JavaScript: de la standardisation à la formalisation

Alan Schmitt

3 juillet 2015

```
console.log((!![]+[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+
            (![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+
            (!![]+[])[+!+[]]][([][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+[+[]]]+
            (![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])[!+[]+!+[]+!+[]]+
            (!![]+[])[+!+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[])[+[]]+
            ([![]]+[][[]])[+!+[]+[+[]]]+(![]+[])[!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[]
```

Merci pour l'invitation

Pourquoi JavaScript ?

Un langage côté client
pour enrichir les pages web

Quelle confidentialité ?

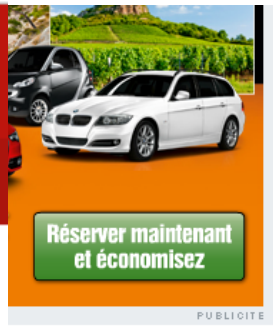
Pourquoi JavaScript ?

BBC | Sign in | News | Sport | Weather | Sh

NEWS

Home | Video | World | UK | Business | Tech | Science | Magazine

Technology



Spotify ads hit by malware attack

29 March 2011 | Technology

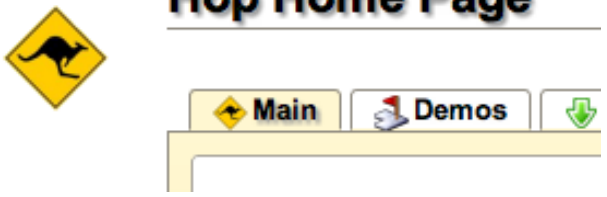
Malware ads hit London Stock Exchange Web site

Malware delivered by Yahoo, Fox, Google ads

Researchers at Avast are pointing fingers at large ad delivery platforms for serving up infected ads in new "malvertising" trend.

by Elinor Mills @elinormills / March 22, 2010 12:57 PM PDT

Un assembleur pour le Web



Emscripten

Emscripten is an LLVM-to-JavaScript compiler. It takes LLVM bitcode - which can be generated from C/C++, using llvm-gcc or clang, or any other language that can be converted into LLVM - and compiles that into JavaScript, which can be run on the web (or anywhere else JavaScript can run).

Un assembleur pour le Web

Hop Home Page

- BananaBread - Port of Sauerbraten/Cube 2. Demo site here. **additional levels added!**
- Graphviz - Graph visualization software (port is here) **new!**
- Me & My Shadow - A 2D SDL platform/puzzle game (original here) **optimized!**
- Heriswap - Port of a match-3 puzzle game.
- Ceferino - Port of a 2D action game (also available here).
- SuperTux WIP - Port of SuperTux. by forandom
- JS-VBA-M - Port of VBA-M. by ILOVEPIE
- Transport Tycoon Deluxe port of OpenTTD by caiiiycuk
- PNG Crush - PNG optimizer in a web page
- OpenGL ES 2.0 Gears - OpenGL ES 2.0 rendering compiled to WebGL.
- XML schema validation - XML validation in pure JS using compiled libxml.
- hpdf.js - Create PDF files in pure JS using compiled libharu. **new!**
- Box2D/WebGL - The Box2D physics engine compiled to JavaScript with convenient automatically-generated bindings (through box2d.js)
- SQLite - SQLite compiled to JavaScript with an easy-to-use API (through sql.js)
- Bullet/WebGL - The Bullet physics engine compiled to JavaScript with convenient automatically-generated bindings (through ammo.js)
- Text-to-Speech - eSpeak, a speech synthesizer, compiled to JavaScript.
- Ray tracing - A simple C++ ray tracer, rendering to a canvas element
- Python, Ruby, Lua - The popular dynamic languages Python, Ruby and Lua, compiled to JavaScript.
  - Older Python demo
- Poppler - PDF rendering in JavaScript, using Poppler and FreeType. Warning: Very large (>12MB) download.
- OpenJPEG - JPEG 2000 decoding in JavaScript, using OpenJPEG (see also j2k.js)
- FreeType - TrueType font rendering in JavaScript, using FreeType
- Lua - The Lua interpreter

Emsci

Emscripten is

any other language that can be converted into LLVM - and compiles that into JavaScript, which can be run on the web (or anywhere else JavaScript can run).

Humble Bundle

Support Blog aaron

Humble Bundle | Weekly Bundle | Books Bundle | Mobile Bundle | Flash Bundle | Humble Store

Humble mozilla Bundle

Powered by asm.js

Pay What You Want!

Time is running out 17 : 23 : 12 : 08

Play in your browser | Redeem on Steam | Pay what you want | Support charity | 0 0 0 0 2 0 Bundles sold

Pay more than the average to unlock!

Pay $8 or more to unlock!

Play	Play	Play	Play	Play	Now Playing	Play		Play
Super Hexagon	Aaaaa...!!! for the Awesome	Osmos	Zen Bound 2	Dustforce DX	Voxatron	FTL: Faster Than Light	Another game coming soon!	Democracy 3

Pourquoi s'intéresser à JavaScript?

1. JavaScript est partout sur le web
2. JavaScript est utilisé pour manipuler nos données

Le langage

Un riche langage impératif et fonctionnel

Variables « classiques »

```
var x = 4
x = (10 * 4) + 2
console.log(x)
```

$\Rightarrow$ 42

Les fonctions sont des valeurs

```
var f = function (g,x) {return (g(x) + 2)}

var fgx = f(function (y){return (10 * y)}, 4)

console.log("f(g,x) = " + fgx)
```

$\Rightarrow$ f(g,x) = 42

Objets en JavaScript

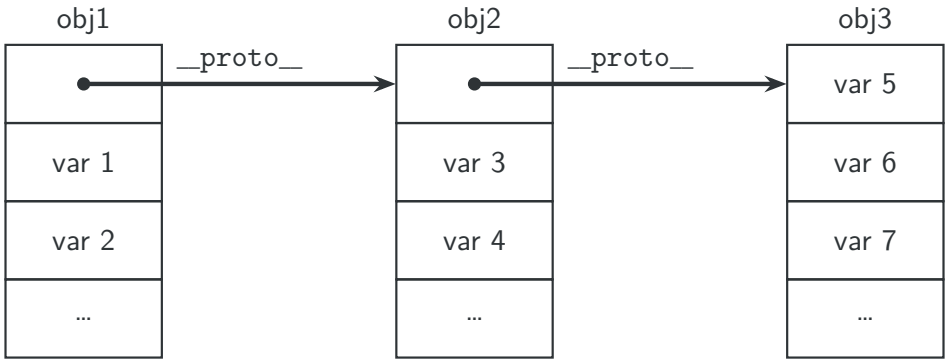
Objets litéraux

```
var obj = { a : 1, b : 2 } /* litéral */
console.log (obj.a)        /* accès   */        ⇒ 1
```

Les fonctions comme usines à objets

```
function f(a) { this.x = a }
var o = new f(42)
console.log (o.x)                               ⇒ 42
```

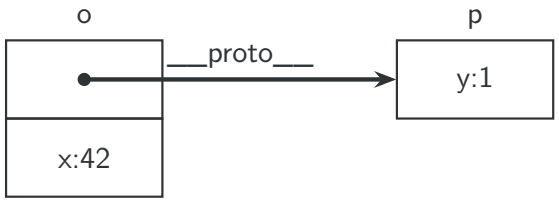
Un langage qui manque de classes



Prototypes et usines

```
function f(a) { this.x = a }
var p = {y : 1}
f.prototype = p

var o = new f(42)

console.log("o.x = " + o.x + ", o.y = " + o.y)
```

$\Rightarrow$ `o.x = 42, o.y = 1`



Attention: le `prototype` explicite d'une fonction devient le champ `__proto__` de l'objet créé.

Un langage dynamique

Le code html

```
<div id="orgquote">
  <script type="text/javascript" src="http://orgmode.org/org-quotes.js"></script>
</div>
```

télécharge le script JavaScript lié, et l'exécute

Un langage dynamique

Le code html

```
<div id="orgquote">
  <script type="text/javascript" src="http://orgmode.org/org-quotes.js"></script>
</div>
```

télécharge le script JavaScript lié, et l'exécute

Cette fonctionalité est dans le langage lui-même

```
eval("console.log(\"Hello World!\")")
```

$\Rightarrow$ `Hello World!`

Un langage sans erreurs

Règles de syntaxes complexes + conversions automatiques $\Rightarrow$ 

Blocs vs Objets

```
var x = eval( "{} + {}" )
var y = eval("({} + {})")
console.log("x = " + x + "; y = " + y)
```

$\Rightarrow$ `x = NaN; y = [object Object][object Object]`

JSf*ck

```
var x = (![]+[])[+!+[]]
      + (![]+[])[!+[]+!+[]]
      + (![]+[])[+!+[]]
      + ([][[]]+[])[+!+[]]
console.log(x)
```

$\Rightarrow$ `alan`

Un langage sans erreurs

Rè

Bl

```
va
va
co
```

⇒

JS

```
va
co
```

⇒

```
failbowl:~(master!?) $ jsc
> Array(16)
,,,,,,,,,,,,,,,
> Array(16).join("wat")
watwatwatwatwatwatwatwatwatwatwatwatwatwatwat
> Array(16).join("wat" + 1)
wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1wat1
wat1
> Array(16).join("wat" - 1) + " Batman!"
NaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaN Batman!
> 
```

Wat

@garybernhardt

Un langage sans erreurs

Règles de syntaxes complexes + conversions automatiques $\Rightarrow$ 

Blocs vs Objets

```
var x = eval( "{} + {}" )
var y = eval("({} + {})")
console.log("x = " + x + "; y = " + y)
```

