# A Trustworthy Mechanized Formalization of R

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

[1]Imperial College London

[2]Millennium Institute Foundational Research on Data

[3]Inria

FMfSS'19

# The Coq Proof Assistant

## A success story

- $CH_2O$: a specification of C, `ch2o.cs.ru.nl`
- CompCert: a fully certified C compiler, `compcert.inria.fr`
- Verasco: a fully certified C analyser, `verasco.imag.fr`
- JSCert: a specification of JavaScript, `jscert.org`
- And a lot of other formalisations and theorems.

What about R?

# 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
```

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

# R's Corner Cases

**cran.r-project.org/doc/manuals/r-release/R-lang.html**

`if ( statement1 ) statement2 else statement3`

- First, `statement1` is evaluated to yield `value1`.
- If `value1` is a logical vector then `statement2` is evaluated when the first element of `value1` is `TRUE`, and `statement3` is evaluated when the first element of `value1` is `FALSE`.
- If `value1` is a numeric vector then `statement3` is evaluated when the first element of `value1` is zero and otherwise `statement2` is evaluated.
- If `value1` has any type other than a logical or a numeric vector an error is signalled.

```r
if (numeric(0)) 42 else 18
if ("TRUE") 42 else 18
"TRUE" || FALSE
```

# R's Corner Cases

**cran.r-project.org/doc/manuals/r-release/R-lang.html**

`if ( statement1 ) statement2 else statement3`

- First, `statement1` is evaluated to yield `value1`.
- If `value1` is a logical vector then `statement2` is evaluated when the first element of `value1` is `TRUE`, and `statement3` is evaluated when the first element of `value1` is `FALSE`.
- If `value1` is a numeric vector then `statement3` is evaluated when the first element of `value1` is zero and otherwise `statement2` is evaluated.
- If `value1` has any type other than a logical or a numeric vector an error is signalled.

```r
if (numeric(0)) 42 else 18  # Error
if ("TRUE") 42 else 18
"TRUE" || FALSE
```

# R's Corner Cases

**cran.r-project.org/doc/manuals/r-release/R-lang.html**

`if ( statement1 ) statement2 else statement3`

- First, `statement1` is evaluated to yield `value1`.
- If `value1` is a logical vector then `statement2` is evaluated when the first element of `value1` is `TRUE`, and `statement3` is evaluated when the first element of `value1` is `FALSE`.
- If `value1` is a numeric vector then `statement3` is evaluated when the first element of `value1` is zero and otherwise `statement2` is evaluated.
- If `value1` has any type other than a logical or a numeric vector an error is signalled.

```r
if (numeric(0)) 42 else 18   # Error
if ("TRUE") 42 else 18       # Returns 42
"TRUE" || FALSE
```

# R's Corner Cases

**cran.r-project.org/doc/manuals/r-release/R-lang.html**

`if ( statement1 ) statement2 else statement3`

- First, `statement1` is evaluated to yield `value1`.
- If `value1` is a logical vector then `statement2` is evaluated when the first element of `value1` is `TRUE`, and `statement3` is evaluated when the first element of `value1` is `FALSE`.
- If `value1` is a numeric vector then `statement3` is evaluated when the first element of `value1` is zero and otherwise `statement2` is evaluated.
- If `value1` has any type other than a logical or a numeric vector an error is signalled.

