JSCert

A Formalisation of JavaScript in Coq

Martin Bodin

CC7125-1 / MA7125-1

15^{th} and 20^{th} of November

Previously

- JavaScript is complex;
- JavaScript is specified by ECMAScript;
- Translating ECMAScript into big-step is long and not scalable;
- We can translate each steps of ECMAScript into one pretty-big-step rule;
- JSCert is a translation of the core of JavaScript into Coq/pretty-big-step;
- JSCert is accompanied with an interpreter, JSRef;
- We can run JSRef against test suites.

Why is JavaScript so Complex?

- Initially, JavaScript was designed for small scripts done by non-professional programmers.
- Also, only designed in 10 days.
- *Don't break the web!*
- There are actually efforts to make JavaScript simpler:
  - `for (/* ... */ of /* ... */)` iteratively replacing `for (/* ... */ in /* ... */);`
  - the strict mode;
  - etc.
- Inertia is the biggest enemy here, but we can fight it progressively.

Why Big-step wouldn't work in JSCert?

What is great about JavaScript

To do anything about JavaScript, you have to be able to *scale*.

- JavaScript forces us to do things in a scalable way.
- Big-step does not scale on ECMAScript.
- Pretty-big-step does.

Why Big-step wouldn't work in JSCert?

What is great about JavaScript

To do anything about JavaScript, you have to be able to *scale*.

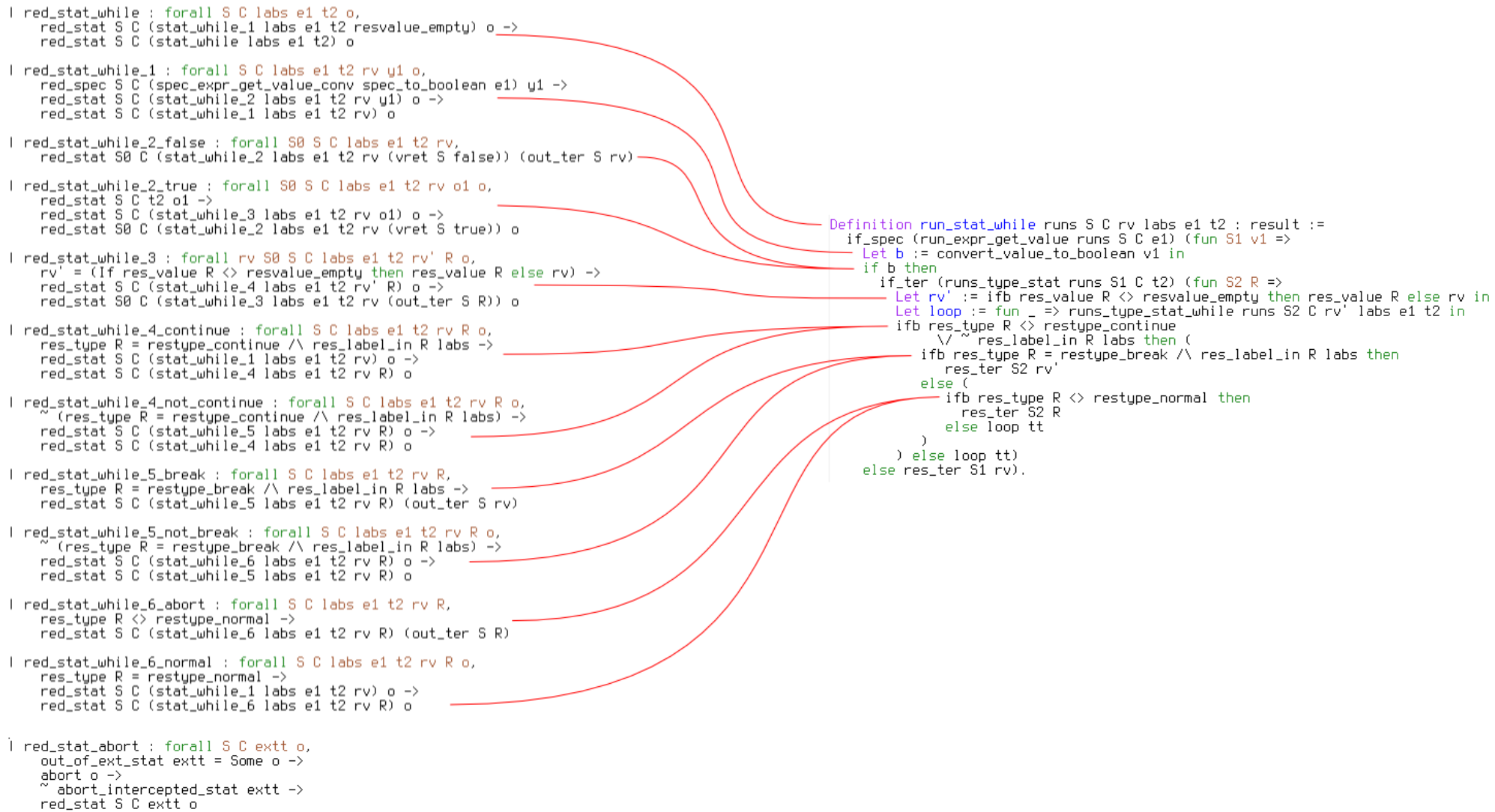
- JavaScript forces us to do things in a scalable way.
- Big-step does not scale on ECMAScript.
- Pretty-big-step does.

