Reviewer #1:

The paper is relatively easy to read, despite some parts that may be heavy for those not familiar with Coq. In general, it's only critical issue is that the "grand finale" is missing: the description of the interpreter itself. On page 32 the paper gives the specification of 'interp', but there are no clues about how it is implemented.

****
Response: In the paper's first version, we avoid giving details about the interpreter's implementation mainly because it is built using tactics (a common pattern when programming with dependent types in Coq) which would difficult the high-level comprehension of our formalization.

We have improved the explanation of our semantics interpreter by showing some parts of its tactic definition.
****

The paper later does some benchmarks over this implementation, which is weird to read without knowing what it is benchmarking. The paper says the implementation is based on a virtual machine, but how is this VM? What instructions does it have, what is its internal state, etc.? Does it do backtracking? When and how?

****
Response: In the introduction, we mention that our view of a VM is the one commonly used by machines for lambda-calculi, like the SECD and Krivine machines. We inspire our semantics for RE parsing on such machines for lambda-calculus.

Essentially, our VM sees a RE as a program executed over the input string, and it backtracks as expected by a greedy RE parser strategy. As suggested by Reviewer #3, we changed our semantics definition in order to produce bit-coded parse trees following the greedy disambiguation strategy.
****

As a minor issue, the paper sometimes gives too much details for some parts of the code that seems mostly bureaucratic. (The removal of these parts will give lots of space for the missing description of the VM.) Some examples:

- Function 'in_regex_p'. As the paper explains, this is a translation to Coq of the rules in Figure 5. It adds nothing to the reader.

- Function 'is_code_of': Another mostly "bureaucratic" function. It is important to have its signature and a brief description, but the code itself adds little to the text.

- Function 'is_tree_of': The same.

****
Response: We try to limit Coq code presentations to things that the reader could readily understand without a deeper comprehension of how to formalize stuff using Coq proof assistant. This was the main motivation for showing all these simple definitions.
****

Follows some other details about the text:

- The parse tree in Example 1 has some "poetic license" against the given grammar regarding lists. (Lists are made of several nodes, each with two children, ending with an end-of-list node.) This is the cause of some bits in bit code given later (on Line 265) not appearing in the tree.

****
Response: We have improved our figure to include all bit separators. Thanks!
****

- Function 'code', ~line 250: it would be useful to give its type. (In general, types are more important than bodies.)

****
Response: Ok, it is fixed. Thanks!
*****

- Line 269: "Remaining three digits" -> "The remaining three digits"

****
Response: Ok, it is fixed. Thanks!
*****

- Line 314: "where 'e' is current RE" -> "where 'e' is the current RE" (In general, there are several places where the article "the" seems to be missing.)

****
Response: Ok, it is fixed. Thanks!
*****

- Figure 5, rule ConsVM: I think the precondition 's_{p1} /= epsilon' is not needed. The text highlights that the original RE is unproblematic; therefore, the body of a Kleene star cannot match the empty string.

****
Response: Ok, it is fixed. Thanks!
*****

- Line 320: "Rules LeftVM and RightVM specifies" -> "specify"

****
Response: Ok, it is fixed. Thanks!
*****

- Line 348: "in type theory-based" -> "in type-theory based" (or "in type-theory--based")

****
Response: Ok, it is fixed. Thanks!
*****

- Line 366: "which provide" -> "which provides"

****
Response: Ok, it is fixed. Thanks!
*****

- In would help to explain why we need 'invalid_code' (to define 'decode' later).

****
Response: In order to define function decode using a dependent type, we need to consider both successful and failure situations. Type nocode_for denotes a proof that a given bit sequence is not a valid code for a specific regular expression. Notice that nocode_for denotes a proof that a full bit list is not valid. However, another possibility is that just a prefix of the input list could be parsed as a code for the RE, and it leaves a remaining non-empty bit sequence without parsing, which is another possibility of a decoding
error. We use type invalid_code to combine these two kinds of errors.
*****

- Line 467: "The completeness express" -> "The completeness expresses"

****
Response: Ok, it is fixed. Thanks!
*****

- Line 499: in the description of the set up for the experiments, it seems important to specify which version of Haskell was used.

