JavaScript Specification: from a Formal Semantics to an Interactive Debugger

Alan Schmitt, with Arthur Charguéraud and Thomas Wood

February 22, 2017

JavaScript is Special

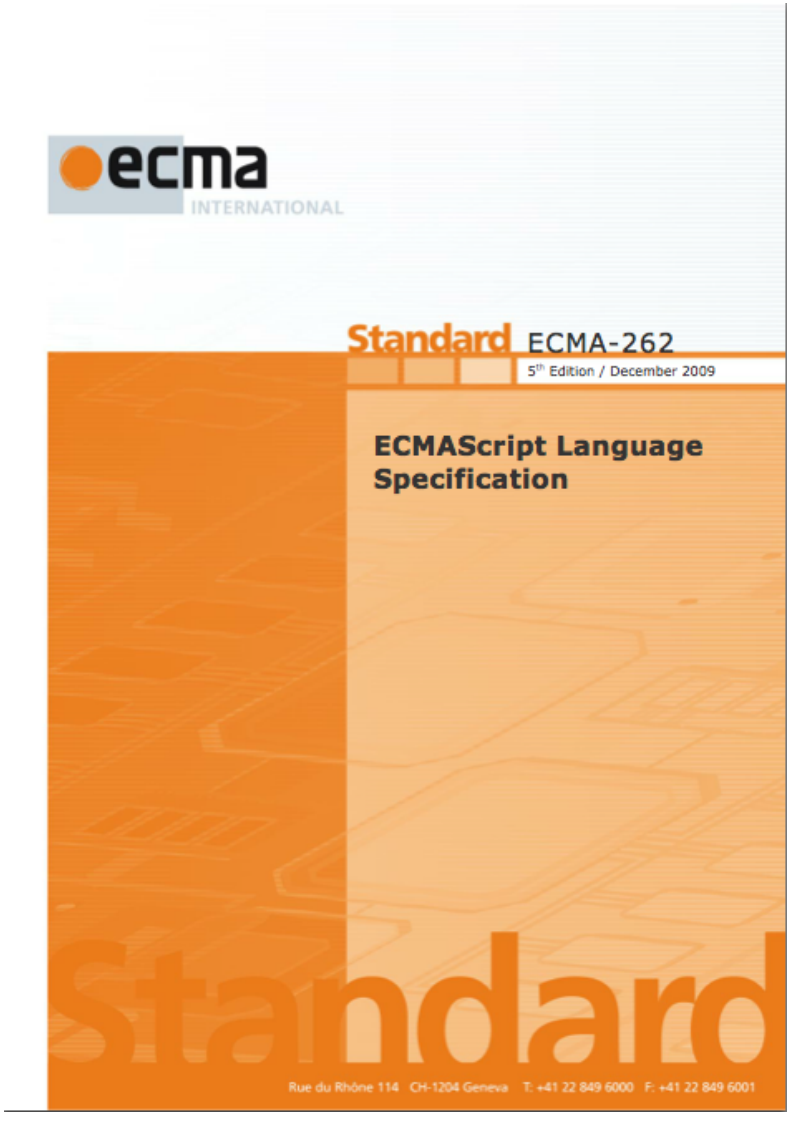
- code is supposed to run identically in different browsers

- strong need for standardization

A quick history of JavaScript and ECMAScript

1995	Brendan Eich hired by Netscape Communications to embed Scheme in Netscape
May 1995	as Java is included in Netscape, scripting should have a similar syntax; JavaScript prototype developed in 10 days
Dec. 1995	deployed in Netscape Navigator 2.0 beta 3
Nov. 1996	JavaScript submitted to Ecma International
June 1997	first edition of ECMA-262 (110 pages)
June 1998	ECMAScript 2 (117 pages)
Dec. 1999	ECMAScript 3 (188 pages)
2000	Work on ECMAScript 4 starts
July 2008	ECMAScript 4 is abandoned
Dec. 2009	ECMAScript 5 (252 pages)
June 2015	ECMAScript 2015 (566 pages)
June 2016	ECMAScript 2016 (586 pages)

The specification



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

JSCert

- we want to do certified analyses of JavaScript programs
- *certified* means *proven in a proof assistant*
- to prove anything we need a formal semantics
- lucky for us, we could rely on the specification
- that's how we created JSCert

