A Trustworthy Mechanized Formalization of R

Martin Bodin
Center for Mathematical Modeling &
Computer Science Department
University of Chile
mbodin@dim.uchile.cl

Tomás Diaz
Pleiad Lab & IMFD Chile
Computer Science Department
University of Chile
tdiaz@dcc.uchile.cl

Éric Tanter
Pleiad Lab & IMFD Chile
Computer Science Department
University of Chile
etanter@dcc.uchile.cl

Abstract

The R programming language is very popular for developing statistical software and data analysis, thanks to rich libraries, concise and expressive syntax, and support for interactive programming. Yet, the semantics of R is fairly complex, contains many subtle corner cases, and is not formally specified. This makes it difficult to reason about R programs. In this work, we develop a big-step operational semantics for R in the form of an interpreter written in the Coq proof assistant. We ensure the trustworthiness of the formalization by introducing a monadic encoding that allows the Coq interpreter, CoqR, to be in direct visual correspondence with the reference R interpreter, GNU R. Additionally, we provide a testing framework that supports systematic comparison of CoqR and GNU R. In its current state, CoqR covers the nucleus of the R language as well as numerous additional features, making it pass a significant number of realistic test cases from the GNU R and FastR projects. To exercise the formal specification, we prove in Coq the preservation of memory invariants in selected parts of the interpreter. This work is an important first step towards a robust environment for formal verification of R programs.

Keywords R, Coq, Language formalization, Testing infrastructure

1 Introduction

The R programming language [28, 12, 29] has gotten a lot of traction in recent years, being used by millions of users in areas as varied as biology and finance. This diversity among R programmers results in largely different programming styles.

*T. Diaz and É. Tanter are partially funded by the Millenium Institute for Foundational Research on Data (IMFD Chile)

DLS '18, November 6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6030-2/18/11…$15.00
https://doi.org/10.1145/3276945.3276946

```
1 f <- function (abc, ab, de) { c (abc, ab, de) }
2 f(1, 2, 3)           # By position
3 f(de=3, abc=1, ab=2) # By name
4 f(1, d=3, 2)         # Mixed
5 f(3, a=1, ab=2)      # a is associated to abc
6 f(a=3, 1, 2)         # Error: several prefixes
```

Figure 1. Exploring function calls in R.

In fact, the language itself is community driven and reflects this diversity. The R programming language is meant to be both expressive and powerful, able to express complex notions in few keystrokes. This sometimes comes at the cost of readability and predictability. Indeed, the semantics of R is subtle and contains numerous corner cases that can result in unexpected behavior.

The reasons for these corner cases are numerous, ranging from backward compatibility to the desire to accommodate the use of R as both a traditional programming language and an interactive shell. Even a feature as basic as function calls can be a source of surprise in R. Indeed, there are many ways to call a function, and in particular there are two ways to provide an argument: either by position or by name (including prefixes). To illustrate the consequences of this feature, Figure 1 defines a function `f` that concatenates its three arguments. The first call associates arguments by position, and the second by name. The third call mixes both mechanisms, and exploits the prefix feature: `d` is associated to `de` because it is the only argument whose name starts with `d`. Now, if more than one argument matches by prefix, then R rejects the call and throws an error, as in the fifth call in Figure 1. However, exact matches are not counted in this process: in the fourth call, the name `ab` is an exact match and thus only the argument `abc` is left to be associated to `a`, leading to no error thrown. R also comprises a `...` feature to define functions accepting any number of arguments. This feature adds an exception to the prefix rule: the variables declared after the `...` special argument are not considered during prefix matching, but only for exact matches.

Such subtle behaviors are numerous in R [3]. Debugging tools exist [22], but they cannot always compensate for the complex semantics of R. Consequently, unexpected bugs occur in R programs, and reasoning about such programs, even informally, can be difficult. Formal methods support reasoning about the behavior of programs. In particular, proof assistants such as Coq [5] enable us to formally prove program

properties. But to formally prove that an R program meets its specification, we first need a formal semantics of R. While there exists a language definition document [27], this document is unfit as the basis of a verification effort: it is both a specification effort and a manual, written in plain English, with ambiguities and incomplete at times. Additionally, we found several mismatches between the text description and the behavior of the reference interpreter, GNU R [29].[1] Crucially, the formal semantics should account for all the corner cases of the R semantics, such as the function call conventions described above, implicit type conversions, and so on. This is necessary because these corner cases are a typical place were bugs appear and are hard to track. A complete semantics for the full R language will inevitably be complex, because of the large amount of such corner cases.

This complexity in turn raises a *trust* issue: how do we know that a formal semantics properly models the real-world language it is supposed to describe? Such trustworthiness is crucial to justify the relevance of any formal reasoning based on such semantics, such as proofs of language properties or of properties of specific programs. Being able to relate a formalism to *trust sources* is a crucial aspect of the formalization process, and often requires a large amount of work to be done properly [17]. This challenge has been faced repeatedly in attempts to provide formal foundations to JavaScript, for instance. Some formalization approaches like λ_{JS} [11] and KJS [25] augment trust through extensive testing and comparison with existing implementations. JSCert [2] also uses testing and further augments trust through the notion of a so-called *eyeball correspondence*, i.e. a line-to-line syntactic connection, between the formalization (in Coq) and the ECMAScript specification (in English and pseudo-code).

Contributions. We present CoqR, a trustworthy formalization of the R programming language in the Coq proof assistant. The formalization is a big-step operational semantics, in the form of an interpreter (Section 2). We say that this interpreter is *trustworthy* because we have followed two complementary techniques to maximize trust: eyeball correspondence, and testing.

First, the Coq interpreter has been written using a monadic encoding that allows for a direct eyeball correspondence with the C source code of GNU R [29]. In the absence of a standardized formal semantics, GNU R is the reference point that defines what R really is. Note that other alternative implementations of R, such as FastR [14], also treat GNU R as the ground truth.

Second, we have developed a testing framework that streamlines the process of running both the Coq interpreter and GNU R on a set of test cases and report on mismatches and errors, among others (Section 3). We use several test suites to measure the advancement of the CoqR implementation.

We also report on a proof effort to establish that memory invariants are preserved during execution, including some specific proof automation (Section 4). Section 5 discusses related work and Section 6 concludes.

