HAL
open science

A Certified JavaScript Interpreter

Martin Bodin, Alan Schmitt

To cite this version:

Martin Bodin, Alan Schmitt. A Certified JavaScript Interpreter. JFLA - Journées francophones des langages applicatifs, Damien Pous and Christine Tasson, Feb 2013, Aussois, France. hal-00779459

HAL Id: hal-00779459

https://inria.hal.science/hal-00779459v1

Submitted on 22 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Certified JavaScript Interpreter

Martin Bodin[1] & Alan Schmitt[2]

1: Projet Celtique, Inria Rennes – Bretagne-Atlantique et ENS Lyon
martin.bodin@inria.fr
2: Projet Celtique, Inria Rennes – Bretagne-Atlantique
alan.schmitt@inria.fr

Introduction

The JavaScript language was initially developed for web pages enrichment, allowing the execution of scripts by the browser. It is now pervasively used on the web, not only to add interactivity in websites or to embed contents from third-party sources, but also as a target platform for deploying applications written in other languages (such as ocaml bytecode [VB11], Hop [SGL06], or LLVM assembly [Zak11]). In some sense, JavaScript has become the assembly language of the web, as most browsers are now able to run it. More recently, it has been used to program user interfaces for embedded systems, such as the defunct WebOS (now Enyo [Eny12]), the Kindle Touch ebook reader, or for the BootToGecko project [Moz12].

In addition to its pervasive use, JavaScript presents two important characteristics. First, as it was initially developed to facilitate its integration with the browser and with web contents, it aims more at providing powerful features than at giving robustness and safety guarantees. These powerful features include first class functions and closures, prototype-based objects, dynamic typing with many conversion functions, explicit scope manipulation, and the evaluation of strings as code. A second, redeeming, characteristic of JavaScript is that it is standardized [A+99], providing more information about how these features interact.

The goal of the JSCert project [BCF+12] is to provide a precise and formal semantics to JavaScript to build tools to certify analyses and compilation procedures. JSCert's collaborators have defined such a semantics in the Coq proof assistant, based both on the paper formalization of Maffeis et al. [MMT11, MMT08] and on the specification. To gain and provide more confidence in this formalization, we have implemented an interpreter that is proven correct in relation to the semantics. We will thus be able to confront our semantics against JavaScript test suites.

This paper describes the design and implementation of the interpreter. It is organized as follows. Section 1 introduces the semantics of JavaScript and highlights some of its peculiarities. Section 2 describes the interpreter's design and implementation. Section 3 addresses the interpreter's correctness. Finally, Section 4 concludes with future and related work.

1. JavaScript 's Semantics

JavaScript is defined by the standards ECMAScript 3 and 5 (ES3 and ES5) [A+99]. Most web browsers implement every feature of ES3 as well as some of ES5. Some also provide features that are not standardized, such as the modification of the implicit prototype of an object. The formalization described here is based on ES3 without any unspecified extension.

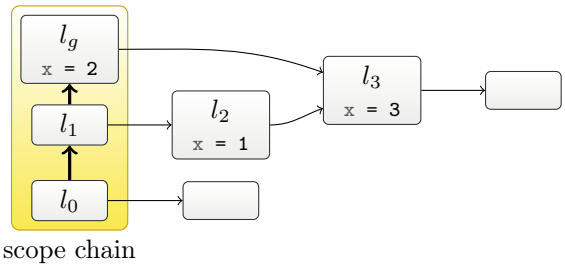


Figure 1: A JavaScript scope chain

1.1. Memory Model

The execution context of a JavaScript program comprises two objects: a *heap* and a *scope chain*. Objects in the heap are indexed by *locations*. Objects are maps from fields to values (including locations). Intuitively, locations can be seen as pointers.

The scope chain is a stack of locations (called *scopes* when they are in this stack). The top of the stack is a special location l_g pointing to the *global object*, where every global variable of the running program resides. When looking up the value of a variable `x`, it is searched in the scope chain. More precisely, the value of `x` will be found in the first location in the scope chain where it is defined. This behavior is similar to the one of a lexical scope (local variables having priority over global variables of the same name). However, as scopes are usual objects, they can be dynamically modified. Moreover, we will see below that scopes can be manually added to the chain using the `with` operator.

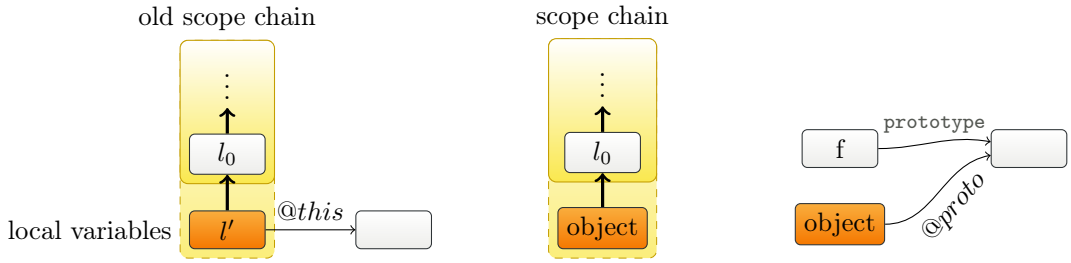
Variable look-up is also determined by the *prototypes* of the objects under consideration. Every object has an implicit prototype (in the form of a special field that *should not* be accessible, which we call @*proto*), possibly pointing to the special location `null`. Unless the prototype is `null`, it also has an implicit prototype, and so on, forming a prototype chain. The semantics of JavaScript guarantees that no loop can appear in the prototype chain. Intuitively, the field @*proto* of a location l points to a location representing the class from which l inherits. More precisely, each time a field `x` of a location l is looked up, l is checked to effectively have this field. If it is not the case, the prototype chain is followed until such an `x` is found.

