We thank the reviewers for the constructive feedback and the opportunity to enhance our submission. We address each review in detail below. The submitted PDF uses blue-colored text for all the changes (except in the code), in order to ease the evaluation of the improvements. We also use blue-colored text below to differentiate our responses from the original reviews.

REVIEW 1

For example, the authors could modify
the paper to more focus on their monadic implementation and how they
model the heap.

We add details about both aspects, see below.

The description of section 2.1 is quite sketchy and
lots of details are not mentioned.
For example, how is result_longjump
implemented and used? A monadic binder for setjump should be also
presented in detail.

We have expanded the description of 2.1 and added some information about setjump, though limited in details due to the little space available overall to integrate all the comments.
(note that as suggested by reviewer 2 we removed the UI screenshots, freeing up space for additional explanations in various places)

Section 2.3. Is this modeling of the heap unique against previous
related work? Is there any other work to model the heap in Coq?

There are other work modeling heaps in Coq (CompCert for C, JSCert for JS, etc.). However, we model a specific heap, specialised for the structures used in the GNU R interpreter. Typically, we consider that there are only pointers to EXPRECs (that is, SEXPs). This guides a lot the formalization, as the objects pointed by pointers are (1) well-defined (contrary to C where a pointer could point in the middle of a structure, for instance, or dangling), (2) have the SEXP/SEXPREC shape. This SEXPREC shape comes in various forms, which our modelling takes into account.
We have clarified in 2.3.

Section 2.4. The static variable do_attr_formals should be included
in Figure 3a. That would help the readers.

Indeed, done.

Section 3. Testing outcomes. In Figure 10, what are differences
between "not implemented" and "not found"?

Figure 10 is now removed, but the explanation of the categories is in the “Testing outcomes” named paragraph (2nd sub-paragraph).

“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.”

In other words, “not implemented” means that we are missing a function/case in the Coq code, while “not found” means that we are missing a function/object in the R code.

We have expanded the description in the “Testing outcomes” paragraph. The distinction is further emphasized in the “Guiding the development process” paragraph.

Please clearly distinguish a functionality and a function/an object. I suppose that a functionality is not implemented because some functions are not provided. So they are the same. Am I wrong?

Indeed, the use of functionality vs. function was quite unfortunate. We have decided to remove all occurences of the word “functionality” in the paper to make things clearer. We are left with two kinds of functions: the ones defined in C or Coq (depending on whether we are talking of GNU R or CoqR), and the ones defined in R, either in the base library or in other libraries. If a function is not (yet) implemented in Coq, we use the special “result_not_implemented” constructor in place of its source code. When executing this expression, the execution is aborted and the “not-implemented” result is returned. If an R function is not present in the CoqR environment, we throw a “not-found” error.

Section 4. It would be better for the readers who are not familiar to Coq (like me) if more details of cutR and computeR tactics are presented.

We added some details about tactics in section 4, although again due to space we focus on the big picture and leave details to the source code. There is nothing specifically remarkable in the definition of the tactics.

Section 5. Line 1156. andalso -> and also

Fixed.

REVIEW 2

- Things like the UI of the testing framework are of little interest.

We agree with the reviewer. Therefore, to free some space in order to accommodate other requests we have removed the screenshots. We still discuss the benefits of the interactive system on the development.

- Since the interpreter written in Coq does not not derive from a concise specification, it does not allow reasoning about the correctness of the specification, but it is more useful to find bugs in (and verify assumptions about) the implementation itself.

[Note: it is not clear what the reviewer means by “the correctness of the specification”, since the only thing one can do with formal methods is to verify the correctness of an implementation *with respect to a specification* (or validate the compatibility of different specifications). The specification is the baseline against which one proves things.]

Here, the goal is not to test GNU R, since GNU R is in fact taken to be *the* specification of R, but rather to provide a Coq version of this specification. Being written in Coq, this specification is useful to reason about meta-properties (e.g. the preservation of heap invariants mentioned in the paper) and to reason about R programs. As we mention in the perspectives, however, for reasoning about R programs at scale, it is best to build a program logic *on top of* CoqR, proven sound with respect to the CoqR semantics.

Additionally, if the R community were to release a complete formal semantics of R, then we could work on establishing the link between CoqR and that semantics, similarly to how JSCert was carried out.

There have, however, been attempts to formalize the language more directly (e.g., in Morandat et al., "Evaluating the Design of the R Language"), which deserve mention.

We apologize for this oversight. We have added a brief discussion of this work in the related work section. In essence, the Core R language formalized by Morandat et al is very minimalistic, while CoqR aims at a much larger subset. It is also not mechanized, nor used to establish any formal result. Also, a major contribution of Morandat et al is to study how R features are used in the wild, which is orthogonal to our work.

- Figure 11 is so small that it's hard to read.

We have removed the UI screenshots, since they indeed contribute little to the discussion, and are available online anyway. The freed up space has been used to accommodate suggested changes and additions.

- Not handling the GC leaves out one of the biggest sources of hard-to-find bugs in the R world, which was already the target of sophisticated static analysis (e.g., https://github.com/kalibera/rchk).