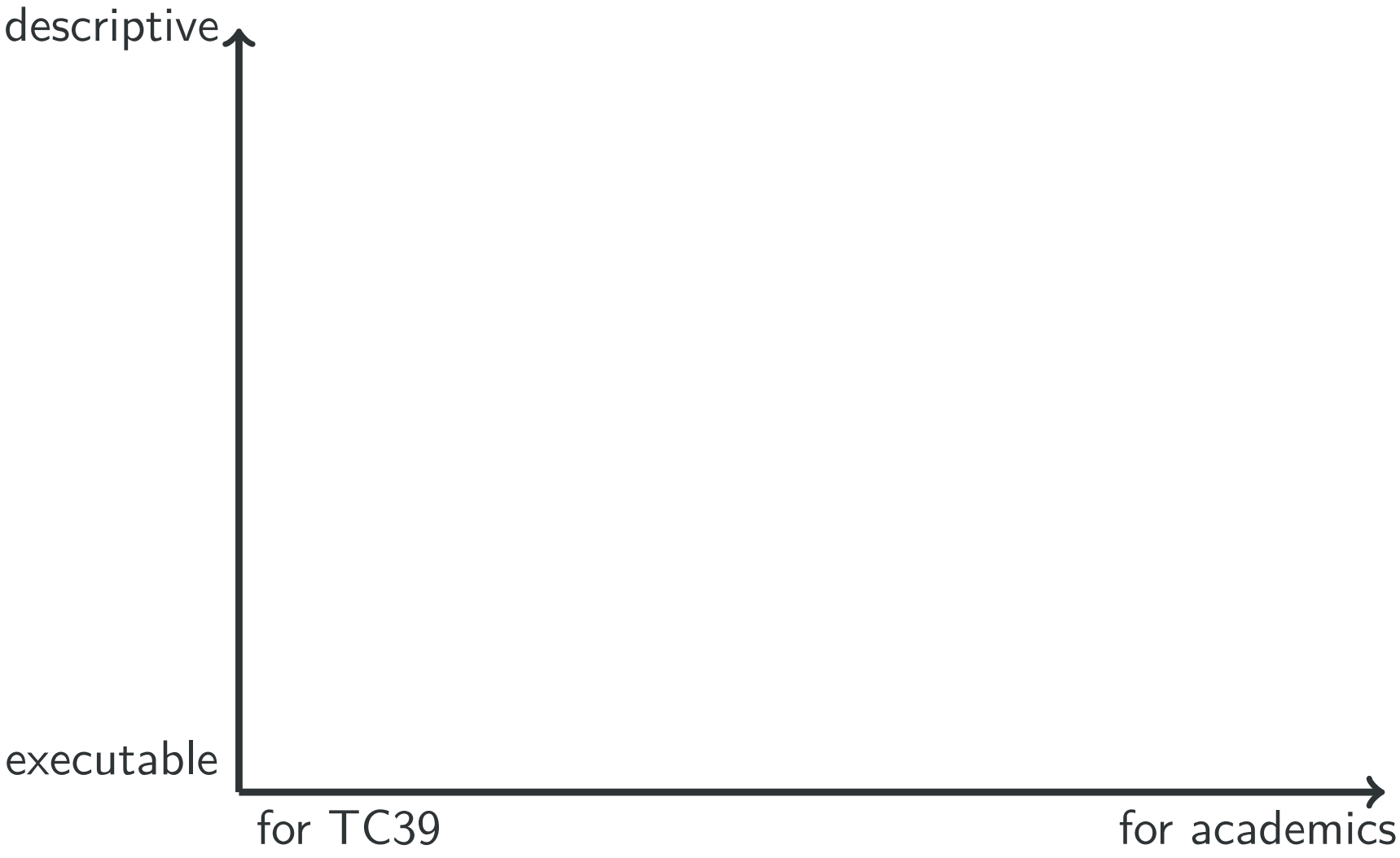
What is JSCert?

JSCert is composed of two related semantics of JavaScript in the Coq proof assistant. One is *descriptive*, to be used to prove properties on programs and the language. The other is *executable*, to actually run code (very slowly).

from JSCert to TC39

- we started proving things with JSCert
  - desugaring in λJS
  - test coverage
- then we got side-tracked
  - the spec is a great thing, but huge and complex
  - we started talking with some people at TC39, wondering if we could do something to help
  - we quickly realized that to help, we would need to formalize ES2015, then ES2016 ...
  - where could we find the manpower?

The Big Picture

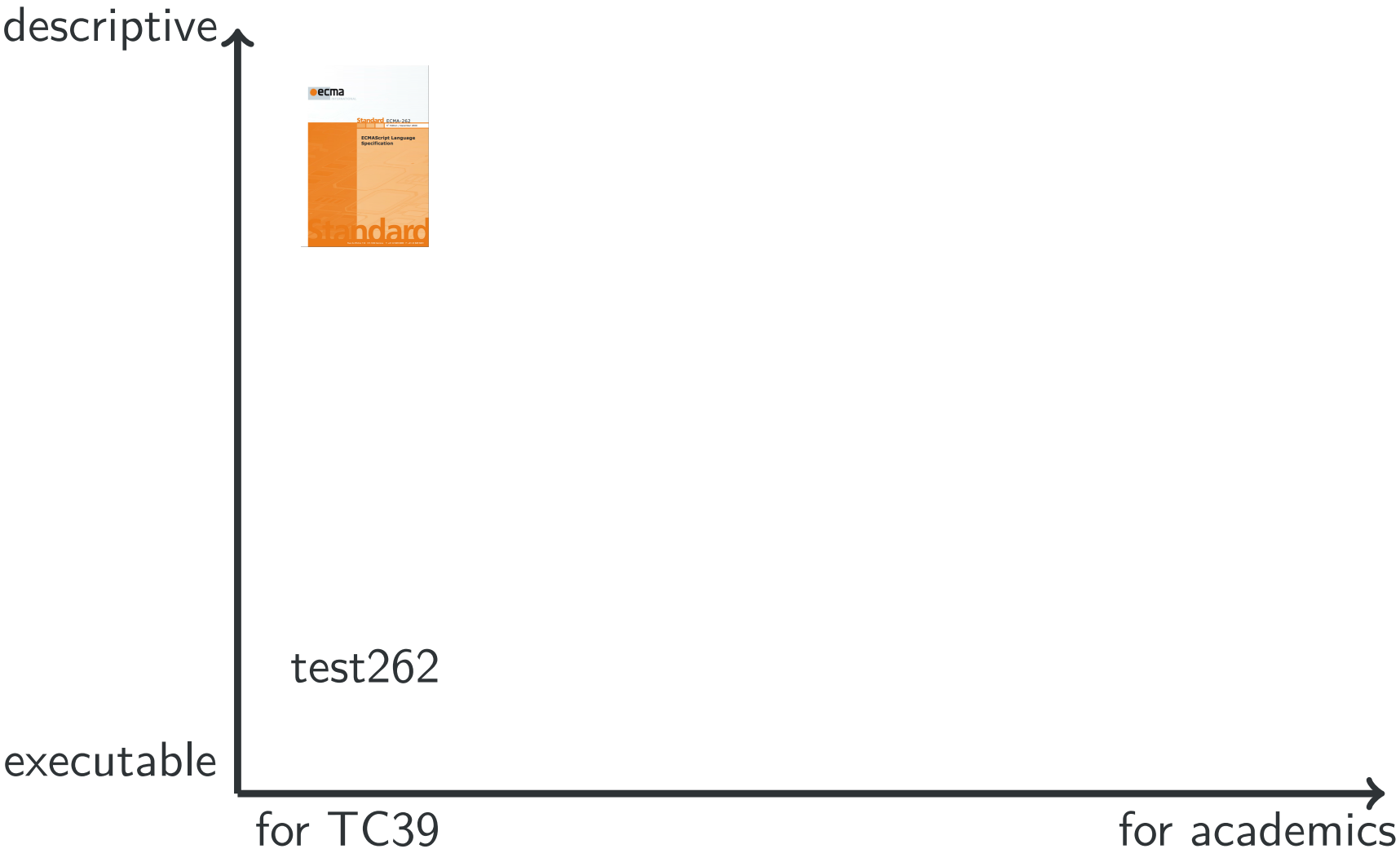


The specification

Evaluation of: *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
   - Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*).