We now describe how this mechanism interacts with the scope chain. Figure 1 shows an example of a JavaScript scope chain, where horizontal arrows depict the prototype chains. To access variable `x` in the current scope l_0, it is first searched in l_0 itself and its prototype chain. As `x` is not found, the scope chain is followed and the variable is looked up in l_1 and its prototype chain. This time, `x` is found in location l_2, thus the value returned is 1. Note that the value 2 of `x` present in l_g is shadowed by more local bindings, as well as the value 3 present in l_3.

Some special objects have a particular use. We have already encountered the global object, located at l_g. This object is where global variables are stored. As an object, its field @*proto* is bound to l_{op}, which we describe below. The global object is always at the top of the scope chain. A second special object is the prototype of all objects, `Object.prototype`, located at l_{op}. Every newly created object has a field @*proto* that is bound to l_{op}. It has some functions that thus can be called on every object (but they can be hidden by local declarations) such as `toString` or `valueOf`. Finally, the prototype l_{fp} of all functions, `Function.prototype`, is a special object equipped with function-specific methods.

1.2. Manipulating Heaps

The model presented above shows how a read is performed in a heap. Let us now see how the scope chain is changed over the execution of a program, typically because of the execution of a function call,

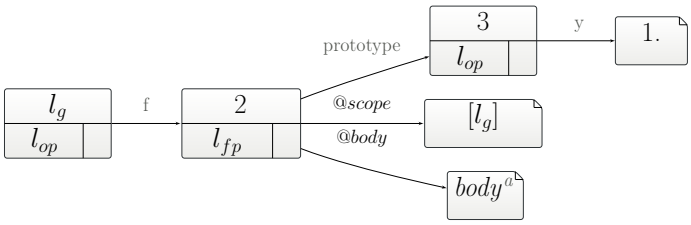


(a) Function Call. (b) `with`. (c) `new`.

Figure 2: Scopes chain manipulation

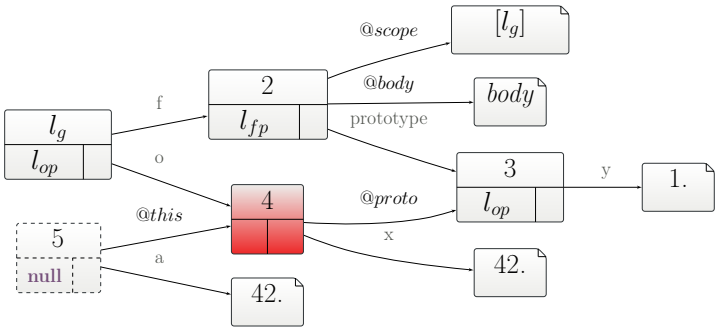
```
1 var f = function(a) {
2   this.x = a;
3 }
4
5 f.prototype = {y : 1};
6
7 var o = new f (42);
```

(a) The executed program.



(b) Heap's state at Line 6.

[a] This is the body of the function `f`, that is `function(a){ this.x = a; }`.



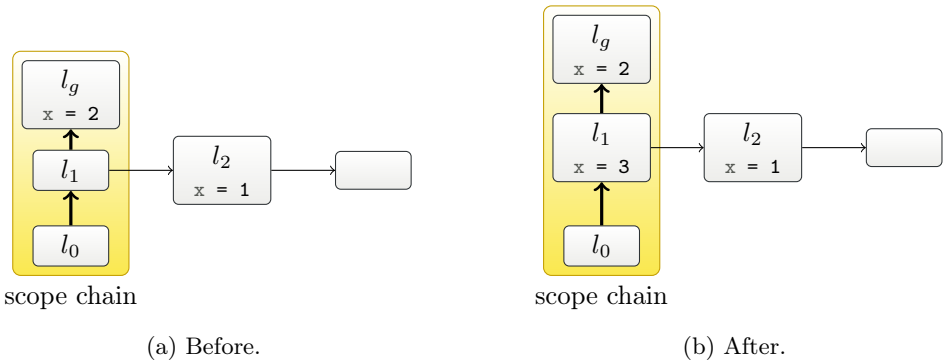
(c) Heap's state at the end of Line 7.

Figure 3: Effect of the `new` instruction

a `with` statement, a `new` statement, and an assignment. A graphical representation of those changes is summed up in Figure 2. In this figure, the "▭" orange blocks represents newly allocated locations.

As usual in functional programming language, the current scope chain is saved when defining new functions. Upon calling a function, the scope chain is restored, adding a new scope at the front of the chain to hold local variables and the arguments of the call. A special field *@this*, used by the `new` rule, is also added. The `with(o){...}` statement puts object `o` in front current scope chain to run the associated block. In the `new f (...)` case, function `f` is called with the special field *@this* assigned to a new object. The implicit prototype *@proto* of this new object is bound to the object pointed by the `prototype` field of `f`. This is how immutable prototype chains are created. The newly created object, which may have been modified during the call to `f` by "`this.x = ...`" statements, is then returned.

Example. A heap modified by a `new` operator called at Line 7 of the program of Figure 3a is shown in Figure 3. As some locations (such as `null`, l_{op}, or l_{fp}) are often used as implicit prototypes, they are put at the bottom left of locations instead of being explicitly depicted in the graph. Upon executing



(a) Before. (b) After.

Figure 4: Assignment

the `new` instruction, function `f` is called, adding a new location (the dashed one in Figure 3c) for the function call. In this location, the argument `a` is set to 42, and the @*this* field points to a new object (red in the figure). The function body is executed, which adds an `x` field to the red object. This object is then returned, setting its implicit prototype @*proto* to the value of the field `prototype` of `f`.

Targeted assignment, of the form `l.x = 3`, are straightforward: variable `x` is written with value `3` in the object at location `l`. For untargeted assignments, such as `x = 3`, things are more complex. The first scope for which the searched field is defined is selected in the current scope chain, following the same variable look-up rules as above. The variable is then written in the found scope. If no such scope is found, then a new variable is created in the global scope.