As mentioned above, our goal here is not to test GNU R, but to provide a trustworthy Coq specification, which subsequently enables formal reasoning about R and R programs.
A GC 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).
We have updated the limitations section to make this clearer.

- It's not clear how the CoqR approach can ever work with code that uses the R native interface. Every non-trivial use of R involves packages that make heavy (and sometimes questionable) use of this API. Would the packages also need to be converted to Coq?

No, these packages would not need to be converted to Coq.
The API presented in R-exts.pdf is just a (long) list of C functions made public to interface with external programs. We have already translated some of these functions (all the nucleus ones, typically) to Coq. We furthermore made sure that the Coq translations are as close as they can be to the original C functions. If a package makes use of this API (to make it interact with parts written in C or FORTRAN, for instance), then it can be linked with the OCaml extracted code of CoqR (the

implemented API functions have already been extracted to OCaml, as any other Coq functions). There already exist OCaml interfaces to perform this task (see e.g. https://caml.inria.fr/pub/docs/manual-ocaml/intfc.html). Such a link will probably require some engineering work to be implemented, but there will be no need to convert external code to Coq.

- The test results should be explained in more detail - including the exact criteria of how tests were selected from GNU R, FastR1 and FastR2.

The main criteria to filter out tests was to prioritize the development effort. In the paper, we have reformulated and added a reference to the Github repository for the details.
For information, the detailed criteria is:

1. Tests generating PDFs
2. Tests requiring additional packages and libraries (PCRE, JIT, MASS, etc)
3. Tests on different encodings
4. Tests requiring Internet connection
5. Tests generating plots in graphical interfaces
6. Windows-only tests
7. Tests related to S4 classes
8. IO tests (Files, sockets)
9. Tests related to limiting CPU time for computations
10. Generated tests

- One could argue that R also specifies the exact formatting of the console output (this is certainly the case for vectors), so it's unfortunate that this is part of the shim.

Indeed, although this part of GNU R is extremely complex: it depends on the value (if present) of various fields (“dim”, “names”, etc.), on the way IEEE floating point numbers are printed (they are stored in binary but printed in decimal: there are thus some non-trivial operations on the way). It also depends on the value of global variables (like `getOption("digits")`) which can be changed dynamically, or by the shell environment. We believe that this part is not the most critical part of R, as soon as we can faithfully print some base values (such as numbers and booleans).
We nevertheless recognize that moving pretty printing from the shim to CoqR would be a valuable future work.
We now explicitly mention this among the perspectives in the conclusion.

- It would be good to expand some more on why it's not possible to annotate C code with the necessary information (as opposed to modifying Coq to look like C).

Annotating the original C code would be possible and would lead to a formalization of the language called a deep embedding (as opposed to the shallow embedding we adopt). We agree with the reviewer that a deep embedding would have required far less work to define than our shallow embedding, but it would have hindered the main objective of CoqR: providing a usable foundation to reason about R and R programs in a proof assistant. The reason is that deep embeddings are notoriously hard to reason with in proof assistants. This is why CompCert and Formalin (for C) or λJS and JSCert (for JavaScript) provide large hand-made semantics instead of just basing themselves on a largely used compiler or interpreter.
We now discuss this briefly at the end of the related work section.

REVIEW 3

First, the paper heavily relies on the notion of "trust" towards a program (or language implementation for that matter). Indeed the word itself is mentioned more than twenty time, however, the authors' understanding of the notion itself is not explicitly given, beyond the statement "proof assistants such as Coq [5] enable us to formally prove program properties with a high amount of trust". For a work that relies on this notion so fundamentally, it should state very clearly and definitely what its understanding of that notion is. Giving this early on would significantly help the reader.

We agree with the reviewer, and in fact our use of the word “trust” in the introduction was multiply overloaded, unfortunately adding to the confusion around this loosely-defined term. The main body of the paper uses trust in the context of relating a formal model to its real-world counterpart (i.e. CoqR behaves the same as GNU R, so that reasoning over CoqR applies to GNU R). To avoid confusion, we have rephrased the introduction, limiting the use of “trust” to this particular meaning.

Second, the paper identifies areas of the GNU R implementation that needed special attention when "translated" to Coq, and does so in a way that makes them seem special or otherwise surprising. Examples include the first paragraphs of 2.3 "Modeling the Heap", 2.4 "Dealing with Global Variables", the third paragraph of that section 2.4, etc. These describe fairly common practice for implementation of interpreters for dynamic languages: regarding 2.3, this coding is not unlike those found in LISPs, some Schemes, or even Smalltalk implementations, the initialization at the beginning of 2.4 could probably be found in this or similar ways in lots of interpreters and runtimes that have to built up an initial world view, and the lazy global initialization (later in 2.4) would probably not be called subtle by a typical VM engineer. Other such examples can be found in 2.7 with regard to the symbol table, or the characterization of lists as being "heterogeneous" (which is somewhat pointless in dynamic languages, to the contrary, homogeneous lists would be more surprising), among others.