The Big Picture



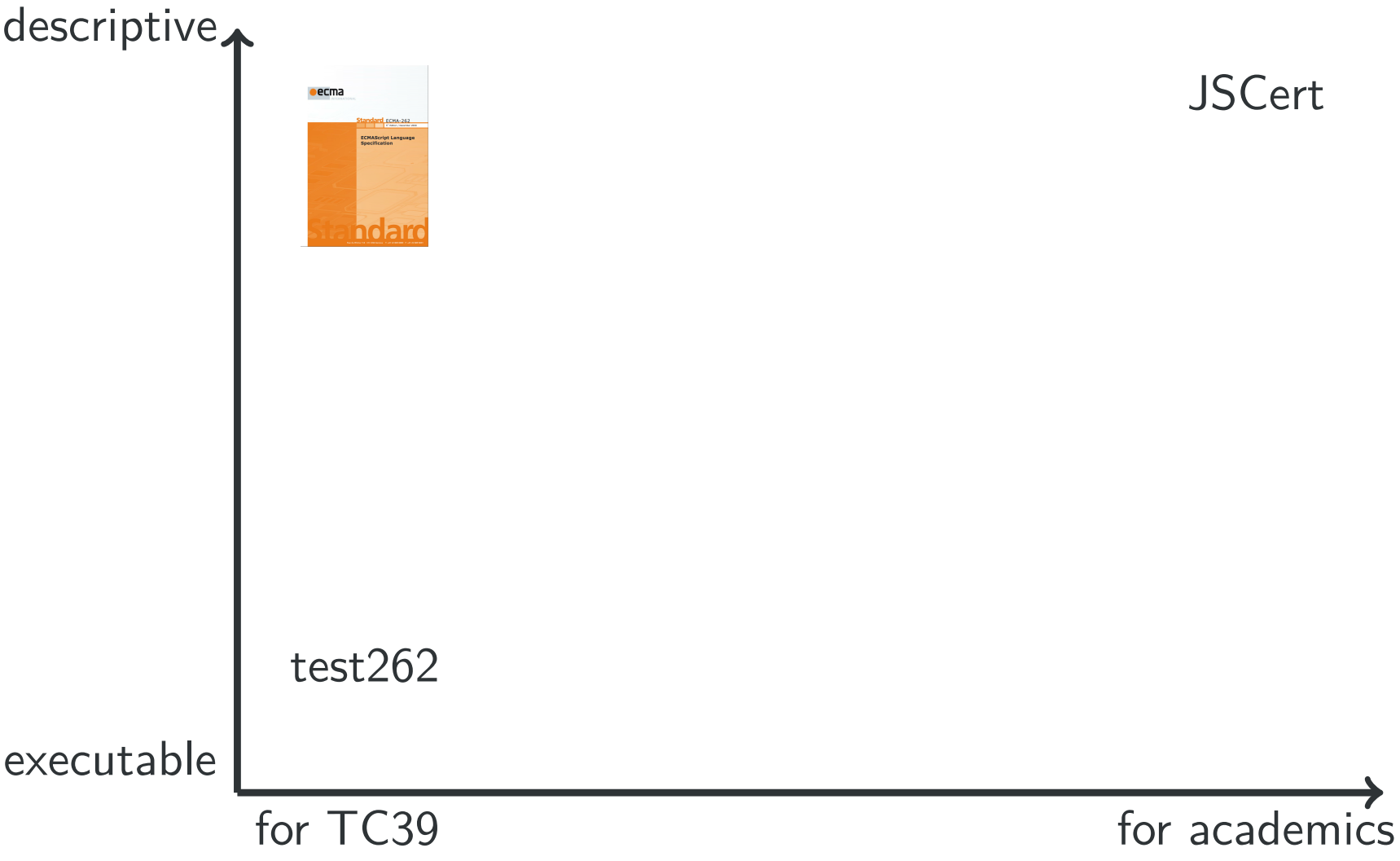
The Big Picture: JSCert (1/2)

```
(** Binary op, common rules for non-lazy operators *)

| red_expr_binary_op : forall S C op e1 e2 y1 o ,
  regular_binary_op op ->
  red_spec S C (spec_expr_get_value e1) y1 ->
  red_expr S C (expr_binary_op_1 op y1 e2) o ->
  red_expr S C (expr_binary_op e1 op e2) o

| red_expr_binary_op_1 : forall S0 S C op v1 e2 y1 o,
  red_spec S C (spec_expr_get_value e2) y1 ->
  red_expr S C (expr_binary_op_2 op v1 y1) o ->
  red_expr S0 C (expr_binary_op_1 op (ret S v1) e2) o

| red_expr_binary_op_2 : forall S0 S C op v1 v2 o,
  red_expr S C (expr_binary_op_3 op v1 v2) o ->
  red_expr S0 C (expr_binary_op_2 op v1 (ret S v2)) o
```

The Big Picture: JSCert (2/2)

```
(** Binary op : addition (11.6.1) *)

| red_expr_binary_op_add : forall S C v1 v2 y1 o,
  red_spec S C (spec_convert_twice (spec_to_primitive_auto v1)
                                   (spec_to_primitive_auto v2)) y1 ->
  red_expr S C (expr_binary_op_add_1 y1) o ->
  red_expr S C (expr_binary_op_3 binary_op_add v1 v2) o

| red_expr_binary_op_add_1_string : forall S0 S C v1 v2 y1 o,
  (type_of v1 = type_string \/ type_of v2 = type_string) ->
  red_spec S C (spec_convert_twice (spec_to_string v1) (spec_to_string v2)) y1 ->
  red_expr S C (expr_binary_op_add_string_1 y1) o ->
  red_expr S0 C (expr_binary_op_add_1 (ret S (v1,v2))) o

| red_expr_binary_op_add_string_1 : forall S0 S C s1 s2 s,
  s = String.append s1 s2 ->
  red_expr S0 C (expr_binary_op_add_string_1 (ret S (value_prim s1,value_prim s2)))
                (out_ter S s)

| red_expr_binary_op_add_1_number : forall S0 S C v1 v2 y1 o,
  ~ (type_of v1 = type_string \/ type_of v2 = type_string) ->
  red_spec S C (spec_convert_twice (spec_to_number v1) (spec_to_number v2)) y1 ->
  red_expr S C (expr_puremath_op_1 JsNumber.add y1) o ->
  red_expr S0 C (expr_binary_op_add_1 (ret S (v1,v2))) o
```

