A Certified JavaScript Interpreter

Martin Bodin
M2 ENS Lyon

Abstract

Although it was initially designed for running small scripts in web pages, JavaScript has become the programming language of the web. It is designed to be very dynamic, for instance by allowing the evaluation of strings as code or by letting programmers to explicitly specify the scope in which a program runs. These aspects allow for great flexibility, but significantly hinder the understanding of the semantics of programs, such as the development of certified analyses.

In practice, it is frequent to insert external code in a web page (such as an advertisement or an interactive map) and make it interact with some scripts carrying some potentially secret information. It would be useful to be able to prove the safety of a web page despite the presence of unknown (thus untrusted) code.

In this internship, we present a formalisation in Coq of JavaScript's semantics. Our main result is a JavaScript interpreter proven correct with respect to the Coq's semantics. This work is the first step in the building of certified analysers.

Supervisors Alan Schmitt, Thomas Jensen

Thanks. I would like to thank Alan Schmitt and Thomas Jensen for this very nice internship in the weather-favored town of Rennes. I also thank Arthur Charguéraud, Sergio Maffeis, Daniele Filaretti, Gareth Smith, Philippa Gardner and Daiva Naudziuniene for their kindness: it is really been a pleasure to work with them. I also thank and the members of the Celtique team for their welcome and of course all the people that have made this internship possible in such good conditions.

Contents

Introduction

JavaScript was first designed to run small scripts in web pages. It was intended to be flexible and was not designed for its current massive use. Nowadays, it is also used as a target language for some programming languages (including Caml, see the Ocsigen project [VB11]), the advantage of such a compilation scheme being the compatibility with every operating system as soon as there is a JavaScript-enabled compatible browser. The prevalence of JavaScript motivates the development of analyses. Such analyses first require a solid specification of the semantics of the language.

JavaScript is not object oriented, but it has some mechanisms used to "mimic" some aspects of object oriented programming. It thus suffers of many features that are complex (or less common in other programming languages) to analyse. The three main ones are the absence of types, some complex rules of variable scope, and an evaluation operator.

JavaScript is not a typed programming language and has a lot of conversion functions. For instance, converting an integer to a string goes through the (implicit) call of the `toString` method. However, mosts attributes (and methods) of an object are dynamically mutable: a simple type conversion can call arbitrary code if the `toString` method has been overwritten. Thus analyses have to trace the assignments of objects's (or classes's) methods and attributes.

Such a trace is especially complex as JavaScript has various way of changing the value of a variable. Indeed, the scope of variables is *dynamically* handled (modifiable for instance using the `with` keyword): a call to the function `f` defined by `function(o){ with(o){ x = 0; } }` can in some environment assign a different variable `x` depending its argument (without having to re-define `f`): `f({x:1})` would change a local `x` whereas `f({})` would change a global one! To give an idea of the particularity of variables's scope, Figure 1 shows an example of a valid JavaScript's program whose final values of its different variables are not that intuitive.

```
1 x = null ; y = null ; z = null ; v = 5 ;
2 f = function(w) { x = v ; v = 4 ; var v ; y = v } ;
3 f (null) ; z = v
```

Figure 1: An example of counter intuitive JavaScript program.

Last but not least, one big difficulty of JavaScript (and of some other scripting languages) is due to the `eval` feature. It evaluates the string given as an argument as a JavaScript program placed in place of the `eval`. In JavaScript, `eval` is a *function* instead of an operator: it is possible to assign `eval` to a variable or to return this value in the end of a function. Analysis on what code can be executed by a given program is thus rather complex.

In practise though, programmers use abstractions to work at higher levels. For instance if they do not use the `with` keyword, some of the problems presented above never appear. The issue is that it is very frequent in JavaScript to communicate with untrusted code. The most frequent case being when adding an applet from an external website (such as an interactive map for instance) and having some interactions with it. In this case, the external (thus untrusted) code will be executed in the same environment than the analysed program. An other case when this problem appears is when writing a library that will be called by untrusted code. Of course, such untrusted code may not use the same abstractions and policies and this may break the correctness of the analysed code. Those aspects of the language have to be taken into account in the analysis.

One advantage in analysing JavaScript's codes is that it has been designed to be similar to many programming languages: it is functional[1], it uses a heap, is has objects, it has `eval` operator, etc.: if one can analyse such a programming language, one can probably analyse a lot of real world scripting languages.

My contributions in this internship has been to specify a JavaScript interpreter, to prove its correctness and to develop some additional tools to run JavaScript programs. After a short

[1] Only in the sense that functions are values: it is not purely functional.

overview of JavaScript's semantics and peculiarities, we will focus on the JavaScript interpreter and its certification.

1 JavaScript's semantics

JavaScript is defined by the standards ECMAScript 3 and 5 (ES3 and ES5) [A+99]. Variants (or "strict modes") have attempted to secure JavaScript, for instance the *secure ECMAScript* 5 that could evolve to a standard ECMAScript 6.

In practice, web browsers implement in a more or less strict way ES3's features and some of ES5's. Some other "low-level features" (which are not part of any standard) seem to also be present, even if those features could break some code security.

1.1 Memory Model

The execution context of a JavaScript program comprises two objects: a *heap* and a *scope chain*.

The objects in the heap are indexed by *locations*; each of those objects being a map from fields to values (possibly locations). Intuitively, each of those locations can be seen as a pointer. JavaScript uses this notion of location to "mimic" other languages's behaviours, with some differences.

The scope chain is a stack of locations (called *scope* when they are in this stack) defined in the heap. When looking for the value of a given variable `x` in a JavaScript program, this variable is searched in this stack until found in one of those locations, the first scope having a defined variable `x` having priority on the other.

This behaviour is exactly the one of a lexical scope (local variables being prioritised over global variable of the same name). This stack intuitively represents this, with the difference that stack's elements are objects dynamically changeable, and that their structure can be lightly manipulated within a given function, typically when using the `with` operator.

Moreover, despite the fact that JavaScript is not object oriented, a construction has been added in order to simulate some of OOP's aspects: implicit prototypes. Each location of a heap contains one (in the form of a special field that *should not* be accessible, called @*proto*), possibly pointing to the special location `null`. Unless this implicit prototype is equal to `null`, it has an implicit prototype too, and so on: this forms a prototype chain. It is possible to show that no loop appears in the execution of a valid JavaScript program. Note that many browsers do not follows the standard and allow code to access and change prototype chains. It's often performed through the field `__proto__` (that should not be special according to the standard). Fortunately, it usually checks the presence of loops in the heap, potentially raising an exception `"cyclic __proto__ value"`.

