An Interactive Debugger for the JavaScript Specification

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.

2016/04/07
Verified trustworthy software systems, Specialist meeting

Motivation

Equip JavaScript with:

1. A formal, executable specification,
2. Formal proofs of security properties,
3. Logics and tools for verification.

Goal: getting JavaScript committee to adopt a formal semantics.

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

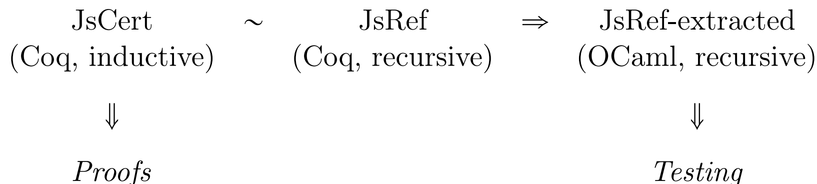
Algorithms steps that say

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



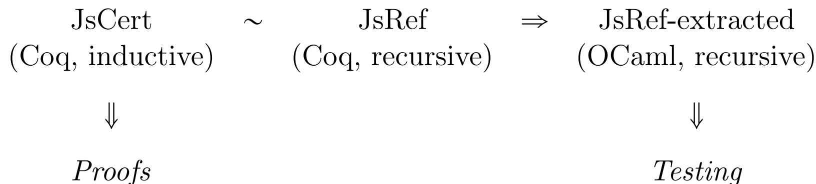
A glance at JsCert (800 rules, 3000 loc)

```
(** 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



We need to:

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

The new plan

		ECMA (English prose)		
		$\Uparrow$		
JsCert (Coq, inductive)	$\Leftarrow$	NewJsRef (OCaml, recursive)	$\Rightarrow$	JsExplain (JS, recursive)
$\Downarrow$		$\Downarrow$		$\Downarrow$
Proofs		*Testing*		*Debugging*

NewJsRef: introduce monadic notation (future work)

```
let rec run_expr_binary_op S C op e1 e2 =
  if not (is_lazy_op op) then
    let%val (S1,v1) = run_expr_get_value r S C e1 in
    let%val (S2,v2) = run_expr_get_value r S1 C e2 in
    run_binary_op S2 C op v1 v2
  else ...

and run_binary_op S C op v1 v2 =
  if op = binary_op_add then
    let%op (S1,(w1,w2)) = apply_twice to_primitive r S C (v1,v2) in
    if type_of w1 = type_string || type_of w2 = type_string then
       let%op (S2,(s1,s2)) = apply_twice to_string r S1 C (w1,w2) in
       return%val S2 (Value_string (JsString.append s1 s2))
    else
       let%op (S2,(n1,n2)) = apply_twice to_number r S1 C (w1,w2) in
       return%val S2 (Value_number (JsNumber.add n1 n2))
  else ...
```

NewJsRef: implicit state and context (future work)

```
let rec run_expr_binary_op op e1 e2 =
  if not (is_lazy_op op) then
    let%val v1 = run_expr_get_value e1 in
    let%val v2 = run_expr_get_value e2 in
    run_binary_op op v1 v2
  else ...

and run_binary_op op v1 v2 =
  if op = binary_op_add then
    let%op (w1,w2) = apply_twice to_primitive (v1,v2) in
    if type_of w1 = type_string || type_of w2 = type_string then
       let%op (s1,s2) = apply_twice to_string (w1,w2) in
       return%val (Value_string (JsString.append s1 s2))
    else
       let%op (n1,n2) = apply_twice to_number (w1,w2) in
       return%val (Value_number (JsNumber.add n1 n2))
  else ...
```

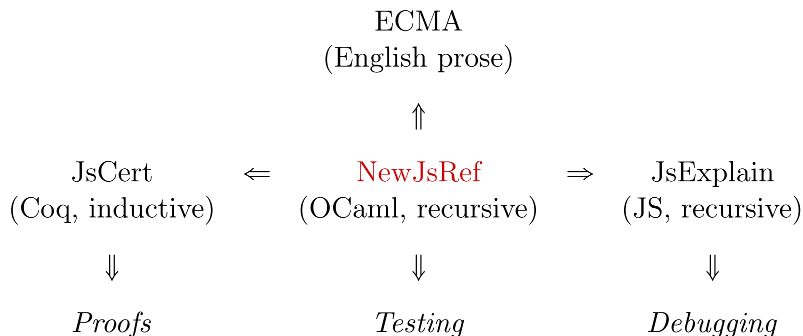
NewJsRef: generate English prose (future work)

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]]".

The new plan



JsExplain

Execute the specification interactively, visualizing at the same time:

- the code of the interpreter (in OCaml, or JavaScript, or English),
- the code of the JavaScript interpreted program.

Intended users: JS committee members, VM developers, test engineers.

Demo.

The new plan

		ECMA (English prose)		
		⇑		
JsCert (Coq, inductive)	⇐	NewJsRef (OCaml, recursive)	⇒	JsExplain (JS, recursive)
⇓		⇓		⇓
Proofs		*Testing*		*Debugging*

From JsRef to JsCert

