A Trusted Mechanised Specification of JavaScript: One Year On

Philippa Gardner, Gareth Smith, Conrad Watt, and Thomas Wood(✉)

Imperial College London, London, UK
{pg,gds,cw2312,tw1509}@ic.ac.uk
http://jscert.org

Abstract. The JSCert project provides a Coq mechanised specification of the core JavaScript language. A key part of the project was to develop a methodology for establishing trust, by designing JSCert in such a way as to provide a strong connection with the JavaScript standard, and by developing JSRef, a reference interpreter which was proved correct with respect to JSCert and tested using the standard Test262 test suite. In this paper, we assess the previous state of the project at POPL'14 and the current state of the project at CAV'15. We evaluate the work of POPL'14, providing an analysis of the methodology as a whole and a more detailed analysis of the tests. We also describe recent work on extending JSRef to include Google's V8 Array library, enabling us to cover more of the language and to pass more tests.

1 Introduction

JavaScript is the most widely used web language for client-side applications. However, JavaScript is complex and the associated ECMAScript standard (edition 5.1 in this paper, abbreviated ES5) is, by necessity, large and full of corner cases. In POPL'14, Gardner, Smith and others developed a Coq mechanised specification of the core JavaScript language, called JSCert ([1] and see acknowledgements). This work provides a foundation for future research projects based on, for example, program logics, type systems, sub-language analyses and abstract interpretation. It demonstrates that modern techniques of mechanised specification can handle the complexity of JavaScript.

An important part of the JSCert project was to develop a methodology for establishing trust: JSCert was designed so that each line of the core language of ES5 corresponds to one or two rules in JSCert; an executable reference interpreter, JSRef, was developed in parallel and proved to be correct with respect to JSCert; and JSRef was tested using Test262, the test suite that accompanies the ES5 standard. The methodology ensured that JSCert is a comparatively accurate formulation of the English standard, which will only improve with time.

In this paper, we describe the state of JSCert at POPL'14 and the current state of JSCert at CAV'15. With JSCert at POPL'14, we evaluate the methodology as a whole, report on the test results presented at the time, and assess our interpretation of the tests. We have found no errors in the Coq proof showing

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 3–10, 2015.
DOI: 10.1007/978-3-319-21690-4_1

that JSRef is correct with respect to JSCert. We have identified a small number of cases where we have misinterpreted ES5, with these misinterpretations occurring consistently in both JSCert and JSRef. These misinterpretations have now been fixed; the close connection between ES5 and JSCert means that local misinterpretations of ES5 results in local fixes to JSCert and JSRef. We have found errors in the analysis of the tests, in particular by misattributing some test failures to the external parser instead of our parser interface code. Since POPL, we have greatly improved the test analysis and report on our results here.

For CAV'15, we give a snapshot of the current state of the JSCert project. The primary criticism of the POPL'14 work was that it only dealt with the core language, not with the associated libraries. In principle, we do not envisage difficulty in extending JSCert to the libraries[1], although covering all such libraries would be a mammoth task. Instead, we explore a different approach, to integrate an existing industrial-strength library implementation with JSRef. We focus on the Array library for illustration. We implement the Array library's low-level functionality using Coq and its high-level functionality using Google's V8 Array library implementation in core JavaScript. The V8 Array library is a good choice for us, as it provides a clear separation between the low-level and high-level functionality.

We can now run more code and pass more tests. We obtain trust in our extended JSRef in as far as we can trust the Google V8 Array library, trust our Coq implementation of the low-level functions, and trust the tests to identify errors in the industry code and our Coq code. However, this does not compete with the strong trust of the original JSCert project; for that, we need to extend JSCert to also specify the Array library.

2 JSCert at POPL'14

JSCert is an inductively-defined Coq semantics of the core part of ES5, suitable for carrying out formal proofs of, for example, safety properties of ES5 and security properties of sub-languages. It identifies the core language of ES5, comprising chapters 8–14 of ES5 and a small amount of Chap. 15. Chapters 1–7 are not directly relevant to JSCert[2]. Chapters 8–14 describe the bulk of the core language. The `for-in` command has not been specified, since it is notoriously difficult to understand and requires a global complication of the specification[3].

