A Coq Formalisation of a Core of R

Martin Bodin

Center for Mathematical Modeling, Universidad de Chile

13th 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

```
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

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

```
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

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

```
f <- function (expr) {
    x <- 2
    y <- 3
    eval (substitute (expr))        # Evaluates the body “expr”
                                    # in the local 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) {
    x <- 2
    y <- 3
    eval (substitute (expr))        # Evaluates the body “expr”
                                    # in the local 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

Correctness

JSCERT

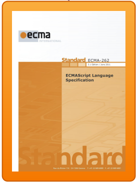
JSREF

Step / rule correspondence

Running tests

COQ definitions

Industrial world


ecma

Standard ECMA-262

ECMAScript Language Specification

Standard


ECMAScript Language test262

ECMAScript.org

Testing complete!

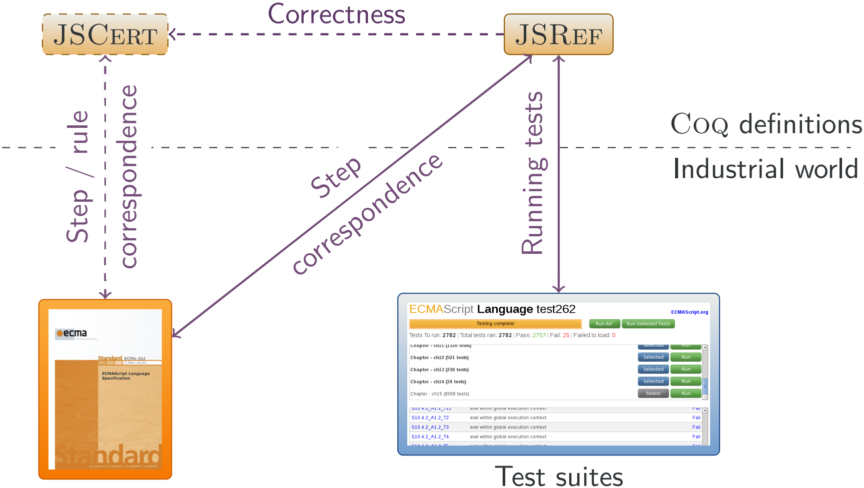
Tests To run: **2782** | Total tests ran: **2782** | Pass: 2757 | Fail: 25 | Failed to load: 0

Chapter - ch11 (1320 tests)	Selected Run
Chapter - ch12 (521 tests)	Selected Run
Chapter - ch13 (230 tests)	Selected Run
Chapter - ch14 (24 tests)	Selected Run
Chapter - ch15 (8068 tests)	Select Run

S10.4.2_A1.2_T11	eval within global execution context	Fail
S10.4.2_A1.2_T2	eval within global execution context	Fail
S10.4.2_A1.2_T3	eval within global execution context	Fail
S10.4.2_A1.2_T4	eval within global execution context	Fail

Test suites

JSCERT Overview



JSCERT

Correctness

JSREF

Step / rule correspondence

Step correspondence

Running tests

COQ definitions

Industrial world

ecma INTERNATIONAL

Standard ECMA-262

5.1 Edition / June 2011

ECMAScript Language Specification

Standard

ECMAScript Language test262 ECMAScript.org

Testing complete! Run All Run Selected Tests

Tests To run: **2782** | Total tests ran: **2782** | Pass: 2757 | Fail: 25 | Failed to load: 0

Chapter - ch12 (521 tests) Selected Run

Chapter - ch13 (230 tests) Selected Run

Chapter - ch14 (24 tests) Selected Run

Chapter - ch15 (8068 tests) Select Run

S10.4.2_A1.2_T2 eval within global execution context Fail

S10.4.2_A1.2_T3 eval within global execution context Fail

S10.4.2_A1.2_T4 eval within global execution context Fail

Test suites

A Stratified Approach

Rule-based semantics

As usable as possible

Coq monadic interpreter

Close to the C

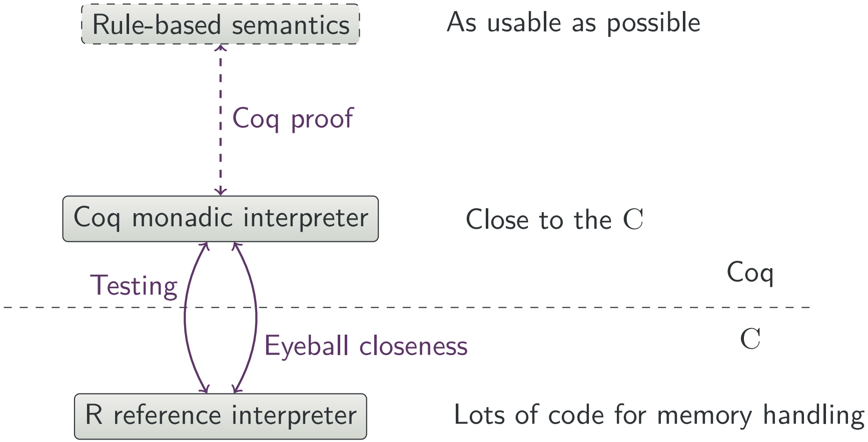
Coq

C

R reference interpreter

Lots of code for memory handling

A Stratified Approach



Rule-based semantics

As usable as possible

Coq proof

Coq monadic interpreter

Close to the C

Testing

Coq

Eyeball closeness

C

R reference interpreter

Lots of code for memory handling

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

