KJS: A Complete Formal Semantics of JavaScript

Daejun Park Andrei Ștefănescu Grigore Roșu

University of Illinois at Urbana-Champaign

{dpark69,stefane1,grosu}@illinois.edu

Abstract

This paper presents KJS, the most complete and throughly tested formal semantics of JavaScript to date. Being executable, KJS has been tested against the ECMAScript 5.1 conformance test suite, and passes all 2,782 core language tests. Among the existing implementations of JavaScript, only Chrome V8's passes all the tests, and no other semantics passes more than 90%. In addition to a reference implementation for JavaScript, KJS also yields a simple coverage metric for a test suite: the set of semantic rules it exercises. Our semantics revealed that the ECMAScript 5.1 conformance test suite fails to cover several semantic rules. Guided by the semantics, we wrote tests to exercise those rules. The new tests revealed bugs both in production JavaScript engines (Chrome V8, Safari WebKit, Firefox SpiderMonkey) and in other semantics. KJS is symbolically executable, thus it can be used for formal analysis and verification of JavaScript programs. We verified non-trivial programs and found a known security vulnerability.

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

General Terms Languages, Standardization, Verification

Keywords JavaScript, mechanized semantics, K framework

1. Introduction

JavaScript is the most popular client-side programming language. Recently, JavaScript has started to be used in not only client-side, but also server-side programming [30], and even beyond web applications [26, 46]. Despite its popularity, JavaScript suffers from several language design inconsistencies [8], which can lead to security vulnerabilities. Nontransparent behaviors are good targets for attackers [17, 41]. To address the utmost importance of security in web applications, there have been several formal analysis studies proposed recently for JavaScript [2, 20, 21, 36, 45], but these address fragments of the language and are not fully validated with a complete, formal JavaScript semantics. Guha *et al.* [22] admit they cannot show their static analysis sound due to the absence of a complete formal semantics of JavaScript.

1.1 Why Yet Another JavaScript Semantics?

A formal semantics should serve as a solid foundation for JavaScript language development, so it must be correct and complete (to be trusted and useful), executable (to yield a reference implementation), and appropriate for program reasoning and verification.

Several efforts to give JavaScript a formal semantics have been made, most notably by Politz *et al.* [36] and Bodin *et al.* [3]. Unfortunately, no existing semantics comes close to having the desired properties mentioned above. First, as shown in Tables 1 and 2, they are incomplete and contain errors. Second, they require different formalizations for different purposes, e.g., an operational/computational semantics for execution and a axiomatic/declarative semantics for deductive reasoning. Having to define two or more different semantics for a real-life language, together with proofs of equivalence, is a huge burden in itself, not to mention that these all need to be maintained as the language evolves. Third, due to the functional nature of their interpreters, these semantics cannot handle the non-determinism of JavaScript well. Finally, their interpreters are not suited for symbolic execution, and thus for developing program reasoning tools. We discuss existing semantics in Section 6.

For these reasons, we developed yet another JavaScript semantics in order to have a single, clean-slate semantics that can be used not only as a reference model for JavaScript, but also to develop formal analysis tools for it. We employed $\mathbb{K}$ [43] (`http://kframework.org`) as the formalism medium. In $\mathbb{K}$, a language semantics is described as a term rewriting system. At no additional cost, $\mathbb{K}$ provides an execution engine, which yields an interpreter for the defined language, as well as a sound and relatively complete deductive verification system based on symbolic execution, which can be used to reason about programs.

1.2 Challenges in Formalizing JavaScript

JavaScript is an unusual language, full of tricky corner cases. Like HTML, JavaScript programs do not easily fail. Seemingly nonsensical programs work by design, i.e., they have properly defined semantics according to the language standard. Completely defining all of the corner cases is highly non-trivial, especially because the language standard, a 250-page English document, contains various ambiguities and unspecified behaviors (which have led to divergence between JavaScript implementations). To handle these difficulties, we decided to make our semantics executable, so that we can test our semantics incrementally. Incremental testing allowed us to eliminate ambiguities one by one and to enhance our understanding of JavaScript's corner cases.

JavaScript is complex. Beside typical difficulties of scripting languages such as dynamic (implicit) casting and the `eval` construct, the latest standard ECMAScript 5.1 introduced new features such as the strict mode and explicit getters/setters. The mixed use of the strict and non-strict modes, and of the data and accessor (getter/setter) properties, makes it inevitable to have complex case analyses in the semantics. For example, Figure 1 describes the "simple" object

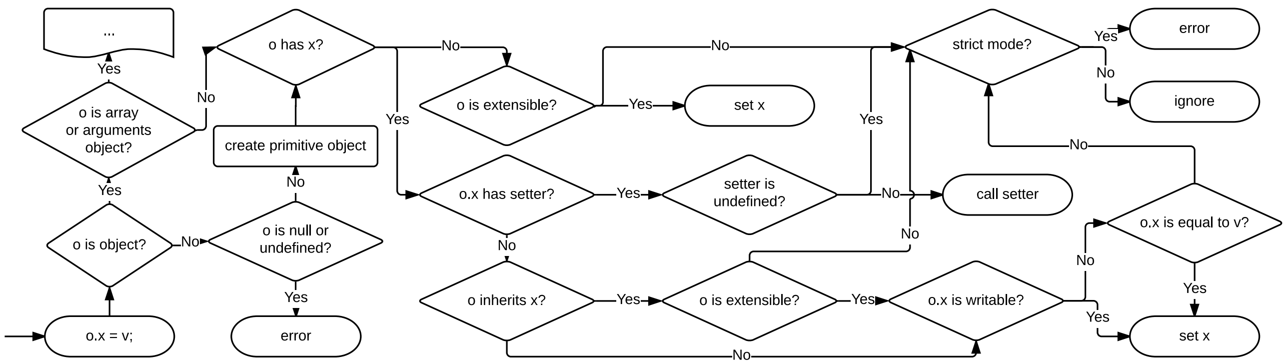

Figure 1. Semantics of Object Property Update: `o.x = v;`

property update `o.x = v` semantics at a high-level, showing how many cases need to be distinguished: `o` is a normal object or not; `o` is extensible or not; `x` is inherited or not; `x` is writable or not; `x` is an accessor property or not; the code is strict or not. To keep better track of all such special cases, we chose to systematically, almost mechanically translate the language standard *as is* into formal semantics (as opposed to defining what we thought JavaScript ought to be doing).

JavaScript is non-deterministic. For example, the `for-in` construct iterates through all the enumerable properties of a given object non-deterministically. The enumeration order is unspecified, implementation-dependent, and may vary for different iterations of a `for-in` loop. Formalizing JavaScript's non-determinism in a semantics that has all the desirable properties listed at the beginning of Section 1.1 is non-trivial. A collection semantics defined as an inductive relation can capture non-determinism easily, but is unsuitable for execution, while a semantics defined as an eval function is suitable for execution but cannot capture non-determinism naturally. Rewriting Logic [31] is a sweet spot, as it can effectively define, execute, and reason about non-deterministic specifications.

1.3 Contribution and Approach

Our main contribution is KJS, a JavaScript formal semantics:

- KJS is the most complete and thoroughly tested formal JavaScript semantics to date, specifically of ECMAScript 5.1, the latest language standard. It has been tested against the ECMAScript conformance test suite, and passed all 2,782 test programs for the core language. Table 1 shows that KJS is far more complete than any other semantics, and even more standards-compliant than production JavaScript engines such as Safari WebKit and Firefox SpiderMonkey.
- KJS closely resembles the language standard (see Figure 2), which facilitates visual inspection, and allows to measure the semantic coverage of a test suite. We found that the ECMAScript 5.1 conformance test suite misses several semantic behaviors described in the language standard. We wrote tests for the uncovered semantics and discovered a number of bugs both in production JavaScript engines and in existing formal semantics. Measuring conformance test suite coverage has been considered infeasible in [5, 6, 13], because JavaScript implementations are highly optimized and do not follow the standard document line by line. KJS thus paves a way for the JavaScript language standard committee to systemically measure the semantic coverage of their conformance test suite.
- KJS has been defined in a style that is suitable also for reasoning about JavaScript programs. We have verified several non-trivial programs and demonstrated how KJS can be used for finding a security vulnerability (Section 5).

