Infra-estrutura de Testes para Implementações de Referência do Standard ECMAScript

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. Keywords: ECMAScript · Specification Language · Reference Interpreters · Test262

Table of Contents

1 Introduction

2 Goals

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

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 second most active languages on GitHub[1] as well as on StackOverflow.[2]

ECMAScript standard[2] 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 [1] and the JSC [5] compilers, the most popular interpreters are Node.js [**?**] and SpiderMonkey [3]. 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,

[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

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 1 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. 1: 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 2 shows a simplified version of an array performing the pop function, where `(a)` and `(b)` are the before and after respectively. Before preforming `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 preformed `(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.

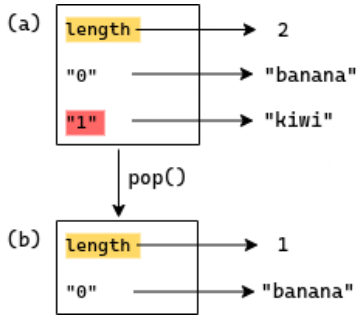

Fig. 2: Example Array.pop

Figure 3 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 2) 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. 3: ECMAScript definition of Array.pop

LengthOfArrayLike internal function Figure 4 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. 4: 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`[4], 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 39837 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 5 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. 5: 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 5 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[**?**], 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[**?**] 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 6 shows the JSON object storing the metadata generated from the Test262 test given in Figure 5. 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 5 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:

```
{
    "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. 6: Metadata generated for test esid: 15.4.5-1.js

- *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 7 illustrates, `MetaData262 v0` starts by running each test in the JS engine corresponding to version 5 of the `ECMAScript` 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.

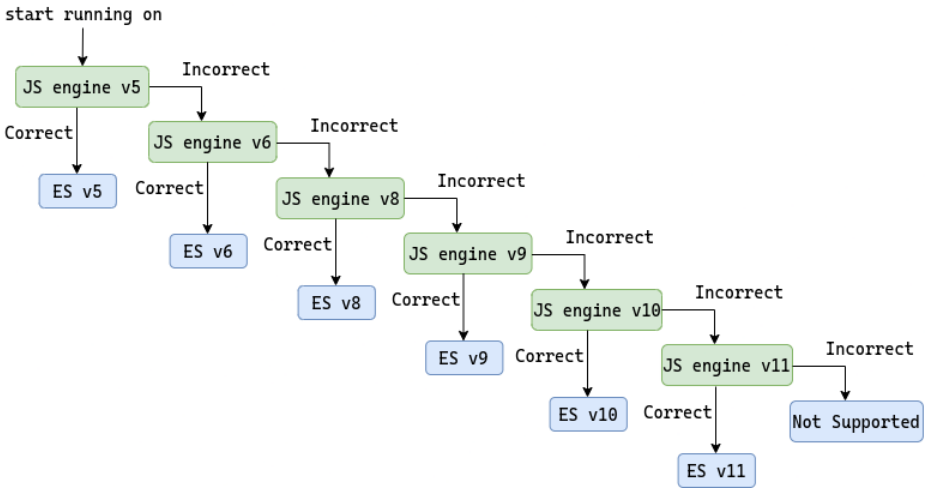

Fig. 7: Waterfall model of the dynamic approach to calculate the ECMAScript version

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

5 Design and Methodology

In this section we will discuss the improvements this thesis will make on `MetaData262 v0` as well as the `MetaData262 Visualization System` that will be created to improve ease of access to the `MetaData262` information.

5.1 Improvements to MetaData262 v0

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

