A Certified JavaScript Interpreter

Martin Bodin
M2 ENS Lyon

Supervisors
Alan Schmitt
Thomas Jensen
INRIA Rennes

June 21, 2012

JavaScript Compared to Other Languages.

1. The "assembly language of the Internet";
2. A *scripting* language;
3. A precise norm of the language: ECMAScript 3;
4. A lot of features mimicking other languages's features:
   - Functional languages;
   - Imperative languages;
   - Prototype oriented languages.

My subject

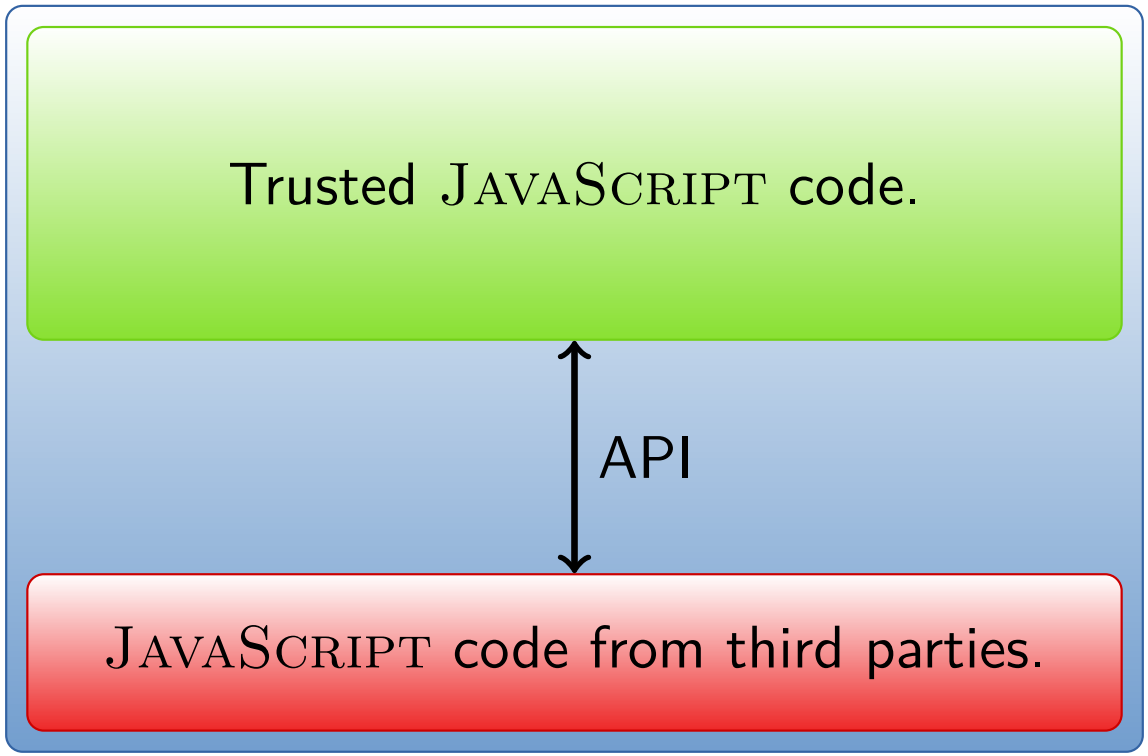
Create tools for formal analysis of JavaScript.

