A Trusted Mechanised JavaScript Specification

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, Gareth Smith

A Trusted Mechanised JavaScript Specification

Martin Bodin
Inria & ENS Lyon
martin.bodin@inria.fr

Arthur Charguéraud
Inria & LRI, Université Paris Sud, CNRS
arthur.chargueraud@inria.fr

Daniele Filaretti
Imperial College London
d.filaretti11@imperial.ac.uk

Philippa Gardner
Imperial College London
philippa.gardner@imperial.ac.uk

Sergio Maffeis
Imperial College London
sergio.maffeis@imperial.ac.uk

Daiva Naudžiūnienė
Imperial College London
d.naudziuniene11@imperial.ac.uk

Alan Schmitt
Inria
alan.schmitt@inria.fr

Gareth Smith
Imperial College London
gareth.smith05@imperial.ac.uk

Abstract

JavaScript is the most widely used web language for client-side applications. Whilst the development of JavaScript was initially just led by implementation, there is now increasing momentum behind the ECMA standardisation process. The time is ripe for a formal, mechanised specification of JavaScript, to clarify ambiguities in the ECMA standards, to serve as a trusted reference for high-level language compilation and JavaScript implementations, and to provide a platform for high-assurance proofs of language properties.

We present JSCert, a formalisation of the current ECMA standard in the Coq proof assistant, and JSRef, a reference interpreter for JavaScript extracted from Coq to OCaml. We give a Coq proof that JSRef is correct with respect to JSCert and assess JSRef using test262, the ECMA conformance test suite. Our methodology ensures that JSCert is a comparatively accurate formulation of the English standard, which will only improve as time goes on. We have demonstrated that modern techniques of mechanised specification can handle the complexity of JavaScript.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords JavaScript, mechanised semantics, Coq

1. Introduction

JavaScript is by far the most widely used web language for client-side applications. Initially, JavaScript development was entirely led by implementation, with Netscape releasing the first JavaScript implementation in 1996 and Microsoft responding with their own version in the same year. Netscape quickly realised that standardisation was crucial. Client code that works on some of the main browsers, and not others, is not useful. The first official standard appeared in 1997. Now we have ECMAScript 3 (ES3, 1999) and ECMAScript 5 (ES5, 2009), supported by all browsers. There is increasing momentum behind the ECMA standardisation process, with plans for ES6 and 7 well under way.

JavaScript is the only language supported natively by all major web browsers. Programs written for the browser are either written directly in JavaScript, or in other languages which compile to JavaScript. This fundamental role seems unlikely to change. However, JavaScript is very complex. The ECMAScript standards, by necessity, are large and full of corner cases. Despite the best efforts of their editors, these documents are sometimes unclear and, in some isolated cases, even inconsistent. We believe the time is ripe for a formal, mechanised specification of JavaScript, to clarify ambiguities in the ECMA standards, to serve as a trusted reference for high-level language compilation and JavaScript implementations, and to provide a platform for high-assurance proofs of language properties.

We introduce JSCert, a mechanised specification of ES5 written in the interactive proof assistant Coq. We also introduce JSRef, an executable reference interpreter extracted from Coq to OCaml. We give a Coq proof that JSRef is correct with respect to JSCert. The correctness proof ensures that we can have full confidence that a JSRef program has the behaviour specified by JSCert. All our Coq code is available at `http://www.jscert.org`. We believe that both JSCert and JSRef are necessary: JSCert, unlike JSRef, is well-suited for developing inductive proofs about the semantics of JavaScript; JSRef, unlike JSCert, can be used to run JavaScript programs.

Our challenge is to convince ourselves and others that JSCert is indeed an accurate formulation of the ES5 English specification. We designed JSCert to follow the structure of the ES5 English standard as much as possible. Whenever we found parts of ES5 English prose to be ambiguous, we checked the browser implementations and were active on discussion groups such as `es-discuss`. We also ran JSRef on the official ECMA test suite, test262. Since JSRef is correct with respect to JSCert, failed tests meant discrepancies between ES5, JSCert, and the tests. Using this methodology, we were able to correct several bugs in JSCert and JSRef. We

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535876

also discovered, and reported [58, 60], a number of bugs in all the browser implementations, the ECMA standards, and test262. We have demonstrated that modern techniques of mechanised specification can handle the complexity of JavaScript.

JSCert. We introduce JSCert, a mechanised Coq specification of ES5. In 2008, Maffeis, with Mitchell and Taly from Stanford, developed a hand-written, small-step operational semantics for formally describing the full behaviour of JavaScript [36, 37], faithfully following ES3 except when the English standard was incorrect or ambiguous. Apart from moving to ES5, we also differ from [36] in that we use a big-step semantics to be closer to the style of the English prose. However, the traditional big-step approach would lead to many duplicated rules, since the JavaScript control flow is quite complex. We use a pretty-big-step semantics, an approach recently developed by Charguéraudfor a simple ML dialect [11]. Our work here demonstrates that the technique scales to a real-world standard, and yields a close connection between ES5 and JSCert.

Note that we are not trying to usurp ES5, which has an ease of readability that would be difficult to match with a mechanised specification. We are aiming for an accurate, mechanised specification of ES5 which can be used for clarification when the ES5 prose is unclear. By gathering a team of Coq and JavaScript experts, we have been able to handle the size and the complexity of the JavaScript semantics.

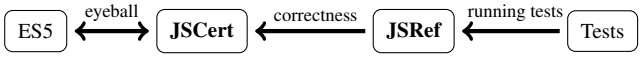
As with most large specification projects, a significant question is when to stop. We have formally specified the core language, where we believe most of the interest for our project lies. We have not specified the for-in command, because in trying to do so we discovered that ES5 and ES6 are broken in this respect.[1] We have not specified the parsing of JavaScript source code, which would be a challenging, orthogonal task. Also, our formalisation of native libraries is partial. We have formalised nearly all of the library functions that expose internal features of JavaScript, with the exception of functions involving arrays which is future work. In particular, we have formalised most of the functions from libraries `Object`, `Function`, `Boolean`, `Number`, and `Errors`. We are not planning to formalise the other native libraries. They contain hundreds of library functions that do not interact with internal JavaScript features, in the sense that they could be implemented as plain JavaScript code.

JSRef. We introduce JSRef, an executable reference interpreter, which we have proved correct with respect to JSCert and tested using the ECMAScript conformance test suite, test262. By design, JSCert is not directly executable. It is presented as an inductive definition, essential for developing safety proofs, and it matches the looseness of ES5's "implementation-dependent" features. We therefore developed a computable Coq specification and, from this, automatically extracted the corresponding OCaml code, obtaining the executable reference interpreter JSRef. Our computable Coq specification follows the ES5 pseudo-code as much as possible. Where the implementation details were left unspecified, we made arbitrary but natural choices.

In fact, we developed JSCert and JSRef in parallel. This approach was invaluable when understanding an unclear part of ES5: we sometimes needed the specifier's broad intuition; sometimes the implementer's pragmatic intuition about what was really meant. Recall that JSCert specifies the core language of ES5, but does not specify the parser nor many of the native libraries. The same restrictions apply for JSRef. In particular, we rely on an off-the-shelf JavaScript parser, taken from the Google Closure Compiler [25], for parsing the initial source code and for implementing `eval`.

[1] We have filed a bug report (n. 1444 of [58]) for ES6.

Trust. Our methodology for obtaining trust in our mechanised semantics can be summarised as follows:



For JSCert, trust arises from the 'eyeball closeness' of JSCert to ES5. We can place the English prose and the formal rules side-by-side, and compare the two: one line of ES5 pseudo-code corresponds to one or two rules in JSCert. If commands need changing, as inevitably they will for future standards, it is relatively straightforward to see which part of JSCert needs changing.

For JSRef, we have a machine-checked correctness proof that it satisfies JSCert. More precisely, we prove in Coq that, 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. Because of the looseness of ES5, our interpreter cannot be proved complete. Nevertheless, we believe that, on the deterministic subset of ES5, our interpreter is complete. We trust the extraction mechanism of Coq, used to obtain the OCaml code of JSRef, since it is standard and widely-used (e.g., in [8]).

We also test JSRef using test262 [18]. JSRef successfully executes all the tests that we expect to pass given our coverage of JSRef. There are 2782 tests associated with the core language (chapters 8–14): we pass 1796 tests; the others fail due to for-in, calls to libraries we have not specified, or the parser. We also use the OCaml bisect tool [14] to investigate the coverage of these tests. Our emphasis on testing provides a different level of trust from that usually found in Coq development (see related work), which is perhaps more accessible to JavaScript implementers and programmers. It also provides a first step in the analysis of what it means to trust test262.

As we were developing the proof of correctness and running the tests, we inevitably found bugs in JSCert and JSRef. It was necessary to go around the loop many times, fixing inaccuracies, continuing with the correctness proof and testing, and so on. Our correctness proof guarantees that JSRef is an accurate reference interpreter for JSCert (up to our trust of the Coq extraction process). However, despite our principled development of JSCert and JSRef, we cannot yet guarantee that JSCert is bug free in the sense that there may well be mismatches with ES5. A few mismatches will be due to the ambiguity of ES5. The ECMA authors have been very responsive to our queries on es-discuss and it has always been possible to reach consensus. Other mismatches, however, will be due to inaccuracies in JSCert. We do believe that, over time, with more proof reading and testing by ourselves, and with help from others, we will be able to eliminate all the bugs.

Applications. There are many potential applications of JSCert and JSRef. For example, we can investigate properties of fragments of JavaScript used for secure sandboxing by companies such as Yahoo!, Google, and Facebook. Several high-level languages are compiled to JavaScript. JSCert provides a formal target to verify the correctness of the compilation. A natural first language to explore in this respect is Microsoft's F* compilation to JavaScript (and back for full abstraction, see related work). JavaScript implementers can compare their implementation with JSRef. For example, a direct extension of our work would be to characterise the differences between Firefox and JSRef by running the Firefox test suite on JSRef. A dream is that our mechanised specification might even be used as part of the creation of future ECMA standards: ES6 is already taking shape, but ES7 is only just being planned. This paper demonstrates that it is feasible to provide a Coq specification of ES5 together with a reference interpreter passing the ES5 tests. We are impacting on ES6 with bug reports. It may not be beyond

reach to provide a mechanised specification for ES7 as part of the official standard.