1.4 Outline

Section 2 recalls ECMAScript and $\mathbb{K}$. Section 3 discusses KJS, our semantics of JavaScript. Section 4 elaborates on our evaluation of KJS and the development costs, and Section 5 discusses applications of KJS. Section 6 discusses related work and Section 7 concludes.

The complete KJS semantics of JavaScript, as well as all the artifacts discussed in the paper are available for download at:

`https://github.com/kframework/javascript-semantics`

2. Preliminaries

We here briefly explain ECMAScript 5.1, the latest JavaScript standard, and the $\mathbb{K}$ framework, a semantics engineering tool in which we chose to formalize our semantics.

2.1 ECMAScript 5.1

ECMAScript is the official JavaScript language standard. The latest version is ECMAScript 5.1 [11]. Compared to the previous version, ECMAScript 3,[1] ECMAScript 5.1 adds new features for more robust programming such as the strict mode, better integration with the DOM object such as accessor (getter/setter) properties, and new APIs such as JSON. The upcoming version ECMAScript 6 [12], under active development, will add new features such as classes, modules, iterators and collections, and generators and promises (for asynchronous programming).

ECMAScript 5.1 specifies not only the language core but also standard libraries. It consists of 16 chapters and 6 appendices, for a total of 258 pages. Chapters 1-5 give an overview of the language; Chapters 6-7 describe lexing and parsing; Chapter 8 describes runtime types such as string, number, and object; Chapter 9 discusses type conversions; Chapter 10 covers environments and execution contexts; Chapters 11-14 and 16 describe the semantics of language constructs: expressions, statements, functions, programs, and errors; Chapter 15 presents the standard libraries; Appendix A is dedicated to the language grammar and Appendix B compatibility.

ECMAScript 5.1 gives algorithmic descriptions for all language constructs, to precisely specify their behaviors. It also defines various internal semantic functions, called abstract operations, to effectively describe high-level language constructs. For example, Figure 2(a) presents an abstract operation, [[Get]], which takes an object *O* and a property name *P*, and returns *P*'s value of *O*. This

[1] ECMAScript 4 was abandoned due to fundamental disagreements between the committee members [10].

When the [[Get]] internal method of *O* is called with property name *P*, the following steps are taken:

1. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, return **undefined**.
3. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
4. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]].
5. If *getter* is **undefined**, return **undefined**.
6. Return the result calling the [[Call]] internal method of *getter* providing *O* as the **this** value and no arguments.

(a) ECMAScript 5

```
rule Get(O:Oid,P:Var)
  => Let $desc = GetProperty(O,P);  /* Step 1 */
     If $desc = Undefined then {
       Return Undefined;            /* Step 2 */
     };
     If IsDataDescriptor($desc) = true then {
       Return $desc."Value";        /* Step 3 */
     };
     Let $getter = $desc."Get";     /* Step 4 */
     If $getter = Undefined then {
       Return Undefined;            /* Step 5 */
     };
     Return Call($getter,O,Nil);    /* Step 6 */
```

(b) KJS

Figure 2. Correspondence between ECMAScript 5 and KJS semantics

property lookup function precisely describes its behavior by using an informal pseudo-code algorithm. It also interacts with other internal semantic functions such as [[GetProperty]] and IsDataDescriptor.

2.2 The $\mathbb{K}$ Framework

$\mathbb{K}$ [43] (`http://kframework.org`) is a framework for defining language semantics. Given a syntax and a semantics of a language, $\mathbb{K}$ generates a parser, an interpreter, as well as formal analysis tools such as model checkers and deductive program verifiers, at no additional cost. Using the interpreter, one can test their semantics immediately, which significantly increases the efficiency of semantics developments. Furthermore, the formal analysis tools facilitate formal reasoning about the given language semantics. This helps both in terms of applicability of the semantics and in terms of engineering the semantics itself; for example, the state-space exploration capability helps the language designer cover all the non-deterministic behaviors of certain constructs or combinations of them in the language definition.

We briefly describe $\mathbb{K}$ here and refer the reader to [39, 43] for more details. In $\mathbb{K}$, a language syntax is given using conventional Backus-Naur Form (BNF). A language semantics is given as a transition system, specifically a set of reduction rules over configurations. A configuration is an algebraic representation of the program code and state. Intuitively, it is a tuple whose elements (called cells) are labeled and possibly nested. Each cell represents a semantic component such as stores, environments, and threads that are used in defining semantics. A special cell, named k, contains a list of computations to be executed. A computation is essentially a program fragment, while the original program is flattened into a sequence of computations. A rule describes a one-step transition relation between configurations, thus giving semantics to language constructs. Rules are modular; they mention only relevant cells that are needed in each rule. For example, a property lookup semantics can be defined as the following $\mathbb{K}$ rule:

$$\left\langle \frac{O[P]}{V} \cdots \right\rangle_{\mathsf{k}} \ \left\langle \langle O \rangle_{\mathsf{oid}} \ \langle \cdots\ P \mapsto V\ \cdots \rangle_{\mathsf{properties}} \cdots \right\rangle_{\mathsf{obj}}$$

The cells are represented with angle brackets notation. The horizontal line represents a reduction (i.e., a transition relation). A cell with no horizontal line means that it is read but not changed by the rule. The rule above mentions two cells: k, and obj. The k cell contains a list of computations to be executed, and the obj cell represents an object. The obj cell contains several sub-cells: e.g., the oid cell contains the object identifier and the properties cell stores a map from property names to values. This rule is applied when the current computation (top of the k cell) is a property lookup and there exists an obj cell whose oid is matched with *O* and properties contains the property name *P*. This rule resolves the property lookup *O*[*P*] to the property value *V*. The "..." is a structural frame, that is, it matches the portions of a cell that are neither read nor written by the rule.

One of the most appealing aspects of $\mathbb{K}$ is its modularity. It is very rarely the case that one needs to touch existing rules in order to add a new feature to the language. This is achieved by structuring the configuration as nested cells and by requiring the language designer to mention only the cells that are needed in each rule, and only the needed portions of those cells. For example, the above rule only refers to the k and obj cells, while the entire configuration contains many more cells (Figure 3). This modularity makes for compact and human readable semantics, and also helps with the overall effectiveness of the semantics development. For example, even if new cells are later added to configuration, to support new features, the above rule does not change.

Another appealing aspect of $\mathbb{K}$ is its inherent support for non-determinism. As $\mathbb{K}$ is based on rewriting logic [31], one can easily define, execute, and reason about non-deterministic specifications in $\mathbb{K}$. For example, a simplified `for-in` loop semantics[2] can be defined as the following $\mathbb{K}$ rules:

$$\frac{\texttt{for}\ I\ \texttt{in}\ E\ Es\ \texttt{\{}\ S\ \texttt{\}}}{I\ \texttt{=}\ E\ \texttt{;}\ S\ \texttt{;}\ \texttt{for}\ I\ \texttt{in}\ Es\ \texttt{\{}\ S\ \texttt{\}}} \qquad \frac{\texttt{for}\ I\ \texttt{in}\ \cdot_{Set}\ \texttt{\{}\ S\ \texttt{\}}}{\cdot}$$

Suppose that `for-in` loop non-deterministically iterates through the given elements. In $\mathbb{K}$, such non-determinism can be easily described by representing the elements as a set and using set matching, which gives us the desired set-theoretical 'choice' operation. In the above semantics, '$E\ Es$' represents the set of elements to be iterated through, where E refers to an arbitrary element of the set, and Es the remaining elements. The rule in the left-hand side says that it chooses an arbitrary element E, runs the loop body S with the element, and proceeds to the next iteration with the remaining elements Es. The rule in the right-hand side specifies the termination condition of the loop. This way, one can easily describe and execute non-deterministic semantics. Furthermore, using $\mathbb{K}$'s 'search'-mode execution, one can explore all possible execution traces, in this case all possible iteration orders.

3. KJS: Formal Semantics of JavaScript in $\mathbb{K}$

KJS faithfully describes ECMAScript 5.1 in $\mathbb{K}$. It defines the core language semantics, and also several standard libraries. KJS is systematically derived from, and has a close correspondence with, the language standard.

