A Coq Formalisation of a Core of R

Martin Bodin

CMM

13^{th} of January

- More than 10,000 packages;
- More than 2 million users worldwide;
- Used by 70% of data miners (24% as primary language).

R: A Programming Language About Vectors

```r
v <- c(10, 12, 14, 11, 13)
v[1]                              # Returns 10
```

R: A Programming Language About Vectors

```r
v <- c(10, 12, 14, 11, 13)
v[1]                          # Returns 10
indices <- c(3, 5, 1)
v[indices]                    # Returns c(14, 13, 10)
```

R: A Programming Language About Vectors

```r
v <- c(10, 12, 14, 11, 13)
v[1]                          # Returns 10
indices <- c(3, 5, 1)
v[indices]                    # Returns c(14, 13, 10)
v[-2]                         # Returns c(10, 14, 11, 13)
```

R: A Programming Language About Vectors

```r
v <- c(10, 12, 14, 11, 13)
v[1]                          # Returns 10
indices <- c(3, 5, 1)
v[indices]                    # Returns c(14, 13, 10)
v[-2]                         # Returns c(10, 14, 11, 13)
v[-indices]                   # Returns c(12, 11)
```

R: A Programming Language About Vectors

```r
v <- c(10, 12, 14, 11, 13)
v[1]                          # Returns 10
indices <- c(3, 5, 1)
v[indices]                    # Returns c(14, 13, 10)
v[-2]                         # Returns c(10, 14, 11, 13)
v[-indices]                   # Returns c(12, 11)
v[c(FALSE, TRUE, FALSE)]      # Returns c(12, 13)
```

R: A Programming Language About Vectors

```r
v <- c(10, 12, 14, 11, 13)
v[1]                          # Returns 10
indices <- c(3, 5, 1)
v[indices]                    # Returns c(14, 13, 10)
v[-2]                         # Returns c(10, 14, 11, 13)
v[-indices]                   # Returns c(12, 11)
v[c(FALSE, TRUE, FALSE)]      # Returns c(12, 13)
f <- function (i, offset)
       v[i + offset]          # ??
```

R: A Lazy Programming Language

```r
f <- function (x, y = x) {
    x <- 1
    y
    x <- 2
    y
}
f (3)
```

R: A Lazy Programming Language

```r
f <- function (x, y = x) {
    x <- 1
    y
    x <- 2
    y
}
f (3)                                   # Returns 1
```

R: A Lazy Programming Language

```r
f <- function (x, y = x) {
    x <- 1
    y
    x <- 2
    y
}
f (3)                                   # Returns 1
```

```r
f <- function (x, y) if (x == 1) y
f (1, a <- 1)
a                                       # Returns 1
f (0, b <- 1)
b                                       # Raises an error
```

R: A Dynamic Programming Language

```r
f <- function (x, y) missing (y)
f (1, 2)                                  # Returns FALSE
f (1)                                     # Returns TRUE
f ()                                      # Returns TRUE
```

R: A Dynamic Programming Language

```r
f <- function (x, y) missing (y)
f (1, 2)                                # Returns FALSE
f (1)                                   # Returns TRUE
f ()                                    # Returns TRUE
```

```r
f <- function (expr, s) {
    x <- 2
    y <- 3
    eval (substitute (expr))            # Evaluates “expr” in the
                                        # current environment
  }
f (x + y)                               # Returns 5
x + y                                   # Raises an error
```

R: A Dynamic Programming Language

```r
f <- function (x, y) missing (y)
f (1, 2)                                # Returns FALSE
f (1)                                   # Returns TRUE
f ()                                    # Returns TRUE
```

```r
f <- function (expr, s) {
    x <- 2
    y <- 3
    eval (substitute (expr))            # Evaluates “expr” in the
                                        # current environment
  }
f (x + y)                               # Returns 5
x + y                                   # Raises an error
```