[1] Maksimović and Schmitt have begun to specify the core Array specification in JSCert and JSRef.

[2] Chapters 1–7 provide hints on how to read the later chapters, information about how the standard relates to the rest of the world and information that is only useful for parsing.

[3] During discussion on the `es-discuss` mailing list, even members of the ECMAScript committee had differing opinions of what the standard meant. The committee came to a consensus and we know how to specify the for-in command in JSCert. However, this specification would involve a global change, with the enumerable fields having to be explicitly recorded throughout the specification. The choice was to omit this

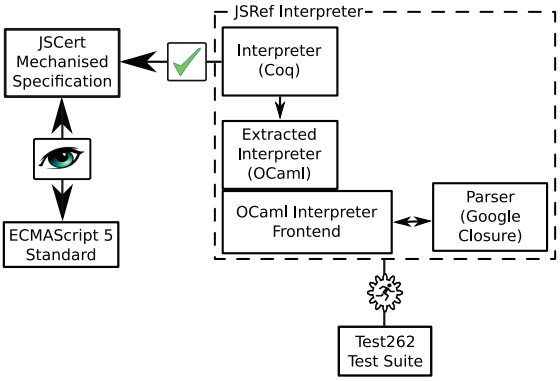

Fig. 1. The JSCert project at POPL'14.

The Array literal syntax has also not been specified, since the project did not specify the Array library.

Chapter 15 specifies objects and functions that should be present in the heap when the JavaScript interpreter is started. These functions provide both 'core' language functionality which must be directly implemented in the interpreter (such as special object constructors; `eval`; or control of property descriptors) and library or helper functionality which (such as URI decoding or array sorting). Unfortunately the split is not always clear, as all of these functions are defined in the ES5 specification in terms of the language's internal behaviour so it is not trivial to determine whether a function only makes use of state accessible to regular JavaScript programs. JSCert has specifications of the 'core' functions of Chap. 15, excluding the Array library.

JSCert is written using the pretty-big-step semantics of Charguéraud [2]. The original operational semantics of JavaScript, created by Maffeis, Mitchell and Taly [3] for ECMAScript 3, was written using a small-step semantics. By contrast, the prose of the standard has a big-step flavour. The aim was to design JSCert to be as close to the standard as possible. However, a traditional big-step operational semantics would lead to many duplicate rules since the JavaScript control flow is quite complex. The pretty-big-step semantics enables JSCert to be closer to the English prose. It was originally developed for a small ML-like language. The JSCert specification demonstrates that it scales to ES5, a real-world language which was not designed with formal methods in mind.

The main challenge was (and still is) to convince people (including ourselves) that JSCert can be trusted as an accurate formulation of the ES5 English standard. The design of the project is illustrated in Fig. 1. JSCert is 'eyeball close' to ES5, in the sense that we can place the English prose and the formal rules side-by-side and compare the two. This closeness is possible due to the

change, rather than complicate the specification, especially as the for-in command has essentially been replaced by the better behaved for-of command in the next standard.

pretty-big-step semantics. In most cases, each line of English corresponds to one or two Coq rules. The reason for the two rules is that, for simplicity, ES5 leaves much behaviour (such as state change and exception handling) implicit, whereas JSCert gives the behaviour explicitly to aid comparison and help with proofs. In some cases, the connection is not quite line by line. A typical example involves the while specification, where two lines of ES5 English specification correspond to two Coq rules: in ES5, the Boolean expression is evaluated to a reference then converted to a Boolean value; in Coq, this is done in one step.