The current implementation of CoqR[2] is not feature-complete: R is a very large programming language and achieving 100% coverage of the language and its main libraries is a huge implementation effort. The contribution of this work is to provide realistic foundations for a trustworthy R formalization in Coq. First, the interpreter covers the nucleus of the language and over 120 additional features. It is implemented in an extensible manner, following the architecture of GNU R itself. Second, the testing infrastructure we provide is designed to help drive the development effort towards the most pressing missing features. At present, the CoqR implementation passes a substantial number of tests from real-world R projects. This is realistic enough to support the claim that, given more engineering power, our approach can scale up to the full language and its main libraries.

2 CoqR: An R Interpreter in Coq

We develop the formal semantics of R in the form of an interpreter defined in Coq; therefore, CoqR is a big-step operational semantics. Operational semantics written as a Coq recursive function is usually not the best fit for Coq proofs—an inductive definition of the operational semantics is usually more convenient for reasoning—but our approach comes with a crucial advantage: it can be run, and thus tested. We show how we defined CoqR in order to achieve the first of the two mechanisms in place for trust, namely the eyeball correspondence with the GNU R interpreter. GNU R's overall structure is fairly standard for an interpreter, so the principles exposed here should be reusable in other contexts.

2.1 Bridging the Gap between C and Coq

The basic principle of the eyeball correspondence is that every one or two lines of the Coq interpreter should correspond to one or two lines of the C reference interpreter.

Of course, achieving a close correspondence between C and Coq versions of the same program is quite challenging, because C and Coq are widely different programming languages: Coq is purely functional whilst global side-effects are frequent in C. Furthermore, Coq is designed to reject any function whose behavior is not entirely defined (it is for instance impossible to miss a case in a pattern-matching), whilst C is known for its undefined behaviors. Finally, contrary to C, Coq programs are required to terminate.

Modeling effectful computation in a purely functional language can be done through the use of *monads* [34]. A monad is essentially a data structure that denotes computation augmented with information that is threaded throughout evaluation using a *binder*. For instance, the *state* monad threads

[1] For instance, according to the language definition, `if ("TRUE") 42` should raise an error, whilst the interpreter returns 42.

[2] https://github.com/Mbodin/CoqR/releases/tag/DLS2018

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

Figure 2. The result monad.

values through a functional computation, accessible via monadic operations like `put` and `get`, thereby simulating a mutable global store. Other typical monads includes the *option* monad to represent potentially-failing computation (with constructors such as `Some` and `None`), and the *fuel* monad to give a finite evaluation budget to potentially diverging computation. Adequate notation, as provided for instance in Haskell, allows monadic programs to look like imperative programs, hiding the additional monadic information.

Therefore, in order to achieve an effective eyeball correspondence between C code and Coq code, we introduce a *result monad*, which combines both the state, error, and fuel monads (Figure 2). The result monad constructors are:[3]

- The main constructor is `result_success`, used when a computation is successful. In addition to the result (of type `A` in the definition), it carries the global state (of type `state`). We describe the representation of state later on in this section.
- The constructor `result_error` is meant to catch errors thrown by GNU R, for instance an R runtime typing error: these errors are not catchable and immediately end the execution. A **`string`** is provided to help the debugging process.
- The constructor `result_longjump` corresponds to a call to the `longjmp` C function in GNU R source code. It only appears in constructs involving non-local jumps, such as **`break`** or **`return`**. The C behavior is mimicked thanks to a special monadic binder, explained further below.
- Unspecified C behaviors (such as dereferencing an invalid pointer) are translated into `result_impossible`. Getting this result immediately ends the Coq interpreter: it is meant to be unreachable. Observing such a result either means CoqR has a bug, or that an actual bug in GNU R has been found, which could possibly be undetectable by running GNU R due to a C compiler optimization. At this date, we have not encountered the latter situation.
- Given the size and complexity of the R language, it is important to be able to execute the Coq interpreter without it being fully complete. The `result_not_implemented` constructor is thus important for development; it is treated specifically in our testing framework, for instance to help identity which features one should focus on next.
- Finally, `result_bottom` is returned to end the execution when reaching the maximum number of executed instructions (also called *fuel*). This is meant to artificially make our interpreter terminate, despite the fact that the interpreted R program may not (see Section 2.5 for more details).

[3] A similar monad is used in the JSCert [2] and JSExplain [4] projects, which aim to define a JavaScript interpreter in Coq, readable by non-specialists.

The result monad is associated with a monadic binder, written **`let`**`%success`. This binder expects its argument to evaluate to a result of the form `result_success`. If so, it binds the carried result to a name, and transparently propagates the possibly-updated state. All other kinds of results are transparently propagated to the top-level. A similar monadic binder has been defined to handle the special semantics of `setjmp`: when calling `setjmp`, a default value is provided to continue computations; when a result of the form `result_longjump` reaches its corresponding `setjmp`, the continuation of `setjmp` is run again, using the value carried by `result_longjump` to replace the default one. This obliviousness to global state, errors, jumps, and non-termination is what enables a close correspondence between a program written in both languages.

2.2 Eyeball Correspondence

By developing CoqR as a monadic interpreter using the result monad introduced above, we are able to achieve an eyeball correspondence between the C and Coq interpreters. This correspondence is extremely helpful during development: whenever a bug is encountered while testing, we localize the function responsible for the bug, then directly and visually compare both interpreters. Such checks are quick and easy to perform, often leading to a quick fix of CoqR.

Figure 3 shows an example of the eyeball correspondence that can be achieved using the result monad. Figure 3a shows a C function, and Figure 3b its Coq translation. The `SEXP` type is an alias for `SEXPREC*`, that is, pointers to *basic language elements* (see Section 2.3). The binder **`let`**`%success` is used when calling a function. Thanks to the monadic encoding, the calls to `R_length` and `matchArgs` may return an unsuccessful result, but the code does not need to be explicit about that possibility; the code after a call is only executed if the result denotes success. Similarly, the **`return`** statement from C is translated to `result_success`, and raising an error is performed using `result_error`.

Other differences are that several arguments are systematically passed to each function (`globals`, `runs`, and `S`), the use of `read`**`%list`** instead of `CAR` in the C version, as well as the absence of the `PROTECT` macro in the Coq version. We clarify each of these points in the following subsections.