Scaling

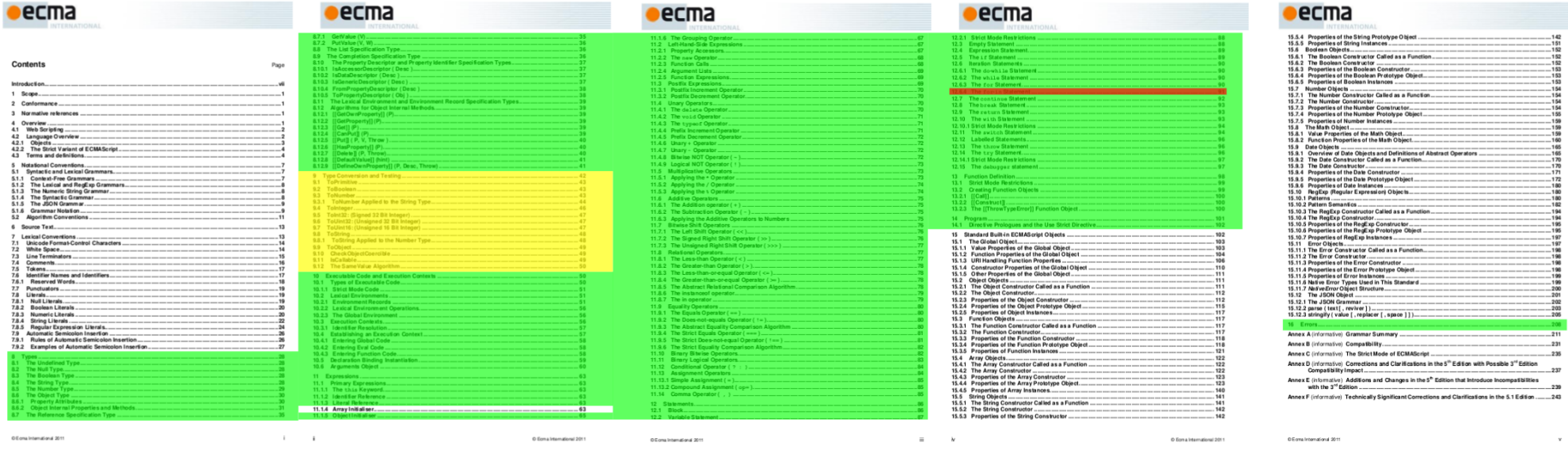
- In semantic size (900 rules just for the core, but what about libraries?);
- In program size (7,500 lines just for Google's main page?);
- In time (ECMAScript 6, 7, 8, ES.Next, ...).