$\Rightarrow$ `x = NaN; y = [object Object][object Object]`

JSf*ck

```
var x = (![]+[])[+!+[]]
      + (![]+[])[!+[]+!+[]]
      + (![]+[])[+!+[]]
      + ([][[]]+[])[+!+[]]
console.log(x)
```

$\Rightarrow$ `alan`

(![]+[])[+!+[]] est 'a'

`[]`	tableau vide	`[]`
`![]`	négation (conversion en booléen)	`false` [1]
`![]+[]`	concaténation (conversion en chaîne)	`"false"`
`[]`	tableau vide	`[]`
`+[]`	conversion en entier	`0`
`!+[]`	négation	`true`
`+!+[]`	conversion en entier	`1`
`(![]+[])[+!+[]]`	accès à un élément	`'a'`

[1] Tout est `true` sauf `false`, `0`, `NaN`, `""`, `null` et `undefined`

Conversion et code utilisateur

```
var o = {}

o.toString = function () {
  o.toString = function () { return "😈" }
  return "😇"
}

console.log("je teste              : " + o)
console.log("c'est bon, j'utilise : " + o)
```

$\Rightarrow$

```
je teste              : 😇
c'est bon, j'utilise : 😈
```

JavaScript et les navigateurs

Intégration aux pages web

- navigation

```
<input action="action" type="button" value="Back"
       onclick="history.go(-1);" />
```

- modification du contenu (DOM)

```
document.title = "Nouveau titre"
var para = document.createElement("p")
para.appendChild(document.createTextNode("Bonjour le monde !"))
document.getElementsByTagName("body")[0].appendChild(para)
```

La boucle d'événements

- JavaScript est un langage séquentiel
- Utilisation pervasive de *callbacks*

Pourquoi vérifier JavaScript?

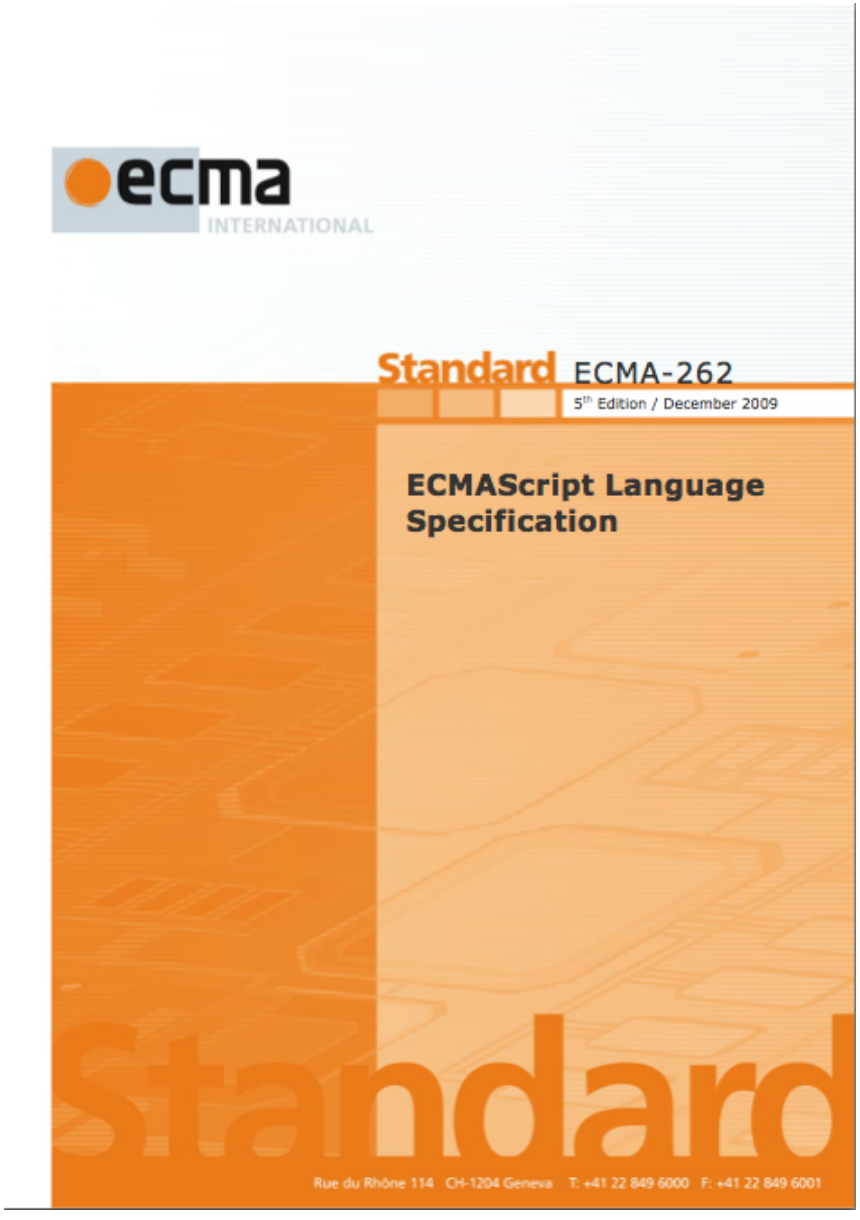
1. JavaScript est partout sur le web
2. JavaScript est utilisé pour manipuler nos données
3. JavaScript est compliqué
4. JavaScript a une spécification

Le standard

La spécification

ECMA-262

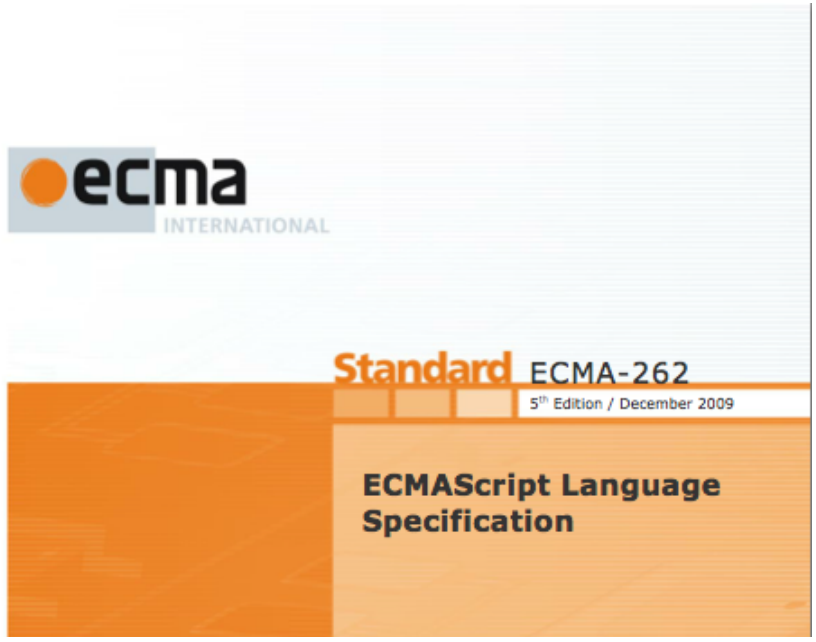
- 252 pages
- Détaillé, en langue naturelle



La spécification

ECMA-262

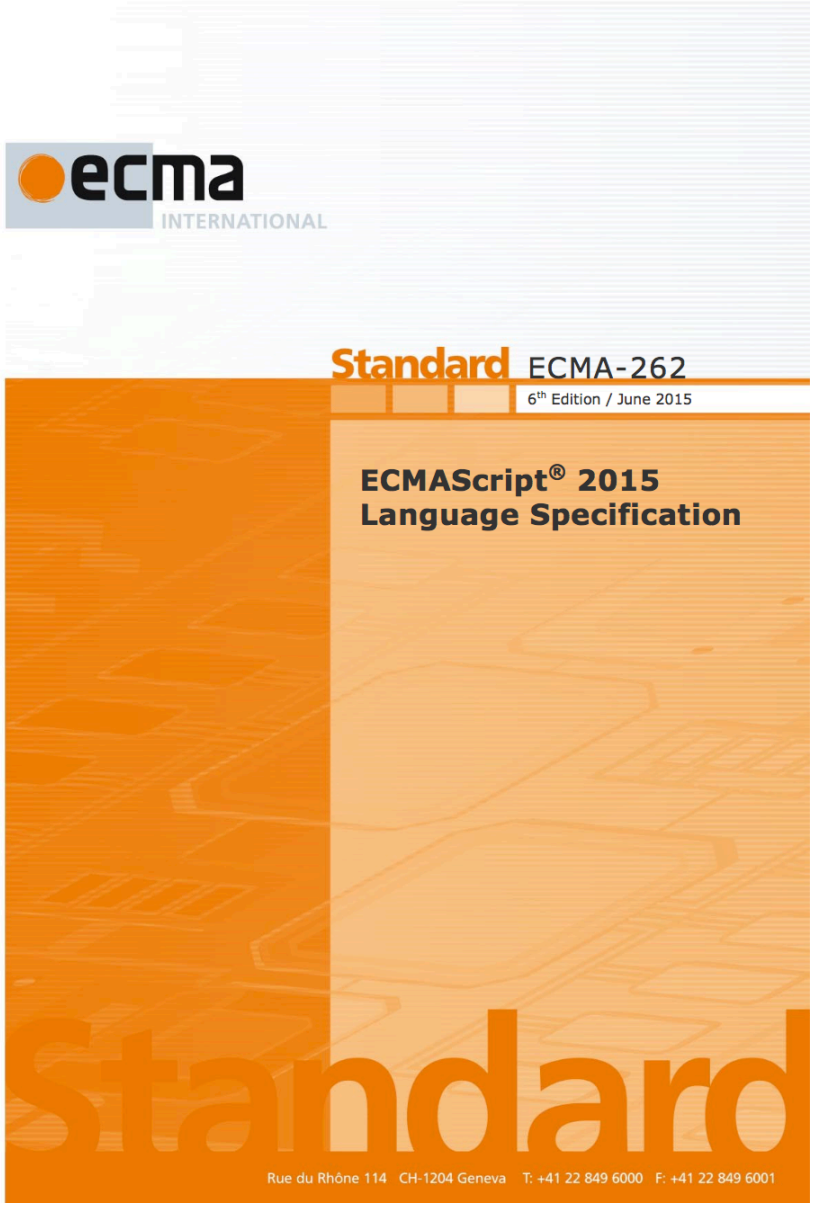
- 252 pages
- Détaillé, en langue naturelle