2.3 Modeling the Heap

We now describe how we modeled GNU R's heap. Various models of C's heap [15, 16] exist in Coq, but in CoqR we exploit the specific structure around which GNU R is built [12]: almost all objects manipulated by GNU R are *basic language elements* (`SEXPREC` in C). Consequently, CoqR's heap is exactly a mapping from pointers to basic language elements.

```c
SEXP do_attr (SEXP call, SEXP op,
              SEXP args, SEXP env){
  SEXP argList, car, ans;
  static SEXP do_attr_formals = NULL;
  /* ... */
  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;
}
```

(a) original C function

```
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.
```

(b) Coq translation

Figure 3. Original C function and Coq translation of `do_attr`

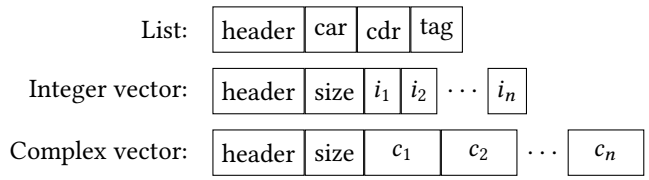

Figure 4. Basic language elements in memory

Each basic language element is composed of a header and some data. The header stores the type of the basic language element, a list of attributes, as well as several boolean informations (for instance whether it can be safely updated in place). There are 24 different types of basic language elements in R, 9 of which being different kinds of vectors. The stored data depends on the type of the element. For instance, lists contain three pointers: one to the first element (named `car`), to the queue of the list (`cdr`), and to an optional name for the first element (`tag`). Vectors store their length, followed by a C array in memory. The size of this array depends both of its length and the type of vector. Figure 4 illustrates this with integer and complex vectors (complexes are composed of two floating-point numbers). The way memory is used in C makes it easy to unguardedly access a cell out of bounds, which would lead to an undefined behavior.

This is an issue, as we cannot directly translate unguarded C accesses into Coq: we need a model of the heap. Figure 5 shows how we defined `SEXPREC` in Coq. Basic language elements are records storing a header and data, which in turn is defined as a sum type. For instance, lists are defined by three pointers. Vectors are records storing their length and a list: in Coq, the data of vector is stored directly in the `SEXPREC` structure and not following it in memory as in C.

To ease readability, coercions have been used extensively. Coercion is a mechanism to mark some constructors as implicit. For instance, if Coq expects a `SEXPRECData` and is given a `ListStruct`, then the constructor `listExp` will be implicitly added thanks to Line 20 of Figure 5. In the context of

```
1  Record ListStruct := make_ListStruct {
2      list_carval : SEXP ;
3      list_cdrval : SEXP ;
4      list_tagval : SEXP
5    }.
6
7  Record Vector_SEXPREC (A : Type) := {
8      Vector_length : nat ;
9      Vector_data :> list A
10   }.
11
12 Inductive SEXPRECData :=
13   | listExp : ListStruct -> SEXPRECData
14   | envExp : EnvStruct -> SEXPRECData
15   | SEXPREC_VectorInteger :
16       Vector_SEXPREC int -> SEXPRECData
17   | SEXPREC_VectorComplex :
18       Vector_SEXPREC complex -> SEXPRECData
19   (* ... *).
20 Coercion listExp : ListStruct >-> SEXPRECData.
21 Coercion envExp : EnvStruct >-> SEXPRECData.
22 (* ... *)
23
24 Record SEXPREC := make_SEXPREC {
25     SEXPREC_header :> SEXPRECHeader ;
26     SEXPREC_data :> SEXPRECData
27   }.
```

Figure 5. Basic language elements (`SEXPREC`) in Coq

CoqR, this is more than a simple syntactic notation as it helps the eyeball correspondence by hiding in the Coq code what is not present in C. Of course, this implicit notation is only one-way: given a `ListStruct`, we can convert it into an `SEXPRECData`, but to perform the converse, we have to pattern-match on the shape of the `SEXPRECData`. This pattern-matching is performed by specific monadic binders. For instance in Figure 3b, `read%list` gets the `SEXPREC` stored in the state `S` and pointed by `argList`, then pattern-matches it as a list, extracting the `car`, `cdr`, and `tag` fields. If the pointer is

not in the domain of the state `S` or if the associated `SEXPREC` object is not a list, then `result_impossible` is returned: this corresponds to an undefined behavior in C (dereferencing an invalid pointer or accessing the wrong projection of a `union`). The accesses in Coq are thus guarded by monadic binders, closely mimicking the unguarded accesses of C.

2.4 Dealing with Global Variables

The GNU R interpreter features over 80 global variables that need initialization, and are subsequently unchanged. The initialization of these internal variables represents a large portion of the source code. These basic language elements are created and stored in global variables before even importing any libraries.

For instance, the often-used variable `R_NilValue` is set to be a list whose `car`, `cdr`, and `tag` fields point to `R_NilValue` itself. This element is used instead of the `NULL` pointer to mark absent elements. Another important global variable is `R_FunTab`, which is the *symbol table* that maps operation and function names to their C implementions. (We exploit this structuring of the interpreter for building CoqR in an incremental manner from a simple nucleus—see Section 2.7.)

The initialization of global variables performs local computations, calling nucleus functions. To avoid this circular dependency—which Coq would not accept—one could parametrize each nucleus function by the value of the global variables it uses. This would however not scale, considering the size of the project and the number of global variables involved. Instead, we parameterize each function of CoqR by a single `globals` environment, which is a mapping from a definite set of global variables to `SEXP`. This additional argument can be seen in the Coq code of Figure 3b: it is passed along at each call site.

To make definitions more convenient, we use coercions to make Coq implicitly perform a lookup in the `globals` environment whenever a global variable is read. Therefore, accessing global variables is transparent, as illustrated by the use of `do_attr_formals` in Figure 3b.

Hidden global variables. In fact `do_attr_formals` is a *hidden* global variable, introduced locally with the C `static` keyword inside the `do_attr` function. Such a variable is persistent across calls, just like a standard global variable.

In place of the commented-out part Line 5 of Figure 3a, the code initializes the `static` variable `do_attr_formals`, as shown in Figure 6a. Upon the first call of `do_attr`, this variable is initialized by a basic language element. The value is kept across calls to avoid a costly reallocation at each call. Observe that this pattern exactly follows the scheme of the other global variables: the variable is initialized once, then never changes. Consequently, we treat `static` variables just like global variables. We extracted out the part of `do_attr` that performs initialization, shown in Figure 6b. This code is executed after the standard global variables are initialized.

