A Coq Formalisation of a Core of R

Martin Bodin
mbodin@dim.uchile.cl
Center of Mathematical Modeling
Santiago, Chile

Abstract

Real-world programming languages have subtle behaviours. In particular, their semantics often contains various corner cases that programmers are unaware of, which can yield serious bugs. This is an opportunity for our community as Coq enables to certify program behaviours. Such certifications rely on a formal semantics, which can sometimes be as difficult to trust as the original program. Previous work proposed ways to trust such large semantics. This work evaluates the feasibility of such a formalisation in the case of the R programming language—a trending programming language specialised in statistics. We introduce a formalisation of the core of R related to the reference R interpreter.

1 Introduction

R [R C15; IG96; Cor] is a trending programming language for statisticians. It is used in a large range of fields (biology, finance, etc.), as illustrated by the list of available packages categories: https://cran.r-project.org/web/views/. This makes R unlikely to disappear in the following years. This diversity is reflected among R programmers, resulting in largely different programming styles.

The R programming language presents itself as an expressive and powerful language, able to express complex notions in few keystrokes. This sometimes come with the cost of readability. The semantics of R is subtle and contains numerous corner cases, as shown in Section 2. As a consequence, bugs can be frequent in R programs (as illustrated by the existence of various debugging tools [McP14]), and trusting such programs can be difficult. Formal methods in general and the Coq proof assistant in particular offer an interesting answer to this trust issue. But to formally prove that an R program meets its specification, we need a semantics of R.

A semantics for the full R language will inevitably be complex, as all corner cases need to be covered. To be able to trust such a formalisation, it is important to identify *trust sources*. These can be language specifications, test suites, or reference interpreters. It can be difficult to convince non-Coq-users that they can trust our formalisation [Ler14]: the most time-consuming part of the formalisation process is to relate the formalisation with its trust sources, not actually defining the formal model itself.

In the case of R, there is unfortunately no precise language specification (there is an ongoing effort to write one, but it is too early to base ourself on it). There are however test suites [TV14; MKV13], as well as a reference interpreter [Cor] against which alternative implementations of R are tested. To trust our formalisation, we wrote it in a similar way than the source code of the reference interpreter. We also made it executable to be able to run it on test suites. We believe that these two relating methods are enough to convince anyone that we closely captured all corner cases of R.

The R language is large, but it contains a core which is both small and easy to identify. We introduce a full Coq specification for this core. The formalisation presents itself as a monadic interpreter. This interpreter mimics the operations of the reference interpreter, using its C source code as a specification. Section 2 presents some subtleties of the R programming language. Then Section 3 presents how our formalisation is defined, and in particular, how it is related to the R source code. The source of this project is available at https://github.com/Mbodin/proveR/tree/CoqPL2018Final.

2 Presentation of R and its Core

R is a weakly typed programming language. Its basic data types mainly consist of various kinds of arrays. It was originally [IG96] defined as a mutation of Scheme designed to look like its predecessor, the S language. As a consequence, functions are first class in R. Furthermore, R follows the code-is-data paradigm: it is possible to delay the evaluation of statements, and even to manipulate their inner structure (for instance using the `substitute` R function).

Almost all constructs of R are treated as functions. This includes features like `if`, `while`, and assignments. R source code contains a table, the *symbol table*, associating each construct to a C function. Some of these functions (`if`, assignments, etc.) are special and directly manipulate the abstract syntax tree of their arguments. In this work, we consider that the features of this table are *not* part of the core of R. In other words, we define the core of R as the parts of R needed to access this table and execute its C functions.

This definition of R's core enables us to focus our formalisation effort on a very restricted part of the language. It includes the execution rules for function calls, environments, closures, promises (delayed evaluation), as well as the parts initialising the symbol table. Constructs like `if` and `while` are not part of the core, but (some kinds of) assignments are, as they are used for function calls. This core is easily extendable: we can add any other feature by implementing its associated function and adding it to the symbol table.

We now present some corner cases of the R semantics. Most R constructs try, in some ways, to "guess" what the programmer meant. The same syntactic construct is thus often associated with different behaviours. For instance, array look-up is written `a[i]`: if `i` evaluates to an array of indices (numbers are considered to be arrays of size 1), then `a[i]` is the array with the corresponding elements of `a`. But if `i` evaluates to an array of negative integers, then the array `a` is copied, and all elements whose (opposite) index is in the array `i` are removed. Other behaviours happen when `i` evaluates to a boolean array or a string array. As R is untyped, a function like `function (a, i) a[i]` is already complex to specify.

Another source of subtleties of R is that it lazily evaluates expressions, including those with side effects. For instance, the following function `f` only evaluates its second argument if the first is 1. Thus, in its second call, the variable `b` has confusingly not been defined because the assignment `b <- 1` has not been evaluated.

