CanBlock` of the `Agent Record` executing it is false and true respectively; (9) `non-deterministic`, indicates that the semantics used in the test are intentionally under-specified and therefore the test passing or failing should not be regarded as an indication of reliability or conformance;
- `features` - contains a list of features that are used in the test;
- `es5id` and `es6id` - indicates that the feature being tested belongs to ECMAScript 5 and 6 respectively and contains the hash identifier of the section of the standard it belongs to; these keywords have been deprecated and substituted by esid.

As Figure 5 illustrates, it is often the case that the metadata of a test is incomplete. Some tests also have the wrong metadata. As part of this thesis, we plan to process all the tests to check and correct their corresponding metadata as well as completing the metadata that is missing.

The example in Figure 6 has 2 keywords, description and the deprecated es5id. Besides the obvious upgrade from `es5id` to `esid` it would be useful to have `includes` with the `harness` files needed to execute the test. The `harness` information is very useful since it makes it easy to identify the part of the `harness` needed to run that test, opening the door for loading only part of the `harness` instead of the whole `harness` which is the current approach.

New Metadata Besides the metadata that Test262 tests currently include, it would be useful for them to have additional information regarding:

- `syntactic construct` - list of all syntactic constructions used in the test;

- `version` - the ECMAScript version of the standard in which the feature being tested was introduced;
- `built-ins` - list of all the built-ins used in the test.
- `harness-functions` - list of all the harness functions required to run the test.

This above metadata critical for filtering the tests when considering partial implementations of the language. For instance, if a JS engine only supports the 5th edition of the standard, one must be able to obtain all the corresponding tests for that `version`. Analogously, if a JS engine only implements certain `built-in` objects, one has to be able to filter out the tests that make use of the `built-in` objects that it does not implement. This would provide consistency and standardization to the selection of applicable tests to a partial implementation of the standard. As for the `harness-functions`, it provides important information about the functions of `harness` that are used in the test. That information is relevant because only a small part of the `harness` is need in each test even though the whole library is loaded and tested. The harness has a total of 7290 lines in 32 files and is tested by 96 tests. By only including the exact functions that a test requires, one can speedup the testing process significantly.

3.3 MetaData262 v0

This thesis continues the master thesis of Miguel Trigo[42], in which he developed a preliminary version of `MetaData262`. More specifically, he built a MongoDB database storing the metadata of all Test262 tests, representing the metadata of each test as a JSON object.

In `MetaData262 v0`, each test is associated with a JSON object storing the various metadata properties of the test and their corresponding values, with those metadata properties being the keywords of the `Frontmatter` mentioned in the previous section. Besides the existing metadata properties, various new properties were added to `MetaData262 v0`. We can divide these new properties into 3 main groups: location properties, extra front matter properties, and statistics properties. Each of these groups will be discussed next.

Location group The location group contains information about the path to the test inside the `Test262`. It consists of the properties `path`, storing the relative path to the test starting at the root of the `Test262` project, and splitPath, storing the array obtained by splitting the `path` string into its corresponding folders.

Extra front matter group The extra front matter group contains new metadata generated for `MetaData262 v0`. The new metadata generated is associated with the properties: `syntactic_constructs`, `version`, and `builtIns` that were mentioned in the previous section as important metadata to add. The property `version` stores the `ECMAScript` standard version the test belongs to. The property `syntactic_construct` stores an array with all the syntactic constructions

that the test contains. As for `builtIns`, it stores all the built-ins that the test interacts with, mapping each built-in to its fields and methods used by test.

Statistics group Finally, the statistics group contains some statistical data about the tests. It contains the properties: `asserts`, `error`, `esprima`, and `lines`. The `asserts` property holds the amount of `assert` statements in the test. The `error` property stores the amount of calls made to `Test262` error functions. The `esprima` property stores a boolean value indicating whether or not the Esprima[4] parser supports the test. `Esprima` is a standard-compliant `ECMAScript` parser fully developed in `ECMAScript`; `Esprima` fully supports up to version 7 of the standard. The property `lines` holds the number of lines of code the test has, excluding comments and empty lines.