Making Coq Proofs Scale

Why proof automation works?



JSCert Specification Coverage



- Chapters 1–7: how to read ECMAScript;
- Chapters 8–14, 16: core JavaScript;
- Chapters 15: standard library.

The `for (/* ... */ in /* ... */)` construct

"`is: for (lhse in e) s`" is evaluated as follows.

1. Let *exprRef* be the result of evaluating e.
2. Let *exprValue* be *GetValue*(*exprRef*).
3. If *exprValue* is `null` or `undefined`, return (*normal*, *empty*, *empty*).
4. Let *obj* be *ToObject*(*exprValue*).
5. Let $V = empty$.
6. Repeat
    1. Let P be the name of the next property of *obj* whose *Enumerable* attribute is `true`. If there is no such property, return (*normal*, V, *empty*).
    2. Let *lhsRef* be the result of evaluating the `lhse` (it may be evaluated repeatedly).
    3. Call *PutValue*(*lhsRef*, P).
    4. Let *stmt* be the result of evaluating s.
    5. If *stmt.value* is not empty, let $V = stmt.value$.
    6. If *stmt.type* is break and *stmt.target* is in the current label set,

The `for (/* ... */ in /* ... */)` construct

"`is: for (lhse in e) s`" is evaluated as follows.

6. Repeat
    1. Let P be the name of the next property of *obj* whose *Enumerable* attribute is `true`. If there is no such property, return (*normal*, V, *empty*).
    2. Let *lhsRef* be the result of evaluating the `lhse` (it may be evaluated repeatedly).
    3. Call *PutValue*(*lhsRef*, P).
    4. Let *stmt* be the result of evaluating `s`.
    5. If *stmt.value* is not empty, let V = *stmt.value*.
    6. If *stmt.type* is break and *stmt.target* is in the current label set, return (*normal*, V, *empty*).
    7. If *stmt.type* is not continue or *stmt.target* is not in the current label set, then
        1. If *stmt* is an abrupt completion, return *stmt*.

The `for (/* ... */ in /* ... */)` construct

"`is: for (lhse in e) s`" is evaluated as follows.

6. Repeat
    1. Let P be the name of the next property of *obj* whose *Enumerable* attribute is `true`. If there is no such property, return (*normal*, V, *empty*).

The mechanics and order of enumerating the properties (Step 1) is not specified. Properties of the object being enumerated may be deleted during enumeration, [they will then] not be visited. If new properties are added to the object being enumerated during enumeration, [they] are not guaranteed to be visited in the active enumeration. A property name must not be visited more than once in any enumeration. Enumerating the properties of an object includes enumerating properties of its prototype.

Zooming Out

What is the most difficult part of a formalisation?

You have to convince *other people* (often non-Coq people) that your semantics is the right one.

- This is actually not about Coq: Coq is useful for the people who will use your semantics, not for you;
- Be sure to understand the original language:
  - What is the most important: the specification or interpreters? The language community?

The JSCert Project

~ 900 rules

~ 3,000 lines of code

JSCert ← Correctness → JSRef

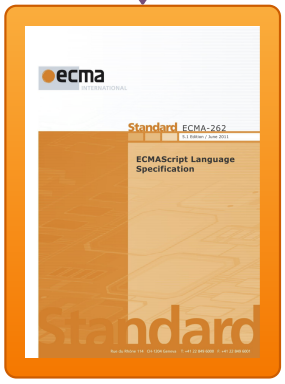
~ 6,000 lines of proof

Step / rule correspondence

Running tests

Coq definitions

Industrial world



~ 900 steps

~ 200 pages

5,126 tests passed

out of 11,725

Bugs found

Bugs in interpreters

- Invalid return values of `try {/* ... */} finally {/* ... */}` blocks;
- Changing dead code altered the final result.

Bugs in ECMAScript

- Broken algorithm;
- Some cases forgotten in the Enumerate method.