Intuitively, the field @*proto* of a location l points to a location representing the class from which l inherits; classes being objects (like all other things) in JavaScript. More precisely, each time one wants to access a field `x` of a location l, one checks if l effectively has this field. If it is not the case, the prototype chain is followed until such an `x` is found.

Figure 2 shows an example of a JavaScript scope chain.

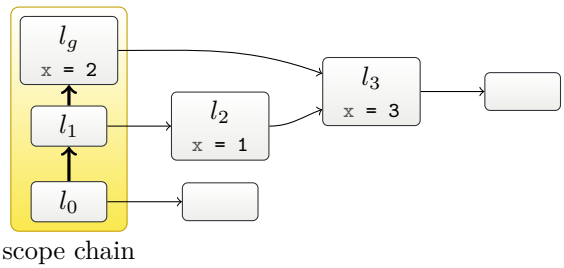


Figure 2: A JavaScript's scope chain.

In the scope chain of Figure 2, if we want to access the variable `x`, a JavaScript interpreter will check if it is present in l_0 and its prototype chain (which is not the case), then will follow the

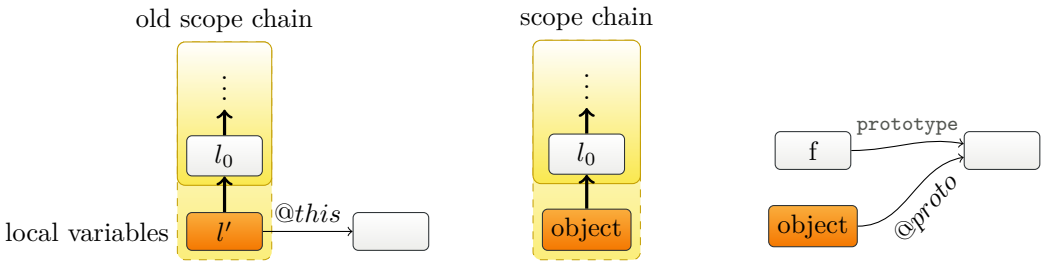
scope chain to l_1 and checks if `x` is present there. It is (as the prototype chain leads to l_2 which defines `x`) and the final value of `x` is there 1. Note that the value 2 of `x` present in l_g is here ignored, as well as the value 3 present in l_3.

There are some special locations that have a particular use:

- The prototype of all objects, `Object.prototype` in JavaScript, noted l_{op}. All newly created object has a field *@proto* that is bound to it. It has some methods that thus can be used in all objects (but they can be hidden by local declarations) such as `toString` or `valueOf`.
- In a similar way, the prototype l_{fp} of all functions, `Function.prototype` in JavaScript, is a special location with some particular methods.
- The global object, noted l_g. This special object is where global variables are stored. As an object, its field *@proto* is bound to l_{op}. It is always at the top of the scope chain.

1.2 Manipulating Heaps

The model presented above shows how a read is performed in a JavaScript's heap. Let us now see how the scope chain is changed over the execution of a JavaScript program, typically on the execution of a function call, a `with`, a `new`, and an assignment. A graphical representation of those changes is summed up in Figure 3. In this figure, the "▭" orange blocks represents newly allocated locations.



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

Figure 3: Main Changes of the Scopes Chain.

As in every functional programming language, the current scope chain is saved when defining new functions. When a function call is performed, the saved scope chain is restored. A new scope is created and pushed in this old scope chain. Local variables and arguments are defined in this new scope. As this new object is added at the head of the scope chain, accessing local variables will return the variables defined in this new scope, which corresponds to the notion of local variable. A special field *@this* is also added. This special field will then be used by the `new` rule.

The way of determining local variables is something very special in JavaScript as it is performed through an additional read of the function body: each variable `v` declared by a `var v` (whatever the place of this instruction) will be declared at the beginning. Thus, in the code of Figure 1, `v` will be a local variable declared at the beginning of the function call, even in instructions placed before its declaration.

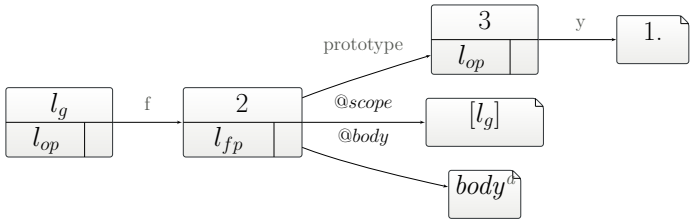
When executing a `with` instruction, a new object (argument of `with`) is simply added to the current scope chain. This thus adds all the members of this object as local variables (as accessible directly in the current scope chain) and allows the programmer to hide variables, which is not always a good thing to do when not aware of the exact hidden variables.

In the `new f (...)` case, the function `f` is called with its arguments and a special field *@this* assigned to a new object. The (non modifiable) *@proto* field of this new object is bound to the (modifiable) `prototype` field of `f`. This new object is then returned. This allows to build prototype chains, but those chains are immutable once created.

Note that new locations are created each time a function is called, but those locations are often never used again after the end of their functions execution. Interpreters use garbage collection to remove those useless locations.

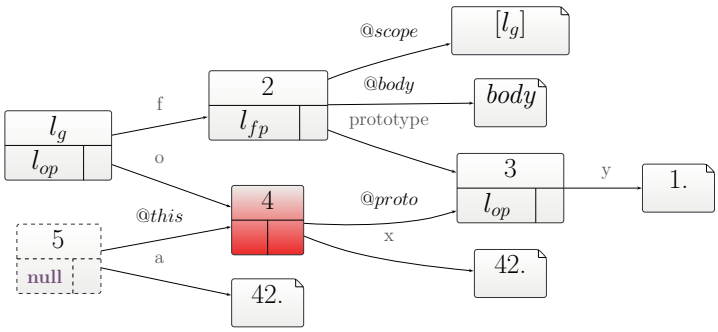
```
1  var f = function(a) {
2    this.x = a;
3  }
4
5  f.prototype = {y : 1};
6
7  var o = new f (42);
```

(a) The executed program.



(b) Heap's state at Line 6.

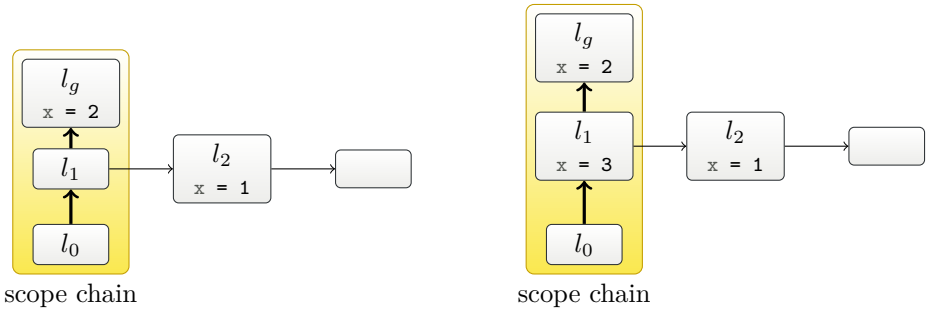
[a] This is the body of the function `f`, that is `function(a){ this.x = a; }`.