2.5 Dealing with Non-Termination

Another additional argument present in Figure 3b is the `runs` argument. This argument aims at factorizing the fuel given to Coq functions to make them artificially terminate. The usual way to do this is to make each function check whether its fuel reached 0, and if so return `result_bottom`. It is however cumbersome to do that manually, and affects the eyeball correspondence, since such a check does not exist in C. We solve this issue by following the same method as in JSCert: instead of a fuel argument, each function takes a record `runs`. This record stores all recursive functions as its projections. Each recursive call in C is then translated by calling the corresponding projection of the record, as in a regular fix-point combinator.

For instance, Figure 7 shows the definition of while loops in CoqR. First the `Runs` record is defined with all the potentially looping functions. A `while`-loop is obviously part of these functions. We then define `while_loop` with an additional argument `runs` of type `Runs`. Line 12 corresponds to the recursive call, but instead of an actual recursive call, we call the `while_loop` function stored in the `runs` record.

Once all functions have been defined, we define `runs` of type `Runs` by linking these functions to themselves. We define `runs` using fuel: at each step, its projections are functions taking a `runs` with less fuel as argument; when reaching 0, all its projections systematically return `result_bottom`. Figure 8 shows its Coq definition. This idiom requires only one pattern-matching on fuel in the whole CoqR interpreter; it is thus a lightweight way to enforce termination.

2.6 Limitations

The use of the `PROTECT` macro in Figure 3a is missing in the Coq translation. This macro use performs a garbage collecting action, namely it saves the object `argList` from garbage collection. We choose not to model the garbage collection aspect of GNU R in CoqR. A garbage collector is an optimization of a language implementation, which is arguably irrelevant for the specification of the language itself (for instance, neither R6RS for Scheme nor ECMAScript for JavaScript even mention garbage collection).

Currently, CoqR only considers ASCII strings, and hence does not support various locales and character encodings supported by R. More generally, CoqR ignores any possibility to dynamically parametrize the behavior of the interpreter using options stored the shell environment.

Note that the eyeball correspondence between GNU R and CoqR would pay off in the future if one wants to extend CoqR to account for some of these features.

2.7 Incremental Development of CoqR

R is by no means a small language. Currently, CoqR spans over 18,000 lines of Coq definitions. This is larger than the interpreter of JSCert, named JSRef [2], which consists of

```
if (do_attr_formals == NULL)
  do_attr_formals =
    allocFormalsList2 (install ("x"),
                       install ("which"));
```

(a) C snippet

```
Definition do_attr_init globals runs S :=
  let%success x :=
    install globals runs S "x" using S in
  let%success which :=
    install globals runs S "which" using S in
  allocFormalsList2 globals S x which.
```

(b) Coq translation

Figure 6. Another snippet of `do_attr` and its Coq translation

```
Record Runs : Type := Runs_intro {
    runs_while_loop : forall A, state -> A ->
      (state -> A -> result bool) ->
      (state -> A -> result A) -> result A ;
    runs_eval : state -> SEXP -> SEXP -> result SEXP ;
    (* ... *)
  }.

Definition while_loop runs A S (a : A) expr stat :=
  if%success expr S a using S then
    let%success a := stat S a using S in
    runs_while_loop runs S a expr stat
  else result_success S a.
```

Figure 7. Definition of loops as a fix-point combinator

```
Fixpoint runs max_step globals : Runs :=
  match max_step with
  | O => {|
      runs_while_loop := fun _ S _ _ _ =>
        result_bottom S ;
      runs_eval := fun S _ _ =>
        result_bottom S ;
      (* ... *) |}
  | S n => {|
      runs_while_loop := fun A S (a : A) expr stat =>
        while_loop globals (runs n) A S a expr stat ;
      runs_eval := fun S e rho =>
        eval globals (runs n) S e rho ;
      (* ... *) |}
  end.
```

Figure 8. Definition of `runs` in Coq

12,500 lines of Coq definitions. Therefore, for pragmatical reasons, it is important to be able to proceed *incrementally* in the development of CoqR. We hence identify a minimal nucleus of R to support initially, and then incrementally extend CoqR. To this end, we exploit the fact that GNU R is structured around a huge *symbol table* with more than 700 entries, corresponding to all functions present in the initial environment of R, whose code is written natively in C.

```
FUNTAB R_FunTab[] = {
  {"if",        do_if,       3,  true,  false},
  {"while",     do_while,    2,  true,  false},
  {"break",     do_break,    0,  true,  false},
  {"return",    do_return,   1,  true,  false},
  {"function",  do_function, -1, true,  false},
  {"<-",        do_set,      2,  true,  false},
  {"(",         do_paren,    1,  true,  true},
  {".Internal", do_internal, 1,  true,  false},
  {"which",     do_which,    1,  false, true},
  {"+",         do_arith1,   2,  true,  true},
  {"-",         do_arith2,   2,  true,  true},
  {"cos",       do_math20,   1,  true,  true},
  {"sin",       do_math21,   1,  true,  true},
  /* ... */ }
```

Figure 9. Symbol table of GNU R

Figure 9 shows an excerpt of the symbol table. This C array associates the name of each function with the corresponding C function implementing it, along with its arity and some additional information.[4] Interestingly, all syntactic constructs correspond to a C function. This includes constructs like `if`, `while`, `return`, and even assignments: although the parser accepts a seemingly imperative syntax, it is internally replaced by Lisp-style function calls. For instance, the two abstract syntax trees generated by GNU R's parser for the two lines below are identical.[5]

```
if (TRUE) x <- 1 else return ()
"if" (TRUE, "<-" (x, 1), "return" ())
```

The symbol table provides us with an opportunity for incremental development, as this array clearly defines a set of functions that can be individually removed from GNU R without breaking the overall interpreter—only the parts using them. We therefore consider the *nucleus* of the R language to consist of the functions used to evaluate R expressions that are *not* present in the symbol table. Nucleus functions include the execution process for function calls, environments, closures, promises (delayed evaluation), as well as the parts initializing the symbol table. Constructs like `if`

[4] Namely, whether the function is to be directly defined in the initial environment or available in the `.Internal` construct, and whether it evaluates lazily or eagerly. Some information has been removed for simplicity.

[5] R usually uses a lazy evaluation strategy: the expression `x <- 1` is only evaluated when actually used. Performing side effects in function arguments can yield to unexpected results, but is acceptable for an `if`.

and `while` are not part of the nucleus, but some assignment functions are, because they are used when calling functions.

All functions in the symbol table are then considered *additional features*. The first version of the interpreter only supported the nucleus of R, allowing us to focus the effort on a very restricted sub-language. We were then able to add additional features one at a time, by implementing the associated function and adding it to the symbol table. In the current CoqR development, out of the 18,000 lines, around 5,000 lines are for the nucleus of R, and 4,000 lines are for additional features. At the time of writing, we support 128 entries from the symbol table.