```
let rec run_expr_binary_op S C op e1 e2 =
  if not (is_lazy_op op) then
    if_spec (run_expr_get_value S C e1) (fun S1 v1 ⇒
      if_spec (run_expr_get_value S1 C e2) (fun S2 v2 ⇒
        run_binary_op S2 C op v1 v2))
  else ...

and run_binary_op S C op v1 v2 =
   If op = binary_op_add then
    if_spec (convert_twice to_primitive 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 S1 C (w1,w2)) (fun S2 (s1,s2) ⇒
          res_ter S2 (JsString.append s1 s2))
        else
          if_spec (convert_twice to_number S1 C (w1,w2)) (fun S2 (n1,n2) ⇒
            res_ter S2 (JsNumber.add n1 n2)))
   else ...
```

```
| 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
```

Study of a similar code, with less noise

```
type 'a out = Res of 'a | Exn of value

let rec run (t:trm) : value out =
  match t with
  | Trm_add (t1,t2) →
      let' v1 = run t1 in
      let' v2 = run t2 in
      let' w1 = to_primitive v1 in
      let' w2 = to_primitive v2 in
      if is_string w1 || is_string w2 then
        let' s1 = to_string w1 in
        let' s2 = to_string w2 in
        Res (Val_string (string_append s1 s2))
      else
        let' n1 = to_number w1 in
        let' n2 = to_number w2 in
        Res (Val_number (float_add n1 n2))
  | ...