The production *IterationStatement* : **`while`** **`(`** *Expression* **`)`** *Statement* is evaluated as follows:

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

ECMAScript 2015



Nouveautés

- Constantes, variables avec portée de block (`let`)
- Symboles
- Classes
- Modules
- Itérateurs et générateurs (`yield`)
- Promesses
- Proxies
- Réflection
- 566 pages

Le comité de standardisation (membres industriels)

Le comité de standardisation (membres à but non lucratif)

Le processus de standardisation

- 6 réunions par an, dont une en Europe
- une trentaine de personnes
- discussion par email (`es-discuss@mozilla.org`)
- propositions publiques (`https://github.com/tc39/ecma262`)

Current Proposals

ES7+ Proposals follow this process document.

🚀	Proposal	Champion	Stage
	Object.observe	Erik Arvidsson	2
	Exponentiation Operator	Rick Waldron	2
	Array.prototype.includes	Domenic Denicola, Rick Waldron	2
	SIMD.JS - SIMD APIs + polyfil	John McCutchan, Peter Jensen, Dan Gohman	2
	function.sent metaproperty	Allen Wirfs-Brock	2
	Async Functions	Brian Terlson	1
	Parallel JavaScript	Tatiana Shpeisman, Niko Matsakis	1

Les implémentations précèdent la spécification

Accès au prototype

```
function f() {}
f.prototype = { y : 2 }

var x1 = new f()
var x2 = new f()

console.log("Before: x1.y = " + x1.y + "; x2.y = " + x2.y)

x1.__proto__.y = 3

console.log("After:  x1.y = " + x1.y + "; x2.y = " + x2.y)
```

$\Rightarrow$

```
Before: x1.y = 2; x2.y = 2
After:  x1.y = 3; x2.y = 3
```

Les implémentations précèdent la spécification

Accès au prototype

```
funct
f.pro

var x
var x

console.log( Before: x1.y =  + x1.y +  ; x2.y =  + x2.y)

x1.__proto__.y = 3

consc
```

⇒ Be
Af

19.1.2.9 Object.getPrototypeOf (O)

When the **`getPrototypeOf`** function is called with argument O, the following steps are taken:

1. Let *obj* be ToObject(O).
2. ReturnIfAbrupt(*obj*).
3. Return *obj*.[[GetPrototypeOf]]().

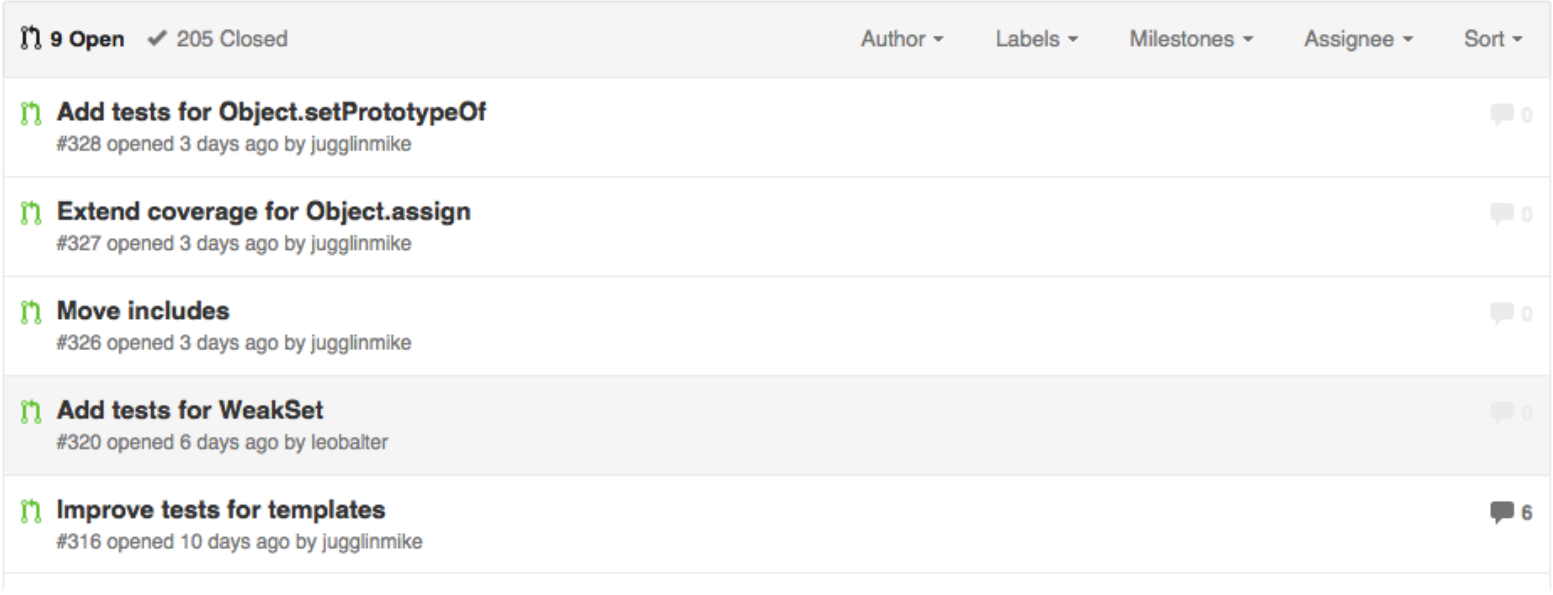
9.1.1 [[GetPrototypeOf]] ( )

When the [[GetPrototypeOf]] internal method of O is called the following steps are taken:

1. Return the value of the [[Prototype]] internal slot of O.

Test262

- `https://github.com/tc39/test262`
- 13686 tests
- approche collaborative



Vérification de JavaScript

Comment faire confiance à son programme ?

- Le relire soigneusement
  - ne passe pas à l'échelle

Comment faire confiance à son programme ?

- Le relire soigneusement
  - ne passe pas à l'échelle
- Le tester
  - peut rater des bugs

Comment faire confiance à son programme ?

- Le relire soigneusement
  - ne passe pas à l'échelle
- Le tester
  - peut rater des bugs
- Utiliser un vérificateur automatique (système de types)
  - trouve tous les bugs s'il est correct

ADSafety

- ADSafe: *sandbox* pour publicités

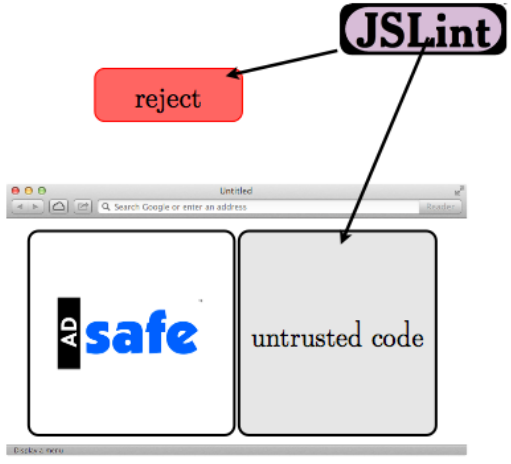


Figure 2: Architecture of ADsafe

ADSafety

- ADSafe: *sandbox* pour publicités

Write a program in the form

```
(function () {
...
})();
```

where the ... is replaced by code that calls the alert function when run on any browser. If the program produces no errors when linted with the ADsafe option, then I will buy you a plate of shrimp.

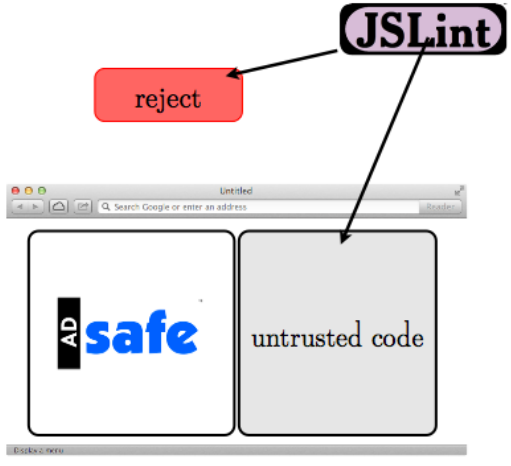


Figure 2: Architecture of ADsafe

ADSafety

Typed-Based Verification of Web Sandboxes

Joe Gibbs Politz
Brown University
Providence, RI 02912

Arjun Guha*
University of Massachusetts
Amherst, MA 01003

Shriram Krishnamurthi
Brown University
Providence, RI 02912

February 23, 2014

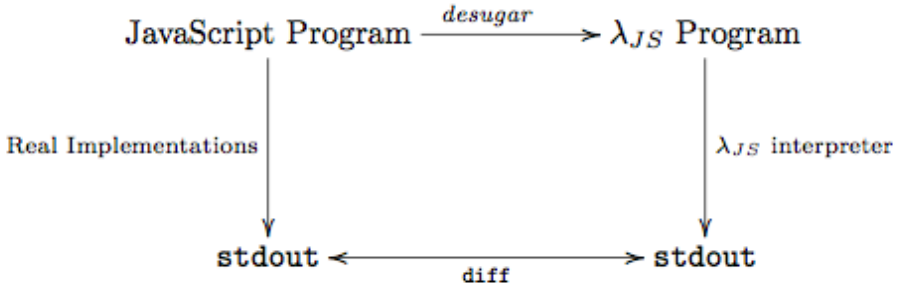


Figure 6: Testing strategy for λ_{JS} [26]

- ADSafe: *sandbox* pour publicités
- Vérification du code d'une avec un système de types.
- La correction du système est prouvée par λJS.