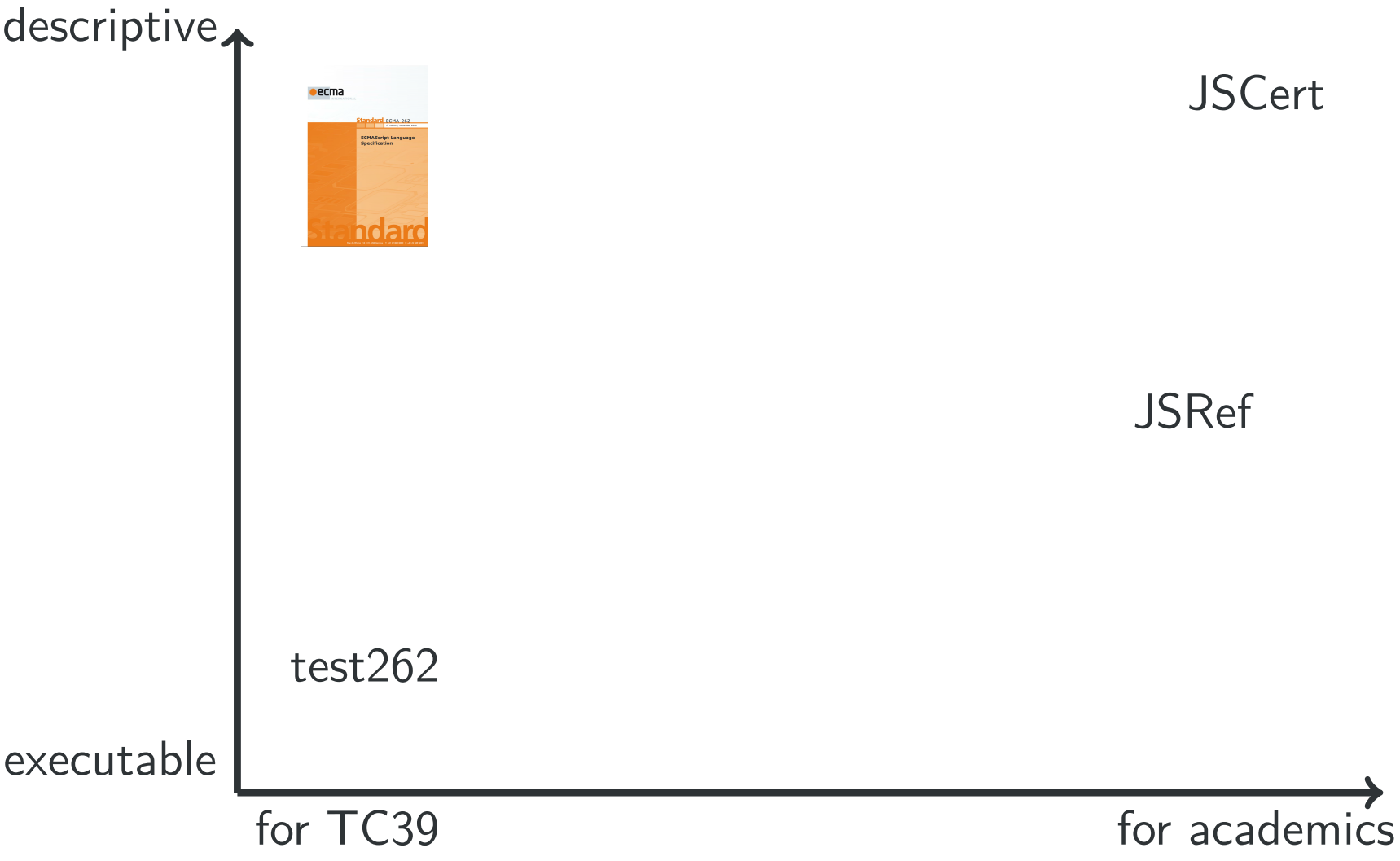
The Big Picture



The Big Picture: JSRef

```
Definition run_expr_binary_op r S C op e1 e2 :=
  if not (is_lazy_op op) then
    if_spec (run_expr_get_value r S C e1) (fun S1 v1 =>
      if_spec (run_expr_get_value r S1 C e2) (fun S2 v2 =>
        run_binary_op r S2 C op v1 v2))
  else ...

Definition run_binary_op r S C op v1 v2 :=
   If op = binary_op_add then
    if_spec (convert_twice to_primitive r S C (v1,v2)) (fun S1 (w1,w2) =>
      If type_of w1 = type_string \/ type_of w2 = type_string then
        if_spec (convert_twice to_string r S1 C (w1,w2)) (fun S2 (s1,s2) =>
          res_ter S2 (JsString.append s1 s2))
        else
          if_spec (convert_twice to_number r S1 C (w1,w2)) (fun S2 (n1,n2) =>
            res_ter S2 (JsNumber.add n1 n2)))
   else ...
```