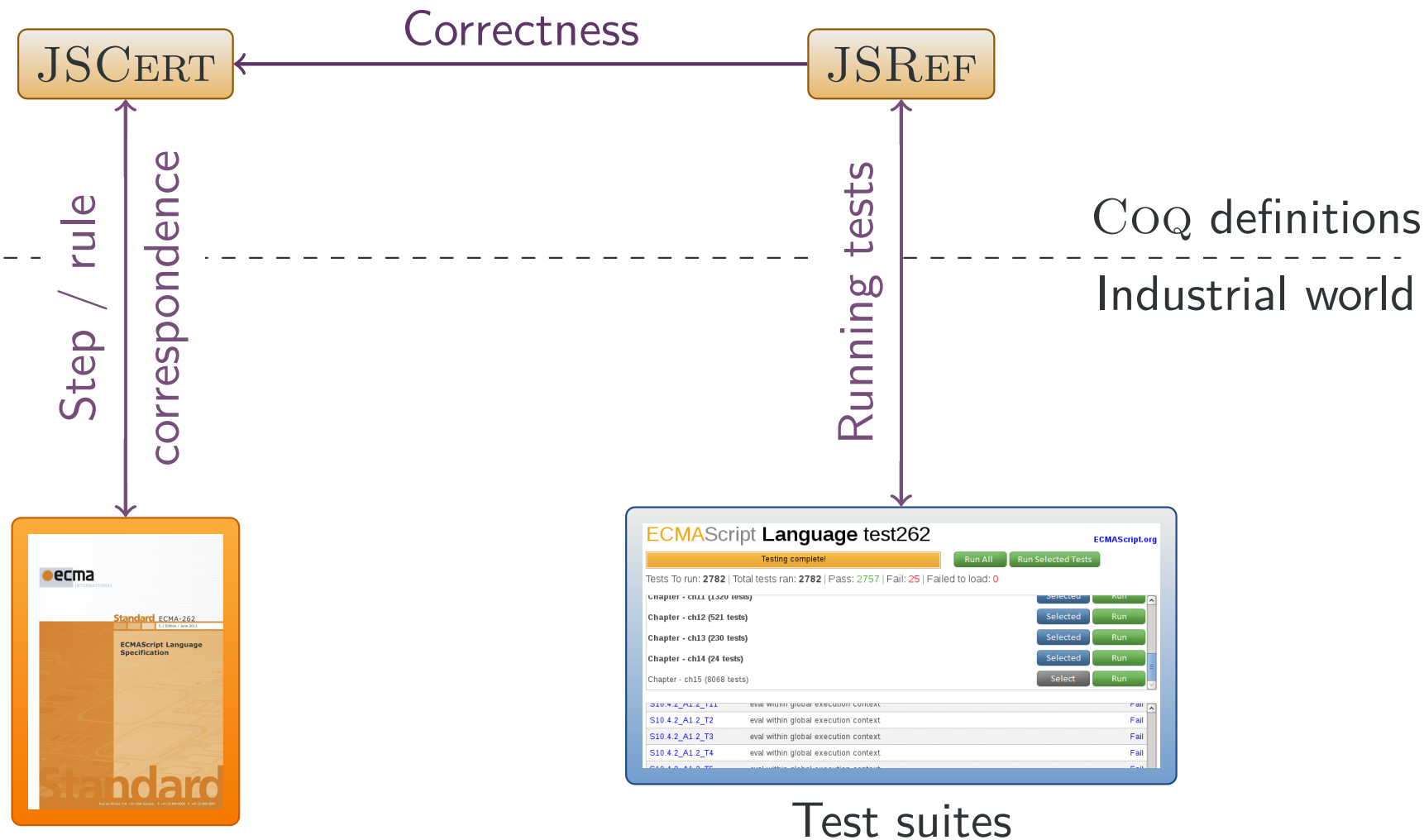
```r
"(" <- function (x) 2 * x
((9))                                   # Returns 36
```