2.8 Parsing R

Verified software, e.g. written in Coq, needs to interact with unverified code, for instance for performing input/output. This additional code is called the *shim*, and unfortunately, it usually concentrates most bugs in certified software [36]. In the case of CoqR, a particularly important part of the shim is the R parser. Parsing real-world programming languages is known to be challenging, so in order to maximize trust, we first tried to develop the parser in Coq using the Menhir tool [13] in order to obtain a formally-verified parser. Unfortunately, the grammar of GNU R (defined in Bison) does not respect the grammar constraints of Menhir's Coq front-end. We had to fall back on Menhir's OCaml front-end, making the parser part of the shim.

In order to achieve high confidence in the parser implementation, we nevertheless can follow the eyeball correspondence methodology. Consequently, we did not optimize or change GNU R's grammar in any way. While doing so might have enabled us to use the Coq front-end of Menhir, it would have resulted in a grammar very different from GNU R's, and spotting bugs and inconsistencies between both grammars would have been challenging.

Sticking to the original grammar has its own downside, though. First, the Bison grammar of GNU R is ambiguous, with 27 shift/reduce conflicts. As a consequence, both Bison and Menhir make some arbitrary choices, which could in theory be different (we checked through testing that they are not). Another source of potential mismatch between the parsers of GNU R CoqR are new lines. There are contexts in R where new lines are significant and others where they are not. For instance, in `{ function () break + 1 }`, adding a new line after the `function` or `+` keywords does not change the final result, but adding a new line after the `break` keyword does. In such situations, lexers usually produce the new-line token nevertheless, leaving to the parser the choice to ignore it or not. However in GNU R, whenever the lexer encounters a new line, it reads a global boolean variable which indicates whether the lexer should ignore the new line. This variable is controlled by the parser: the lexer and the parser thus communicate through side-effects. Such effects are usually considered to be bad practice as they heavily depend on when the parser calls the lexer (otherwise they might become desynchronized). To keep the eyeball correspondence, we nevertheless built the same communication channel between the lexer and the parser in OCaml. Unfortunately, Bison and Menhir do not always call the lexer the same way. For instance, empty blocks `{}` wrongly lead new lines to be eaten in our parser. This is easily fixable by replacing them by the equivalent R constant `NULL`.

Despite the few differences of behavior, we believe that our methodology helped us reduce the amount of bugs in the parser. The differences of behavior are precisely known and easily fixable in the parsed R code.

3 Testing CoqR

Being able to syntactically relate our interpreter with the source code of GNU R helped us catch numerous bugs—in some sense as a variant of Linus's law.[6] To further augment the level of trust of CoqR, we have developed a testing framework that supports comparative execution of CoqR and GNU R. The framework additionally helps the development process by providing useful reports. This section presents the general testing methodology and the framework we developed to support it, discusses the support for the base library of R, and then presents the results of the current status of CoqR with respect to the chosen test suites.

3.1 Testing Methodology and Framework

Test suites. To exercise CoqR, we exploit different existing test suites: the test suite of GNU R itself [29], and two test suites from the FastR project [14], which we refer to as FastR1[7] and FastR2[8]. These test suites are first curated to eliminate expressions that rely on external effects and resources (internet connection, files, specific packages), or produce charts or PDFs (whose comparison is cumbersome); additionally, we excluded generated tests.[9] In total, these curated test suites include over 17,000 tested expressions.

In addition to these test suites, we have developed our own test suite, referred to as Corners, specifically tailored for testing corner cases of the R language. It is also meant to be used as a faster way of testing some implemented features, during the development process. Corners features around 3,000 tested expressions.

Note that tests come in various shapes: some files provide expressions supposed to evaluate to `TRUE`, some are just supposed not to raise any error, and some tests print on the standard output whether they passed. To uniformly process these test suites, we compare the results produced by GNU R and by CoqR, at each step of execution, as described further below. Also, some tests are supposed to be executed as a

[6] https://en.wikipedia.org/wiki/Linus%27s_Law
[7] https://github.com/h2oai/fastr
[8] https://github.com/oracle/fastr
[9] The details of the filtering can be found on the CoqR Github repository.

normal file, while some tests are meant to be read line-by-line, resetting the environment at each line. We manually marked the latter files with a `#@line` tag, recognized by our testing framework.

Comparing results. After interpretation of the R code, we obtain the raw outputs for each interpreter, GNU R and CoqR. These strings cannot be compared directly for equality as many syntactic details makes them superficially different. Indeed, CoqR's pretty-printer is part of the shim (the non-formalized part) and does not follow an eyeball correspondence with GNU R's pretty-printer, which is quite complex. This introduces harmless output mismatches. For instance, when printing vectors with GNU R, some are left padded, others are right padded. Other minor sources of mismatch are spaces and newlines. Therefore, the testing framework first processes the result strings of both interpreters into a common set of values that can meaningfully be compared.

This step is carried out by *result processors* (implemented in Python). There are two classes of processors; one that handles GNU R outputs, and one that handles CoqR outputs. Each processor matches the received raw string against a list of regular expressions (regexes). As soon as a regex matches, an object of type `Result` is created and the matching stops. There are nine subclasses of `Result`, corresponding to different result types, such as `ListResult` and `VectorResult`, the latter with further subclasses for boolean, numeric and string vector results.

The differences between the GNU R and CoqR processors lies in the regexes they use to recognize result strings, and hence in the particular instances of `Result` they produce. When creating result objects, processor include additional information that can be necessary for later comparison. For example, if the regex for numerical vector matches, the processor extracts the actual numerical value contained in the raw string and stores it in the `NumericVectorResult` object.

Testing outcomes. After processing, the testing framework obtains the result objects and uses their interface to compare them, returning the corresponding outcome of the comparison. The testing framework categorizes each step of computation in one of 8 categories, denoting different possible scenarios. The passed category corresponds to successfully executed test cases. The failed category is used when an actual mismatch in results is observed.

All other categories are there to help the development of CoqR by refining failure cases. The not implemented category gathers tests whose execution has reached the Coq constructor `result_not_implemented` inside CoqR (recall Figure 2). This therefore corresponds to a test execution reaching some as-yet-unimplemented function (or case) in the Coq code of CoqR. Additionally, the not found category corresponds to failures due to a test requiring an R function or object that is not defined in the CoqR environment.