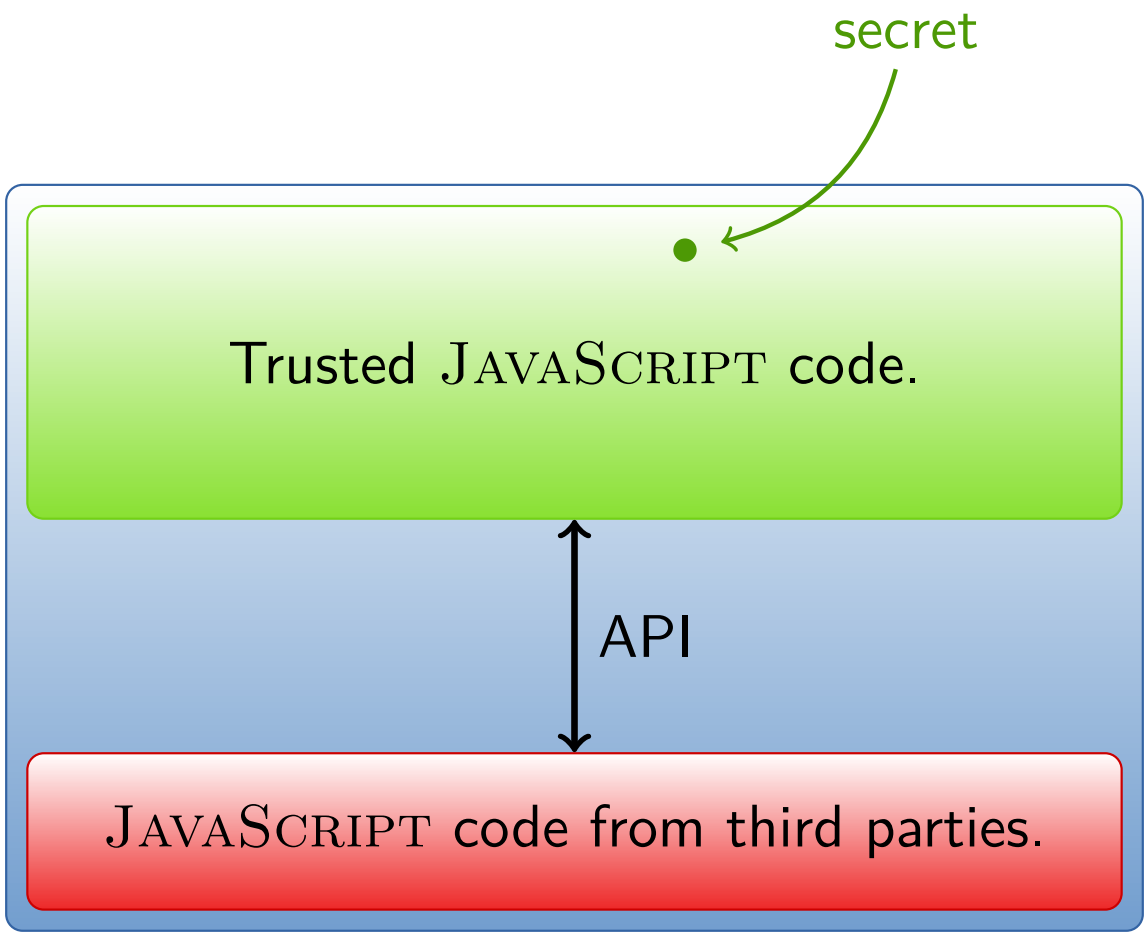
JavaScript Compared to Other Languages.

1. The "assembly language of the Internet";
2. A *scripting* language;
3. A precise norm of the language: ECMAScript 3;
4. A lot of features mimicking other languages's features:
   - Functional languages;
   - Imperative languages;
   - Prototype oriented languages.

My subject

Create tools for formal analysis of JavaScript.


Trusted JavaScript code.
API
JavaScript code from third parties.
Web page


secret
Trusted JavaScript code.
API
JavaScript code from third parties.
Web page

JavaScript's `var`. Counter-intuitive behaviour.

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

Dynamic scopes. Difficult to analyse.

```
1 function f(o) { with(o) { x = 0 } } ;
2 f({x : 1}) ; // x is undefined.
3 f({})        // x is defined.
```

JavaScript's `var`. Counter-intuitive behaviour.

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

Dynamic scopes. Difficult to analyse.

```
1  function f(o) { with(o) { x = 0 } } ;
2  f({x : 1}) ; // x is undefined.
3  f({})        // x is defined.
```

1. Implicit type conversion that can call arbitrary code;
2. The `eval` function;
3. Difficut to parse;
4. Dialects of JavaScript different in each browser.

Analyses would be very welcomed!

1. Implicit type conversion that can call arbitrary code;
2. The `eval` function;
3. Difficut to parse;
4. Dialects of JavaScript different in each browser.

Analyses would be very welcomed!

The heap

- Contains a lot of *locations*.
- Each locations contains some fields, associated to a *value*.
- Some special locations: `null`, l_g, `Object.prototype`, etc.
- Each writes are performed on the *same* heap!