Formalising R

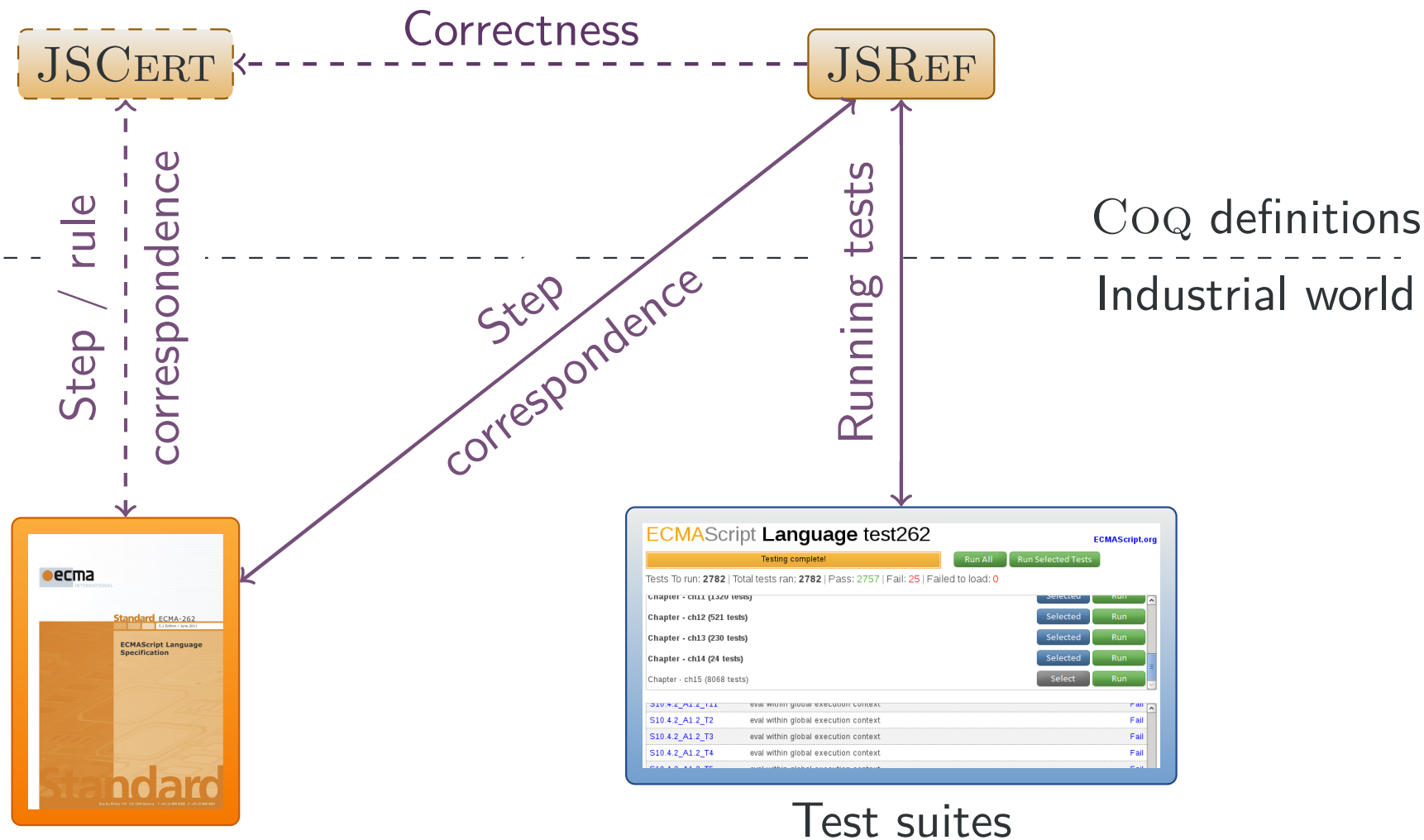
R Trust Sources

- Specification ✗
- Reference interpreter ✓
  - GNU R.
- Test suites ✓
  - TestR, Genthat, etc.

JSCert Overview



JSCert Overview



A Stratified Approach

Rule-based semantics — As usable as possible

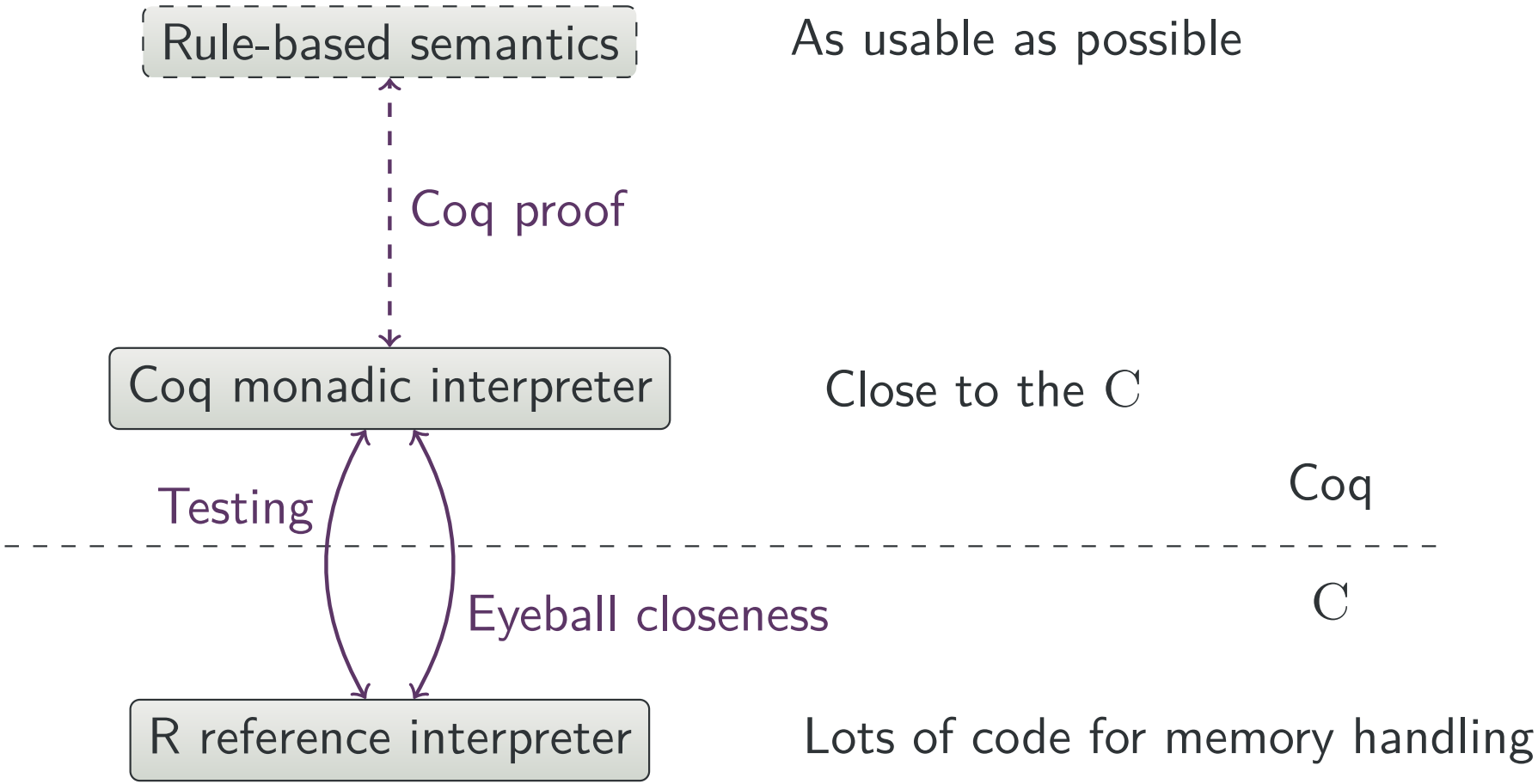
Coq monadic interpreter — Close to the C