The impossible category corresponds to the constructor `result_impossible`. As mentioned before, if CoqR were bug free, an impossible result would correspond to a critical bug in GNU R. All the impossible results we have observed so far were instead due to CoqR bugs.

Sometimes processing a result string fails. For instance, there exist special attributes that affect the way vectors are printed, and our pretty-printer is not aware of all of these special attributes. In such cases, the processor fails, and returns an `UnknownResult` object. Such tests are consequently marked as unknown; they could be functionally correct or not, so classifying them apart is practical. Also, once a single step in a line-by-line test file fails, or leads to a not-implemented result, we cannot say anything about the steps that follow in the test file: subsequent errors and success could both be coincidental. To categorize these tests, we use the potential pass/potential fail categories; while they mean nothing for correctness, they help in the development process by showing possible future results.

Guiding the development process. Having different categories of testing outcomes is crucial when facing a huge development, in order to apprehend the task at hand. In particular, we found extremely helpful to separate actual failures from internal uses of `result_not_implemented`, and from missing definitions (mostly functions).

Recall that a not-implemented feature generally corresponds to an entry of the symbol table that has not been translated into CoqR. It can also correspond to some branch in a large `switch` statement that we decided to postpone for some reason. On the other hand, a missing function definition corresponds to the use of an undefined R function in the test; this is typically because the function was not imported in the CoqR test code base. (Missing objects are often failed assignments due to previous errors.)

Therefore, in addition to counting the classified testing outcomes for the run of a test suite, our testing framework reports the 10 most common not-implemented features and the 10 most common missing function definitions. Both reports are useful to identify low-hanging fruits to make progress in running the selected test suite. For instance, at the initial stages of the project, the base library of R showed up prominently in the list of missing functions, as discussed below. Also, we used the report of most used not-implemented features iteratively in order to decide which features from the symbol table to implement first, leading us to the 128 features that are currently implemented.

3.2 Base Library

Unsurprisingly, our initial experiments running the test suites of GNU R and FastR raised a huge number of missing function errors, whose consolidated report allowed us to locate in the base library of R. As anecdote, having implemented 105 features from the symbol table, around 1,000 of 1,200

Table 1. Current results of running the different test suites.
(P = Pass, F = Fail, NI = Not Implemented, NF = Not Found,
I = Impossible, U = Unknown, PP = Potential Pass, PF = Potential Fail)

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

failures for the GNU R test suite were due to missing definitions. After extending CoqR to be able to load the base library, the total number of failures fell to 700, with only 300 due to missing definitions, corresponding to other basic libraries, most importantly the statistics library.

The base library consists of about 160 R source files, totalizing 19,000 lines of code, which are executed by GNU R prior to any other expression. This base library includes the definition of several functions and objects, such as `mean`, `matrix` or `pi`. Several of these functions are just wrappers over internal functions, adding argument checking or default arguments. For example, the `eval` function sets default values for the arguments of the internal `eval` function:

```
1 eval <- function(expr, envir = parent.frame(),
2     enclos = if(is.list(envir) || is.pairlist(envir))
3               parent.frame() else baseenv())
4     .Internal(eval(expr, envir, enclos))
```

Most of these definitions do not require many features to be implemented in the interpreter in order to be executed: only the `function` keyword and the assignment `<-` have to be implemented. However, there are cases where some computation is performed on the right-hand side of the assignment. For instance, the file `constants.R` of the base library contains the definition `pi <- 4 * atan (1)`. To evaluate this expression, the `atan` function has to be implemented.

Of course, being able to execute the definitions and include them in the CoqR environment does not mean that these definitions are usable: the internal function called by a library function may not yet be implemented in CoqR, so if a test executes that function, it will yield a not-implemented result. This is why it is crucial to be able to differentiate actual failures from not-implemented and not-found results. Note that several functions may end up relying on the same internal (not implemented) feature, giving more meaning to the report of the most common unimplemented features.

3.3 Current Status

Table 1 presents the current status of executing the four test suites described earlier with CoqR. On a 2-core Intel Xeon 2.20GHz with 4 GB of RAM, execution times range from 1.5 hrs for GNU R's test suite to over 12 hrs for FastR2. CoqR is several orders of magnitude slower than GNU R, but its aim is to provide a faithful formalization of R's semantics, not to be an alternative production-level interpreter.

Overall, CoqR successfully covers 30% of all the tests. The Corners test suite is almost completely covered, while FastR2, which we integrated more recently in our development process, is lagging behind compared to FastR1.

Unsurprisingly, 50% of the negative results are either not implemented or not found. This was to be expected considering in particular that we have currently implemented only 128 out of the 700 features of the symbol table (Section 2.7). We manually analyzed the reported failed results (5.7%) and observed that about 40% are due to mismatch of double numbers precision. We are currently looking for ways to eradicate such mismatch, and exploring the actual discrepancies. 10.7% of test cases end up classified as unknown, which subsequently lead to potential passes and fails (5%). Recall that these are due to mismatches with GNU R's pretty printer. However, the relatively low percentage of these cases does not seriously question the validity of the implementation. Fixing these unknown results is tiresome, but not conceptually challenging.

Finally, it is worth highlighting that only 1 impossible result remains in the current version. This corresponds to an actual bug in CoqR, which is currently under scrutiny. Overall, this report shows that CoqR is a realistic implementation (coverage-wise, not performance-wise) that, given more engineering power, can be grown to fully cover R and its libraries. The testing framework we developed provides important clues into where to direct implementation efforts.

4 Formal Reasoning about R

We now address one of the objective stated in the introduction: our operational semantics should be usable to build proofs about the R language. This section describes one use case: proving that invariants about the state of the memory are preserved during execution. We also discuss the associated need for proof automation.

As described in Section 2.3, each basic language element in R is associated one of 24 types, and each of these types are differently stored in memory. Figure 10 shows a snippet of the invariants defined by the `safe_SExp` inductive property (the full definition is 230 loc). The constructor `safe_ListStruct` captures the requirements on well-formed lists. The requirements are expressed using the predicate `may_have_types S l p`, which states that the pointer `p` is associated in the state `S` with an object whose type is in the list `l`. Hence, the hypothesis on `cdr` states that the tail of a list is either a list (`ListSxp`) or nil (`NilSxp`).[10] Similarly, the constructor states that the tag of a list is either a character vector or a `NilSxp`

[10] The type `NilSxp` is the type of the `R_NilValue` global variable (see Section 2.4); it is used both to end lists and to indicate a missing information.

```
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)
  (* ... *).
```

Figure 10. Typing invariants for memory

element. Note that no constraint is given for the first element `car` of the list: lists in R are heterogeneous. As another example, the constructor `safe_StrStruct` states that a string vector contains an array of C pointers, all of which associated with a character vector in memory.