```coq
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

```coq
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.

```coq
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

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

Coq code

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

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

```
Notation ”’read%sym’ p_sym ’:=’ p ’using’ S ’in’ cont” :=
  (match read S p with
   | Some p_ =>
     match p_ with
     | symSxp p_sym => cont
     | _ => result_error
     end
   | None => result_error
   end).
```

Eyeball Closeness: C Code

```c
EXP* applyClosure (EXP* op, EXP* arglist, EXP* rho){

  EXP* formals, actuals, savedrho, newrho, res;

  if (rho->type != ENVSXP)
    error (”’rho’ must be an environment.”);

  formals = op->clo.formals;
  savedrho = op->clo.env;

  PROTECT (actuals = matchArgs (formals, arglist));

  /* ... */

  return res;
}
```

Eyeball Closeness: Coq Code

```
Definition applyClosure (S : state) (op arglist rho : EXP_pointer)
    : result EXP_pointer :=

  read%defined rho_ := rho using S in
  ifb type rho_ <> EnvSxp then
    result_error S "'rho' must be an environment."
  else
    read%clo op_clo := op using S in
    let formals := clo_formals op_clo in
    let savedrho := clo_env op_clo in

    let%success actuals := matchArgs S formals arglist using S in

    (* ... *)

    result_success S res.
```

R Features

R Core

```
FUNTAB R_FunTab[] = {
  {”if”,        do_if,        2},
  {”while”,     do_while,     2},
  {”break”,     do_break,     0},
  {”return”,    do_return,    1},
  {”function”,  do_function,  -1},
  {”<-”,        do_set,       2},
  {”(”,         do_paren,     1},
  /* ... */
  {”+”,         do_arith1,    2},
  {”-”,         do_arith2,    2},
  {”*”,         do_arith3,    2},
  {”/”,         do_arith4,    2},
  /* ... */
  {”cos”,       do_math20,    1},
  {”sin”,       do_math21,    1},
  {”tan”,       do_math22,    1},
  /* ... */ }
```

R Core

```
FUNTAB R_FunTab[] = {
  {”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.