Scope chain

- It's a stack of location.
- Similar to a call stack, with variations.

The heap

- Contains a lot of *locations*.
- Each locations contains some fields, associated to a *value*.
- Some special locations: `null`, l_g, `Object.prototype`, etc.
- Each writes are performed on the *same* heap!

Scope chain

- It's a stack of location.
- Similar to a call stack, with variations.

The heap

- Contains a lot of *locations*.
- Each locations contains some fields, associated to a *value*.
- Some special locations: `null`, l_g, `Object.prototype`, etc.
- Each writes are performed on the *same* heap!

Scope chain

- It's a stack of location.
- Similar to a call stack, with variations.

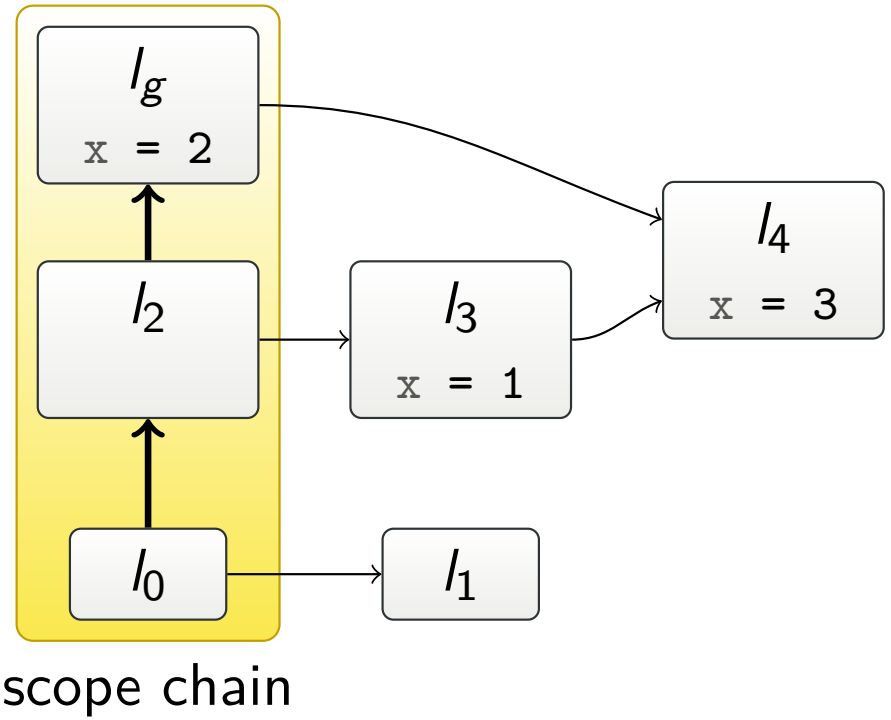
The heap

- Contains a lot of *locations*.
- Each locations contains some fields, associated to a *value*.
- Some special locations: `null`, l_g, `Object.prototype`, etc.
- Each writes are performed on the *same* heap!

Scope chain

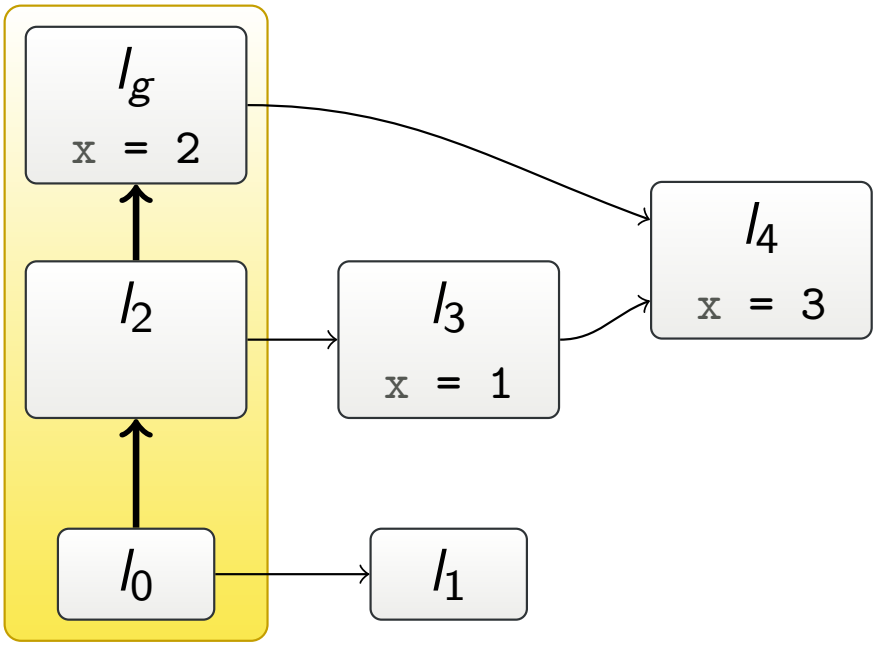
- It's a stack of location.
- Similar to a call stack, with variations.