Bugs in test suites

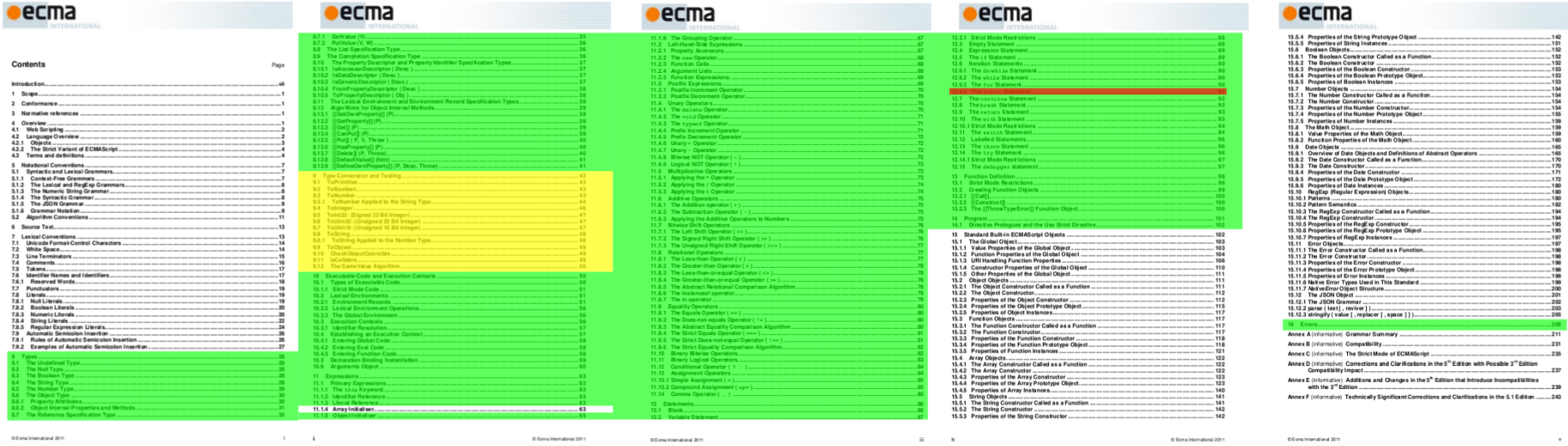
- Tests checking the value of unspecified fields;
- Bugs in tests, mimicking implementation bugs.

Reporting bugs are great way to make people trust you!