Example Figure 7 shows the JSON object storing the metadata generated from the Test262 test given in Figure 6. The JSON object begins with the `path` property storing the path to the test at hand. After, the `version` property indicates that this test belongs to the fifth version of the standard. If `MetaData262 v0` was not able to compute the version of the test, this field is omitted. Next, we have the properties corresponding to the test's `Frontmatter` (note that the deprecated `es5id` is replaced to the `esid` property). Next we have the `static_constructs` property that holds an array with the syntactic constructors of the test. The `builtIns` property holds a JSON object, in which, each property corresponds to a built-in mapped to its functions and fields that are interacted with in the test. Finally, we have the statistics group of properties. The `asserts` property indicates that this test uses one assert statement. The `error` holds the numeric value zero since this test uses no error functions of `Test262`. The `esprima` property indicates this test is supported by `Esprima`. Finally, the `lines` property indicates that this test has 3 lines of code. Note that the example in Figure 6 splits the last line of the test in order to improve the readability of the figure hence only 3 lines are counted in the actual test).

Metadata Computation As part of the setting up of MetaData262 v0, M. Trigo had to compute the missing `Frontmatter` properties as well as the values of the new properties that he introduced. In most cases, the computation was straightforward, only involving a simple syntactic analysis of each test. However, the computation of the version and the built-ins used in each test was more involved. We will go through this process in more detail below.

Calculation of Version metadata The `version` of the standard a test belongs to is calculated using 3 different approaches: dynamic, static, and hybrid.Below we give a brief overview of each method:

– *Dynamic Approach:* The dynamic approach is based on the waterfall model, running the tests in various JS engines, each corresponding to a specific version of the standard. As Figure 8 illustrates, `MetaData262 v0` starts by running each test in the JS engine corresponding to version 5 of the `ECMAScript`

```
{
    "path": "./test262-main/test/built-ins/Array/15.4.5-1.js",
    "version": 5,
    "esid": " 15.4.5-1",
    "description": " Array instances have [[Class]] set to 'Array'",
    "built-ins": "Array",
    "Array": "15.4.5-1.js",
    "syntactic_construct": [
        "Identifier",
        "ArrayExpression",
        "VariableDeclarator",
        "VariableDeclaration",
        "MemberExpression",
        "CallExpression",
        "Literal",
        "ExpressionStatement",
        "Program"
    ],
    "builtIns": {
        "Array": [],
        "Object": [
            "prototype",
            "prototype.toString"
        ],
        "Function": [
            "call"
        ]
    },
    "asserts": 1,
    "error": 0,
    "esprima": "supported",
    "lines": 3
}
```


Fig. 7: Metadata generated for test esid: 15.4.5-1.js

standard (represented as ES). If the test output is correct then the test is labelled as belonging to version 5 of the standard, otherwise the test will be run in the engine implementing the next version of the standard. This process repeats until the last engine, associated with version 11 of the standard. If the test output is not the expected again, then the test is not supported.

– *Static Approach:* The static approach determines the version of a test by analyzing the static keywords of the test. Essentially, a test is labeled with the version corresponding to the keyword with the latest associated version; formally:

$$version(test) = max\{version(keyword) | keyword \in test\}$$

For instance, if a test contains two keywords introduced in versions 6 and 8, then the test will belong to the version 8 of the standard. `MetaData262 v0` implements the static approach in the following way. It maintains an internal

map associating each keyword with the version in which it was introduced. Given a test to be labeled, `MetaData262 v0` first computes the AST of the test using the `Esprima` parser and then traverses the AST to obtain all the keywords used in the test. Finally, it labels the test with the version of the latest keyword (as discussed above).

– *Hybrid Approach:* The hybrid approach is calculated using the results of the dynamic and static approaches. The hybrid approach merges the results by maintaining the higher version detected between each approach for each test. For instance, if the dynamic approach's result for a test is version 6 and the static's result for the same test is version 8, then, the test is labeled as belonging to version 8 of the standard according to the hybrid approach.

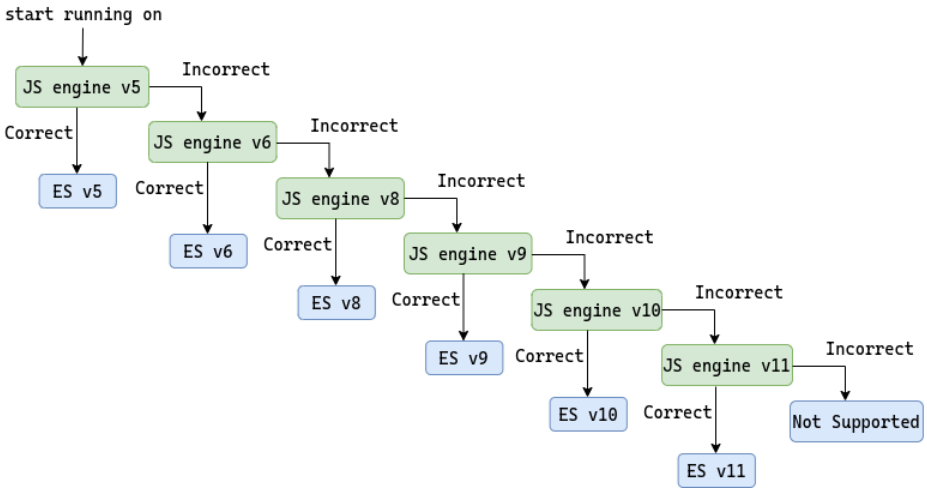

Fig. 8: Waterfall model of the dynamic approach to calculate the ECMAScript version

