// INTRO: We have made all the spelling/grammar changes you have suggested.
// All other changes we have highlighted in blue in the pdf, which we have
// placed markers, of the form **^<number>**, which we refer to in our responses below.

The reviewers agree that the paper has been improved. Reviewer 1 and 2 are quite happy, but reviewer 3 still has major concerns, and feels that some of his comments have been addressed in an unsatisfactory manner. I would like to ask you address the remaining comments. If you feel that certain comments should not be addressed in the paper, I would like to see a clear explanation of why you decided to do so.

Reviewer #1: * Comments after revision
I think this revised version does a good job of addressing my previous critique without compromising the strong points. The comparisons to related work has been improved, the explanations of the different concepts are much clearer, and the paper is strengthened by the inclusion of a formal type system. The `rep` terminology works a lot better than overloading the `capsule` concept. I think the paper could be accepted after a few minor revisions (see detailed comments below).

Personally I think Appendix C detracts from the main story, so it could be removed. As before, I think Section 8 about L42 could also be removed, since it is too short to give a detailed description of the whole language (and trying to do so would definitely be out of scope). Leaving these sections in would not be a catastrophe, but I think it would make the paper stronger. Incidentally, it was from Section 8 that I previously got the misconception that rep mutators needed to be identified by the programmer. Regardless of what the previous version said, the sentence "The type-system requires that any method that could alter the result of a `@Cache.Now` method (except via a field update) must be marked with `@Cache.Clear` [...]" makes perfect sense to me now.
// DONE: We've removed appendix C

** Highlights (same comments as last time)
- "magnitudo" -> "magnitude"
- "Much less annotation burden" --> "Lighter annotation burden"?
  "Lower annotation burden?"

** Section 1
- 61: In this example I was thinking that it would be a problem if you could pass `this` to `box.set()`. Later it was explained that this was not allowed. You might consider mentioning it.
// DONE: See ^1
- 88: “of the L42, and similar, type systems” → “of the type system of L42 and similar languages”
// DONE
- 94: “show” → “shows”
// DONE

** Section 2
- This section is a lot better now!
// Thanks!
- 112: “Very useful” → “useful”
// DONE
- 129: “the referenced object cannot mutate” → “the referenced object cannot be mutated”
// DONE
- 140: Somewhere around here I started getting annoyed at the many sentences and paragraphs beginning with “That is,”. Many times, these words could be removed without changing the meaning of the sentence.
// DONE: See ^2.1, ^2.2, ^2.3, and ^2.4
- 188: “contains” → “contain”
// DONE
- 190: I can see how one could use `capsule` fields to “model various kinds of ownership”, but what does it mean to model parallelism? Maybe “facilitate parallelism”?
// DONE: See ^3
- 192: “alias to” → “aliases to”
// DONE
- 216: “magine” -> “imagine”
// DONE
- 216: “in a simple minded imperative style:” remove altogether: “...belong to classes designed without worrying...”
// DONE

** Section 3
- 291: I don’t see why you couldn’t write invariants over a doubly linked list with mutable elements as long as these elements are also encapsulated? In later examples it doesn’t seem to be a problem to have invariants that access internal state that is otherwise mutable (in rep mutators).

- Another thought: could it be the case that parametricity saves you here? If your list is polymorphic over its elements, then the invariant of the list cannot dereference an element and thus the state of the elements cannot affect the result of the invariant.

// DONE see ^4.1, ^4.2, and ^4.3

- 302: When you talk about pure methods not mutating existing memory, maybe clarify that you mean heap memory (writing to local variables on the stack is fine).

// RESPONSE: Our statement applies to both heap and stack memory. Writing to *existing* stack locations (e.g. in languages where a method can take as an argument references or pointers to local variables, like the “out” parameter in C#) would make the method *not pure*. Writing to new variables is OK, but then those stack locations will not be "pre-existing" in memory (i.e. they would have been pushed/reserved on the stack when the pure method was called).

- 344: It was not immediately obvious to me why a rep mutator must "have no mut or read parameters". After some thinking it made sense (you might otherwise bring an alias of `this`), but you might consider clarifying this point (and the others in the same list).