`http://jscert.org/popl14/?full#20`

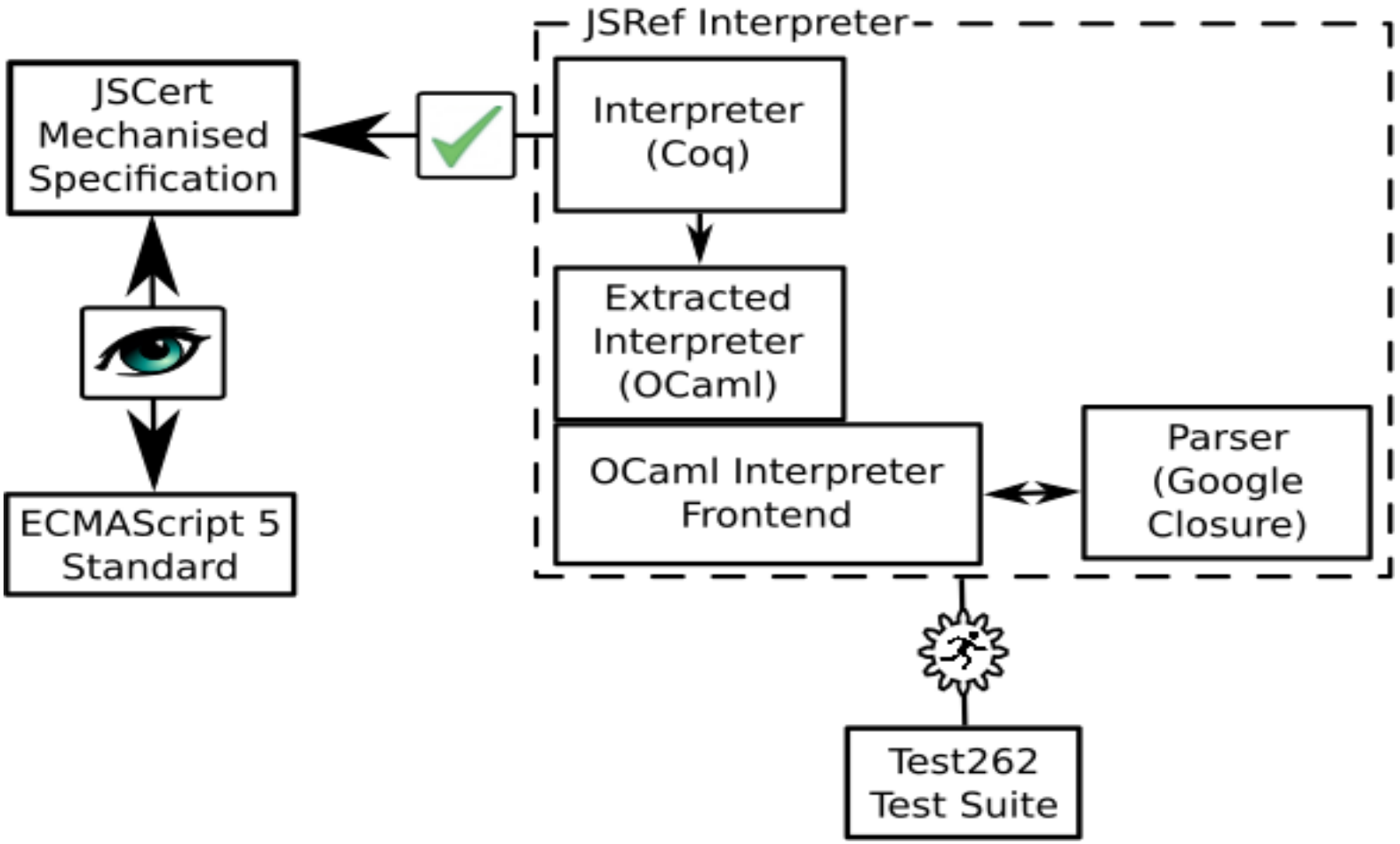
Increasing the Coverage of JSCert

Coverage of JSCert



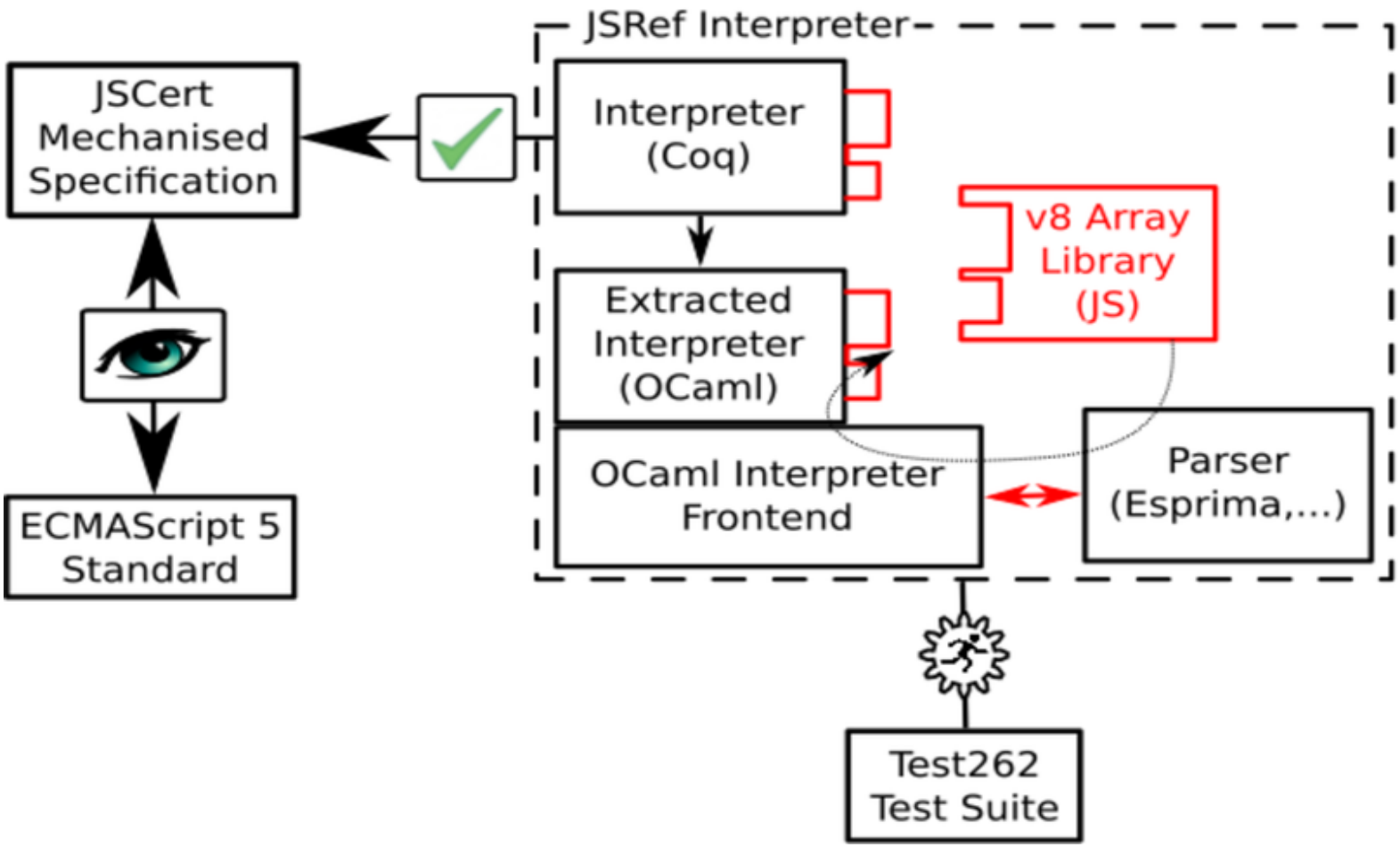
How to easily add Chapter 15?

Reusing already existing libraries



Philippa Gardner et al. "A Trusted Mechanised Specification of JavaScript: One Year On". In: *CAV*. 2015.

Reusing already existing libraries



Philippa Gardner et al. "A Trusted Mechanised Specification of JavaScript: One Year On". In: *CAV*. 2015.

Bisect

Let us test ECMAScript!

A code coverage tool for OCaml applied on JSRef, which is closed to ECMAScript:

```
let run_call_full max_step s c l vthis args =
    (*[ 3311 ]*)(fun f0 fS n -> (*[ 3311 ]*)if n=0 then
      (*[ 0 ]*)f0 () else (*[ 3311 ]*)fS (n-1))
  (fun _ -> (*[ 0 ]*)Coq_result_bottom)
  (fun max_step' ->
    (*[ 3311 ]*)let run_expr' = run_expr max_step' in
    (*[ 3311 ]*)if_some (run_object_method object_call_ s l)
    (fun c0 ->
      match c0 with
      | Coq_call_default ->
        (*[ 2555 ]*)entering_func_code runs' s c l vthis args
      | Coq_call_after_bind -> (*[ 0 ]*)Coq_result_stuck
      | Coq_call_prealloc b -> (*[ 756 ]*)run_call' s c b args))
    max_step
```

Bisect

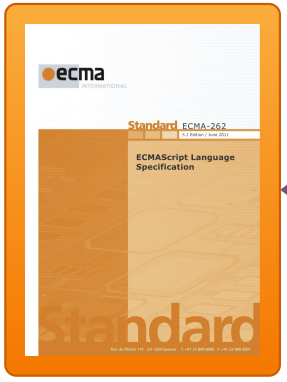
Let us test ECMAScript!