Calculation of built-ins metadata The `built-ins` used by each test are calculated using two separate approaches: dynamic and static. As in the calculation of the `ECMAScript` version both approaches are combined to improve the results. Below, we describe each approach separately:

– *Static Approach:* The static approach maintains an internal map associating each built-in method and property with the name of the corresponding built-in object (e.g. the method hasOwnProperty is associated with the built-in object `Object`). The static approach first computes the AST of the test using `Esprima` and then traverses the AST checking for each method and property for the built-in object it is associated with. In the end, it outputs

a JSON object mapping each built-in object to an array with the names of its methods and properties that the test uses.
– *Dynamic Approach:* The dynamic approach works by running the actual code of the test and logging the interactions of the test with the JavaScript built-ins. To this end, `MetaData262 v0` wraps the methods of every built-in object in an auxiliary function that calls the corresponding methods and adds an entry to the log registering that call. Finally, `MetaData262 v0` analyses the log information to determine the built-in methods used by the test.

4 Related Work

Scientific studies about analysis of programs and techniques of instrumentation of JavaScript encompasses a varied range of specifications such as: type systems ([40], [21], [28], [18], [36], [16], [39]), abstract interpreters ([30], [43], [19], [37]), *points-to* analysis ([22]), Hoare logic ([23], [31]), operational semantics ([17], [34], [20]), intermediate representations and compilers ([33]). Next, we have the related work most relevant for this thesis, it is organized into two Tables **??** and **??**. This time line focuses on the most important investigation work, focused on the JavaScript language specification since its creation in 1995.

In 1995, JavaScript is launched and implemented in Netscape Navigator 2.0 beta 3. In 2005, Thiemann presents the first try at defining a system of types possible for JavaScript [41]. The system detects the possible characteristics of an object and signals suspected type conversions. The work done does not cover important aspects of JavaScript, such as inheritance based on prototypes, and was not created along with an implementation at the time of the publication. In 2008, Maffeis et al. proposes a first operational semantics for the language ECMAScript version 3. (ES3) [32]. This first semantics was used to analyze various properties of the security of web applications. In 2010, Guha et al. [29] defines a language λJS, a formal calculation that encapsulates the fundamental characteristics of ES3. λJS comes along an interpreter *Racket* [11] and a translator from ES3 programs to expressions λJS. In 2012, Gardner et al. [26] adapts the logic ideas of separation to build a scalable Hoare logic for a subset of JavaScript, based of semantics faithful to ES3. In 2012, Politz et al. [35] extends the language λJS of ES3 to ES5 included, creating for the first time, a formal semantics for JavaScript `property descriptors` and complete handling of the `eval` declaration. In 2014 Bodin et al. presents *JSCert* [17], a formalization of the ES5 standard written in *Coq* [1], and *JSRef* an executable interpreter of references extracted from *Coq* to *OCaml* [10], proving correct when related to *JSCert* and tested using the test suite Test262 [14]. In 2015, Park et al. presents *KJS* [34], a formal semantics for ES5, the most complete to that moment. The *KJS* was developed using the framework *K* [7], a consolidated system of rewriting of terminology that supports various types of symbolic analyses based on *reachability logic* (All-Path Reachability Logic [38]). *KJS* was tested carefully using the test suite, passing all the 2.782 tests of the language core. The *KJS* can be executed symbolically, being able to be used for formal analyses and verification of ES5