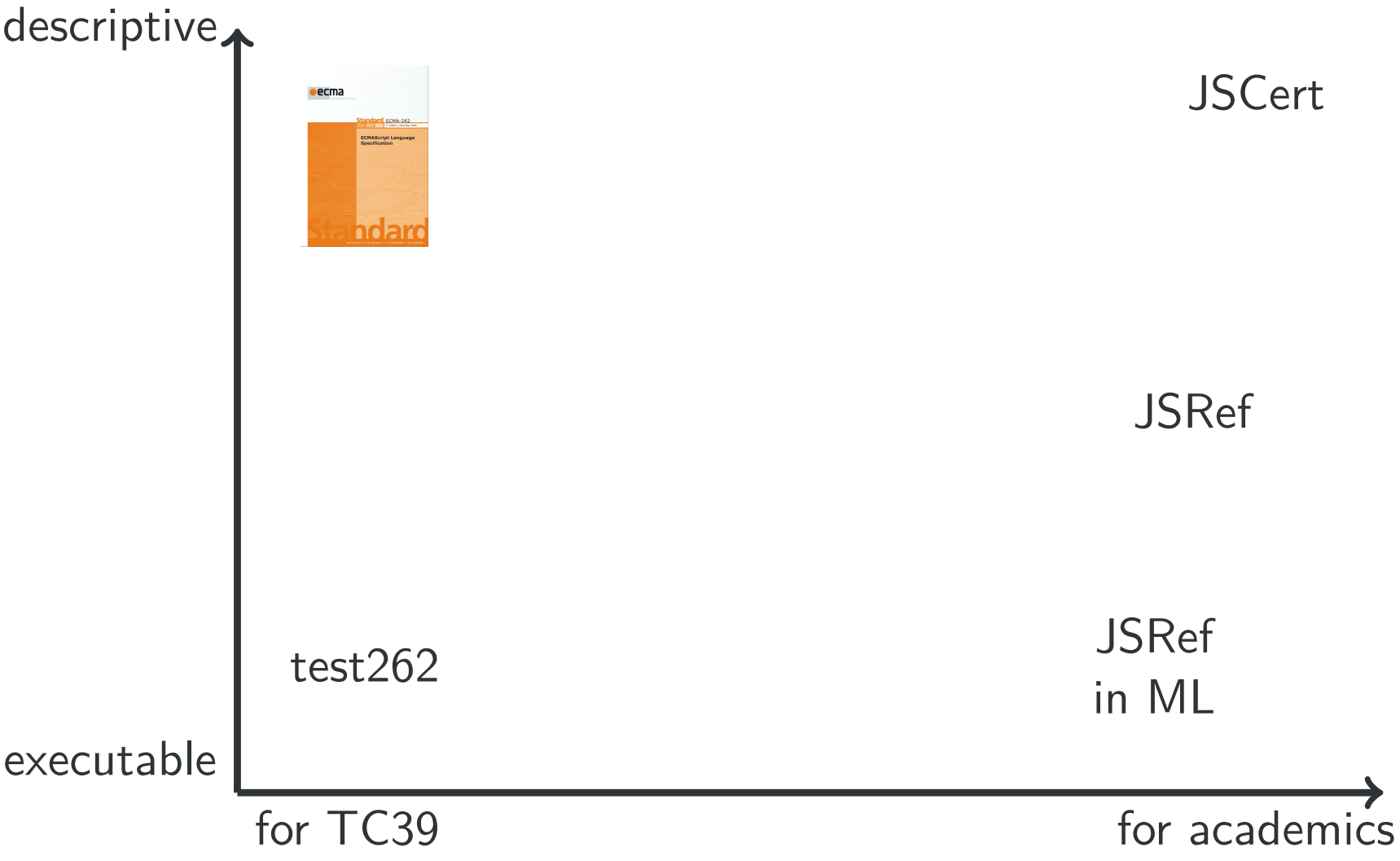
The Big Picture



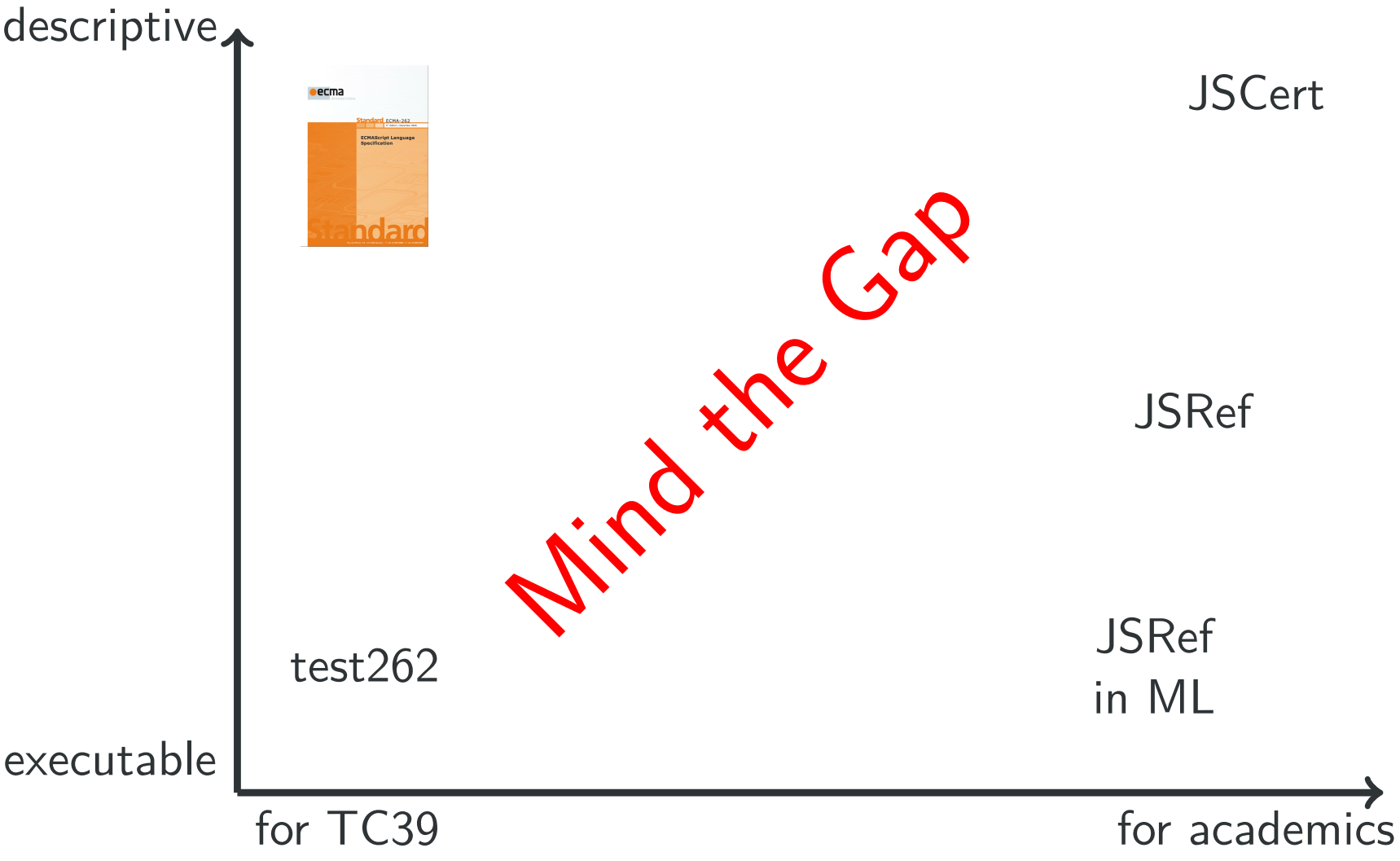
The Big Picture: Extracted OCaml

```
let run_expr_binary_op runs0 s c op e1 e2 =
  if not (is_lazy_op op) then
    if_spec (run_expr_get_value runs0 s c e1) (fun s1 v1 ->
      if_spec (run_expr_get_value runs0 s1 c e2) (fun s2 v2 ->
        run_binary_op runs0 s2 c op v1 v2))
  else ...

let run_binary_op runs0 s c op v1 v2 =
  if binary_op_comparable op Coq_binary_op_add
  then if_spec (convert_twice_primitive runs0 s c v1 v2) (fun s1 ww ->
         let (w1, w2) = ww in
         if or_decidable
              (type_comparable (type_of (Coq_value_prim w1)) Coq_type_string)
              (type_comparable (type_of (Coq_value_prim w2)) Coq_type_string)
         then if_spec
                (convert_twice_string runs0 s1 c (Coq_value_prim w1)
                  (Coq_value_prim w2)) (fun s2 ss ->
                let (s3, s4) = ss in
                res_out (Coq_out_ter (s2,
                  (res_val (Coq_value_prim (Coq_prim_string (append s3 s4)))))))
         else if_spec
                (convert_twice_number runs0 s1 c (Coq_value_prim w1)
                  (Coq_value_prim w2)) (fun s2 nn ->
                let (n1, n2) = nn in
```

