A Trustworthy Mechanized Formalization of R

Martin Bodin[1,3] Tomás Diaz[1] Éric Tanter[1,2]

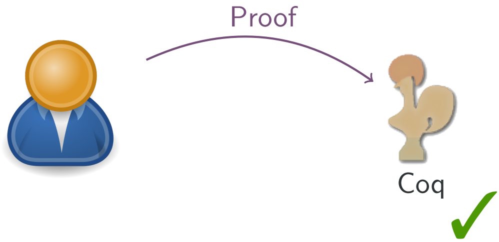
[1]University of Chile

[2]Inria

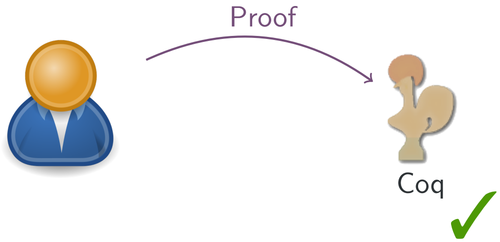
[3]Imperial College London

DLS'18

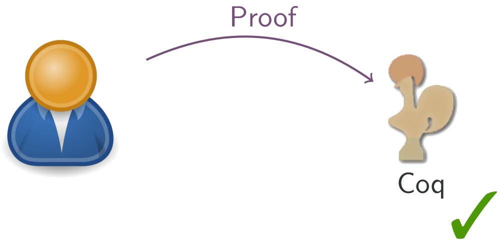
The Coq Proof Assistant



The Coq Proof Assistant




```
Theorem OKprogram : forall state,
  OK state ->
  exists result,
    eval state program = Some result /\ OK result.
Proof.
  (* Lots of lines of proof *)
Qed.
```



This talk

```
Theorem OKprogram : forall state,
  OK state ->
  exists result,
    eval state program = Some result /\ OK result.
Proof.
  (* Lots of lines of proof *)
Qed.
```

- More than **2 million users** worldwide;
- More than **13,000 packages**:
  - `ggplot2`: elegant data visualisations
  - `lawstat`: tools for public policy, and law;
  - `ptstem`: Stemming algorithms for the Portuguese language;
  - ...
- Used by 70% of **data miners** (24% as primary language).

```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

```
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

```
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 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 "expr" in
                                          # the local environment
  }
f(x + y)                                  # Returns 5
x + y                                     # Raises an error
```

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 "expr" in
                                     # the local environment
  }
f(x + y)                             # Returns 5
x + y                                # Raises an error
```

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

```r
if ("TRUE") 42                # Returns 42
"TRUE" || FALSE               # Type error
```

```r
c(c(1, TRUE), "a")            # Returns c("1", "1", "a")
c(1, TRUE, "a")               # Returns c("1", "TRUE", "a")
```

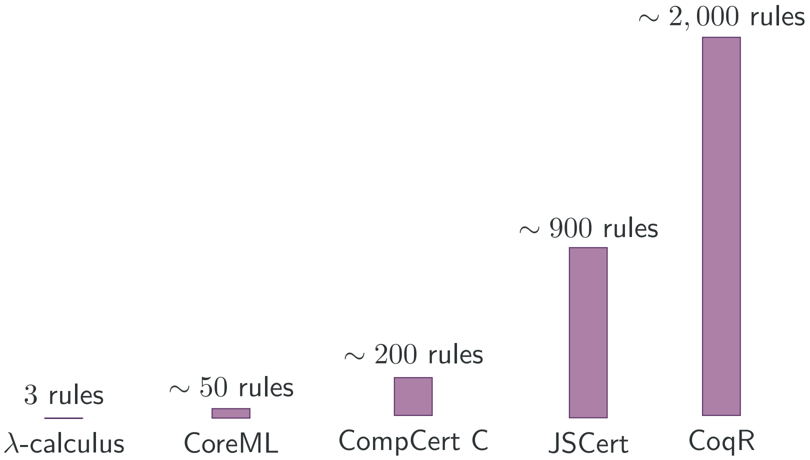
```r
"x" <- 18
x                             # Returns 18