```r
if (numeric(0)) 42 else 18   # Error
if ("TRUE") 42 else 18       # Returns 42
"TRUE" || FALSE              # Error
```

# Consequences of such Corner Cases

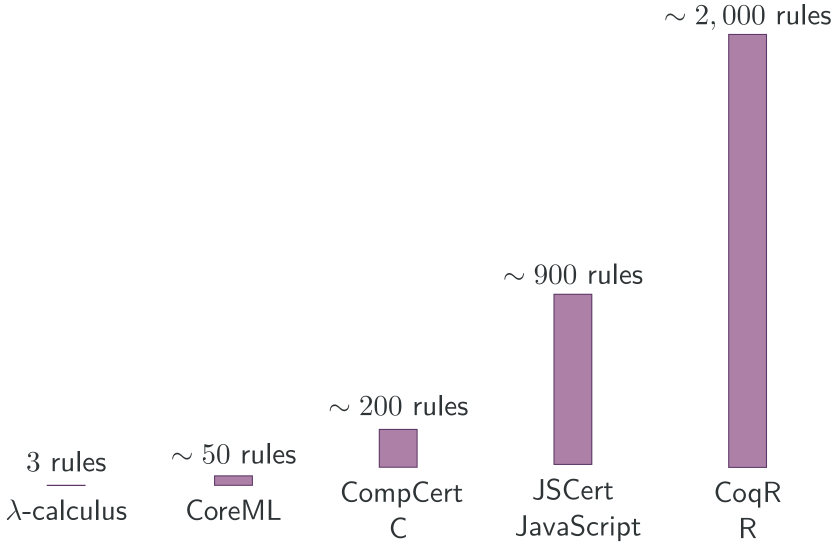
- Programmers may be aware of some corner cases, but not all.
- Programs can be difficult to understand or certify.
- A program may return a wrong result... that looks right.
- We need formal verification.

# CoqR

- A Coq formalisation of R;
- Supports a non-trivial subset of R, and **fully** support them.
- Can be used to prove theorems about R programs.

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

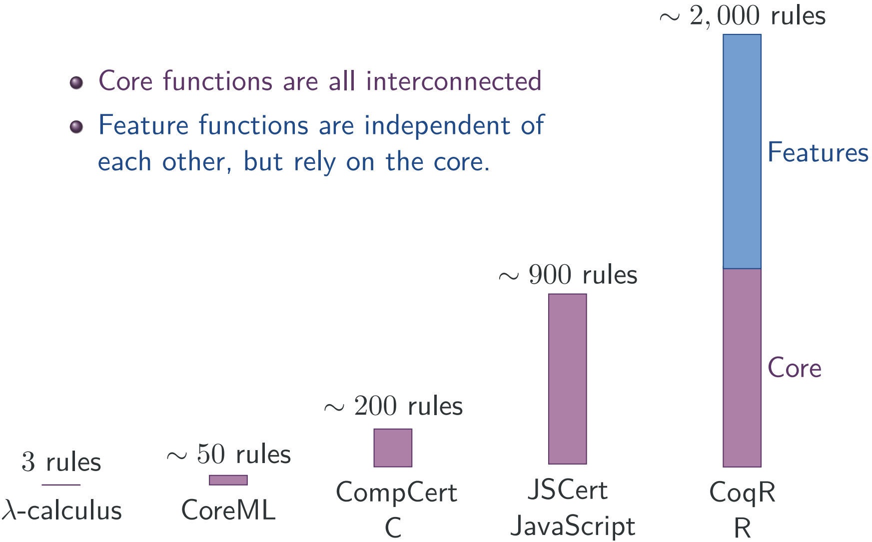
# Semantic Sizes




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

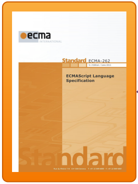
# Semantic Sizes

- Core functions are all interconnected
- Feature functions are independent of each other, but rely on the core.




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

# Trusting JavaScript: JSCert




Specification

Line-by-line correspondence

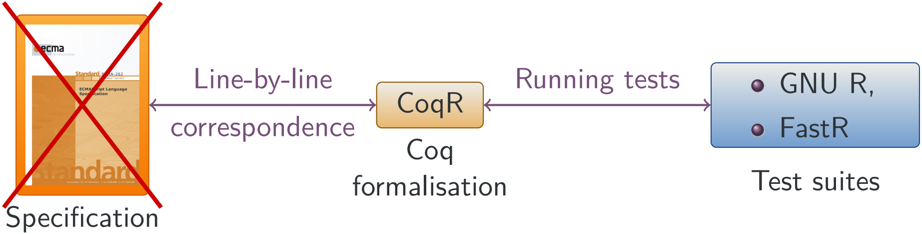
JSCert

Coq formalisation

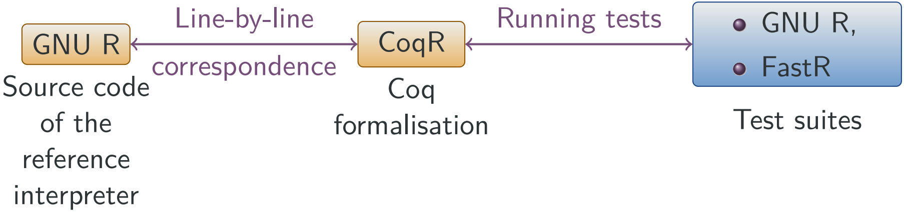
Running tests


Test suites

# Trusting R: CoqR



# Trusting R: CoqR



# How close CoqR is from GNU R?

Thanks to monads and Coq notations, pretty close.

# Line-by-line Correspondence

## GNU R