Write a program in the form

```
(function () {
...
})();
```

where the ... is replaced by code that calls the alert function when run on any browser. If the program produces no errors when linted with the ADsafe option, then I will buy you a plate of shrimp.

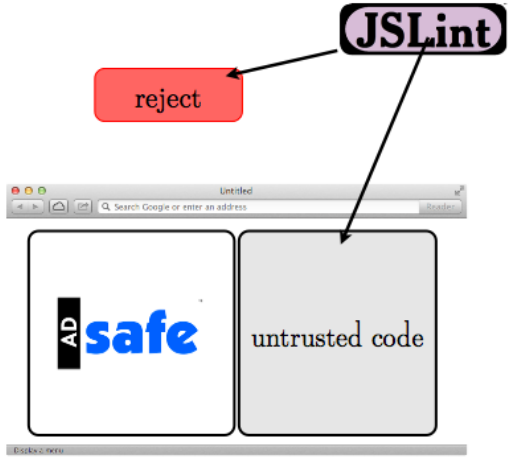


Figure 2: Architecture of ADsafe

ADSafety

Typed-E

gram

Joe Gi
Brown
Providen

JS interpreter

ıt

-
-
-

JSLint

Write a

(functio

...

})();

sted code

where th
run on a

the ADsafe option, then I will buy you a plate of shrimp.

Figure 2: Architecture of ADsafe

Notre slogan

Qui vérifiera les vérificateur eux-mêmes ?

Comment vérifier un vérificateur ?

Une idée toute simple

Programmer et prouver le vérificateur dans le même langage

Quel langage ?

Coq, un animal aux deux visages

Premier visage

- un outil pour prouver des propriétés

Second visage

- un langage de programmation fonctionnel avec un système de types très riche

```
sort: forall l: list int,
                { l': list int | Sorted l' /\ PermutationOf l l' }
```

- et avec un mécanisme d'extraction vers Ocaml

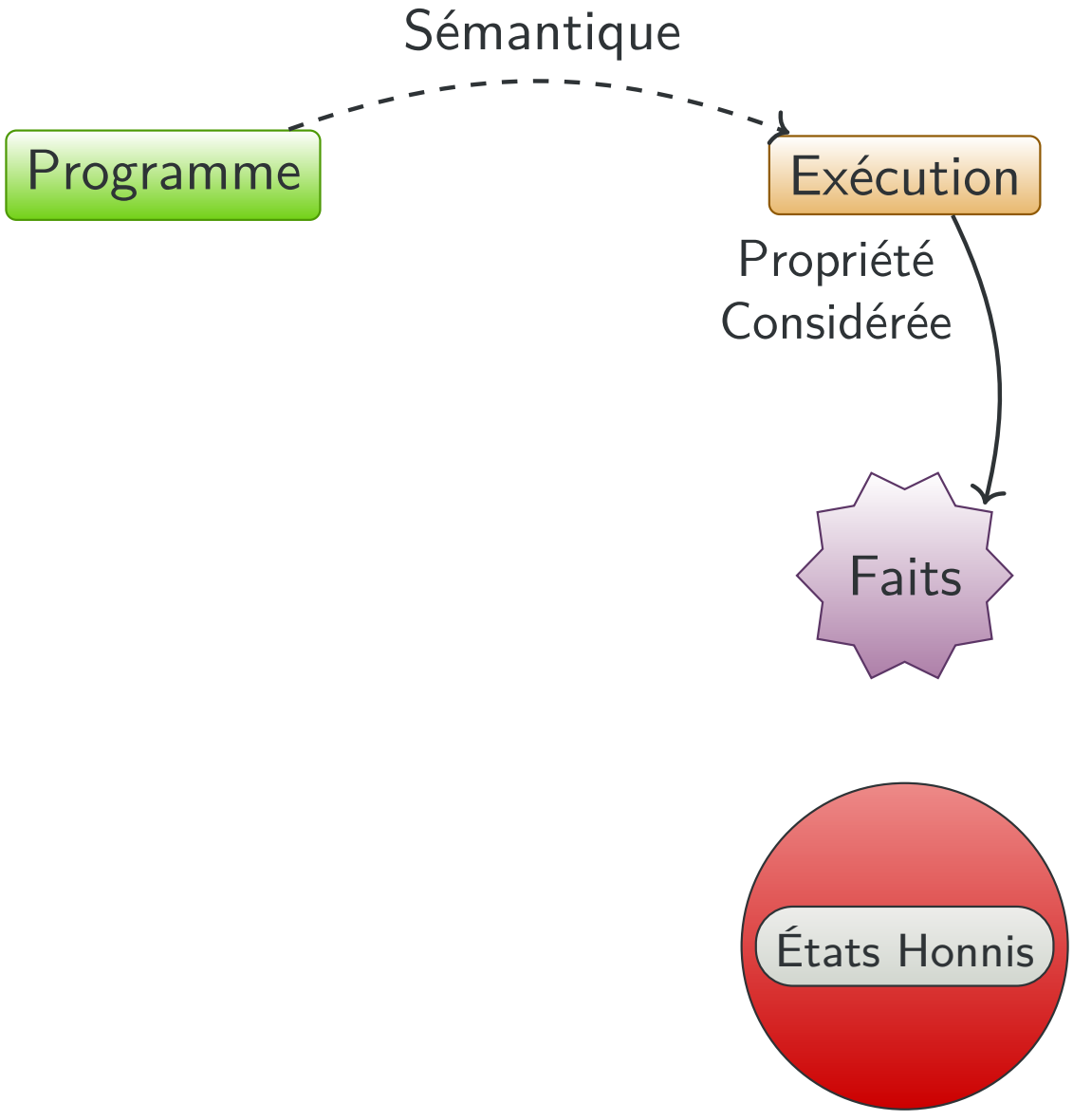
```
sort: int list -> int list
```

Méthode

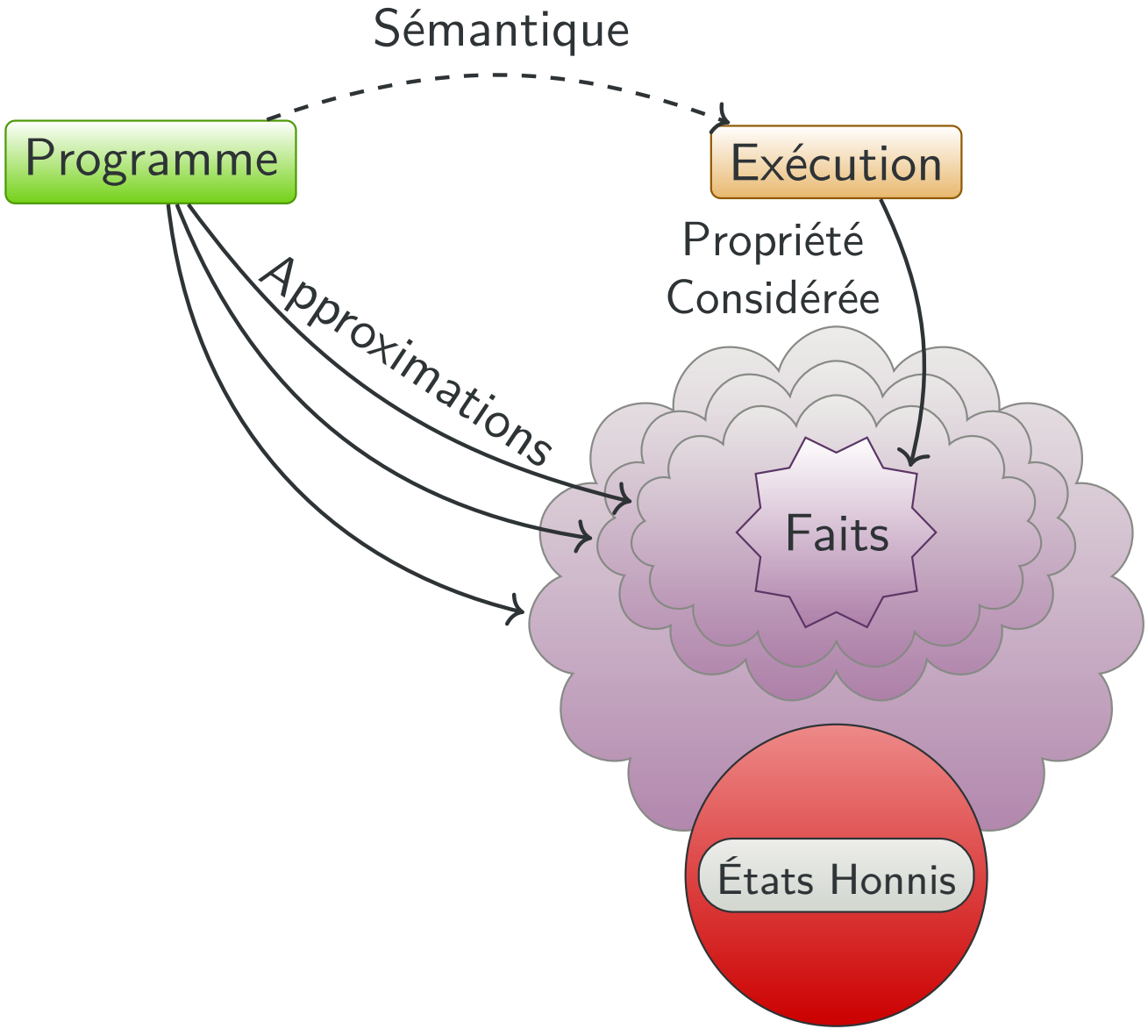
1. Définir une sémantique formelle de JavaScript
2. Définir la propriété nous intéressant
3. Écrire une analyse
4. Prouver que l'analyse est correcte