The Big Picture



The Big Picture



Proposal: a new JSRef

```
and run_expr_binary_op s c op e1 e2 =
  match op with
  | Coq_binary_op_and -> run_binary_op_and s c e1 e2
  | Coq_binary_op_or -> run_binary_op_or s c e1 e2
  | _ ->
    let%run (s1,v1) = run_expr_get_value s c e1 in
    let%run (s2,v2) = run_expr_get_value s1 c e2 in
    run_binary_op s2 c op v1 v2

and run_binary_op s c op v1 v2 =
  match op with
  | Coq_binary_op_add -> run_binary_op_add s c v1 v2
  ...

and run_binary_op_add s c v1 v2 =
  let%run (s1, (w1, w2)) = (convert_twice_primitive s c v1 v2) in
  if  (type_compare (type_of w1) Coq_type_string)
   || (type_compare (type_of w2) Coq_type_string)
  then let%run (s2, (str1, str2)) = (convert_twice_string s1 c w1 w2) in
    res_out (Coq_out_ter (s2, (res_val (Coq_value_string (strappend str1 str2)))))
  else let%run (s2, (n1, n2)) = (convert_twice_number s1 c w1 w2) in
    res_out (Coq_out_ter (s2, (res_val (Coq_value_number (n1 +. n2))))))
```

Goals of the new JSRef

- tiny subset of OCaml
- extensive use of monadic operators to prevent this:

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be GetValue(*lref*).
3. ReturnIfAbrupt(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be GetValue(*rref*).
6. ReturnIfAbrupt(*rval*).
7. Let *lprim* be ToPrimitive(*lval*).
8. ReturnIfAbrupt(*lprim*).

Goals of the new JSRef

ES2016 (Section 5.2)

Abstract operations [...] that are prefixed by ? indicate that ReturnIfAbrupt should be applied to the resulting Completion Record. For example, ? operationName() is equivalent to ReturnIfAbrupt(operationName()).

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *lprim* be ? ToPrimitive(*lval*).

