Formal Semantics and Certified Analyses of Hop.js

Alan Schmitt

May 31, 2018

Goal

+

Agenda

- Formalizing JavaScript is tricky
- The evolution of our approach (JSCert and JSExplain)
- Challenges of applying it to Hop

JavaScript

A quick history of JavaScript and ECMAScript

1995	Brendan Eich hired by Netscape to embed Scheme
May 1995	as Java is included in Netscape, scripting should have a similar syntax; JavaScript prototype developed in 10 days
Dec. 1995	JavaScript deployed in Netscape Navigator 2.0 beta 3
Aug. 1996	JScript deployed in Internet Explorer 3.0

A quick history of JavaScript and ECMAScript

1995	Brendan Eich hired by Netscape to embed Scheme
May 1995	as Java is included in Netscape, scripting should have a similar syntax; JavaScript prototype developed in 10 days
Dec. 1995	JavaScript deployed in Netscape Navigator 2.0 beta 3
Aug. 1996	JScript deployed in Internet Explorer 3.0

code is supposed to run identically in every browser

$\Rightarrow$ strong need for standardization

A quick history of JavaScript and ECMAScript

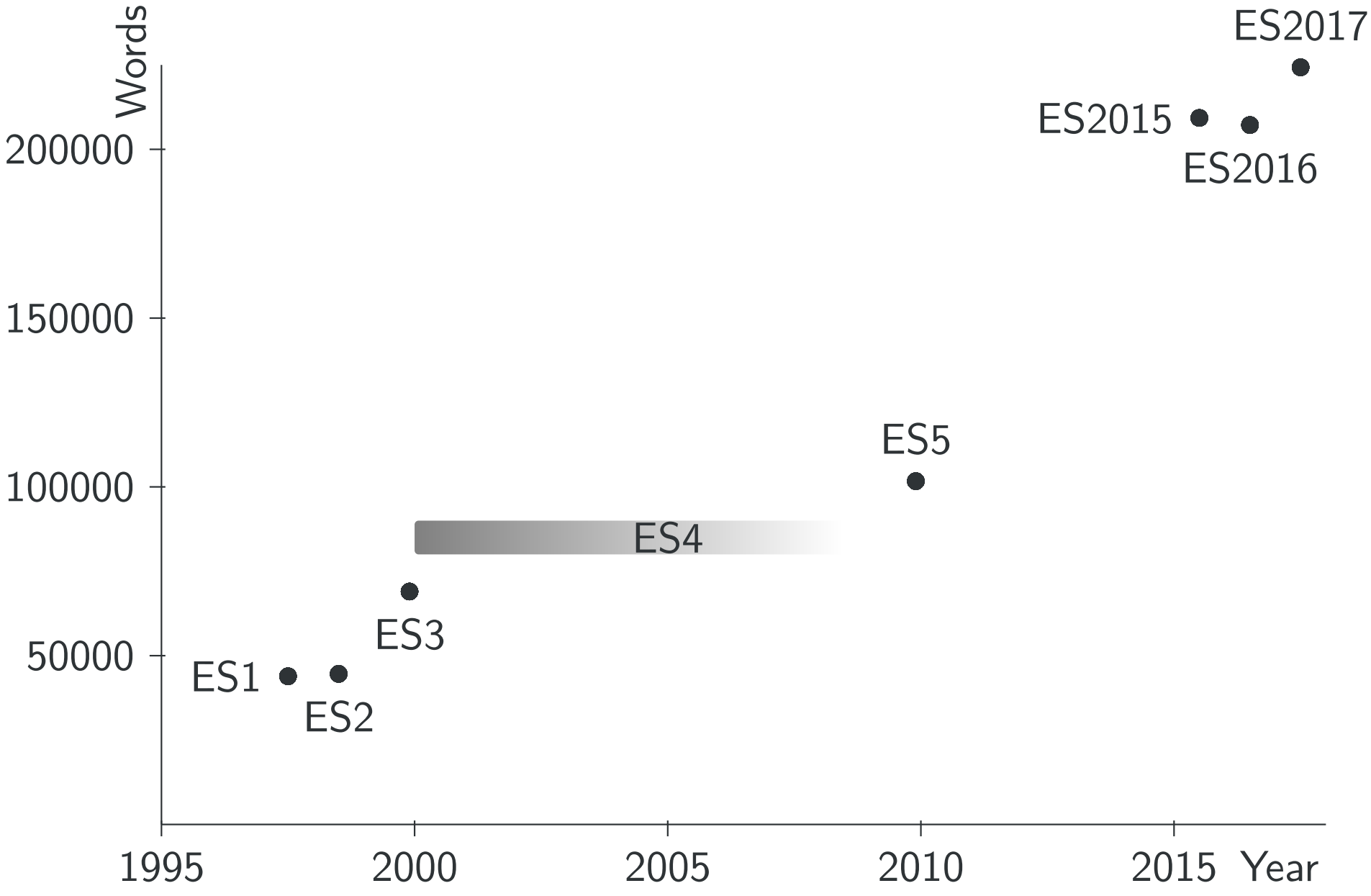
1995	Brendan Eich hired by Netscape to embed Scheme
May 1995	as Java is included in Netscape, scripting should have a similar syntax; JavaScript prototype developed in 10 days
Dec. 1995	JavaScript deployed in Netscape Navigator 2.0 beta 3
Aug. 1996	JScript deployed in Internet Explorer 3.0

