From JsCert Onwards

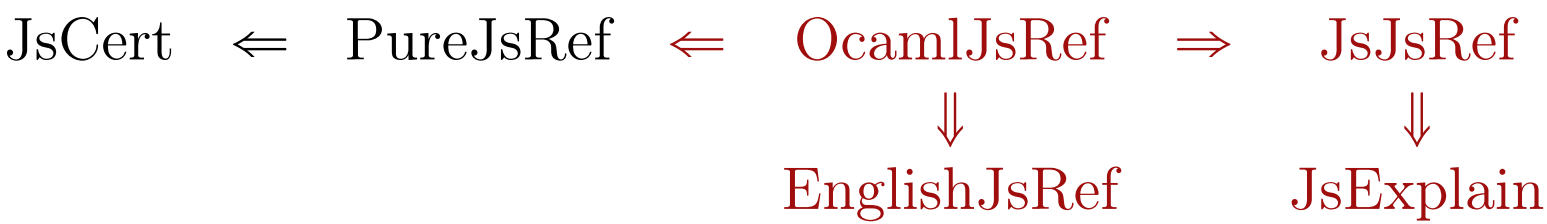
Formal Methods Meets JavaScript in Paris

Arthur Charguéraud

Inria

2015/03/23

Overview



In black: existing.
In red: future work.

Running example: addition

11.6.1 The Addition operator (+)

The addition operator either performs string concatenation or numeric addition. The production *AdditiveExpression* : *AdditiveExpression* + *MultiplicativeExpression* is evaluated as follows:

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*).

JsCert

JsCert: the best-suited presentation for formal proofs.

Useful for verified translation, type soundness, isolation property, ...

JsCert features:

- inductively-defined, pretty-big step semantics,
- explicit state, context, exceptions,
- maximal factorization of similar rules,
- great for proofs, not so much for humans.

JsCert example (1/2)

```
(** 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
```

JsCert example (2/2)

```
(** 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
```

PureJsRef

PureJsRef: an executable interpreter easily proved correct w.r.t. JsCert.

Was the shortest path for us to execute test suites against JsCert.

PureJsRef features:

- purely-functional code, explicit threading of state,
- monad for exceptions, break/continue/return, timeout,
- conditional on predicates, boolean code automatically derived.

PureJsRef example

```
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))
  ...

Definition run_binary_op r S C op v1 v2 :=
  If op = binary_op_add then
    if_spec (convert_twice_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_string r S1 C w1 w2) ((fun S2 (s1,s2) ⇒
          res_ter S2 (JsString.append s1 s2)))
        else
          if_spec (convert_twice_number r S1 C w1 w2) ((fun S2 (n1,n2) ⇒
            res_ter S2 (JsNumber.add n1 n2)))))
   ...
```

OcamlJsRef

OcamlJsRef: an executable interpreter with code easy to read.

OcamlJsRef features:

- use side effects for the state, the context, and the entry points.
- introduce notation for monadic operators,
- conditional on boolean expressions.

Idea: PureJsRef could be automatically generated from OCamlJsRef.

OcamlJsRef example

```
let run_expr_binary_op op e1 e2 =
  if not (is_lazy_op op) then
    Let v1 = run_expr_get_value e1 in
    Let v2 = run_expr_get_value e2 in
    run_binary_op op v1 v2
  ...

let run_binary_op op v1 v2 =
  if op == binary_op_add then
    Let w1 = to_primitive v1 in
    Let w2 = to_primitive v2 in
    if type_of w1 == type_string || type_of w2 == type_string then begin
       Let s1 = to_string w1 in
       Let s2 = to_string w2 in
       Return (JsString.append s1 s2)
    end else begin
       Let n1 = to_number w1 in
       Let n2 = to_number w2 in
       Return (JsNumber.add n1 n2)
    end
  ...
```

EnglishJsRef

EnglishJsRef: produce parts of the english specification automatically from OCamlJsRef.

EnglishJsRef features:

- output close to the style of ECMA specifications,
- flexibility for tuning the output for each definition.