2. Related work

A real-world language specification can come in many forms: an implementation of the compiler/VM as specification (e.g., PHP); an English definition with varying degrees of rigour (e.g., the C standard [10] and ES5 are fairly precise and complete); a formal mathematical specification (e.g., Standard ML [41]); and mechanised specification. Our work on JavaScript is part of the established tradition to mechanise existing, real-world programming language specifications. In this section, we survey related work on the large-scale mechanisations of programming languages in general, and on various formalisations of JavaScript in particular.

2.1 Large-scale mechanised specifications

One of the most prominent, fully formalised presentations of a programming language is Standard ML by Milner, Tofte, Harper, and Macqueen [41]. A mechanised specification was given by Lee, Crary, and Harper [34] in the Twelf theorem prover [48]. Unlike ML, many real-world languages are designed without formalism in mind. Such languages provide a considerable challenge to mechanisation.

There has been a wide body of work on mechanised language specifications in HOL. For example, Norrish [44] specifies a small-step operational semantics of C in HOL [43], and proves substantial meta-properties of the semantics. Norrish's formalism has not been tested for conformance with implementations. Another example is Sewell *et al.*'s work [7] on formalising transmission control protocols (TCP) in Isabelle/HOL [43]. They created a post-hoc specification of TCP from several prominent implementations, an enterprise not so far off the creation of the original ECMA standard for JavaScript. Although not focusing on language specification, their work is notable in the context of our work as it validates the specification against several thousand test traces captured from implementations, using a special-purpose symbolic model checker programmed above HOL.

In the CompCert project [15], Blazy and Leroy [8] built a verified optimising compiler for CLight, a significant fragment of C, with a Coq proof that the generated compiled code behaves exactly as prescribed by the semantics of the source program. The CompCert project initiated major technological breakthroughs in Coq mechanisation, some of which we substantially use in this project. The aims of the projects are different. CLight was not intended to capture precisely the C specification. Also, CLight is not directly executable, although it would be possible to obtain an interpreter without too much additional effort, by leveraging the Coq code extraction mechanism. Several substantial projects build on CompCert: Appel's verified software tool chain [3] combining program verification with verified compilation; Shao's project to certify an OS kernel; Zhao *et al.*'s verified LLVM which extracts an interpreter from Coq code that is tested using the LLVM regression suite (134 out of 145 runnable tests); and Sewell's CompCertTSO [61], verifying compilation the x86 weak memory model [1].

The use of proof assistants such as HOL and Coq requires quite a substantial learning curve. Researchers are beginning to explore how to make mechanised specification easier. The $\mathbb{K}$ framework is designed specifically for writing and analysing language definitions using a rewrite logic [19]. In particular, Ellison and Rosu [19] have defined an executable formal semantics of C in $\mathbb{K}$ [53]. Their formalisation has been extensively tested against the GCC torture suite [22]. Besides being executable, their semantics also comes with an explicit-state model checker. Their aim is to define an accurate reference interpreter, justified by testing, rather than provide a full semantics of the C standard. Their work is analogous to JSRef. In addition, the Ott tool [55] provides a lightweight environment using ASCII notation for writing definitions of programming languages and calculi, which automatically translate to HOL, Isabelle, and Coq. Owens *et al.* have developed a mechanised semantics of OCaml Light using Ott.

For space reasons, we cannot detail all the interesting examples of mechanised specifications of programming languages. Inevitably, there is a wide body of work on mechanised specifications of Java and C#: e.g., Syme's HOL semantics [56] of Drossopoulou and Eisenbach's formal Java semantics [17]; the executable formalisation of the C# standard by Börger *et al.* [9] using Abstract State Machines [29]; and the executable formal semantics of Java 1.4 in rewrite logic by Farzan *et al.* [20]. We should also mention the formal semantics of Batty *et al.* on C++ concurrency [4, 5], which is currently having real impact on the C11 standard [10]. The work is currently not mechanised but, considering the research group, it will surely happen soon.

Our mechanised specification of ES5 shares many of the difficult challenges faced by the work described above, and involves many new ones due to the complex dynamic nature of JavaScript. They are detailed in § 4. We now provide a more detailed survey on formalising and mechanising the JavaScript semantics.

2.2 Formal JavaScript specifications

In 2005, Anderson *et al.* [2] and Thiemann [59] were the first to propose formal type systems for subsets of JavaScript. To prove type-soundness, they formalised idealised cores of the language that abstracted away features not crucial for the type analysis at hand, focusing instead on the challenges addressed by their analyses. Since then, researchers have studied various typed JavaScript subsets and static analyses, including [12, 13, 26, 27, 30, 32, 33, 45–47]. For example, Jensen *et al.* [33] used abstract interpretation to develop a tool to infer abstract types for the full language, although the formal theory only works for subsets. Others have studied information flow [13], with [30] proving their results in Coq.

All these techniques have been helpful for addressing specific safety problems. None provide general-purpose analyses, most do not work with the full language and, of those that prove soundness, all do so with respect to their abstract models rather than the ECMA semantics or an actual concrete implementation. The security issues identified in [35, 38, 39] demonstrate that the semantic subtleties of corner cases of the language matter crucially. Moreover, the empirical analysis by Richards *et al.* [52] confirms that some of the language features, excluded by construction from the work mentioned so far, are in fact important for actual web programmers.

In 2007, Herman and Flanagan [31] proposed a formalisation of the JavaScript semantics as a definitional interpreter written in ML. Their interpreter was the first executable formal semantics for a non-trivial subset of JavaScript. The advantages of their approach were the ability to test their interpreter and the familiarity of their work to functional programmers. The drawbacks were a loose correspondence with the specification and implementation details that sometimes obscured the semantics of the language features.

In 2008, Maffeis, Mitchell, and Taly [37] defined the first full operational semantics for the ES3 language. They covered the whole language, apart from a few corner cases such as regular expressions, dates, and machine arithmetic. The formalisation consists of a large set of small-step operational semantics rules and some theorems about the determinacy and well-definedness of the language. This work has been useful to prove soundness of security-related JavaScript subsets [35, 38, 39], and influenced the definition of further JavaScript formalisations. For example, the semantics of Secure ECMAScript underlying [57], and the big-step operational semantics of core JavaScript, proposed in [24] and used in [6], is based on this work. This formal semantics differentiates it-

self from the work described above because its goals were to cover the entire language, without excluding "uncomfortable" features, to serve as a basis for formal proofs of real language properties. The main shortcomings of this work is that, since the rules are not mechanised, the proof of language properties is labour intensive, the maintenance and extension of the semantics is not easy, and a comparison with implementations is impossible.

In 2010, Guha *et al.* [28] came up with a completely different approach to developing language semantics. They provide a translation from JavaScript to a Scheme-like, executable language, called λ_{JS}, which has as its core a simple λ-calculus with references. Their aim was to develop provably sound type systems to reason about the safety of client-side web applications. They target the Firefox implementation of ES3, and validate their semantics by testing it against the test262 and Mozilla test suites [18, 42]. More recently, λ_{JS} has been extended to model the strict mode of ES5 [50], and an unpublished, small-scale Coq formalisation of λ_{JS} has been announced on the Brown PLT blog [51].

The work on λ_{JS} has been influential in proving properties of well-behaved JavaScript typed subsets, where the programmer accepts restrictions on full JavaScript in exchange for safety guarantees. For example, Politz *et al.* [49] define a type system for λ_{JS} that captures the informal restrictions enforced by Crockford's ADSafe [16], a subset of JavaScript for sandboxing web advertising. Fournet *et al.* [21] define a translation between F*, a subset of Microsoft F# with refinement types, and λ_{JS}. They show that their encoding is fully abstract, hence the safety properties enjoyed by a source F* program are preserved when it is translated to JavaScript and run on a *trusted* web page. The λ_{JS}work has also been a significant source of inspiration for our development of our executable semantics, JSRef, and our focus on testing. We have tested JSRef using the test262 conformance tests, and in future will use the Firefox test suite to compare JSRef with the Firefox implementation.

Our work is not the first to give a formal and/or executable semantics to JavaScript. However, JSCert is the first semantics for the entire language, closely reflecting the official standard, which is both executable and formalised in a proof assistant. Working with the source language itself, and reflecting the structure of the specification has several advantages over a translational approach: the JavaScript programmer intuition is better reflected; the semantics is robust to local changes of the standard, such as those anticipated for ECMAScript 6 and 7; and the correctness of the semantics does not rely on possibly unknown assumptions associated with translation.[2]

Throughout this project, it has been important for us to establish *trust* in our mechanised semantics. As JSCert and JSRef are explored by ourselves and other formalists, by people compiling to JavaScript, by JavaScript implementers, and maybe even by ECMA authors, so our trust will increase. If someone questions our interpretation of ES5 or if a test fails, we know which part of JSCert to check and alter. Building on much of the work detailed above, we believe we have developed a methodology for providing a trusted mechanised specification of an English language standard. We do just wonder if this methodology has been fully demonstrated before.

3. The JavaScript standard

We give an overview of the semantics of JavaScript, as described in the ECMAScript 5 standard and implemented in current browsers (§ 3.1). We introduce our running example (§ 3.2), and identify the parts of the standard that have not been included in our mechanised specification (§ 3.3).

3.1 ECMAScript 5

ECMAScript 5 (ES5) is the current standard for JavaScript. Existing browsers largely implement ES5, but sometimes introduce custom extensions or make different choices in how to implement certain cases that the specification defines as "implementation dependent". In addition to the language, ES5 also describes a collection of native libraries, which provide convenient functionality (e.g., handling of regular expressions, date formats) as well as a few extra language features (useful for reflection). In this paper, we focus on the language itself, only specifying the native libraries necessary for the extra language features.