[2] The `for-in` construct of JavaScript has a more complex semantics; by these sample rules, we here only mean to illustrate $\mathbb{K}$'s capabilities.

$$\left\langle \begin{array}{c} \langle K \rangle_{\mathsf{k}} \ \langle \langle \langle ID_{\mathsf{obj}} \rangle_{\mathsf{oid}} \ \langle Var \mapsto Val_{Prop} \rangle_{\mathsf{properties}} \ \langle Var \mapsto Val_{Prop} \rangle_{\mathsf{internalProperties}} \rangle_{\mathsf{obj}*} \rangle_{\mathsf{objs}} \\ \langle \langle \langle ID_{\mathsf{env}} \rangle_{\mathsf{eid}} \ \langle ID_{\mathsf{env}} \rangle_{\mathsf{outer}} \ \langle Bool \rangle_{\mathsf{strict}} \ \langle Var \mapsto Val_{Env} \rangle_{\mathsf{declEnvRec?}} \ \langle \langle ID_{\mathsf{obj}} \rangle_{\mathsf{bindingObj}} \ \langle Bool \rangle_{\mathsf{provideThis}} \rangle_{\mathsf{objEnvRec?}} \rangle_{\mathsf{env}*} \rangle_{\mathsf{envs}} \\ \langle \langle \langle List_{\mathsf{running}} \rangle_{\mathsf{activeStack}} \ \langle \langle ID_{\mathsf{env}} \rangle_{\mathsf{lexicalEnv}} \ \langle Val \rangle_{\mathsf{thisBinding}} \rangle_{\mathsf{running}} \rangle_{\mathsf{ctx}} \ \langle List_{\mathsf{ctrl}} \rangle_{\mathsf{excStack}} \ \langle List \rangle_{\mathsf{pseudoStack}} \rangle_{\mathsf{ctrl}} \end{array} \right\rangle_{\mathsf{T}}$$

Figure 3. Configuration

3.1 Program Configuration

Figure 3 shows the KJS configuration, or state, which holds objects, environments, and the execution context.

Objects An object is a map from property names to values with attributes. Each object is connected with another object via a [[Prototype]] link. An object inherits other objects along with the prototype chain. In the configuration, an object is represented by an obj cell. The "*" appearing next to the obj cell name in the configuration tells $\mathbb{K}$ that zero, one or more cells with that name can occur at that position in the configuration. An obj is identified by oid, and contains two maps: properties and internalProperties. The properties stores user-level properties, while internalProperties is for internal use only.

Environments An environment is a map from variables to values. Each environment is created when the program control enters a new scope, and is connected with its outer scope environment. The environment remains even after the program control exits from the scope. In the configuration, an environment is represented by the env cell. An env is identified by eid and contains an outer link and a declEnvRec map. In case of the global scope and the `with` block, however, the env has an objEnvRec map instead of declEnvRec. A "?" appearing next to a cell name tells $\mathbb{K}$ that zero or one cells with that name can appear in the configuration at that position.

Execution context An execution context consists of an environment and the `this` value. A new execution context is created whenever the program control enters a function, and discarded when the function returns. In the configuration, the current execution context is represented by the running cell. When a new execution context is created, the current one is pushed into the activeStack cell (structured as a list).

3.2 Semantics Description Language

KJS essentially defines two languages: the JavaScript language and its semantics description language. ECMAScript 5.1 presents semantic behaviors in pseudo-code; see Figure 2. To faithfully describe them, we first formally define this pseudo-code language, which is a minimal imperative language with let-bindings and branches. It does not have loops, since iteration can be achieved by recursively applying rules.

3.3 Semantics of Language Constructs

We define the semantics of each language construct by systematically translating its informal algorithmic description in the language standard into formal pseudo-code as defined in Section 3.2. Figure 2 shows an example of the translation. Each step of (a) is translated to its corresponding pseudo-code statement of (b). For example, step 1

> Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.

is translated to the formal definition of (b):

```
Let $desc = GetProperty(O,P);
```

This approach not only contributes to the faithfulness of our semantics, but also expedites the semantics development.

We only describe a few relevant or interesting constructs.

```
var base = Object.create(Object.prototype, {
  y : {value:0, enumerable:false,configurable:true} });
var derived = Object.create(base, {
  x : {value:1, enumerable:true, configurable:true},
  y : {value:2, enumerable:true, configurable:true} });
var i = 0;
for (var k in derived) {
  if (i === 0) delete derived.y;
  console.log(k + ":" + derived[k] + ";"); ++i; }
```

Figure 4. Undefined `for-in` program: Safari WebKit and Chrome V8 output `x:1; y:0;`, while Firefox SpiderMonkey outputs `x:1;`.

For-in construct The `for-in` construct, which iterates through all the enumerable properties of a given object, is non-deterministic. The enumeration order of the properties is not specified, but implementation-dependent. A loop may have a different iteration order even in the same program. In order to correctly specify this non-determinism, our semantics employs the set-theoretical 'choice' operation to select each property non-deterministically. $\mathbb{K}$ provides a 'search'-mode execution feature which explores all possible execution traces, in this case all possible enumeration orders.

Furthermore, certain semantic behaviors are under-specified in the language standard [44]. A property is enumerable when its `enumerable` attribute is true. The iterated properties include not only the object's own properties, but also the inherited ones. An inherited property, however, is excluded when it is shadowed. Also, if a property is deleted during the iteration before it is visited, the property is skipped. But what if a property is shadowed and the property causing the shadowing is deleted before its visit? Is the original property supposed to be visited? The language standard leaves this behavior unspecified, without even stating if it is implementation-dependent or not. The consequence is that different JavaScript implementations have different behaviors in this situation. Figure 4 shows a `for-in` loop on the `derived` object, which inherits the `base` object shadowing the property `y`. In the loop, the shadowing property `derived.y` is deleted before it is visited;[3] the shadowed property `base.y` now becomes visible and can be considered for enumeration in the next iteration. For this program, Safari WebKit and Chrome V8 output `x:1;y:0;` since they decided to visit `base.y`, while Firefox SpiderMonkey outputs `x:1;` since it does not visit `base.y` whose `enumerable` attribute is false.

KJS makes these unspecified behaviors explicit: it reports an 'unspecified' error when a `for-in` loop encounters the unspecified situation in Figure 4. This feature needs to be defined in order to have a complete semantics, and can be used to check the portability of JavaScript programs. Section 5.1 discuses this in more detail.

Exceptions While user-level exceptions (raised with `throw`) are well described in ECMAScript 5.1, internal exceptions (e.g., ReferenceError) are not. The described exception propagation mechanism only applies to the user-level exceptions. To define both user-level and internal exceptions in a uniform way, KJS employs an exception handling mechanism that is commonly used by many programming language semantics. Figure 5 shows the essential rules. The rule TRY starts to execute *S*, pushing the current execution context in the excStack cell. If an exception occurs, the rule

[3] Suppose an iteration order where `x` is visited first.

RULE TRY

$$\left\langle \frac{\texttt{try}\ S\ \texttt{catch}\ (X)\ S'}{S \curvearrowright\ \texttt{endtry}} \curvearrowright K \right\rangle_{\mathsf{k}} \left\langle \left\langle \frac{\cdot}{(X, S', K, C)}\ \_ \right\rangle_{\mathsf{excStack}}\ C \right\rangle_{\mathsf{ctrl}}$$

RULE THROW

$$\left\langle \frac{\texttt{throw}\ V_e \curvearrowright \_}{\texttt{let}\ X = V_e\ \texttt{in}\ S' \curvearrowright K} \right\rangle_{\mathsf{k}} \left\langle \left\langle \frac{(X, S', K, C)}{\cdot}\ \_ \right\rangle_{\mathsf{excStack}}\ \frac{\_}{C} \right\rangle_{\mathsf{ctrl}}$$

RULE END-TRY

$$\left\langle \frac{V_S \curvearrowright\ \texttt{endtry}}{V_S} \cdots \right\rangle_{\mathsf{k}} \left\langle \left\langle \frac{(\_, \_, \_, \_)}{\cdot}\ \_ \right\rangle_{\mathsf{excStack}}\ \_ \right\rangle_{\mathsf{ctrl}}$$

Figure 5. Exception semantics

THROW restores the saved context *C* and executes the catch block *S'*. If no exception occurs, the rule END-TRY discards the saved execution context and proceeds to the next computation.