Monadic operators

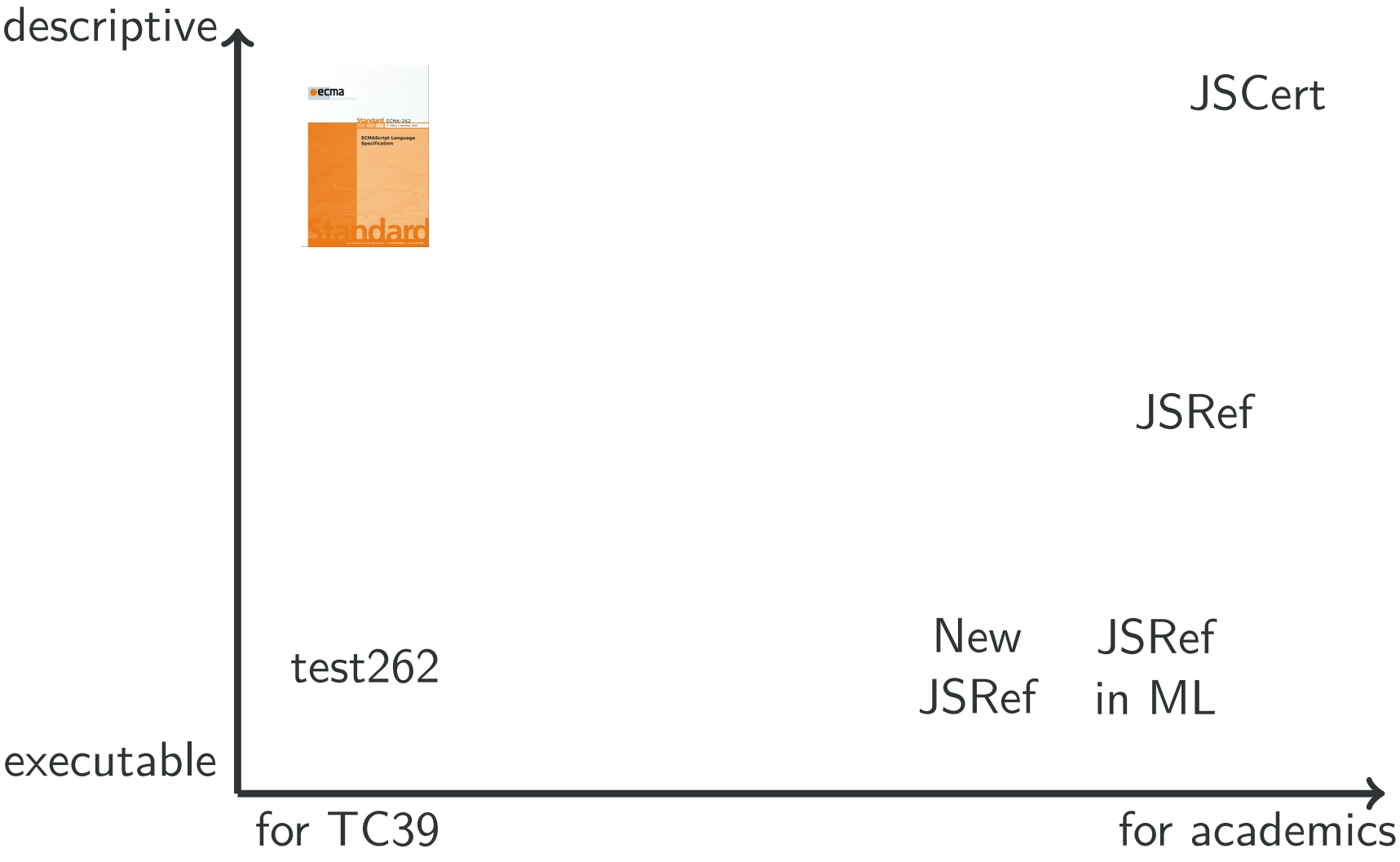
- `let%run (s,r) = e in …`
  1. execute `e`
  2. make sure there is no internal problem (is so, propagate it)
  3. check the result type, if it's not `normal`, then propagate it
  4. bind the state and value to `s` and `r` and continue

Subset of OCaml

pure functional language (no side effects)

- types: bool, string, int, float
- algebraic datatypes, tuples, records
- simple pattern matching
- no partial application
- custom notation for monadic operators (ppx)

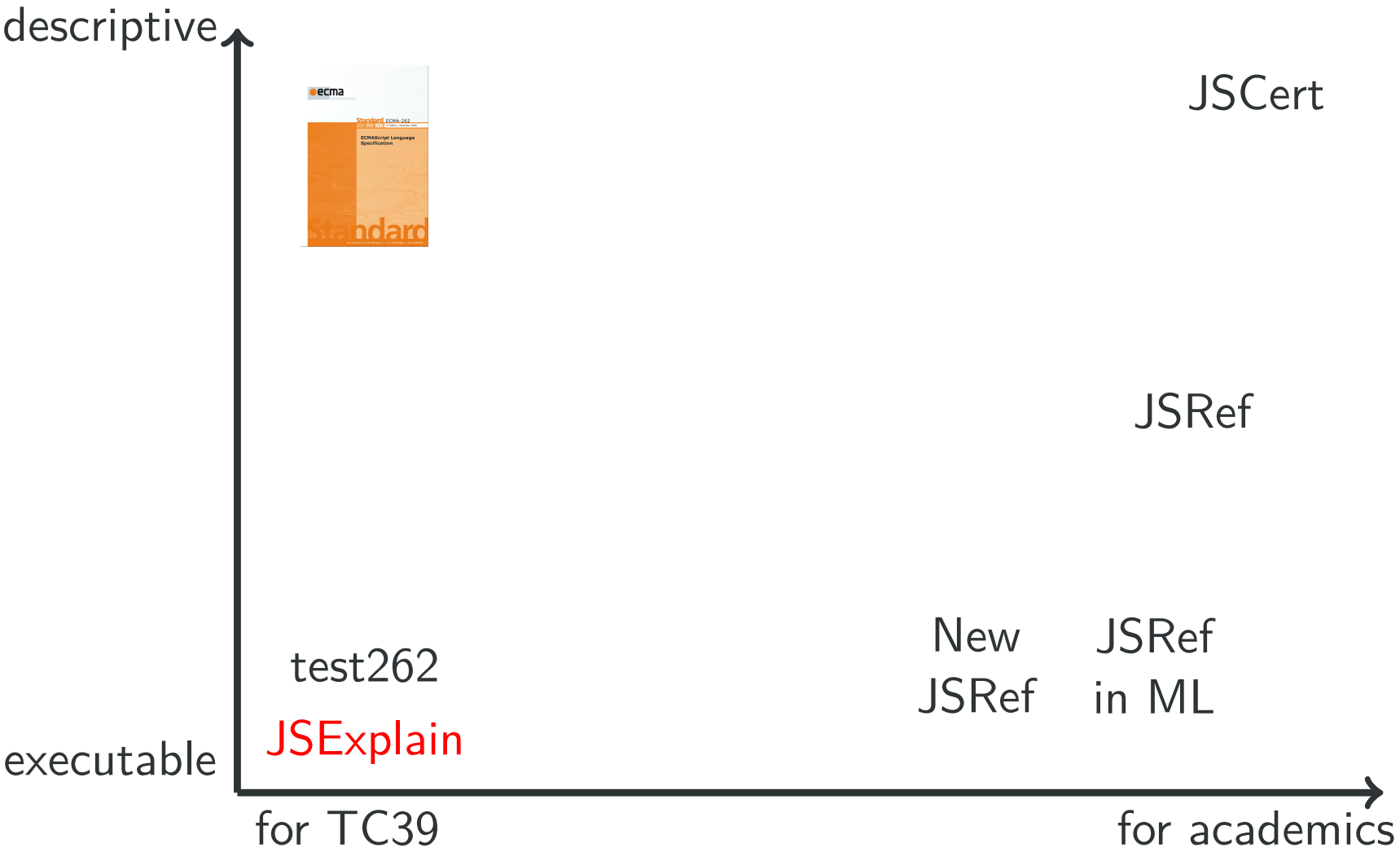
The Big Picture



From New JSRef to JSExplain

- OCaml might still be too academic
- tighten the connection between the interpreter and the spec
  - symbolic execution of the specification (request at Dagstuhl in 2014)
- how about translating our code to JavaScript and instrumenting it?