Coq

C

R reference interpreter — Lots of code for memory handling

A Stratified Approach



Eyeball Closeness

- C is imperative, pointer-based;
- Coq is purely functional, value-based;
- The translation is based on a monad $\mathrm{state} + \mathrm{error}$.

Eyeball Closeness: Enumeration

C code

```c
typedef enum {
    NILSXP  = 0,
    SYMSXP  = 1,
    LISTSXP = 2,
    CLOSXP  = 3,
    ENVSXP  = 4,
    PROMSXP = 5,
    /* ... */
} SEXPTYPE;
```

Coq code

```
Inductive SExpType :=
  | NilSxp
  | SymSxp
  | ListSxp
  | CloSxp
  | EnvSxp
  | PromSxp
  (* ... *)
  .
```

Eyeball Closeness: Records

C code

```c
struct sxpinfo_struct {
  SEXPTYPE type      :  5;
  unsigned int obj   :  1;
  unsigned int named :  2;
  unsigned int gp    : 16;
  unsigned int mark  :  1;
  unsigned int debug :  1;
  unsigned int trace :  1;
  unsigned int spare :  1;
  unsigned int gcgen :  1;
  unsigned int gccls :  3;
};
/* Total: 32 bits */
```

Coq code

```
Inductive named_field :=
  | named_temporary
  | named_unique
  | named_plural
  .

Record SxpInfo :=
  make_SxpInfo {
    type : SExpType ;
    obj : bool ;
    named : named_field ;
    gp : nbits 16
  }.
```

Eyeball Closeness: Unions

```c
union {
    struct primsxp_struct primsxp;
    struct symsxp_struct symsxp;
    struct listsxp_struct listsxp;
    /* ... */
};
```

C code

- Accesses are unsafe.

```
Inductive SExpRec_union :=
  | primSxp : PrimSxp_struct -> SExpRec_union
  | symSxp : SymSxp_struct -> SExpRec_union
  | listSxp : ListSxp_struct -> SExpRec_union
  (* ... *)
  .
```

Coq code

- Accesses must be guarded.

Eyeball Closeness: Reading Pointers

C code

```
1 symsxp_struct p_sym = p->symsxp;
2 /* ... */
```

Coq code

```
1 read%sym p_sym := p using S in
2 (* ... *)
```

```
1 Inductive result (T : Type) :=
2   | result_success : state -> T -> result T
3   | result_error : result T.
```

```
1 Notation "'read%sym' p_sym ':=' p 'using' S 'in' cont" :=
2   (match read S p with
3    | Some p_ =>
4      match p_ with
5      | symSxp p_sym => cont
6      | _ => result_error
7      end
8    | None => result_error
9    end).
```