```
  {”tan”,       do_math22,    1},
  /* ... */ }
```

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.).

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

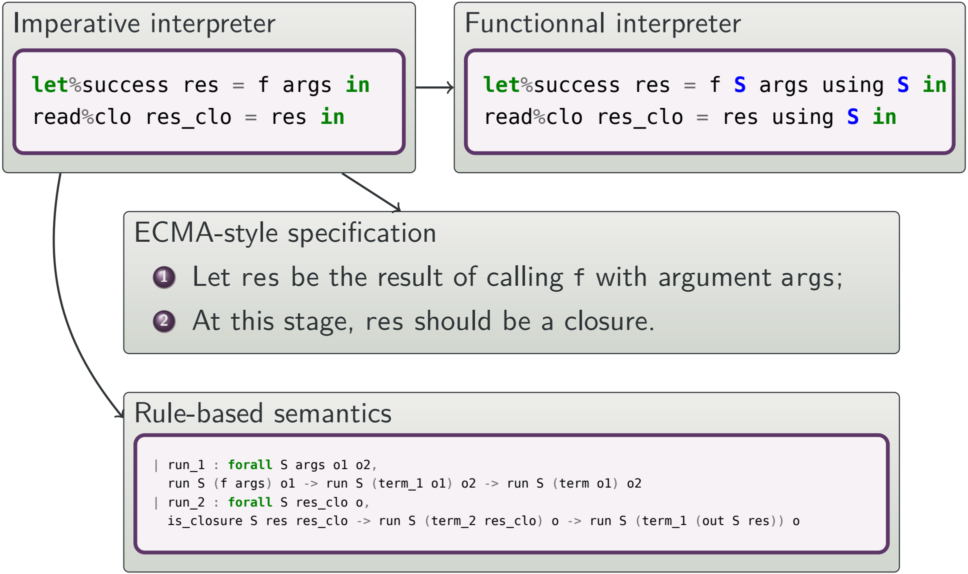
- Building a rule-based formalisation;
- A more functional interpreter.

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

Proof that $1+1$ reduces to 2 in JSCert

```
Lemma one_plus_one_exec : forall S C,
  red_expr S C one_plus_one (out_ter S (prim_number two)).
Proof.
  intros. unfold one_plus_one.
  eapply red_expr_binary_op.
   constructor.
   eapply red_spec_expr_get_value.
    eapply red_expr_literal. reflexivity.
   eapply red_spec_expr_get_value_1.
   eapply red_spec_ref_get_value_value.
  eapply red_expr_binary_op_1.
   eapply red_spec_expr_get_value.
    eapply red_expr_literal. reflexivity.
   eapply red_spec_expr_get_value_1.
   eapply red_spec_ref_get_value_value.
  eapply red_expr_binary_op_2.
  eapply red_expr_binary_op_add.
   eapply red_spec_convert_twice.
    eapply red_spec_to_primitive_pref_prim.
   eapply red_spec_convert_twice_1.
    eapply red_spec_to_primitive_pref_prim.
   eapply red_spec_convert_twice_2.
  eapply red_expr_binary_op_add_1_number.
   simpl. intros [A|A]; inversion A.
   eapply red_spec_convert_twice.
    eapply red_spec_to_number_prim. reflexivity.
   eapply red_spec_convert_twice_1.
    eapply red_spec_to_number_prim. reflexivity.
   eapply red_spec_convert_twice_2.
  eapply red_expr_puremath_op_1. reflexivity.
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

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

~ 3,000 lines of code

JSRef

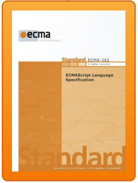
~ 6,000 lines of proof

Step / rule correspondence

Running tests

Coq definitions

Industrial world



ecma INTERNATIONAL

Standard ECMA-262

5.1 Edition / June 2011

ECMAScript Language Specification

Standard

~ 900 steps

ECMAScript **Language** test262

ECMAScript.org

Testing complete!