code is supposed to run identically in every browser

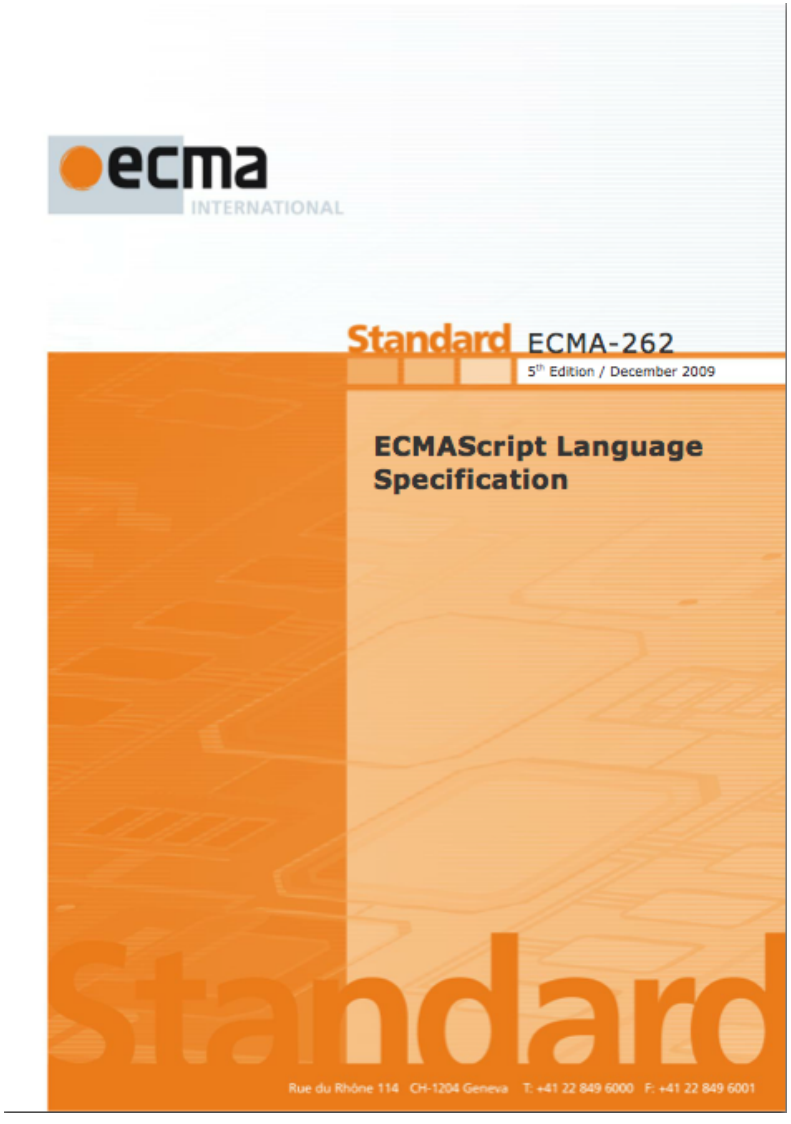
$\Rightarrow$ strong need for standardization

Nov. 1996	JavaScript submitted to Ecma International
June 1997	first edition of ECMA-262 (110 pages)

A quick history of JavaScript and ECMAScript



The specification



- new version every year
- 6 meetings of TC39 each year
- transparent process, on github
- don't break the web

JSCert

What is JSCert?