// DONE see ^5

- 350: "than" -> "then"

// DONE

** Section 4

- Some of these examples use inheritance, which is a bit strange since you ended the previous section explaining how L42 does not have sub-classing.

// DONE: see ^6

- I still like this section very much!

// Thanks!

** Section 5

- 537: "generic" --> "generics"

// DONE

- 563: "We assume that the type system imposes any additional constraints it needs on method bodies". This seems like it doesn't add anything. If anything, maybe phrase it as "We *allow* the type system to impose any additional constraints it needs...".

// DONE

- 585: You should move the explanations of booleans before 573 where it is used.

// DONE

- 619: "read l.invariant()". Consider adding parentheses for
  clarity: "(read l).invariant()".
// DONE
- 645: You should move the explanation of the different expression
  contexts before the definition of error states, which requires
  them.
// DONE
- Figure 2: Consider adding parentheses as suggested for 619.
// DONE

** Section 6
- 815: Consider replacing "many" by a number (or weaken to
  "several").
// DONE (used "several")

** Section 7
- I find it interesting how the transform pattern basically solves
  the same problem as iterators in ownership systems; it is easier
  to move computations to the data than the other way around.
// Thank you, we've mentioned this in the paper, see ^32

** Section 8
- "In the last version of L42 [...]". What does this mean? Do you
  mean "the latest"?
// DONE

** Section 9
- 1535: "However, ownership does not require the whole reachable
  object graph of an object to be 'owned'. This complicates
  restricting the data accessible by invariants". Traditional
  owners-as-dominators types systems only allow referencing an
  object if you can mention its owner, so an ownership context
  that is not passed any external owners will dominate its whole
  reachable object graph (with the possible exception of the
  top-level `World` owner, if that exists).

// RESPONSE:
There are multiple forms of ownership in the literature, and we have not examined all of them. We have considered the seminal “owners-as-dominators: paper:

"Ownership types for flexible alias protection" by David Clarke, John Potter, and James Noble (https://dl.acm.org/doi/10.1145/286942.286947)

In our understanding, the restriction is only that "**rep**" data cannot be accessed outside of the owning object's methods ("**rep**" data being owned by "this").

It does not restrict the access of data owned by someone else, provided that you can state its full type declaration (including any relevant ownership parameters).

For example, we believe the following would be a valid use of ownership types:

```
class Baz {
   ...
   norep Boolean something() { ... }
}
class Foo<m> {
    m Baz x;
    norep Baz y;
}
class Bar<m> {
    rep Foo<m> f;
    norep Boolean invariant() {
        this.f.x.something() && this.f.y.something();
    }
}
```

However, this example shows that ownership types are not sufficient to support the runtime invariant protocol. The problem is that `'this.f.x'` and `'this.f.y'` are NOT owned by the `'this'` in the invariant method, which means that there could be aliases to these objects elsewhere. Hence, they can be mutated --- changing the result of `something()` --- in a context that is unaware of `Bar`'s invariant, thus breaking the invariant while bypassing all runtime checks.

The ownership type system *does* prevent other objects from accessing things *owned* by the `'Bar'`: i.e. other objects cannot read or write directly to the object in the `'f'` field of `Bar`, except by going through `Bar` itself. However, it *does not* prevent the invariant method from accessing things that are *not owned* by the `Bar`: either because they are globally owned (as in `norep`), or because they are owned by a parameter of the `Bar` class (`'m'`).

- 1575: This whole paragraph makes some vague claims about the
  field of "static verification" and does not back it up with any
  citations.
// DONE (Added Citations)
- 1613: "difficoult" --> "difficult"
// DONE
- 1651: Unless you have a strict page limit, I would avoid
  introducing the abbreviation "RV". As far as I can tell you only
  use it twice.
// DONE

** Section 10
- 1695: I disagree that you have "identified the essential
  language features that support representation invariants in
  object-oriented verification". You have presented *a* set of
  language features that cleanly supports representation
  invariants in an efficient manner, but there is nothing that
  says that they are the only set of features or that they are
  more essential than any other features.
// DONE: see ^7
- 1698: "we require many order of magnitude less runtime checking".
  "order" --> "orders", and strike "many". Also, "less runtime
  checking" reads weird to me. "fewer runtime checks" maybe?
// DONE
- 1708: "such a support" --> "such support"
// DONE
- This ending is much stronger than before!
// Thanks!