Approche globale

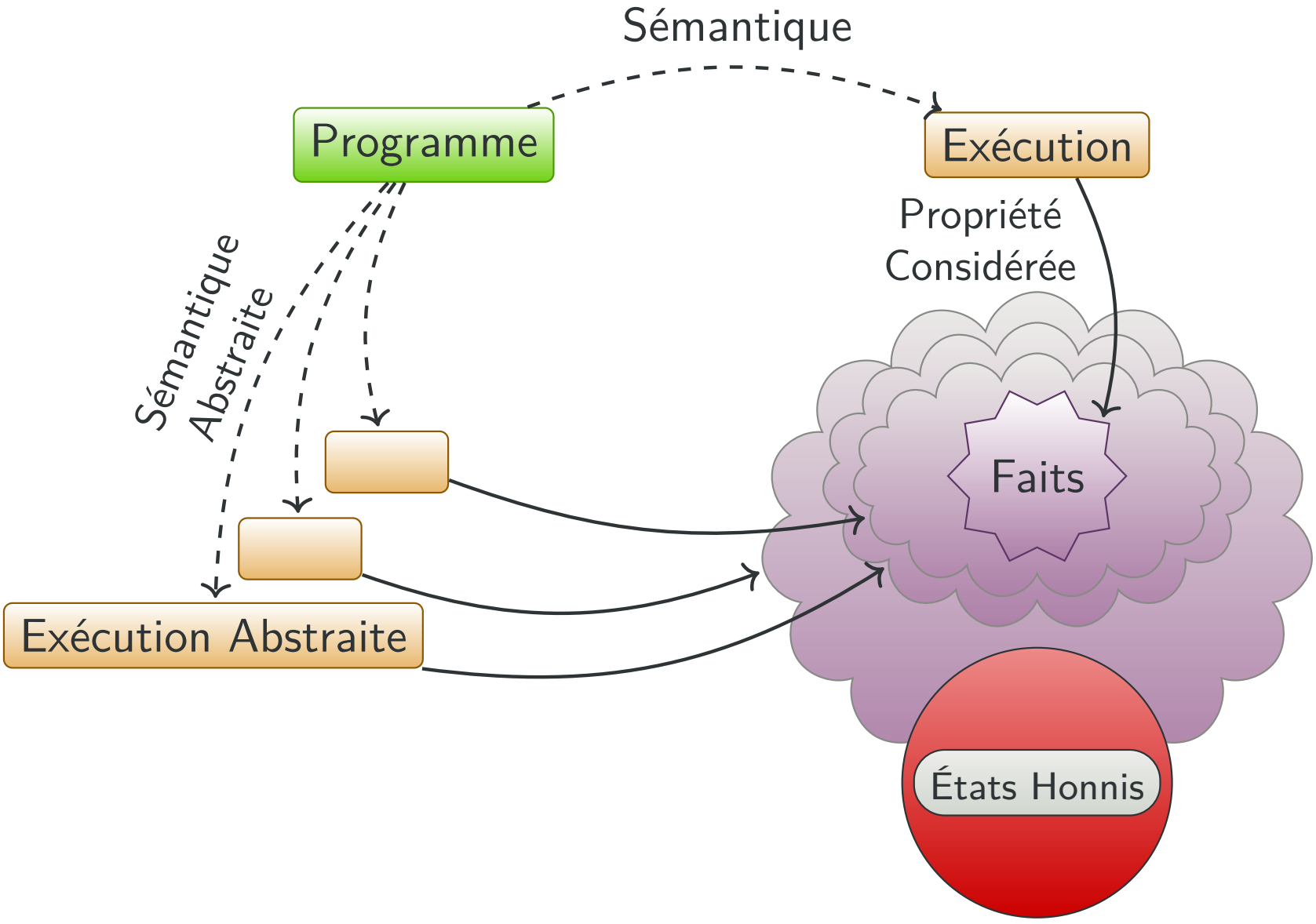
Approche globale



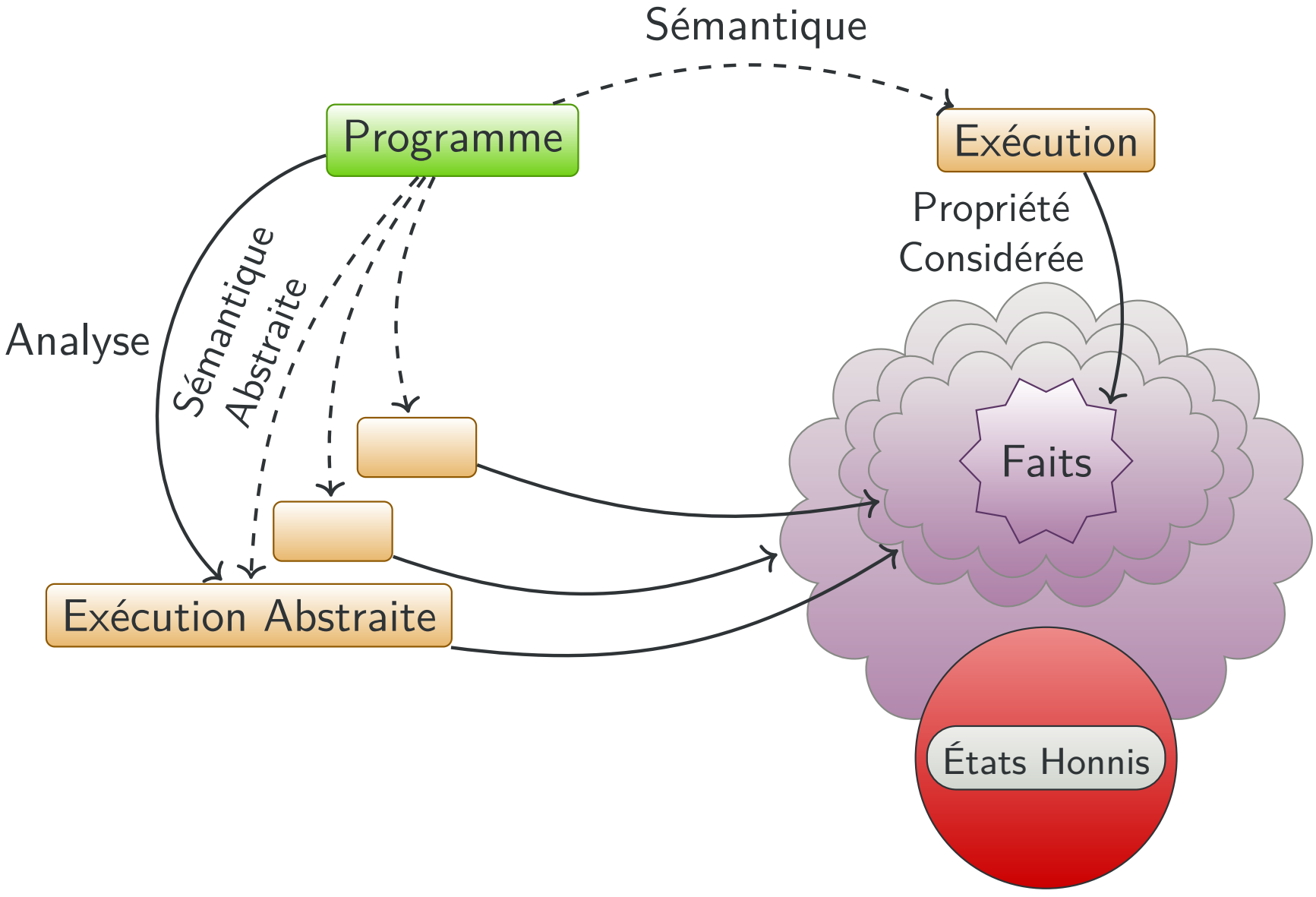
Approche globale



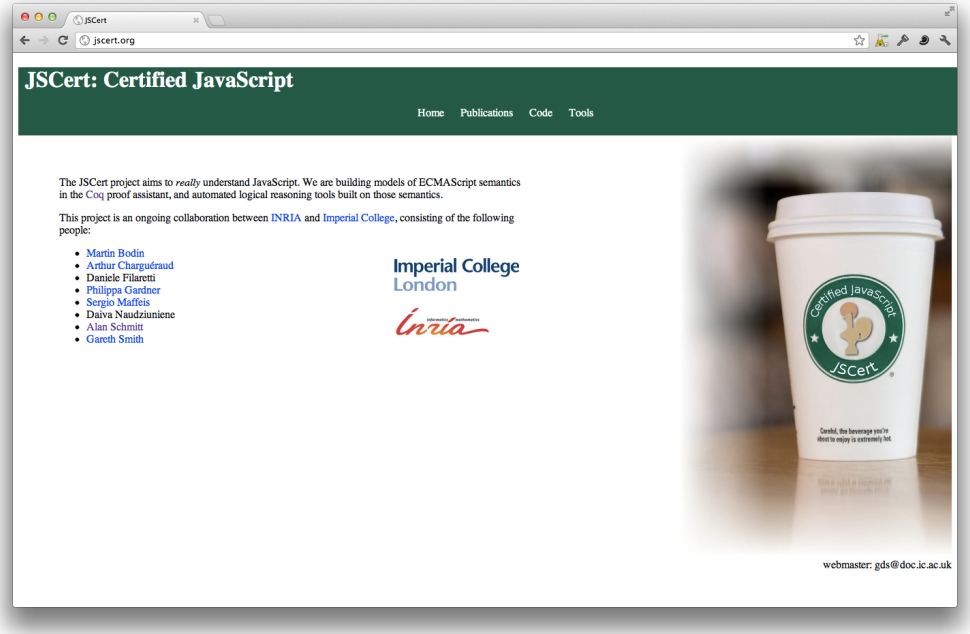
Approche globale



Approche globale



Le projet JSCert



- Martin Bodin
- Arthur Charguéraud
- Daniele Filaretti
- Philippa Gardner
- Sergio Maffeis
- Marek Materzok
- Daiva Naudžiūnienė
- Alan Schmitt
- Gareth Smith
- Thomas Wood

Comment s'assurer qu'on a bien capturé le langage ?

Rester proche de la spécification papier

`jssec.net`

Comment s'assurer qu'on a bien capturé le langage ?

Rester proche de la spécification papier

`jssec.net`

Tester la formalisation

λ_{JS}

Comment s'assurer qu'on a bien capturé le langage ?

Rester proche de la spécification papier

`jssec.net`

`jscert.org`

Tester la formalisation

λ_{JS}

Les formalisations

Proche de la spécification

12.6.2 The `while` Statement

The production *IterationStatement* **`: while (`** *Expression* **`)`** *Statement* is evaluated as follows:

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

```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv') o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_4_not_continue : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_continue /\ res_label_in R labs) ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_while_6_normal : forall S C labs e1 t2 rv R o,
    res_type R = restype_normal ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o

| red_stat_abort : forall S C extt o,
    out_of_ext_stat extt = Some o ->
    abort o ->
    ~ abort_intercepted_stat extt ->
    red_stat S C extt o
```

Sémantique de while

1. Let $V =$ empty.
2. Repeat
    1. Let *exprRef* be the result of evaluating `Expression`.
    2. If *ToBoolean*(*GetValue*(*exprRef*)) is `false`, return (*normal*, *V*, *empty*).
    3. Let *stmt* be the result of evaluating `Statement`.
    4. If *stmt.value* is not `empty`, let $V =$ *stmt.value*.
    5. If *stmt.type* is not `continue` or *stmt.target* is not in the current label set, then
        1. If *stmt.type* is `break` and *stmt.target* is in the current label set, then Return (*normal*, *V*, *empty*).
        2. If *stmt* is an abrupt completion, return *stmt*.

Sémantique de while

1. Let $V =$ empty.

```
(* Step 1 *)
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o
```

Sémantique de while

1. Let $V =$ empty.
2. Repeat
   1. Let *exprRef* be the result of evaluating `Expression`.
   2. If *ToBoolean*(*GetValue*(*exprRef*)) is `false`, return (*normal*, *V*, *empty*).

```
(* Steps 2a and 2b *)
| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

(* Step 2b False *)
| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)
```

La formalisation inductive

Technologie