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

Figure 4: Effect of the allocation of a new object.

Example. A heap modified by a `new` operator called at Line 7 of the program of Figure 4a is shown in Figure 4. As some locations (such as `null`, l_{op} or l_{fp}) are often used as a value of fields @*proto*—which can make the figure unreadable—, these depictions of heaps represents special values of @*proto* at the bottom left of locations. Once the `new` operation is executed, the function `f` is called, thus adding a new location (the dashed one in Figure 4c) for the function call. The argument `a` of `f` is set to the parameter 42 of the `new`, which adds a field in the new location. After the function call, the @*this* object thus created (in red in the figure) is then manipulated: its @*proto* is set to the value to the field `prototype` of `f`. The dashed location is now useless and can now be garbage collected.



(a) Before. (b) After.

Figure 5: Assignment.

Figure 5 shows how writes are performed, for instance in `x = 3`: in a given scope chain, it searches for the first location that has the searched field, including in its prototype chain. Here, only locations on the current scope chain can be changed. It is possible to change another, but one needs to do a direct assignment of the form `l.x`.

In the example, it is `x` that is searched, and l_1 is the first location defining it in its prototype chain (more precisely in l_2). This returns a reference $(l_1, \texttt{x})$ (and not $(l_2, \texttt{x})$) and the new value of `x` is written in this reference, i.e. in l_1. Note that this new 3 value is not written in the same place as the old value 1. If one tries to get the new value for the variable `x`, it is the new value 3 that is given: the old one is hidden. This allows objects constructed from the same function `f` (using `new`)—which thus have the same prototype object—to share some values, while allowing to specify one of those object without changing the shared value. Those shared variables are not copied when constructing a new object, in a *copy-on-write* approach.

If a variable is not defined in the entire scope chain, then a new variable is created at global scope. Such a situation can be relatively frequent in real world scripts (it happens each time a variable is used but not declared), but it can be dangerous as it relies on the fact that such a variable is defined nowhere in the scope chain.

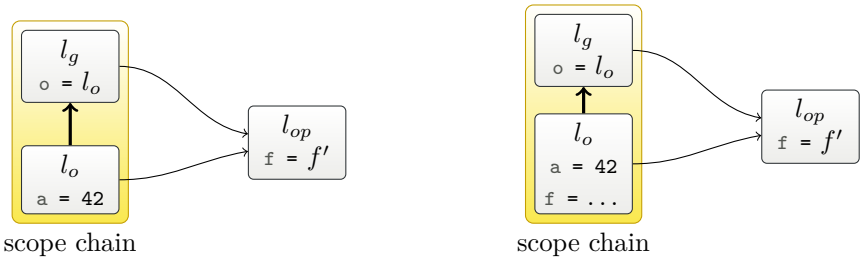
Example. Let us consider the following example:

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

If it is executed in an initial heap, such a program will return `42`. Indeed, when defining function `f`, no such function already exists and `f` is defined at global scope. When it tries to access `a`, the object `o` is in the scope chain (as it is executed in the scope chain of the definition of `f`) and will give the value `42` for it.

Let us now consider it in a slightly different scope, where `Object.prototype.f` has been set to a given function (say $f' =$ `function(){return 18;}`). As the code `var o = {a : 42}` is almost equivalent to `var o = new Object(); o.a = 42`, `o` has a field *@proto* pointing to the location l_{op}, which is `Object.prototype`.

Figure 6 shows a heap representation of the state 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. Thus the call of Line 5, which is performed with only l_g in the scope chain, gets f' which was assigned to `f` in l_{op}. The function that used to be called was thus not called at all. Even worse, f' could have executed any untrusted code there. This short program illustrate potential security issues.



(a) at the beginning of Line 3.

(b) at the end of Line 3.

Figure 6: Heap's state of the program in Page 6.

Armed with those rules, it is now possible to analyse the example of Figure 1: when calling the function `f`, the local variable `v` is declared to `undefined`, then changed to `4` *after* assigning its value to `x`. At the end, the final values of respectively `x`, `y` and `z` are `undefined`, `4` and `5`.

1.3 Type Conversion

There is no static typing in JavaScript, but there are types: boolean, number, string, object and the type of `undefined` and `null`.

Some basic operators perform some conversions: for instance, the unary operator "`+`" converts its argument to a number. There are various and complex rules to convert each type to another, some that raise `TypeError` exceptions (which is rare).

The main issue here is that an apparently harmless type conversion can lead to an arbitrary function call. Indeed, converting an object `obj` to either a string or a number will call an abstract function `toPrimitive` that will try to access the values `toValue` and `toString` and try to call them. Note that those functions can return anything (including nothing) and may performs some side effects: this really leads to arbitrary code being executed.

All the modifications of the values `toValue` and `toString` have thus to be traced, including the ones of all the objects of the prototype chain of `obj`. But the modifications of these values can be subtly hidden. For instance using JavaScript's associative maps, which are in fact a shortcut for objects: writing `t[e]` evaluates `e`, then converts it to a string `f`, then accesses `t.f`. Thus, in such a program, one has to determine whether the conversion to a string of the result of `e` can return `"toString"` or `"toValue"`.

Programs that read a string from an input, then assign it to an associative map can thus be attacked if there is no sanitiser for the expression `i`. There exists some JavaScript programs called *filters* whose goal is to restrict the execution of some other JavaScript programs which performs such operations, as in `api.store = function(i, x){ priv[i] = x }`.

2 The Interpreter

2.1 Overview

A first step of this internship has been to design an interpreter that rigorously follows the standard ES3, or at least the Coq-formalised part of it.

The Coq formalisation is based on the big step semantics presented in Paper [GMS12]. It formalises a significant part of the core language, rich enough to observe some complex behaviours.

A small step semantics [MMT08] has been formalised by Sergio Maffeis and has been a support to create the interpreter—in particular in the cases of primitive operators unformalised in the big step semantic. This semantics implements a very large part of JavaScript: only some libraries (DOM objects, the `Math` object, etc.) are not represented there. It is available on [MMT11].

Figure 7 shows how big are each Coq file of the formalisation. The trusted base is composed of files `JsSyntax.v`, `JsSemantic.v` and `JsWf.v` for respectively the syntax, the semantics and the invariants over heaps in JavaScript. As can be seen, it is relatively short compared to proofs.