programs. In 2015, Gardner et al. [27] extend the project *JSCert* to support *arrays* ES5. In order to do this, the authors connected the *JSRef* to the implementation of the Array library of the V8 engine [15] from Google. Additionally, the authors evaluated the state of the project *JSCert* at the time, giving a detailed analysis of the methodology as a whole and a detailed analysis of the tests that the project passed or failed. In 2017, Fragoso Santos et al. creates the tool *JaVert* [23]: a tool for verifying JavaScript that allows semiautomatic reasoning about the fundamental properties of correctness of ES5 programs. JaVert does not assume any simplification of the language semantics, analyzing all the *corner-cases* of the language. The load of necessary notation is substantial. In 2018, Charguéraud et al. presents *JSExplain* [20], an reference interpreter for the ES5 language that follows the specifications and produces an inspectable execution traces. It also includes an interface of code inspection that allows the programmer to execute his ES5 code step by step as well as the code of the ES5 interpreter, while executing ES5 programs. In 2020, Sampaio et al. [24] extend the JaVerT tool to support automatic reasoning about JavaScript events. In order to do this, the authors developed a reference implementation of JavaScript promises directly in JavaScript. The authors used it to test their reference implementation, passing 344 tests of the 474 total dedicated to JavaScript promises. Around 106 testes were excluded due to ES6 features that were not yet supported and to tests that require *non-strict* mode. In 2020, Jihyeok Park et al. presents JISET [25] the first tool that synthesizes automatically the analyzer and that translates AST-IR directly from a specification of a given language.

5 Design and Methodology

We recall that this thesis has two goals. Firstly, we want to improve `MetaData262 v0` in two ways: add support for missing metadata elements, and improve the infrastructure for the metadata computation, making the obtained metadata more reliable and the process faster and reproducible. Secondly, we want to create a visualization system for easy selection of tests and display of test statistics.

5.1 Improvements to MetaData262 v0

The metadata calculated by `MetaData262 v0` is flawed in several ways: in the version calculation, in the built-in calculation, and in the harness calculation. We describe these flaws and how to fix them in more detail next.

– *Version Calculation* - The version calculation is flawed in both the static and dynamic approaches, meaning that the hybrid approach that uses the results of the previous two will also be compromised. The static approach is flawed since the `Esprima` parser only fully supports up to `ECMAScript` version 7. Improvements can be made by finding a `JavaScript` parser that supports a later version of the standard. The dynamic approach's flaw lies in the way it works, since there is no JS engine that fully supports a version of

the standard and only that one. To improve the dynamic approach we will add more JS engines. By adding more data points to the calculation, we will increase the reliability of the obtained metadata.
- *Built-in Calculation* - The built-in calculation is also flawed in both the static and dynamic approaches. The static approach has the same flaw since it also uses `Esprima` as parser. Therefore the same improvement can be made by finding a newer parser. As for the dynamic approach, it is flawed due to the fact that only function calls are logged and therefore detected. If the properties of a built-in are changed directly then it will not be detected. For example if a test performs `Array.prototype.pop = 3;` then the property `prototype` of the Array built-in will be changed even though no function call was made. This can not be detected by the current dynamic approach.
- *Harness Calculation* - The `harness` calculation is not done, including only the little information available in the `Frontmatter` of a few tests. This calculation can be improved by completing `includes` metadata and adding an additional `harness-functions` metadata. Both `includes` and `harness-functions` are mentioned in earlier sections, the first stores the files of the `harness` needed to run the test and the second will store the functions from those files that are actually used in the test.

The improvements the `MetaData` calculation precision, in particular the version dynamic approach, will enhance the issue of performance. This issue will be improved with the use of `docker`[2] containers and this way we will also improve the ease to replicate the calculations of the `MetaData`. The duration to calculate the version with the dynamic approach scales linearly with the amount of engines per standard version being used. In `MetaData262 v0` only two JS engines are being used and the dynamic approach already takes multiple hours to run. With the use of `docker` containers we will be able to parallelize the work of the dynamic approaches by multiplying the docker images that run the JS engines. Containers running in parallel will not only improve the performance but also allow MetaData262 to be replicated in an easier way since it requires a smaller configuration effort to replicate the `MetaData` calculation.

5.2 MetaData262 Visualization System

The `MetaData262 Visualization System` appears to ease the barrier to access the `MetaData262 v0` information and statistics. Right now the metadata is hosted in a MongoDB database leading to the need to have knowledge of MongoDB to get the information. After getting the information there would most likely be a need to process that metadata and maybe even generating charts to provide statistics. This is visualization system will make this process much easier.

The `MetaData262 Visualization System` is in its core a web application that allows users visualize and filter the `Test262` metadata easily. The system will also update itself whenever there are changes in `Test262`. Users are able to filter the metadata by the built-ins and version of the test. Users will also be

able visualize the queried information in different ways: `PATH`, `JSON`, and `STATS`. The `PATH` will only show the path to the tests in `Test262`. The `JSON` will allow the users to see the full metadata calculated for the tests. Finally, the `STATS` will show some statistics about the tests queried. To ensure that the information is up to date, this system needs to update itself since this is a thesis project and there is no active developer team. In order to accomplish that, the system will need check for changes, be it in the form of new tests or changes to existing ones, and calculate the new metadata.