– *Version Calculation* - The version calculation is flawed in both the static and dynamic approach, 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 this way adding more data points to the calculation.
– *Built-in Calculation* - The built-in calculation is also flawed in both the static and dynamic approach. The static approach has the same flaw since it also uses `Esprima` as parser. Therefore the same improvement can be made, finding a newer parser. As for the dynamic approach is is flawed due to the fact that only function calls are logged and therefore detected. If the properties of a built-in are change directly then it will not be detected. For example if a test preforms `Array.prototype.pop = 3;` then the property `prototype` of the Array built-in will be changed even though no function call was made which makes undetected by current the dynamic approach.

– *Harness Calculation* - The `harness` calculation very flawed containing only the little information available in the `Frontmatter` of a few tests. This calculation can be improved by completing `includes` metadata and adding the `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`[?] 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 less configuration 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 metadata calculated easily. The system will also update it self whenever there are changes in `Test 262`. 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 it self 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, in Figure 8 it is possible to see 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. After 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. Finally, 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. Afterwards the image shows it took 16 milliseconds to have the query results available to the user. Next the image displays that a total of 32 tests satisfy the filters selected. Finally, the image shows the path of the tests 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 8 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 9 and Figure 10 as well as some other pie charts and bar charts. In the web 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`[**?**] an UI library and `Victory`[**?**] 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.

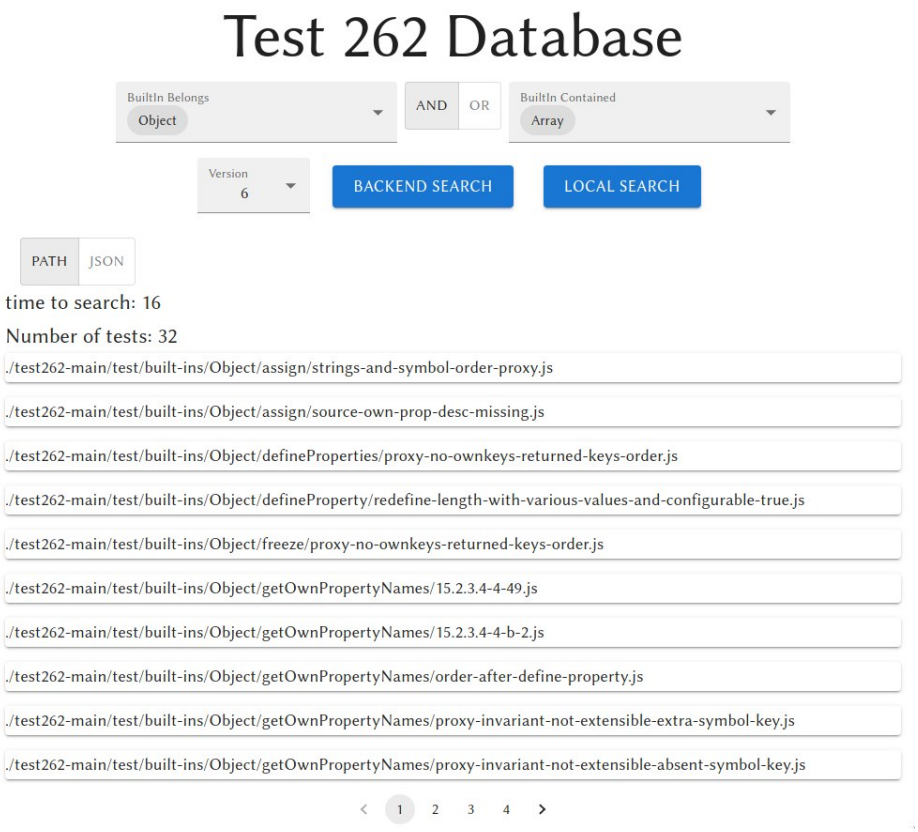

Fig. 8: Website for searching the metadata in construction

6 Evaluation and Planning

This thesis we will make improvements to the metadata calculation of `MetaData262 v0` and create a visualization system for it. The metadata improvements are divided into precision and performance. The evaluation of this thesis contributions is described in more detail bellow.

Evaluation of metadata calculation precision The precision of the metadata calculated is very hard to access since the is no ground truth available. For example the versions of the tests are not available. Thus, to evaluate the the metadata calculation we will select a small random group of tests to analyze manually and compare the the version calculated. A second evaluation will be made to the tests that have the biggest difference between the version calculated and the version that was the latest at the time the test was added to `Test262`. Firstly, we will find the date the test was added to `Test262` and associate it with the latest `ECMAScript` version at that time. After, we will select the tests with the biggest