The interpreter has been proven correct in Coq with the formalised semantics (see Section 3.2). It takes as an argument the initial heap, a scope chain, the program to be executed, and a bound on the maximum reduction steps to be performed. This last argument is forced by the fact that Coq requires its functions's termination (even if it is, like here, artificial), but all JavaScript's programs do not terminate. It returns either `out_bottom` when this bound argument wasn't large enough (or when its input code loops) or `out_return` when a result or an exception is returned. Note that exceptions are considered as a result as they can happen even in valid programs.

2.2 Dependencies

The TLC library [Cha10a] has been largely used in proofs. It contains a lot of alternative Coq definitions for basics operations and a lot of automation tactics. The automation tactics have been especially useful when definitions of rule reductions changed a lot while developing the proofs to deal with more JavaScript constructs. Those automated tactics are quite resilient to minor definitions changes, such as the order of lemma's arguments.

The Flocq [BM10] library has been used to model `IEEE` floats used by JavaScript in Coq.

File name	Size (Kio)		Description
	Definitions	Proofs	
`JsSyntax.v`	4.5	0	Syntax of JavaScript
`JsSyntaxAux.v`	1.5	7.5	Basic properties on objects defined in `JsSyntax.v`
`JsSemantic.v`	21.5	0	Reduction rules for JavaScript
`JsSemanticAux.v`	0.5	15.5	Basic properties on objects defined in `JsSemantic.v`
`JsWf.v`	4.5	0	Definition of inductive principles over heaps conserved through execution
`JsWfAux.v`	0	6.5	Basic properties on objects defined in `JsWf.v`
`JsSafety.v`	0	33.5	Proof that the inductive principles defined in `JsWf.v` holds in a JavaScript program execution
`JsScopes.v`	0	1.5	Some lemmas for the definition of the interpreter
`JsInterpreter.v`	17	1.5	Definition of the JavaScript interpreter
`JsInterpreterProof.v`	0.5	42.5	Proof of the interpreter's correctness
Total	50	108.5	

Figure 7: Approximative size of each Coq file.

Proofs and definitions have mostly been done in classical logic, which can lead to some computability problems when extracting code from Coq's proofs. For instance a Coq expression of the form `If x = y then e1 else e2` would be extracted to Caml to something like `if isTrue then e1 ' else e2'` where `isTrue` (independent of `x` and `y`) would be defined by an exception:

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

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

`If x = y then e1 else e2` is defined by `if classicT (x = y)then v1 else v2`, where `classicT` is a lemma of type $\forall$`(P : Prop), {P} + {¬P}` (which is equivalent in classical logic to $\forall$`P, P ∨¬P`) proven using some classical logic's axioms (not extracted as not constructive). These constructions are part of the TLC library (here in the file `LibLogic.v`).

In order to bypass this problem, decidable tests had to be separated from classical ones in a more or less transparent way in the Coq code (to avoid the already complex Coq proofs to be worse). This has been performed by the definition of a decidability predicate of a formula, which allows to replace (once decidability has been proven) in the Caml code each test on this formula by the corresponding function. Proving this predicates amounts to define the boolean function `decide` which will then be instantiated in a transparent way.

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

Implicit Arguments decide [[Decidable]].
Extraction Inline decide.
```

This approach can be considered as a case of small-scale reflection, packing together a proposition and a decidable Boolean function computing it is truth value.

The choice of using type classes [SO08] to define it allows to forget (once the decidability of the property has been defined) mentioning which function `decide` use. The code `If x = y then e1 else e2` can now be written `if decide (x = y)then e1 else e2` (`ifb x = y then e1 else e2` for short). This will then be extracted to something like `if field_comparable x y then e1 else e2`, which is exactly what we want.

An *optimal fixed point* has been used to define the function π of Paper [GMS12]. This function searches through a prototype chain for a given variable and returns the location where it has been found first (or `null` if not). This particular fixed point has been defined by Arthur Charguéraud in [Cha10b] to avoid using dependant types that would force to make a part of the correctness proof directly in the code declaration rather than in the tactic language of Coq. The main idea of it is to delay the need for a proof of decreasing argument.

Here is how it works: firstly, give a "body" function, that is a function that performs one step of reduction. Here is how this function 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.
```

To compare, here are the rules for π:

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

In those rules, $\mathrm{dom}(H)$ represents all the couples of locations and field defined in the heap H and $H(l,x)$ is the associated value for (l,x).

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 inhabited. In the extracted code, those calls to `arbitrary` are replaced by exceptions. Of course, the cases where this function returns `arbitrary` should never happens (see Section 3.1 for more information about it). If we were using a usual `FixPoint` there, dependant types would have been necessary to show that these cases never happens, but we do not need it there.