Eyeball Closeness: C Code

```
1  EXP* applyClosure (EXP* op, EXP* arglist, EXP* rho){
2
3    EXP* formals, actuals, savedrho, newrho, res;
4
5    if (rho->type != ENVSXP)
6      error ("'rho' must be an environment.");
7
8    formals = op->clo.formals;
9    savedrho = op->clo.env;
10
11   PROTECT (actuals = matchArgs (formals, arglist));
12
13   /* ... */
14
15   return res;
16 }
```

Eyeball Closeness: Coq Code

```
1  Definition applyClosure (S : state) (op arglist rho : EXP_pointer)
2      : result EXP_pointer :=
3
4    read%defined rho_ := rho using S in
5    ifb type rho_ <> EnvSxp then
6      result_error S "'rho' must be an environment."
7    else
8      read%clo op_clo := op using S in
9      let formals := clo_formals op_clo in
10     let savedrho := clo_env op_clo in
11
12     let%success actuals := matchArgs S formals arglist using S in
13
14     (* ... *)
15
16     result_success S res.
```

R Features

R Core

```
1  FUNTAB R_FunTab[] = {
2    {"if",         do_if,         2},
3    {"while",      do_while,      2},
4    {"break",      do_break,      0},
5    {"return",     do_return,     1},
6    {"function",   do_function,   -1},
7    {"<-",         do_set,        2},
8    {"(",          do_paren,      1},
9    /* ... */
10   {"+",          do_arith1,     2},
11   {"-",          do_arith2,     2},
12   {"*",          do_arith3,     2},
13   {"/",          do_arith4,     2},
14   /* ... */
15   {"cos",        do_math20,     1},
16   {"sin",        do_math21,     1},
17   {"tan",        do_math22,     1},
18   /* ... */ }
```

R Core

```
1  FUNTAB R_FunTab[] = {
2    {"if",          do_if,          2},
```

The core is what is needed to call these functions.

- The core is small;
- The formalisation is easily extendable.

Content of the core

- Expression evaluation;
- Function calls;
- Environments, delayed evaluation (promises);
- Initialisation of the global state.

```
17   {"tan",         do_math22,      1},
18   /* ... */ }
```

Future

The current formalisation is modular

- It is easy to add features.
- We can implement specific features and certify their implementations.

Future

The current formalisation is modular

- It is easy to add features.
- We can implement specific features and certify their implementations.

Providing trust

- Test the formalisation...
- ...or certify it (CompCert's semantics, Formalin, etc.).

Building proofs

- Building a rule-based formalisation;
- A more functional interpreter.

} What is the best to build large proofs of programs?

Proof that 1 + 1 reduces to 2 in JSCert

```
1  Lemma one_plus_one_exec : forall S C,
2    red_expr S C one_plus_one (out_ter S (prim_number two)).
3  Proof.
4    intros. unfold one_plus_one.
5    eapply red_expr_binary_op.
6     constructor.
7     eapply red_spec_expr_get_value.
8      eapply red_expr_literal. reflexivity.
9     eapply red_spec_expr_get_value_1.
10    eapply red_spec_ref_get_value_value.
11   eapply red_expr_binary_op_1.
12    eapply red_spec_expr_get_value.
13     eapply red_expr_literal. reflexivity.
14    eapply red_spec_expr_get_value_1.
15    eapply red_spec_ref_get_value_value.
16   eapply red_expr_binary_op_2.
17   eapply red_expr_binary_op_add.
18    eapply red_spec_convert_twice.
19     eapply red_spec_to_primitive_pref_prim.
20    eapply red_spec_convert_twice_1.
21     eapply red_spec_to_primitive_pref_prim.
22    eapply red_spec_convert_twice_2.
23   eapply red_expr_binary_op_add_1_number.
24    simpl. intros [A|A]; inversion A.
25    eapply red_spec_convert_twice.
26     eapply red_spec_to_number_prim. reflexivity.
27    eapply red_spec_convert_twice_1.
28     eapply red_spec_to_number_prim. reflexivity.
29    eapply red_spec_convert_twice_2.
30   eapply red_expr_puremath_op_1. reflexivity.
31 Qed.
```

RExplain

Imperative interpreter

```
let%success res = f args in
read%clo res_clo = res in
```

Functionnal interpreter

```
let%success res = f S args using S in
read%clo res_clo = res using S in
```

ECMA-style specification

1. Let res be the result of calling f with argument args;
2. At this stage, res should be a closure.

Rule-based semantics

```
| run_1 : forall S args o1 o2,
  run S (f args) o1 -> run S (term_1 o1) o2 -> run S (term o1) o2
| run_2 : forall S res_clo o,
  is_closure S res res_clo -> run S (term_2 res_clo) o -> run S (term_1 (out S res)) o
```

https://github.com/jscert/jsexplain

Thank you for listening!