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

R contains numerous semantic exceptions. None are inherently complex to deal with, but their quantity complexifies the language.

```
EXP* applyClosure (EXP* op, EXP* arglist, EXP* rho){
  EXP* formals, actuals, savedrho, newrho, res;
  if (rho->type != ENVSXP)
    error ("'rho' must be an environment.");
  formals = op->clo.formals;
  savedrho = op->clo.env;
  PROTECT (actuals = matchArgs (formals, arglist));
  /* ... */
  return res;
}
```

Figure 1. The original C function

3 Formalisation

Our formal semantics is presented as a monadic interpreter in Coq. It is related to the C source code of the R reference interpreter by a line-to-line correspondence: every one or two lines of our interpreter correspond to one or two lines of the reference interpreter. Our semantics is runnable, which enables us to test it against R test suites. However, as the core of R is very limited, non-core features have to be added for testing. These features can be added following the same methodology than the one presented here. Figure 2 shows our Coq translation of the C function shown in Figure 1.

We used monads very similar to the ones of JSRef [Bod+14]: a `result` is either a success—it then carries the actual result and the resulting state `S` of the program—, an error, or `result_bottom`—meaning that the interpreter ran out of fuel during the execution. These monads enable us to abstract most of the aspects of imperativity and memory handling from C. Examples of monads include `let%success`, to extract the result given by a function (it corresponds to the monadic *bind* operator), or `read%defined`, to read the content of a pointer. Coq notations have heavily been used for the line-to-line correspondence. For instance, let us consider the line `if (rho->type != ENVSXP)` of Figure 1. It has been translated into two lines: the monad `read%defined` first dereferences the pointer `rho` (which may fail if the pointer is unbound), then the test is performed. Every line has been similarly translated.

We chose to ignore some aspects of the original C source code during the formalisation. For instance, the support for various locales and character encodings: our interpreter only considers ASCII strings. More importantly, the garbage collector of the reference interpreter has not been translated. We indeed consider that, as it is assumed not to change the final result, its formalisation is not needed to prove the functional correctness of R programs. For instance, the `PROTECT` macro from Figure 1 temporarily disables the garbage collector: it has thus been formalised out in Figure 2.

Instead of defining such an interpreter, we could have used already existing Coq semantics for C [KW11; Ler09] to directly define a semantics for R. We however believe that the abstraction layers of C would come with a non-trivial additional complexity to the formalisation, making it more difficult to use. Furthermore, the R reference interpreter has been built with a strong intuition in mind [IG96]. This intuition is reflected in R internals, which would have been hidden if we attempted such a direct approach. As a future work, it would be a very valuable contribution to use these semantics of C to relate our formal model to the source code of R. We furthermore believe that our line-to-line correspondence would greatly help such a proof.

```
Definition applyClosure S (op arglist rho : EXP_pointer)
    : result EXP_pointer :=
  read%defined rho_ := rho using S in
  ifb type rho_ <> EnvSxp then
    result_error S "'rho' must be an environment."
  else
    read%clo op_clo := op using S in
    let formals := clo_formals op_clo in
    let savedrho := clo_env op_clo in
    let%success actuals :=
      matchArgs S formals arglist using S in
    (* ... *)
    result_success S res.
```

Figure 2. The Coq translation

4 Conclusion and Future Work

In this work, we presented a Coq monadic interpreter mimicking the behaviours of the R reference interpreter. We believe that our approach provides a high level of trust for our formal semantics, on which new projects about R can be based—for instance proving that a given R program meets its specification. Our formalisation is easily extendable through the symbol table, and we aim to continue the formalisation process to cover new features of R.

We think that the usability of this work can be improved: the current formalisation is so close to the C source code that implementation details still appear in it. We plan to build a more abstract formalisation, more suitable to prove the functional correctness of real-world R programs. In particular, we believe that most pointers can be formalised out, the semantics of R being mostly functional.

References

[Bod+14] Martin Bodin et al. "A Trusted Mechanised JavaScript Specification". In: *POPL*. 2014.

[Cor] R Core Team. *The Comprehensive R Archive Network*. URL: https://cran.r-project.org/ (visited on 2017).

[IG96] Ross Ihaka and Robert Gentleman. "R: a Language for Data Analysis and Graphics". In: *Journal of Computational and Graphical Statistics* (1996).

[KW11] Robbert Krebbers and Freek Wiedijk. "A Formalization of the C99 Standard in HOL, Isabelle and Coq". In: *Calculemus/MKM*. 2011.

[Ler09] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communications of the ACM* (2009).

[Ler14] Xavier Leroy. "How much is a mechanized proof worth, certification-wise?" In: *Principles in Practice*. 2014.

[McP14] Jonathan McPherson. "Debugging in R". In: *The R User Conference, UseR!* 2014.

[MKV13] Petr Maj, Tomas Kalibera, and Jan Vitek. "TestR: R Language Test Driven Specification". In: *The R User Conference, UseR!* 2013.

[R C15] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2015. URL: https://www.R-project.org/.

[TV14] Roman Tsegelskyi and Jan Vitek. "TestR: Generating Unit Tests for R internals". In: *The R User Conference, UseR!* 2014.