Switch statement JavaScript's `switch` has a surprising fall-through semantics: it does *not* fall through at `default`. For example, the following switch statement contains two regular `case`s and one `default` with no `break` statement.

```
function foo(n) {
  switch(n) { case 1:  console.log("case 1;");
              default: console.log("default;");
              case 2:  console.log("case 2;"); } }
```

`foo(1)` outputs "`case 1; default; case 2;`", and `foo(2)` outputs "`case 2;`". However, `foo(0)` outputs only "`default;`", because JavaScript does not allow the fall-through at `default`. This behavior is unusual compared to other programming languages in which `foo(0)` would output "`default; case 2;`".

Strict mode code JavaScript has a strict mode execution feature, which also contains tricky corner cases. It was newly introduced in the latest language standard, ECMAScript 5.1, as a workaround for several design mistakes (e.g., the `this` resolution). A strict mode execution is only applied to a strict mode code, indicated by a '`use strict`' directive. For example, the following is a strict mode code; its execution throws a `ReferenceError` exception since the undeclared variable '`x`' is not allowed to be used in strict mode:

```
'use strict';
eval('x = 1;'); // throws a ReferenceError
```

However, the following program, which appears to be equivalent to the above, does not report the exception:

```
'use strict';
var myeval = eval;
myeval('x = 1;'); // no ReferenceError
```

The reason is that an eval code inherits the strict mode only when it appears in a direct call to `eval`. In the first program, '`x = 1;`' is evaluated in strict mode because `eval` is called directly on it. However, in the second program, '`x = 1;`' is evaluated in *non*-strict mode because `eval` is called indirectly, and `x` is assigned `1` in the global scope.

Function and variable declarations are evaluated before other statements, with function declarations evaluated before variable declarations. In combination with shadowing lack of block scoping, unexpected results can occur. The following seemingly equivalent functions return different values, `2` and `1`:

```
function f1() { function f() { return 1; }
                function f() { return 2; }
                return f(); } // 2
function f2() { var f = function () { return 1; };
                function f() { return 2; }
                return f(); } // 1
```

This is because the first line of `f2` is a variable and not a function declaration, so is evaluated after the function declaration in the next line, overwriting it.

Arguments objects When a function is called, an `arguments` object is created holding the function's arguments values. Modifying the `arguments` object is allowed, but it has different semantics depending on whether we are in a strict mode or not. If non-strict, `arguments` is aliased with the formal parameters; if strict, `arguments` has its own properties, not affecting the formal parameters. For example, below `f(0)` returns `1` while `g(0)` returns `0`:

```
function f(x) { arguments[0] = 1; return x; }
function g(x) { "use strict";
                arguments[0] = 1; return x; }
```

Eval function The definition of the `eval` function is straightforward in KJS: it parses the argument and then evaluates it in the eval execution mode. Parsing is handled by the '`#parse`' primitive of the $\mathbb{K}$ framework, which uses a parser automatically generated from the given syntax declarations.

3.4 Standard Libraries

Although KJS aims at defining the semantics of the core JavaScript language, we have also given semantics to some essential standard built-in objects. For example, we completely defined the `Object`, `Function`, `Boolean`, and `Error` objects, because they expose internals of the language semantics. Also, we partially defined the `Array`, `String`, `Number` and `Global` objects; specifically, all their constructors and only a group of internal methods, such as `Array`'s [[DefineOwnProperty]] and `String`'s [[GetOwnProperty]]. These internal methods are essential because they determine the fundamental behavior of their corresponding objects, so that the rest of these objects' behaviors can be defined entirely in JavaScript invoking these internal methods, as explained shortly. Finally, we have not given semantics to the `Math`, `Date`, `RegExp`, and `JSON` objects, because these are orthogonal to the semantic approach and can be implemented in plain JavaScript [7].

Figure 6 shows by means of an example our simple approach to give semantics to built-in objects based on the already defined internal methods: JavaScript itself. Each step of (a) is translated to the corresponding JavaScript code of (b); Steps 1 and 3 employ the internal methods `@IsObject` and `@SetInternalProperty`.[4] KJS defines dozens of such internal methods that are difficult or impossible to define in JavaScript. Based on these, the built-in objects can be completely defined in JavaScript, concisely and independently from the employed semantic formalism.

4. Evaluation

We evaluate KJS w.r.t. completeness and development cost.

4.1 Completeness

To evaluate the completeness of KJS and to measure the progress during its development, like the authors of previous JavaScript semantics [3, 37], we tested our semantics against the official ECMAScript 5.1 language conformance test suite, test262 [14]. The test262 consists of 11,578 test programs which are classified according to each of the chapters of ECMAScript 5.1. Chapters 1-5 have no tests; Chapters 6-7 have 716 tests for parsing; Chapters 8-14 have 2,782 tests for the language core; and Chapter 15 and Annex B have 8,080 tests for standard libraries. Like previous JavaScript

[4] We employ a different namespace for the internal semantic functions, using names starting with '`@`' which cannot appear as program variables (since '`@`' is not an *IdentifierStart* character [11]). Thus we can safely introduce internal functions without polluting the global object.

15.2.3.5 Object.create (O [, Properties])

The **create** function creates a new object with a specified prototype. When the **create** function is called, the following steps are taken:

1. If Type(*O*) is not Object or Null throw a **TypeError** exception.
2. Let *obj* be the result of creating a new object as if by the expression new Object() where Object is the standard built-in constructor with that name
3. Set the [[Prototype]] internal property of *obj* to *O*.
4. If the argument *Properties* is present and not **undefined**, add own properties to *obj* as if by calling the standard built-in function `Object.defineProperties` with arguments *obj* and *Properties*.
5. Return *obj*.

(a) ECMAScript 5

```
Object.create = function (O, Properties) {
  // Step 1
  if (!(@IsObject(O) || O === null))
    throw TypeError("Invalid arguments");
  // Step 2
  var obj = new Object();
  // Step 3
  @SetInternalProperty(obj, "Prototype", O);
  // Step 4
  if (Properties !== undefined)
    Object.defineProperties(obj, Properties);
  // Step 5
  return obj;
};
```

(b) Our semantics, independent of semantic framework

Figure 6. Self-hosted standard built-in objects semantics

Formal Semantics	Passed	Failed	% passed
KJS	2,782	0	100.0%
Politz *et al.* [37][6]	2,470	345	87.7%
Bodin *et al.* [3]	1,796	986	64.6%
JavaScript Engines	**Passed**	**Failed**	**% passed**
Chrome 35.0 (V8 3.25.28)	2,782	0	100.0%
Firefox 30.0 (SpiderMonkey 30)	2,780	2	99.9%
Safari 7.0.4 (WebKit 537.76.4)	2,780	2	99.9%

Table 1. Comparison of formal semantics and product engines tested against the ECMAScript conformance test suite

semantics efforts, to keep the project manageable we targeted only the 2,782 tests corresponding to the core language. As explained in Section 3.4, we have also defined some essential standard built-in objects and internal methods, so that the remaining methods can be implemented in plain JavaScript. However, providing JavaScript code for the hundreds of standard library methods is beyond the scope of this paper.

Table 1 shows that KJS is the most complete JavaScript semantics to date, passing all of the 2,782 ECMAScript 5.1 core tests. It is even more standards-compliant than production JavaScript engines such as Safari WebKit and Firefox SpiderMonkey. While the 2,782 tests are supposed to test the language core, several tests use library calls, e.g. to trigonometric functions. To test such programs modulo the unsupported libraries, we used a feature of $\mathbb{K}$ allowing to employ an external library implementation; specifically, we used the Node.js implementation of `Math.sin`, `Number.toFixed`, and `Number.toString`.[5] Further, to overcome some current parsing limitations of $\mathbb{K}$ (acknowledged by $\mathbb{K}$'s developers and scheduled for fixing), we pre-process the input JavaScript program using the SAFE framework [28] for automatic semicolon insertion and the `sed` utility for translating unicode characters.