Each bound location have a @*proto* field.



Reading Variables.

Each bound location have a @*proto* field.

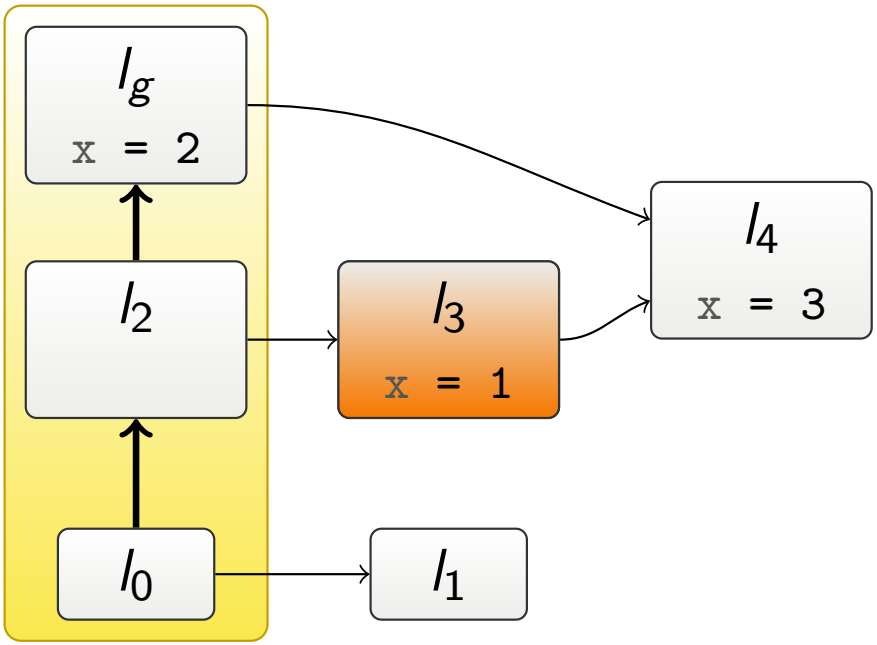


scope chain

```
1  var y = x ;
```

Reading Variables.

Each bound location have a @*proto* field.