The development of the web application has already began, Figure 9 shows the progress already made. The image has 2 main parts: the filters part, and the information part. The upper part is the filters part, where it is possible to select the filters for querying the metadata. The first filter, `BuiltIn Belongs` allows to select tests from the subfolders of the built-in folder in the `Test262`. The `BuiltIn Contained` enables the user to filter tests by the built-ins the tests can use. The `BuiltIn Belongs` and `BuiltIn Contained` filters are combined using `AND` or `OR` relations. The `AND` relation will only return tests that satisfy both filters. As for the `OR` relation, it will return every test that satisfies either of the filters. Then, we have the `Version` filter that allows filtering the tests by the `ECMAScript` version. Next, we have the `BACKEND SEARCH` and `LOCAL SEARCH` search buttons that will query the metadata database for the tests that satisfy the filters set. The `BACKEND SEARCH` filter the tests in the backend server while the `LOCAL SEARCH` it will filter the tests in the machine that is used to access the website. In the example, the filters selected are `Object` and `Array` for the `BuiltIn Belong` and `BuiltIn Contained` respectively with an `AND` relation and the sixth version is also selected. The information part of the image contains the result of the query. The information can already be displayed in two of the three ways mentioned above, in the image the `PATH` is selected. The image also shows it took 16 milliseconds to have the query results available to the user. The image displays that a total of 32 tests satisfy the filters selected. The image shows the path of the tests selected. Note that not all tests are shown at the same time, in order to improve performance only 10 tests are shown at a time, at the bottom of the image is the pagination menu.

In Figure 9 there are two search buttons because there at the time trade offs being made with either way of searching, the following results were run in the laptop serving the web application. The backend search starts getting slow whenever the amount of tests being returned surpasses 2500 in the local machine, taking on roughly half a second to get the results. The local search has the problem that it takes around 4 seconds to get all the metadata from the server at page load. The local search has the advantage of filtering the tests in a few milliseconds. Caching the metadata in the users browser would allow the metadata to be only loaded one, making the usability of the web application much better.

This web application will allow the user to view charts about the metadata of the tests queried. M. Trigo's thesis he generated the charts shown in Figure 10 and Figure 11 as well as some other pie charts and bar charts. In the web