Currently, the $\mathbb{K}$ interpreter takes an hour to execute all of 2,782 test programs in a machine with Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB 1333MHz. $\mathbb{K}$ development team, however, is currently working on an OCaml backend to compile $\mathbb{K}$ definitions to OCaml programs for faster execution. With that, the execution time is expected to drop from an hour to minutes.

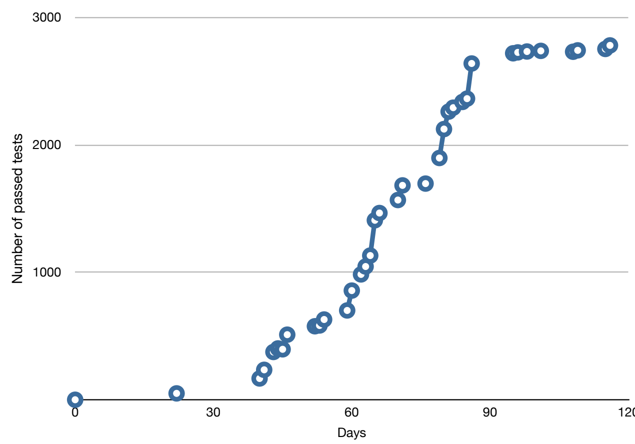

Figure 7. KJS semantics development progress

4.2 Development Cost

The development of KJS took only four months by a first year PhD student, with no prior knowledge of JavaScript or of the $\mathbb{K}$ semantic framework. We believe that this was possible thanks to the following: (1) $\mathbb{K}$'s executability, allowing us to test and fix the semantics immediately as inconsistencies were detected; (2) Formalizing the pseudo-code used in the language standard, which allowed us to easily and systematically formalize the informal semantics; (3) $\mathbb{K}$'s modularity, allowing us to change the structure of the program configuration (e.g., to add new features to the language) without having to change the existing rules (e.g., to add exceptions we had to add new cells to the configuration and three independent rules, but no other rules had to be touched—Figure 5).

A side objective of our effort was to demonstrate that the programming language semantics field has matured enough that language designers should consider defining a complete formal semantics to their language as part of the (long) standardization process. It is no longer true that defining a formal semantics to a language takes too long to be worthwhile. To bring more evidence in this direction, we measured and logged the KJS development progress rigorously. Figure 7 shows how many tests passed each day during the project timeframe. In the first month we developed the semantic foundations such as syntax, program configuration, prototype chains, environments, and execution contexts. In the next two months, we defined individual language constructs. Due to the modularity of the employed framework, during this period the number of passed tests linearly increased as each language

[5] Only a dozen of tests depend on this, which is not a significant number.

[6] Note that S5 was tested against the previous version of the ECMAScript 5 test suite, and the total number of tests is slightly bigger than the latest one. Also, S5 reported test results for standard libraries, which is not presented here since we focus on the language core.

construct was defined. In the last month we finished our semantics by addressing specific details and corner cases revealed by failed tests, until all of them eventually passed.

5. Applications

Here we list a few applications of our semantics, mentioning that these were driven by our own interests and that they are by no means exhaustive. The message we want to convey is that a formal semantics can be useful well beyond just giving a reference model/implementation for the defined language.

5.1 Checking Portability

As seen in Section 3.3, ECMAScript 5.1 contains unspecified behaviors, e.g., the `for-in` loop. Since unspecified behaviors are implementation-dependent, JavaScript programs may not be portable, working differently with different JavaScript engines in different web browsers. Detecting unspecified behaviors in JavaScript programs is not trivial. Simply running the program in different JavaScript engines is not sufficient: even if they all agree on some unspecified behavior now, this may change in future releases.

KJS can be trivially used to detect unspecified behaviors of JavaScript programs, as it 'gets stuck' when no rule matches (i.e., no semantics exist). For the unspecified behavior in Figure 4, e.g., KJS gets stuck when the loop iteration encounters `y`, after the output `x:1;`. Besides unspecified behaviors, we also need to check for non-deterministic behaviors; e.g., to ensure that the iteration order of a `for-in` loop is irrelevant. $\mathbb{K}$ provides a 'search'-mode execution feature which explores all feasible execution traces.[7]

5.2 Finding Bugs and Improving the Test Suite

The ECMAScript standards committee has made an impressive effort to provide a conformance test suite that systematically ensures that all the features of ECMAScript 5.1 and their subtle interactions are covered, so that JavaScript engines converge on a language standard. However, the semantic coverage of the test suite has not been well-studied, and indeed, some behaviors have escaped untested [5]. Using KJS, we found that despite the large number of tests, certain semantic behaviors are still not tested. For example, surprisingly, there is no test for the peculiar fall-though semantics of the `default` case for `switch` (Section 3.3). Writing tests to cover the untested behaviors, we found bugs in all production JavaScript engines and in previous semantics.

How can we measure the semantic coverage of a conformance test suite? One possibility is to run it through several JavaScript implementations using code coverage tools, and project the result back to ECMAScript 5.1. However, this is impractical, as it is not viable to match optimized implementation code to corresponding ECMAScript 5.1 pseudo-code and filter out implementation-specific code [6].

Due to its one-to-one correspondence with ECMAScript 5.1, KJS provides a direct semantic coverage measure for a test suite. This way we found that there are exactly 17 semantic rules in the core semantics which are not covered by the test suite, each corresponding to the language standard as shown in Table 2. We succeeded to manually write test programs that hit 11 out of 17 behaviors, thus improving the overall quality of the conformance test suite. It took two days to manually write (or show infeasibility of) the tests for the 17 cases. Finding tests for the semantics is essentially the same as finding tests for conventional programs. For each uncovered semantic rule, we examine a kind of a path condition that leads to the rule, and find a solution (i.e., a test program) that satisfies the path condition. Automatic test case generation techniques may be used to mechanize this process, but in this paper we have done all the work manually.

```
function mkSend(rawSend) {
  var whiteList = { "http://www.trust.com": true,
                    "http://www.good.com" : true };
  function newSend(target, msg) {
    if (whiteList[target]) rawSend(target,msg);
    else console.error("Rejected."); }
  return newSend; }

var send = mkSend(function (target, msg) {
  console.info("Sent " + msg + " to " + target);});
```

Figure 8. Secure Message Sending

As seen in Table 2, the 11 new tests uncovered bugs in both production JavaScript engines and in existing semantics. Moreover, the remaining 6 semantic behaviors are infeasible, that is, they represent flaws in the language standard itself. These bugs were reported, confirmed, and fixed [34].[8] Below we discuss two out of the 11 new tests, and one of the 6 infeasible behaviors.

Step 5.e.iv of Section 10.5 in the language standard describes how to handle duplicate global function declarations and is not covered by the test suite. The following program

```
Object.defineProperty(this, "f", {
  "value"    : 0,     "enumerable"   : false,
  "writable" : false, "configurable" : false });
eval(" function f() { return 0; } "); // TypeError
```

is supposed to raise a `TypeError` exception according to the standard, since the function `f` is declared while there already exists another `f` whose `writable`, `enumerable`, and `configurable` attributes are all false. Safari WebKit wrongly ignores the duplicate function declaration, disobeying the standard; Chrome V8 and Firefox SpiderMonkey behave correctly.

Step 4 of Section 10.2.1.1.3 in the standard describes a case of updating an immutable variable which is not covered by the test suite either. In the following program

```
"use strict";
var f = function g() { g = 0; /*TypeError*/ }; f();
```

`g` is immutable, but the body attempts to update it. According to the standard, a TypeError exception must be raised. However, only Firefox SpiderMonkey conforms, while Chrome V8[9] and Safari WebKit do not, wrongly ignoring the update statement.

For an example of infeasible semantic behavior, consider Section 10.2.1.1.4 GetBindingValue(N,S) in the standard which describes the environment lookup semantics for a given variable N, and its Step 3.a which discusses the case where N has an uninitialized immutable binding. However, this case is infeasible. There are only two situations where immutable bindings can occur, namely in the `arguments` object in a strict mode function and in the name of a recursive function expression[10] in its function body's environment. But according to the standard, in both cases the bindings are initialized right after creation, thus there is no way to have uninitialized immutable bindings.

We also ran the additional 11 tests on the existing semantics, and discovered a number of bugs, as shown in Table 2.