The current formalisation is modular

- It is easy to add features.
- We can implement specific features and certify their implementations.

Providing trust

- Test the formalisation...
- ...or certify it (CompCert's semantics, Formalin, etc.).

Building proofs

- Building a rule-based formalisation;
- A more functional interpreter.

} What is the best to build large proofs of programs?

`https://github.com/Mbodin/proveR`

1. R
2. R Trust Sources
3. Eyeball Closeness
4. R Features

Bonuses

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

The JSCert Project

~ 900 rules

JSCert

Correctness

~ 6,000 lines of proof

~ 3,000 lines of code

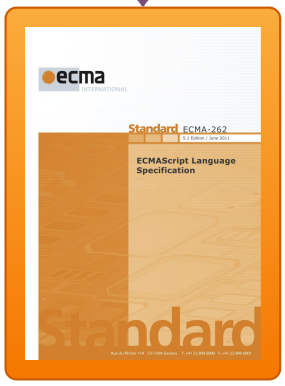
JSRef

Step / rule correspondence

Running tests

Coq definitions

Industrial world



~ 900 steps

5,126 tests passed

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

How to Represent Imperative Features in a Functional Setting

- Structures like maps are easy to implement;
- We can represent every element of the state of a program (memory, outputs, etc.) in a data-structure;
- We have to pass this structure along the program.

Enter the monad

```
1 if_success (run s1 p) (fun s2 =>
2   let s3 = write s2 x v in
3   if_success (run s3 p') (fun s4 =>
4     return_success s4))
```

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

Formalisation of Semantics in Coq

```
1  Inductive semantics : state -> prog -> state -> Prop ->
2
3    | semantics_skip : forall s p, semantics s p s
4
5    | semantics_seq : forall s1 s2 s3 p1 p2,
6      semantics s1 p1 s2 ->
7      semantics s2 p2 s3 ->
8      semantics s1 (seq p1 p2) s3
9
10   | semantics_asgn : forall s x v,
11     semantics s (asgn x v) (write s x v)
12   .
```

Sequence in JSCert (Paper Version)

"`s1 ; s2`" is evaluated as follows.

1. Let o_1 be the result of evaluating s1.
2. If o_1 is an exception, return o_1.
3. Let o_2 be the result of evaluating s2.
4. If an exception V was thrown, return $(Throw, V, empty)$.
5. If $o_2.value$ is empty, let $V = o_1.value$, otherwise let $V = o_2.value$.
6. Return $(o_2.type, V, o_2.target)$.

Sequence in JSCert (Paper Version)

"`s1 ; s2`" is evaluated as follows.

1. Let o_1 be the result of evaluating `s1`.
2. If o_1 is an exception, return o_1.
3. Let o_2 be the result of evaluating `s2`.

Sequence in JSCert (Paper Version)

"s1 ; s2" is evaluated as follows.

1. Let o_1 be the result of evaluating s1.
2. If o_1 is an exception, return o_1.
3. Let o_2 be the result of evaluating s2.

$$\frac{S, C, s_1 \Downarrow o_1 \qquad o_1, seq_1\ s_2 \Downarrow o}{S, C, seq\ s_1\ s_2 \Downarrow o}\ \text{SEQ-1}(s_1, s_2) \qquad \frac{}{o_1, seq_1\ s_2 \Downarrow o_1}\ \text{SEQ-2}(s_2) \quad \textbf{abort}\ o_1$$

$$\frac{o_1, s_2 \Downarrow o_2 \qquad o_1, o_2, seq_2 \Downarrow o}{o_1, seq_1\ s_2 \Downarrow o}\ \text{SEQ-3}(s_2) \qquad \neg\textbf{abort}\ o_1 \qquad \ldots$$

Sequence in JSCert

```
1  Inductive red_stat : state → scope → stat → out → Prop :=
2
3  | red_stat_seq_1 : forall S C s1 s2 o1 o,
4    red_stat S C s1 o1 →
5    red_stat S C (seq_1 s2 o1) o →
6    red_stat S C (seq s1 s2) o
7
8  | red_stat_seq_2 : forall S C s2 o1,
9    abort o1 →
10   red_stat S C (seq_1 s2 o1) o1
11
12 | red_stat_seq_3 : forall S0 S C s2 o2 o,
13   red_stat S C s2 o2 →
14   red_stat S C (seq_2 o2) o →
15   red_stat S0 C (seq_1 s2 (out_ter S)) o
16
17 (* ... *).
```