The grammar of JavaScript is divided into three main categories: expressions, statements, and programs. A JavaScript program consists of a list of statements. The body of a function definition is a JavaScript program. Similarly, the argument of the notorious `eval` statement, once parsed, is a JavaScript program. The result of the evaluation of an expression, statement, or program is a *completion triple*. A completion triple is composed of a type, a value, and a label. The type is one of `Normal`, `Return`, `Break`, `Continue`, or `Throw`, corresponding to the termination mode of the evaluation. The value, which can be empty, describes the result of an expression or statement (if the type is `Normal`), the value carried by a return statement (if the type is `Return`), or the object describing the exception being thrown (if the type is `Throw`). The label, which can also be empty, is only used for `Break` and `Continue` types, in order to divert the execution flow to the instruction annotated with the target label.

One of the reasons we were attracted to study ES5 is that it has not been defined with mechanised specification in mind. The standard is not optimised for conciseness and reuse. It has a lot of copy/paste which increases the workload of our formalisation. The standard uses representations that are practical for e.g., VM implementers, but less helpful for reasoning. For example, completion triples have implicit invariants that for our purposes would be better captured with an inductive definitions. Also, the standard becomes very particular when describing browser-specific extensions, and is not explicit about whether under-specified functions should have deterministic behaviours.

Apart from this determinism issue, we have found ES5 to be precise and non-ambiguous, with two exceptions. First, the specification of *data attributes*, used to represent parameters of properties (object fields) such as writable and enumerable flags, has two different representations in different parts of the specification. We give both representations in our specification and prove that they are equivalent, so that we can follow ES5 as closely as possible. Second, the specification of `for-in` suffers from major issues related to the loose specification of the order of enumeration of the property names [58].

3.2 Running example: the `while` statement

There are many interesting features in the semantics of JavaScript. These include prototype-based inheritance, return value propagation through "empty" statements, implicit type conversion (which may result in arbitrary side effects), the unique JavaScript approach to variable resolution, and the notorious `with` and `eval` statements. All of these features are properly described by our JSCert semantics. Since space is limited, we chose just one language feature to demonstrate our approach, and at least *some* of the subtleties of the JavaScript language. Our running example will be the `while` statement.

[2] In order to prove the correctness of such a translation, one needs a formal semantics for the source language in the first place. In this sense, our work will make it possible to prove the correctness of the encoding of JavaScript in λ_{JS}, increasing confidence in the validity of λ_{JS} or F* safety properties on actual JavaScript programs.

"`while` (*Expression*) *Statement*" is evaluated as follows:

1. Let V = `empty`.
2. Repeat
   a. Let *exprRef* be the result of evaluating *Expression*.
   b. If `ToBoolean(GetValue(`*exprRef*`))` is `false`, return (`normal`, V, `empty`).
   c. Let *stmt* be the result of evaluating *Statement*.
   d. If *stmt*.value is not empty, let V = *stmt*.value.
   e. If *stmt*.type is not `continue` || *stmt*.target is not in the current label set, then
      i. If *stmt*.type is `break` and *stmt*.target is in the current label set, then return (`normal`, V, `empty`).
      ii. If *stmt* is an abrupt completion, return *stmt*.

Figure 1. ECMAScript 5 semantics of while loops

Fig. 1 shows the English specification of while exactly as it appears in the ES5 standard. The text should be interpreted as pseudo-code, executing each statement in order. The ES5 specification of while is relatively short in comparison with other constructs, such as `switch`, whose specification spans more than one full page. This ES5 pseudo-code, like a traditional imperative programming language, leaves completely implicit three major aspects of the semantics. The first aspect is divergence; ES5 never talks explicitly about diverging programs, but it should be understood that if the evaluation of a sub-expression diverges, then the parent expression will also diverge. The second aspect is the threading of the mutable state; ES5 assumes that there is one global heap storing objects, and that the instructions in the pseudo-code can modify such heap. The third is the propagation of exceptions through expressions; ES5 describes the semantics of expressions in terms of the result of the sub-expressions, but does not recall every time that exceptions propagate outwards from expressions.[3] Unlike ES5, our JSCert specification is fully explicit about divergence, mutation, and exceptions.

We describe the pseudo-code from Fig. 1 in more detail. The basic skeleton is standard: repeat the loop body until the loop condition becomes false, or until the body of the loop produces a break, a return, or an exception. Step 2b is non-standard, illustrating one of the mechanisms necessary to interact with JavaScript's variable store. The result of an expression is not necessarily ready to be used, but may be a *reference* to a variable stored as a field of an object. The internal `GetValue` function is used to de-reference it. In addition, JavaScript uses the internal function `ToBoolean` to implicitly coerce the loop guard to a boolean before attempting to test it. A further complication comes from the fact that JavaScript allows labelled break and continue statements that may refer to an outer loop. A current loop may be interrupted by such labelled break and continue operations, just like it would be interrupted if an exception was thrown. In Fig. 1, the "current label set" refers to the set of labels that are associated with the current loop (since a loop might have several labels as in e.g., `a:b:c:while(1){break b;}`). Finally, notice that while loops have a return value in JavaScript. The definition of the JavaScript syntax forbids us from writing statements like `1+while(x>3){x--}`, however it does allow `1+eval("while(x>3){x--}")`. The assignments to the internal pseudo-code variable V in Fig. 1 are aimed at maintaining the value that will serve as the "result" of the current while loop. The value V is used in one of two cases: a break or a continue has been issued for this loop, or a break or a continue has been issued for an outer loop and does not yet carry a non-empty value.

Without further details, the reader may appreciate the complexity of JavaScript semantics with respect to output values of statements by looking at the following while loop examples, which return the last value that was produced by a statement in their body (if any):

```
eval("a: while(1){ while(1){ break a; }}")
==> undefined
eval("a: while(1){ while(1){ y=2; break a; }}")
==> 2
eval("a: while(1){ x=3; while(1){ y=2; break a; }}")
==> 2
eval("a: while(1){ x=3; while(1){ break a; }}")
==> 3
```

The most surprising example is the fourth one: 3 is returned because the statement `x=3` has completion value 3 and the statement `while(1){ break a; }` has an empty completion value.

3.3 What we do not specify

The ES5 standard is a document of 16 chapters, ranging over 209 pages, plus some additional annexes. It consists largely of pseudo-code in the style of Fig. 1, with clarifications in English. The document includes: the specification of parsing; the specification of the syntax (e.g., objects, heaps, environment records); the pseudo-code describing the semantics of expressions, statements, programs, and internal properties; and the pseudo-code describing the semantics of native library functions.

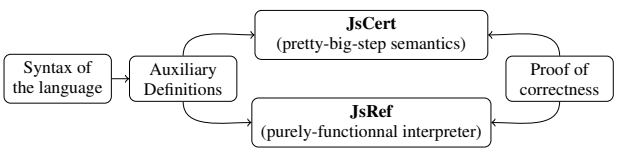
We have given a mechanised Coq specification for the important part of the language: the syntax (as an abstract syntax tree); the semantics of expressions, statements and programs; and most native library functions that expose internal features of JavaScript. In particular, we have specified the `Object`, `Function` and `Errors` libraries, with the only exception being functions involving the manipulation of arrays. We also have specifications for the constructors and most functions from the libraries `Boolean` and `Number`. For floating-point numbers, we rely on the Coq formalisation of IEEE 754 standard provided by the Flocq library [40].

We have not formalised other standard libraries, such as `Array`, `String` and `Date`. These libraries involve hundreds of functions. The vast majority of them do not interact with any internal feature of JavaScript, and could be implemented as plain JavaScript code [54]. The specification of these JavaScript standard library functions is orthogonal to the formalisation of the JavaScript programming language *per se*.

We also have not specified the parsing of JavaScript programs. Parsing is important because of JavaScript's `eval` statement, which requires parsing source code at runtime. The normalisation of the formal grammar of a programming language is more complicated than one might expect. The parsing of JavaScript programs is made particularly tricky by the treatment of implicit semi-colons, as it involves a one token look-ahead. We therefore leave the specification of parsing to future work, and use the parser of the Google Closure Compiler [25] in our reference interpreter JSRef.

4. JSCert: JavaScript specification in Coq

Our formal development in Coq, available from our project's webpage `http://www.jscert.org`, consists of five main parts:



[3] In ES5, the propagation of exceptions out of expressions is implicit, whereas the propagation of exceptions out of statements is explicit. This asymmetry is not only awkward, it is also a potential source of ambiguities. This issue has been resolved in ES6, where, like in our formalisation, the propagation of exceptions is always explicitly specified.

```
Inductive restype := (* result type *)
  | restype_normal
  | restype_break
  | restype_continue
  | restype_return
  | restype_throw.

Inductive resvalue := (* result value *)
  | resvalue_empty : resvalue
  | resvalue_value : value → resvalue
  | resvalue_ref : ref → resvalue.

Inductive reslabel := (* result label *)
  | reslabel_empty : reslabel
  | reslabel_string : string → reslabel.

Record res := { (* completion triple *)
  res_type : restype;
  res_value : resvalue;
  res_label : reslabel }.
```

Figure 2. JSCert completion triples

The first part describes the syntax and data structures, such as heaps and scopes, that are used to describe the formal semantics. The second part contains a collection of auxiliary definitions, such as functions used to convert a primitive value (a value that is not object) into a boolean, a number, or a string. These first two parts are described in § 4.1. The next two parts correspond to JSCert (§ 4.2) and JSRef (§ 5). The last part contains the correctness proof, that any result computed by JSRef is correct with respect to the semantics from JSCert (§ 6).

4.1 Syntax and auxiliary definitions

We give a simple summary of the syntax and auxiliary definitions shared by JSCert and JSRef. The full Coq specifications are given in the file `JsPreliminary.v`.

Abstract syntax tree. Following common practice, we do not model the parsing of language source code. Instead we work directly on the AST (abstract syntax tree). We obtain the AST by running the parser of the Google Closure Compiler [25], extended with some additional interface code defined by us.