"TRUE" <- 18                  # No error
TRUE                          # Returns TRUE
```

CoqR

- A Coq formalisation of R;
- Supports a non-trivial subset of R, and **fully** support them.

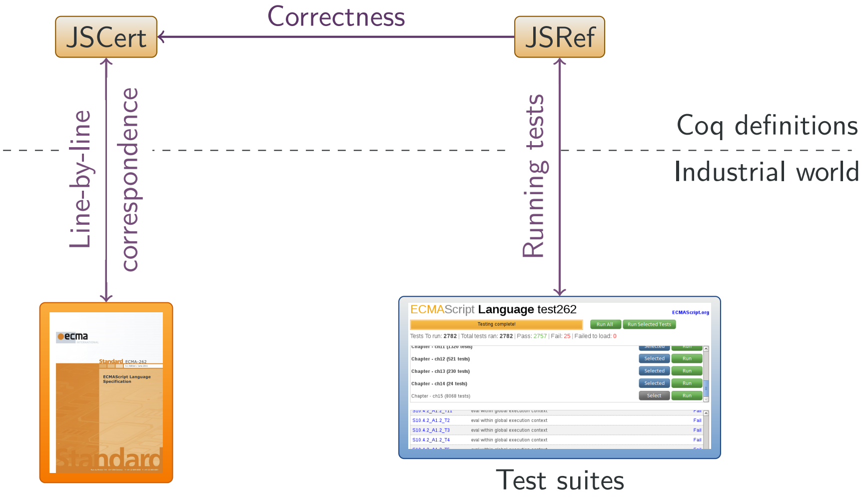
`https://github.com/Mbodin/CoqR`

Semantic Sizes

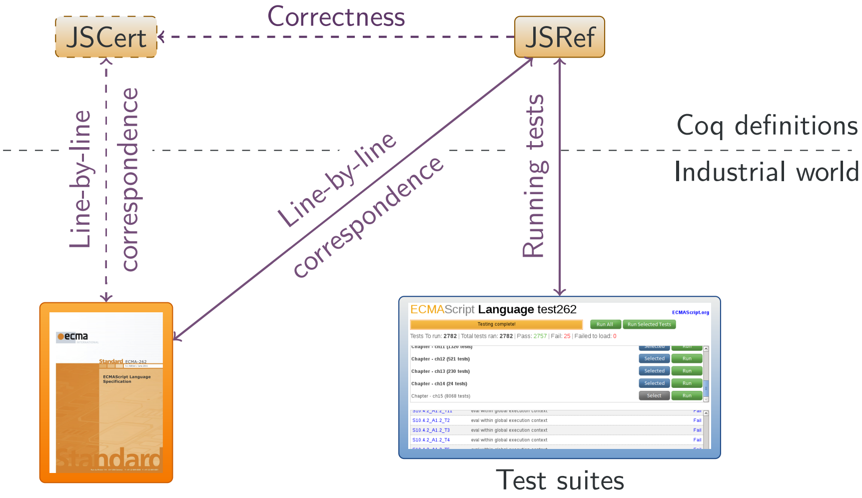


(Rough estimation of the size of each project if we were to entirely translate them into a small-step semantics.)

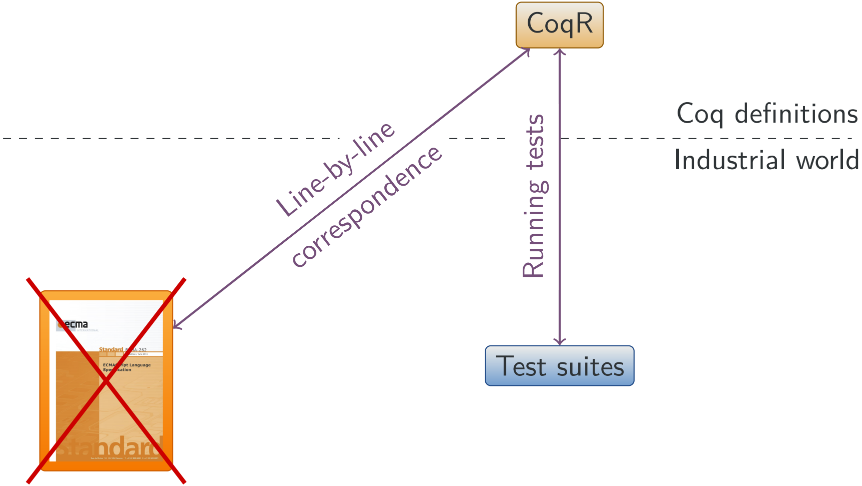
Trusting JavaScript: JSCert



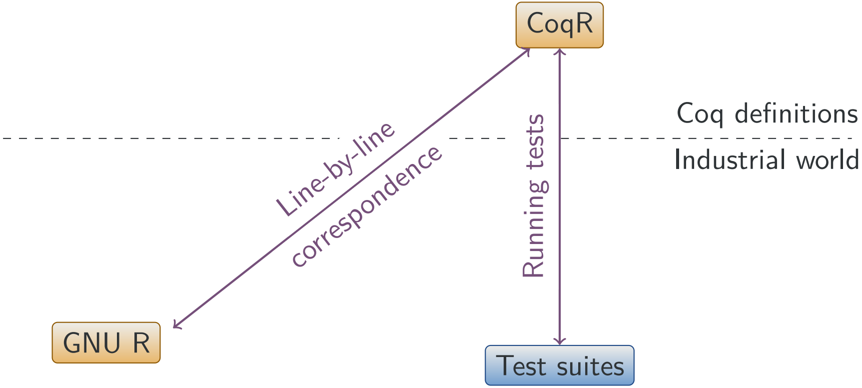
Trusting JavaScript: JSCert



Trusting R


CoqR
Coq definitions
Industrial world
Line-by-line correspondence
Running tests
Test suites
ecma
Standard ECMA-262
ECMAScript Language Specification
standard

Trusting R



CoqR
Line-by-line
correspondence
Running tests
Coq definitions
Industrial world
GNU R
Test suites

How close CoqR is from GNU R?

Thanks to monads and Coq notations, pretty close.

Line-to-line Correspondence: C Code from GNU R

