Interactive Execution of a Formal Semantics

Arthur Charguéraud
Inria

Joint work with: Alan Schmitt (Inria) and Thomas Wood (Imperial College London)

Building on prior work, joint with:
M. Bodin, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, and G. Smith.

2016/07/04
Gallium seminar

A glance at ECMA5 (280 pages)

The addition operator either performs string concatenation or numeric addition.

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*).

A glance at ECMA6 (560 pages)

The addition operator either performs string concatenation or numeric addition.

Evaluation of: *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).
 9. Let rprim be ToPrimitive(rval).
10. ReturnIfAbrupt(rprim).
11. If Type(lprim) is String or Type(rprim) is String, then
    11.1 let lstr be ToString(lprim).
    11.2 ReturnIfAbrupt(lstr).
    11.3 let rstr be ToString(rprim).
    11.4 ReturnIfAbrupt(rstr).
    11.5 Return the String that is the result of concatenating lstr and rstr.
12. let lnum be ToString(lprim).
13. ReturnIfAbrupt(lnum).
14. let rnum be ToString(rprim).
15. ReturnIfAbrupt(rnum).
16. Return the result of applying the addition operation to lnum and rnum.
```

Definition of ReturnIfAbrupt

6.2.2.4 ReturnIfAbrupt

Algorithms steps that say

1. ReturnIfAbrupt(*argument*).

mean the same thing as:

1. If *argument* is an abrupt completion, return *argument*.
2. Else if *argument* is a Completion Record, let *argument* be *argument*.[[value]].

POPL'14 formalization

$$\begin{tikzcd}
 & \begin{array}{c}\text{ECMA5}\\ \text{(English prose)}\end{array} & \\
\begin{array}{c}\text{JsCert}\\ \text{(Coq, inductive)}\end{array} \arrow[r, phantom, "\sim"] \arrow[d, Rightarrow] & \begin{array}{c}\text{JsRef}\\ \text{(Coq, recursive)}\end{array} \arrow[r, Rightarrow] & \begin{array}{c}\text{JsRef-extracted}\\ \text{(OCaml, recursive)}\end{array} \arrow[d, Rightarrow] \\
\textit{Proofs} & & \textit{Testing}
\end{tikzcd}$$

A glance at JsCert (800 rules, 3000 loc)

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

| red_expr_binary_op : ∀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 : ∀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 : ∀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

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

| red_expr_binary_op_add : ∀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 : ∀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 : ∀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 : ∀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
```

A glance at JsRef (2000 loc)

```
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 ...
```

Obstacles to adoption by the JavaScript committee

$$\begin{tikzcd}
 & \begin{array}{c}\text{ECMA5}\\ \text{(English prose)}\end{array} & \\
\begin{array}{c}\text{JsCert}\\ \text{(Coq, inductive)}\end{array} \arrow[r, phantom, "\sim"] \arrow[d, Rightarrow] & \begin{array}{c}\text{JsRef}\\ \text{(Coq, recursive)}\end{array} \arrow[r, Rightarrow] & \begin{array}{c}\text{JsRef-extracted}\\ \text{(OCaml, recursive)}\end{array} \arrow[d, Rightarrow] \\
\textit{Proofs} & & \textit{Testing}
\end{tikzcd}$$

We wanted to:

- avoid the need to maintain JsCert and JsRef independently,
- provide tools for executing the specification step by step,
- close the gap between the English prose and the Coq code.

NewJsRef

$$\begin{tikzcd}
 & \begin{array}{c}\text{ECMA}\\ \text{(English prose)}\end{array} & \\
\begin{array}{c}\text{JsCert}\\ \text{(Coq, inductive)}\end{array} \arrow[d, Rightarrow] & \begin{array}{c}\text{NewJsRef}\\ \text{(Core-ML, recursive)}\end{array} \arrow[u, Rightarrow] \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \begin{array}{c}\text{JsExplain}\\ \text{(JS, recursive)}\end{array} \arrow[d, Rightarrow] \\
\textit{Proofs} & \textit{Testing} & \textit{Debugging}
\end{tikzcd}$$

NewJsRef: example

```
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, (s3, s4)) = (convert_twice_string s1 c w1 w2) in
    res_out (Coq_out_ter (s2, (res_val (Coq_value_string (strappend s3 s4)))))
  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)))))
```

NewJsRef: a functional semantics

Language used to write the interpreter:

1. core ML, without any side-effects,
2. with modules for namespaces,
3. with bool, string, int, and float,
4. with algebraic datatypes, tuples, and records,
5. with simple pattern matching (non-nested),
6. with n-ary functions (no partial applications),
7. with custom notation for monadic operators (using ppx).