```
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;
}
```

## CoqR

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

# Line-by-line Correspondence

## GNU R

```
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;
}
```

## CoqR

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

```
Definition if_success A B (a : result A) (f : A -> result B)
    : result B := fun S =>
  let r := a S in
  match r with
  | result_success a S' => f a S'
  | _ => r
  end.

Notation "'let%success' a ':=' r 'in' cont" :=
  (if_success r (fun a => cont)).
```

# Line-by-line Correspondence

## GNU R

```
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;
}
```

## CoqR

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

# Line-by-line Correspondence

## GNU R

```
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;
}
```

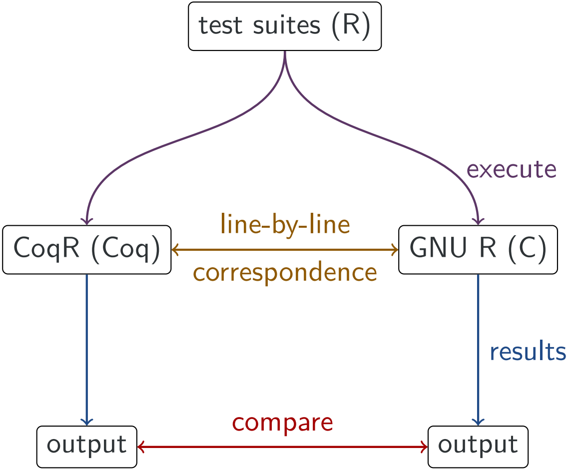
## CoqR

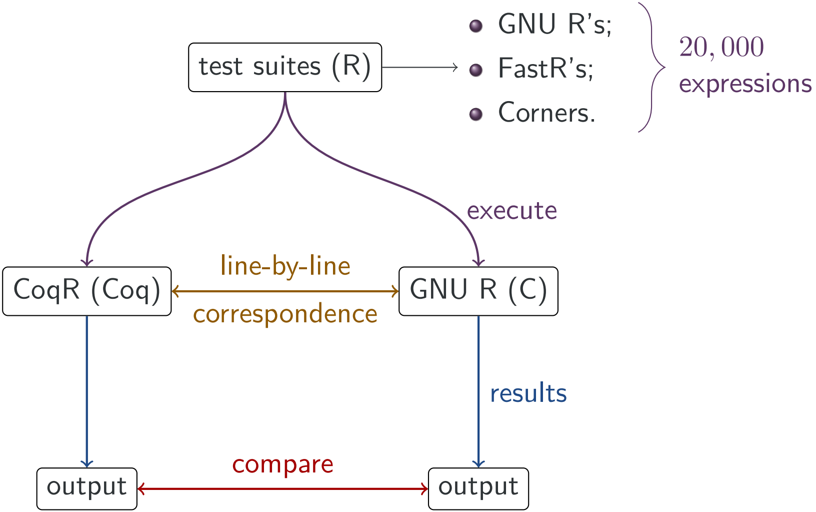
```
Definition do_attr
    (call op args env : SEXP)
    : result SEXP :=
  let%success nargs := R_length args in
  let%success argList :=
    matchArgs do_attr_formals args call in
  if nargs <? 2 || nargs >? 3 then
    result_error "Wrong argument count."
  else
    read%list car, _, _ := argList in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