```c
SEXP do_attr
    (SEXP call, SEXP op, SEXP args, SEXP env){
  SEXP argList, car, ans;
  int nargs = R_length (args);
  argList =
    matchArgs (do_attr_formals, args, call);
  PROTECT (argList);
  if (nargs < 2 || nargs > 3)
    error ("Wrong argument count.");
  car = CAR (argList);
  /* ... */
  return ans;
}
```

Line-to-line Correspondence: Coq Code from CoqR

```
Definition do_attr globals runs S
    (call op args env : SEXP) : result SEXP :=
  let%success nargs :=
    R_length globals runs S args using S in
  let%success argList :=
    matchArgs globals runs S
      do_attr_formals args call using S in
  if nargs <? 2 || nargs >? 3 then
    result_error S "Wrong argument count."
  else
    read%list car, _, _ := argList using S in
    (* ... *)
    result_success S ans.
```

Line-to-line Correspondence

```
SEXP do_attr
    (SEXP call, SEXP op, SEXP args, SEXP env){
  SEXP argList, car, ans;
  int nargs = R_length (args);
  argList =
    matchArgs (do_attr_formals, args, call);
  PROTECT (argList);
  if (nargs < 2 || nargs > 3)
    error ("Wrong argument count.");
  car = CAR (argList);
  /* ... */
  return ans;
}
```

```
Definition do_attr globals runs S
    (call op args env : SEXP) :=
  let%success nargs :=
    R_length globals runs S args using S in
  let%success argList :=
    matchArgs globals runs S
      do_attr_formals args call using S in
  if nargs <? 2 || nargs >? 3 then
    result_error S "Wrong argument count."
  else
    read%list car, _, _ := argList using S in
    (* ... *)
    result_success S ans.
```

Line-to-line Correspondence

```c
SEXP do_attr
    (SEXP call, SEXP op, SEXP args, SEXP env){
  SEXP argList, car, ans;
  int nargs = R_length (args);
  argList =
    matchArgs (do_attr_formals, args, call);
  PROTECT (argList);
  if (nargs < 2 || nargs > 3)
    error ("Wrong argument count.");
  car = CAR (argList);
  /* ... */
  return ans;
}
```