** Appendix A
- 1906: "typesystem" --> "type system" (twice, and again later)
// DONE
- 1943: "As we do not have a concrete type system [...]". Yes you
  do! It's in Appendix B!
// DONE: see ^8
- 2004: "Usefull" --> "Useful"
// DONE

** Appendix B
- Consider restating the `<=` relation for capabilities here for
  completeness.
// DONE: see ^9
- 2538: "...we can always extract a \sigma; \Gamma |- e :: T from a
\sigma; \Gamma |- e :: T'
  judgement". One of these should only have a single colon.
// DONE

** Appendix C
- 3196: "references" --> "reference"
// N/A: appendix has been deleted

- 3211: "L42 does not support destructive reads". This surprised me! It may make sense to make this point clear earlier in the main part of the paper. While I agree that adding uniqueness tracking complicates the story, it seems like allowing moving capsules between objects would make your verification story stronger, especially in a concurrent setting.
// DONE: we pointed this out earlier ^25
- 3226: "a 42 user" --> "an L42 user"
// N/A: appendix has been deleted
- 3232: "parallelism in 42 is unobservable" --> "parallelism in L42 is unobservable:"
// N/A: appendix has been deleted

Reviewer #2: I am generally quite happy with this revision. Much of my concern for the prior submission derived from lack of clarity around some of the context, meanings, and terminology. These were clarified very directly in the author comments on the last round of reviews, and I see the clarifications the paper made towards these ends in the main text (clarifications around what are now called rep fields, that L42 does not permit storing read-only references into the heap, and restrictions on concurrency in L42). It's possible a "fresh" reader of this paper might easily overlook those clarifications (I was specifically looking for them). Nonetheless I don't think those important clarifications are in odd locations or particularly subtle, so I don't think there's anything specific to improve there.

I appreciate the addition of a complete, small type system in Appendix B, which does go some way towards clarifying the important ways L42 differs from work on Gordon et al.'s language and Pony, and convinces me the type system requirements are feasible, and subjectively it now seems clear (to me, at least) that the other reference capability systems could be adapted to make this paper's ideas work there, too.

I'm torn on the value of Appendix C. On one hand it goes much further than the paper's updated main text in clarifying the highly-structured parallelism in L42. On the other hand it gets bogged down in some details of L42 that really are orthogonal to this paper's contributions, and start pulling in more and more of the L42 details the paper otherwise works hard to stay mostly-independent of. I don't think it's problematic to keep it as-is, but if the authors are on the fence about it (which is how I interpret the author comments responding to reviews), a short \paragraph{} dedicated to the forms of fork-join parallelism supported in L42 earlier in the paper (perhaps where the claims about supporting safe parallelism are made), at a high level, and how they are tied to the reference capabilities here, without any syntax or examples, would be sufficient.
// DONE: removed the appendix, also see ^26.1

This paper is mostly in great shape now, and definitely the most thorough examination of invariant protocols I've ever seen. I do however have some mild concerns about the newly-added material; nothing fundamental, more a function of the fact that it is significant additional technical development which necessarily benefits from some additional attention to detail and polish. My one remaining significant concern is that the proofs added in this major revision are a bit sloppy, in that they state one thing and prove something that is intuitively related but not formally derivable from the stated assumptions. This must be fixed. In particular, Appendix B Theorem 6 (which connects the core type system to the requirements for the technique) is not stated in a form that is provable. The statement of the theorem makes claims about something being true for any valid state. But the proof conjures a reduction trace out of thin air and does induction on that. So it seems that Theorem 6 should really say its conclusion holds for any valid state *reached via reducing a valid program* (appropriately formalized). This is mostly a matter of correcting the formal statement of Theorem 6 rather than changing the proof (which itself seems fine if the reduction assumptions are appropriately introduced), and ensuring that is consistent with Requirements 2 and 3 on the type system as listed in appendix A. Currently those Appendix A requirements of a type system *also* don't talk about reductions, but only about valid states, which means there is currently a misalignment between the stated requirements and what is actually proven about the type system. This should be fixed. I think this is *mostly* a matter of giving the proper formal statement of Appendix B Theorem 6, adjusting Requirements 2 and 3 to mention the reduction assumptions, and plumbing those requirements the rest of the way through the other proofs (Appendix A Theorem 2 Rep Field Soundness, Appendix A Lemma 5 Imm Not Circular, and Appendix A Theorem 1 Soundness, plus the proofs in Appendix B that Theorem 6 implies Requirements 2 and 3).
// RESPONSE: The definition of ValidState directly requires that such reduction sequences exist, we have clarified the proofs, see ^27.1, ^27.2, and ^27.3