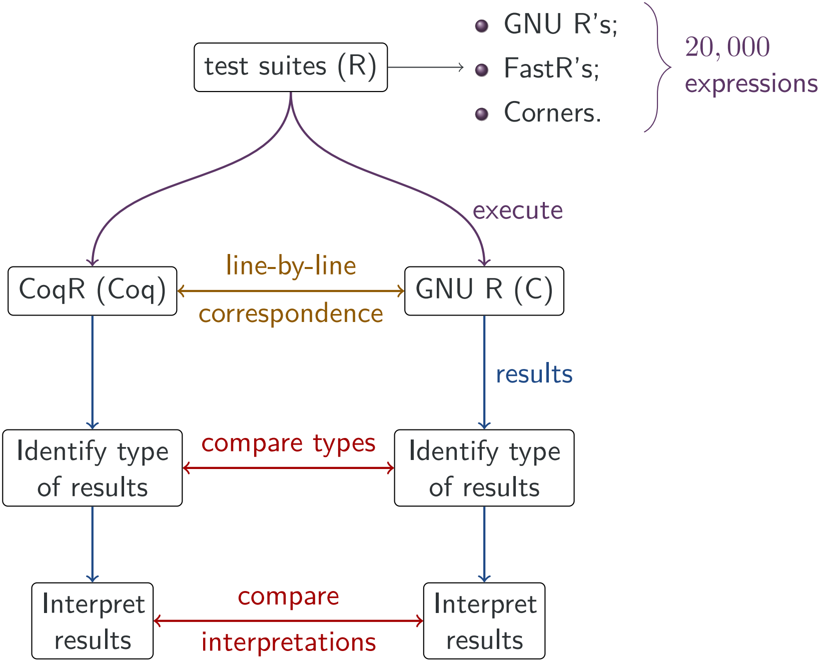
### Not an exact match, but easily verifiable

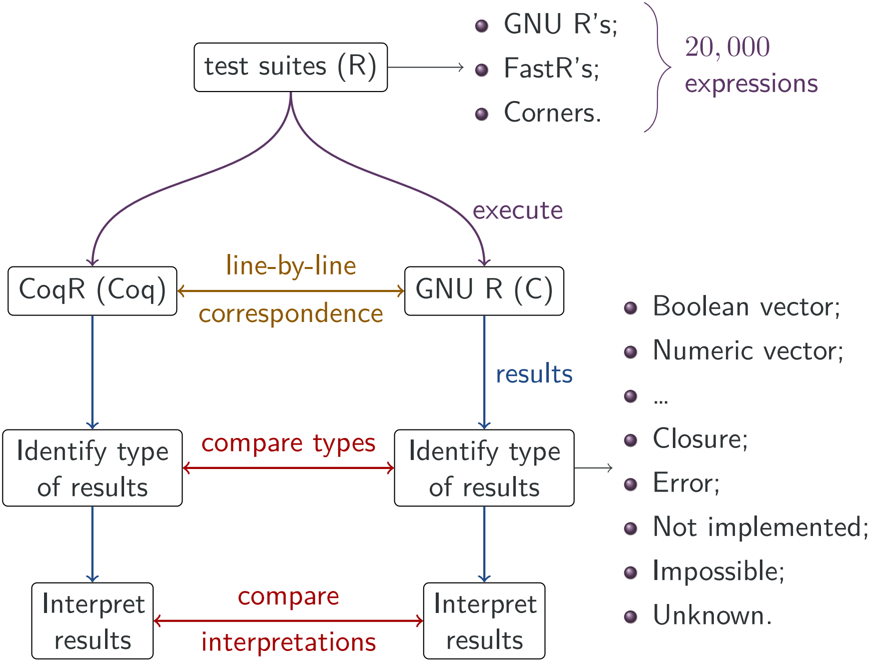
- Monads encode the semantics of GNU R's subset of C;
- Coq notations ease the line-to-line correspondence.