[7] It is also possible to check confluence of unspecified behaviors (i.e., ensuring that unspecified behaviors are irrelevant) using the 'search'-mode execution, but developing such a sophisticated portability checker is an orthogonal problem, which we leave as future work.

[8] It turned out that two of them had already been reported [1, 33].

[9] Fixed in Chrome 41.0 (V8 4.1.0).

[10] Function 'expression' and *not* 'declaration', because in the latter the function name is declared in a global environment and is mutable.

Page #	Section # - Step #	KJS	Po	Bo	CR	FF	SF
p35	8.7.1 GetValue (V) - [[Get]], Step 6	○	×	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 2.a	○	○	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 2.b	○	⊗	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 4.a	-	-	-	-	-	-
p36	8.7.2 PutValue (V, W) - [[Put]], Step 4.b	-	-	-	-	-	-
p36	8.7.2 PutValue (V, W) - [[Put]], Step 6.a & 6.b	○	○	⊗	○	○	○
p36	8.7.2 PutValue (V, W) - [[Put]], Step 7.a	○	×	○	○	×	○
p40	8.12.4 [[CanPut]] (P) - Step 8.a	○	⊗	⊗	○	○	○
p53	10.2.1.1.3 SetMutableBinding (N,V,S) - Step 4	○	×	○	×	○	×
p53	10.2.1.1.4 GetBindingValue(N,S) - Step 3.a	-	-	-	-	-	-
p53	10.2.1.1.5 DeleteBinding (N) - Step 2	-	-	-	-	-	-
p54	10.2.1.1.5 DeleteBinding (N) - Step 4 & 5	○	⊗	○	○	○	○
p55	10.2.1.2.4 GetBindingValue(N,S) - Step 4.a	-	-	-	-	-	-
p59	10.5 Declaration Binding Instantiation - Step 5.e.iii.1	○	○	○	○	○	○
p59	10.5 Declaration Binding Instantiation - Step 5.e.iv, 1st condition is true	○	⊗	⊗	○	○	×
p59	10.5 Declaration Binding Instantiation - Step 5.e.iv, 2nd condition is true	○	⊗	⊗	○	○	×
p62	10.6 Arguments Object - [[DefineOwnProperty]], Step 4.a, else-branch	-	-	-	-	-	-

○: Passed   ×: Failed   ⊗: Not applicable (failed due to unsupported semantics)   -: Infeasible semantic behaviors

Po: Politz *et al.* [37]   *Bo*: Bodin *et al.* [3]   *CR*: Chrome 38.0 (V8 3.28.71)   *FF*: Firefox 32.0 (SpiderMonkey 32)   *SF*: Safari 7.0.4 (WebKit 537.76.4)

Table 2. Behaviors *not* covered by the ECMAScript 5.1 conformance test suite. Manually written tests exercising these uncovered behaviors revealed bugs in production JavaScript engines and in previous JavaScript semantics.

5.3 Symbolic Execution

Here and in Section 5.4 we illustrate how to derive JavaScript program reasoning tools from generic tools offered by the employed semantic framework. $\mathbb{K}$ allows for terms it reduces to be symbolic, that is, to contain mathematical variables and constraints on them. As semantic rules are applied, constraints are accumulated and solved using Z3 [9] (which is incorporated in $\mathbb{K}$). In this section we show how this capability can be used to find a known security vulnerability, and in the next section how it can be lifted into a fully-fledged JavaScript program verifier.

Consider the program in Figure 8, introduced by Fournet *et al.* [17], which contains a secure message sending function. The `send` method sends messages only to addresses in the white list. For example, the following should be rejected:

```
send("http://www.evil.com","msg"); // Rejected
```

Suspecting a global object poisoning attack [42], we construct a configuration adding a symbolic property P with symbolic value V in the `Object.prototype` object, equivalent to executing `Object.prototype[`P`]` = V. Then we execute the `send` request above using $\mathbb{K}$'s search mode, looking for a state where the message was sent. The symbolic search execution then returns the constraint

$$P = \texttt{"http://www.evil.com"} \;\wedge\; (V = \texttt{true} \vee V \textit{ is a non-empty string} \vee V \textit{ is a non-zero number} \vee V \textit{ is an object})$$

modeling the instances of the suspected attack model; e.g.,

```
Object.prototype["http://www.evil.com"] = true;
```

executed before the malicious `send` call above allows the message to be sent to the malicious address. That is because `Object.prototype` is inherited by all objects, so the `if`-condition `whiteList["http://www.evil.com"]` returns true even if the `whiteList` does not include the evil address. This problem can be fixed by creating an isolated object for `whiteList` using `Object.create(null)`:

```
var whiteList = Object.create(null);
whiteList["http://www.trust.com"] = true;
whiteList["http://www.good.com" ] = true;
```

Function	Size (LOC)	Time (s)
List reverse	13	8
List append	12	13
BST find	12	7
BST insert	23	12
BST delete	34	17
AVL find	11	7
AVL insert	87	109
AVL delete	106	174

Table 3. Verification Result

5.4 Program Verification

$\mathbb{K}$ offers support for program verification based on rule-based semantics, at no additional cost (with no need to define another semantics) [40]. Program properties are specified as reachability rules. $\mathbb{K}$ uses a sound and relatively complete proof system for deriving such rules from the operational semantics rules, which amounts to:

1. Performing symbolic execution of code without repetitive behavior using the semantics rules; and
2. Reasoning about repetitive constructs (loops, recursion).

Like in Hoare logic, all the repetitive constructs need to be annotated with specifications. The verification is automatic: the user only provides the specifications. The specifications are given as reachability rules between symbolic configurations with constraints. We keep the rules compact by:

1. Using the $\mathbb{K}$ notations and conventions (as described in Section 2.2) to describe the symbolic configurations; and
2. Computing the static part of the symbolic configurations (e.g. the builtin-in objects) using the semantics.

For all practical purposes, the standard pre-/post-conditions can be automatically desugared into reachability rules, although we have not implemented it yet.

To test the viability of using the generic reachability verification infrastructure with the JavaScript semantics, we verified a few

```
function insert(v, t) {
  if (t === null) return make_node(v);
  if (v < t.value) t.left = insert(v, t.left);
  else if (v > t.value) t.right = insert(v, t.right);
  else return t;
  update_height(t); return balance(t); }

function balance(t) {
  if (height(t.left) - height(t.right) > 1) {
    if (height(t.left.left) < height(t.left.right))
      t.left = left_rotate(t.left);
    t = right_rotate(t); }
  else if (height(t.left) - height(t.right) < -1) {
    if (height(t.right.left) > height(t.right.right))
      t.right = right_rotate(t.right);
    t = left_rotate(t); }
  return t; }

function left_rotate(x) {
  var y = x.right; x.right = y.left; y.left = x;
  update_height(x); update_height(y); return y; }

function right_rotate(x) { ... }
```

Figure 9. AVL Tree Insertion

JavaScript programs implementing data-structures operations. Table 3 summarizes our experiments. For each function we verified the full functional correctness. Due to space limitations, we discuss only the AVL insert function (the code is shown in Figure 9). The specification of AVL insert in a form of a pre-/post-condition that would desugar into our current reachability rule (shown in the supplementary material [35]) is:

```
function insert(v, t)
//@requires tree(t)(T) /\ avl(T)
          /\ tree_height(T) < INT_MAX
//@ensures tree(t)(T') /\ avl(T')
         /\ tree_keys(T') == { v } U tree_keys(T)
         /\ | tree_height(T') - tree_height(T) | <= 1
```

The precondition requires that the function is passed an AVL tree t, and that the height h of t is small enough such that both h and $h+1$ can be represented on a float-point number without precision loss. The postcondition ensures that the function returns an AVL tree t', that the keys of t' are the keys of t plus the inserted key, and that the height h' of t' is either h or $h+1$. The bound on h is specific to JavaScript, because JavaScript only provides floating-point arithmetic. The AVL, keys, and height abstractions are defined recursively in a standard way.