Two JavaScript semantics in Coq

- descriptive given a program and a result, say if they are related
- executable given a program, compute the result

Correctness

If program P executes to v, then P and v are related

- 2 years, 8 people
- 18 klocs of Coq

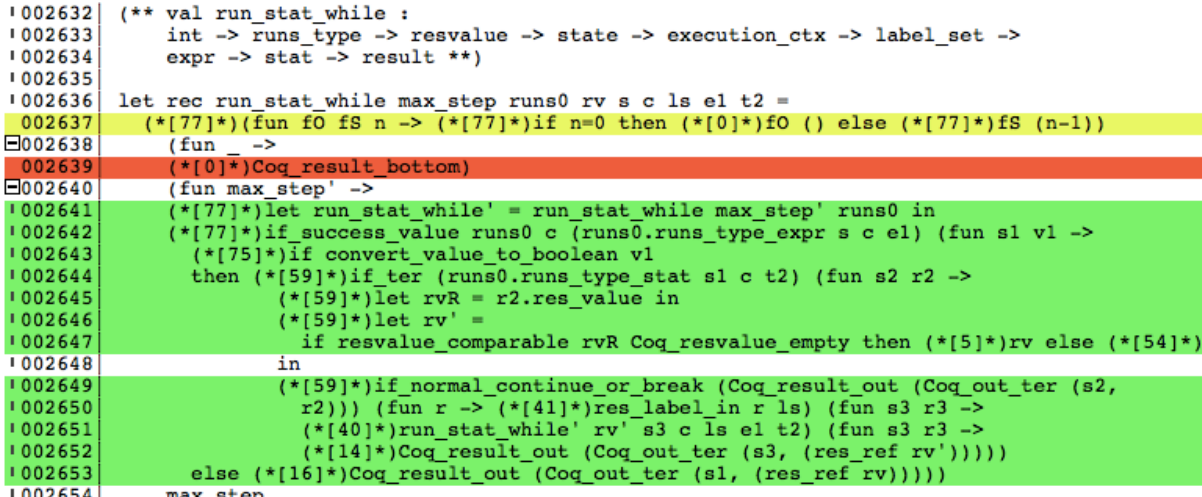






Positive Outcomes

- good coverage of the core of ECMAScript 5.1
- code extraction from JSRef
  1. instrumented to report coverage
  2. run the test suite
  3. find places not executed (not tested)
  4. relate to parts of the spec not tested
  5. discover discrepancies between implementations



Scaling Issues

- Hard to keep pace with the standardisation
  - need to update two formalizations and a correctness proof
- JSCert inductive definition is too big
  - no `inversion` possible, preventing most proofs

Lessons

- Many low hanging fruits from an implementation close to the spec
- Maintain a single artefact, derive other formats from it
- Coq formalization should be usable for proofs

JSExplain

An OCaml interpreter of JavaScript

- close to the specification
- uses a tiny subset of OCaml in monadic style
  - functions, tuples, shallow pattern matching, records

```
1. Let lprim be ? ToPrimitive(lval).
2. Let rprim be ? ToPrimitive(rval).
3. If Type(lprim) is String or Type(rprim) is String, then
    a. Let lstr be ? ToString(lprim).
    b. Let rstr be ? ToString(rprim).
    c. Return the string-concatenation of lstr and rstr.
4. Let lnum be ? ToNumber(lprim).
5. Let rnum be ? ToNumber(rprim).
6. Return the result of applying the addition operation to lnum and rnum.
```