****
Response: Ok, fixed. Thanks!
***

- Line 518: "Such RE pose a challenge" -> "poses".

****
Response: Ok, it is fixed. Thanks!
*****

- Line 526: Why/how does the presented VM have linear performance on those problematic inputs? (A benchmark where we cannot understand its results is not very helpful.)

****
Response: We improve our evaluation section introducing an analytical model of our parsing VM semantics. Basically, the model shows that the semantics exhibits a performance of the order O(n). A proof sketch of this property is included in an appendix.
****
- Last line: What is the difficulty of knowing the time complexity of the proposed algorithm? (Why is it future work?) It seems important some discussion about the complexity (which, of course, demands a discussion of the algorithm itself).

***
Response: As we have answered in the previous question, we introduced an analytical model that is used to formally justify the linear time performance of our semantics.
******

Reviewer #2:

The paper develops and formalizes a virtual machine for parsing regular languages. The specification of the virtual machine is given in terms of a big-step semantics which produces a bit-code representation of parse trees. The machine is proved correct wrt the membership semantics of regular expressions. The formalization of the virtual machine and its ecosystem (associated definitions and properties) is performed in Coq. Finally, a Haskell implementation of the VM-based interpreter is obtained from the formalization by using Coq's program extraction mechanism; experimental results are presented for that implementation.

In comparison with the original paper published at SBLP'18, the present paper presents several improvements and differences.

The paper is interesting, in general well-written, and presents a good account of related work. The contribution of the paper is much stronger than that of the SBLP'18 paper as it presents the complete formalization of the big-step semantics for the VM and its proof of correctness. I have some comments and suggestions that I present as part of the specific comments below.

Regarding the structure of the paper, I would move the section 2 on Coq to an appendix and give more importance and space to the results associated with the VM.

****
Response: We moved the Coq introductory section to an appendix.
*****

Specific comments:

- page 10
l.44-45 maybe (fun x => P) instead of maybe P

*****
Response: Keeping P is correct since, in maybe type definition, we have defined maybe's type parameter as A -> Prop, so there is no need to add an anonymous function to introduce an element of type A.
*****

l.46 extra curly braces missing in {x|P}, correct is {{x|P}}
l.46 [n] is actually [|n|]

****
Response: We have changed our specification, and now, our Coq code does not use the maybe type. Now, instead of using the refine tactic to give the computational content of a dependently typed function, we are constructing everything using Coq tactics.
*****

- page 11
l.45 (Figure ??) is (Figure 1)

****

Response: We have changed our specification, and now, our Coq code does not use the maybe type. Now, instead of using the refine tactic to give the computational content of a dependently typed function, we are constructing everything using Coq tactics.
*****

- page 13
l.10 RE -> REs
****

Response: Ok, it is fixed. Thanks!
*****

- page 14
l.48-49 add a reference to Fig. 4
****

Response: We have added a reference to Fig. 4. Thanks!
*****

- page 15
The bit-code generation does not need to be driven by an input RE. Like flatten, function code can produce its output string (a bit-code) without looking at the type (RE) of the input tree. Defining the way it is defined gives the impression that code is actually taking proofs of the parse tree typing relation (|- t : e) as input. The relationship between parse trees and the corresponding bit-codes is clearly stated in Thm 3 (page 17) and there you take as hypothesis that |- t : e.

****

Response: We keep RE on the input of function code to follow Heinglein et al. original formulation of coding / decoding of RE parsing trees. We introduce the following sentence to clarify this:

While coding a parse tree does not depend on its underlying RE, we keep it on the code function to follow the original definition of Heinglein et al.
*****

- page 16
l.9-14 The explanation about the bit-code of Example 2 at the beginning of the page does not clarify much; rewrite it or simply drop it.

****

We have improved the explanation of this example and the drawing of its bit-annotated parse tree. Thanks!
*****

l.44 approach _to_
l.47 _an_ equivalent

****

Response: Ok, it is fixed. Thanks!