The overall verification times in Table 3 are quite acceptable, considering that our program verifier is obtained for free from KJS and that, at the best of our knowledge, there is no other program verifier for JavaScript that can verify such complex programs to compare with ours. Also, our times are only twice slower on average than those in [40] for similar properties but for a toy C-like language. The times for AVL insert and delete are large due to the fact that the helper functions (`balance`, `left_rotate`) are not given specifications, instead they are called using their operational semantics, which leads to a larger number of paths to analyze. The effort to verify these examples took approximately one man-week. Most of the work went into finding the JavaScript specific part of the specifications (like the bound on the height in the AVL example). We believe that our preliminary evaluation shows a realistic potential of using the KJS semantics for JavaScript program verification.

6. Related Work

There is a large body of literature on real language semantics. Due to space, we only discuss efforts that directly influenced us: JavaScript semantics and other large semantics in $\mathbb{K}$.

6.1 Other JavaScript Semantics

We only consider JavaScript semantics attempting to define the full language, not a subset, i.e., ones which like ours aim at establishing a solid foundation for formal JavaScript tools.

Herman and Flanagan (2007) [25] gave an executable semantics of ECMAScript 4. As language standard committee members (Ecma TC39-ECMAScript), their objective was to specify a definitional interpreter of the language. They used ML as a specification language, since it is executable, more precise than English prose, and more easily understandable than mathematical notation. They separately defined the standard libraries in JavaScript itself, which is also what we did. Their semantics, however, is based on ECMAScript 4 which was abandoned, never approved as a standard. Furthermore, unlike ours, their semantics does not facilitate formal reasoning.

Maffeis et al. (2008) [29] defined a small-step semantics of ECMAScript 3 and proved some basic properties. Their semantics is based on the older ECMAScript 3, and does not cover the modern JavaScript features such as the strict mode. Also, it is not executable, and cannot be validated against conformance test suites.

Guha et al. (2010) [23] and Politz et al. (2012) [37] presented a reduced semantics of JavaScript, based on ECMAScript 3 and 5, respectively. They defined a core language, λ_{JS}, and a translation from JavaScript to λ_{JS} together with a (runtime) environment containing internal semantic functions written in λ_{JS} itself. They also implemented an interpreter for λ_{JS}, which, combined with the translator and the runtime environment, allows to execute and test their semantics. Although the reduced semantics is helpful to understand the essentials of JavaScript, there is a gap between it and the actual language specification. Since their semantics does not directly follow the structure of the language specification, it is difficult to manually/visually inspect it and, indeed, it contains a number of bugs (see Table 2). We found that the JavaScript language specification, unlike for other languages, is quite well written, so we decided to follow it faithfully.

Bodin et al. (2014) [3] defined a JavaScript semantics in Coq, which, like KJS, follows ECMAScript 5.1. To execute and thus test it, they also implemented an interpreter, manually. Moreover, in order to link it to their semantics, they had to prove their interpreter correct. This step was inevitable, because their Coq specification is not executable—Coq can only extract executables from functions or proofs, not from specifications defined as inductive relations—yet testing is paramount when it gets to large semantics. Defining an interpreter and proving it correct for a complex language like JavaScript is a huge effort;[11] while a laudable and impressive feat in itself, we believe that such heavy approaches may demotivate language designers, for example the standards committee, to adopt a formal semantics. Compare that with KJS, where an interpreter is obtained directly from the semantics at no additional effort, together with other language analysis tools. Moreover, their semantics is incomplete. They omitted several language components such as the `for-in` loop and array manipulations. Table 1 shows that their semantics passes only about 65% of the conformance test suite.

On non-determinism To our knowledge, KJS is the only JavaScript semantics that captures the non-determinism of the language. For

[11] Indeed, Bodin *et al.* [3] involved 8 people, including domain experts of JavaScript and of Coq, for a year.

example, for the `for-in`'s iteration order, the standard says that the mechanics and order of enumerating the properties is left to the implementation; so from a semantic perspective, any order is possible. Without properly capturing the non-determinism of JavaScript, a semantics of it cannot execute and at the same time formally analyze JavaScript programs (e.g., show that the enumeration order is irrelevant in a given program). For example, Bodin *et al.* [3] chose to not provide a semantics for the `for-in` construct at all, Maffeis *et al.* [29] to define a partial semantics (with a hole for the enumeration order), and Guha *et al.* [23] and Politz *et al.* [37] to only consider a fixed, arbitrary order (given by Haskell's Hash Tables or OCaml's Map iteration order, respectively).

Verification of JavaScript programs While there is much work on finding bugs and security violations in JavaScript programs, verification of functional correctness of JavaScript programs is less developed. Gardner *et al.* [18] propose a (Hoare logic semantics with state properties specified using) separation logic for a JavaScript fragment. They follow the standard approach by defining an operational semantics as a model of the language, and then proving the separation logic sound w.r.t. the operational semantics. Like [3], this has the disadvantage of having to define different semantics of the same language for different purposes, together with soundness proofs, all huge efforts that require maintenance as the language evolves. Compare that to KJS, where only the operational semantics is required, and a deductive program verifier is automatically derived at no additional effort. Furthermore, their separation logic only supports manual reasoning and the programs they verified are significantly simpler than the programs in Table 3 which were verified automatically by KJS. Nordio *et al.* [32] present a program verifier for a JavaScript fragment. Their tool is implemented by translation to Boogie, and thus lacks a formal basis. Moreover, they can only verify simple properties that can be directly translated in Boogie.

Semantics for static analysis Other efforts to formally specify JavaScript semantics for the purpose of static analysis have been made. Lee *et al.* [28] provides a reduced semantics (i.e., defining an intermediate language into which the original language is translated), based on ECMAScript 5. Like Guha *et al.* [23] and Politz *et al.* [37], they do not directly follow the actual language specification, making manual/visual inspection hard. Kashyap *et al.* [27] also provides a reduced semantics for the purpose of abstract interpretation. Their semantics, however, is based on ECMAScript 3, and omitted the semantics of `eval`.

6.2 Other Large Language Semantics in $\mathbb{K}$

There are four major large language semantics defined in $\mathbb{K}$ so far, which served as a great source of inspiration for our JavaScript semantics: C [15], PHP [16], Python [24], and Java [4]. All these semantics are executable and they have been validated by a large volume of tests, and demonstrated useful through formal analysis tools produced by the $\mathbb{K}$ framework, same like our KJS.

Ellison and Rosu [15] defined a formal semantics of C11, which was extensively tested against the GCC torture test suite passing 99.2% of the tests, which is more than GCC and Clang passed. The C semantics was also evaluated by debugging, monitoring, and (LTL) model checking of example programs using corresponding tools provided by the $\mathbb{K}$ framework. A main application of their C semantics is undefinedness checking, e.g., in the context of compiler testing, for automatic test-case reduction [38].

Filaretti and Maffeis [16] defined a formal semantics of PHP. Since, unlike for JavaScript, C and Java, there is no official language standard for PHP, they had to heavily rely on testing against the reference implementation. They evaluated their semantics by model checking certain properties of a web database management tool, phpMyAdmin, and a cryptographic key generation library, pbkdf2.

Bogdanas and Rosu [4] gave a formal semantics of Java 1.4. To mitigate Java's complexity, they split their semantics into two phases: (1) the static semantics enriches the original program by annotating statically inferred information (e.g., types), and (2) the dynamic semantics gives the executable semantics. They evaluated the semantics by model checking multi-threaded programs.

Guth [24] defined a formal semantics of Python 3.3, providing semantics not only for the language constructs but also for the garbage collection mechanism. Being executable, it has been thoroughly tested against more than 600 hand-crafted tests. Like KJS, their semantics covers the core language but only essential parts of the standard libraries.

The most distinguished aspect of our semantics, compared to other language semantics described in $\mathbb{K}$, is the resemblance to the language standard (Figure 2); this facilitates visual inspection and allows us to measure the semantic coverage of a test suite. We did it by defining JavaScript on top of a semantics description language (Section 3.2), which was possible thanks to the JavaScript language standard being algorithmically described (unlike the language standards of other languages defined in $\mathbb{K}$).

7. Discussion and Future Work

Although KJS passes all the tests in the ECMAScript 5.1 conformance test suite for the core language, which is the reason why we call it a 'complete semantics', there is no guarantee that our semantics is necessarily correct. In the absence of a reference semantics, we believe that the best we can do to validate our semantics at this stage is to test it heavily against as many tests as possible, which we did, and to reason with it and prove certain expected properties of it, which we have not done yet but we plan to do as soon as a Coq backend becomes available for $\mathbb{K}$. In particular, a formal relationship between our semantics and that by Bodin *et al.* [3] can also be shown then using Coq.