The grammar of JavaScript expressions and statements is defined, in file `JsSyntax.v`, through a conventional inductive definition that represents the abstract syntax tree. This definition is relatively short, since there are only 13 kinds of expressions and 17 kinds of statements. We should note that we have factorised similar forms of if-statements and try-statements using option types, and that the definition of expressions depends on the grammar of literals (null, bool, number, string), the grammar of the 11 unary operators, and the grammar of the 24 binary operators. A JavaScript program consists of a list of statements, plus a strictness flag which stipulates the evaluation order of the list of statements. The body of a function definition is itself a JavaScript program. The argument of a call to `eval`, once parsed, is also a JavaScript program.

Completions. As we noted in § 3, the result of the evaluation of an expression, a statement or a program produces a *completion triple*, as given in Fig. 2. A completion triple (type `res`) is made of a type, an optional value (or reference, if the expression is a variable name), and an optional label. The type is one of `Normal`, `Return`, `Break`, `Continue`, or `Throw`, corresponding to the control flow directive obtained from an evaluation. The value, if any, is either a primitive value (a literal or `undefined`) or the location of an allocated object. This value describes the result of a `Normal` expression or statement, the value carried by a `Return` statement, or the exception being thrown in a `Throw` result. The optional label of a completion triple is only used for `Break` and `Continue` types, in order to implement the `break label` and `continue label` instructions.

A number of invariants apply to completion triples. For example, if a completion triple carries a non-empty label, then it must be of type `Break` or `Continue`. In order to better capture all the invariants on completion triples, we were tempted to use a clean inductive definition with one constructor per type, and different arguments for each type. However, recall that one of our goals is a direct correspondence between our JSCert specification and the ES5 standard. A change in the representation of something as fundamental as completion triples would put considerable distance between the English prose of the ES5 and our formalisation of it. Forced to choose between a more elegant Coq representation and a more direct correspondence with the ECMA standard, we chose the latter.

Program execution. A JavaScript program is always executed in a given *state* and in a given *execution context*. The state consists of the set of allocated objects (the *heap*), plus the set of allocated *environment records*.

The heap. The JavaScript object heap is represented as a finite map from locations to objects. An object is represented as a record with 25 fields, including the `prototype` object, the `class` name, a boolean `extensible` property, an optional `primitive value` (for objects of class `Boolean`, `Number` or `String`), the *property map*, as well as internal methods such as `get` or `put`, optional fields such as `body` and `scope` for functions, and optional internal methods such as `call` or `construct`. Each internal method may have a small number of different behaviours depending on the object on which it is called (Arguments objects, for instance, have a different implementation of `[[Get]]`). We distinguish these different behaviours using a tag that describes how the internal method should be processed.

Object properties. The property map binds field names to *property attributes*, rather than directly to values. A property attribute is a record that contains a number of fields. There are two kinds of property attributes: *data property attributes* which have a `value` field, a `writable` field, an `enumerable` field, and a `configurable` field; and *data accessor attributes* which have a `get` field, a `set` field, an `enumerable` field, and a `configurable` field.

The ES5 standard suggests that property attributes should be represented as a record with 6 optional fields. In particular, the specification of the function `DefineOwnProperty` involves the construction of a property attribute, which explicitly manipulates records using arbitrary subsets of the 6 optional fields. At the same time, in many other places, the standard uses exactly 4 fields, (loosely) making the assumption that particular fields are present and implicitly exploiting the fact that the property attribute is known to be either a data property or a data accessor attribute from the context and implicit invariants.

Since we wanted to be faithful to the standard everywhere, we chose to provide two distinct representations of data properties: the first consists of a record with 6 optional fields; and the second consists of an inductive type with two cases, one for data property attributes and one for data accessor attributes, both represented as records with exactly 4 mandatory fields. The overhead of defining conversion functions between the two forms was negligible compared to the benefits of avoiding the pollution of many rules with accesses to optional fields.

Environment records. Recall that the JavaScript state consists of the heap and the set of allocated environment records. We store

environment records in a similar data structure to that used for the object heap, using a pointer to refer to a given environment record. An environment record can take one of two forms: a *declarative environment record*, which maps variable names to values (and to mutability flags); or an *object environment record*, which consists of the location of a JavaScript object (and some "provide-this" flags). Declarative environment records provide the local scoping associated with function calls, while object environment records provide the dynamic scoping associated with the `with` construct. In addition, there is always at least one object environment record which points to *the global object*. To properly model function closures, a given function object represents the environment it closes over as a *lexical environment* stored in one of its internal fields.

Execution context. The execution context is represented as a triple that consists of the scope (both the *lexical environment* and the *variable environment*, as specified by ES5), the current "this" object, and the current strictness flag. We represent both lexical environments and variable environments as a stack (a list) of pointers to environment records.

4.2 JSCert

Based on the syntax and auxiliary definitions described above, the main JSCert rules are defined in the file `JsPrettyRules.v`. We formalise the semantics of JavaScript statements using a judgement of the form $t/S/C \Downarrow_s o$, where: t denotes a statement; S denotes the state comprising the object heap and the environment record heap; C denotes the context comprising the lexical and variable environment, the ES5 "this" object, and the strictness flag; and o denotes the *output*. We have similar judgements $\Downarrow_e, \Downarrow_i$ and $\Downarrow_p$ for expressions, internal reductions, and full programs.

For terminating computations, the output is a pair made of the final state and the completion triple produced by the evaluation. Our JSCert semantics, expressed in pretty-big-step style [11], also captures diverging computations, using a coinductive interpretation of the same set of evaluation rules. We discuss this further below. One key ingredient to the factorisation of the evaluation rules is the definition of the output of an evaluation as a sum type, shown below, that describes either termination in a given state with a given result, or divergence:

```
Inductive out :=
| out_ter : state → res → out
| out_div : out.
```

Internal reductions, however, return results of many different types. Some, for instance, return property descriptors. We thus use a more general return type, `specret T`, that is parameterised by the type `T` of what is returned. To add further complexity, internal reductions may call arbitrary user code which may terminate with an abrupt termination, such as throwing an exception, or which may diverge. Their return type is, thus, not uniform: it returns a modified state and a term of type `T` when the computation is successful; it returns a term of type `out` when the computation diverges or results in an abrupt termination. This behaviour is captured by the following type:

```
Inductive specret T :=
  | specret_val : state → T → specret T
  | specret_out : out → specret T.
```

Pretty-big-step semantics. We express the JSCert semantics using the *pretty-big-step* operational semantics recently developed by Charguéraud [11]. The key difference between the traditional big-step semantics and Charguéraud's pretty-big-step semantics is that we can decompose the evaluation of a single program construct using *intermediate forms* (defined in `JsPrettyInterm.v`), which extend the grammar of program statements and can be evaluated just like any other program. This style of semantics allows us to more effectively match the modularity of the ES5 standard.

The advantages of the *pretty-big-step* semantics have been described in [11]. Here, we only recall the arguments applicable to large existing language standards such as ES5. Nearly all English standards for industrial programming languages, including ES5, use sentences of the form "let R be the result of evaluating t". These sentences relate a term directly to its result, just as a big-step judgement would do. Because we want to be close to ES5, we cannot work with a small-step presentation, as in [37], with rules of the form "to evaluate `if e then t1 else t2`, execute one step to reduce `e` into `e1`, and then evaluate `if e1 then t1 else t2`."

If we attempt to use traditional big-step semantics, we quickly find that we have to duplicate a significant amount of material across several rules. For example, consider the loop `while(e){ t }`. In big-step, we need a rule to handle the case where e evaluates to an exception, another rule for the case where e evaluates to a value converted to the boolean false, yet another rule for the case where e evaluates to a value converted to the boolean true and the body of the loop evaluates to an exception, and so on.

The problem here is that the steps made by a big-step semantics are "too big" to cleanly correspond to ES5. As suggested by our example, if we attempt to shoehorn ES5 into a big-step presentation, our repetition of premises will lead to a quadratic explosion in the size of our rule set. For an idealised research programming language, this sort of duplication may not present much of a problem. However, we soon realised that formalising ES5 in this way would quickly become unmanageable. This observation motivated the development of the pretty-big-step semantics, which has enabled us to write rules which very closely follow the structure of ES5.

Pretty-big-step semantics for while loops. Consider the ES5 description of while loops given in Fig. 1. The corresponding JSCert semantics is given in Fig. 3. Notice the close correspondence between the steps of the ES5 pseudo-code and the JSCert rules. Step 1 of the ES5 pseudo-code says "Let V=empty". The corresponding rule red_while_1 uses the notation `stat_while L e1 t2` to refer to a `while` construct with guard `e1`, body `t2` and label set `L` (used for managing `break` and `continue` statements). The rule says that in a given state, the loop may evaluate to the output o if the *intermediate form*

```
stat_while_1 L e1 t2 resvalue_empty
```

also evaluates to o. This intermediate form carries all the information of the original `while` construct, and also carries the additional information that the value of "V" is `resvalue_empty`, which is our Coq representation of the ES5 "empty" value.

Step 2 of Fig. 1 begins the loop in the ES5 pseudo-code. In our JSCert rules, we may loop back to this point at any time, by using the `stat_while_1` intermediate form as a premise of a rule.

Now let us consider Steps 2a and 2b. These steps represent a pattern which is very common in ES5. First we evaluate some sub-expression, then we perform a `GetValue` *and* a type conversion on the result of that evaluation. In this case, we are converting to a boolean. Notice that a great deal is left implicit in the ES5 pseudo-code, which we wish to make explicit in our rules: the sub-expression or statement could diverge or throw an exception; the type conversion could diverge or throw an exception;[4] and both the sub-evaluation *and* the type conversion could have side effects on the program state. This pattern occurs so frequently that we introduced a special intermediate form to handle

[4] This is actually not the case for conversions to booleans, but it may happen in conversions to other types, such as String or Number.