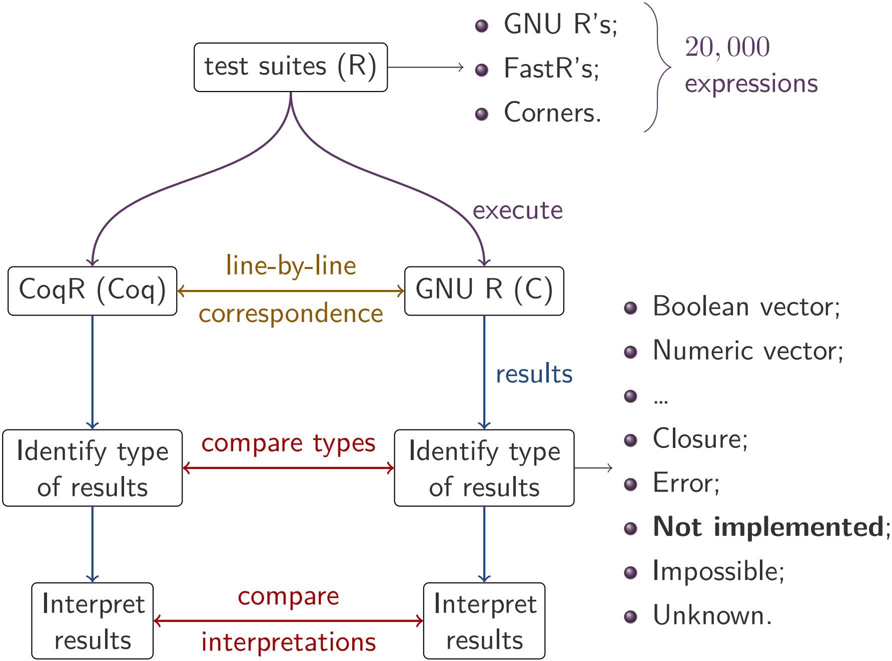
$$\boxed{\text{CoqR (Coq)}} \xleftrightarrow[\text{correspondence}]{\text{line-by-line}} \boxed{\text{GNU R (C)}}$$


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

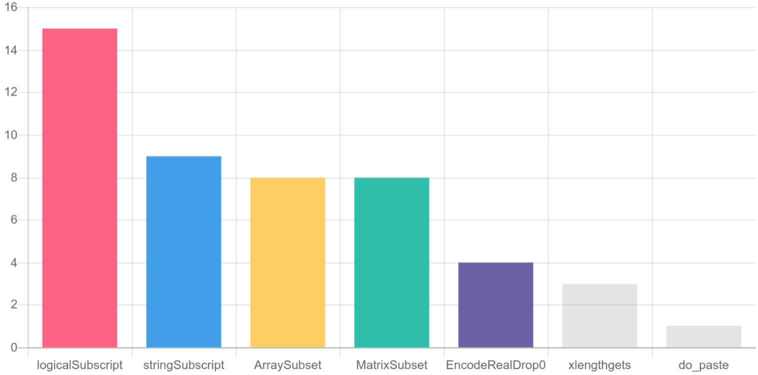

test suites (R)
GNU R's;
FastR's;
Corners.
20, 000
expressions
execute
CoqR (Coq)
line-by-line
correspondence
GNU R (C)
results
output
compare
output


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

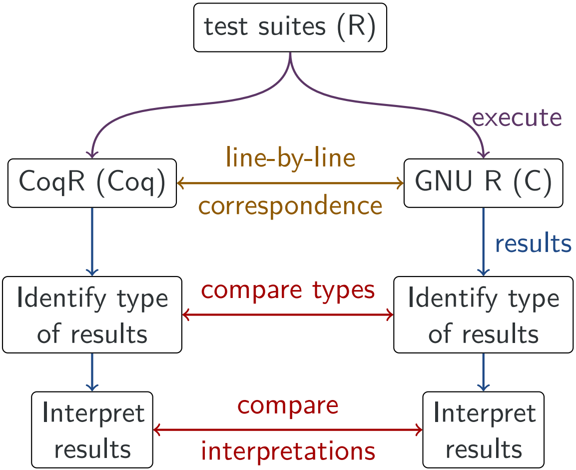

test suites (R)
GNU R's;
FastR's;
Corners.
$20,000$ expressions
execute
CoqR (Coq)
line-by-line
correspondence
GNU R (C)
results
Identify type of results
compare types
Identify type of results
Boolean vector;
Numeric vector;
...
Closure;
Error;
Not implemented;
Impossible;
Unknown.
Interpret results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