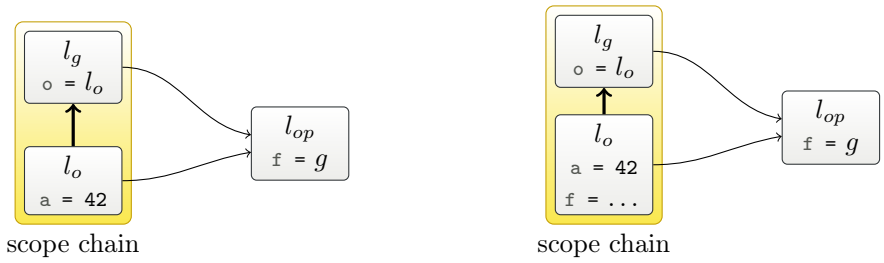
Figure 4 describes the assignment `x = 3`. Location l_1 is the first to define `x` in its prototype chain (in l_2). The new value of `x` is then written in l_1. Note that it is not written in l_2, allowing other objects that have l_2 in their prototype chain to retain their old value for `x`. Nevertheless, if one accesses `x` in the current scope chain, the new value 3 is returned. This allows objects constructed from the same function `f` (using `new`)—which thus have the same implicit prototype—to share some values, while letting them set those values without changing the shared one, similar to a *copy-on-write* approach. This approach may lead to surprising behaviors, as we now illustrate.

Example. Let us consider the following program.

```
1 var o = {a : 42};
2 with (o) {
3   f = function() {return a;};
4 };
5 f ()
```

If it is executed in an empty heap, it returns `42`. Indeed, when defining `f`, no such function already exists, thus `f` is stored in the global scope. When `f` accesses `a` upon its call, the object `o` is in the scope chain (as the call is executed in the scope chain of the definition of `f`), thus the result is `42`.

Let us now consider it in a slightly different scope, where `Object.prototype.f` has been set to a given function (say g =`function(){ return 18; }`). As the code `var o = {a : 42}` is almost equivalent to `var o = new Object(); o.a = 42`, object `o` has an implicit prototype set to l_{op}, which is `Object.prototype`. Figure 5 shows a representation of the heap at Line 3. As there is a variable `f` defined in the scope chain at position l_o (because of its prototype chain to l_{op}), the assignment is local to l_o and not global. At Line 5, the variable look-up for `f` returns the function g which is found in the prototype of the global object l_g (at this point the scope chain only contains l_g), and not the f defined in the `with` block. Thus the call returns `18`.



(a) at the beginning of Line 3.

(b) at the end of Line 3.

Figure 5: Heap state of the program in Page 4

There are many other subtleties in JavaScript's semantics which can be used to execute arbitrary code. For instance, implicit type conversion uses `toPrimitive` conversion functions, which could be redefined in a way similarly difficult to detect.

2. The Interpreter

2.1. Overview

We now describe the main contribution of this paper: an interpreter that rigorously follows a subset of the ES3 standard, more precisely a Coq-formalization of it. This formalization is based on the big step semantics presented in [GMS12] and covers a significant part of the core language, rich enough to observe some complex behaviors. We also rely on a formalization of a small-step-semantics [MMT11, MMT08], in particular for the primitive operators not formalized in the big-step-semantics.

The Coq formalization follows the specification closely and makes the distinction between expressions, statements, and programs. It includes function declarations, tests, `while` loops, `with` and `try`...`catch`...`finally` blocks, `throw` instructions, function calls, `new`, `delete`, assignments, construction of objects, objects accessors, `this`, `typeof`, and most unary and binary operators (including boolean lazy operators). We are currently working on adding type conversion and `eval` to this semantics. Labels and `break` statements are not yet supported.

This interpreter, proven correct with respect to our semantics, will allow to confront our semantics against JavaScript test suites. This will highly enforce its trustability.

The trusted base of the Coq formalization is composed of the three files `JsSyntax.v`, `JsSemantic.v`, and `JsWf.v` that respectively define the syntax, the semantics, and the invariants over heaps. They are fairly short (600 loc) when compared to the interpreter and the associated proofs (1800 loc). We will give more details about the correctness proof in Section 3.2.

The interpreter takes as arguments the initial heap, a scope chain, a program to be executed, and a bound on the number of reduction steps. This last argument is mandated by the termination requirement of Coq. The interpreter returns either `out_bottom`, when the bound is not large enough, for instance when the input program loops, or `out_return` when a result or an exception is returned.

2.2. Dependencies

The development relies heavily on the TLC library [Cha10b]. This library includes very powerful automation tactics, which has been especially useful to maintain the proofs as additional JavaScript constructs are added. We also use the Flocq library [BM10] to model `IEEE` floats in Coq.

$$\frac{}{\pi(H, \texttt{null}, x) \triangleq \texttt{null}} \qquad \frac{(l,x) \in \mathrm{dom}(H)}{\pi(H,l,x) \triangleq l} \qquad \frac{(l,x) \notin \mathrm{dom}(H) \quad H(l, @proto) = l'}{\pi(H,l,x) \triangleq \pi(H,l',x)}$$

Figure 6: Reduction rules associated with the function π.