and to_primitive (v:value) : value out = ...
and to_number (v:value) : number out = ...
and to_string (v:value) : string out = ...
```

Big-step rules

Big-step evaluation judgment without exceptions: $t_{/m} \Downarrow v_{/m'}$

$$\frac{\begin{array}{c} t_{1/m_1} \Downarrow v_{1/m_2} \\ t_{2/m_2} \Downarrow v_{2/m_3} \\ (\mathsf{to_primitive}\, v_1)_{/m_3} \Downarrow w_{1/m_4} \\ (\mathsf{to_primitive}\, v_2)_{/m_4} \Downarrow w_{2/m_5} \\ (\mathsf{is_string}\, w_1 \vee \mathsf{is_string}\, w_2) \\ (\mathsf{to_string}\, w_1)_{/m_5} \Downarrow s_{1/m_6} \\ (\mathsf{to_string}\, w_2)_{/m_6} \Downarrow s_{2/m_7} \end{array}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \Downarrow (\mathsf{Val_string}\, (\mathsf{string_append}\, s_1\, s_2))_{/m_7}}$$

$$\frac{\begin{array}{c} t_{1/m_1} \Downarrow v_{1/m_2} \\ t_{2/m_2} \Downarrow v_{2/m_3} \\ (\mathsf{to_primitive}\, v_1)_{/m_3} \Downarrow w_{1/m_4} \\ (\mathsf{to_primitive}\, v_2)_{/m_4} \Downarrow w_{2/m_5} \\ \neg(\mathsf{is_string}\, w_1 \vee \mathsf{is_string}\, w_2) \\ (\mathsf{to_number}\, w_1)_{/m_5} \Downarrow n_{1/m_6} \\ (\mathsf{to_number}\, w_2)_{/m_6} \Downarrow n_{2/m_7} \end{array}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \Downarrow (\mathsf{Val_number}\, (\mathsf{float_add}\, n_1\, n_2))_{/m_7}}$$

Big-step rules for exceptions

Extension of evaluation judgment to extensions: $t_{/m} \Downarrow o_{/m'}$.

$$o \;:=\; \mathsf{Res}\, v \;\mid\; \mathsf{Exn}\, v \qquad\qquad \text{(Res is left implicit)}$$

$$\frac{t_{1/m_1} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_2}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_2}}$$

$$\frac{t_{1/m_1} \;\Downarrow\; v_{1/m_2} \qquad t_{2/m_2} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_3}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_3}}$$

$$\frac{t_{1/m_1} \;\Downarrow\; v_{1/m_2} \qquad t_{2/m_2} \;\Downarrow\; v_{2/m_3} \qquad (\mathsf{to_primitive}\, v_1)_{/m_3} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_4}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_4}}$$

$$\frac{t_{1/m_1} \;\Downarrow\; v_{1/m_2} \qquad t_{2/m_2} \;\Downarrow\; v_{2/m_3} \qquad (\mathsf{to_primitive}\, v_1)_{/m_3} \;\Downarrow\; w_{1/m_4} \qquad (\mathsf{to_primitive}\, v_2)_{/m_4} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_5}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \;\Downarrow\; (\mathsf{Exn}\, v)_{/m_5}}$$

...

Big-step rules for exceptions, continued

$$\frac{\begin{array}{c} {t_1}_{/m_1} \Downarrow {v_1}_{/m_2} \\ {t_2}_{/m_2} \Downarrow {v_2}_{/m_3} \\ (\mathsf{to_primitive}\, v_1)_{/m_3} \Downarrow {w_1}_{/m_4} \\ (\mathsf{to_primitive}\, v_2)_{/m_4} \Downarrow {w_2}_{/m_5} \\ (\mathsf{is_string}\, w_1 \lor \mathsf{is_string}\, w_2) \\ (\mathsf{to_string}\, w_1)_{/m_5} \Downarrow (\mathsf{Exn}\, v)_{/m_6} \end{array}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \Downarrow (\mathsf{Exn}\, v)_{/m_6}}$$

$$\frac{\begin{array}{c} {t_1}_{/m_1} \Downarrow {v_1}_{/m_2} \\ {t_2}_{/m_2} \Downarrow {v_2}_{/m_3} \\ (\mathsf{to_primitive}\, v_1)_{/m_3} \Downarrow {w_1}_{/m_4} \\ (\mathsf{to_primitive}\, v_2)_{/m_4} \Downarrow {w_2}_{/m_5} \\ (\mathsf{is_string}\, w_1 \lor \mathsf{is_string}\, w_2) \\ (\mathsf{to_string}\, w_1)_{/m_5} \Downarrow {s_1}_{/m_6} \\ (\mathsf{to_string}\, w_2)_{/m_6} \Downarrow (\mathsf{Exn}\, v)_{/m_7} \end{array}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \Downarrow (\mathsf{Exn}\, v)_{/m_7}}$$

$$\frac{\begin{array}{c} {t_1}_{/m_1} \Downarrow {v_1}_{/m_2} \\ {t_2}_{/m_2} \Downarrow {v_2}_{/m_3} \\ (\mathsf{to_primitive}\, v_1)_{/m_3} \Downarrow {w_1}_{/m_4} \\ (\mathsf{to_primitive}\, v_2)_{/m_4} \Downarrow {w_2}_{/m_5} \\ \neg(\mathsf{is_string}\, w_1 \lor \mathsf{is_string}\, w_2) \\ (\mathsf{to_number}\, w_1)_{/m_5} \Downarrow (\mathsf{Exn}\, v)_{/m_6} \end{array}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \Downarrow (\mathsf{Exn}\, v)_{/m_6}}$$