*****

- page 18

l.32 obeys -> obey

****

Response: Ok, it is fixed. Thanks!

*****

- page 20

It would be nice if you can motivate somewhere the need for introducing a VM, compared with other approaches like eg. parsing functions for REs.

l.16-18 may produce as a side effect -> produces as result

****

Response: Ok, it is fixed. Thanks!

*****

Fig. 5: you should mention that the semantics is non-deterministic.

****

Response: We have fixed our semantics in order to follow the greedy disambiguation strategy. Now our semantics is deterministic.

*****

===> is it necessary the precondition s_{p_1} <> \epsilon in rule ConsVM? Putting the precondition that the RE is unproblematic (as it is done in the formalization of the interpreter in Coq) it is not necessary to check within the rule that the string s_{p_1} is non-empty.

****

Response: You are right. We change rule ConsVM dropping the precondition that s_{p1} <> \epsilon. Thanks!

*****

l.46-47 semantics rules -> semantic rule

****

Response: Ok, it is fixed. Thanks!

*****

- page 21

l.12 for the _RE_ e_1 e_2

****

Response: Ok, it is fixed. Thanks!

*****

l.14-15 You say that "rules NilVM and ConsVM deal with unproblematic Kleene star REs". As far as I see only ConsVM deals with unproblematic Kleene stars, I don't see how NilVM do so. ConsVM properly works when the Kleene star is unproblematic because of the condition s_{p_1} <> \epsilon. Perhaps it would be better if you simply define the big-step semantics assuming that the RE is unproblematic. That way it would be unnecessary to check that property in ConsVM. At the end you want that condition to hold on the RE when parsing a string. In fact, you do that in the formalization of the interpreter in Coq: you first convert the RE to an unproblematic one and then you run the VM on the input string. Therefore you are always executing your semantics on an unproblematic RE. Of course, adding that precondition on the RE surely would require to add it in all properties involving the big-step semantics of the VM (eg. Thms 5, 6 and 7).

****

Response: We follow your recommendation, and now we assume that our semantics deal only with non-problematic expressions. In our interpreter definition, we convert any regular expression into an equivalent non-problematic one.

****

Thms 4, 5 and 6 state the main properties of your semantics. Being a journal paper I would assign more space for those properties and show, at least, proof sketchs. That may appear in an appendix.

****

Response: We have added an appendix containing the proofs sketches of the main paper properties.

*****

- page 22

l.31-32 You need to mention that ($ c) denotes Chr c. I assume that String c "" means "c"; for readability, I would directly write "c" in the paper.

****

Response: Ok, it is fixed. Thanks!

*****

l.51-55 The where clause and the sentence just after speak about the same. Use one for or the other. In case of using explanations about notation it is better that they appear before the the Inductive definitions seem more readable and

l.42-43 why the condition s <> "" in InStarRight?

****
Response: This condition is not necessary. We removed it.
*****

- page 23
l.31 its -> the
l.53 its types -> their types
l.54-55 has its -> satisfies the

****
Response: Ok, it is fixed. Thanks!
*****
- page 26
l.27 _an_ input
l.31-32 _an_ unproblematic
l.42-43
Function unprob_rec's
return_ed_ RE
l.44 accepts -> accept

****
Response: Ok, it is fixed. Thanks!
*****

- page 27
l.53-54 Function flatten _(shown in page 13)_

****
Response: Ok, it is fixed. Thanks!
*****

- page 28
Be careful that the name "code" is used for the function defined in page 15, but also as a type synonym for list bit (Definition on line 13). It is better to name that function as "encode" from the beginning (when it is introduced in page 15), since that is its name in the Coq formalization.
****
Response: Ok, we have changed all references to function name "code" to "encode". Thanks!
*****

- page 29
l.12-13 code -> encode
l.53-54 empty set RE, ##0 -> empty RE #0.

****
Response: Ok, it is fixed. Thanks!
*****

- page 30
l.24 code -> encode

****
Response: Ok, it is fixed. Thanks!
*****

In the existential, invalid_code should say that the tree t is of type e.