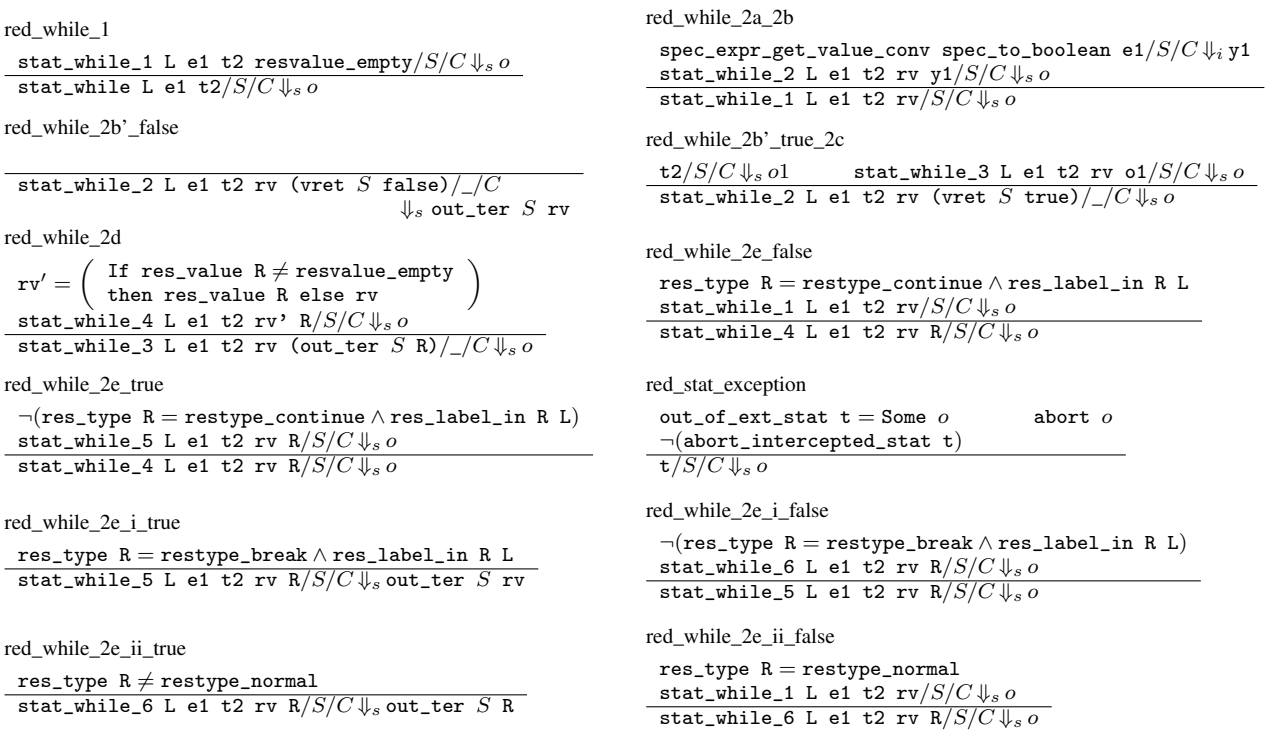


Figure 3. JSCert semantics of while loops

it, while making these side effects, divergence, and exception propagation clear. The reader may see this special intermediate form at work in the rule red_while_2a_2b. Here, the intermediate form `spec_expr_get_value_conv` takes care of the evaluation of `e1`, its `GetValue`, and its type conversion. In this case, we specify which type to convert to using the flag `spec_to_boolean`. The remaining work of Step 2b is performed by rule red_while_2b'_false. Notice that since the type-conversion may have side effects, the rule red_while_2b'_false takes its initial state S from the result of the type conversion as given by the intermediate form. Rule red_while_2b'_false is a terminating rule, since the corresponding ES5 step says to "return (normal, V, empty)". In our formalisation, the pseudo-code variable V corresponds to the variable `rv`. Since we do not mention a completion type, our formalism assumes the completion type "normal", and since we do not mention a label set, our formalism assumes the label set "empty". This is done in Coq using a type coercion:

```
Coercion res_normal rv := { res_type = restype_normal;
    res_value = rv; res_label = label_empty }
```

Step 2c (which corresponds to rule red_while_2b'_true_2c) follows the pretty-big-step pattern: evaluate some sub-expression or statement (in this case `t2`), and store the result in some pseudo-code variable. Each new pseudo-code variable becomes a parameter of a new intermediate form – in this case the `o1` parameter of `stat_while_3`. Notice that the rules red_while_2b'_false and red_while_2b'_true_2c both assume that the result of the type-conversion performed by rule red_while_2a_2b was successful. In the event that the type conversion diverges, or throws an exception, that "aborting computation" will be propagated by the general rule red_stat_exception[5]. We handle divergence and exceptional termination of the loop body (*Statement* in ES5 and `t2` in our formalism) in exactly the same way. The rule red_stat_exception uses the `out_of_ext_stat` function to extract the output from any intermediate form, checks that this output is an aborting one, and checks that this aborting computation should not be intercepted by a more specific rule. For example, in a while loop computation, if an exception or divergence happens in Step 2c, instead of always propagating the exception/divergence, in some cases we want to intercept it. That is, the part of the definition of `abort_intercepted_stat t` is the case where `t` can be pattern-matched with `stat_while_3 L e1 t2 rv (out_ter S R)`. In this case the value of `abort_intercepted_stat t` is:

$$\texttt{res\_label\_in R L} \wedge \left(\begin{array}{l} \texttt{res\_type R} = \texttt{restype\_continue} \vee \\ \texttt{res\_type R} = \texttt{restype\_break} \end{array} \right)$$

This condition checks for exceptional terminations caused by `continue` and `break` statements that target the particular loop that we are currently evaluating, and should be properly handled by Step 2e (below).

Step 2d is another conditional assignment, which we handle with a condition of our own in rule red_while_2d. Notice that we insist that we will only proceed with step 2d if our previous steps terminated. This insistence is encoded in our pattern matching for `out_ter S R` — which is the result of a terminating computation. The case in which the previous computation did not terminate is handled also by the abort rule which propagates diverging results.

Step 2e begins a nested conditional expression. It is simplest to describe the "false" case first, since this results in simply looping back to the beginning of Step 2. This case is handled by rule red_while_2e_false, the first premise of which is the negation of the guard written in Fig. 1. The second premise of this rule begins the next iteration of the loop by re-using the intermediate form

[5] We have similar "aborting computation" rules for expressions, internal reductions, and full programs.

stat_while_1. In the "true" case, rule red_while_2e_true continues with a new intermediate form.

Step 2e(i) is another conditional expression, which we describe with two simple rules. The "true" case which describes the terminating computation breaking out of the loop corresponds to rule red_while_2e_i_true. The "false" case, which continues to step 2e(ii) corresponds to rule red_while_2e_i_false.

Finally, we come to step 2e(ii): another conditional, modelled with a pair of rules. The "abrupt completion" case, which terminates the computation corresponds to rule red_while_2e_ii_true. The "normal" case, which loops back to the beginning of Step 2 using the stat_while_1 intermediate form corresponds to rule red_while_2e_ii_false.

It is a simple matter to express the rules given above as Coq definitions.[6] For example the rule red_while_2e_ii_false can be written in Coq as follows.

```
| red_while_2e_ii_false : ∀S C labs e1 t2 rv R o,
    res_type R = restype_normal →
    red_stat S C (stat_while_1 labs e1 t2 rv) o →
    red_stat S C (stat_while_6 labs e1 t2 rv R) o
```

JavaScript formalisation challenges. The first challenge with formalising ES5 is the size of the definition. ES5 consists of 16 chapters (209 pages, plus some additional annexes) of English language and pseudo-code description. While we do not aim to formalise the parsing of concrete JavaScript syntax (most of chapters 1-7), JSCert and JSRef cover, and have a correctness proof for, all of chapters 8-14 and 16, except for the for-in statement (chapter 12), array initialisation (chapter 11) and some type conversions (chapter 9). See http://www.jscert.org for the current state of our JSCert specification, which will grow over time. We choose not to formalise for-in at this stage because it is loosely specified and broken: based on the ES5 standard and conversations with ES5 authors and browser implementors, we filed bugs in the current draft of the upcoming ES6 standard (bugs 1444 and 1443 of [58]). Still, it is an interesting challenge to give a more precise description of the semantics of for-in and we leave it for future work. Chapter 15 contains the native libraries, and would be considerable work to specify in full. We have JSCert rules and JSRef code (but no correctness proof) for those parts of chapter 15 where the libraries expose important JavaScript features.

The inductive definition of JSCert is sometimes a little verbose, as illustrated by the rules for while. This verbosity is due to the fact that we closely follow ES5, which is itself quite verbose, and we are being fully explicit about the evolution of the state and the context. In contrast, we shall see that the JSRef fix-point presentation is significantly more compact (§ 5). Despite its verbosity, the inductive presentation of JSCert is ideally suited to conducting formal proofs, because it gives a fine-grained view of the different steps, allowing modular reasoning about the evaluation of a given JavaScript language construct. Moreover, as we have experienced with the verification of the code of JSRef with respect to JSCert, the regularity of the pretty-big-step rules (which never have more than two evaluation premises), significantly eases the development of specialised tactics that help automate formal proofs.

The second main challenge with formalising ES5 is the high complexity of the behaviour of a few features such as the representation of scopes and variable lookup, the non-obvious representation of data attributes, the treatment of function calls and arguments object, and the aforementioned evaluation of for-in. Several features introduced in ES5, such as execution contexts and strict mode, are spread throughout the specification and hence are non-trivial to capture.

Finally, JavaScript "internal methods" can return types not allowed to user code, and yet may call user code. We thus had to devise a definition that was flexible, yet allowed for proofs to be automated as much as possible. The next section on JSRef illustrates how such an effort enabled us to factorise much of the presentation using a monadic approach.

Semantics of diverging programs. ES5 never explicitly mentions diverging programs — a program diverges as soon as the evaluation of any subprogram diverges. Even though we have not conducted so far any formal reasoning related to divergence, we could formally capture the semantics of diverging JavaScript programs without requiring additional work. The constant out_div, which we introduced in the grammar of outputs o, materialises divergence. This constant is propagated through the evaluation rules, just as exceptions are propagated. We derive the judgement capturing the semantics of diverging programs by considering a *coinductive* interpretation (greatest fixed-point) of the exact same set of evaluation rules as we have for terminating programs. Additional details may be found in [11].