Then the final value for `proto_comp` is given using a special operation `FixFun3`. To simplify, this function defines a lazy function. It will be extracted to Caml as a `let rec` construction.

```
1 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`:

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

For the extraction, those steps are enough to get something that works. Of course, if some measure does not decrease the extracted code may fail to terminate. For the proof, we need the fact that it is well defined, that is:

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

But at this point the function `proto_comp` has already been defined and can thus be unfolded to its definition (as soon as we unfold a finite number of steps). This is enough to prove its definition.

Note that in this process, no dependant type did appear in the definition of `proto_comp`.

2.3 Structure of the Interpreter

The interpreter is based on a presentation derived from the big step semantic, called *pretty big step semantic*, described in [Cha12]. This presentation allows to factorise error and exception handling (like in small step presentation) while writing rules in a way similar to big-step.

Let's take the example of the rule associated to binary operators in a big step semantic:

$$\frac{H, L, \texttt{e1} \longrightarrow H', \texttt{v1} \quad H', L, \texttt{e2} \longrightarrow H", \texttt{v2} \quad \texttt{v1} \overline{\oplus} \texttt{v2} = \texttt{v}}{H, L, \texttt{e1} \oplus \texttt{e2} \longrightarrow H", \texttt{v}}$$

This rule is simple, but it does not take into account errors and exceptions. The problem is that adding those constraints in this rule will lead to a duplication of some parts of it: if `e1` $\oplus$ `e2` is executed and that `e2` throws an exception, `e1` should have been executed firstly, which force to rewrite twice the fact that `e1` terminates on something different than an error. Here are the resulting rules in big-step:

$$\frac{H, L, \texttt{e1} \longrightarrow H', \texttt{exn}\, e}{H, L, \texttt{e1} \oplus \texttt{e2} \longrightarrow H', \texttt{exn}\, e}$$

$$\frac{H, L, \texttt{e1} \longrightarrow H', \texttt{v1} \quad H', L, \texttt{e2} \longrightarrow H", \texttt{exn}\, e}{H, L, \texttt{e1} \oplus \texttt{e2} \longrightarrow H", \texttt{exn}\, e}$$

$$\frac{H, L, \texttt{e1} \longrightarrow H', \texttt{v1} \quad H', L, \texttt{e2} \longrightarrow H", \texttt{v2} \quad \texttt{v1} \overline{\oplus} \texttt{v2} = \texttt{v}}{H, L, \texttt{e1} \oplus \texttt{e2} \longrightarrow H", \texttt{v}}$$

Note that only exceptions (represented here by the notation `exn` e) are dealt there. In a real world semantic, one has to represent errors and all the instructions breaking the normal control flow (such as `return` or `break`). This duplicates uselessly such rules, which make them more difficult to read and increase the redundancy (and the development) of proofs.

The part $H, L, \texttt{e1} \longrightarrow H', \texttt{v1}$ is duplicated twice in the new rules. In this simple example, there are only two sub-rules that have to be reduced, but there exists rules (such as the ones for `for` loops or for `new`) with a lot of such sub-rules, which leads to a lot of redundancy in the big step rules.

Pretty big step semantics consists on splitting $\oplus$'s rule to 3 sub-rules representing the partial reductions of the arguments of $\oplus$, as if we were defining a small step semantic. The expressions $\oplus_1$ and $\oplus_2$ are thus introduced. A new predicate **abort** o is also introduced, whose meaning is that the result o "goes through" its context, which allows to merge the rules used for modelling exceptions, errors and `return`, `break` and `continue` statements. The rule for `e1` $\oplus$ `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 operation $\oplus$. Figure 8 shows those new rules.

$$\frac{\texttt{e1} \Downarrow o_1 \quad o_1 \oplus_1 \texttt{e2} \Downarrow o}{\texttt{e1} \oplus \texttt{e2} \Downarrow o} \quad \text{(a)} \qquad \frac{\textbf{abort}\, o}{o \oplus_1 \texttt{e2} \Downarrow o} \quad \text{(b)} \qquad \frac{\texttt{e2} \Downarrow o_2 \quad \texttt{v1} \oplus_2 o_2 \Downarrow o}{\texttt{v1} \oplus_1 \texttt{e2} \Downarrow o} \quad \text{(c)}$$

$$\frac{\textbf{abort}\, o}{\texttt{v1} \oplus_2 o \Downarrow o} \quad \text{(d)} \qquad \frac{\texttt{v1} \overline{\oplus} \texttt{v2} = \texttt{v}}{\texttt{v1} \oplus_2 \texttt{v2} \Downarrow \texttt{v}} \quad \text{(e)}$$

Figure 8: Rules for binary operators in pretty big step semantic.

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 of Coq files directly based on ES3. Note that such a rule duplication implies a proof duplication of their correctness. This also increases proof automation performance as automatic tactics won't try to re-evaluate two times identical sub-relations. Note that speaking of tactic automation's performance may seem odd, but the fact is that the proofs done in this internship use a lot of automation, and their generated proofs terms are huge. During development, final `Qed` of big proofs have been replaced by `Admitted` (even though the proof was correct and accepted) to remove the final type and guardedness checking which takes several minutes of compilation on my laptop. Of course those checks will be re-established in the final version of the proofs.

Figure 9 shows an example of reduction rule from the interpreter compared to the reduction rule in the semantics (in big step without exception). The assignment rule `red_assign` presented

```
1  | red_assign : ∀l f v h0 h1 h2 h3 s e1 e2 r2,
2    red h0 s e1 h1 (Ref l f) →
3    red h1 s e2 h2 r2 →
4    getval h2 r2 v →
5    h3 = update h2 l f v →
6    red h0 s (exp_assign e1 e2) h3 v
```

(a) The access rule expressed in the Coq's semantic.

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

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

Figure 9: Comparison of a rule expressed in the Coq semantics and the Coq interpreter.

in (a) means that when evaluating `e1 := e2` (which is written `exp_assign e1 e2` in the Coq syntax for JavaScript), `e1` is evaluated first, which changes `h0`, the current heap, to a new one `h1`. It should returns a reference to a field `f` at location `l`. Then `e2` is evaluated (giving a heap `h2`), returning a result which is dereferenced using `getval` to a value `v`. This last function corresponds to the γ function of Paper [GMS12]: if its argument is a reference, it returns the value stored at this reference; if its argument is a value, it returns this value. Then a new heap `h3` is defined, in which the value stored in field `f` at location `l` is now `v`. The final value is at last `v` in the heap `h3`. This operation thus evaluates `e1` and `e2` and assign the result of `e2` in the reference given by `e1`.

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 9b) `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 for $\oplus_1$ in Figure 8, the two rules (b) and (c) would be packed in one function matching the return of `e1`. Regrouping the reduction rules in such functions makes the writing style of the interpreter monadic, which helped writing its proofs.

The interpreter is thus written in a continuation-style, which is relatively closed 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 isn't a reference.

This difference of presentation between the reduction rules and the interpreter can be explained by the fact that at one hand the reductions rules have to be a trusted base and thus be as simple as possible. At the other hand, the interpreter is proven correct and should thus be written in a way simplifying later addition of new features of the languages, such as exceptions and errors handling.

Excepting those minor differences, the definition of the interpreter and the reductions rules are very similar: for each predicates defined in the semantic, 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`. This difference comes from the fact that it is easier to manipulate inductive propositions in proofs than functions. Indeed the function π is defined in Paper [GMS12] using reductions rules (see Figure 11) and not by a given implementation, which couldn't be as clear. In proofs it is even clearer: if we got the fact that π applied on something gives a result with some properties, the Coq tactic `inversion` can go backward to properties about the arguments. If π is defined with

an implementation, that would force to use `destruct` (or variant of it) on its arguments, which is much more difficult to handle if the arguments have a complex form. An other argument is that as the predicate representation is closer to the definitions on papers, it is better to have inductive predicates rather than some function implementations in the trusted base. Separating like that predicates and implementation can also allow to make some optimisations in the implementation that can be proved using the corresponding predicate.

In practise, it is better that way: for each function (there are a lot of then in fact) given with a predicate, an implementation have to be given. Then some proofs of equivalence have to be done between the predicate and the implementation. In the later proofs of all the other functions using those functions, it is then possible to choose whether a given property should use the predicate or the implementation and even to rewrite from one to the other (it is often the predicate which is chosen).

For instance, Figure 10 shows the properties proven for π stating that it is equivalent to its predicate. The Coq name of the predicate π is `proto`, and the Coq name of its implementation is `proto_comp`. Note that further properties are given as arguments to those lemmas: correctness and completeness are only valid when the arguments of π are valid!

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

2.4 Some Words About Parsing...

In most programming languages, this part is not very interesting, but as JavaScript has an `eval` operator, not interpreting correctly a code may lead to non-standard behaviour. Furthermore, if the parser wrongly raises an exception because it thinks that a given code is invalid, this exception can be caught by the JavaScript program and may change its global behaviour. Having a correct parser is thus rather important to work on `eval`.

JavaScript has a lot of minor syntax issues that make it difficult to understand. For instance let us consider the JavaScript binary operator "`+`". It first evaluates its two arguments. If one of them is a string, it converts the other one to a string and returns their concatenation. Otherwise, it converts both of them to numbers and performs an addition.

Now consider the code "`{} + {}`", which looks like being the empty object "`{}`" added to another one (which should give as a return value the string `"[object Object][object Object]"`, following the standard), but is in fact interpreted as an empty block of instruction followed by an expression "`+ {}`", which means "convert to a number the empty object" and returns `NaN`. Note that the returned result doesn't even have even the same type as the expected result!

In this case, the problem comes from the differences in JavaScript between *statements* and *expressions*: statements can be either instructions such as `if` blocks, function declaration, or can be expressions. To be short, statements cannot be put in any context (as an argument of a function, between parenthesis, ...) whereas expressions can. The case where a statement is an expression has the lowest priority (and sometimes is not even accepted by JavaScript without adding parenthesis, that is forcing the statement to be an expression). In this example, the code `({} + {})` returns the expected string value.

But the worst of those issues occurs with semi-colons: some semi-colons can be removed, but in a rather strange way. The exact rule from the standard is that if a parsing fails, but that adding a semi-colon at the beginning of a line makes it accept the code, then add this semi-colon (of course, there are some exceptions, such as semi-colons in `for` loops).

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

Figure 11: Reduction rules associated with the function π.

Here follows an example of a valid JavaScript program that uses this feature. In practise though, such programs are not frequent (even in real world scripts). This program was found in https://github.com/berke/jsure/blob/master/README. But as making an analyser that does not implement well the standard wouldn't give many garanties, this part of JavaScript has to be taken into account.