```coq
Definition do_attr globals runs S
    (call op args env : SEXP) :=
  let%success nargs :=
    R_length globals runs S args using S in
  let%success argList :=
    matchArgs globals runs S
      do_attr_formals args call using S in
  if nargs <? 2 || nargs >? 3 then
    result_error S "Wrong argument count."
  else
    read%list car, _, _ := argList using S in
    (* ... *)
    result_success S ans.
```

Line-to-line Correspondence

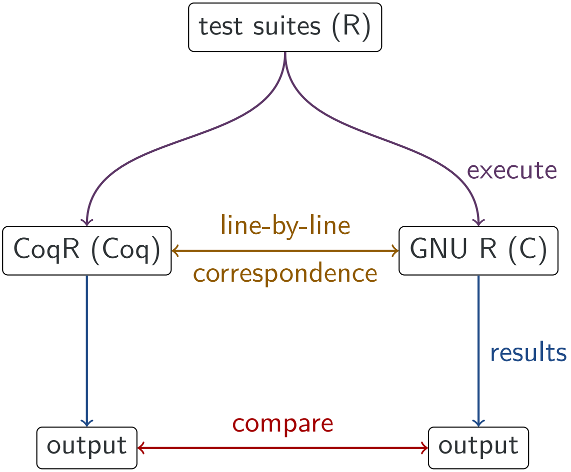
```c
SEXP do_attr
    (SEXP call, SEXP op, SEXP args, SEXP env){
  SEXP argList, car, ans;
  int nargs = R_length (args);
  argList =
    matchArgs (do_attr_formals, args, call);
  PROTECT (argList);
  if (nargs < 2 || nargs > 3)
    error ("Wrong argument count.");
  car = CAR (argList);
  /* ... */
  return ans;
}
```

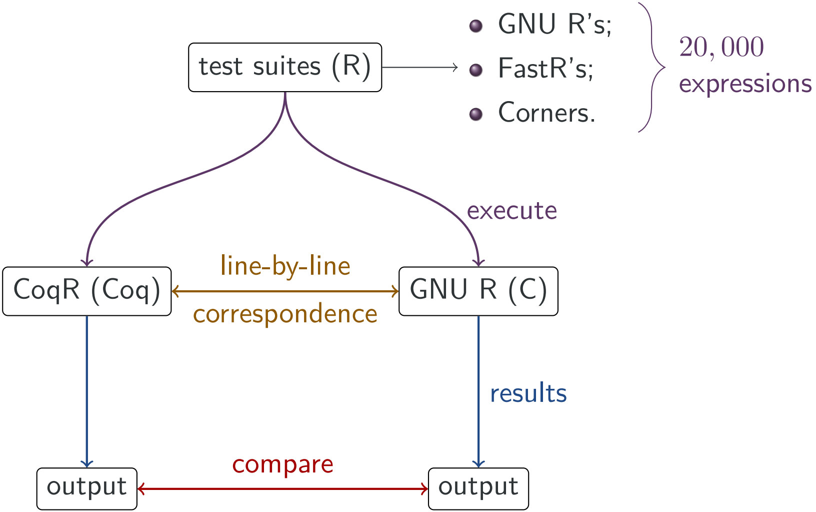
```coq
Definition do_attr globals runs S
    (call op args env : SEXP) :=
  let%success nargs :=
    R_length globals runs S args using S in
  let%success argList :=
    matchArgs globals runs S
      do_attr_formals args call using S in
  if nargs <? 2 || nargs >? 3 then
    result_error S "Wrong argument count."
  else
    read%list car, _, _ := argList using S in
    (* ... *)
    result_success S ans.
```

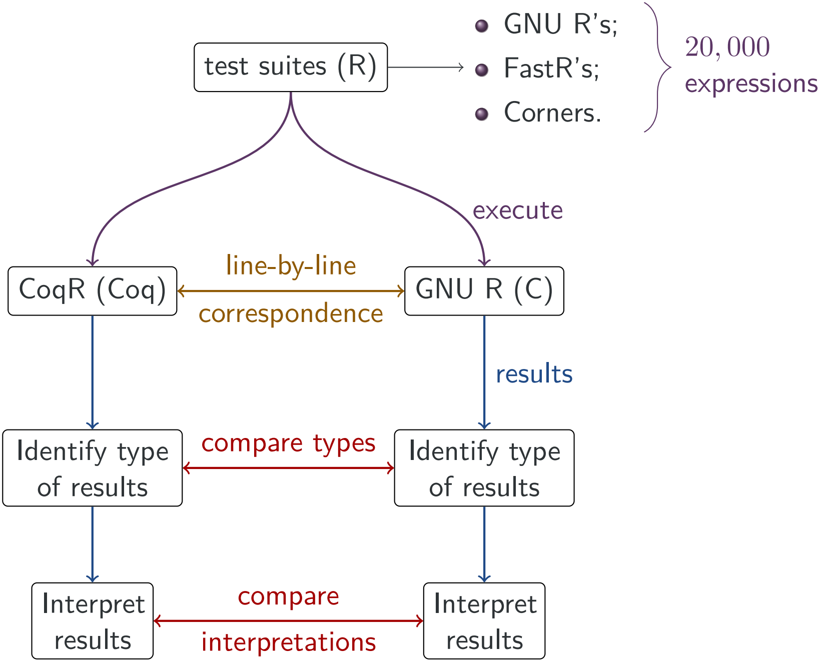
Not an exact match, but easily verifiable

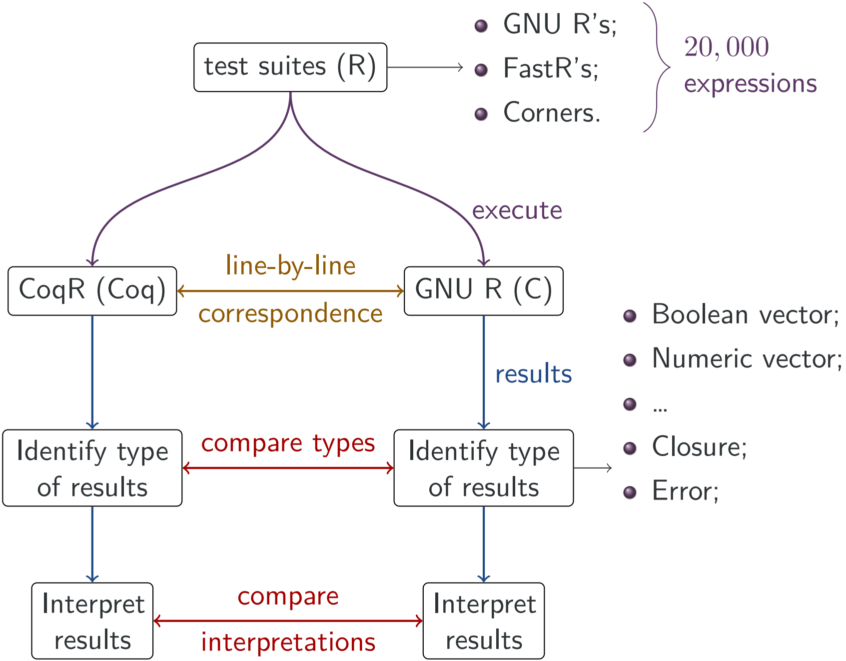
- Monads encode the semantics of GNU R's subset of C;