In the spirit of TLC, proofs and definitions have mostly been done in a classical setting, which simplifies the development but requires additional care during extraction. For instance, consider the TLC expression `If x = y then e1 else e2`. It is defined as `if classicT (x = y)then v1 else v2`, where `classicT` is a lemma of type ∀(`P : Prop`), {`P`} + {¬`P`}. Such an expression is extracted to OCaml to `if isTrue then e1' else e2'` where `isTrue` is defined by an exception, independently of `x` and `y`.

```
(** val isTrue : bool **)

let isTrue =
  if let h = failwith "AXIOM TO BE REALIZED" in if h then true else false
  then true
  else false
```

To address this issue, decidable tests are associated to classical ones in a fairly transparent way, by defining for each test a boolean function `decide` to be used during extraction. This approach can be considered as a form of small-scale reflection, packing together a proposition stating a property and a decidable boolean function computing it is truth value.

```
Class Decidable (P:Prop) := make_Decidable {
  decide : bool;
  decide_spec : decide = isTrue P }.
```

The use of type classes [SO08] lets us avoid mentioning which function `decide` shall be used. We can now replace the code `If x = y then e1 else e2` by `if decide (x = y)then e1 else e2` (which we abbreviate to `ifb x = y then e1 else e2` for short). During the OCaml extraction, the comparison `if field_comparable x y then e1 else e2` is generated.

We also rely on an *optimal fixed point* to define the function π of [GMS12]. Function π searches through a prototype chain for a given variable and returns the location where it has been found (or `null` if not). It is defined formally in Figure 6, where $\mathrm{dom}(H)$ represents all the pairs of locations and field defined in the heap H, and $H(l,x)$ is the associated value for (l,x).

To facilitate the extraction of such a function, we use the optimal fixed point library [Cha10a]. This lets us separate the definition of the function from the proof that it terminates, avoiding in particular the use of dependent types in the definition. More precisely, we first define a function that performs one step of reduction. Here is how it is defined for the π function.

```
1 Definition proto_comp_body proto_comp h f l :=
2   ifb l = loc_null then loc_null
3   else ifb indom h l f then l
4   else ifb indom h l field_proto then
5       match read h l field_proto with
6       | val_loc l' ⇒proto_comp h f l'
7       | _ ⇒arbitrary
8       end
9    else arbitrary.
```

The identifier `arbitrary` at Lines 7 and 9 represents an arbitrary element of the return type (here a location), which is available as the return type is proven to be inhabited. In the extracted code, those calls to `arbitrary` are replaced by exceptions and should never happens (Section 3.1 details

the invariants of the heap that guarantee this). The final value for `proto_comp` is defined using the operator `FixFun3`, which will be extracted to OCaml as a `let rec` construction.

```
Definition proto_comp := FixFun3 proto_comp_body.
```

No proof is performed there as `FixFun3` is defined using (in addition with some classical logic) the predicate `Fix_prop` defining the greatest fixed point of `proto_comp_body`.

```
Fix_prop = fun (A : Type) (E C : binary A) (F : A → A) (x : A) ⇒
  greatest C (fixed_point E F) x
```

These steps are sufficient to define a function that can be extracted. Note that no property has been shown about this function. To be able to reason about it, we need to show that it is actually a fixpoint and that recursive calls are made according to a decreasing measure.

```
Lemma proto_comp_fix : ∀h f l,
  ok_heap h → proto_comp h f l = proto_comp_body proto_comp h f l.
```

In the proof of the preceding lemma, the only way to destruct the `FixFun3` operator is by a lemma that requires a decreasing measure related to `proto_comp_body` arguments. We can then use this fixpoint relation when reasoning about `proto_comp`. See [Cha10a] for additional details on optimal fixed points.

2.3. Structure of the Interpreter

The semantics and the interpreter are based on a presentation derived from the big-step-semantics, called *pretty-big-step-semantics*, described in [Cha12]. This presentation allows to factorize error and exception handling, as in a small-step presentation, while writing big-step rules.

Let us take the example of the rule associated to variable assignment in a big-step-semantics. This rule takes an initial heap H, a scope chain L, and an expression $\mathtt{e}$ to a final heap H_3 and a value $\mathtt{v}$. We write $H[\mathtt{l} \leftarrow \mathtt{v}]$ for the heap H where location[1] $\mathtt{l}$ has been updated with value $\mathtt{v}$.

$$\frac{H, L, \mathtt{e1} \longrightarrow H_1, \mathtt{Ref\ l} \quad H_1, L, \mathtt{e2} \longrightarrow H_2, \mathtt{v} \quad H_3 = H_2[\mathtt{l} \leftarrow \mathtt{v}]}{H, L, \mathtt{e1} = \mathtt{e2} \longrightarrow H_3, \mathtt{v}}$$

This rule is simple, but it does not take into account errors and exceptions. The problem is that adding those constraints would lead to some duplications in this rule: the part $H, L, \mathtt{e1} \longrightarrow \ldots$ executing $\mathtt{e1}$ would appear four times:

$$\frac{H, L, \mathtt{e1} \longrightarrow H_1, \mathtt{exn}}{H, L, \mathtt{e1} = \mathtt{e2} \longrightarrow H_1, \mathtt{exn}}$$

$$\frac{H, L, \mathtt{e1} \longrightarrow H_1, \mathtt{v} \quad \mathtt{v} \neq \mathtt{Ref\ l}}{H, L, \mathtt{e1} = \mathtt{e2} \longrightarrow H_1, \mathtt{exn}}$$

$$\frac{H, L, \mathtt{e1} \longrightarrow H_1, \mathtt{Ref\ l} \quad H_1, L, \mathtt{e2} \longrightarrow H_2, \mathtt{exn}}{H, L, \mathtt{e1} = \mathtt{e2} \longrightarrow H_2, \mathtt{exn}}$$

$$\frac{H, L, \mathtt{e1} \longrightarrow H_1, \mathtt{Ref\ l} \quad H_1, L, \mathtt{e2} \longrightarrow H_2, \mathtt{v} \quad H_3 = H_2[\mathtt{l} \leftarrow \mathtt{v}]}{H, L, \mathtt{e1} = \mathtt{e2} \longrightarrow H_3, \mathtt{v}}$$

Note that only exceptions are dealt with there: the same duplication occurs for all the instructions breaking the normal control flow (such as `return` or `break`). This uselessly duplicates some rules, making them difficult to read and increasing the redundancy in the proofs.