EnglishJsRef example input

```
\defineRenamingPattern{
  "type_of" ⇒ "Type",
  "type_string" ⇒ "String",
  "get_value" ⇒ "GetValue",
  "to_primitive" ⇒ "ToPrimitive",
  "runs_expr($e)" ⇒ "the result of evaluating $e",
  "JsNumber.add($n1,$n2)" ⇒
      "the result of applying the addition operation to $n1 and $n2",
  "JsString.append($s1,$s2)" ⇒
      "the result of concatenating $s1 followed by $s2",
  ...
  }

\section{Binary operator ( + )}

The addition operator either performs string concatenation or numeric addition.

\generateFrom
  [specialization: op:=op_add]
  [inlining: run_expr_get_value, run_binary_op]
  [inlining-intermediate-applications-of: to_string, to_number]
  {run_expr_binary_op}
```

EnglishJsRef example output

The addition operator either performs string concatenation or numeric addition. The production "$e1 + e2$" is evaluated as follows:

1. Let $r1$ be the result of evaluating $e1$.
2. Let $v1$ be GetValue($r1$).
3. Let $r2$ be the result of evaluating $e2$.
4. Let $v2$ be GetValue($r2$).
5. Let $w1$ be ToPrimitive($v1$).
6. Let $w2$ be ToPrimitive($v2$).
7. If Type($w1$) is String or Type($w2$) is String, then
   - Return the result of concatenating ToString($w1$) followed by ToString($w2$).
8. Return the result of applying the addition operation to ToNumber($w1$) and ToNumber($w2$).

OcamlJsRef example 2

```
let run_stat t =
  match t with
  ...
  | stat_block ts ⇒ run_block ts

let run_block ts =
  if Seq.is_empty ts_rev then
    ReturnBehavior (res_intro restype_normal resvalue_empty label_empty)
  else begin
    let (ts',t) = Seq.pop_back ts in
    Let rv = run_block ts' in
    LetBehavior S = runs_type_stat t in
    if res_type S == restype_throw then
      ReturnBehavior S
    else
      let V = if res_value S == resvalue_empty then rv else res_value S in
      ReturnBehavior (res_intro (res_type S) V (res_label S))
  end
```

EnglishJsRef example output 2

Generation hints: specializing on the empty list, then on a nonempty list, unfolding the "Let" monad on "Let rv".

The production $StatementList$: [the empty list] is evaluated as follows:

1. Return (normal, empty, empty).

The production $StatementList$: $StatementList\ Statement$ is evaluated as follows:

1. Let L be the result of evaluating $StatementList$.
2. If L is an abrupt termination, return L.
3. Let rv denote $L.value$.
4. Let S be the result of evaluating $Statement$.
5. If $S.type$ is exception, then return S.
6. If $S.value$ is empty, then let V be rv, else let V be $S.value$.
7. Return ($S.type$, V, $S.label$).

See `http://www.ecma-international.org/ecma-262/5.1/#sec-12.1`

OcamlJsRef example 3

```
let convert_prim_to_number w =
  match w with
  | prim_undef → JsNumber.nan
  | prim_null → JsNumber.zero
  | prim_bool b → if b then JsNumber.one else JsNumber.zero
  | prim_number n → n
  | prim_string s → JsNumber.from_string s

let to_primitive v preftypeOpt =
  match v with
  | value_prim w → Return w
  | value_object p → Let w = object_default_value p preftypeOpt in Return w

let to_number v =
  match v with
  | value_prim w → Return (convert_prim_to_number w)
  | value_object p → Let w = to_primitive p (Some preftype_number) in
                     Return (convert_prim_to_number w)
```

EnglishJsRef example input 3

```
\section{ToNumber}

The abstract operation ToNumber($v$) converts its argument $v$ to a value of type
     Number according to the table below.

\begin{patchDefinition}
let to_number v =
  match v with
  | value_prim w → ...
  | value_object p → Let w = to_primitive p (Some preftype_number) in
                     Return (to_number w)
\end{patchDefinition}

\generateDispatchOnTypeTable
  [specialization: op:=op_add]
  [inlining: convert_prim_to_number]
  {to_number}
```

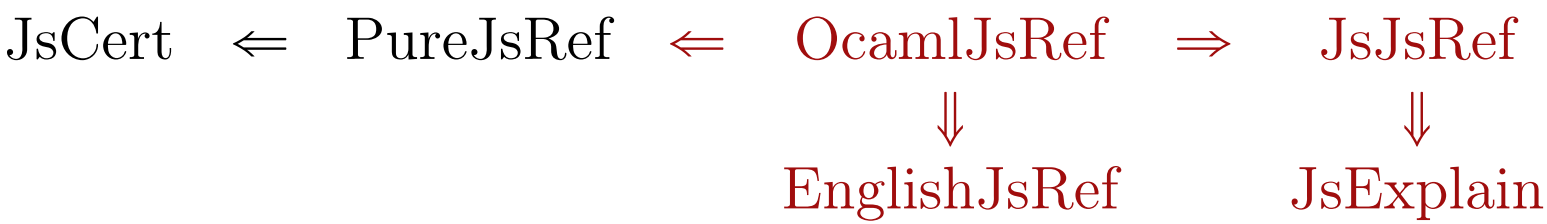
EnglishJsRef example output 3

The abstract operation ToNumber(v) converts its argument v to a value of type Number according to the table below.