- Coq notations ease the line-to-line correspondence;
- Main differences:
  - the global state is propagated all along;
  - no garbage collection.

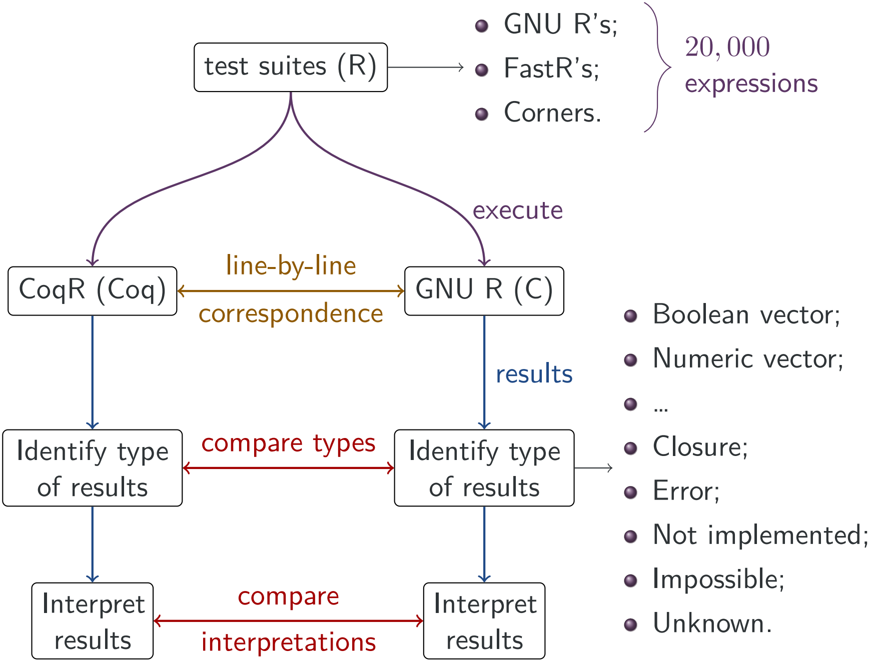

CoqR (Coq)
line-by-line
correspondence
GNU R (C)


test suites (R)
execute
CoqR (Coq)
line-by-line
correspondence
GNU R (C)
results
compare
output
output


test suites (R)
GNU R's;
FastR's;
Corners.
20, 000
expressions
execute
CoqR (Coq)
line-by-line
correspondence
GNU R (C)
results
output
compare
output


test suites (R)
GNU R's;
FastR's;
Corners.
$20,000$ expressions
execute
CoqR (Coq)
line-by-line
correspondence
GNU R (C)
results
Identify type of results
compare types
Identify type of results
Interpret results
compare
interpretations
Interpret results


test suites (R)
GNU R's;
FastR's;
Corners.
$20,000$ expressions
execute
CoqR (Coq)
line-by-line
correspondence
GNU R (C)
results
Boolean vector;
Numeric vector;
...
Closure;
Error;
Identify type of results
compare types
Identify type of results
Interpret results
compare interpretations
Interpret results


test suites (R)
GNU R's;
FastR's;
Corners.
20,000 expressions
execute
CoqR (Coq)
line-by-line correspondence
GNU R (C)
results
Identify type of results
compare types
Identify type of results
Boolean vector;
Numeric vector;
...
Closure;
Error;
Not implemented;
Impossible;
Unknown.
Interpret results
compare interpretations
Interpret results

https://coqr.dcc.uchile.cl

COREUI