The upcoming ECMAScript 6 [12] is now being actively developed and will be released soon. A natural question is whether a new version of the KJS semantics can be derived automatically or semi-automatically from the standard. Since our semantics is already systematically, but manually translated from the language standard, an automatic or semi-automatic translator may not be totally unfeasible. Recently, Ghosh *et al.* [19] studied automatic extraction of requirements specifications from natural language documents, showing that natural language processing (NLP) is now mature enough to be used in this context.

One of the most promising directions of future work is to use KJS to formally verify JavaScript programs against security properties of popular JavaScript applications.

Acknowledgments

We thank to $\mathbb{K}$ development team for many insightful discussions that helped develop the ideas in this paper, and to the anonymous reviewers for their helpful comments and suggestions. The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2014-2015, the NSF grants CCF-1218605, CCF-1318191 and CCF-1421575, and the DARPA grant under agreement number FA8750-12-C-0284.

References

[1] E. Arvidsson. V8 Issue 2243. `https://code.google.com/p/v8/issues/detail?id=2243`, 2012. Accessed: April 28, 2015.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX Security*, pages 22–22. USENIX, 2010.

[3] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A Trusted Mechanised JavaScript Specification. In *POPL*, pages 87–100. ACM, 2014.

[4] D. Bogdanas and G. Rosu. K-Java: A Complete Semantics of Java. In *POPL*, pages 445–456. ACM, 2015.

[5] D. Bruant. ECMAScript Bug 56. `https://bugs.ecmascript.org/show_bug.cgi?id=56#c3`, 2011. Accessed: April 28, 2015.

[6] D. Bruant. Mozilla Bug 641214. `https://bugzilla.mozilla.org/show_bug.cgi?id=641214`, 2011. Accessed: April 28, 2015.

[7] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a Self-hosted Research Virtual Machine for JavaScript: An Experience Report. In *Proceedings of the 7th Symposium on Dynamic Languages*, pages 61–72. ACM, 2011.

[8] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[9] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963, pages 337–340. LNCS, 2008.

[10] Ecma TC39. ECMAScript Harmony. `https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html`, 2008. Accessed: April 28, 2015.

[11] Ecma TC39. Standard ECMA-262 ECMAScript Language Specification Edition 5.1, June 2011.

[12] Ecma TC39. Draft Specification of ECMA-262 6th Edition. `http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts`, 2014. Accessed: April 28, 2015.

[13] Ecma TC39. TC39 Meeting Minutes. `https://github.com/rwaldron/tc39-notes/blob/master/es6/2014-09/sept-23.md#somehow-we-started-talking-about-test262`, 2014. Accessed: April 28, 2015.

[14] Ecma TC39. Test262: ECMAScript Language Conformance Test Suite. `http://test262.ecmascript.org`, 2014. Accessed: April 28, 2015.

[15] C. Ellison and G. Rosu. An Executable Formal Semantics of C with Applications. In *POPL*, pages 533–544. ACM, 2012.

[16] D. Filaretti and S. Maffeis. An Executable Formal Semantics of PHP. In *ECOOP*, volume 8586, pages 567–592. LNCS, 2014.

[17] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully Abstract Compilation to JavaScript. In *POPL*, pages 371–384. ACM, 2013.

[18] P. A. Gardner, S. Maffeis, and G. D. Smith. Towards a Program Logic for JavaScript. In *POPL*, pages 31–44. ACM, 2012.

[19] S. Ghosh, D. Elenius, W. Li, P. Lincoln, N. Shankar, and W. Steiner. Automatically Extracting Requirements Specifications from Natural Language. *CoRR*, abs/1403.3142, 2014.

[20] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code. In *USENIX Security*, pages 151–168. USENIX, 2009.

[21] 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. ACM, 2011.

[22] A. Guha, S. Krishnamurthi, and T. Jim. Using Static Analysis for Ajax Intrusion Detection. In *WWW*, pages 561–570. ACM, 2009.

[23] A. Guha, C. Saftoiu, and S. Krishnamurthi. The Essence of Javascript. In *ECOOP*, volume 6183, pages 126–150. LNCS, 2010.

[24] D. Guth. A Formal Semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

[25] D. Herman and C. Flanagan. Status Report: Specifying Javascript with ML. In *Proceedings of the 2007 Workshop on Workshop on ML*, pages 47–52. ACM, 2007.

[26] D. Herman, L. Wagner, and A. Zakai. asm.js. `http://asmjs.org`, 2014. Accessed: April 28, 2015.

[27] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *FSE*, pages 121–132. ACM, 2014.

[28] H. Lee, S. Won, J. Jin, J. Cho, and S. Ryu. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proceedings of the 2012 International Workshop on Foundations of Object-Oriented Languages*. ACM, 2012.

[29] S. Maffeis, J. C. Mitchell, and A. Taly. An Operational Semantics for JavaScript. In *APLAS*, volume 5356, pages 307–325. LNCS, 2008.

[30] Mean.io. MEAN: A Fullstack Javascript Framework. `http://mean.io/`, 2014. Accessed: April 28, 2015.

[31] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[32] M. Nordio, C. Calcagno, and C. A. Furia. Javanni: A Verifier for JavaScript. In *Fundamental Approaches to Software Engineering*, volume 7793, pages 231–234. LNCS, 2013.

[33] J. Orendorff. Mozilla Bug 779682. `https://bugzilla.mozilla.org/show_bug.cgi?id=779682`, 2012. Accessed: April 28, 2015.

[34] D. Park. WebKit Bug 138859, 138858; V8 Issue 3704; ECMA-262 Bug 3427, 3426; S5 Issues 55, 57, 59. `https://bugs.webkit.org/show_bug.cgi?id=138859`, `https://bugs.webkit.org/show_bug.cgi?id=138858`, `https://code.google.com/p/v8/issues/detail?id=3704`, `https://bugs.ecmascript.org/show_bug.cgi?id=3427`, `https://bugs.ecmascript.org/show_bug.cgi?id=3426`, `https://github.com/brownplt/LambdaS5/issues/55`, `https://github.com/brownplt/LambdaS5/issues/57`, `https://github.com/brownplt/LambdaS5/issues/59`, 2014. Accessed: April 28, 2015.

[35] D. Park and A. Stefanescu. Supplementary material. `https://github.com/kframework/javascript-semantics`, 2014. Accessed: April 28, 2015.

[36] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based Verification of JavaScript Sandboxing. In *USENIX Security*, pages 12–12. USENIX, 2011.

[37] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A Tested Semantics for Getters, Setters, and Eval in JavaScript. In *Proceedings of the 8th Symposium on Dynamic Languages*, pages 1–16. ACM, 2012.

[38] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case Reduction for C Compiler Bugs. In *PLDI*, pages 335–346. ACM, 2012.

[39] G. Rosu and T. F. Serbanuta. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6): 397–434, 2010.

[40] G. Rosu and A. Stefanescu. Checking Reachability Using Matching Logic. In *OOPSLA*, pages 555–574. ACM, 2012.

[41] M. Samuel. Properties of Interpreters or the Browser Environment that allow Privilege Escalation. `https://code.google.com/p/google-caja/wiki/AttackVectors`, 2009. Accessed: April 28, 2015.

[42] M. Samuel. Attack Vectors: Global Object Poisoning. `https://code.google.com/p/google-caja/wiki/GlobalObjectPoisoning`, 2009. Accessed: April 28, 2015.

[43] T. F. Serbanuta, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Rosu. The K Primer (version 3.3). In *Proceedings of the Second International Workshop on the K Framework and its Applications*, volume 304, pages 57–80. ENTCS, 2013.

[44] G. Smith. ECMA-262 Bug 1444. `https://bugs.ecmascript.org/show_bug.cgi?id=1444`, 2013. Accessed: April 28, 2015.

[45] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *S&P (Oakland)*, pages 363–378. IEEE, 2011.

[46] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *SPLASH*, pages 301–312. ACM, 2011.