JSExplain

Alan Schmitt, with Arthur Charguéraud and Thomas Wood

May 25, 2016

Motivation

Equip JavaScript with

1. a formal, executable specification
2. logics and tools for verification
3. formal proofs of security properties

Quick JSCert history

ECMA5
(English prose)

JSCert (Coq, inductive) $\sim$ JSRef (Coq, recursive) $\Rightarrow$ JSRef – extracted (OCaml, recursive)

$\Downarrow$ $\Downarrow$

Proofs *Testing*

JSCert (800 rules)

```
(** 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
```

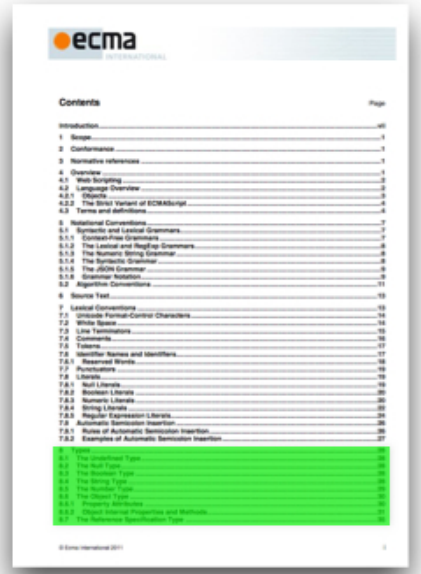
JSCert (800 rules)

```
(** 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
```

JSRef

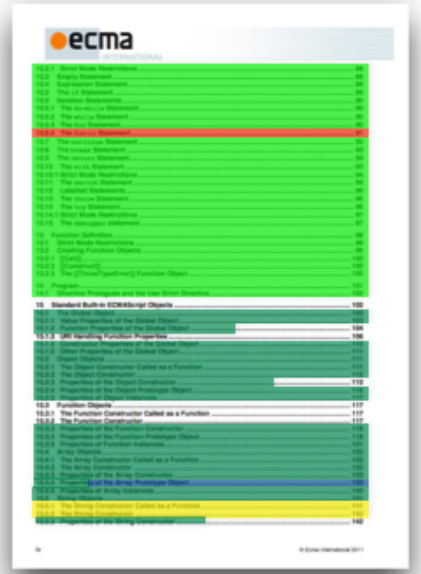
```
Definition run_expr_binary_op r S C op e1 e2 :=
  if not (is_lazy_op op) then
    if_spec (run_expr_get_value r S C e1) (fun S1 v1 =>
      if_spec (run_expr_get_value r S1 C e2) (fun S2 v2 =>
        run_binary_op r S2 C op v1 v2))
  else ...

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 ...
```

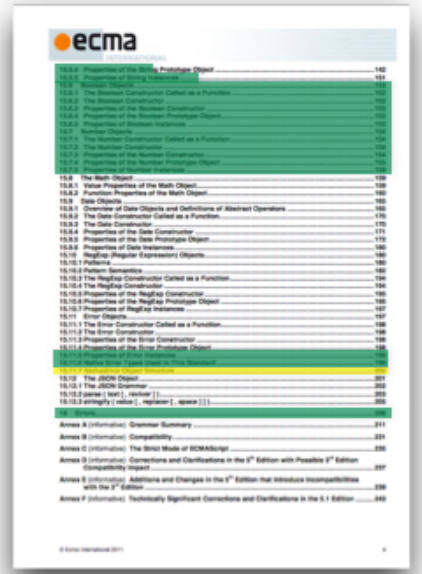
What we have

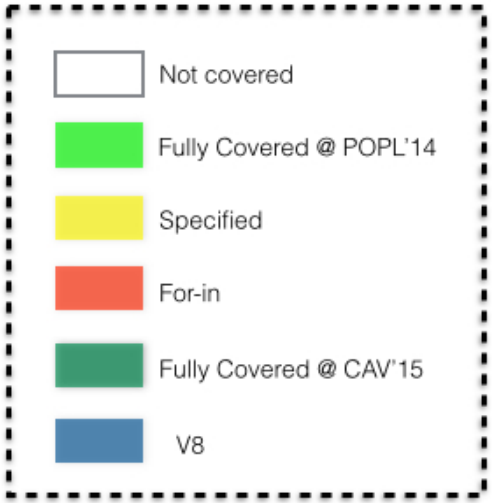

ecma
Contents


ecma


ecma


ecma


ecma

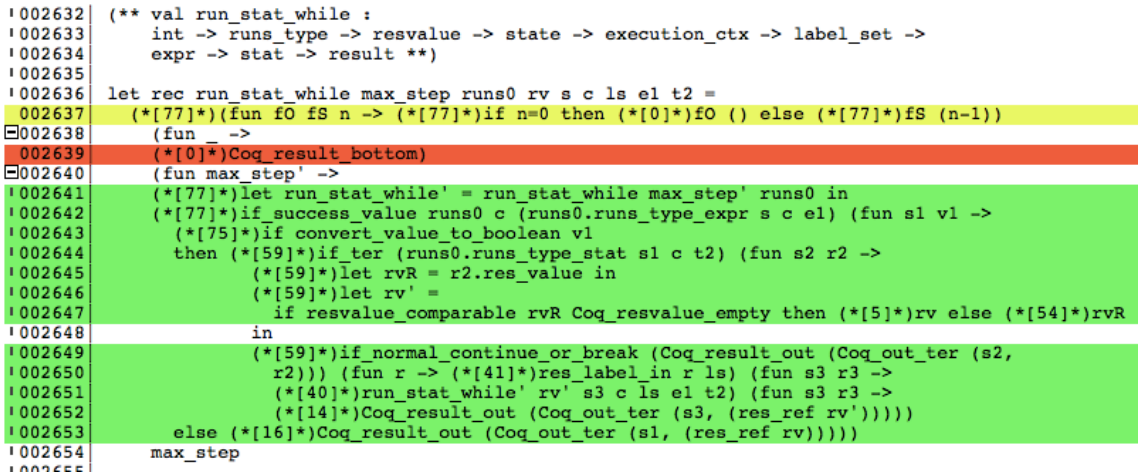

Not covered
Fully Covered @ POPL'14
Specified
For-in
Fully Covered @ CAV'15
V8

Evaluation

- it's already useful
  - currently used to prove the desugaring of λJS
  - testing coverage