2613	7	48	119	0	149	20 / 6
PASSED	FAILED	NOT IMPLEMENTED	NOT FOUND	IMPOSSIBLE	UNKNOWN	POTENTIAL PASS/FAIL

Test Detail

Pass | Fail | Not Implemented | Not Found | Impossible | Unknown | Potential Pass | Potential Fail

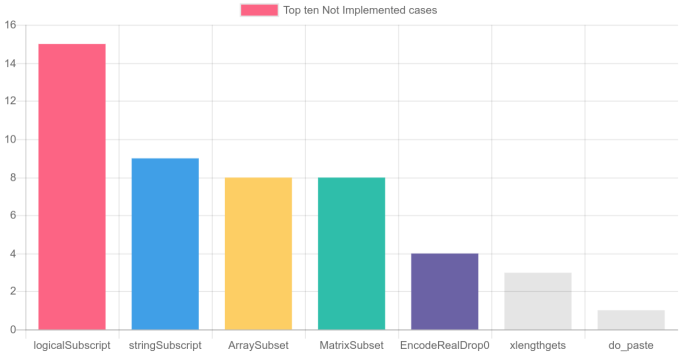
	Filename	L...	Expression	Coq Raw Output	R Raw Output
▸	substitute.R	3	substitute()	""	
▸	do_cat.R	4	.Internal (cat (list (), 1, "-", 1000, "", FALSE))	NULL	NULL
▸	do_cat.R	3	.Internal (cat (list ("Hello", "world"), 1, " ", 100...	Hello world NULL	Hello world NULL
▸	do_makevec...	72	vector ("character", 1L)	[1] NULL	[1] ""
▸	do_makevec...	71	vector ("complex", 1L)	[1] 0+0i	[1] 0+0i
▸	do_makevec...	60	vector ("pairlist", 1L)	(pairlist: NULL)	[[1]] NULL
▸	do_makevec...	39	vector ("integer", 0L)	integer(0)	integer(0)

Previous | Page 4 of 15 | 10 rows | Next

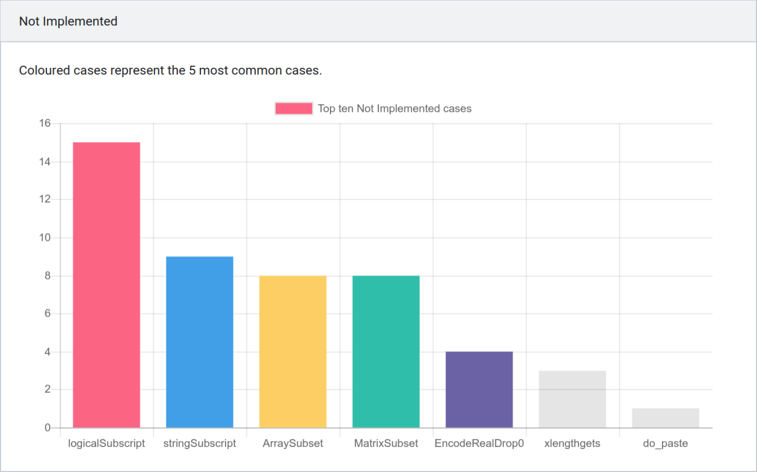
Identifying low-hanging fruits

Not Implemented

Coloured cases represent the 5 most common cases.



Identifying low-hanging fruits



CoqR supports a **non-trivial** subset of R, and fully supports them.

This way of doing things is generic!

From:

- Two interpreters with similar inputs;
- A set of result types;
- A meaningful way to interpreter these results

We get:

- A customized testing framework;
- Meaningful testing results;
- A way to prioritise functions to be implemented.

In CoqR we trust

Let's build proofs!

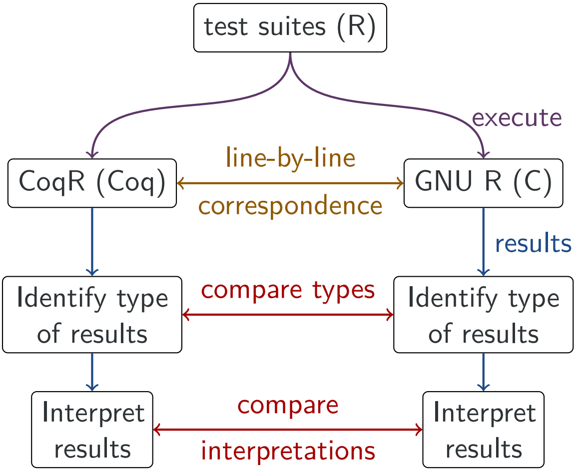
Typing Invariants