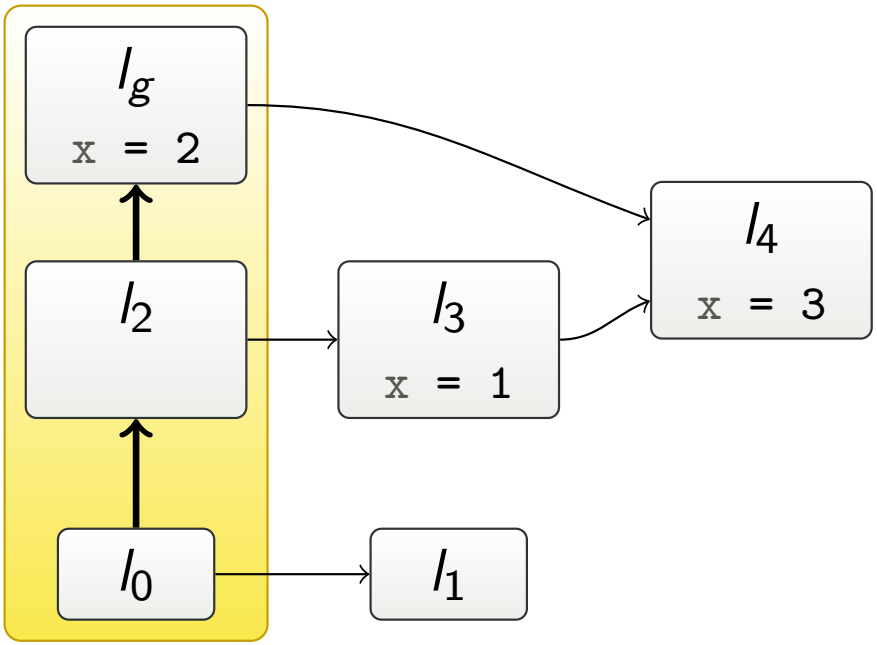


scope chain

```
1  var y = x ;
```

Writing Variables.

Each bound location have a @*proto* field.

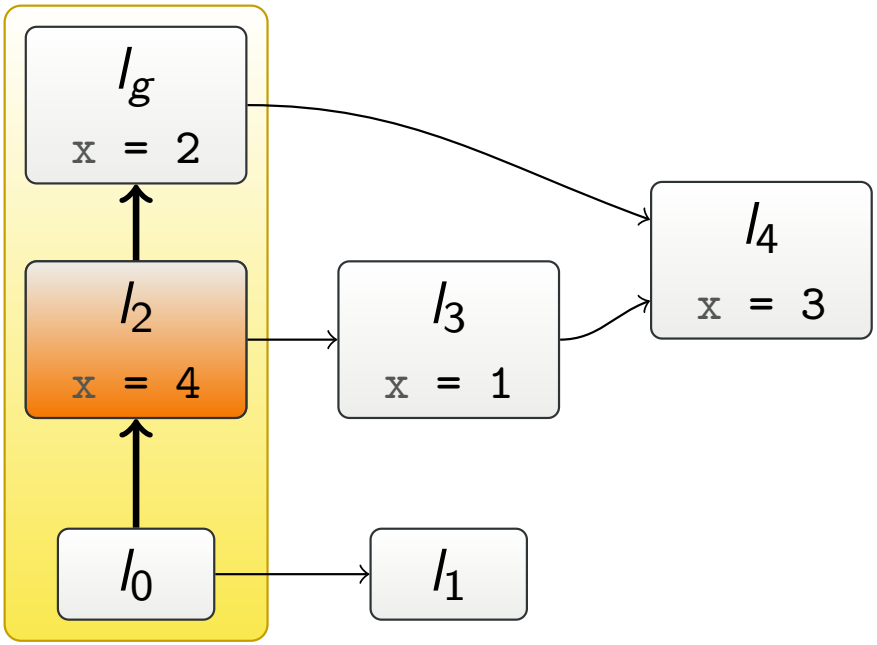


scope chain

```
1  x = 4 ;
```

Writing Variables.

Each bound location have a @*proto* field.



scope chain

```
1  x = 4 ;
```

- A Coq formalisation of JavaScript's semantic.
- Based on a small step semantic made by Sergio Maffeis.
- It uses the Flocq formalisation for IEEE floats.
- It uses TLC, a Coq library made by Arthur Charguéraud, which contains some decidability predicates, optimal fixed points, etc.

- A Coq formalisation of JavaScript's semantic.
- Based on a small step semantic made by Sergio Maffeis.

- It uses the Flocq formalisation for IEEE floats.
- It uses TLC, a Coq library made by Arthur Charguéraud, which contains some decidability predicates, optimal fixed points, etc.

```
H,l,v=va -> H,l,@PutValue(v,va) [E-Asgn]

Type(l1*m)=Reference
---------------------------------------- [R-PutValue-val]
H,l,@PutValue(l1*m,va) -> H,l,l1.@Put(m,va)
```

```
1  Inductive red : heap → scope → expr →
2                  heap → result → Prop :=
3                                    ⋮

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

File name	Size (Kio) Definitions	Proofs	Description
`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	

File name	Size (Kio) Definitions	Proofs	Description
`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	
Trusted base	30.5	0	

For every predicate in the semantic, the interpreter defines an equivalent function.