```
function GetLen_(G) { var K, L, C = ""  // Get good margin
  G.Len.value = K = L = Math.max(G.Len.value, 1)
  while (--K) C += " " ; G.Result.value = C + "|"
  return L }
```

Expressing it is very difficult with a LALR grammar, and making a parser in Coq is somewhat painful. The solution of this problem (which has not been implemented yet) is to make a Coq verifier, that checks if a given syntax tree can possibly corresponds to a given string. The parser can thus be performed in Caml without having to prove the correctness of it (just by calling the Coq verifier on its result).

Such techniques are similar to the ones performed in the certified compiler CompCert [LBDT08] in case of functions that are not realistically programmable in Coq.

3 Proving Properties

3.1 Heap's correctness

Let us re-consider the rules for the π function of the paper, the function that looks for a given variable in a prototype chain. Figure 11 gives its definition.

There exists some cases where this function is not defined. For instance when $l \neq \texttt{null}$, $(l,x) \notin \mathrm{dom}(H)$ and $(l, @proto) \notin \mathrm{dom}(H)$. We have to prove that π is never called in such cases (otherwise the reduction is unsound).

A definition of heap correctness is needed. It is defined as a record of some properties which we do not detail. Here are two examples of correctness condition:

- `ok_heap_null` states that the location `null` is not bound in the heap.
- `ok_heap_protochain` states that each bound location has a correct prototype chain: the field $@proto$ has to be defined for each non-`null` element in it and no loop should appear.

Similarly, there is a notion of scope chain correctness and of result correctness.

A safety theorem proven by Daniele Filaretti and Sergio Maffeis states that this notion of correctness is conserved through the formalised JavaScript reduction rules:

```
Theorem safety : ∀h s e h' r,
  red h s e h' r →
  ok_heap h →
  ok_scope h s →
  ok_heap h' ∧
  ok_scope h' s ∧
  ok_result h' r ∧
  extends_proto h h'.
```

The predicate `extends_proto` states that the prototypes (i.e. 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:

```
Lemma ok_scope_extends : ∀h1 h2 s,
  ok_scope h1 s →
  extends_proto h1 h2 →
  ok_scope h2 s.
```

This proof of correction is very long as the correctness of each intermediary heap and scope have to be proven, some reduction rules having a lot of such intermediary heaps:

```
  | red_new : ∀l l1 l2 l3 l4 v1 lv2 s3 lx ys e3,
    ∀h0 h1 h2 h3 h4 h5 h6 h7 h8 s e1 le2 r1 r3 v3 lfv,
    (* Evaluate constructor *)
    red h0 s e1 h1 r1 →
    getval h1 r1 l1 →
    l1 ≠loc_null →
    binds h1 l1 field_scope (val_scope s3) →
    binds h1 l1 field_body (val_body lx e3) →
    ys = defs lx e3 →
    binds h1 l1 field_user_prototype v1 →
    l2 = obj_or_glob_of_value v1 →
    (* Evaluate parameters *)
    red_list_val h1 s le2 h2 lv2 →
    (* Init new object *)
    fresh h2 l4 →
    h3 = alloc_obj h2 l4 l2 →
    (* Create activation record *)
    fresh h3 l3 →
    ys = defs lx e3 →
    h4 = write h3 l3 field_proto (val_loc loc_null) →
    h5 = write h4 l3 field_this (val_loc l4) →
    arguments lx lv2 lfv →
    h6 = write_fields h5 l3 lfv →
    h7 = reserve_local_vars h6 l3 ys →
    (* Execute function (constructor) body *)
    red h7 (l3::s3) e3 h8 r3 →
    getval h8 r3 v3 →
    l = obj_of_value v3 l4 →
    red h0 s (exp_new e1 le2) h8 l
```

3.2 Interpreter's Correctness

The interpreter has been proven correct, that is each time it returns a defined result, 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 reduction was not large enough, or any other error, then the theorem does not apply. Here is the statement of the theorem:

```
Theorem run_correct : ∀m h s e h' v,
  run m h s e = out_return h' (ret_res v) →
  ok_heap h →
  ok_scope h s →
  red h s e h' v.
```

An other advantage of using pretty big step semantics instead of a big step one is that it simplifies the way proofs are written. Indeed each intermediate function that represents one type of reduction have to be proven only once. This thus avoids the useless duplication of proofs.

For instance let us consider the example of the assignment rule given in Figure 9. Three functions are used there, thus three elimination lemmas will be used. Let us consider the one for `if_success`:

```
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 an argument the fact the computation stops, and give a disjunction: either `r0` was an error and has been returned, or it is not and there exists a result. When applying this lemma, the goal is split in two. In one case, the result was an error, which is not possible as it was supposed to be a result (see Line 2 of the theorem `run_correct`'s statement). 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
2       (fun (l : loc) (f : field) ⇒
3        if_success_value (run m h1 s e2)
4            (fun (h2 : jsheap) (v : val) ⇒out_return (update h2 l f v) v)) =
5     out_return h' v
```

The first step of computation of the assignment rule is thus performed quite easily and the proof can continue. In this case, the fact that the new heap `h1` is correct is given directly by the safety theorem, but this is not always the case. Indeed, when some writes are performed on heaps, the safety theorem does not always apply for those intermediary results. This has led to some code copied/pasted from the safety proof to the interpreter correctness proof. It may be interesting to change a little the safety proof's structure to factorise those proof steps.

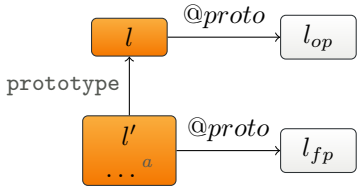
3.3 Completeness

The interpreter hasn't been proven complete during this internship: it is possible that it was supposed to give a result but did not. The main reason is that the heap manipulations are not deterministic (but the final result is).

Let us consider the simplest reduction rule that involves some allocations in the heap: the declaration of a new unnamed function. This rule is applied to construct a function in a JavaScript program such as `function(x){ return x; }`. Figure 12 shows the Coq rule for it and a graphical representation of its meaning.

```
1   | red_func_unnamed : ∀l l' h0 h1 h2 s lx e,
2       fresh h0 l →
3       h1 = alloc_obj h0 l loc_obj_proto →
4       fresh h1 l' →
5       h2 = alloc_fun h1 l' s lx e l →
6       red h0 s (exp_func None lx e) 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 expression `e`.

Figure 12: Allocating new unnamed function.

The predicate `fresh` is called two times. This predicate is defined by stating that the given location is not `null` and is not bound in the current heap. There is thus a lot of possibilities! In the

interpreter, the function `fresh_for` is defined as the minimum location number not yet allocated in the current heap.

To prove completeness, an equivalence relation over heaps is needed to make the reduction deterministic over equivalent heaps. This relation can be informally stated as "it is possible to injectively rename each non-special location of the heap without changing the semantics".

It is possible to express that in Coq, but it is some work. The proofs wouldn't be complex as the name of a given location is never checked (except some special locations such as `null` or l_g). It even seems automatisable.

The main problem is that such a construction is not currently automatised in Coq and would be too particular to the reduction rules currently chosen: the current formalisation is not finished. For instance differences between statements and expressions has not yet been formalised in Coq. As adding those difference may change a lot the way the reduction rules are defined, working on the structure of those reductions is not primary.

This lack of equivalent relation between heap has another consequence: it is not possible to formalise heap manipulation and optimisations. This forbids the interpreter to performs some garbage collection, which leads to unreadable generated heaps. Figure 13 shows a heap computed by the interpreter in a program with a lot of function calls (which is one case where locations are allocated only temporary). The red location in the center represents the global object l_g. All the locations not bound from l_g are temporary locations created by a function call.

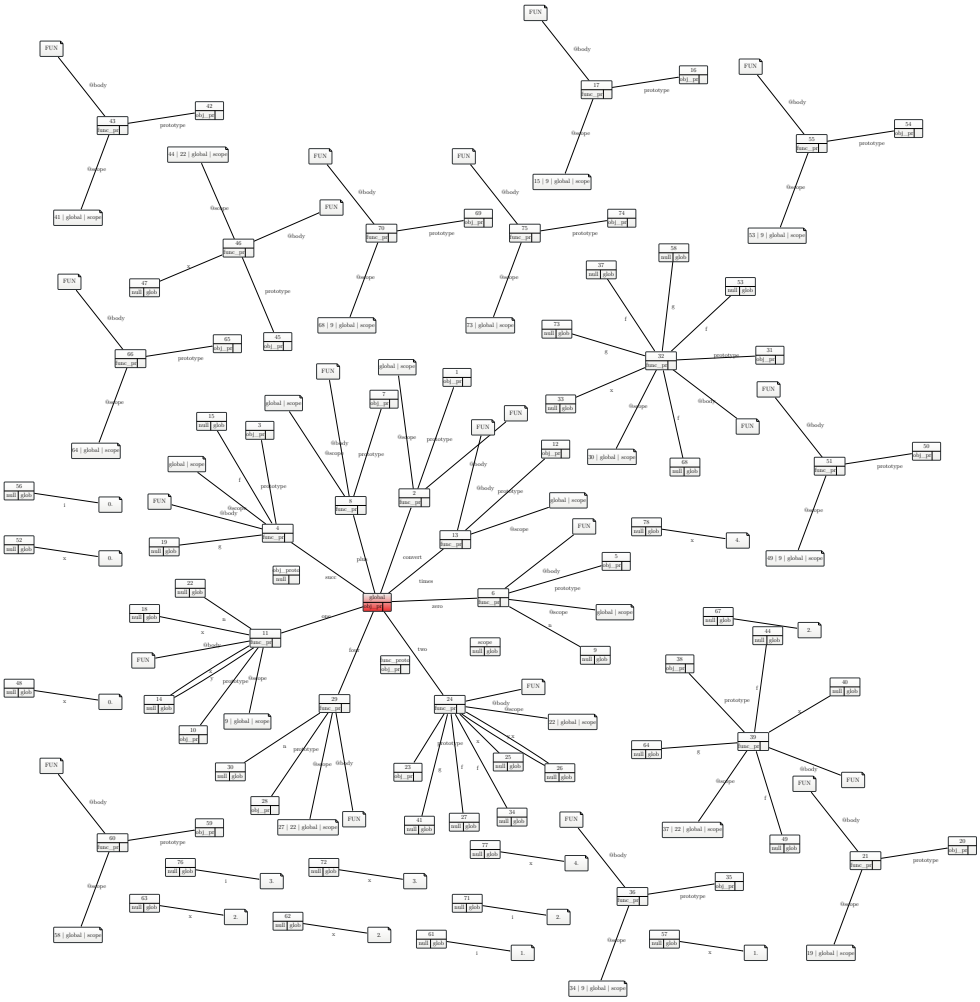


Figure 13: A heap computed by the interpreter in a program with a lot of function calls.

4 Future Work

This section shows some works that has not yet been done but that could be achieved within the scope of a PhD thesis.

4.1 Extensions of the Interpreter

The current interpreter strictly follows the formalised standard. In practise though, most real world interpreters are slightly different, typically by accepting reads or even writes on some (theoretically) unaccessible fields (typically @*proto* which is then called `__proto__`).