```
Inductive safe_SExp S : SExp -> Prop :=
  | safe_ListStruct : forall car cdr tag,
      may_have_types S [NilSxp ; ListSxp] cdr ->
      may_have_types S [NilSxp ; CharSxp] tag ->
      safe_SExp S (make_ListStruct car cdr tag)
  | safe_StrStruct : forall data,
      (forall a, Mem a data ->
        may_have_types S [CharSxp] a) ->
      safe_SExp S (make_StrStruct data)
  (* ... *).
```

Tactic Usage

```
Lemma do_attr_result :
  forall S globals call op args env,
  safe_state S ->
  safe_globals S globals ->
  safe_pointer S args ->
  may_have_types S [NilSxp; ListSxp] args ->
  (* ... *)
  result_prop (fun S' ans =>
      safe_state S' /\ safe_globals S' globals
      /\ safe_pointer S' ans)
    (do_attr globals runs S call op args env).
Proof.
  introv OKS OKglobals OKargs Targs. unfolds do_attr.
  cutR R_length_result. computeR.
  cutR matchArgs_result. computeR.
  (* ... *)
Qed.
```

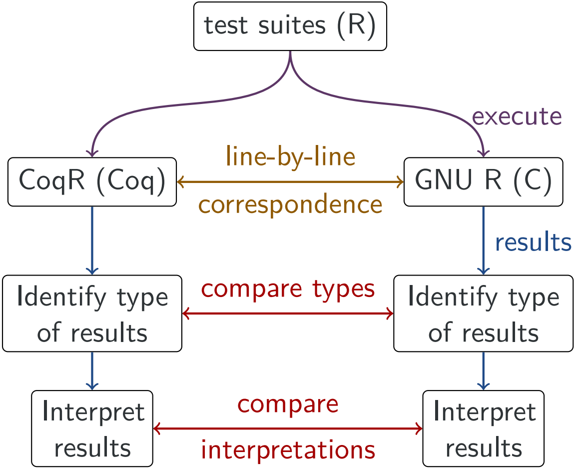
Conclusion



https://github.com/Mbodin/CoqR

https://coqr.dcc.uchile.cl

Thank you for listening!



https://github.com/Mbodin/CoqR

https://coqr.dcc.uchile.cl

1. R
2. CoqR
3. Line-to-line Correspondence
4. Testing Framework

Bonuses

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

R: A Lazy Programming Language

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

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

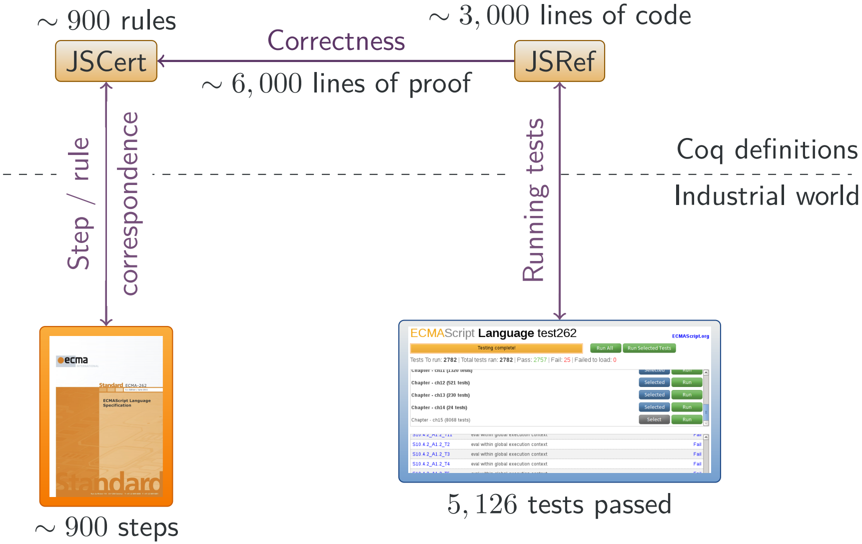
R: A Lazy Programming Language