5. JSRef: a reference interpreter for JavaScript

Our goal in defining JSRef is to obtain an executable JavaScript interpreter, whose definition closely follows ES5 and can be proved correct with respect to JSCert. Note that JSRef does not need to be fast. It simply needs to be efficient enough to interpret programs such as those found in the JavaScript test suites.

Host language for JSRef. We have chosen to write JSRef directly in Coq: that is, in a core, purely-functional language that admits only total functions. We could have chosen to use an imperative language with implicit state in the hope that this would make translating ES5 pseudo-code easier. However, this would have made the correspondence between our interpreter and our formal specification much more difficult to establish. By using Coq as a programming language, we make JavaScript's state and exception propagation explicit (which we consider an advantage), and make the formal statement and proof that our interpreter is sound comparatively straightforward, as described in § 6.

Structure of JSRef. The interpreter consists of a record that contains functions for evaluating programs, statements, expressions, function calls, and so forth. The record type, slightly simplified for clarity, appears below:

```
Record runs_type : Type := runs_type_intro {
    runs_type_expr : state → execution_ctx → expr → result;
    runs_type_stat : state → execution_ctx → stat → result;
    runs_type_prog : state → execution_ctx → prog → result;
    runs_type_stat_while : state → execution_ctx → resvalue
        → label_set → expr → stat → result; ...}.
```

Since our Coq implementation must consist only of terminating functions, we bound the number of steps that JSRef may take in any given run, a standard technique in ACL2 and Coq. This is similar to decide in advance "If it hasn't finished in 30 days, I'll hit Control-C". The token result_bottom is the result of a computation that was interrupted in this way. In practice, it is rare to observe result_bottom, because we can specify a very large bound on the number of steps. We thus define our runs function as a fixpoint which takes an integer as an argument and returns a record of functions with the type of the record above:

[6] In our Coq development, we use a slightly different naming convention for the evaluation rules, which we believe will be easier to maintain when migrating from one version of the ECMA standard to the next. That said, for pseudo-code involving more than 5 steps, we left a comment next to each evaluation rule indicating its corresponding ES5 step number.

```
Fixpoint runs max_step : runs_type := match max_step with
 | 0 ⇒
  { runs_type_expr := fun S _ _ ⇒result_bottom S;
    runs_type_stat := fun S _ _ ⇒result_bottom S;
    runs_type_prog := fun S _ _ ⇒result_bottom S;
    runs_type_stat_while := fun S _ _ _ _ _ ⇒result_bottom S;
  ...}
 | S max_step' ⇒(* max_step = 1 + max_step' *)
  { runs_type_expr := fun S ⇒run_expr (runs max_step') S;
    runs_type_stat := fun S ⇒run_stat (runs max_step') S;
    runs_type_prog := fun S ⇒run_prog (runs max_step') S;
    runs_type_stat_while := fun S ⇒
      run_stat_while (runs max_step') S;
  ...}
 end.
```

Each function in the record takes the record itself as its first parameter, instantiated with one fewer step, and all recursive calls are routed through the record. For example, `run_stat runs S C s` evaluates the statement `s` in the *state* `S` (heap of objects and environment records) and in the *evaluation context* `C` (used for variable lookups), so long as the record `runs` contains functions for running any kind of term (programs, statements, expressions, etc.). As a first approximation, the reader may simply think of the projections `runs.runs_type_expr` as a direct recursive call to the function for evaluating expressions.

The function `run_javascript` below uses a record `runs` built with a sufficiently large number to serve as a bound on our recursive calls. Since the bound is handled by this record, it does not pollute our main development.

```
Definition run_javascript runs p : result :=
  runs.runs_type_prog state_init execution_ctx_init p.
```

An evaluation using a function from JSRef returns a *result*, which is either a completed computation, or a special token that states the interpreter has reached an impossible state or that the computation did not terminate in the allocated time. As with JSCert, the type of the result may depend on what is being evaluated (typically for internal reductions); we thus define the following polymorphic `resultof` type:

```
Inductive resultof T :=
| result_some : T → resultof T
| result_impossible
| result_bottom : state → resultof T.
```

The token `result_impossible` is returned by the interpreter if an invariant of the JavaScript language is violated: e.g., if the ES5 internal `GetOwnProperty` method is somehow called on a primitive value. We claim that starting from a well-formed initial state, the interpreter will never return `result_impossible` because we conjecture than all invariants are preserved by execution.

As first approximation, terminating programs have a result of type `resultof out` (as defined in § 4.2, except that `out_div` is never used), whereas internal reduction have a result of type `resultof (specret T)` for some `T`. In fact, for purpose of better code factorisation, terminating programs have the following `result` type, which is isomorphic to `resultof out`:

```
Inductive nothing : Type :=. (* uninhabited *)
Definition result := resultof (specret nothing).
```

Since there is no inhabitant of the `nothing` type, the only way to obtain a `result_some` of the `result` type is through `specret_out`, which carries a value of type `out`. This approach guarantees that every result type is an instance of `specres T = resultof (specret T)`, greatly simplifying the definition of monadic operators.

Monadic-style programming in JSRef. In order to avoid clutter, JSRef is programmed in a *monadic style*. For example, say

```
Definition if_result_some (A B : Type)
  (W : resultof A) (K : A → resultof B) : resultof B :=
  match W with
  | result_some a ⇒K a
  | result_impossible S ⇒result_impossible S
  | result_bottom S ⇒result_bottom S end.

Definition if_spec (A B : Type)
  (W : specres A) (K : state → A → specres B) : specres B :=
  if_result_some W (fun sp ⇒
    match sp with
    | specret_val S0 a ⇒K S0 a
    | specret_out o ⇒
        if_abort o (fun _ ⇒result_some (specret_out o))
    end).
```

Figure 4. Two JSRef monadic operators

```
1  Definition run_stat_while runs S C rv labs e1 t2 : result :=
2    if_spec (run_expr_get_value runs S C e1) (fun S1 v1 ⇒
3      Let b := convert_value_to_boolean v1 in
4      if b then
5        if_ter (runs.runs_type_stat S1 C t2) (fun S2 R ⇒
6          Let rv' := ifb res_value R ≠resvalue_empty
7                      then res_value R else rv in
8          Let loop := fun _ ⇒runs.runs_type_stat_while S2 C rv'
                labs e1 t2 in
9          ifb res_type R ≠restype_continue
10               ∨ res_label_in R labs
11         then (ifb res_type R = restype_break
12                ∧ res_label_in R labs
13                then res_ter S2 rv'
14           else (ifb res_type R ≠restype_normal
15                 then res_ter S2 R else loop tt))
16         else loop tt)
17     else res_ter S1 rv).
18
19 Definition run_stat runs S C t : result :=
20   match t with
21   | stat_while ls e1 t2 ⇒
22     runs.runs_type_stat_while S C ls e1 t2 resvalue_empty ...
```

Figure 5. JSRef semantics of while loops

we want to evaluate a composite expression e which contains subexpression e_1. First we evaluate e_1. If this evaluation terminates with a completion triple of type normal, then we want to use the value produced by e_1. However, if the evaluation of e_1 produces either `result_bottom` or `result_impossible`, we want to propagate this result immediately, without executing the rest of the code for processing e. In JSRef, this pattern requires a simple use of the `if_spec` monadic operator: `if_spec (runs.run_expr_get_value S C e1)(fun S' v1 → ...)`, where the "..." contains the code for processing the result value `v1` of `e1` in the updated state `S1`. Fig. 4 shows the definition of `if_spec`. It depends on `if_result_some` which first filters out the cases where the computation failed. Otherwise, the continuation `K` is called on the result. The `if_spec` function constructs such a continuation (`fun sp ⇒...`), which ensures that the result of the computation is not an abrupt termination (a result of the form `specret_out o` where `o` satisfies the predicate `abort`).

JSRef function for while loops. Fig. 5 shows the function `run_stat_while` used to interpret while statements. The three first arguments have been described before. Arguments `e1` and `t2` are respectively the condition and body of the while loop. Argument `labs` is a set of labels annotating the loop (to deal with `break` and `continue` statements). Finally, `rv` is the current value to return in

the completion triple. It corresponds to the V of Fig. 1. Intuitively, calling this function amounts to executing the JavaScript statement `labs:while(e1){t2}` starting from heap `S` and execution context `C`. As we reuse this function for the next step of the loop, we also need to carry around the last computed value `rv`. As shown in the extract from `run_stat`, `rv` is initially set to the empty value.

We now detail the code of the loop. We first evaluate the condition `e1` and capture its result using the continuation (`fun S1 v1` ⇒...). Remember that this continuation runs only if the result of the computation is successful and if it is not an abrupt termination. The value `v1` is then converted to a boolean to choose which branch of the `if` to execute. If it is `false`, the `else` branch is taken (Line 17), and the current state is returned alongside with the last computed value `rv` (coerced to the triple (`normal`,`rv'`,`empty`)). Otherwise, the statement `t2` is evaluated using the monadic operator `if_ter`. This operator is very similar to `if_spec`, except that it applies the continuation even if the result is an abrupt termination. This allows us to check for a `break` or `continue` result from running the statement. Lines 6 and 7 update `rv` if the result value of the statement was not empty. We then inspect the termination type of the result to proceed. Line 16 is taken if it is a `continue` with its label in `labs`. Otherwise, if the result is a `break` with its label in `labs`, then the computation terminates as a normal result (Line 13). If the result is not normal (e.g., a `return` or a `break` with a different label), then it is returned as such (Line 15, then branch). Finally, if the result is normal, then the next iteration of the while loop is run (else branch).

Note how the while loop code in JSRef is more concise than that in JSCert. This observation applies in general to most parts of the definitions. Overall, the full definition of JSCert is ∼3,000 lines of Coq, whereas the corresponding definition of JSRef is ∼2,000 lines.