The Big Picture



Demo

jsexplain

The ultimate polyfill

tiny subset of JavaScript, used as a functional language

- types: bool, string, number
- no type conversion (i.e., it's a typed JS)
- functions
- objects for encoding records and algebraic datatypes (with a `tag` field), no use of prototypes
- arrays for encoding tuples
- switch on strings for pattern matching
- `Object.assign`

How logging works: Events

The compiler outputs *trace generating* code, that tracks events

Event contents

- a filename
- a token (position in the file)
- a context
- a type

Event types

- function call
- entering a function
- doing a return
- branching on a if
- branching on a switch
- choosing a case in a switch
- binding a variable

How logging works: Contexts

Contexts are chained and track the value of variables in scope. The chain is extended each time a variable is bound:

- entering a function, with the arguments
- entering a let
- entering the branch of a match, if there are binders

How logging works: Applied to JSExplain

context inspection for bindings with well-known names or fields (used in the interpreter):

- `_term_` is the term being evaluated
- state has a `state_object_heap` field
- execution context has a `execution_ctx_lexical_env` field

display of this information separately, alongside the interpreted program

Pseudo code

- some functions treated as “coercions” are hidden
- all arguments of type state or context are hidden
- use of pattern maching notation (similar to the original code)
- custom monadic notation is used (similar to the original code)

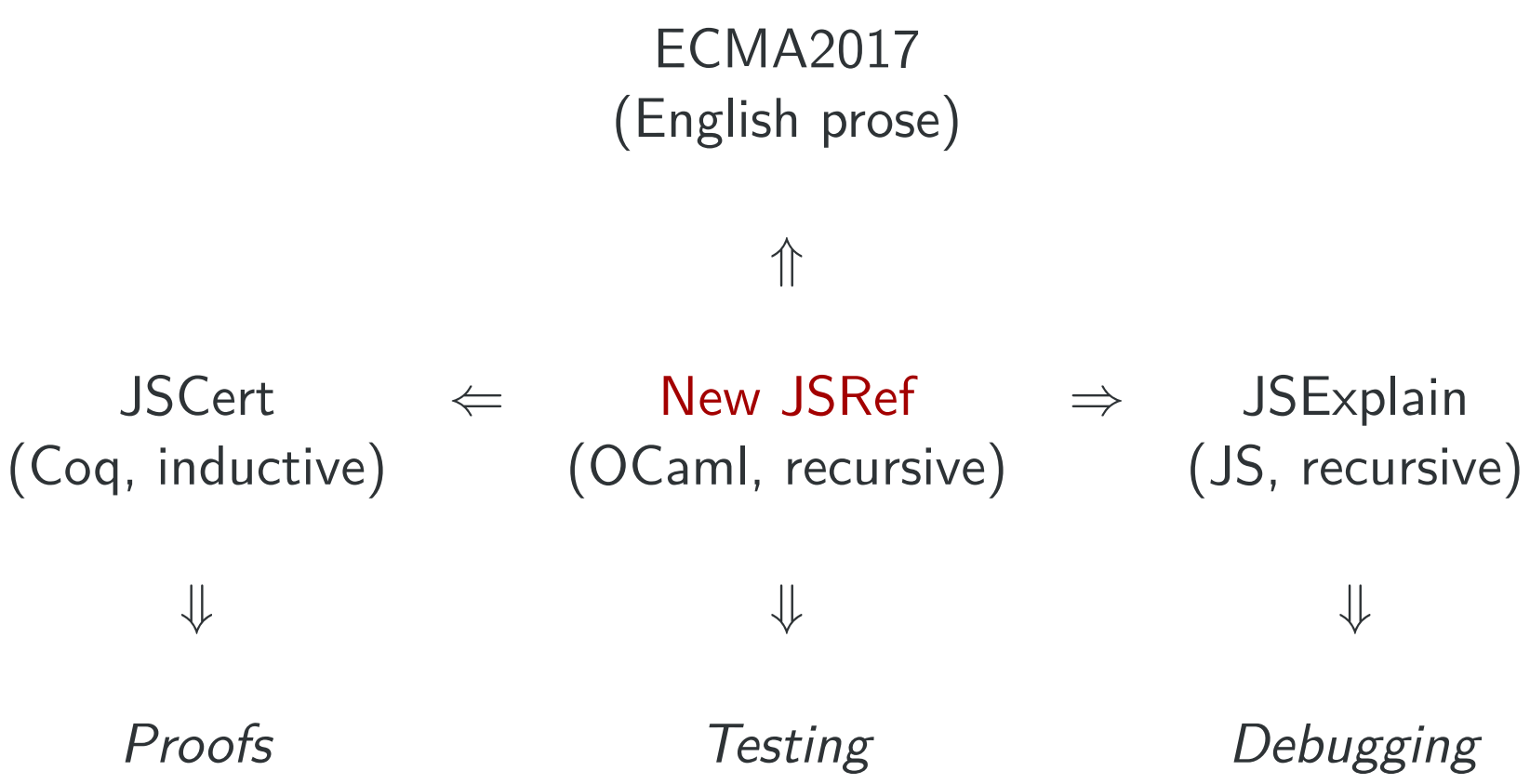
Why not use BuckleScript?

- we target a very tiny subset of OCaml
- we want to use the smallest subset of JS
- we need to generate annotated code for the interactive debugger
- we can always go back to JS when we don't know what to do (floating point numbers comparison)

Industrial plans

- public release (documentation needed)
- implementing ES2015 and ES2016
  - proxies are ongoing
- lobby for use in TC39
- switch to Reason?

Research plans



- readable
- easier to extend (ultimate polyfill)
- easier to instrument