Such invariants are relatively simple, but given the size of the formalization, they are quite long to establish. Proving that some step of the interpreter preserves the invariants quickly becomes very tedious. In order to address this, we developed various Coq tactics to ease the proof process. *Tactics* are programs manipulating Coq proof terms. They are an important component of interactive theorem provers, allowing a controlled form of automation. Our tactics are mainly useful to propagate known information through state changes. Overall, we defined 80 tactics, most of which perform simple operations on the proof context. We first proved about 200 simple lemmas about monadic binders and heap invariants. Our tactics look in the proof context for patterns where these lemmas can be fruitfully applied. Auxiliary tactics try to rewrite the context into specific forms that allow these lemmas to be applied.

Figure 11 illustrates the use of these tactics, with a lemma that states that the `do_attr` function defined in Figure 3b produces a result that satisfies the invariants. The `safe_state` predicate specifies that the given state only stores objects that satisfy the invariants, and similarly for the global variables of `globals` with the predicate `safe_globals`. The arguments `call`, `op`, `args`, and `env` of `do_attr` are also assumed to satisfy these invariants, as well as being of the expected type. In the conclusion of the lemma, the `result_prop` predicate specifies what the resulting state of `do_attr` satisfies in the interesting cases (ignoring cases such as `result_bottom`).

Thanks to our tactics, the proof closely follows the source code of `do_attr`. After introducing the hypotheses, the function `do_attr` is unfolded, leaving its definition ready to be processed by further tactics. The function `do_attr` starts by calling `R_length` (see Figure 3b). We here call the `cutR` tactic with a lemma about `R_length`. This novel tactic discharges the premises of the lemma and introduces the resulting state: only the success case is left, the other kinds of results being transparently propagated. The call to `R_length` is thus rewritten to a simple expression of the form `result_success S' n`, where `n` is the number returned by `R_length`. We then apply the `computeR` tactic. This is the most important tactic provided by our framework: it moves forwards in the current expression, propagating everything that can be propagated. In the example, it unfolds the `let%success` monadic binder, as this binder is now given a fully-computed result—thanks to the `cutR` tactic. It also updates all properties known to hold for the previous state `S` to the new state `S'` using the results of the `R_length_result` lemma. For instance, the hypothesis `safe_pointer S args` is replaced by `safe_pointer S' args`. This transition would not be particularly difficult to prove by hand given the right lemma, but it would be cumbersome and repetitive. The proof continues by applying the tactic `cutR` again with a lemma about `matchArgs`, then `computeR` to propagate the hypotheses. And so on.

```
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.
```

Figure 11. Example of specification and tactic usage.

We have proven that the invariants are safely propagated throughout the execution of 20 simple functions. Some of these significantly change the structures described by the invariants, but eventually preserve the invariants. In Coq, this translates into a large number of intermediate states being generated, and some invariants proofs are not automatically discharged by the `computeR` tactic, as some invariants do not hold in them. The proofs about such functions sometimes involve proving from scratch that the invariants hold. This typically involve following all possible paths from the newly-allocated objects and show that the invariants hold for each of them; this is a rather complex proof to build.

The proofs of our 20 simple functions spans over 440 lines of proofs. This is to be compared to the size of the definition of the automation: 1,900 lines of tactic definition, and more than a thousand lines for the lemmas used by these tactics. The compiled file for the proofs about our 20 simple functions is however larger than the one containing all the lemmas used by the tactics: automation enabled us to build large proofs through relatively small proof scripts. However,

in the long run, in order for formal reasoning about R to scale, it will be necessary to develop a program logic on top of CoqR. This is a major undertaking, towards which CoqR is an essential first step.

5 Related Work

R is a notably difficult programming language [3], whose semantics is constantly moving—see for instance the recent addition of R's alternative representation [32]. In our formalization, we chose to ignore these fast moving parts, but these parts are used by real-world R programs. Furthermore, the diversity of R users is such that different R programs will use widly different libraries and features, as the generally accepted guidelines [10] do not restrain users about them. This makes it difficult to build tools. Consequently, relatively few tools for R exist in comparison to the size of its community.

In particular, there exist few testing frameworks in R. The testR project [21, 33], which later evolved into the Genthat library [9], aims at generating unit tests for R functions. It starts from a program using the functions to be tested. It then annotates and executes this program, storing the trace of the calls to the functions to be tested. Unit tests are then generated from this trace. These tests can be used to ensure that further versions of a library do not break existing code.

GNU R is not the only R interpreter that exists. Many existing interpreters are based on the same C nucleus code, but use different libraries for linear algebra, usually optimized for a specific usage. The FastR project [14] takes a different approach as it also reimplements the nucleus of the R interpreter, using the Truffle self-optimizing framework [35]. FastR is faster than the reference interpreter not only in the linear-algebraic part, but also in the language interpretation layer. FastR agrees with us that the GNU R interpreter is the reference semantics: any behavioral difference between FastR and GNU R is considered a FastR bug.

Work on formal specification of R is sparse. The language definition document [27] is unfit for verification due to its ambiguities and mismatches with the behavior of GNU R. In an effort to understand R features and their usage in the wild, Morandat *et. al.* formalize a minimal core of R [23]. This core models functions and some assignment forms, but no control-flow constructs (such as `if` or `return`); neither do they attempt to capture the semantics of additional features from the symbol table. To the best of our knowledge, our work is also the first mechanized specification of R.

However, the general goal of formalizing full real-world languages—as opposed to small subsets—is not new. JavaScript is a particularly relevant example. Empirical analyses have indeed confirmed that the language features that are usually ignored in formalized subsets of JavaScript are actually important for actual web developers [30]. In the case of JavaScript, there are several trust sources. First, the language is precisely specified by the ECMAScript specification [6]. Second, there exist various test suites [7, 24] as well as several widely used interpreters. Consequently, various formal specifications of JavaScript exist, each related to some of its trust sources.

The first full formal semantics of JavaScript [18] has been related to the ECMAScript specification. It had a major influence on the definitions of further JavaScript formal specifications [1, 2, 8, 31] and also served as the formal basis to prove the soundness of security-related JavaScript subsets [19, 20]. This work was however not mechanized.