[1] Technically $\mathtt{l}$ is a pair of a location and a field name, but to simplify the presentation we leave it abstract.

$$\frac{H : \texttt{e1} \Downarrow o_1 \qquad H : o_1 =_1 \texttt{e2} \Downarrow o}{H : \texttt{e1} = \texttt{e2} \Downarrow o} \qquad \frac{\textbf{abort}\, o}{H : o =_1 \texttt{e2} \Downarrow o} \qquad \frac{v \neq \texttt{Ref l}}{H : (H_1, v) =_1 \texttt{e2} \Downarrow (H_1, \texttt{exn})}$$

$$\frac{H_1 : \texttt{e2} \Downarrow o_2 \qquad H_1 : \texttt{l} =_2 o_2 \Downarrow o}{H : (H_1, \texttt{Ref l}) =_1 \texttt{e2} \Downarrow o} \qquad \frac{\textbf{abort}\, o}{H : \texttt{l} =_2 o \Downarrow o} \qquad \frac{H_3 = H_2[\texttt{l} \leftarrow \texttt{v2}]}{H : \texttt{l} =_2 (H_2, \texttt{v2}) \Downarrow (H_3, \texttt{v3})}$$

Figure 7: Rules for binary operators in pretty-big-step-semantics.

```
| red_expr_expr_assign : ∀h0 s e1 e2 o o1,
    red_expr h0 s e1 o1 →
    red_expr h0 s (ext_expr_assign_1 o1 e2) o →
    red_expr h0 s (expr_assign e1 e2) o

| red_expr_ext_expr_assign_1 : ∀h0 h1 s e2 re o o2,
    red_expr h1 s e2 o2 →
    red_expr h1 s (ext_expr_assign_2 re o2) o →
    red_expr h0 s (ext_expr_assign_1
      (out_expr_ter h1 re) e2) o

| red_expr_ext_expr_assign_2 : ∀h0 h1 h2 s r l x v,
    getvalue h1 r v →
    h2 = update h1 l x v →
    red_expr h0 s (ext_expr_assign_2 (Ref l x)
      (out_expr_ter h1 r)) (out_expr_ter h2 v)
```

(a) Main rules related to assign expressed in the Coq 's pretty big step semantics.

```
| exp_assign e1 e2 ⇒
  if_success (run' h0 s e1) (fun h1 r1 ⇒
    if_is_ref h1 r1 (fun l f ⇒
      if_success_value (run' h1 s e2) (fun h2 v ⇒
        out_return (update h2 l f v) v)))
```

(b) The assign rule expressed in the Coq interpreter.

Figure 8: Comparison of a rule expressed in the Coq semantics and the Coq interpreter.

The pretty-big-step semantics consists in splitting the assignment rule to several sub-rules representing the partial reductions of the arguments of the assignment, as if we were defining a small-step-semantics. To this end, outcomes o that are pairs of the final heap and either a value or an exception are added. The expressions $o =_1 \texttt{e}$ and $\texttt{l} =_2 o$ are also introduced, as well as a new predicate $\textbf{abort}\, o$. This predicate is satisfied when o is $(H, \texttt{exn})$. The rule for $\texttt{e1} = \texttt{e2}$ is now split between computing (and eventually propagation of special results) of its sub-expressions `e1` and `e2`, and in the effective computing of the assignment. Note that in rules of the form $H : o_1 =_1 \texttt{e2} \Downarrow o$, the heap H is not taken into account, it is the heap in o_1 that will be used for the remainder of the computation. Figure 7 shows those new rules.

In this presentation, only the last rule computes, the other ones simply evaluate intermediate results and propagate errors. This avoids duplication of nearly identical reduction rules, which are frequent in the semantics. As a side effect, this increases proof automation performance as automatic tactics will not have to evaluate twice identical sub-reductions. We have found this approach to be very useful as we added constructs to the language which changed the control flow.

Figure 8 shows an example of reduction rule from the interpreter compared to the reduction rule in the semantics. The main difference between the two is that the semantics is defined as an inductive predicate, whereas the interpreter is a fixpoint. The three main cases of Figure 7 can be seen in (a) (we do not depict the abort rules). The first rule evaluates `e1` and passes the result `o1` to the second rule. The second rule extracts from its first argument the heap and result `re` of the evaluation of `e1` and uses them first to evaluate `e2`, then to pass the result `o2` to the third rule. The third rule is only defined if `re` is actually a reference. Another rule would cover the other cases, resulting in an abort. The third rule also extracts from its second argument the heap and result `r` of evaluating `e2`. It may then get the value `v` from `r` using `getvalue`. This function corresponds to the γ function of [GMS12]:

```
Lemma proto_comp_correct : ∀h f l l',
  ok_heap h →
  bound h l →
  proto_comp h f l = l' →
  proto h f l l'.
```

(a) Correctness.

```
Lemma proto_comp_complete : ∀h f l l',
  ok_heap h →
  bound h l →
  proto h f l l' →
  proto_comp h f l = l'.
```

(b) Completeness.

Figure 9: Correctness and completeness of the function π with respect to the `proto` predicate.

if its argument is a reference, it returns the value stored at this reference; if its argument is a value, it returns this value. A final heap is built, updating the reference with `v`, and the returned outcome is this heap and value `v`.

The `run'` function is the interpreter's main function, set with a bound on the maximum number of reduction inferior to the current one. Each of the functions of the interpreter (see Figure 8b) `if_success_value`, `if_defined`, ..., represents one pair of reduction rule in the pretty big step presentation. Indeed in pretty-big-step, rules can often be grouped in pairs: one that deals the case where the previous execution sent an exception, and the other case that deals the normal one. For instance in Figure 7, the two first rules would be packed in one function matching the return of `e1`. Grouping the reduction rules in such functions makes the writing style of the interpreter monadic, which helps proving its correctness.