Tests To run: **2782** | Total tests ran: **2782** | Pass: 2757 | Fail: 25 | Failed to load: 0

Chapter - ch11 (1320 tests) Selected Run

Chapter - ch12 (521 tests) Selected Run

Chapter - ch13 (230 tests) Selected Run

Chapter - ch14 (24 tests) Selected Run

Chapter - ch15 (8068 tests) Select Run

S10.4.2_A1.2_T2	eval within global execution context	Fail
S10.4.2_A1.2_T3	eval within global execution context	Fail
S10.4.2_A1.2_T4	eval within global execution context	Fail

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

```
if_success (run s1 p) (fun s2 =>
  let s3 = write s2 x v in
  if_success (run s3 p’) (fun s4 =>
    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

```
Inductive semantics : state -> prog -> state -> Prop ->

  | semantics_skip : forall s p, semantics s p s

  | semantics_seq : forall s1 s2 s3 p1 p2,
    semantics s1 p1 s2 ->
    semantics s2 p2 s3 ->
    semantics s1 (seq p1 p2) s3

  | semantics_asgn : forall s x v,
    semantics s (asgn x v) (write s x v)
  .
```

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 $(\mathit{Throw}, V, \mathit{empty})$.
5. If $o_2.\mathit{value}$ is empty, let $V = o_1.\mathit{value}$, otherwise let $V = o_2.\mathit{value}$.
6. Return $(o_2.\mathit{type}, V, o_2.\mathit{target})$.

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`.

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

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

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

Sequence in JSCERT

```
Inductive red_stat : state → scope → stat → out → Prop :=

| red_stat_seq_1 : forall S C s1 s2 o1 o,
  red_stat S C s1 o1 →
  red_stat S C (seq_1 s2 o1) o →
  red_stat S C (seq s1 s2) o

| red_stat_seq_2 : forall S C s2 o1,
  abort o1 →
  red_stat S C (seq_1 s2 o1) o1

| red_stat_seq_3 : forall S0 S C s2 o2 o,
  red_stat S C s2 o2 →
  red_stat S C (seq_2 o2) o →
  red_stat S0 C (seq_1 s2 (out_ter S)) o

(* ... *).
```

SEQ-①(s_1, s_2)

$$\dfrac{S, C, s_1 \Downarrow o_1 \qquad o_1, seq_1\ \ s_2 \Downarrow o}{S, C, seq\ s_1\ \ s_2 \Downarrow o}$$

SEQ-②(s_2)

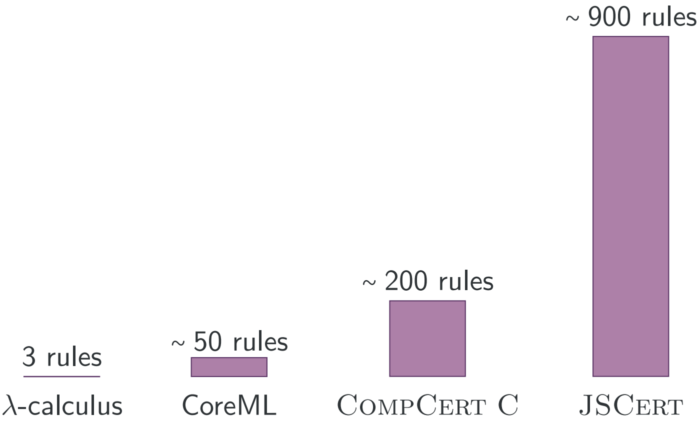
$$\dfrac{}{o_1, seq_1\ \ s_2 \Downarrow o_1} \qquad \textbf{abort}\ o_1$$

SEQ-③(s_2)

$$\dfrac{o_1, s_2 \Downarrow o_2 \qquad o_1, o_2, seq_2 \Downarrow o}{o_1, seq_1\ \ s_2 \Downarrow o} \qquad \neg\textbf{abort}\ o_1$$

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

```
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

```
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

```
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

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

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

Coq code, third try

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

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

Eyeball Closeness: Reading Pointers

C code

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

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

Coq code, fourth try

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

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

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

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

```
Inductive result (A : Type) :=
  | result_success : state -> A -> result A
  | result_error : state -> string -> result A
  | result_longjump : state -> nat -> context_type -> result A
  | result_impossible : state -> string -> result A
  | result_not_implemented : string -> result A
  | result_bottom : state -> result A
  .
```

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.