An OCaml interpreter of JavaScript

- close to the specification
- uses a tiny subset of OCaml in monadic style
  - functions, tuples, shallow pattern matching, records

```
and run_binary_op_add s0 c v1 v2 =
  let%prim (s1, w1) = to_primitive_def s0 c v1 in
  let%prim (s2, w2) = to_primitive_def s1 c v2 in
  if  (type_compare (type_of (Coq_value_prim w1)) Coq_type_string)
   || (type_compare (type_of (Coq_value_prim w2)) Coq_type_string)
  then
    let%string (s3, str1) = to_string s2 c (Coq_value_prim w1) in
    let%string (s4, str2) = to_string s3 c (Coq_value_prim w2) in
    res_out (Coq_out_ter (s4, (res_val (Coq_value_prim (Coq_prim_string (strappend str1 str2))))))
  else
    let%number (s3, n1) = to_number s2 c (Coq_value_prim w1) in
    let%number (s4, n2) = to_number s3 c  (Coq_value_prim w2) in
    res_out (Coq_out_ter (s4, (res_val (Coq_value_prim (Coq_prim_number (n1 +. n2))))))
```

Compiled to JavaScript

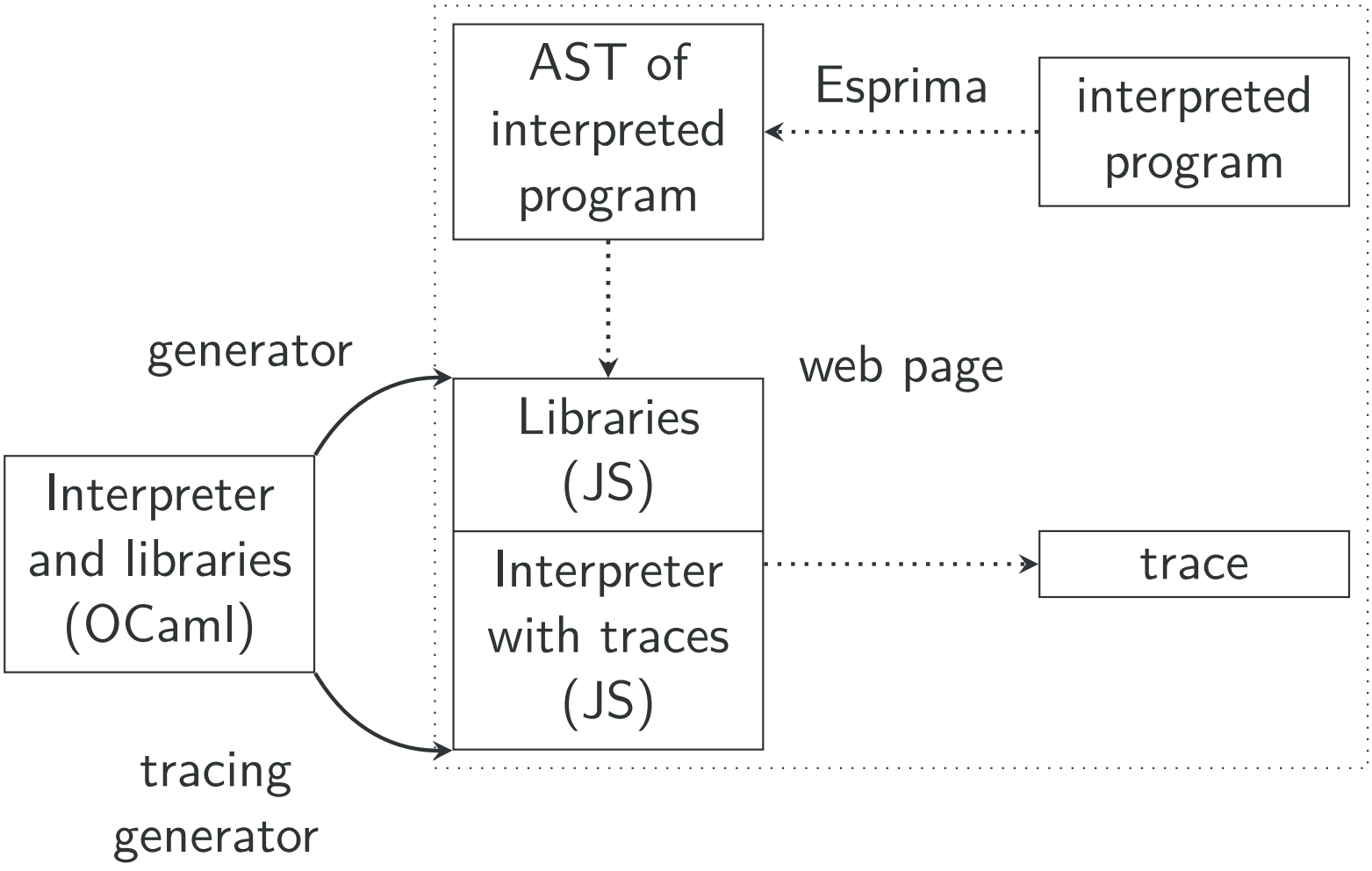
- motivation: run it in a browser
- uses compiler-libs to generate a typed AST, which we translate
- target is a tiny subset of JS
  - functions, objects (no prototype), arrays, string, numbers
  - no type conversion

```
var run_binary_op_add = function (s0, c, v1, v2) {
  return (if_prim(to_primitive_def(s0, c, v1), function(s1, w1) {
    return (if_prim(to_primitive_def(s1, c, v2), function(s2, w2) {
      if ((type_compare(type_of(Coq_value_prim(w1)), Coq_type_string())
        || type_compare(type_of(Coq_value_prim(w2)), Coq_type_string()))) {
          return (if_string(to_string(s2, c, Coq_value_prim(w1)), function(s3, str1) {
            return (if_string(to_string(s3, c, Coq_value_prim(w2)), function(s4, str2) {
              return (res_out(Coq_out_ter(s4, res_val(
                      Coq_value_prim(Coq_prim_string(strappend(str1, str2)))))); }));}));
      } else { ... }})); }));
};
```