The interpreter is thus written in a continuation-style, which is relatively close to the pretty big step reduction. When the result of `run' h0 s e1` is a failure, the continuation is not evaluated and the failure is returned. Similarly, when `if_is_ref` is called at Line 3, `r1` is destructed to a reference to location `l` and field `f`, a failure being raised if it is not a reference.

Except for those minor differences, the definition of the interpreter and the reductions rules are very similar. For each predicate defined in the semantics, a function is defined in the interpreter. For instance, to `proto` (which is the predicate representation of π in the semantic file) of type `jsheap → field → loc → loc → Prop` corresponds a function `proto_comp` of type `jsheap → field → loc → loc`. Those duplications are needed as in proofs it is easier to manipulate inductive propositions than functions. Furthermore, as predicate representation is closer to paper definitions, it is better to have inductive predicates rather than implementations in the trusted base. This separation also allows to optimize the implementation, as long as it is proven to correspond to the predicate. For instance, Figure 9 shows that `proto_comp` is equivalent to the predicate `proto` under the assumption that their arguments are valid.

2.4. Extracting the Interpreter

The extraction of the interpreter in OCaml is fairly straightforward. We extract natural numbers to `int` and Coq strings to list of characters. As the floats use in JavaScript are the same one that of OCaml, we extract floats from Flocq into OCaml floats as follows.

```
Require Import ExtrOcamlZInt.
Extract Inductive Fappli_IEEE.binary_float ⇒float [
  "(fun s →if s then (0.) else (-0.))"
  "(fun s →if s then infinity else neg_infinity)"
  "nan"
  "(fun (s, m, e) →let f = ldexp (float_of_int m) e in if s then f else -.f)"
].
Extract Constant number_comparable ⇒"(=)".
```

```
Extract Constant number_add ⇒"(+.)".
Extract Constant number_mult ⇒"( *. )".
Extract Constant number_div ⇒"(/.)".
Extract Constant number_of_int ⇒float_of_int.
```

The extracted interpreter is about 5000 loc. As efficiency is not our goal, we have not run benchmarks. We have however tested it on small programs, allowing to enforce our trust on the 600 loc semantics.

3. Proving Properties

3.1. Heap's Correctness

Let us re-consider the rules of Figure 6 for the π function, that looks for a given variable in a prototype chain. There exist some cases where this function is not defined. For instance when $l \neq$ `null`, $(l, x) \notin \mathrm{dom}(H)$, and $(l, @proto) \notin \mathrm{dom}(H)$. We have to prove that π is never called in such cases (otherwise the reduction is unsound).

To this end, we rely on a definition of heap correctness. It is defined as a record of some properties. Among those correctness condition, there are for instance `ok_heap_null` that states that the location `null` is not bound in the heap, or `ok_heap_protochain` that states that each bound location has a correct prototype chain (the field *@proto* has thus to be defined for each non-`null` element in it and no loop should appear). Similarly, a notion of scope chain correctness and of result correctness have been defined.

One important result of the JSCert project is a safety theorem, stating that this notion of correctness is conserved through the formalized JavaScript reduction rules. As we need to cover expressions, statements, and programs which all depend on each other, we write the theorem as a fixpoint, giving explicitly the decreasing argument on which to base the inductive proof.

```
Fixpoint safety_expr h s e o (R : red_expr h s e o) {struct R} :
  ok_heap h →
  ok_scope h s →
  ok_ext_expr h e →
  ok_out_expr h o
with safety_stat h s p o (R : red_stat h s p o) {struct R} :
  ok_heap h →
  ok_scope h s →
  ok_ext_stat h p →
  ok_out_prog h o
with safety_prog h s P o (R : red_prog h s P o) {struct R} :
  ok_heap h →
  ok_scope h s →
  ok_ext_prog h P →
  ok_out_prog h o.
```

This theorem is actually expressed over extended expressions, which contain some heaps or intermediary results. The `ok_ext_ext`, `ok_ext_stat`, and `ok_ext_prog` predicates state that those intermediary objects are also correct.

To better understand the statement of the theorem, here is one of the constructors of `ok_out_expr` (the other constructors deal with exceptions).

```
Inductive ok_out_expr h0 : out_expr → Prop :=
```

```
2     ok_out_expr_normal : ∀h (r : ret_expr),
3                          ok_heap h →
4                          ok_ret_expr h r →
5                          extends_proto h0 h →
6                          ok_out_expr h0 (out_expr_ter h r)
```

The predicate `extends_proto` states that the implicit prototypes (the value of their fields @*proto*, not to be confused with their fields `prototype`) of objects cannot be removed through reduction: their values can change, but those fields cannot be deleted. There are indeed checks that may lead to exceptions on the deletion of some fields when calling the `delete` operator (for instance, it is not possible to delete a @*proto* field). This predicate is needed in function calls as old scope chains are restored; those old scope chains having to be still correct in the new heap, and thus have to keep their prototypes. This is ensured by lemmas such as the following.

```
1 Lemma ok_scope_extends : ∀h1 h2 s,
2   ok_scope h1 s →
3   extends_proto h1 h2 →
4   ok_scope h2 s.
```

3.2. Interpreter's Correctness

The interpreter has been proven correct: each time it returns a defined result (a value or an exception), this result was correct with respect to the semantic rules. If it returns `out_bottom` because the bound it was given over the maximum number of reductions was not large enough, or any other error, then the theorem does not apply. Here is the property the theorem proves, the predicates `ret_res_expr` interfacing the data-types of the interpreter and of the semantics.

```
1 Definition run_expr_correct_def m := ∀h s e h' r re,
2   run_expr m h s e = out_return h' r →
3   ret_res_expr r re →
4   ok_heap h →
5   ok_scope h s →
6   red_expr h s e (out_expr_ter h' re).
```

For most cases of this theorem, the proof strategy consists in splitting the goal using elimination lemmas in all the possible cases to deconstruct the equality of Line 2 until a result is returned by `run_expr`, then proving properties of intermediary results and at last folding the `red_expr` predicate. Let us consider the example of the assignment rule given in Figure 8. Three functions are nested there, thus three elimination lemmas will be used. Let us consider the one for `if_success` (see Section 2.3).