I have one other general comment on a general way the paper could be strengthened, but I consider it optional (I'd be okay with the authors *not* taking action on this). Currently it seems like the paper is of two minds with regards to what it's about: Is it about the type system, or not? The paper is full of offhand remarks about how certain problematic code would be rejected by the type system, while much of the paper maintains the position that the specific type system is not of critical importance (in the sense that Appendix B is "just" an example, and the approach should work with many other plausible type systems).

Currently this results in the paper sometimes reading a
bit oddly at times. I think the paper would do well to early on call
out the notion of a *compatible type system* , and discuss how any
compatible type system would rule out this example or another
(doesn't have to be the word "compatible," but I think that word could
work). This idea is clearly
present in the paper, but not explicitly named, which makes it awkward
to refer to. Naming it with a specific term, early on, would allow
cleaning up awkward examples where the paper claims certain code is or
is not well-typed without officially committing to a type system. (An
alternative would be to commit to the type system of Appendix B, but I
think that would require more substantial rewriting to accomplish, and
would actually make the paper weaker by de-emphasizing the abstract
requirements for the type system that I like in this paper.)

- (Also) p12 line 466: "well-typed" Is this paper about the type system, or not?

// RESPONSE: We agree but we could not work out where we should use it though so left it. Thank you!

Miscellaneous comments:

- p3 (paper page number, not pdf page number), line 73: A one-line
summary of Gopinathan et al.'s work here would be helpful
// DONE: See ^10
- p4 line 112: "usueful" please run spell check; there aren't too many
misspellings, but I'm not going to document them all
- p13 line 539: "turing complete" should be "Turing-complete"
// BOTH DONE
- p14 footnote 17: This is actually broken as written, because the
formal calculus uses call-by-value semantics, so "b.if(t,f)" would
execute *both* branches' side effects. t and f would need to be
'thunked' in such a way that the boolean's implementation of 'if'
explicitly triggered the computation of one or the other. (This is a
common exercise when teaching CBV vs. CBN semantics.) For the paper,
I'd actually just abstract even further in this footnote, and just
note that in a language where one can implement thunks, one can
implement conditionals.
// DONE see ^11
- p34 line 1497: "last version of L42"? Last rather than latest? This
wording suggests work on L42 has stopped.
// DONE
- p35 line 1519: "current concrete L42 syntax.... various kinds of
restrictions over fields" This is hard to interpret from seeing just a
single use, which inherently doesn't demonstrate "various"
restrictions.
// DONE: see ^32.1 and ^32.2

Reviewer #3: It is my second review of this paper. I would like to thank the author to have answer or take into account some of my remarks, but I am disappointed that:
- the differences between the former and the current versions of the paper are not highlighted, so it is very hard to see what has precisely changed.
- several of my remarks have been silently ignored (neither discussed nor taken into account)

Therefore, I still recommend a major revision for this paper. Please, answer each remark: either explain why you do not take it into account, or precisely indicate which part of the paper has been modified accordingly (and how).

// RESPONSE: We have made all the spelling/grammar changes you have suggested.
// All other changes we have highlighted in blue in the pdf, which we have
// placed markers, of the form **^<number>**, which we refer to in our responses below.