```
Inductive red : heap → scope → expr →
                heap → result → Prop :=

                        ⋮
```

```
Fixpoint run (max_step : nat) (h0 : heap) (s : scope)
             (e : expr) : out :=
  match max_step with
    | O ⇒out_bottom
    | S max_step' ⇒

                        ⋮
```

For every predicate in the semantic, the interpreter defines an equivalent function.

```
1  Inductive red : heap → scope → expr →
2                  heap → result → Prop :=
                                    ⋮
```

```
1  Fixpoint run (max_step : nat) (h0 : heap) (s : scope)
2               (e : expr) : out :=
3    match max_step with
4      | O ⇒out_bottom
5      | S max_step' ⇒
                                    ⋮
```

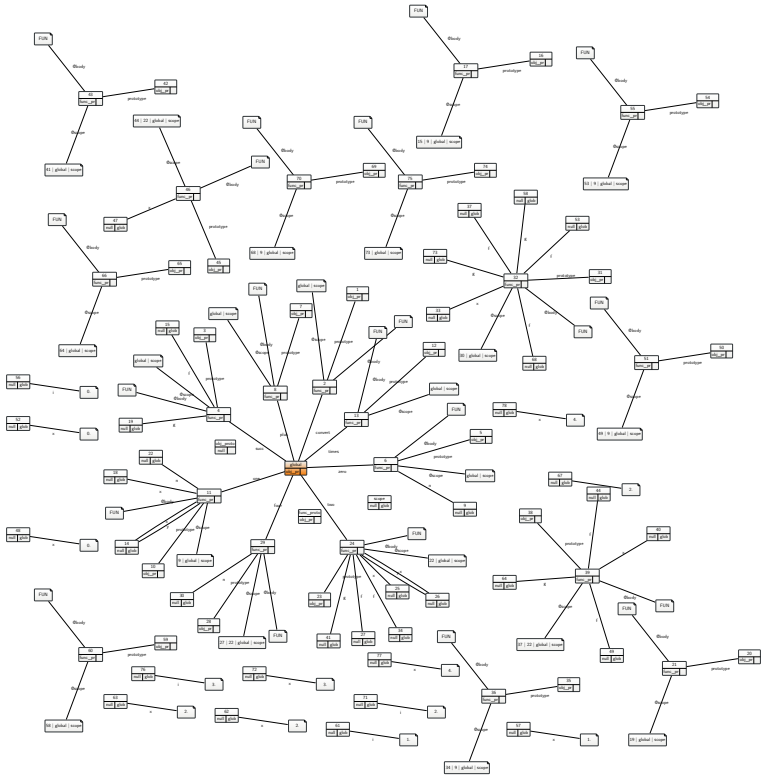
Rule expressed in the semantic

```
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
```

Rule expressed in the interpreter

```
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)))
```

It Now Works...



...But is it correct?

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

- For each sub-function, prove that it's correct according to its corresponding predicate.
- For each case of the proof, prove the correctness of intermediary heaps and scope chains. This can be done as in the safety proof.
- Combine all these lemmas to get the final result.

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

- For each sub-function, prove that it's correct according to its corresponding predicate.
- For each case of the proof, prove the correctness of intermediary heaps and scope chains. This can be done as in the safety proof.
- Combine all these lemmas to get the final result.