In parallel, several formal semantics for JavaScript are based on a JavaScript interpreter [11, 25, 26]. These semantics are related to JavaScript test suites, either by comparing the results with the expected result, or by comparing results with widely-used JavaScript interpreters. These formalizations tend to be easier to build as testing frameworks already exist. Furthermore, they are usually easier to understand by non-specialists. However, such formalizations suffer from all the issues of test suites: for instance, in JavaScript the `for-in` feature was then loosely tested, and its behavior varied from interpreters to interpreters.

The JSCert formalization [2] is an interesting step forward as it was designed to be related with both the ECMAScript specification and the JavaScript test suites. The formalization is composed of two parts: a mechanized specification and an interpreter. The JSCert specification is syntactically related with the ECMAScript specification through an eyeball correspondence, and the interpreter passes its test suite. The specification and the interpreter are related to each other by a Coq proof. This double-relation provides a large amount of trust to JSCert. These relations served to find issues in both the JSCert specification and in other interpreters, as well as mistakes in the ECMAScript specification. In CoqR, we take a different approach by defining the big-step operational semantics as an interpreter: the same definition is both executable and syntactically close to its specification (the GNU R interpreter). CoqR nevertheless shares several design ideas with JSCert. JSCert served as a basis for the JSExplain tool [4], aiming to explain JavaScript to non-specialists. As CoqR uses a similar monadic structure than JSCert, a similar tool for R is an interesting future work.

The Coq proof assistant has already been used in a variety of mechanized language formalization projects. The most well-known is the CompCert project [16], a verified optimizing compiler for C. This compiler is proven to be free of compilation bugs, leading to safer programs in critical software. This project comes with a formalization of the C programming language, as well as its intermediate compilation languages. Due to the compilation nature of the CompCert project, it was acceptable to restrict the behaviors of the C programming language in their formalization, restricting it to the behaviors that will actually be compiled by CompCert.

The Formalin project [15] is another formalization of the C language. It aims at precisely listing all the possible behaviors of C programs. We could have reused these formalizations of C to build a deep-embedding semantics, by defining R to be the interpretation of GNU R's source code by one of these semantics. Such a semantics would however be of little practical use, as deep embeddings are notoriously hard to reason about in proof assistants. We instead define CoqR as a shallow embedding, suitable for building proofs about R and R programs, as shown in Section 4.

6 Conclusions

We present the first formal specification of R, written as an interpreter in Coq. We present the monadic encoding that allows CoqR to be in eyeball correspondence with the GNU R reference interpreter, provide a testing infrastructure useful to incrementally enrich CoqR, and show that CoqR can be used to formally reason about R. Considering the number of realistic test cases that CoqR currently passes, and its extensible architecture, we believe that CoqR can be grown to fully cover R and its main libraries. Among the different venues for future work, it would be particularly interesting to reduce the shim further, by moving the parser and the pretty-printer to Coq. On the formal reasoning side, CoqR should be the basis on top of which to develop an appropriate program logic, to more conveniently reason about R programs. As such, it is a necessary first step towards a robust environment for formal verification of R programs.

References

[1] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2013. Language-Based Defenses Against Untrusted Browser Origins. In *USENIX Security Symposium*.

[2] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudinien, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *POPL*.

[3] Patrick Burns. 2011. *The R Inferno*.

[4] Arthur Charguéraud, Alan Schmitt, and Thomas Wood. 2018. JSExplain: A Double Debugger for JavaScript. In *The Web Conference*.

[5] The Coq development team. 1984. the Coq Proof Assistant. https://coq.inria.fr/.

[6] ECMA International, editor. ECMAScript language specification. Standard ECMA-262. https://tc39.github.io/ecma262/.

[7] ECMA International. 2010. Test262. https://github.com/tc39/test262.

[8] Philippa Gardner, Sergio Maffeis, and Gareth Smith. 2012. Towards a Program Logic for JavaScript. In *POPL*.

[9] Filippo Ghibellini. 2017. *Dynamic test generation for R packages*. Bachelor's Thesis.

[10] Google. [n. d.] R Style Guide. Retrieved 2018 from https://google.github.io/styleguide/Rguide.xml.

[11] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of JavaScript. In *ECOOP*.

[12] Ross Ihaka and Robert Gentleman. 1996. R: a Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*.

[13] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 2012. Validating LR(1) Parsers. In *ESOP*.

[14] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Virtual Execution Environments*.

[15] Robbert Krebbers and Freek Wiedijk. 2011. A Formalization of the C99 Standard in HOL, Isabelle and Coq. In *Calculemus/MKM*.

[16] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Communications of the ACM*.

[17] Xavier Leroy. 2014. How much is a mechanized proof worth, certification-wise? In *Principles in Practice*.

[18] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2008. An Operational Semantics for JavaScript. In *APLAS*.

[19] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2009. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS*.

[20] Sergio Maffeis, John C. Mitchell, and Ankur Taly. 2010. Object Capabilities and Isolation of Untrusted Web Applications. In *SP*. IEEE.

[21] Petr Maj, Tomas Kalibera, and Jan Vitek. 2013. TestR: R Language Test Driven Specification. In *The R User Conference, UseR!*

[22] Jonathan McPherson. 2014. Debugging in R. In *The R User Conference, UseR!*

[23] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the design of the R language. In *ECOOP*.

[24] Mozilla. 2013. Mozilla Automated JavaScript Tests. https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests.

[25] Daejun Park, Andrei Stefnescu, and Grigore Rou. 2015. KJS: A Complete Formal Semantics of JavaScript. In *PLDI*.

[26] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and `eval` in JavaScript. *DLS*.

[27] R Core Team. 2000. R Language Definition. *R Foundation for Statistical Computing*.

[28] R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. https://www.R-project.org/.

[29] R Core Team. [n. d.] The Comprehensive R Archive Network. Retrieved 2018 from https://cran.r-project.org/.

[30] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The `eval` that Men Do. A large-scale study of the use of `eval` in javascript applications. In *ECOOP*.

[31] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In *SP*.

[32] Luke Tierney, Gabe Becker, and Tomas Kalibera. 2017. ALTREP and Other Things. In *R-devel*.

[33] Roman Tsegelskyi and Jan Vitek. 2014. TestR: Generating Unit Tests for R Internals. In *The R User Conference, UseR!*

[34] Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science*.

[35] Thomas Wuerthinger. 2012. Truffle: A Self-Optimizing Runtime System.

[36] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*.