Argument Type	**Result**
Undefined	The result is NaN.
Null	The result is $+0$.
Boolean	If v is **true** then the result is 1, else the result is $+0$.
Number	The result is v.
String	The result is NumberOfString(v).
Object	1. Let w be ToPrimitive(v). 2. The result is ToNumber(w).

See `http://www.ecma-international.org/ecma-262/5.1/#sec-9.3`

Overview


JsCert
PureJsRef
OcamlJsRef
JsJsRef
EnglishJsRef
JsExplain

JsJsRef

JsJsRef: same as OCamlJsRef, but implemented using JavaScript syntax.

JsJsRef features:

- ability to execute the JS specification on a JS program in JS,
- code in the core fragment of JS,
- encoding of algebraic data types using JS objects,
- representation of the heap as a JS array.
- explicit monad (unless using a syntax extension for JS),

Idea: JsJsRef could be automatically generated from OCamlJsRef.

JsJsRef example

```
function run_expr_binary_op(op, e1, e2) {
  if (! (is_lazy_op(op))) {
    if_spec(run_expr_get_value(e1), function(v1) {
      if_spec(run_expr_get_value(e2), function(v2) {
        run_binary_op(op,v1,v2) }) } )
  }
  ...
}

function run_binary_op(op, v1, v2) {
  if (op == binary_op_add) {
    if_spec(to_primitive(v1), function(w1) {
      if_spec(to_primitive(v2), function(w2) {
        if (type_of(w1) == type_string || type_of(w2) == type_string) {
          if_spec(to_string(w1), function(s1) {
            if_spec(to_string(w2), function(s2) {
              return_ter(JsString.append(s1,s2)) }) })
        } else {
          if_spec(to_number(w1), function(n1) {
            if_spec(to_number(w2), function(n2) {
              return_ter(JsNumber.add(n1,n2)) }) })
        } )} )}
  } ...
```

JsExplain

JsExplain: instrumented version of JsJsRef, to trace the execution.

JsExplain features:

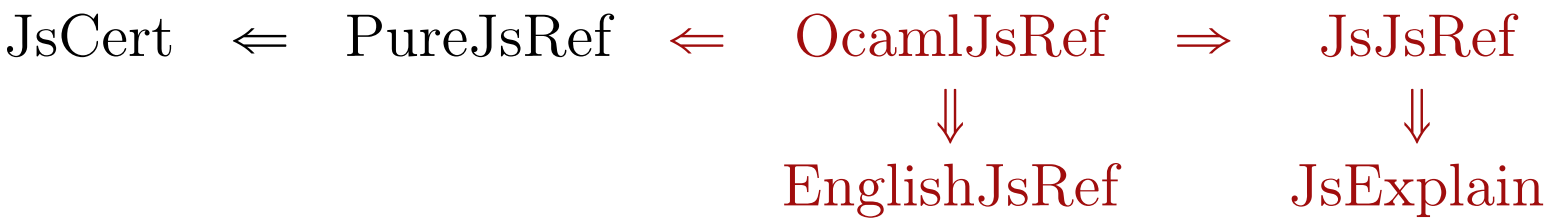
- line-by-line execution of the interpreter,
- full logging of the execution trace,
- interactive investigation of the trace in a browser,
- back-in-time investigation of the trace.

Idea: JsExplain could be automatically generated from OCamlJsRef.

JsExplain example code

```
function run_binary_op(op, v1, v2) {
  var ctx = { v1: v1; v2: v2 };
  log(145, ctx, "run_binary_op");
  if (op == binary_op_add) {
    log(146, ctx);
    if_spec(to_primitive(v1), function(w1) {
      ctx.w1 = w1;
      log(148, ctx);
      if_spec(to_primitive(v2), function(w2) {
        ctx.w2 = w2;
        log(149, ctx);
        if (type_of(w1) == type_string || type_of(w2) == type_string) {
          log(150, ctx);
          ...
        } else {
          ...
        }
  } }
}
```

Overview


JsCert
PureJsRef
OcamlJsRef
JsJsRef
EnglishJsRef
JsExplain