A code coverage tool for OCaml applied on JSRef, which is closed to ECMAScript:

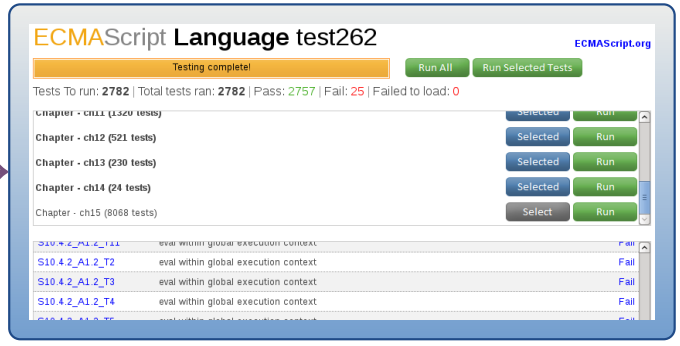
```
let run_call_full max_step s c l vthis args =
    (*[ 3311 ]*)(fun f0 fS n -> (*[ 3311 ]*)if n=0 then
      (*[ 0 ]*)f0 () else (*[ 3311 ]*)fS (n-1))
  (fun _ -> (*[ 0 ]*)Coq_result_bottom)
  (fun max_step' ->
    (*[ 3311 ]*)let run_expr' = run_expr max_step' in
    (*[ 3311 ]*)if_some (run_object_method object_call_ s l)
    (fun c0 ->
      match c0 with
      | Coq_call_default ->
        (*[ 2555 ]*)entering_func_code runs' s c l vthis args
      | Coq_call_after_bind -> (*[ 0 ]*)Coq_result_stuck
```

JSCert/JSRef enabled to test the coverage of the test suite for the first time!

JSExplain



Using JSCert/JSRef



- `http://ajacs.inria.fr/jsexplain/driver.html`
- `https://github.com/jscert/jsexplain`

Central idea

- Everyone can read and understand JSRef;
- We could use the interpreter to explain JavaScript's behaviours;
- JSRef should be able to generate everything else.

Implement real browser behaviours

- The initial heap is different;
- Some behaviours (like `Function.prototype.toString ()`) are implementation-dependent.

We could formalise all these

- By adding special rules into JSCert;
- By adding a special argument to the predicates `red_expr` to denote the browser;
- By executing their JavaScript code to build another initial heap.

Logics for JavaScript?

Complex, but possible

Philippa Gardner, Sergio Maffeis, and Gareth Smith. "Towards a Program Logic for JavaScript". In: *POPL*. 2012.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript". In: *SAS*. 2009.

Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. "Automatic Analysis of Open Objects in Dynamic Language Programs". In: *SAS*. 2014.

The JSCert Project

~ 900 rules

~ 3,000 lines of code

JSCert ← Correctness → JSRef

~ 6,000 lines of proof

Step / rule correspondence

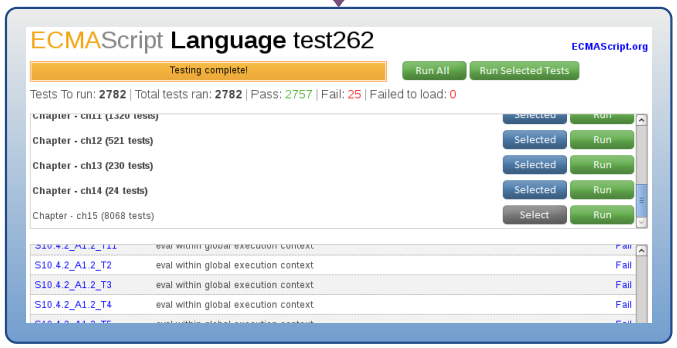
Running tests

Coq definitions

Industrial world

~ 900 steps

~ 200 pages



5,126 tests passed

1. Previously
2. Zooming Out
3. Increasing the Coverage of JSCert