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 it's multiple compilers and interpreters implementions are coherent. Some of the `JavaScript` compilers are the [1] and the JSC [5] compilers, the most popular interpreters are nodejs [?] and spidermonkey [3]. These are only four implementations amongst many others, which come along with the many use cases that `JavaScript` has. `JavaScript` is used 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, there 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. 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).

[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

- 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, amongst 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 internat 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` being preformed `(b)`, the last element is of the array is removed (highlighted in red at `(a)`). The `length` property highlighted in green is also updated since the size of the array changes to one.

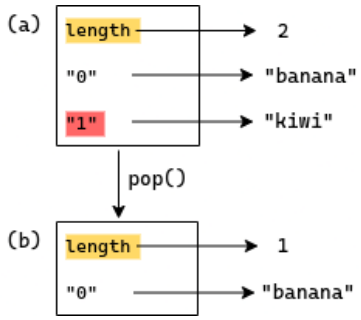

Fig. 2. Example Array.pop

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 $\mathbb{F}$(*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 $\mathbb{R}$(? 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 (large quantity of tests and features).Test262 complexity grows with changes to the standard since in most cases retrocompatibility is maintained except for a few select cases.

Due to ECMAScript standard being so extensive most implementations are only partial (namely implementations in academic context). Since the implementations are only partial, only tests for features that are implemented are relevant. Since selecting the relevant tests is not a trivial matter, each development team selects the appropiate tests. Each development selecting the tests raises the problem that there is not standard and precise way of picking the all the right tests.

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 12). The copyright section has information about the owner and licence of the test. The Frontmatter section has the 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]', '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 some type of metadata about the test. Bellow is the list of possible keywords and its 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 throw (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;
- `author` - contains the identification of the author of the test;
- `flags` - contains a list of booleans for each 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`, only for asyncronus tests; 6 `generated`, flag that identifies files created by a test; 7 `CanBlockIsFalse` and 8 `CanBlockIsTrue`, indicatethe value of the property `CanBlock` ; 9 `non-deterministic`, idicates that the semantics used on 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` - containds 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 bellongs to; these keywords have been deprecated and substituted by esid;

The example in 5 has 2 keywords, description and the deprecated es5id. Besides the obvious upgrade from `es5id` to `esid` it would be usefull to have `includes` with the harness files needed to execute the test. The harness information is very usefull since it makes easly available what part of the harness is 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.

In order to have a more complete `Frontmatter` we suggest adding the following information:

- `static construct` - list of static constructions used in the test
- `version` - the ECMAScript version of the standard after which the test belongs to
- `built-ins` - list of all the built-ins used in the test

This metadata provides helpfull information to solve the problem mentioned before of selecting the relevant tests for partial implementations.

3.3 An Infrastructure for testing reference implementations of the ECMAScript standard

4 Related Work

5 Design and Methodology

6 Evaluation and Planning

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