Evaluation

- it's already useful
  - currently used to prove the desugaring of λJS
  - testing coverage



```
!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 fO fS n -> (*[77]*)if n=0 then (*[0]*)fO () 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
```

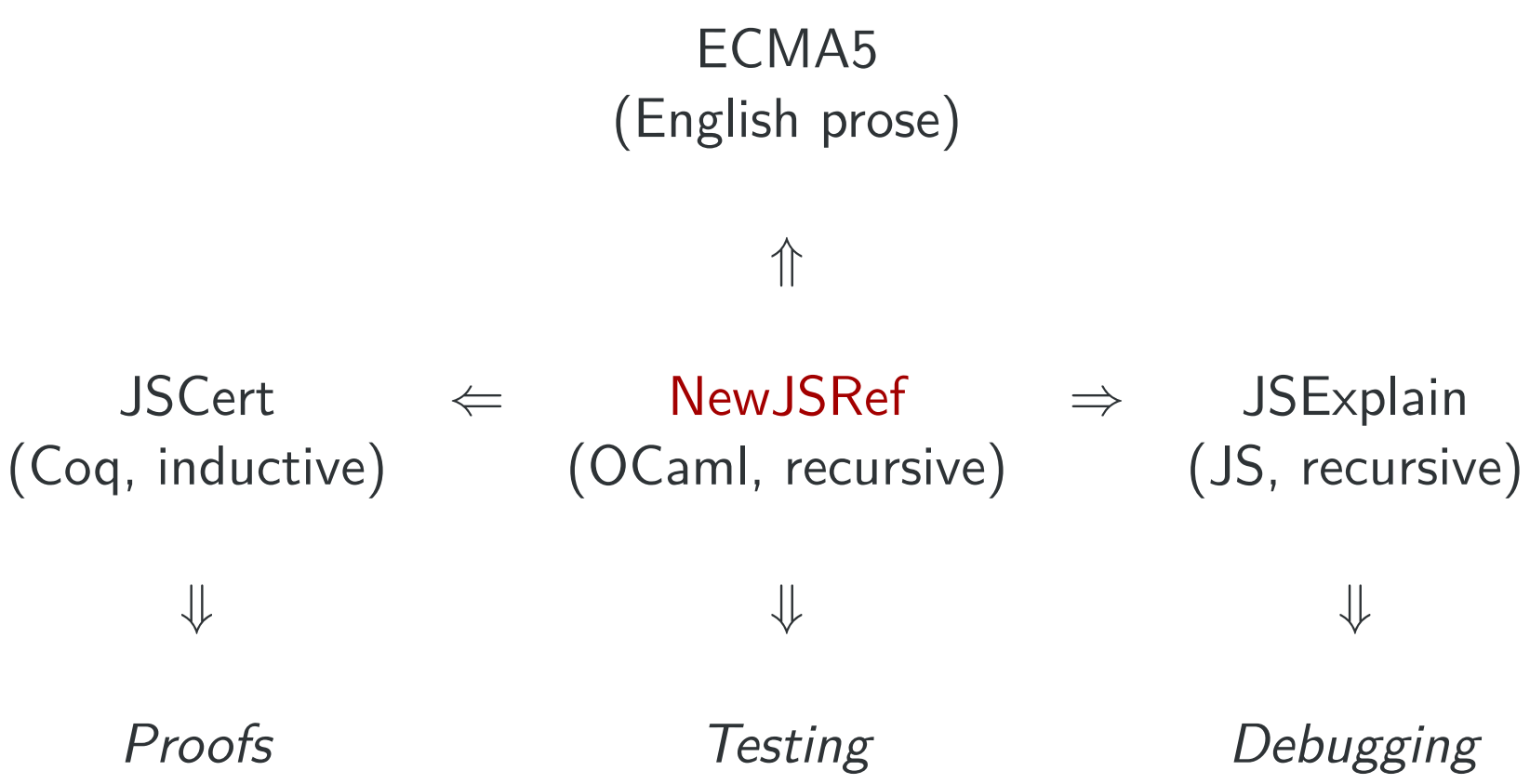
Evaluation

- it's already useful
  - currently used to prove the desugaring of λJS
  - testing coverage

```
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 fO fS n -> (*[77]*)if n=0 then (*[0]*)fO () 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
```

- issues
  - hard to read
  - hard to maintain
  - extending to ES6 and beyond is daunting

New approach



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

Demo

http://ajacs.inria.fr/jsexplain/driver.html

Language status

Source (ML) 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

Target (JS) used as a functional language

- types: bool, string, number
- objects for encoding records and algebraic datatypes (with a `tag` field)
- arrays for encoding tuples
- switch for pattern matching

Logging

- enter function
- exit function
- case of switch and if
- binding of variable

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)

Questions to move forward

- Is this useful for you?
- What could make it more useful?
- What do you want the language to look like?
- What would incite you to test features in that language?