Running the interpreter. The Coq system provides a way to automatically extract OCaml code from the definition of computable functions. This is crucial to our ability to run our interpreter and test it against existing test suites. There are experimental tools that allow the extraction of OCaml code directly from an inductive definition, such as the one of JSCert. We did not pursue this approach for the following reasons: these tools are not yet mature; the development of JSRef is independently interesting; and, this way, we could make natural choices when ES5 was underspecified.

At the expense of some trust, Coq provides the ability to locally override the default mapping from Coq values and types to OCaml values and types. Of course, this feature should be used sparingly. We use it in two ways. First, as previously explained, we rely on an untrusted parser. More precisely, our development assumes the existence of a parser returning either *some* AST or *none* in case of a parse error: `Axiom run_parse : string → option prog` In order to run tests and execute the `eval` operator, we provide an OCaml function that implements `run_parse` by calling an existing JavaScript parser [25] and then translating the output to the OCaml representation of our AST for JavaScript code. The second customisation regards numbers. In JavaScript, all numbers are IEEE 754 double precision floating-point numbers. In our formalisation, we use the Flocq library [40] to precisely model IEEE 754 floating-point numbers and their operations. Since the OCaml type `float` corresponds to IEEE 754, it is safe for us to extract JavaScript numbers directly to OCaml `float`. Similarly, we provide direct OCaml implementations for other operations on numbers mentioned in ES5, such as conversion to and from `Int32` types.

6. Evaluation: establishing trust

An important aim for us was to design JSCert and JSRef in such a way that they could be evaluated, and hence eventually trusted, by e.g., ECMA authors, implementors of JavaScript virtual machines, designers of secure subsets or compilers targeting JavaScript, and developers of JavaScript analysis tools. In this section, we describe our methodology for establishing trust, how to extend our results in future to strengthen such trust, and the bugs that we have found in ES5 and ES6, test262, and several major browsers.

Our methodology for establishing trust involves four components: the English standard ES5 and the ECMA conformance test suite, test 262, pre-existed our work; the mechanised specification, JSCert, and the certified interpreter, JSRef, are introduced in this paper. We establish connections between ES5, JSCert, JSRef and test 262, to justify our claim that JSCert and JSRef have been designed in such a way that they can be evaluated and trusted. We also explore ways of establishing further connections in future.

We have constructed JSCert to be as close as possible to ES5, and have proved JSRef correct with respect to JSCert. Independently, engineers have developed test 262 to cover as many aspects of ES5 as they reasonably could. We have been able to check that JSRef behaves as expected on all the appropriate tests, given our coverage of ES5. JSCert and JSRef can therefore be challenged through two distinct paths: through the eyeball correspondence with ES5; and through the execution of tests by our reference interpreter. Having these two *independent* paths significantly decreases the likelihood of bugs remaining in JSCert.

6.1 Connections to establish trust

Eyeball closeness. As discussed in § 4, we have designed JSCert to be as close to ES5 as we can. We follow the ES5 data structures precisely. Every line of pseudocode in ES5 corresponds to one or two rules in JSCert, and our code is commented to make these correspondences explicit. Anyone with basic training in reading Coq specifications should be able to check the correspondence between the ES5 prose and our JSCert definitions.

We have intentionally chosen to differ from ES5 at a few places. We make explicit several constructs that are left implicit in ES5, to help with the eyeball assessment that JSCert is a correct formalisation of ES5. Unlike ES5, we treat the state, exceptions, and divergence explicitly. We always mention the current evaluation context and strictness flag, whereas ES5 only describes places where they are modified. Moreover, contrary to ES5 pseudo-code, in JSCert we do not use a "repeat" statement but rely instead on an explicit control-flow jump. We did experiment with an higher-order intermediate form to capture "repeat" loops in the inductive semantics. However, we concluded that it obfuscated the definitions and added technicalities for very limited benefit. Setting aside these differences in the style of presentation, we believe that JSCert is close enough to ES5 that (at least several) ECMA authors should be able to proof read our definitions

Correctness. We have formally proved in Coq that JSRef is correct with respect to JSCert. More precisely, if the JSRef interpreter evaluates program p to output o, in a finite number of steps bounded by n, then it must be the case that the program p is related to the output o by the inductive semantics of JSCert. More formally, the Coq statement of the theorem is

```
Theorem run_javascript_correct : ∀(n:nat) (p:prog) (o:out),
  run_javascript (runs n) p = result_some (specret_out o) →
  red_javascript p o.
```

where `red_javascript` corresponds to the evaluation judgement $\Downarrow_p$ specialised to the initial state. The proof of this theorem consists of 3500 lines of Coq and is relatively straightforward. We inspect the code of JSRef line by line, following the case analyses and the function call performed and showing that each of these operations corresponds to the application of one or two pretty-big-step evaluation rules from JSCert.

More precisely, recall from § 5 that JSRef is written in a monadic style. To step through the code of JSRef, we need to reason about the behaviour of the monadic operators. To do this, we prove, for each monadic operator, a lemma specifying its behaviour. We then define a custom Coq tactic that looks at the head monadic operator of the JSRef piece of code at hand, and automatically applies the corresponding lemma. Thanks to this tactic, we are able to automate the reasoning on the abrupt termination cases. As a result, our proof script basically consists of case analyses and calls to our custom tactic with the names of the evaluation rules to be applied given as arguments.

As we were proving correctness, we were able to detect and correct many typos, and also a small number of more serious misinterpretations of ES5. JSCert and JSRef were intentionally developed by different researchers. Despite close interaction between researchers, there were inevitable discrepancies in interpretation between JSCert and JSRef, which were picked up during the course of proving correctness. This proof is a cornerstone of our work, as it enables us directly to validate our JSCert specification with testing.

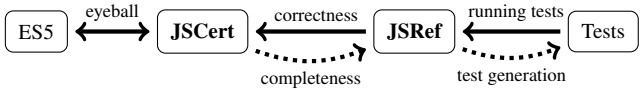
Tests. We ran JSRef against test262, the ES5 conformance test suite. JSRef successfully executed all the tests that we expected it to pass, given our coverage of ES5. In test262, there are 11746 tests, organised by chapters. There are no tests for chapters 1–5. Chapters 6 and 7 relate to the parser rather than the language, chapter 15 corresponds to native libraries, and there are some additional test directories (e.g. "best practice" and "intl402") which we do not study. There are 2782 main tests associated with chapters 8–14. Our of these, we pass 1796. We have been able to check (with the help of search tools) that all the remaining tests fail because: either they use for-in; or they use chapter 15 library functionalities which we have not implemented; or they fail due to the Google Closure parser being slightly more permissive than ES5 and hence not failing where expected.

Running JSRef over thousands of tests has been very useful, as it allowed us to detect and fix several bugs in JSRef and JSCert. Most of these bugs were simple typos, but a few of them were more serious: e.g., converting a record to a data property instead of a data accessor.

6.2 Increasing trust in JSCert and JSRef

Since we are discussing trust, we ought to also recall that, by design, the correctness of our development relies on the formal tools, formal libraries, parsers, compilers and glue code involved in our tool chain. More precisely, trust in JSCert and JSRef requires trust in Coq, Coq's extraction mechanism, the OCaml compiler, the Google Closure parser, our glue code for linking the parser to the extracted code of JSRef, and our glue code for binding boolean and floating-point JavaScript values to their OCaml counterparts.

There are three main ways in which we may further increase our trust in JSCert and JSRef: by establishing determinacy for most of the evaluation rules for JSCert; by establishing the completeness of JSRef with respect to JSCert; and by using a code-coverage analysis of JSRef to help us complete existing test suites so as to cover all aspects of ES5. The last two correspond to the dotted arrows in the following diagram:



ES5 does not explicitly say that the semantics of JavaScript is deterministic. Yet, it turns out that, with the exception of the loosely-specified "for-in" statement and "implementation-dependent" constructs, the standard only describes deterministic behaviour. We should be able to prove this. Conducting such a proof would ensure that JSCert does not contain errors leading to more behaviours being accepted by JSCert than ES5. If JSCert was erroneous in this way, it could compromise e.g. the verification of tools that compile other programming languages into JavaScript. In future, we would like to prove in Coq that JSCert is deterministic. We have not attempted to do this proof yet due to a limitation of the case-analysis tactic of Coq. This tactic leads to an (unnecessary) quadratic explosion of the size of the proof-term when performing a nested case analysis on two derivations, precisely what is necessary for proving determinacy. We are waiting for the release of the new case-analysis tactic, under development, which should enable us to build proof terms of size linear in that of the proof.

We would also like to prove the completeness of JSRef with respect to JSCert. Currently, when executing a JavaScript program that terminates within a reasonable amount of time, our interpreter might, in theory, return `result_impossible` or `result_bottom` instead of the expected result. (Note that, as established by the correctness result, if JSRef produces some result then it must be the right result.) In practice, on all the tests that we considered, we never obtained `result_impossible` or `result_bottom`. This situation is very unlikely to happen because, from looking at the source code of JSRef, it appears obvious that we never return `result_bottom` except when the bound on the number of steps is reached, and it is relatively straightforward to check we would only return `result_impossible` if the invariants on the state (e.g., there are no dandling pointers) were to be broken. That said, it would be more satisfying to prove in Coq that if a program is related to some output using JSCert, then executing this program using JSRef would produce the same output. The reason we have not done it yet is because the invariants on the states involved are numerous and quite arduous to define, and because completeness is not critical to safety.

Another way to increase the trust in JSCert and JSRef would be to extend existing test suites so as to include sufficiently many tests to cover all the paths of ES5. With such full coverage, we would significantly increase the degree of confidence in the correctness of JSRef through the testing path. The ECMAScript community acknowledges that the test262 coverage needs improvement, and in fact has a long-standing open bug on the test262 bugzilla entitled "Need academic-like review of existing test coverage versus ES5.1" [23]. Using JSRef in combination with a coverage analysis tool for OCaml programs, called the *Bisect* tool [14], we have been able to visualise the lines of JSRef that are never executed by any test program. We therefore have at least some technology for investigating the coverage of JSRef, and hence of ES5 (provided that JSRef does not miss entire pieces of ES5). We need more manpower to extend test262 with the missing tests.