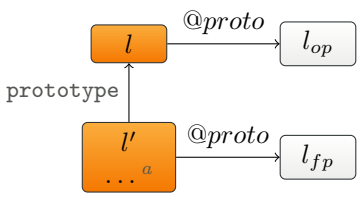
```
1 Lemma elim_if_success : ∀r0 k h r,
2   if_success r0 k = out_return h r →
3   (r0 = out_return h r ∧ ∀v, r ≠ret_res v) ∨
4     ∃r1 h0, r0 = out_return h0 (ret_res r1).
```

It takes as argument the fact the computation stops, and gives a disjunction: either `r0` was an exception or an error, or there exists a result. After applying this lemma, the goal is split in two. In one case, the result was either an error (which is not possible as Line 2 of the theorem's statement expect it to be a result), or it is an exception and we may directly apply an `abort` rule to conclude. In the other case, we get a result `r1` and a new heap, which allows us to rewrite the equality `if_success r0 (...)= out_return h' (ret_res v)` to:

```
1   R : if_is_ref h1 r1 (fun (l : loc) (f : field) ⇒
2         if_success_value (run m h1 s e2) (fun (h2 : jsheap) (v : val) ⇒
```

```
| red_expr_expr_function_unnamed : ∀l l' h0
    h1 h2 s lx P,
  fresh h0 l →
  h1 = alloc_obj h0 l loc_obj_proto →
  fresh h1 l' →
  h2 = alloc_fun h1 l' s lx P l →
  red_expr h0 s (expr_function None lx P)
      (out_expr_ter h2 l')
```

(a) Coq code to define a new function.



(b) A graphical representation of it.

[a] This location now contains the current scope chain and the function body P.

Figure 10: Allocating new unnamed function.

```
      out_return (update h2 l f v) v))
  = out_return h' v
```

The first step of the assignment rule's computation has now been performed and the proof continues. In this case, the fact that the new heap `h1` is correct is directly given by the safety theorem, but this is not always the case. Indeed, when some writes are performed on heaps, this last theorem does not always apply for those intermediary results. This has led to some code copied from the safety proof to the interpreter correctness proof. It may be interesting to slightly change the structure of the safety proof to factorize those steps.

3.3. Completeness

The interpreter has not yet been proven complete. The main reason is that in the inductive semantics, heap allocation is unspecified, whereas the interpreter allocates locations in a deterministic way. Let us consider the simplest reduction rule that involves some allocations in the heap: the declaration of a new unnamed function. Figure 10 shows the corresponding Coq rule and a graphical representation of its meaning. The predicate `fresh`, called twice, is defined by stating that the given location is not `null` and is not bound in the current heap, with no other restriction. In the interpreter, however, the function `fresh_for` is defined as the minimum location number not yet allocated in the current heap.

To prove completeness, one has to show that given equivalent heaps, where every location is injectively renamed, an expression reduces to the same value and to equivalent heaps. As we keep extending the semantics to take additional constructs into account, we have not yet proven this result. Going further, given a heap and a list of live locations, we could define an equivalent heap which may contain fewer locations, thus add garbage collection to the interpreter. We are currently exploring another option, which consists in making the semantics deterministic by equipping the heap with an unspecified function that deterministically returns fresh locations.

A second reason for delaying the proof of completeness concerns parsing. Indeed, as JavaScript has an `eval` operator, parsing is part of the semantics of programs and the exact parsing rules have to be formalized. As first approximation, we could define a partial parsing algorithm which may wrongly reject some programs. This approach would obviously not be complete, but it would also not be correct, as a parse error in the context of the execution of an `eval` operator is a runtime exception, which we need to model. Thus the property of a string not being parseable must be formally defined and we have to aim for completeness to correctly support the `eval` operator. The problem with JavaScript parsing is that it is quite complex. For instance, semicolons are not always mandatory, and the rules expressing *automatic semicolon insertion* use backtracking, exceptions, and further tests. This precludes the use of classical parser technology. We plan on working on this issue when most of the language is formalized, relying on and extending existing work on parser validation [JPL12].

Finally, a practical way to test for completeness would be to confront our interpreter to existing test suites. We are also planning on doing so as soon as we have covered the language sufficiently.

4. Related and Future Work

4.1. Related Work

The work on JSCert and on the interpreter could not have been done without relying on the formalization by Maffeis et al. [GMS12, MMT11, MMT08]. There have been other attempts to give formal accounts of JavaScript's semantics, which we now review.

A common approach is to define a simple formal language or calculus on which properties are more easily proved, then write a desugaring function from JavaScript to the simpler language. The λ_{JS} project [GSK10] follows such an approach in a small-step setting. They have proven, in Coq, that the produced terms never get stuck (they either can reduce, are a value, or are an error). However, they only relate the JavaScript terms and the desugared version through testing. To this end, they implemented an interpreter for λ_{JS}. In [CHJ12], Chugh et al. present DJS, an extension of their calculus for dynamic languages [CRJ12], with features to mimic JavaScript constructions such as imperative updates, prototype inheritance, and arrays. As in λ_{JS}, they use desugaring to go from JavaScript to DJS, with no formal claim of correctness.

Other approaches focus more on the formalization of how JavaScript interacts with the browser, from network communication to the DOM representation [Boh12, YCIS07]. These works do not aim at covering the whole language and present very promising extensions to our semantics and interpreter, once we have finished formalizing the core language.