JsExplain

$$\begin{tikzcd}
 & \begin{matrix}\text{ECMA}\\ \text{(English prose)}\end{matrix} & \\
\begin{matrix}\text{JsCert}\\ \text{(Coq, inductive)}\end{matrix} \arrow[d, Rightarrow] & \begin{matrix}\text{NewJsRef}\\ \text{(Core-ML, recursive)}\end{matrix} \arrow[u, Rightarrow] \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \begin{matrix}\text{JsExplain}\\ \text{(JS, recursive)}\end{matrix} \arrow[d, Rightarrow] \\
\textit{Proofs} & \textit{Testing} & \textit{Debugging}
\end{tikzcd}$$

Intended users: JS committee members, VM developers, test engineers, or any programmer curious about JS behavior.

JsExplain

Demo.

JsExplain: overview

From the functional semantics in core-ML, we generate:

1. an equivalent functional semantics expressed in JavaScript,
2. a variant instrumented with logging instructions,
3. a variant with syntactic sugar to improve readability.

JsExplain 1: translation to JavaScript

Naive translation from core-ML to JavaScript syntax.

Subset of JS used:

1. core JavaScript, with variables and functions,
2. with bool, string, and number (for int and float),
3. with objects (for records and algebraic datatypes),
4. with arrays (for tuples),
5. with switch (for pattern matching),
6. with “with” (for opening scopes).

Result: a JavaScript interpreter, implemented in human-readable JS code.

JsExplain 1: translation to JavaScript, example code

```
var run_binary_op_add = function(s, c, v1, v2) {
  return (if_run(convert_twice_primitive(s, c, v1, v2), function(s1, arg_22) {
    var w1 = arg_22[0], w2 = arg_22[1];
    if ((type_compare(type_of(w1), Coq_type_string())
       || type_compare(type_of(w2), Coq_type_string()))) {
     return (if_run(convert_twice_string(s1, c, w1, w2), function(s2, arg_21) {
           var s3 = arg_21[0], s4 = arg_21[1];
           return (res_out(Coq_out_ter(s2, res_val(
              Coq_value_string(strappend(s3, s4))))));
         }));
   } else {
     return (if_run(convert_twice_number(s1, c, w1, w2), function(s2, arg_20) {
         var n1 = arg_20[0], n2 = arg_20[1];
         return (res_out(Coq_out_ter(s2, res_val(
           Coq_value_number((n1 + n2))))));
       }));
   }}));
};
```

JsExplain 2: generation of execution traces

Extend the translation with generation of logging instructions for:

1. enter in a function (also record the arguments),
2. return from a function (also record the result value),
3. let-bound variables (also record the bindings),
4. branch entered in a conditional/switch.

All events also log the location in interpreter and interpreted program.

Result: a complete execution trace for a given input program.

JsExplain 2: generation of execution traces, example code

```
var run_binary_op_add = function (s, c, v1, v2) {
  var ctx_744 = ctx_push(ctx_empty, [
    {key: "s", val: s}, {key: "c", val: c},
    {key: "v1", val: v1}, {key: "v2", val: v2}]);
  log_event("JsInterpreter.js", 4004, ctx_744, "enter");
  var _return_1713 = if_run((function () {
    log_event("JsInterpreter.js", 3974, ctx_744, "call");
    var _return_1697 = convert_twice_primitive(s, c, v1, v2);
    log_event("JsInterpreter.js", 3973, ctx_push(ctx_744,
     [{key: "#RETURN_VALUE#", val: _return_1697}]), "return");
    return (_return_1697); }()), function(s1, arg_1712) {
    var w1 = arg_1712[0], w2 = arg_1712[1];
    var ctx_745 = ctx_push(ctx_744,
     [{key: "s1", val: s1}, {key: "w1", val: w1}, {key: "w2", val: w2}]);
    log_event("JsInterpreter.js", 4002, ctx_745, "let");
    var _ifarg_1698 = (function () {
      ...
```

JsExplain 3: syntactic sugar

Alter the translation to target an hypothetical language with:

1. the flavor of JavaScript syntax (adapted to the audience),
2. custom notation for monadic operators (like in core-ML),
3. custom notation for ML-style pattern matching (like in core-ML),
4. hiding of calls to functions declared as coercions (like in Coq),
5. hiding of values representing states or contexts (as if imperative).

Result: easy-to-read functional semantics for JavaScript.

JsExplain 3: syntactic sugar, example code

```
var run_expr_binary_op = function(op, e1, e2) {
  switch (op) {
    case Coq_binary_op_and:
      return run_binary_op_and(e1, e2);
    case Coq_binary_op_or:
      return run_binary_op_or(e1, e2);
    default:
      var%run v1 = run_expr_get_value(e1);
      var%run v2 = run_expr_get_value(e2);
      return run_binary_op(op, v1, v2);
  }
};
var run_binary_op_add = function(v1, v2) {
  var%run (w1, w2) = convert_twice_primitive(v1, v2);
  if ((type_compare(type_of(w1), Type_string)
      || type_compare(type_of(w2), Type_string))) {
    var%run (s3, s4) = convert_twice_string(w1, w2);
    return strappend(s3, s4);
  } else {
    var%run (n1, n2) = convert_twice_number(w1, w2);
    return n1 + n2;
  } };
```