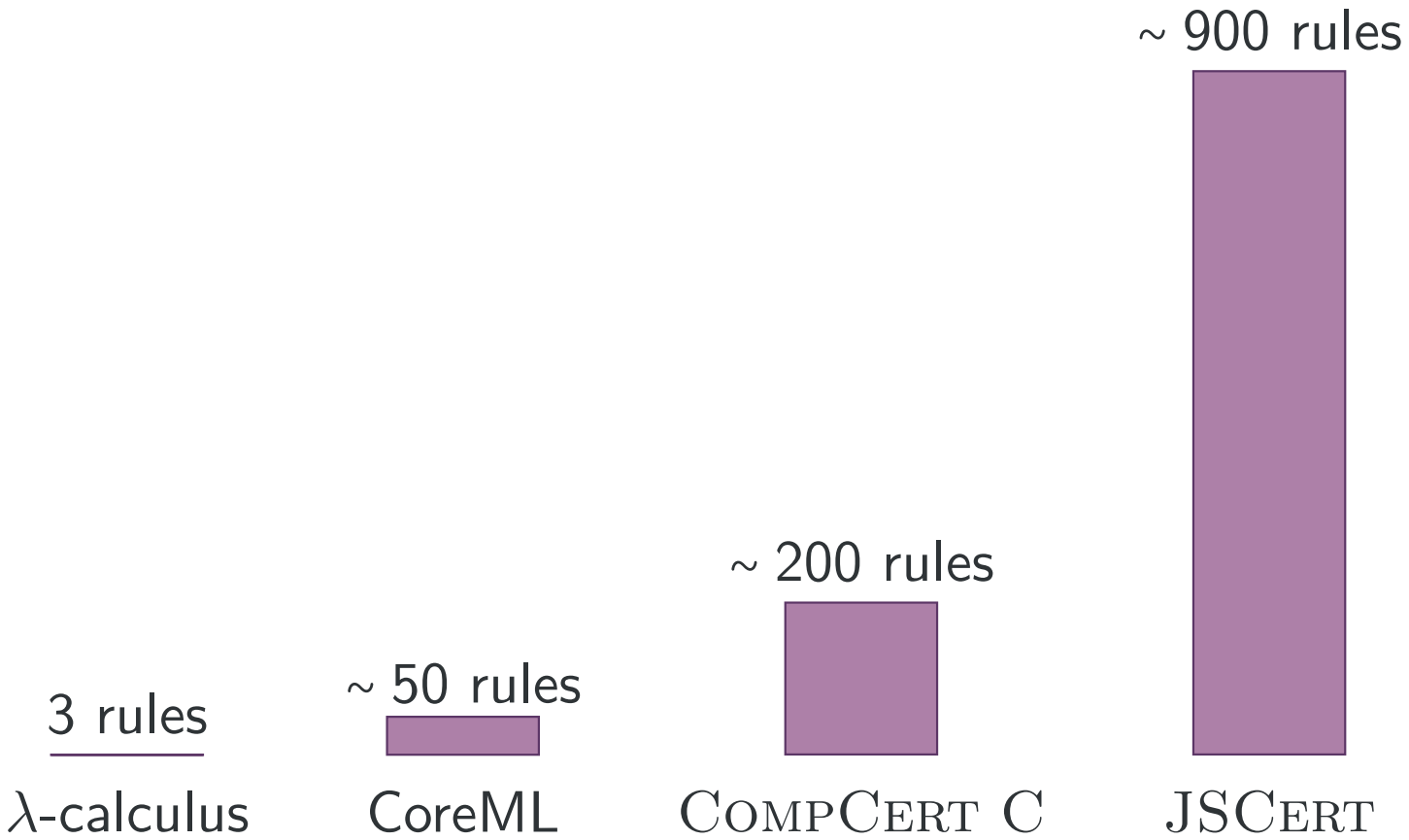
$$\frac{S, C, s_1 \Downarrow o_1 \qquad o_1, seq_1\ s_2 \Downarrow o}{S, C, seq\ s_1\ s_2 \Downarrow o}\ \text{SEQ-1}(s_1, s_2)$$

$$\frac{}{o_1, seq_1\ s_2 \Downarrow o_1}\ \text{SEQ-2}(s_2)\quad \textbf{abort}\ o_1$$

$$\frac{o_1, s_2 \Downarrow o_2 \qquad o_1, o_2, seq_2 \Downarrow o}{o_1, seq_1\ s_2 \Downarrow o}\ \text{SEQ-3}(s_2)\quad \neg\textbf{abort}\ o_1$$

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

Semantic Sizes



1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

Other Subtleties

```r
f <- function (x, y, option, longArgumentName) ...

# All the following calls are equivalent.
f (1, 2, ”something”, 42)
f (option = ”something”, 1, 2, 42)
f (opt = ”something”, long = 42, 1, 2)
```