Finally, many other works have formalized part of JavaScript in order to develop static or dynamic analyses, focusing on particular aspects of the language and its runtime. In particular, API access has been studied based on a Datalog model of JavaScript [TEM+11]; Hedin and Sabelfeld have written a big-step paper semantics of JavaScript to dynamically track information flow [HS12]; Luo and Rezk have proposed a decorated semantics to prove correctness and security properties of a JavaScript to JavaScript compiler for mashups [LR12]. It is our hope that a complete and precise formalization will help avoid further duplication of effort.

4.2. Extensions of the Interpreter

Our goal is to include all of the core specification from ES3. We are currently working on adding `eval` and type conversion to the semantics and the interpreter. We will then turn to some primitive objects and features, such as arrays, to test the interpreter against more realistic programs. We will then consider the additions of ES5, in particular *strict mode* and accessors.

Further, we want to go beyond the specification. In practice, most real world interpreters do not strictly follow it, typically by accepting reads or even writes on some (theoretically) unaccessible fields (such as the implicit prototype @*proto*, often called `__proto__` by those non-standard interpreters). We plan to add such features in the formalization (and thus in the interpreter) in a modular way. To this end, we will parameterize the interpreter by some flags describing which standards it should follow (ES3, ES5, Firefox's one, etc.). It will then be possible to prove security of a given program for a given non-standard browser.

Finally, we plan to use the interpreter to test the semantics on real JavaScript test suites. This will provide additional trust on the semantics and thus everything that depends on it (such as certified code analysis). Interestingly enough, writing and proving the interpreter has actually helped uncovering some missing cases and misconceptions in the Coq semantics. These bugs were fairly subtle, most

of them being a confusion between extended expressions `ext_expr` (introduced by the pretty big step presentation) and expressions `expr`, which is difficult to verify by hand.

Conclusion

We have presented the design and implementation of a JavaScript interpreter proven correct in relation with the semantics developed in the JSCert project. The main motivation for developing a formal semantics for JavaScript is twofold: JavaScript has become pervasive in web development, and its semantics is fairly complex. As a result, providing strong guarantees for JavaScript programs is difficult yet would have a significant impact.

The proof of correctness of the interpreter has been done in the Coq proof assistant, and the implementation is extracted in OCaml from the development. Many features of JavaScript are supported, including prototype-based inheritance, explicit scope manipulation, and exceptions. The development is available on the JSCert web site [BCF+12].

The completeness of the interpreter has not yet been proven, for two main reasons. First, it requires relating the undeterministic heap allocation of the semantics to the deterministic one of the interpreter. Second, it needs to formally specify the parsing of strings to correctly and completely model the `eval` operator.

We believe that our three-tiered approach—write a semantics following closely the specification, independently write and prove correct an interpreter, and test the interpreter—yields a good level of trust in the formal semantics to base further works on it. We have started to investigate the formal development of static analyses of JavaScript programs as a continuation of this work.

References

[A+99] European Computer Manufacturers Association et al. Ecmascript language specification, 1999.

[BCF+12] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. Jscert: Certified javascript. http://jscert.org/, 2012.

[BM10] S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. Soumis à ARITH-20 (2011), October 2010.

[Boh12] A. Bohannon. *Foundations of Web Script Security*. PhD thesis, University of Pennsylvania, 2012.

[Cha10a] A. Charguéraud. The optimal fixed point combinator. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceeding of the first international conference on Interactive Theorem Proving (ITP)*, volume 6172 of *LNCS*, pages 195–210. Springer, 2010.

[Cha10b] A. Charguéraud. TLC: a non-constructive library for coq based on typeclasses. http://www.chargueraud.org/softs/tlc/, 2010.

[Cha12] A. Charguéraud. Pretty-big-step semantics. Submitted, October 2012.

[CHJ12] R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *Proceedings of OOPSLA 2012*, 2012.

[CRJ12] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: a logic for duck typing. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 231–244, New York, NY, USA, 2012. ACM.

[Eny12] Enyo. Enyo web site. http://enyojs.com/, 2012.

[GMS12] P.A. Gardner, S. Maffeis, and G.D. Smith. Towards a program logic for javascript. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–44. ACM, 2012.

[GSK10] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. *ECOOP 2010–Object-Oriented Programming*, pages 126–150, 2010.

[HS12] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 3–18, Cambridge, MA, USA, June 2012.

[JPL12] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating lr(1) parsers. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pages 397–416, Berlin, Heidelberg, 2012. Springer-Verlag.

[LR12] Z. Luo and T. Rezk. Mashic compiler: Mashup sandboxing based on inter-frame communication. In IEEE, editor, *Proceedings of 25th IEEE Computer Security Foundations Symposium*, pages 157–170, Cambridge, MA, USA, June 2012.

[MMT08] S. Maffeis, J. Mitchell, and A. Taly. An operational semantics for javascript. *Programming Languages and Systems*, pages 307–325, 2008.

[MMT11] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for javascript. http://jssec.net/semantics/, 2011.

[Moz12] Mozilla. B2g wiki. https://wiki.mozilla.org/B2G, 2012.

[SGL06] M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, pages 975–985, Portland, OR, USA, October 2006. ACM.

[SO08] M. Sozeau and N. Oury. First-class type classes. *Theorem Proving in Higher Order Logics*, pages 278–293, 2008.

[TEM+11] A. Taly, Ú. Erlingsson, J.C. Mitchell, M.S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 363–378. IEEE, 2011.

[VB11] J. Vouillon and V. Balat. From bytecode to javascript: the js_of_ocaml compiler. unpublished, 2011.

[YCIS07] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 237–249, New York, NY, USA, 2007. ACM.

[Zak11] A. Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 301–312, New York, NY, USA, 2011. ACM.