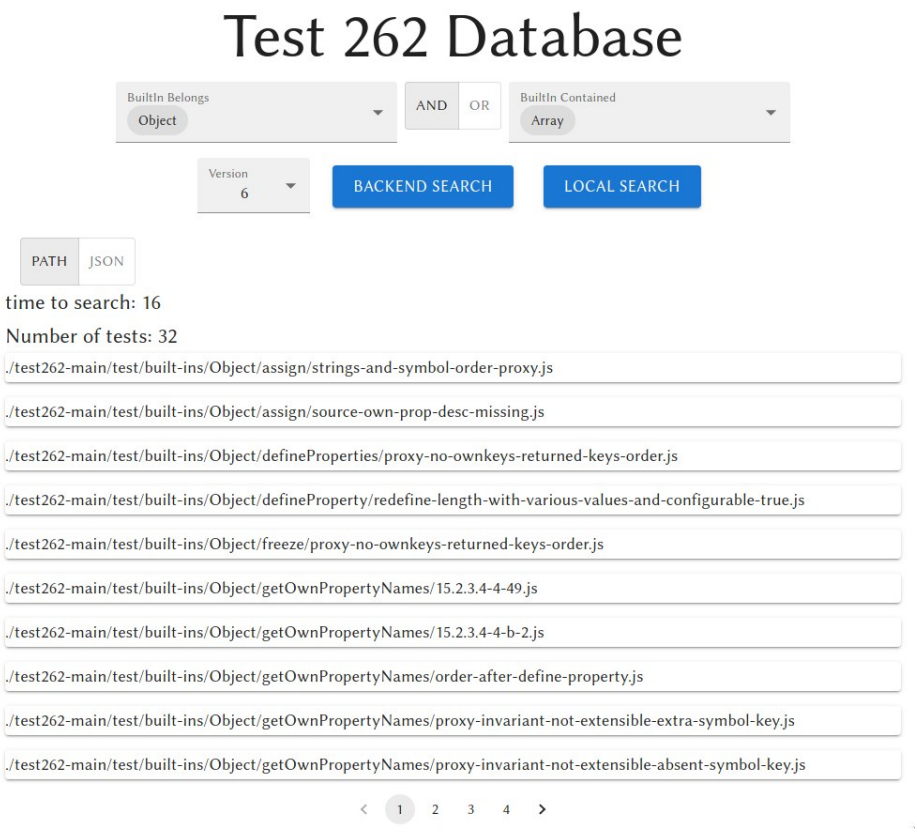

Fig. 9: Website for searching the metadata in construction

application those charts will be generated in real time with the help of external libraries. Especially the charts below, the box and whiskers chart and the stacked bar chart, are not supported by many of the most popular libraries. Two of the libraries that are capable of generating all the charts needed are the `KendoReact`[8] an UI library and `Victory`[12] a chart library. `KendoReact` is a full UI library, charts being only part of its 126 components. Whereas, the `Victory` library has only 34 components and all are directly related with generating charts. The Victory library also seams simpler to use, the community around it seams more active which would be important in case of difficulty while implementing the charts needed.

6 Evaluation and Planning

The evaluation of this thesis will focus on its two main elements: the metadata calculation infrastructure and the `Metadata262 Visualization System`. The metadata calculation infrastructure will be evaluated with respect to its precision

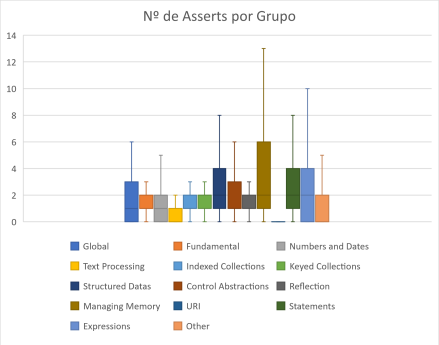

Fig. 10: M. Trigo chart number of asserts by category

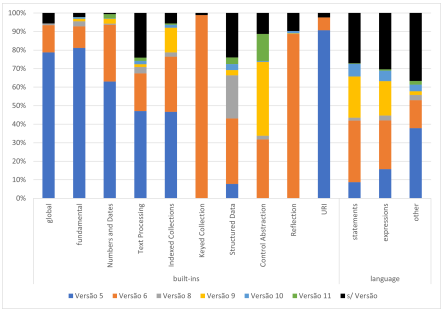

Fig. 11: M. Trigo chart versions by category

and performance. As for the `Metadata262 Visualization System`, it will be evaluated for its reliability. The evaluation criteria will be described in more detail bellow.

Metadata calculation precision The precision of the metadata calculated is very hard to access since there is no ground truth available. The version metadata calculated is being calculated for the first time. To evaluate the precision of the metadata calculation we will analyze a small subset of the Test262 tests manually, comparing the version calculated with their real version. We will use two separate strategies for selecting tests for manual inspection:

- *Random Strategy:* we will select a small subset of tests randomly, guaranteeing uniform coverage of the various test folders;
- *Heuristic Selection:* we will select the tests that exhibit the largest different between the computed version and the version that was the latest at the time the test was added to `Test262`. To do this, we will extend the metadata of the tests with the date in which they were added to Test262 or updated. It is often the case that tests are added for features of previous versions; this happens for instance when Test262 maintainers detect missing coverage issues, in such cases the newly added tests target the version that was active when the corresponding uncovered feature was introduced instead of the current one.

Metadata calculation performance To evaluate performance improvements made to `MetaData262 v0` we will compare the time needed to perform the metadata calculation between the old and new infrastructures. This must be run in the same system and with the least concurrent processes running in order to make sure we have the least interference, this way achieving the most trustworthy results.

Visualization system reliability The visualization system evaluation will have three types of tests: unit tests, end-to-end tests and load tests. The unit tests will ensure the correctness of the system's results. The end-to-end tests will guarantee that the website interface works correctly. Finally, the load tests will allow us to estimate the amount of users that the system can support, this way allowing us to estimate the resources need.

6.1 Planning

The Plan for this thesis is based on the tasks that need to be performed in order to accomplish the objectives defined in Section 2. Thus, this thesis has 4 main task to accomplish: improving the metadata calculation, concluding the visualization system, evaluating the results, and writing the thesis. These tasks are represented in the Gantt diagram in Figure 12 and each task is described in more detail afterwards. Note that the month of August has a grey background due to it being a vacations period.