- définie comme un prédicat
- en Coq

Avantages

- isomorphe à la specification (confiance, maintenance)
- donne un principe d'induction

Inconvénient

- 800 règles
- ce n'est qu'une définition, non exécutable

Les formalisations

JSCert

JSRef

le monde selon Coq

le monde “réel”

Un interprète en Coq

```
Definition run_stat_while runs S C rv labs e1 t2 : result :=
 if_spec (run_expr_get_value runs S C e1) (fun S1 v1 =>
 Let b := convert_value_to_boolean v1 in
 if b then
  if_ter (runs_type_stat runs S1 C t2) (fun S2 R =>
  Let rv' := ifb res_value R <> resvalue_empty then res_value R else rv in
  Let loop := fun _ => runs_type_stat_while runs S2 C rv' labs e1 t2 in
  ifb res_type R <> restype_continue \/ ~ res_label_in R labs then (
   ifb res_type R = restype_break /\ res_label_in R labs then
     res_ter S2 rv'
   else (
    ifb res_type R <> restype_normal then
     res_ter S2 R
    else loop tt
   )
  ) else loop tt)
 else res_ter S1 rv).
```

Extrait en Ocaml

```
(** val run_stat_while :
    runs_type -> state -> execution_ctx -> resvalue -> label_set -> expr ->
    stat -> result **)

let run_stat_while runs0 s c rv labs e1 t2 =
  if_spec (run_expr_get_value runs0 s c e1) (fun s1 v1 ->
    let_binding (convert_value_to_boolean v1) (fun b ->
      if b
      then if_ter (runs0.runs_type_stat s1 c t2) (fun s2 r ->
             let_binding
               (if not_decidable
                     (resvalue_comparable r.res_value Coq_resvalue_empty)
                then r.res_value
                else rv) (fun rv' ->
               let_binding (fun x ->
                 runs0.runs_type_stat_while s2 c rv' labs e1 t2) (fun loop ->
                 if or_decidable
                      (not_decidable
                        (restype_comparable r.res_type Coq_restype_continue))
                      (not_decidable (bool_decidable (res_label_in r labs)))
                 then if and_decidable
                           (restype_comparable r.res_type Coq_restype_break)
                           (bool_decidable (res_label_in r labs))
                      then res_ter s2 (res_normal rv')
                      else if not_decidable
                                (restype_comparable r.res_type
                                  Coq_restype_normal)
                           then res_ter s2 r
                           else loop ()
                 else loop ())))
      else res_ter s1 (res_normal rv)))
```

Une formalisation exécutable

Technologie

- opérateurs monadiques
- extraction en OCaml

Avantages

- peut être testée
- démos ! (and débogage)

Inconvénients

- seulement correct pour les tests
- lent

Les formalisations

JSCert

JSRef

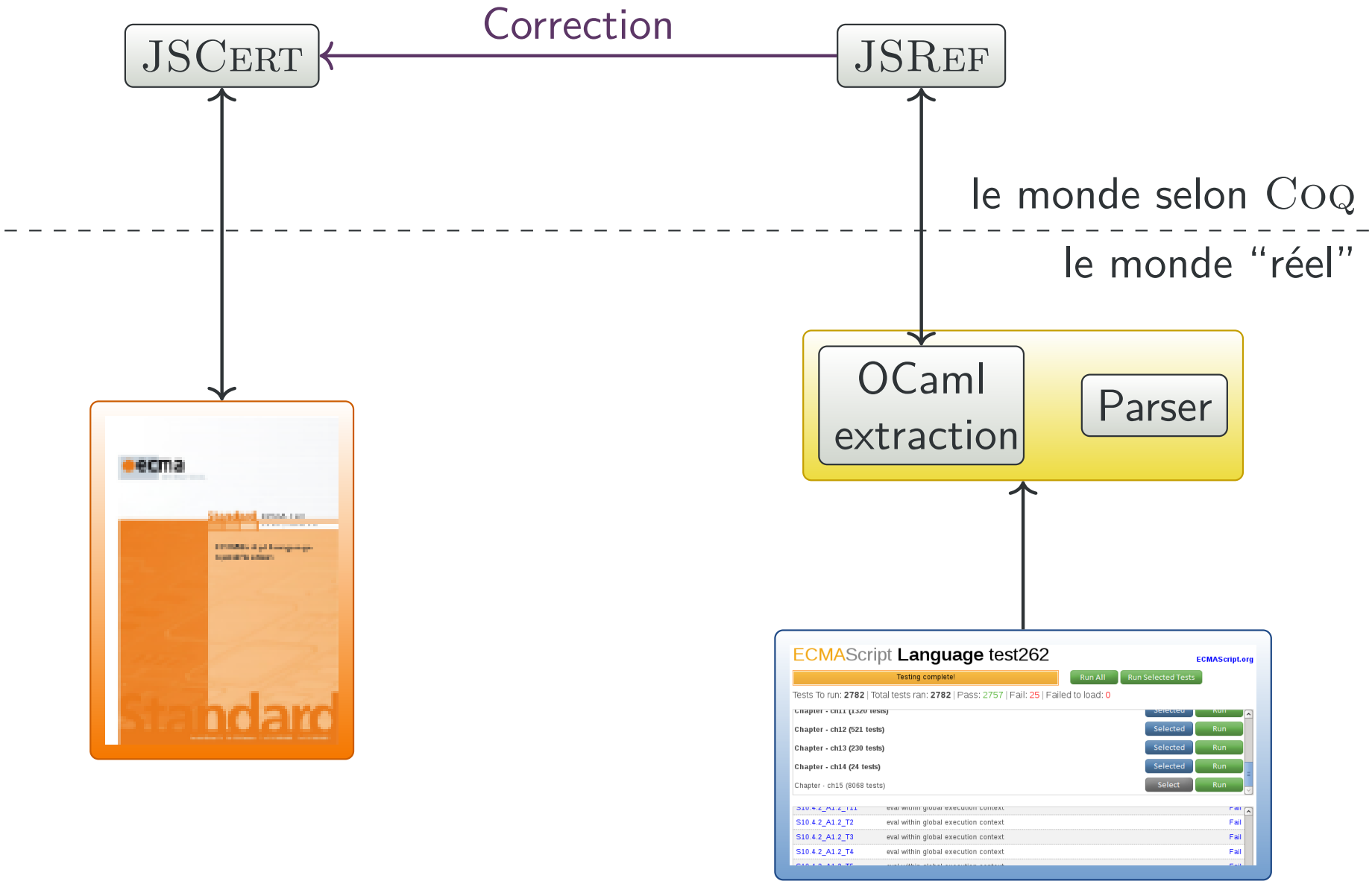
le monde selon Coq

le monde “réel”