```
1   (* assign *)
2   forwards [(?&?) | (r1&h1&eq1)]: elim_if_success R; tryfalse.
3   rewrite eq1 in R. simpl in R.
4   forwards [(eqw&v'&eqv') | (l&f&eq')]: elim_if_is_ref R.
5   lets (h'0&eqw'): eqw. forwards*: wrong_not_ret eqw'.
6   rewrite eq' in R. simpl in R.
7   forwards [(?&?) | [(v0&h2&eq2&eqv0) | (v0&h2&b&eq2&eqv0)]]:
        elim_if_success_value R; tryfalse.
8   rewrite eq2 in R. simpls. rewrite eqv0 in R. simpls. forwards*:
        wrong_not_ret R.
9   rewrite eq2 in R. simpls. rewrite eqv0 in R. simpls.
10  inverts* R.
11  forwards* R1: run_correct eq1.
12  forwards* (OK1&OKL1&OKr1): sub_safety R1.
13  inverts* OKr1; tryfalse.
14  inverts H0.
15  forwards* R2: run_correct eq2.
16  forwards* (OK2&OKL2&OKr2): sub_safety R2.
17  apply* red_assign.
18  apply* getvalue_comp_correct.
```

Deconstruction of the interpreter's definition

Proof of correctness of intermediate results

Construction of the derivation's proof

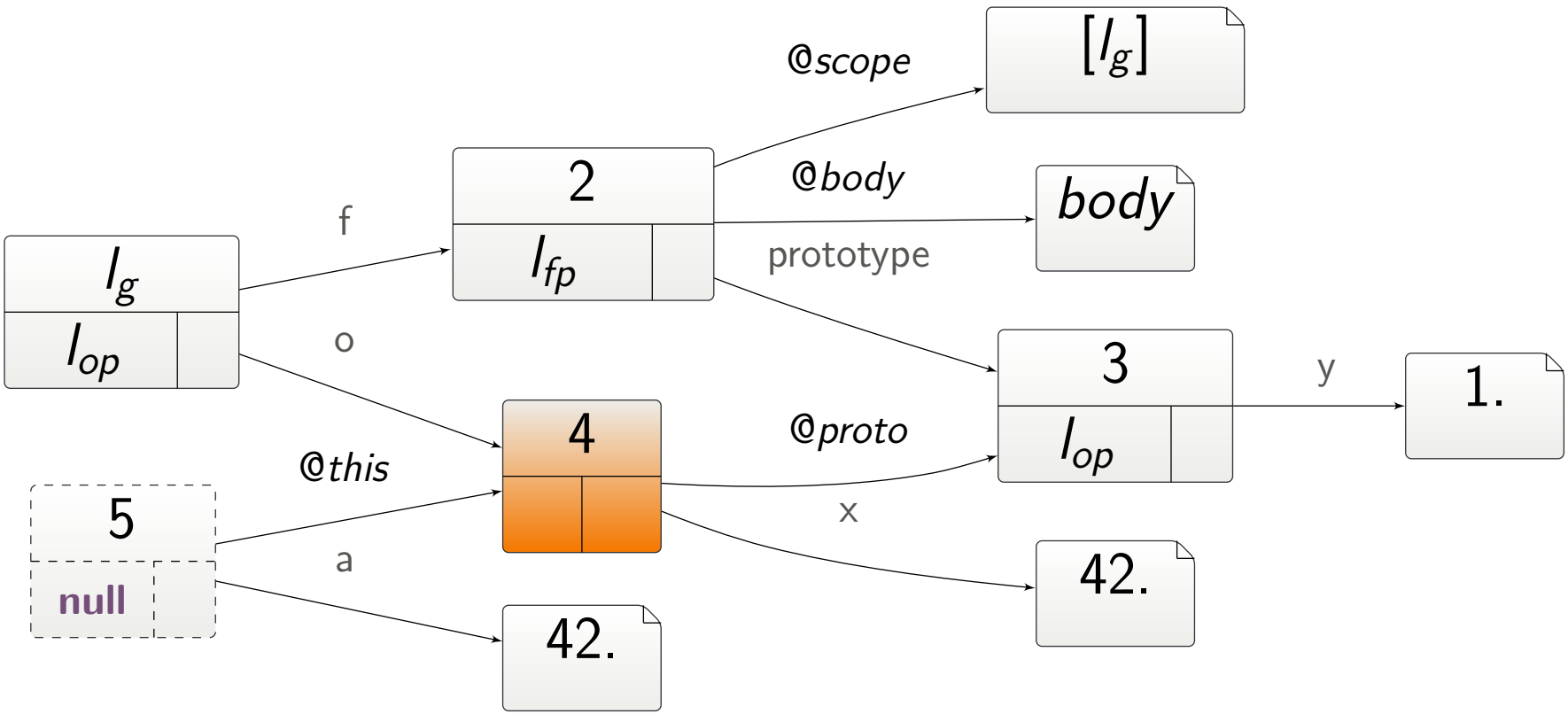
What Has Been Done So Far.

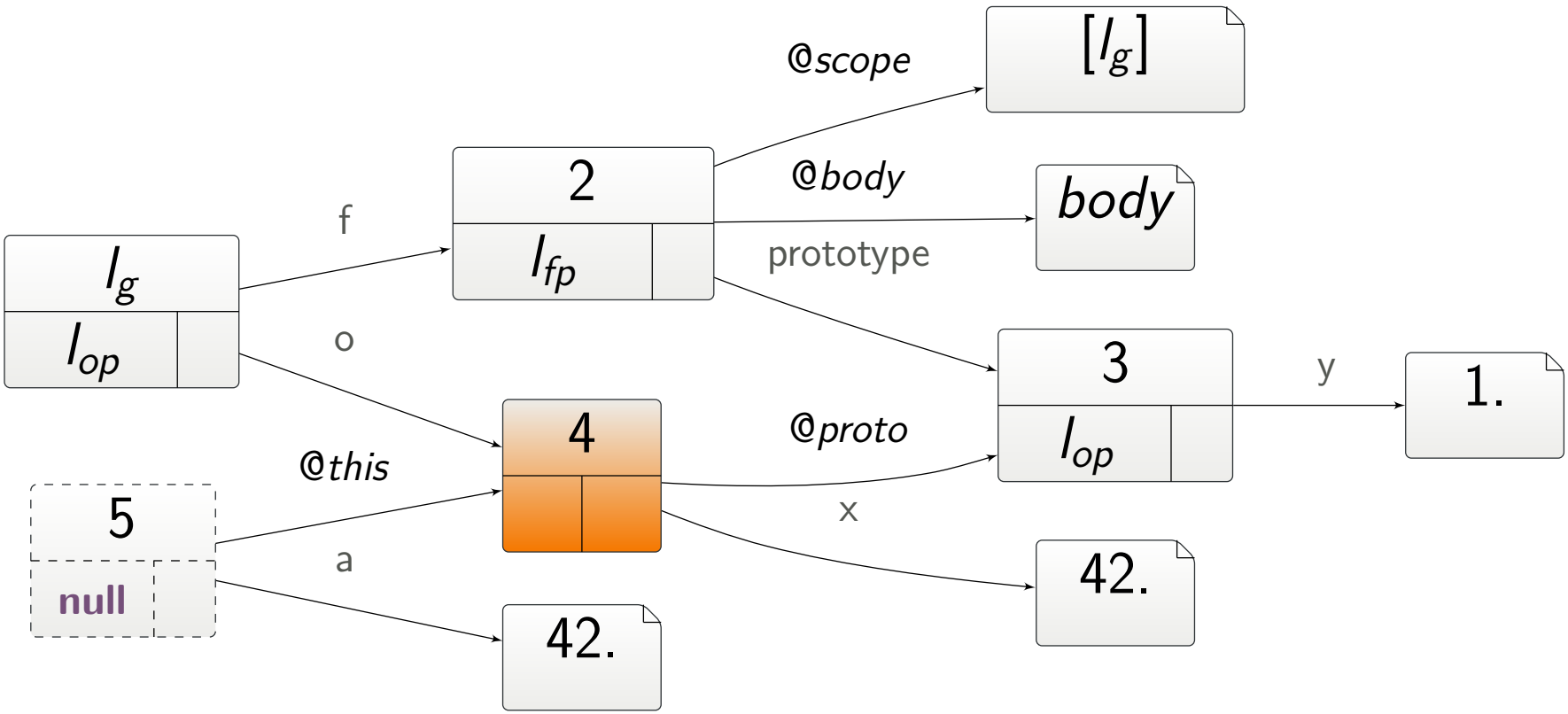
- A formalisation of JavaScript's semantic in Coq.
- Some properties about execution objects proven as being invariant.
- A JavaScript interpreter, proven correct.

Future Work.

- Extending the formalised semantic:
  - Support of exceptions and errors.
  - Support of other objects.
- A proof of completeness of the interpreter.
  - Some equivalence properties over heaps are needed.
- Proven analyses of JavaScript program.

Demo


@scope
$[l_g]$
2
l_{fp}
@body
body
prototype
f
l_g
l_{op}
o
3
l_{op}
y
1.
4
@proto
@this
5
null
a
x
42.
42.



Any questions?