An alternative approach to producing tests manually would be to generate them automatically, by using automated theorem provers to exhibit tests whose execution is able to reach a particular line of JSRef code. As these tests would be generated from JSRef, they would certainly not trigger bugs in JSRef. However, if these tests were to fail on other JavaScript virtual machines, then we would know that there is either a bug in JSRef or in the virtual machines. This alternative approach to producing tests would increase our trust in JSRef, by showing that it mainly behaves like existing JavaScript implementations on tests that cover all of its source code and by highlighting where it diverges from particular implementations.

6.3 Bugs discovered

Through our work, we have been trying to understand and express in Coq the semantics of JavaScript, including its darkest corners. To understand the semantics intended by ECMA authors and

the semantics intended by implementers of JavaScript virtual machines, we looked not only at ES5, but also at the next version of ECMAScript (ES6), at several test suites, and at the behaviour of the major browsers. We spotted bugs in all of these places, obtaining confirmation from ECMA authors and JavaScript implementers that they were indeed bugs.

In ES6, we discovered three bugs, ranging from simple typos to an "informative algorithm" which did not correctly implement the behaviour it was claiming. In test262, we discovered three bugs, the most interesting being a test which contains code that seems to do nothing and whose purpose is unknown to the current maintainers. V8 and Webkit incorrectly implement enumeration in the presence of shadowing, as used by `for−in`. This was discovered by Miller [60] and strengthened by ourselves. All browser implementations give (different) incorrect completion values for `try`...`catch` ...`finally`, also discovered in part by Bargull [60]. In addition, we observed in V8 that dead code after a `try`...`catch`...`finally` block may incorrectly change the value that is returned by the block [60].

Finding bugs in browsers, standards and industry tests was not the main motivation for the work presented here. The bugs mentioned are simply a side effect of attempting to understand ES5 well enough to build JSCert and JSRef. In future, we hope to use JSRef to construct more complete test suites, for potentially detecting unknown bugs in existing browsers.

7. Conclusions and future work

This paper investigates a scientific method for developing a trusted, mechanised specification of a programming language. We focused on JavaScript because the language is and will continue to be used by many people, the standards are quite mature and still evolving, and, for many applications, JavaScript programs need to be secure. There are a few key tools used to support large-scale language formalisations; see related work. We used the Coq theorem prover, because it supports large-scale mechanised proof as well as specification, and it has a well-used mechanism for extracting executable OCaml code.

We have introduced JSCert, a Coq specification of ES5, and JSRef, an executable reference interpreter which is provably correct with respect to JSCert. JSCert closely corresponds to ES5, following its natural modularity. JSRef is tested against industrial test suites. Our design methodology means that (assuming you trust Coq), if you trust either the eyeball connection to ES5 or the industrial tests, you can trust *both* JSCert and JSRef.

It remains to be seen what impact might be possible with the ECMA standards. For ES6, which is nearing the end of its development, we have filed bug reports and are having on-going discussions with some of its authors. For ES7, our work demonstrates that it might be feasible for a Coq specification to evolve at nearly the same time as the English standard. One key question is what might we be able to offer the ECMA standards. It is not our role to make key design decisions for ECMAScript. We do, however, believe that providing an official mechanised specification alongside the English prose standard would add significant value for tool developers and language analysts. Moreover, an executable reference interpreter that provably complies with the Coq specification and passes the ECMA tests can help validate language implementations or optimisations, by providing an oracle to define the expected results of new tests.

We have many future plans. To begin with, we would like to increase as much as possible the trust in our definitions, as explained in § 6, by formalising determinacy and completeness in Coq, by running additional test suites (e.g., Firefox tests), and by designing new tests towards achieving full coverage of ES5. Then, we would also like to extend our formalisation. First, it could be interesting to investigate the formalisation of parsing. Second, we may want to formalise browser-specific behaviours. Third, we would be interested in extending our work to ES6 and ES7. We are, of course, also interested in many of the practical applications of our work, as discussed in § 1.

Acknowledgements. We are grateful to the POPL'14 reviewers for their insightful comments and suggestions. We would like to thank our interns Lorenz Breidenbach for help with testing and debugging, and Benjamin Farinier for his help in setting up the code coverage tool. Bodin and Schmitt are partially supported by the French ANR-10-LABX-07-01 Laboratoire d'excellence CominLabs. Filaretti and Maffeis are supported by EPSRC grant EP/I004246/1. Gardner and Smith are partially supported by EPSRC grant EP/K032089/1 and EPSRC programme grant EP/H008373. Naudziuniene is supported by an EPSRC DTA award.

References

[1] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV: 22nd International Conference on Computer Aided Verification, LNCS 6174*, pages 258–272, 2010.

[2] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *ECOOP'05*, pages 428–452. Springer, 2005.

[3] A. W. Appel. Verified software toolchain - (invited talk). In *ESOP*, pages 1–17, 2011.

[4] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66, 2011.

[5] M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013.

[6] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium*, 2013.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *POPL*, pages 55–66, 2006.

[8] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *J. Autom. Reasoning*, 43(3):263–288, 2009.

[9] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theor. Comput. Sci.*, 336 (2-3):235–284, 2005.

[10] C11. ISO/IEC 9899:2011. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`, 2011.

[11] A. Charguéraud. Pretty-big-step semantics. In *ESOP*, pages 41–60, 2013.

[12] R. Chugh and R. Jhala. Dependent types for JavaScript. *CoRR*, abs/1112.4106, 2011.

[13] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *PLDI'09*, pages 50–62. ACM, 2009.

[14] X. Clerc. Bisect. `http://bisect.x9c.fr/`, 2012.

[15] Compcert Team. Compcert. `http://compcert.inria.fr/`, 2013.

[16] D. Crockford. ADSafe: Making JavaScript safe for advertising. `http://adsafe.org/`, 2007.

[17] S. Drossopoulou and S. Eisenbach. Is the Java type system sound? In *FOOL4*, 1997.

[18] ECMA International. Test262. `http://test262.ecmascript.org/`, 2010.

[19] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.

[20] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *CAV*, pages 501–505, 2004.

[21] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to JavaScript. In *POPL*, pages 371–384, 2013.

[22] Free Software Foundation. C language testsuites: “C-torture" version 4.4.2. http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html, 2010.

[23] D. Fugate. Test262 bug 56. https://bugs.ecmascript.org/show_bug.cgi?id=56, 2011.

[24] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for JavaScript. In *POPL'12*, pages 31–44. ACM, 2012.

[25] Google Inc. The Closure Compiler. https://developers.google.com/closure/compiler/, 2009.

[26] S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, pages 151–168, 2009.

[27] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable JavaScript. In *ISSTA*, pages 177–187, 2011.

[28] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. *ECOOP 2010*, pages 126–150, 2010.

[29] Y. Gurevich. Evolving algebras. In *IFIP Congress (1)*, pages 423–427, 1994.

[30] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF'12*, pages 3–18. IEEE, 2012.

[31] D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *ML*, pages 47–52, 2007.

[32] D. Jang and K. Choe. Points-to analysis for JavaScript. *Proc. of SAC '09*, page 1930, 2009.

[33] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS '09*. Springer, 2009.

[34] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *POPL*, pages 173–184, 2007.

[35] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *CSF'09*. IEEE, 2009.

[36] S. Maffeis, J. Mitchell, and A. Taly. ECMA 262-3 operational semantics. http://jssec.net/semantics/, 2007.

[37] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS'08*, pages 307–325. Springer, 2008.

[38] S. Maffeis, J. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *ESORICS'09*. Springer, 2009.

[39] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Security and Privacy (SP)*, pages 125–140. IEEE, 2010.

[40] G. Melquiond. Floats for Coq 2.1.0. http://flocq.gforge.inria.fr/, 2012.

[41] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0262631814.

[42] Mozilla. Mozilla automated JavaScript tests. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests, 2013.

[43] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[44] M. Norrish. *Formalising C in HOL*. PhD thesis, Computer Lab., University of Cambridge, 1998.

[45] D. S. P. Phung and A. Chudnov. Lightweight self protecting JavaScript. In *ASIACCS 2009*. ACM Press, 2009.

[46] C. Park, H. Lee, and S. Ryu. An empirical study on the rewritability of the with statement in JavaScript. In *FOOL*, 2011.

[47] C. Park, H. Lee, and S. Ryu. SAFE: Formal specification and implementation of a scalable analysis framewrok for ECMAscript. In *FOOL*, 2012.

[48] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *CADE*, pages 202–206, 1999.

[49] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[50] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. *SIGPLAN Not.*, 48(2):1–16, Oct. 2012.

[51] J. G. Politz, B. S. Lerner, H. Q. de la Vallee, T. Nelson, A. Guha, M. Carroll, and S. Krishnamurthi. Mechanized λ_{JS}. http://blog.brownplt.org/2012/06/04/lambdajs-coq.html, 2012.

[52] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do - A large-scale study of the use of eval in JavaScript applications. In *ECOOP'11*, pages 52–78. Springer, 2011.

[53] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[54] T. Schneidereit. Convert some array extras to self-hosted js implementations. https://hg.mozilla.org/mozilla-central/rev/5593cd83590e, 2012.

[55] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[56] D. Syme. Proving Java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[57] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, pages 363–378, 2011.

[58] The JSCert Team. Mozilla Bug 862771, V8 Issue 2529, ES6 Bugs 1442-1444, Test262 Bug 1445, 1450, 1600. https://bugzilla.mozilla.org/show_bug.cgi?id=862771, http://code.google.com/p/v8/issues/detail?id=2529, https://bugs.ecmascript.org/show_bug.cgi?id={1442-1445,1450,1600}, 2013.

[59] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *ESOP'05*, pages 140–140. Springer, 2005.

[60] Various developers. Bugs in ES6, Test262 and browsers. https://bugs.webkit.org/show_bug.cgi?id={38970,104309} http://code.google.com/p/v8/issues/detail?id={705,2446} https://bugzilla.mozilla.org/show_bug.cgi?id=819125, 2010–2012.

[61] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *Proc. POPL*, 2011.