OCaml extraction

Parser

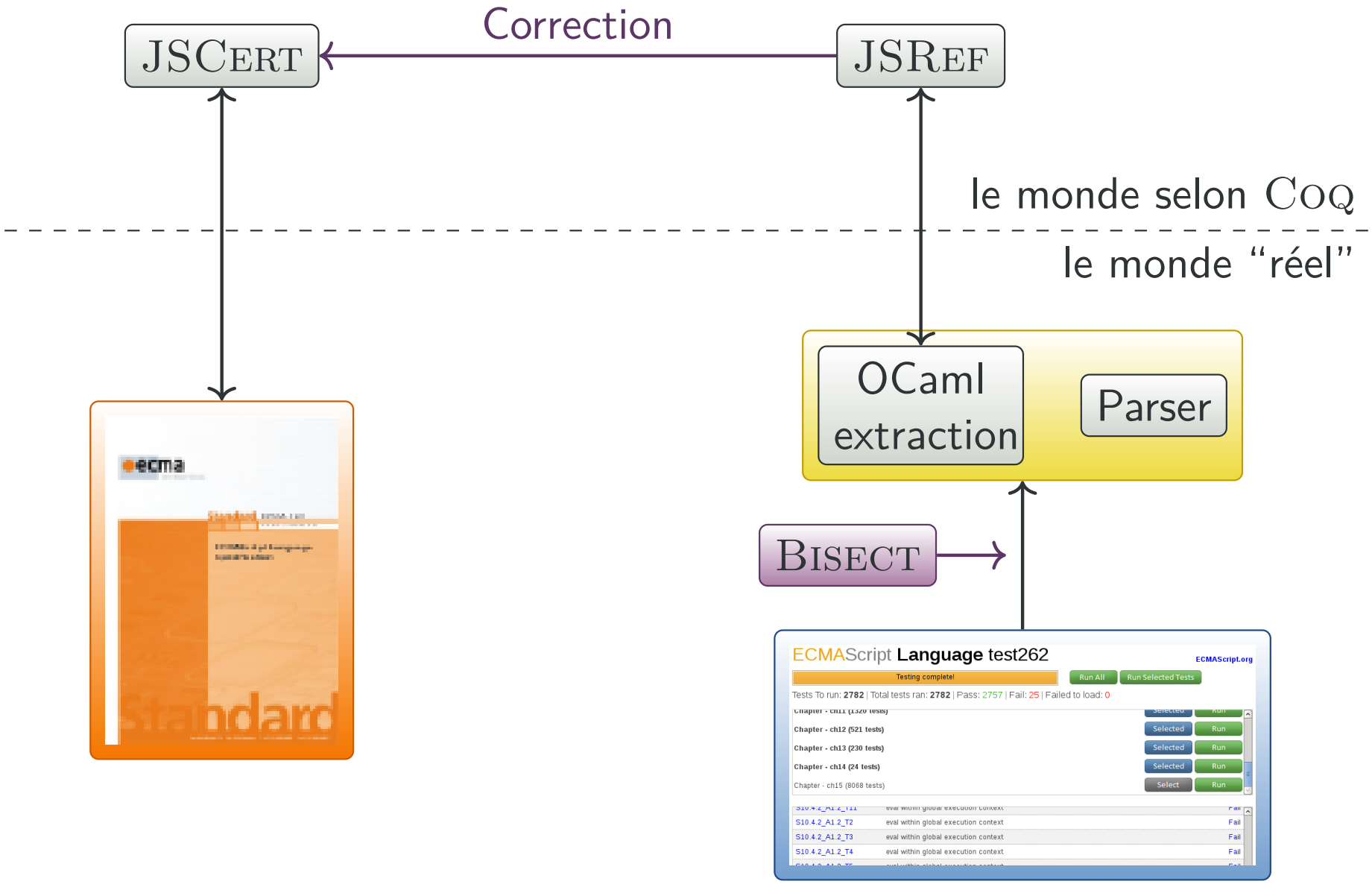
Les formalisations



Preuve de correction

```
Lemma run_stat_while_correct : forall runs S C rv ls e t o,
  runs_type_correct runs ->
  run_stat_while runs S C rv ls e t = o ->
  red_stat S C (stat_while_1 ls e t rv) o.

Corollary run_javascript_correct_num : forall num p o,
  run_javascript (runs num) p = result_out o ->
  red_javascript p o.
```