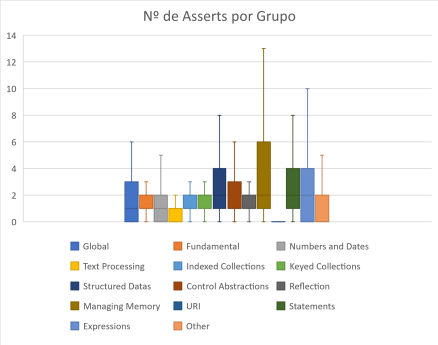

Fig. 9: M. Trigo chart number of asserts by category

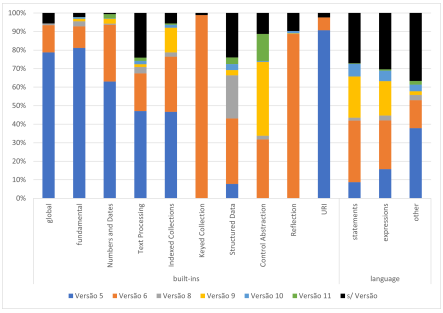

Fig. 10: M. Trigo chart versions by category

discrepancy between the versions. Finally we will preform a manual evaluation to ensure that the right version was calculated. Note that the version discrepancy is merely indicative as more the once tests were detected as missing and subsequently added to complement `Test262`.

Evaluation of metadata calculation performance To evaluate performance improvements made to `MetaData262 v0` we will compare the time needed to preform the metadata calculation. This must be ran in the same system and with the least concurrent processes running in order to make sure we have better results.

Evaluation of the visualization system 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 and reliability of the system. 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 systems can support.

6.1 Planning

The Plan for this thesis is based on the tasks that need to be preformed 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. The planning for these tasks is represented in the Gantt diagram in Figure 11 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.

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 more JS engines and a JS parser that supports

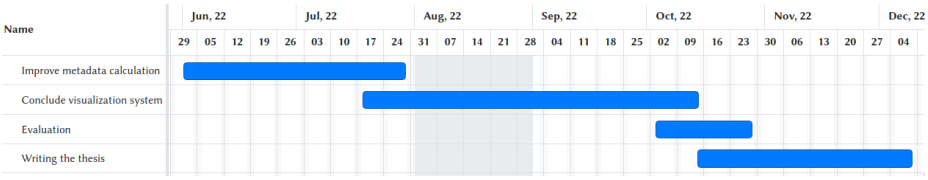

Fig. 11: Gantt diagram with the thesis plan

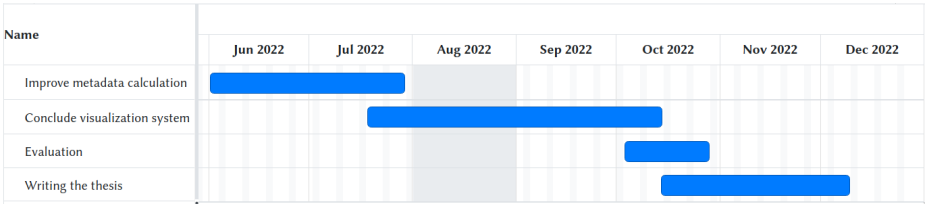

Fig. 12: Gantt diagram with the thesis plan

more recent versions of the standard. As for the performance, we will parallelize the metadata calculation.

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 through preparing it to handle wrong queries.

Evaluation This task consists of evaluating the results of the metadata generated. The most prone to error metadata will be checked by hand in order to guarantee the correctness of the results presented.

Writing the thesis This last task will consist in a report of the execution of the previous tasks. In the report we will present the results obtained as well as the conclusions reached after finishing this project.

7 Conclusion

References

1. Ambiente de programação multi-tier para o javascript, https://github.com/manuel-serrano/hop, acedido a 2020-12-21
2. especificação da linguagem ecmascript® , 6.0 edition / june 2015, http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf, acedido a 2020-12-21
3. Spidermonkey is mozilla's javascript engine written in c and c++, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, acedido a 2020-12-21
4. Test262 - official ecmascript conformance test suite, https://github.com/tc39/test262/, acedido a 2020-06-07

5. Um motor javascript que corre em cima do motor javascript do browser, https://github.com/mbbill/JSC.js, acedido a 2020-12-21