JSCert is accompanied by the JSRef reference interpreter which comprises several parts. It comprises an interpreter written in Coq which is automatically extracted to an interpreter written in OCaml. This interpreter is then linked to a front-end which provides interfaces to the end-user and to a third-party JavaScript parser (for POPL'14, the Google Closure parser). The interpreter in Coq has been proved correct with respect to JSCert for Chaps. 8–14. More precisely, if the execution of a JavaScript program in JSRef returns a result, then there exists a reduction derivation in JSCert relating this program to this result. The creation of the interpreter in OCaml uses automatic Coq extraction techniques which are standard and well-used. Our trust that the extracted interpreter is an accurate reference interpreter for JSCert is based on the correctness proof for the interpreter in Coq, our trust in the Coq extraction process, the minimal amounts of unverified front-end code, and the testing using the ES5 conformance test suite, Test262.

The test results focused on Chaps. 8–14. Those reported in the POPL paper and talk are given in Table 1[4]. The paper stated that 'JSRef successfully executes all the tests *we expect to pass* given our coverage of ES5'. The original analysis reported that the failed and aborted tests were due to: for-in not implemented; Chap. 15 library functionality not implemented; and failures due to a non-conforming parser. This analysis was improved by the time of the POPL talk, hence the two rows in the table: the `for` command and associated tests (28 tests) had been omitted due to confusion with the `for-in` command; and some further tests (27 tests)[5] had been omitted.

Evaluation. An original aim of the project was to assess how much of ES5 it was possible to specify in Coq. JSCert covers the core language of ES5 (Chaps. 8–14 plus the some of Chap. 15), except for the for-in command and the Array literal syntax, as discussed. The fact that the specification was able to cope with all the corner cases was a surprise and a considerable achievement. The 'eyeball closeness' of JSCert with ES5 has been a success. In our experience, it is possible for a Coq expert reading JSCert and a JavaScript expert reading the ES5 standard to have a detailed discussion about the different formulations.

[4] We have separated the fails and the aborts. Most aborts are due to tests calling functions 'Not Yet Implemented', although a few aborts are real parser failures. Some fails are also due to tests calling functions 'Not Yet Implemented'. The other fails are more significant.

[5] Those associated with the Argument object and those calling the `hasOwnProperty` method.

Recall that JSRef comprises an interpreter in Coq which is extracted to an interpreter in OCaml. The correctness proof between JSCert and the interpreter in Coq has also stood the test of time. The proof was given for Chaps. 8–14. This gave a precise, clear description of what had been proved. In future, we would like extend the proof to the core language specified by JSCert. We have not found any mistakes in this proof. We have found some misinterpretations of the ES5 standard: for example, strict mode delete was not throwing an exception for unresolvable references. These misinterpretations are present in both JSCert and JSRef. JSCert and JSRef were developed separately, by different teams, but there was much interaction between the teams. When the ES5 standard was unclear, they reached consensus, both between themselves and with the help of `es-discuss`. So far, we have discovered that just a small amount of the ES5 core language was misinterpreted and this has been fixed.

The test analysis needs improvement. Many failed and aborted tests were due to for-in and Chap. 15 library functionality not being implemented, and this was correct. However, the failures of many strict-mode tests (at most 237 tests) were attributed solely to the parser, and this was incorrect: some failures were, indeed, due to the parser; other failures were due to the misinterpretation of ES5: for example, strict mode delete previously discussed; and most failures were due to mistakes in our parser interface code. These mistakes were not picked up by the tests because the test filtering at the time was ad-hoc and over-zealous for the strict-mode tests. The test filtering is now much better, the test failures and aborts are properly attributed, and the mistakes in our parser interface code are fixed.

Table 1. JSRef test results as at POPL'14 and CAV'15. The Array results for POPL'14 were not previously reported and are shown for comparison. Two rows of results are shown for CAV'15, the first without the Google V8 Array library loaded and the second with it loaded.