** New technical remarks

p1,l38: "In a pure functional setting, sound runtime checking is trivial"

I disagree: it depends on the properties that you want to check. For instance, [1] and some of the papers in its related work give a few examples of properties that are not trivial to check at runtime (some are still considered as open problems). However, arguably, for the same kind of properties, it is usually easier to check them in a pure functional setting than in an impure one.
[1] Jean-Christophe Filliâtre and Clément Pascutto. Ortac: Runtime Assertion Checking for OCaml. In International Conference on Runtime Verification, 2021.
// DONE see ^12

** New typos

p3, l21--44: a dot is missing at the end of each itemize
// DONE
p26, l25, In Spec# objects --> In Spec#, objects
// DONE
p31, l13: interface HasSubInvariant, that --> remove the comma
// DONE
p43, l29: need not --> does not need to
// DONE

** Comments from review 1 that are answered in a partial or unsatisfactory way:

- p16, l56: The verifier available online [...] behaves differently:
please add explanations here (and maybe as plain text and not as a
footnote).
Answer: "Unfortunately it was a long time ago so we do not remember the
exact details, but the online verifier was accepting incorrect
programs that the offline verifier was appropriately rejecting."

Actually, this online verifier does not exist anymore because rise4fun
is dead (see [2] for instance). Therefore, this sentence should be
removed.
[2] https://github.com/Z3Prover/z3/discussions/5473
// DONE

- p17, l27: the number of characters is meaningless (e.g., it is
debatable which keyword is better between 'mut' and 'mutable'). I am
not even sure whether the number of tokens is relevant (liking
verbosity or not is a matter of taste)
p31, l46: three times less annotation burden than [...] Spec# --> I
do not agree with this conclusion if you only look at the number of
annotations (which is the most meaningful measure IMO)

Your answer to the second question: "Some Spec# annotations are very
involved, consider this single annotation:
"Owner.Same(Owner.ElementProxy(children), children)" In our
understanding this is a single annotation, so when we count
annotations, that whole string would count as 1. On the other hand,
when we
count tokens, that string would count as 6. (note how it does not
count 13 since we do not count " ()[]{},;" as tokens) That is why we
considered tokens to be a better metric.

In general, the number of tokens/characters highly depends on the
verbosity of the underlying programming language or the way the code
is written, while the number of annotations is more closely related to
the specification language and its associated (sets of) technique(s).
I agree that it is debatable because my claim is not proven. However,
I know several formal methods' papers involving formal annotations
that use the number of annotations as a metrics and none (but yours)
that uses the number of tokens/characters. Therefore, I think that you
should explain why you use these new metrics in your context, and be
more cautious about your conclusion that relies on them (except if you
have a very strong convincing statement about why your metrics is
better that the number of annotations).

// RESPONSE: We have updated our conclusion to report both the difference in annotations, and the total difference in tokens ^29.1, ^29.2, and ^29.3.

One of the reasons why we also report 'tokens' and not just the number of 'annotations', is that in Spec# the number of annotations depends of the programmer's style.

For example, in our Spec# code we have annotated a method with two annotations:

```
ensures Owner.Same(Owner.ElementProxy(list), list);
ensures result == list;
```

Whereas we could have equivalently written:

```
ensures Owner.Same(Owner.ElementProxy(list), list) && result == list;
```

Which would be a single annotation (but both options have the same number of tokens). On the other hand, one annotation in our approach is always just a single concept: a keyword such as **mut**, or **rep**. That is, a Spec# annotation can be an arbitrarily complex Spec# expression (potentially any Spec# expression which is a general-purpose language).

The exact token count depends on language design decisions, for example using `Owner.Same` instead of `Owner_Same`, which would be a single token.

However, other complexity, such as having to specify that the owners must be the same in the first place, and having to specify that you want to talk about the `ElementProxy` of the list (and not just the list itself) are fundamental requirements of Spec#'s ownership discipline.

Thus, though the token count is not a perfect metric, we believe it complements the count of the number of annotations as it helps give an idea of this extra complexity that programmers would need to consider.

The exact character count is the least important of the three: it depends heavily on such design decisions as whitespace sensitivity or how much to abbreviate English words. We could remove character counts, but we think it still gives a nice and intuitive feel for the difference.