We agree with the reviewer that our description of these features was not factual enough, and gave the feeling that these features were somewhat surprising. We rephrased these parts in the paper (mainly the introduction and Section 2) to make them more neutral. We also added a sentence at the beginning of Section 2 to appeal to the fact that GNU R’s structure is fairly standard and therefore the principles of our approach could be applicable to other contexts.

Regardless of whether the authors intended to convey that the R implementation is special in some way, this Reviewers take-away is that:

(a) the authors could concentrate their understanding of the GNU R implementation structurally before explaining their implementation, thereby helping readers to see similarities to "their" interpreters/implementation, and maybe

(b) extract the techniques found here for R (heap modeling, function calling etc) in such a way that it is reusable for formalization of other dynamic languages, too.

While the latter probably amounts to a different paper, the former maybe be worth a though at least. Also, both would increase the overall relevance to DLS.

We did consider a rewriting of section 2 that would first put the info about GNU R, and then expose CoqR, but it turned out less effective in our opinion. The current exposition favors a topic-by-topic decomposition, with the GNU R/CoqR correspondence exposed progressively.

We agree with the reviewer that it would be interesting to integrate lessons learnt from different Coq models of dynamic languages (such as those for JavaScript and R) but, indeed, it would be a different paper. This paper is really about the specificities of R, though we hope the presentation of the different aspects holds interesting insights for someone who would be interested in developing a similar interpreter for another dynamic language.

Third, a lot of R programs rely on functionality _not_ part of the original interpreter. Numerous libraries are written in R itself, but rely on data from the internet, or are written in other languages, such as C, Java, or Fortran. These make up large parts of common R programs. This combined with restricting the tests to somewhat isolated ones (i.e., without access to files, the internet, or generating graphs) makes the overall usefulness for the end user a little questionable. While it is indeed understandable why to restrict the test set from an development process point of view, the overall goal of trustworthiness is at least slightly reduced. The authors could help that by giving pointers on how to increase trust in _these_ parts of R programs.

Please see the response to Reviewer 2 above about the native interface. Programs using this API can be linked to the OCaml extracted code, as usual. One could also, in theory, prove properties about such programs given a semantics of the external part (which could be a simple axiomatized semantics).

Fourth, the C source code given does not actually match that of GNU R [...]

We agree. To simplify the presentation, we had replaced every occurence of “SEXP” in C by “EXP*” and every occurence of “SEXP” in Coq by “EXP_pointer”. This choice was meant to simplify the presentation: our Coq code actually uses names that follow the original C code.
Based on the reviewer’s observation, we reverted back to our actual Coq code, which uses the real C names.

Minor remarks:

- The authors should consider explaining the meaning of "monadic encoding", which may not be immediately understood in the DLS community.

We have expanded the explanation of monads as a technique for representing effects in a pure functional setting. We realize that this might be insufficient for some readers, but an introduction to monadic programming is outside the scope of this paper, especially considering the space constraints. The canonical reference (Wadler’s 1992 paper) is a brilliant and accessible exposition of the technique.

- Footnote 1 states that "..." is omitted for simplification, however, several arguments in the paper are based on the complexity of R, so maybe this construct should not be omitted.

We now explain “...” as well.

- The division between "core" and "additional features" based on the symbol table is comprehensible but might be confusing for interpreter developers. In fact, the sum of what the paper calls "core" and "additional features" would typically be called the core of a language. Maybe another term than core would help (nucleus?)

We have changed the term “core” to “nucleus” as suggested. Note however that Morandat et al. (pointed out by Reviewer 2) use “Core R” to denote an even smaller subset of R. We found that adopting “nucleus” additionally helps us avoid introducing potential confusion with Morandat et al’s Core R.

- The allusion to Linus' law is probably not apt in a paper that deals with trust.

Trust in this work is defined by our ability to claim that GNU R and CoqR behave the same. This trust can be measured by the “trust sources”, which in our case are two: the paired testing with GNU R, and the eyeball correspondence. We believe that the eyeball correspondence is essentially (a variant of) Linus’s law, which states that “given enough eyeballs, all bugs are shallow”.

- footnote 9 gives the source for R's test suite by some github mirror of R's source. While it seems usable, giving the original R's repository would be better.

Fixed (we keep the reference to GNU R (/cran) only, which is indeed enough).

- The list of the 9 Result subclasses in 3.1 is maybe a bit extensive, as is maybe the first paragraph of "testing outcome".

Fixed.

- Testing outcomes mentions 7 categories but Figure 10 and Figure 12 list 8.

Fixed.

- Figure 10 and figure 11 are illegible when printed.

The figures have been removed.

- The overall number of "features" (700) is only mentioned on page 9, but should be mentioned earlier alongside with the description of the symbol table.

It was in fact mentioned in 2.7 alongside the description of the table, but we have moved the sentence further up.

- 3.3 current status says the test suite takes eight times the time GNU R takes, which is reasonable for the projects status. However, to be a "realistic implementation" as stated later, performance is very important given the use cases of R. It should be hence stated more clearly what is meant by realistic.

Indeed. The realistic aspect of CoqR only refers to its coverage, not its performance.
We clarified in 3.3.

- Page 10, around line 1093: lemmae -> either lemmata or lemmas

Fixed.