JsEnglish

$$\begin{tikzcd}
 & \begin{array}{c}\text{ECMA}\\ \text{(English prose)}\end{array} & \\
\begin{array}{c}\text{JsCert}\\ \text{(Coq, inductive)}\end{array} \arrow[d, Rightarrow] & \begin{array}{c}\text{NewJsRef}\\ \text{(OCaml, recursive)}\end{array} \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[u, Rightarrow] \arrow[d, Rightarrow] & \begin{array}{c}\text{JsExplain}\\ \text{(JS, recursive)}\end{array} \arrow[d, Rightarrow] \\
\textit{Proofs} & \textit{Testing} & \textit{Debugging}
\end{tikzcd}$$

Towards a replacement for ECMA6's prose

The addition operator either performs string concatenation or numeric addition.

Evaluation of: *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($lprim$).
9. Let $rprim$ be ToPrimitive($rval$).
10. ReturnIfAbrupt($rprim$).
11. If Type($lprim$) is String or Type($rprim$) is String, then
    11.1 let $lstr$ be ToString($lprim$).
    11.2 ReturnIfAbrupt($lstr$).
    11.3 let $rstr$ be ToString($rprim$).
    11.4 ReturnIfAbrupt($rstr$).
    11.5 Return the String that is the result of concatenating $lstr$ and $rstr$.
12. let $lnum$ be ToString($lprim$).
13. ReturnIfAbrupt($lnum$).
14. let $rnum$ be ToString($rprim$).
15. ReturnIfAbrupt($rnum$).
16. Return the result of applying the addition operation to $lnum$ and $rnum$.
```

JsEnglish: generate English prose (future work)

Evaluation of: *expr1* + *expr2*

1. Let' *lval* be EvalExpressionGetValue(*expr1*).
2. Let' *rval* be EvalExpressionGetValue(*expr2*).
3. Let' *lprim* be ToPrimitive(*lval*).
4. Let' *rprim* be ToPrimitive(*rval*).
5. If Type(*lprim*) is String or Type(*rprim*) is String, then
    - Let' *lstr* be ToString(*lprim*).
    - Let' *rstr* be ToString(*rprim*).
    - Return the value computed as the string concatenation of *lstr* and *rstr*.
6. Let' *lnum* be ToNumber(*lprim*).
7. Let' *rnum* be ToNumber(*rprim*).
8. Return the value computed as the number addition of *lnum* and *rnum*.

where "Let' x be e" is:
"Let c be e ; If c is abrupt Then return c Else Let x be c.[[value]]".

JsCert: from a recursive to an inductive definition

$$\begin{tikzcd}
 & \begin{array}{c}\text{ECMA}\\ \text{(English prose)}\end{array} & \\
\begin{array}{c}\text{JsCert}\\ \text{(Coq, inductive)}\end{array} \arrow[d, Rightarrow] & \begin{array}{c}\text{NewJsRef}\\ \text{(OCaml, recursive)}\end{array} \arrow[u, Rightarrow] \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \begin{array}{c}\text{JsExplain}\\ \text{(JS, recursive)}\end{array} \arrow[d, Rightarrow] \\
\textit{Proofs} & \textit{Testing} & \textit{Debugging}
\end{tikzcd}$$

From JsRef to JsCert

```
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, (s3, s4)) =
      (convert_twice_string s1 c w1 w2) in
    res_out (Coq_out_ter (s2, (
      res_val (Coq_value_string (strappend s3 s4)))))
  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)))))
```

```
| red_expr_binary_op : ∀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 : ∀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 : ∀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

| red_expr_binary_op_add : ∀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 : ∀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 : ∀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 : ∀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
```

Summary

$$\begin{tikzcd}
 & \begin{array}{c}\text{ECMA}\\\text{(English prose)}\end{array} & \\
\begin{array}{c}\text{JsCert}\\\text{(Coq, inductive)}\end{array} \arrow[d, Rightarrow] & \begin{array}{c}\text{NewJsRef}\\\text{(OCaml, recursive)}\end{array} \arrow[u, Rightarrow] \arrow[l, Rightarrow] \arrow[r, Rightarrow] \arrow[d, Rightarrow] & \begin{array}{c}\text{JsExplain}\\\text{(JS, recursive)}\end{array} \arrow[d, Rightarrow] \\
\textit{Proofs} & \textit{Testing} & \textit{Debugging}
\end{tikzcd}$$

Application to OCaml: `ocamlref`

In a similar way, we could define an untyped semantics for OCaml.

However, this would assign a semantics to many ill-typed programs.

```
type u = { x : int; y : int }
type v = { y : int; z : int }
let f a = a.y
let _ = f { x = 1; y = 2 }
let _ = f { y = 3; z = 4 }
```

Theorem:

- If `ocamlc` successfully type-checks and compiles e into a program p,
- then p evaluates to v on the machine
  $\iff$ e evaluates to v in the untyped semantics.
- and p diverges on the machine
  $\iff$ e diverges in the untyped semantics.

Thanks!