- p30, l18: a study [19] discovered that developers expect specification languages to follow the semantics of the underlying language --> written this way, this statement is just wrong even though the two examples that follow are correct. For instance, a previous study (*) from the same author (P. Chalin) concludes that "there is a semantic gap between user expectations and the current language design and semantics of JML numeric types" (the semantics of JML numeric types at that time was the one of Java) and it contributes to modify this semantics that comes from Java (even if it is still possible to activate it).
(*) Patrice Chalin. JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics. Journal of Object Technology, 2004.

Your answer: "We are confused about this remark. We checked again [19] (now called [69]) and it seams to confirm our sentence (for example in pg10 Table 2). We wonder if this is simply an English problem. What should we say to be more clear?"

The question of P. Chalin that is answered in [69], page 10, Table 2 is (quoting [69]) "in general, what should be done if an exception is raised during the evaluation of an assertion expression". Therefore, there is no general statement about what developers expect about the specification language semantics, but
only one particular statement related to the semantics of exceptions when evaluating assertions. As explained in my initial comment, another Chalin's study (see above) concludes that developers expect two different semantics for JML and Java for numerical types and it leads to changing the semantics of JML w.r.t. the one of Java for numerical types. Therefore, in your paper, you can talk about the particular cases of short-circuits and exceptions (for which the developers indeed expect the same semantics), but not about the general case because there is at least one counter-example.

DONE: We have weakened our claim in the paper (please see note ^30). We were confused by the paragraph in [69], immediately following the one you quoted above:

“The purpose of C.3 was to determine, from the respondent’s point of view, if there should be any difference in the semantic interpretation of a Boolean expression when used as an assertion vs. outside the context of an assertion. Seventy percent (70%) of respondents answered that expressions should be given the same interpretation."

Which led us to our more general claim.

** Comments from review 1 that remains unanswered (as far as I have seen):
(the page and line numbers are not the same in the current version of the paper)

- Even though the related work is very precisely made w.r.t. the visible state semantics and Spec#, it only refers to non-recent papers (all of them were published in 2017 or earlier, and most of them before 2013). For instance, the authors do not compare their notion of invariants to invariants of other (non-OO) specification language such as ACSL, Spark2014 and WhyML.
// DONE: See ^31.1, ^31.2, and ^31.3

- p2, l21: the execution of this.min = min will --> use "would" (this line of code is actually never executed)
// DONE

- p2, l29: at this point, the reader cannot understand the advantage of using the Box pattern (as opposed to its drawbacks, which are easy to see)
// DONE: see ^24

- p2, l46: With appropriate type annotations --> which ones?
// DONE see ^13
- p4, l45-51: examples for mut and capsule variables would be welcome
// DONE see ^14, we already have an example for mut variables above.

- p5, l29: what does "opt-in" mean?
// DONE we decided not to use that term, and clarified what we meant, see ^15

- p5, Exceptions: an example would be welcome
// DONE see ^16

- p7, Capsule Fields: an example would be welcome
// DONE see ^17

- p11, Formal Language Model: it should be clear from the introduction of this section that it handles OC and I/O.
// DONE see ^18

- p12, l29: the notation '\sigma \setminus l' looks to be only used in the Appendix. Therefore, it should be introduced there and not in the core paper.
// DONE see ^19

- p13, l27: an example for M(l;e1;e2) would be welcome, as well as another one for the strong exception safety remark.
// DONE see ^20 and ^21

- p13, l49: "For example, in L42 we implement [...]" --> do not mix the formal presentation with implementation details. Instead, you could add this remark in Section 8.
// DONE We just deleted it, as we mentioned our implementation strategy elsewhere already

- p15, l50-54: the invariant performs n*n comparisons while it could do only n*(n+1)/2 (if not comparing twice each pair of widgets). If you do not want to rewrite it, add at least a remark explaining that the code efficiency is not the priority here.
// DONE: we had already fixed this, see see ^22

- p35, l28: the formal definition of ‘rog’ is missing
// DONE: it had already been fixed, see ^23 (in the appendix as we don’t use it formally within the main-paper)

- p1, l37: boolean --> Boolean
// DONE
- p2, l21: >= --> greater than
// DONE: used "greater than or equal to"