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

# Not implemented cases, and their usages in our test suites




- CoqR supports a **non-trivial** subset of R.
- Is this subset large enough?
- Would you be interested in adding a feature you care about?


test suites (R)
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

# Conclusion

## CoqR

- A formalisation of R in Coq;
- Two means of **trusting** it:
  - line-by-line correspondence and testing;
- Can be used to build **proofs about R programs**.

## Questions and Future Work

- Is CoqR big enough for your programs?
- What kind of properties would you like to be able to prove with CoqR?
- CoqR is quite complex: how to ease proofs about such a beast?

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

# Thank you for listening!

## CoqR

- A formalisation of R in Coq;
- Two means of **trusting** it:
  - line-by-line correspondence and testing;
- Can be used to build **proofs about R programs**.

## Questions and Future Work

- Is CoqR big enough for your programs?
- What kind of properties would you like to be able to prove with CoqR?
- CoqR is quite complex: how to ease proofs about such a beast?

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

# Bonuses

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

```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
```

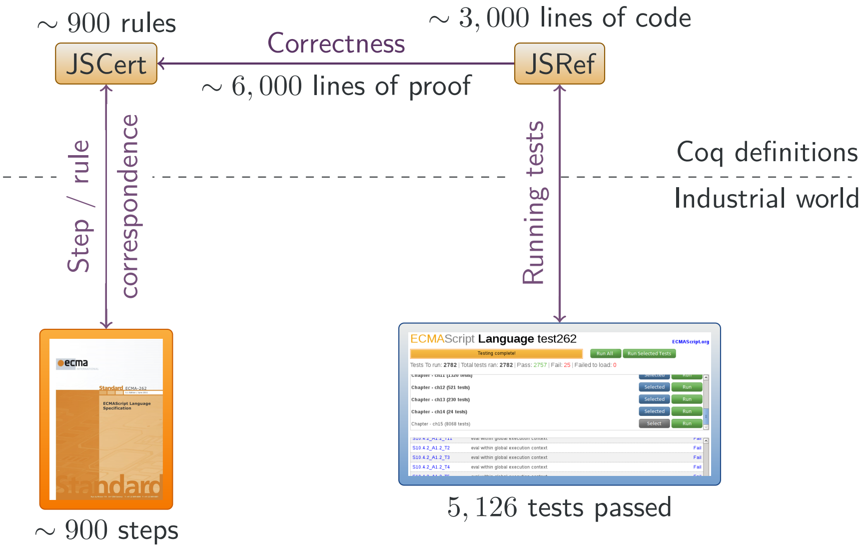
# R: A Lazy Programming Language

```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
```

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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`.

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

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

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

# 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

```
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

(* ... *).
```

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

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

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

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

```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
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]          # ??
```

1. R: A Lazy Programming Language;
2. JSCert;
3. Representing imperativity in a functional setting;
4. Semantics in Coq;
5. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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.
```

# Line-by-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-by-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-by-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`.

C code

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

Coq code, third try

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

- 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
  .
```

# Line-by-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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. Full testing results.

# Line-by-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-by-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
  (* ... *)
  .
```

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-by-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-by-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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. 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. Subsets in R;
6. Other Subtleties of R;
7. Reading pointers;
8. Parsing R;
9. The eyeball closeness;
10. The full monad;
11. R features;
12. Inputs and outputs;
13. RExplain;
14. Basic language elements in memory;
15. More details about the website's results;
16. Full testing results.