```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
```

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

The JSCert Project



1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

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. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

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 $(\texttt{throw}, V, \mathit{empty})$.
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`.

$$\textsc{seq}\text{-}1(s_1, s_2)\ \frac{S, C, s_1 \Downarrow o_1 \qquad o_1, \mathit{seq}_1\ s_2 \Downarrow o}{S, C, \mathit{seq}\ s_1\ s_2 \Downarrow o}$$

$$\textsc{seq}\text{-}2(s_2)\ \frac{}{o_1, \mathit{seq}_1\ s_2 \Downarrow o_1}\quad \textbf{abort}\ o_1$$

$$\textsc{seq}\text{-}3(s_2)\ \frac{o_1, s_2 \Downarrow o_2 \qquad o_1, o_2, \mathit{seq}_2 \Downarrow o}{o_1, \mathit{seq}_1\ s_2 \Downarrow o}\quad \neg\textbf{abort}\ o_1 \qquad \ldots$$

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

(* ... *).
```

Sequence in JSCert

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

```
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-❷(s_2)

$$\frac{}{o_1, seq_1\ s_2 \Downarrow o_1} \quad \textbf{abort}\ o_1$$

SEQ-❸(s_2)

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

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

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

```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)
```

```r
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. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

Line-to-line Correspondence: 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`.

Line-to-line Correspondence: 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.

Line-to-line Correspondence: 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`.

Line-to-line Correspondence: 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
  .
```

Line-to-line Correspondence: Reading Pointers

C code

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

Coq code, fourth try

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

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

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

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

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

```
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. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

Line-to-line Correspondence

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

Line-to-line Correspondence: 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
  (* ... *)
  .
```

Line-to-line Correspondence: 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
  }.
```

Line-to-line Correspondence: 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.

Line-to-line Correspondence: 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).
```

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

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. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

```
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. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

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},
  /* ... */ }
```

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

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
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

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

List:	header	car	cdr	tag		
Integer vector:	header	size	i_1	i_2	$\cdots$	i_n
Complex vector:	header	size	c_1	c_2	$\cdots$	c_n

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

https://coqr.dcc.uchile.cl

COREUI

2613	7	48	119	0	149	20 / 6
PASSED	FAILED	NOT IMPLEMENTED	NOT FOUND	IMPOSSIBLE	UNKNOWN	POTENTIAL PASS/FAIL

Test Detail

Pass | Fail | Not Implemented | Not Found | Impossible | Unknown | Potential Pass | Potential Fail

	Filename	Line	Expression	Coq Raw Output	R Raw Output
▸	ControlFlow.R	76	if (1:3) NA else NULL	Error: The condition has le...	[1] NA Warning message: I...
▸	do_set.R	40	T <<- 1	[1] 1	Error: cannot change value ...
▸	do_arith.R	11	x * x	[1] 4611686014132420609	[1] NA Warning message: I...
▸	do_arith.R	5	NA + 2.5	[1] NaN	[1] NA
▸	do_arith.R	4	2.5 + NA	[1] NaN	[1] NA
▸	do_arith.R	3	1 + NA	[1] NaN	[1] NA
▸	attr.R	102	"attr<-" <<- function (x, y, va...	(closure)	Error: cannot change value ...

Previous | Page 1 of 1 | 10 rows | Next

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

Full results

Suite	P	F	NI	NF	I	U	PP	PF
Corners	2,613	7	48	119	0	149	20	6
GNU R	243	31	739	723	1	27	0	0
FastR1	1,103	25	987	115	0	161	59	326
FastR2	2,411	1,128	6,888	493	0	1,914	297	343
Total	6,370	1,191	8,662	1,450	1	2,251	376	675

total number of tests: 20,976

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Other Subtleties of R;
6. Reading pointers;
7. Parsing R;
8. The eyeball closeness;
9. The full monad;
10. R features;
11. Inputs and outputs;
12. RExplain;
13. Basic language elements in memory;
14. More details about the website's results;
15. Full testing results.

1. R
2. CoqR
3. Line-to-line Correspondence
4. Testing Framework