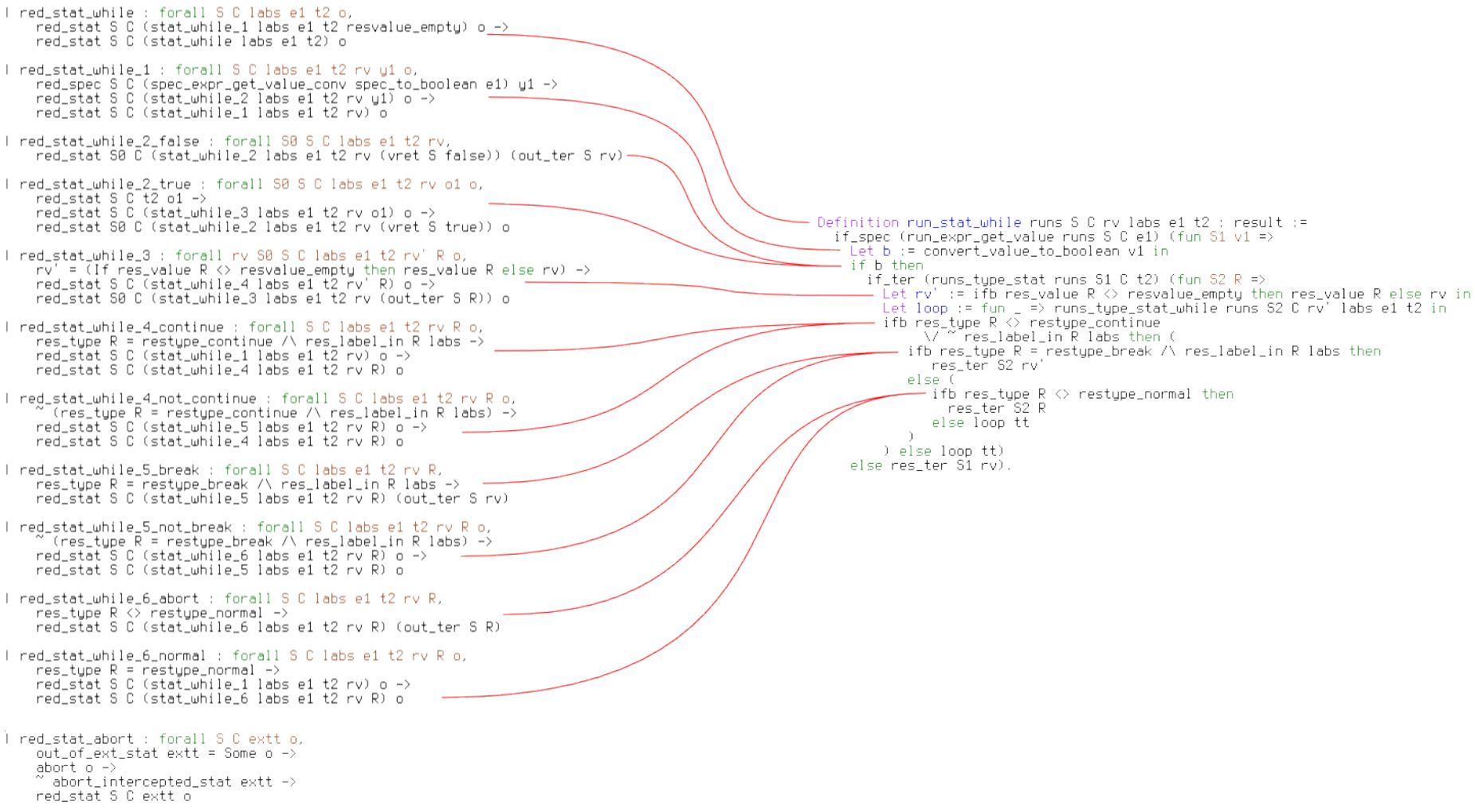
Couverture de code avec Bisect



Couverture de code avec Bisect

```
!002632| (** val run_stat_while :
!002633|     int -> runs_type -> resvalue -> state -> execution_ctx -> label_set ->
!002634|     expr -> stat -> result **)
!002635|
!002636| let rec run_stat_while max_step runs0 rv s c ls e1 t2 =
 002637|   (*[77]*)(fun f0 fS n -> (*[77]*)if n=0 then (*[0]*)f0 () else (*[77]*)fS (n-1))
-002638|     (fun _ ->
 002639|     (*[0]*)Coq_result_bottom)
-002640|     (fun max_step' ->
!002641|     (*[77]*)let run_stat_while' = run_stat_while max_step' runs0 in
!002642|     (*[77]*)if_success_value runs0 c (runs0.runs_type_expr s c e1) (fun s1 v1 ->
!002643|       (*[75]*)if convert_value_to_boolean v1
!002644|       then (*[59]*)if_ter (runs0.runs_type_stat s1 c t2) (fun s2 r2 ->
!002645|              (*[59]*)let rvR = r2.res_value in
!002646|              (*[59]*)let rv' =
!002647|                if resvalue_comparable rvR Coq_resvalue_empty then (*[5]*)rv else (*[54]*)rvR
!002648|              in
!002649|              (*[59]*)if_normal_continue_or_break (Coq_result_out (Coq_out_ter (s2,
!002650|                r2))) (fun r -> (*[41]*)res_label_in r ls) (fun s3 r3 ->
!002651|                (*[40]*)run_stat_while' rv' s3 c ls e1 t2) (fun s3 r3 ->
!002652|                (*[14]*)Coq_result_out (Coq_out_ter (s3, (res_ref rv')))))
!002653|       else (*[16]*)Coq_result_out (Coq_out_ter (s1, (res_ref rv)))))
!002654|     max_step
```

Des exécutions aux règles inductives



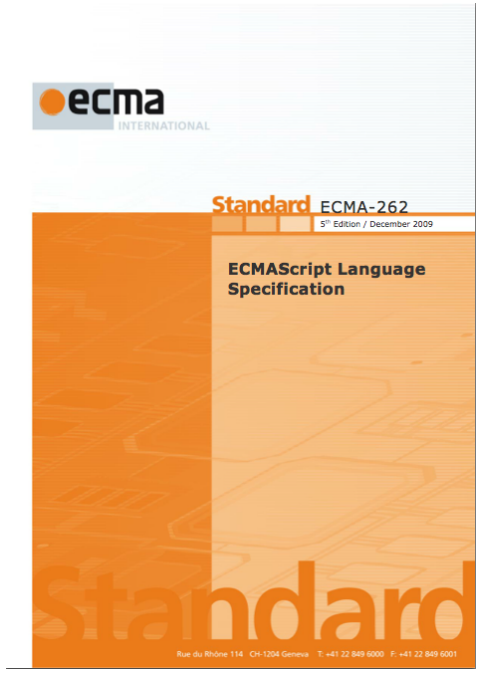
```
| red_stat_while : forall S C labs e1 t2 o,
    red_stat S C (stat_while_1 labs e1 t2 resvalue_empty) o ->
    red_stat S C (stat_while labs e1 t2) o

| red_stat_while_1 : forall S C labs e1 t2 rv y1 o,
    red_spec S C (spec_expr_get_value_conv spec_to_boolean e1) y1 ->
    red_stat S C (stat_while_2 labs e1 t2 rv y1) o ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o

| red_stat_while_2_false : forall S0 S C labs e1 t2 rv,
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S false)) (out_ter S rv)

| red_stat_while_2_true : forall S0 S C labs e1 t2 rv o1 o,
    red_stat S C t2 o1 ->
    red_stat S C (stat_while_3 labs e1 t2 rv o1) o ->
    red_stat S0 C (stat_while_2 labs e1 t2 rv (vret S true)) o

| red_stat_while_3 : forall rv S0 S C labs e1 t2 rv' R o,
    rv' = (If res_value R <> resvalue_empty then res_value R else rv) ->
    red_stat S C (stat_while_4 labs e1 t2 rv' R) o ->
    red_stat S0 C (stat_while_3 labs e1 t2 rv (out_ter S R)) o

| red_stat_while_4_continue : forall S C labs e1 t2 rv R o,
    res_type R = restype_continue /\ res_label_in R labs ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_4_not_continue : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_continue /\ res_label_in R labs) ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_4 labs e1 t2 rv R) o

| red_stat_while_5_break : forall S C labs e1 t2 rv R,
    res_type R = restype_break /\ res_label_in R labs ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) (out_ter S rv)

| red_stat_while_5_not_break : forall S C labs e1 t2 rv R o,
    ~ (res_type R = restype_break /\ res_label_in R labs) ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o ->
    red_stat S C (stat_while_5 labs e1 t2 rv R) o

| red_stat_while_6_abort : forall S C labs e1 t2 rv R,
    res_type R <> restype_normal ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) (out_ter S R)

| red_stat_while_6_normal : forall S C labs e1 t2 rv R o,
    res_type R = restype_normal ->
    red_stat S C (stat_while_1 labs e1 t2 rv) o ->
    red_stat S C (stat_while_6 labs e1 t2 rv R) o


| red_stat_abort : forall S C extt o,
    out_of_ext_stat extt = Some o ->
    abort o ->
    ~ abort_intercepted_stat extt ->
    red_stat S C extt o
```

```
Definition run_stat_while runs S C rv labs e1 t2 : result :=
  if_spec (run_expr_get_value runs S C e1) (fun S1 v1 =>
    Let b := convert_value_to_boolean v1 in
    if b then
      if_ter (runs_type_stat runs S1 C t2) (fun S2 R =>
        Let rv' := ifb res_value R <> resvalue_empty then res_value R else rv in
        Let loop := fun _ => runs_type_stat_while runs S2 C rv' labs e1 t2 in
        ifb res_type R <> restype_continue
            \/ ~ res_label_in R labs then (
          ifb res_type R = restype_break /\ res_label_in R labs then
            res_ter S2 rv'
          else (
            ifb res_type R <> restype_normal then
              res_ter S2 R
            else loop tt
          )
        ) else loop tt)
    else res_ter S1 rv).
```

Le projet AJACS

```
002632 (** val run_stat_while :
002633     int -> runs_type -> resvalue -> state -> execution_ctx -> label_set ->
002634     expr -> stat -> result **)
002635
002636 let rec run_stat_while max_step runs0 rv s c ls e1 t2 =
002637   (*[77]*)(fun f0 fS n -> (*[77]*)if n=0 then (*[0]*)f0 () else (*[77]*)fS (n-1))
002638     (fun _ ->
002639     (*[0]*)Coq_result_bottom)
002640     (fun max_step' ->
002641     (*[77]*)let run_stat_while' = run_stat_while max_step' runs0 in
002642     (*[77]*)if_success_value runs0 c (runs0.runs_type_expr s c e1) (fun s1 v1 ->
002643       (*[75]*)if convert_value_to_boolean v1
002644       then (*[59]*)if_ter (runs0.runs_type_stat s1 c t2) (fun s2 r2 ->
002645             (*[59]*)let rvR = r2.res_value in
002646             (*[59]*)let rv' =
002647               if resvalue_comparable rvR Coq_resvalue_empty then (*[5]*)rv else (*[54]*)rvR
002648             in
002649             (*[59]*)if_normal_continue_or_break (Coq_result_out (Coq_out_ter (s2,
002650               r2))) (fun r -> (*[41]*)res_label_in r ls) (fun s3 r3 ->
002651               (*[40]*)run_stat_while' rv' s3 c ls e1 t2) (fun s3 r3 ->
002652               (*[14]*)Coq_result_out (Coq_out_ter (s3, (res_ref rv')))))
002653       else (*[16]*)Coq_result_out (Coq_out_ter (s1, (res_ref rv)))))
002654     max_step
```

1. spécification
   - couverture de la suite de test
   - interprète de spécification
   - évolutions du langage
   - standardisation
2. sous-langages
   - preuves de λJS (Marek Materzok)
   - formalisation de sous-ensembles de JS (Defensive JS)
3. analyses
   - flots d'informations
   - analyses statiques



Le projet AJACS

Mini-ML Interpreter

```
1 var x = alloc;
2 { x.foo = 12;
3   x.bar = x.foo;
4   x.cycle = x; }
5
```

x: <Object>(1)
  foo: 12

RUN Navigation: 34 / 80 Reset Prev Next Reach condition: Reach

Run successful!

interp.js

```
1  function run_trm(t) {
2    switch (t.tag) {
3    case "trm_var":
4      var v = lookup_var(t.name);
5      return res_val(v);
6    case "trm_cst":
7      return res_val({ tag: "val_cst", cst: t.cst });
8    case "trm_let":
9      return if_success(run_trm(t.t1), function(v1) {
10       env_push(t.name, v1);
11       var res = run_trm(t.t2);
12       env_pop();
13       return res;
14     });
15   case "trm_seq":
16     return if_success(run_trm(t.t1), function(v1) {
17       return if_success(run_trm(t.t2), function (v2) {
18         return(res_val(v2));
19       });
```

t:


```
{
  "tag": "trm_set",
  "loc": {
    "tag": "trm_var",
    "name": "x",
    "line": 3,
    "start": {
      "line": 3,
      "column": 2
    },
    "end": {
      "line": 3,
      "column": 3
    }
  },
  "field": "bar",
  "arg": {
    "tag": "trm_get",
    "loc": {
      "tag": "trm var",
```

Conclusion

Conclusion

- Un langage de programmation intéressant
- Des fondations formelles pour les programmes JavaScript
- Essayez-moi : `http://github.com/jscert/`

<votre projet>

JSCert

Correction

JSRef

le monde selon Coq

le monde “réel”

extraction OCaml

Parser

Bisect