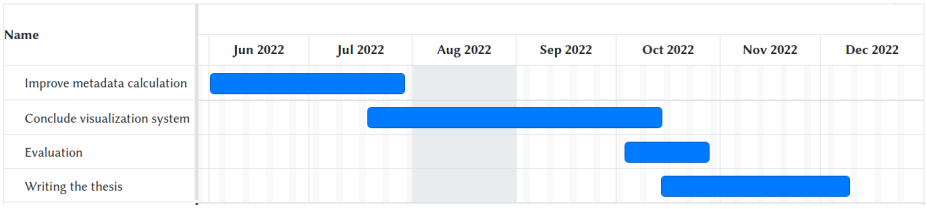

Fig. 12: Gantt diagram with the thesis plan

Improving metadata calculation This task consists of improving `MetaData262 v0` by improving its precision and performance as discussed in Section 5. The precision will be improved with the inclusion of more JS engines and a JS parser that supports more recent versions of the standard. As for the performance, we will parallelize the metadata calculation using `docker` containers.

Concluding visualization system This task consists of concluding `MetaData262 v0` visualization system by adding the possibility to view statistics about the queried tests and improving the system's reliability.

Evaluation This task consists of evaluating the results of the metadata generated. We will manually check the metadata calculated for a small subset of Test262 tests, using the selection strategies discussed in this. The development of the tests for the visualization system and the system itself will overlap during the first two weeks of the evaluation.

Writing the thesis This last task will consist of reporting the execution of the previous tasks, by presenting the results obtained as well as the conclusions reached after finishing this project.

7 Conclusion

With this thesis we will build an infrastructure that enables convenient access to the metadata of the `Test262` test suite. We will improve the metadata calculation of `MetaData262 v0` and make it accessible to users via a web interface. This will enable `JavaScript` engine developers to filter tests easily, in particular developers of partial implementations. This will allow developers to find and correct errors more quickly and also enable a standardization of test selection process for partial implementations.

References

1. Coq - interactive formal proof management system, https://coq.inria.fr/, accessed at 2022-05-27
2. Docker is a set of platform as a service products that uses os-level virtualization to deliver software in packages called containers, https://www.docker.com/, accessed at 2022-05-27
3. Ecmascript - this standard defines the ecmascript 2021 general-purpose programming language, https://www.ecma-international.org/publications-and-standards/standards/ecma-262/, accessed at 2022-05-27
4. Esprima is a high performance, standard-compliant ecmascript parser written in ecmascript, https://esprima.org/, accessed at 2022-05-27
5. Hop - programming environment multitier for javascript, https://github.com/manuel-serrano/hop, accessed at 2022-05-27
6. Jsc - javascript engine that runs on top of the browsers javascript engine, https://github.com/mbbill/JSC.js, accessed at 2022-05-27
7. K - rewrite-based executable semantic framework, http://www.kframework.org/, accessed at 2022-05-27
8. Kendoreact is a professional ui kit to design build business apps with react, https://www.telerik.com/kendo-react-ui/, accessed at 2022-05-27
9. Node.js - javascript runtime built on chrome's v8 javascript engine, https://nodejs.org/, accessed at 2022-05-27
10. Ocaml - general-purpose, multi-paradigm programming language, https://ocaml.org/, accessed at 2022-05-27
11. Racket - general-purpose programming language, https://racket-lang.org/, accessed at 2022-05-27
12. React.js components for modular charting and data visualization, https://formidable.com/open-source/victory/, accessed at 2022-05-27
13. Spidermonkey is mozilla's javascript and webassembly engine written in c++, rust and javascript, https://spidermonkey.dev, accessed at 2022-05-27
14. Test262 - official ecmascript conformance test suite, https://github.com/tc39/test262/, accessed at 2022-05-27
15. V8 - google's open source high-performance javascript and webassembly engine, written in c++., https://v8.dev/, accessed at 2022-05-27