Other Subtleties

```
f <- function (x, y, option, longArgumentName) ...

# All the following calls are equivalent.
f (1, 2, "something", 42)
f (option = "something", 1, 2, 42)
f (opt = "something", long = 42, 1, 2)
```

```
f <- function (abc, ab, de) c (abc, ab, de)

# All the following calls are equivalent.
f (1, 2, 3)
f (de = 3, 1, 2)
f (d = 3, 1, 2)
f (ab = 2, 1, 2)
f (ab = 2, a = 1, 3)

f (a = 3, 1, 2) # Returns an error.
```

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

Eyeball Closeness: Reading Pointers

C code

```c
symsxp_struct p_sym = p->symsxp;
/* ... */
```

- May fail because the pointer `p` is unbound;
- May fail because the union `*p` is not a `symsxp`.

Eyeball Closeness: Reading Pointers

C code

```c
symsxp_struct p_sym = p->symsxp;
/* ... */
```

Coq code, first try

```
match read p with
  (* ... *)
end
```

- May fail because the pointer `p` is unbound;
- May fail because the union `*p` is not a `symsxp`.

Eyeball Closeness: Reading Pointers

C code

```c
symsxp_struct p_sym = p->symsxp;
/* ... */
```

Coq code, second try

```
match read S p with
| Some p_ =>
  match p_ with
  | symSxp p_sym =>
    (* ... *)
  | _ => (* ??? *)
  end
| None => (* ??? *)
end
```

- May fail because the pointer `p` is unbound;
- May fail because the union `*p` is not a `symsxp`.

Eyeball Closeness: Reading Pointers

C code

```
1 symsxp_struct p_sym = p->symsxp;
2 /* ... */
```

- May fail because the pointer p is unbound;
- May fail because the union *p is not a symsxp.

Coq code, third try

```
1 match read S p with
2 | Some p_ =>
3   match p_ with
4   | symSxp p_sym =>
5     (* ... *)
6   | _ => error
7   end
8 | None => error
9 end
```

```
1 Inductive result (T : Type) :=
2   | success : state -> T -> result T
3   | error : result T
4   .
```

Eyeball Closeness: Reading Pointers

C code

```
1 symsxp_struct p_sym = p->symsxp;
2 /* ... */
```

- May fail because the pointer p is unbound;
- May fail because the union *p is not a symsxp.

Coq code, fourth try

```
1 read%sym p_sym := p using S in
2 (* ... *)
```

```
1 Inductive result (T : Type) :=
2   | success : state -> T -> result T
3   | error : result T
4   .
```

```
1 Notation "'read%sym' p_sym ':=' p
2     'using' S 'in' cont" :=
3   (* ... *).
```

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

```
expr:
  | NUM_CONST                          { $$ = $1;  setId( $$, @$); }
  | STR_CONST                          { $$ = $1;  setId( $$, @$); }
  | NULL_CONST                         { $$ = $1;  setId( $$, @$); }
  | SYMBOL                             { $$ = $1;  setId( $$, @$); }
  | LBRACE exprlist RBRACE
    { $$ = xxexprlist($1,&@1,$2); setId( $$, @$); }
  | LPAR expr_or_assign RPAR
    { $$ = xxparen($1,$2);  setId( $$, @$); }
```

```
expr:
  | c = NUM_CONST                              { c }
  | c = STR_CONST                              { c }
  | c = NULL_CONST                             { c }
  | c = SYMBOL                                 { c }
  | b = LBRACE; e = exprlist; RBRACE
    { eatLines := false ; lift2 (only_state xxexprlist) b e }
  | p = LPAR; e = expr_or_assign; RPAR
    { lift2 (no_runs xxparen) p e }
```

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

The Full State+Error Monad

```
1  Inductive result (A : Type) :=
2    | result_success : state -> A -> result A
3    | result_error : state -> string -> result A
4    | result_longjump : state -> nat -> context_type -> result A
5    | result_impossible : state -> string -> result A
6    | result_not_implemented : string -> result A
7    | result_bottom : state -> result A
8    .
```

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

```
Record input := make_input {
    prompt_string : stream string ;
    random_boolean : stream bool
  }.
```

```
Record output := make_output {
    output_string : list string
  }.
```

```
Record state := make_state {
    inputs :> input ;
    outputs :> output ;
    state_memory :> memory ;
    state_context : context
  }.
```

1. JSCert;
2. Representing imperativity in a functional setting;
3. Semantics in Coq;
4. Semantic sizes;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The full state+error monad;
9. Inputs and outputs.

1. R
2. R Trust Sources
3. Eyeball Closeness
4. R Features