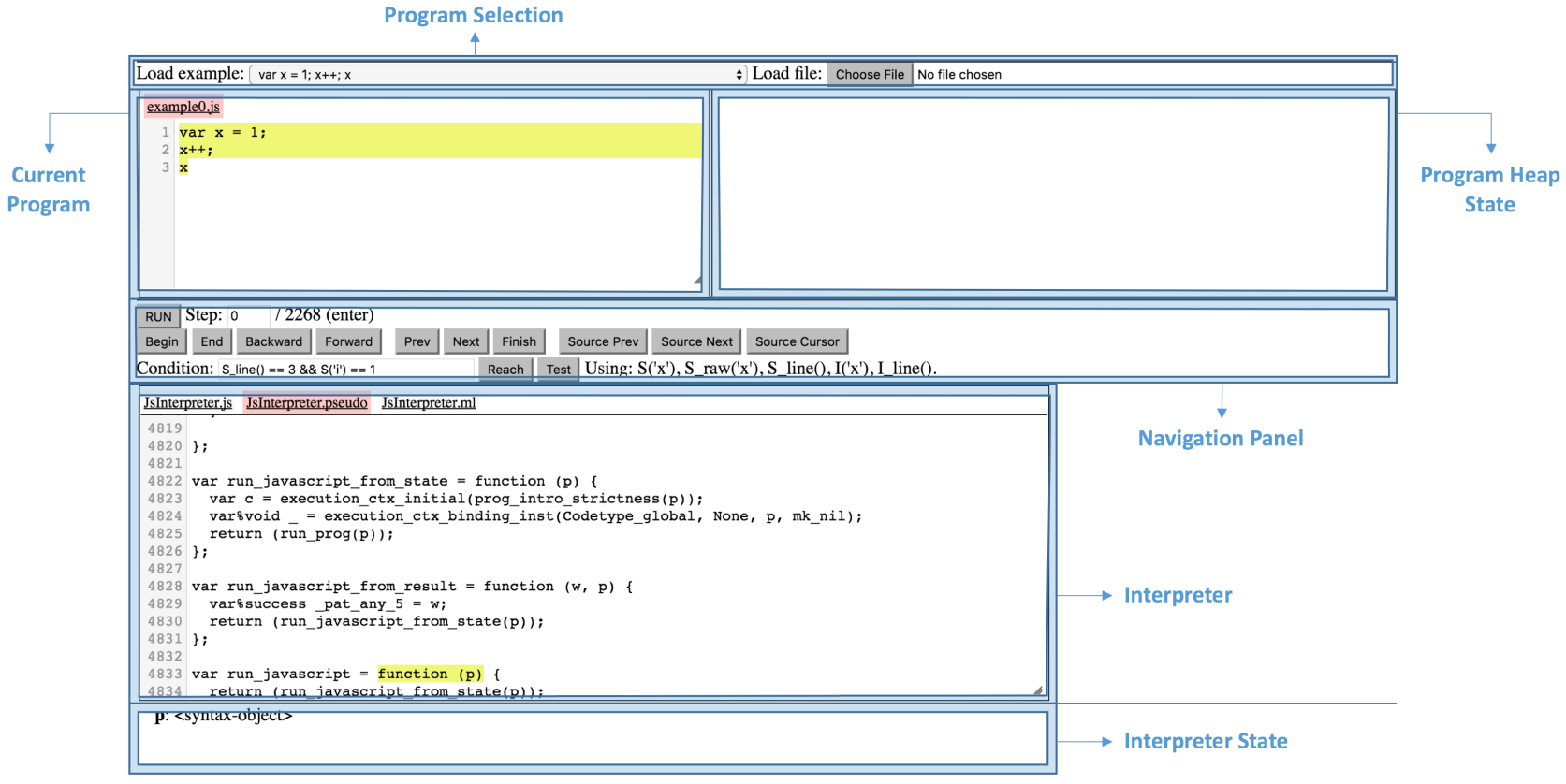
JSExplain

- request by Shu-yu Guo (Dagstuhl, 2014): a step by step execution of the spec
- instrument the generated JavaScript to record *events*
  - `Enter` (enter a function)
  - `CreateCtx(ctx)` (new function scope)
  - `Add(ident,value)` (`let` binding)
  - `Return` (return from a function)
- executing the instrumented interpreter generates a trace of events
- web tool to navigate these traces

Architecture



Demo


Program Selection
Load example: var x = 1; x++; x
Load file: Choose File No file chosen
example0.js
1 var x = 1;
2 x++;
3 x
Current Program
Program Heap State
RUN Step: 0 / 2268 (enter)
Begin End Backward Forward Prev Next Finish Source Prev Source Next Source Cursor
Condition: S_line() == 3 && S('i') == 1 Reach Test Using: S('x'), S_raw('x'), S_line(), I('x'), I_line().
Navigation Panel
JsInterpreter.js JsInterpreter.pseudo JsInterpreter.ml
4819
4820 };
4821
4822 var run_javascript_from_state = function (p) {
4823 var c = execution_ctx_initial(prog_intro_strictness(p));
4824 var%void _ = execution_ctx_binding_inst(Codetype_global, None, p, mk_nil);
4825 return (run_prog(p));
4826 };
4827
4828 var run_javascript_from_result = function (w, p) {
4829 var%success _pat_any_5 = w;
4830 return (run_javascript_from_state(p));
4831 };
4832
4833 var run_javascript = function (p) {
4834 return (run_javascript_from_state(p));
Interpreter
p: <syntax-object>
Interpreter State

HopExplain

Extensions of JSExplain

- extension to current version of JavaScript
  - ongoing, we now can debug it using JSexplain itself
  - engineer hired to work on this in September
- towards a typed specification?
  - PR 1135: Explicitly note mathematical values
  - Issue 496: abstract operations don't always return Completion Records
- continuous participation in the committee
- better trace navigator
  - links to the specification

Coq Extraction

- needed to prove invariants of the specification and certify analyses
- modular description of the semantics with a simpler induction principle
  - POC for a small language
- postdoctoral topic in SPAI, recruitment ongoing

Challenges for HopExplain

- JSExplain easily adapted to other languages (MLExplain[1])
- challenge: capture the distributed and concurrent aspects
  - close collaboration with Indes, co-supervising a PhD student
- framework and rule format to describe semantics (collaboration with Imperial College London)
- use the Coq extraction to certify selected analyses of Hop.js programs

[1] `https://github.com/Docteur-Lalla/mlexplain/tree/mlexplain`