	Chs. 8–14			Ch. 15.4 – Array		
	Pass	Fail	Abort	Pass	Fail	Abort
POPL'14 paper results	1796	404	582	(139)	(873)	(1307)
POPL'14	1851	392	539	(149)	(864)	(1306)
CAV'15	2437	129	216	180	1204	935
CAV'15 (+V8 Array)	2440	126	216	1309	59	951

3 JSCert at CAV'15

We report on the current state of the JSCert project at CAV'15. JSCert remains largely the same. We have fixed the known inconsistencies with the ES5 standard as noted in the previous section. The JSRef interpreter has changed. From

POPL'14, many of the failed and aborted tests seemed to be due to library functions not yet implemented. In particular, there were many tests for the core language that called the Array library. We therefore extend the JSCert project with this library. One approach is to extend JSCert with a Coq specification of the Array library; Maksimović and Schmitt are beginning to do this. Another approach is to extend JSRef with an existing industrial-strength library implementation; we study this approach here.

Most of the Array functions (and, indeed, most of the Chap. 15 functions in general) do not directly access the language's internal state. They can, therefore, be implemented in the core JavaScript language and then loaded, parsed and interpreted to yield an initial heap state which declares these functions. The major JavaScript interpreters are using or moving towards this approach, which we explore here for the JSCert project. Rather than implementing this library ourselves, we use portions of Google's V8 Array library implementation as it has a clear separation of core functionality, which requires access to the language's internal state, and the higher-level functionality, which is implemented in the core JavaScript language.

The new structure of the JSCert project is given in Fig. 2. JSCert remains largely the same. The parser has been changed from Google's Closure to jQuery's Esprima for improved correctness, execution speed and web compliance. This change involved adding support for translation from the de facto SpiderMonkey AST to our internal AST representation, enabling us to use a wide range of third-party parsers for the front-end of JSRef. The JSRef interpreter has been extended to include the V8 Array library. To support the execution of this library, we extended the interpreter written in Coq with a number of low-level functions: some of these functions are defined in other sections of Chap. 15, such as those associated with `Object` or `Function`; and some provide access to a small amount of usually restricted internal state used, for example, to modify the prototype of an object or set the normally immutable `length` field of a function. In addition, V8 has some minor helper functions implemented in C++ to improve performance. We implement these helper functions in core JavaScript to minimise the size of the native/interpreted interface.

The test results are given in Table 1. We provide a more careful analysis of the tests for Chaps. 8–14. We also execute and analyse the tests for the Chap. 15.4. For the Chap. 8–14 tests, we believe that all the failed and aborted tests are doing so for valid reasons. These are mostly due to parts of the language that are not yet implemented: namely, the for-in statement (93 tests failing); array literal syntax (26 tests); 78 tests failing for missing Chap. 15 functionality; and 135 tests failing for other non-implemented features. In addition: 7 tests are failing because they use strictly invalid, but commonly used, syntax; 1 test is failing due to a parser bug (reported to the vendor); and 2 tests are failing due to the method of executing multiple programs in sequence by the unverified interpreter front-end.

We have run the tests for the Chap. 15.4, but currently have only a partial analysis of the tests. Since we use Google's V8 Array library, we can probably trust the implementation of the high-level functions and do not expect many

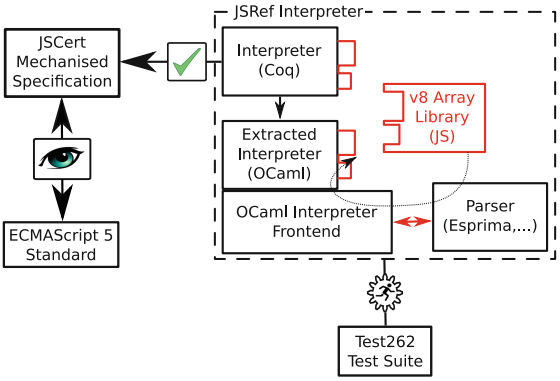

Fig. 2. The JSCert Project at CAV'15.

test failures associated with them. We do expect test failures in our Coq and JavaScript code which replaces the V8 C++ code, partially because it is code we have written and partially because the interface between the JavaScript and C++ code is not documented. The only way to establish trust in our code is through testing.