16. Aseem Rastogi, Nikhil Swamy, C.F.G.B.P.V.: Safe efficient gradual typing for typescript (2015). https://doi.org/10.1145/2775051.2676971
17. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised javascript specification. vol. 49, pp. 87–100 (01 2014). https://doi.org/10.1145/2578855.2535876
18. Brian Hackett, S.y.G.: Fast and precise hybrid type inference for javascript (2012). https://doi.org/10.1145/2345156.2254094
19. Changhee Park, S.R.: Scalable and precise static analysis of javascript applications via loop-sensitivity (2015)
20. Charguéraud, A., Schmitt, A., Wood, T.: Jsexplain: A double debugger for javascript. pp. 691–699 (04 2018). https://doi.org/10.1145/3184558.3185969
21. Christopher Anderson, Sophia Drossopoutou, P.G.: Towards type inference for javascript (2005). https://doi.org/10.1007/11531142$_1$9
22. Dongseok Jang, K.M.C.: Points-to analysis for javascript (2009). https://doi.org/10.1145/1529282.1529711
23. Fragoso Santos, J., Maksimović, P., Naudziuniene, D., Wood, T., Gardner, P.: Javert: Javascript verification toolchain. Proceedings of the ACM on Programming Languages **2**, 1–33 (12 2017). https://doi.org/10.1145/3158138
24. Gabriela Sampaio, Fragoso Santos, P.M.e.P.G.: A trusted infrastructure for symbolic analysis of event-driven web applications. European Conference on Object-Oriented Programming (ECOOP 2020) **166**, 1–28 (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.28
25. Gabriela Sampaio, Fragoso Santos, P.M.e.P.G.: A trusted infrastructure for symbolic analysis of event-driven web applications. European Conference on Object-Oriented Programming (ECOOP 2020) **166** (2020). https://doi.org/10.4230/LIPIcs.ECOOP.2020.28
26. Gardner, P., Maffeis, S., Smith, G.: Towards a program logic for javascript. vol. 47, pp. 31–44 (01 2012). https://doi.org/10.1145/2103621.2103663
27. Gardner, P., Smith, G., Watt, C., Wood, T.: A trusted mechanised specification of javascript: One year on. vol. 9206, pp. 3–10 (07 2015). https://doi.org/10.1007/978-3-319-21690-4_1
28. Gavin Bierman, Martím Abadi, M.T.: Understanding typescript (2014). https://doi.org/10.1007/978-3-662-44202-9$_1$1
29. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javascript. pp. 126–150 (06 2010). https://doi.org/10.1007/978-3-642-14107-2_7
30. Hongki Lee, Sooncheol Won, J.J.J.C.S.R.: Safe: Formal specification and implementation of a scalable analysis framework for ecmascript (2012)
31. José Fragoso Santos, Petar Maksimović, G.S.P.G.: Javert 2.0: compositional symbolic execution for javascript. Proceedings of the ACM on Programming Languages **3**, 1–31 (01 2019). https://doi.org/10.1145/3290379
32. Maffeis, S., Mitchell, J., Taly, A.: An operational semantics for javascript. pp. 307–325 (12 2008). https://doi.org/10.1007/978-3-540-89330-1_22
33. Manuel Serrano, R.B.F.: Dynamic property caches: a step towards faster javascript proxy objects (2020). https://doi.org/10.1145/3377555.3377888
34. Park, D., Stefănescu, A., Roşu, G.: Kjs: A complete formal semantics of javascript. ACM SIGPLAN Notices **50**, 346–356 (06 2015). https://doi.org/10.1145/2813885.2737991
35. Politz, J., Carroll, M., Lerner, B., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in javascript. vol. 48, pp. 1–16 (10 2012). https://doi.org/10.1145/2384577.2384579

36. Ravi Chugh, David Herman, R.J.: Dependent types for javascript (2012). https://doi.org/10.1145/2398857.2384659
37. Roberto Amadini, Alexander Jordan, G.G.F.G.P.S.H.S.P.J.S.C.Z.: Combining string abstract domains for javascript analysis: an evaluation) (2017). https://doi.org/10.1007/978-3-662-54577-5$_3$
38. ŞtefănescuŞtefan CiobâcăRadu MereutaBrandon M. MooreTraian Florin ŞerbănuţăGrigore Roşu, A.: All-path reachability logic (2019). https://doi.org/10.23638/LMCS-15(2:5)2019
39. Satish Chandra, Colin S. Gordon, J.B.J.C.S.M.S.F.T.Y.C.: Type inference for static compilation of javascript (2016). https://doi.org/10.1145/2983990.2984017
40. Simon Holm Jensen, Anders Møller, P.T.: Type analysis for javascript (2009). https://doi.org/10.1007/978-3-642-03237-0$_1$7
41. Thiemann, P.: Towards a type system for analyzing javascript programs. vol. 3444 (04 2005). https://doi.org/10.1007/978-3-540-31987-0_28
42. Trigo, M.M.M.P.: Infra-estrutura de Testes para Implementações de Referência do Standard ECMAScript. Master's thesis, Instituto Superior Técnico (On going)
43. Vineeth Kashyap, Kyle Dewey, E.A.K.J.W.K.G.J.S.B.W.B.H.: Jsai: a static analysis platform for javascript (2014). https://doi.org/10.1145/2635868.2635904