$$\frac{\begin{array}{c} {t_1}_{/m_1} \Downarrow {v_1}_{/m_2} \\ {t_2}_{/m_2} \Downarrow {v_2}_{/m_3} \\ (\mathsf{to_primitive}\, v_1)_{/m_3} \Downarrow {w_1}_{/m_4} \\ (\mathsf{to_primitive}\, v_2)_{/m_4} \Downarrow {w_2}_{/m_5} \\ \neg(\mathsf{is_string}\, w_1 \lor \mathsf{is_string}\, w_2) \\ (\mathsf{to_number}\, w_1)_{/m_5} \Downarrow {n_1}_{/m_6} \\ (\mathsf{to_number}\, w_2)_{/m_6} \Downarrow (\mathsf{Exn}\, v)_{/m_7} \end{array}}{(\mathsf{Trm_add}\, t_1\, t_2)_{/m_1} \Downarrow (\mathsf{Exn}\, v)_{/m_7}}$$

Pretty-big-step semantics (ESOP'13)

Pretty-big-step evaluation judgment: $e_{/m} \Downarrow o_{/m'}$

$$
\begin{array}{rl}
e := & t \\
& \mid \textsf{Trm_add_1 o t} \mid \textsf{Trm_add_2 v o} \mid \textsf{Trm_add_3 o v} \mid \textsf{Trm_add_4 v o} \\
& \mid \textsf{Trm_add_5 o v} \mid \textsf{Trm_add_6 s o} \mid \textsf{Trm_add_5' n o} \mid \textsf{Trm_add_6' o v}
\end{array}
$$

$$
\frac{{t_1}_{/m_1} \Downarrow {o'}_{/m_2} \qquad (\textsf{Trm_add_1}\, o'\, t_2)_{/m_2} \Downarrow o_{/m_3}}{(\textsf{Trm_add}\, t_1\, t_2)_{/m} \Downarrow o_{/m_3}}
$$

$$
\frac{{t_2}_{/m_1} \Downarrow {o'}_{/m_2} \qquad (\textsf{Trm_add_2}\, v_1\, o')_{/m_2} \Downarrow o_{/m_3}}{(\textsf{Trm_add_1}\, (\textsf{Res}\, v_1)\, t_2)_{/m_1} \Downarrow o_{/m_3}}
$$

$$
\frac{}{(\textsf{Trm_add_1}\, (\textsf{Exn}\, v)\, t_2)_{/m} \Downarrow (\textsf{Exn}\, v)_{/m}} \qquad \text{subsumed by:} \qquad \frac{\textsf{outcomeof}\, e = \textsf{Some}\, (\textsf{Exn}\, v) \quad \neg\, \textsf{is-exception-handler}\, e}{e_{/m} \Downarrow (\textsf{Exn}\, v)_{/m}}
$$

Pretty-big-step rules

$$\frac{(\mathsf{to_primitive}\, v_1)_{/m_1} \Downarrow {o'}_{/m_2} \qquad (\mathsf{Trm_add_3}\, o'\, v_2)_{/m_2} \Downarrow o_{/m_3}}{(\mathsf{Trm_add_2}\, v_1\, v_2)_{/m_1} \Downarrow o_{/m_3}}$$

$$\frac{(\mathsf{to_primitive}\, v_2)_{/m_2} \Downarrow {o'}_{/m_2} \qquad (\mathsf{Trm_add_4}\, w_1\, o')_{/m_2} \Downarrow o_{/m_3}}{(\mathsf{Trm_add_3}\, w_1\, v_2)_{/m_1} \Downarrow o_{/m_3}}$$

$$\frac{\begin{array}{c}(\mathsf{is_string}\, w_1 \ \vee\ \mathsf{is_string}\, w_2) \\ (\mathsf{to_string}\, w_1)_{/m_1} \Downarrow {o'}_{/m_2} \\ (\mathsf{Trm_add_5}\, o'\, w_2)_{/m_2} \Downarrow o_{/m_3}\end{array}}{(\mathsf{Trm_add_4}\, w_1\, w_2)_{/m_1} \Downarrow o_{/m_3}}$$

$$\frac{\begin{array}{c}\neg\,(\mathsf{is_string}\, w_1 \ \vee\ \mathsf{is_string}\, w_2) \\ (\mathsf{to_number}\, w_1)_{/m_1} \Downarrow {o'}_{/m_2} \\ (\mathsf{Trm_add_5'}\, o'\, w_2)_{/m_2} \Downarrow o_{/m_3}\end{array}}{(\mathsf{Trm_add_4}\, w_1\, w_2)_{/m_1} \Downarrow o_{/m_3}}$$

$$\frac{\begin{array}{c}(\mathsf{to_string}\, w_2)_{/m_1} \Downarrow {o'}_{/m_2} \\ (\mathsf{Trm_add_6}\, s_1\, o')_{/m_2} \Downarrow o_{/m_3}\end{array}}{(\mathsf{Trm_add_5}\, s_1\, w_2)_{/m_1} \Downarrow o_{/m_3}}$$

$$\frac{\begin{array}{c}(\mathsf{to_number}\, w_2)_{/m_1} \Downarrow {o'}_{/m_2} \\ (\mathsf{Trm_add_6'}\, n_1\, o')_{/m_2} \Downarrow o_{/m_3}\end{array}}{(\mathsf{Trm_add_5}\, n_1\, w_2)_{/m_1} \Downarrow o_{/m_3}}$$

$$\frac{v = \mathsf{Val_string}\,(\mathsf{string_append}\, s_1\, s_2)}{(\mathsf{Trm_add\, 6}\, s_1\, s_2)_{/m} \Downarrow v_{/m}}$$

$$\frac{v = \mathsf{Val_number}\,(\mathsf{float_add}\, n_1\, n_2)}{(\mathsf{Trm_add\, 6'}\, n_1\, n_2)_{/m} \Downarrow v_{/m}}$$

Size and generation of pretty-big-step rules

- Total size of the rules improved from quadratic to linear.
- Intermediate forms correspond to the states of the interpreter.
- We should be able to generate pretty-big-step rules automatically.

Summary