At the moment, 904 tests fail or abort due to Array literal syntax from Chaps. 8–14 not being implemented. This is potentially masking many bugs. For POPL'14, the Array literal syntax was not implemented because we were not calling the library. Now, the library is being called and the Array literal syntax needs to be specified in JSCert, interpreted in JSRef and the correctness proof extended. This is currently being done by Maksimović and Schmitt as part of their specification of the Array library. Many of the other tests fail due to parts of the language that are not yet implemented: namely, 30 tests because of the missing for-in statement; 53 because of missing Chap. 15 functionality; 19 tests because of other non-implemented features; and 1 test because of the use of invalid syntax. Additionally, 3 tests are failing due to an error in the ES5 specification[6]. This error was introduced as a typographic error between versions 3 and 5 of the ECMAScript specification. Test262 captures the intended semantics as per ES3, JSCert captured the incorrect semantics of ES5. The resulting discrepancy revealed itself as a set of test failures discovered during this test evaluation.

Evaluation. This paper assesses the current state of the JSCert project, reports on the improved analysis of the tests for chapters 8–14, and describes the extension of JSRef with Google's V8 Array library implementation.

JSCert provides a mechanised specification of the core JavaScript language, as described by the ES5 standard. It comprises Chaps. 8–14 and parts of Chap. 15, omitting the for-in command and the Array literal syntax. Following our work in POPL'14, the ES5 standard has also been specified in the K framework [4].

[6] https://bugs.ecmascript.org/show_bug.cgi?id=162.

In this work, the definition of the core language is that required to pass the core tests. In fact, it is not completely clear what the core language should be, since it is not precisely described by the ES5 standard. We should at some point compare the choices in the JSCert and the K specifications.

The proof that JSRef is correct with respect to JSCert has only been done for Chaps. 8–14, not the core language. The choice to focus on Chaps. 8–14 was made to present a clear boundary of what had been proved. However, in future, we would like to extend the proof to the core language. For now, the analysis of the tests focused on Chaps. 8–14 and the Chap. 15.4 Array library. The infrastructure for analysing the tests has been vastly improved: the test run takes considerably less time; the filtering is more accurate; and the test analysis can be more trusted. We believe the tests for Chaps. 8–14 are well done. The tests for Chap. 15.4 Array library are on-going. Many tests involve the Array literal syntax from chapters 8–14 which has not being implemented. These might be hiding many bugs, and we will investigate this once Maksimović and Schmitt have extended JSCert, JSRef and the proof to include the Array literal syntax. Otherwise, the other failed tests are understood.

We believe our experiment to extend JSRef with Google's V8 Array library has been a success. A next step is to extend this approach to the String, Boolean and Number libraries. Our overall aim is to pass as many tests as we can.

Acknowledgements. Two of the authors of this paper, Gardner and Smith, were part of the original team working on JSCert. We would like to thank the other co-authors for continuing invaluable discussions about this project: Martin Bodin, Arthur Charguéraud and Alan Schmitt from Inria; and Daniel Filaretti, Sergio Maffeis and Daiva Naudžiūnienė from Imperial. We also would like to thank Petar Maksimović and Alan Schmitt for interesting discussions and interaction about the Array library. They are beginning to specify the core Array library in Coq.

Gardner and Smith are supported by EPSRC Grant EP/K032089/1. Watt was supported by a GCHQ Undergraduate Internship Project award. Wood is supported by an EPSRC DTA award.

References

1. Bodin, M., Charguéraud, A., Filiaretti, D., Gardner, P., Maffeis, S., Naudžiūnienė, D., Schmitt, A., Smith, G.: A trusted mechanised javascript specification. In: Proceedings of the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 2014, ACM (2014)
2. Charguéraud, A.: Pretty-big-step semantics. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 41–60. Springer, Heidelberg (2013)
3. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for javascript. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 307–325. Springer, Heidelberg (2008)
4. Park, D., Ştefănescu, A., Roşu, G.: KJS: A complete formal semantics of javascript. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2015, pp. 428–438. ACM (2015)