****
Response: It is not necessary to include such a condition because of the theorem "decode_sound", which states that the produced tree is well-typed.
*****
l.35 code -> encode

****
Response: Ok, it is fixed. Thanks!
*****

- page 31
l.34-37 Write the explanation about notation before the definition of in_regex_p.

The definition of in_regex_p does no follow completely the semantics in Fig 5 because it does not consider the production of bit-codes. You should comment why you are not considering the production of bit-codes as part of in_regex_p. In principle, I would have considered their production for the definition of the interpreter (interp).

****
Response: As requested by Reviewer #3, we have changed our semantics in order to consider the greedy disambiguation strategy and also include the bit-codes as a result of our semantics.
******

Mention that the Theorems in_regex_p_complete and in_regex_p_sound correspond to the Thms 6 and 5, respectivly.

******

Response: We have added the corresponding references. Thanks!
******

l.52-54
express_es_
an string -> a string

******
Response: Ok, it is fixed. Thanks!
****

- page 32
l.29 _an_obvious

*****
Response: Ok, it is fixed. Thanks!
******
In the interpreter, how you deal with the fact that the semantics of the VM is non-deterministic?

****
Response: We have changed our semantics definition, and now it is deterministic.
****

- page 34
Figue 7: input size thousands of _"ab"s_

****
Ok, fixed. Thanks!
*****

l.27-29 You present the experiment by saying that you parse strings of the form a^n. Why not simply saying, as in the previous experiment, strings of a's? The RE in this case should be (a+\eps)^*a^*.

*****
Ok, fixed. Thanks!
*****

l.34 is preented below -> are presented in Figures 8 and 9.

*****

Response: Ok, it is fixed. Thanks!
******

- page 35
I would eliminate footnote 6.

******
Ok, we have removed it. Thanks!
****

- page 36
I would eliminate completely all Haskell code of this page. It seems unncessary to show it. You can give a rough idea of how you generate REs and strings for that RE without showing the code.
****
We believe that the generation algorithm is an important component of how our experiments are executed. This is the main reason that we keep the Haskell code of
random test case generation algorithm.
****

- page 37
l.14-15 "to" appears twice

*****
Response: Ok, it is fixed. Thanks!
******

Reviewer #3:

The notion of equivalence of REs is defined as being language equivalence.
Thus statements about turning, for example, a problematic regex into an equivalent non-problematic one, is not that interesting. A more interesting research question would be to consider what happens when
(the infinitely many) parse trees of a problematic regex and given input string are considered and compared to those obtained by the
non-problematic regex obtained via the construction of Medeiros (or related constructions).

*****
Response: The research question of what is the computational content (on parse trees) of Medeiros et. al. construction on problematic REs is left for future work. We intend to characterize Medeiros et al. construction in terms of the coinductive axiomatization of Heinglein and its co-workers in "Regular expression containment: coinductive axiomatization and computational interpretation".

******

In my opinion, the proposed approach should be extended in order to handle
Greedy and/or Posix regex matching. In short, considering all parse
trees (or bit-codes) when an input string is matched by a non-problematic regular expression, is too close to classical
formal language theory that is well understood. Regular expression matching becomes interesting
when disambiguation policies (such as Greedy or Posix) is used to associate at most one parse tree (or bit-code)
with a given RE and input string.

****
Response: We have modified our formalization to follow the greedy disambiguation strategy. Now our semantics is deterministic: given a regular expression e and a string s
it outputs only one bit-coded parse tree or no parse is possible.
****

Alternatively (or in addition), research in line with the following statement on p39 will improve novelty: "In many cases, the goal of a RE is not only to match a given text, but also to extract information from it."

****
Response: The main novelty of Radanne paper is to use a GADT to provide a type-safe interface for RE parsing. While our extracted Haskell code does not use GADTs, we have proved that our algorithm is correct, which includes proofs that its parsing / unparsing behavior is correct.

Providing a similar GADT-based interface that encapsulates the verified extracted code is a simple task, but not related with our main work objective: formalizing an operational semantics for RE parsing and use it to build a tool with a reasonable performance.
****