It is planed to add such features in the formalisation (and thus in the interpreter), but in a modular way. The idea is to parameter the interpreter by some flags describing which standard it should follow (ES3, ES5, Firefox's one, etc.). That way it will be possible to prove security of a given program for a given non-standard browser.

It's also possible to add a lot of JavaScript constructs in the formalisation and the interpreter not yet implemented. One of those extension will be to add exceptions and errors handling. This will require to rewrite a new semantics in pretty big step and to prove the equivalence with the old one. This last point is there in order to avoid having a trusted base written in pretty big step semantic, which add partially evaluated operators and is thus less easy to read (even if it would be nice ultimately to only work on a pretty big step semantic).

4.2 Real World Program Analysis

Now that a formalised semantics of JavaScript exists, it is possible to implement a separation logic analysis of JavaScript, such as the one introduced in [GMS12], in Coq. Such an analysis would be able to prove properties of real world programs.

It defines for each program statement $\mathcal{C}$ a Hoare triple $\{P\}\ \mathcal{C}\ \{Q\}$ where P and Q are predicates representing pre- and post-conditions over the heap and the returned result. The predicates P and Q are separation logic formulae. Inference rules are then given for each construction of JavaScript. Proving those inference rules in Coq with respect to the semantics will then prove correctness of an analyser based on these rules without changing the Coq trusted base (the JavaScript semantics written in Coq).

Proving separation logic rules can be interesting as [GMS12] introduces a new operator $\bowtie$ over heaps, which is difficult to handle as it allows a lot of things. Its meaning is between the one of the usual operators $\wedge$ and $\star$:

- The predicate $P \star Q$ states that the given heap h can be split in two disjoint heaps h_1 and h_2, h_1 satisfying P and h_2 satisfying Q.
- The predicate $P \wedge Q$ states that the given heap h satisfies both predicates P and Q. For instance, emp$\wedge T$, where emp is a predicate stating a heap is empty and T is the true predicate (accepting every heap—it can be useful to state that there is some garbage collected objects in the heap, or to state that there might be something else in the heap at some point).
- In between, $P \bowtie Q$ states that h can be split in two heaps h_1 and h_2 not necessarily disjoint, one satisfying P and the other Q.

Thus $\bowtie$ is a weakening of both $\star$ and $\wedge$. It has to be used in JavaScript because of prototype chains: two objects can share such a common chain. This implies that stating two locations l_1 and l_2 have both a correct prototype chain cannot be done with a predicate such as $\text{chain}\, l_1 \star \text{chain}\, l_2$, but have to be done with this new operator: $\text{chain}\, l_1 \bowtie \text{chain}\, l_2$. This operator is difficult to handle with and using Coq to work with it would be appreciated.

In Coq, the pre-condition P would be of type `jsheap` $\rightarrow$ `Prop` and the post-condition Q of type `result` $\rightarrow$ `jsheap` $\rightarrow$ `Prop` as the post-conditions have to give properties on the returned result of $\mathcal{C}$. Note that $\star$, $\wedge$ and $\bowtie$ are operations over *predicates* and not heap properties.

This formalisation could also be used to prove other analysers, such as the one described in Paper [TEM+11]: this analysis translates a JavaScript program to a list of reduction rules written in DataLog. These rules over-approximates the behavior of the input JavaScript program.

After this over-approximation, an analysis is performed on the DataLog to check if there exists a bad state that can be reached. Those bad states are usually states where a secret variable was given to an untrusted code.

The issue with their formalisation is that the translation from JavaScript to DataLog is relatively complex to understand. As JavaScript is now formalised in Coq, it would be now possible to prove that their translation is effectively an over-approximation of JavaScript's behavior.

4.3 A Bytecode for JavaScript

One difference between JavaScript and similar languages is that it does not have any standardised bytecode. Many developers are hostile to this idea of a JavaScript bytecode: it seems that program analysis is the only advantage of such a bytecode, but that the inconvenient are huge (for browser developers).

Java for instance is compiled to a bytecode that will be executed in a virtual machine. Such a bytecode, in addition to be faster to execute, allows the Java virtual machine to make a lot of code analysis (which are used in this case for optimisations).

Analysing bytecode is much easier than analysing a general code. It would for instance solve the parsing problem of JavaScript (see Section 2.4).

There exists a language relatively closed to JavaScript studied in the Celtique team by Vincent Monfort: ActionScript. It's a dialect of JavaScript specified by an extension of the ECMAScript norm. Excepting the `eval` operator, it is a superset of JavaScript that have a defined bytecode. In ActionScript, in addition to the scope chains and the prototype chains, there are classes which can also hide variables. A subset of ActionScript's bytecode thus seems to be sufficient, but it may be optimised for JavaScript.

This could simplify analysis as it would decrease the number of instruction, erasing the complex rules of parsing or evaluation order. It could also simplify the resolution of local variable's locations.

Conclusion

The now well spread language JavaScript has a semantics containing a lot of oddities. The main three are the way variables's name are resolved using scope chains and prototype chains, the way implicit type conversion can call arbitrary code in some environment, and the `eval` operator and all that it implies. Those oddities make analysis really hard to be performed.

In this internship, a semantics of a part of the JavaScript language has been formalised in Coq. An interpreter has been defined and proved correct with respect to this semantic. Some features (such as `eval` or type conversion) are not yet implemented, but it is enough to run relatively simple examples.

The main thing left to be done in this interpreter is the support of breaks in control flow (which happens with errors and the statements `return`, `break` and `continue`). This can be done using an alternative presentation for the semantics in-between small step and big step presentation: the pretty big step semantic, which avoid useless repetitions in the reduction rules.

The completeness of the interpreter has not been proven. It indeed requires some equivalent relations over heaps stating that the exact position of a given location is not important, as long as it is coherence all over the heap. Such a property could make the reduction rules deterministic with respect to this equivalent relation and thus allow to prove that the interpreter gives the same result as the reduction rules.

This work allows now to work on JavaScript using Coq, which is a first step for analysis real world programs. The advantage of being certified by Coq is that all analysis performed by a proved tool would be certified correct.

References

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

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

[Cha10a] A. Charguéraud. TLC: a non-constructive library for coq based on typeclasses. http://www.chargueraud.org/softs/tlc/, 2010.

[Cha10b] Arthur 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.

[Cha12] A. Charguéraud. Great-step semantics. 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.

[LBDT08] X. Leroy, S. Blazy, Z. Dargaye, and J.-B. Tristan. Compilers you can formally trust. http://compcert.inria.fr/, 2008.

[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.

[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. 2011.