#59 Sound Invariant Checking Using Type Modifiers and Object Capabilities

Email notification
Select to receive email on updates to reviews and comments.

PC conflicts
Alex Potanin
Bruno Oliveira
David Pearce

Submitted

Submission 12 Jan 2019 12:10:32am GMT · 3205c089

▶ Abstract

In this paper we use pre existing language support for type modifiers and object capabilities to enable a system for sound runtime verification of invariants. Our system guarantees that class invariants hold for all objects involved in execution. Invariants are specified simply as methods whose execution is statically guaranteed to be deterministic and not access any externally mutable state. We automatically call such invariant methods only when objects are created or the state they refer to may have been mutated. Our design restricts the range of expre [more]

▶ Authors (blind until review)

I. Gariano, M. Servetto, A. Potanin [details]

Paper category

Research Paper

▶ Topics

	OveMer	RevExp
Review #59A	B	Y
Review #59B	B	Y
Review #59C	A	Y
Review #59D	D	X

You are an **author** of this submission.

Edit submission Add response

Reviews in plain text

Review #59A

Overall merit

B. Weak accept: I think this paper should be accepted but I will not champion it

Reviewer expertise

Y. Knowledgeable

Paper summary

Verifying code is a holy grail and this paper makes a contribution to it with a dynamic class invariant checking system. It introduces the system through examples and makes a convincing case that the new system is more flexible for the programmer and adds less runtime overhead than earlier attempts. There is a detailed worked example with its closest competitor Spec# where It does look easier for the programmer to write. The annotations do not look onerous to use.

The underpinning of this contribution is based on type modifiers and object capabilities. The authors implement their class invariant checking in L42 and prove that it is sound.

Comments for author

Verifying code is a holy grail and this paper makes a contribution to it with a dynamic class invariant checking system. It introduces the system through examples and makes a convincing case that the new system is more flexible for the programmer and adds less runtime overhead than earlier attempts. There is a detailed worked example with its closest competitor Spec# where It does look easier for the programmer to write. The annotations do not look onerous to use. The underpinning of this contribution is based on type modifiers and object capabilities. The authors implement their class invariant checking in L42 and prove that it is sound.

When looking at a program verification system there are a few questions that need to be answered. Is the system expressive enough for programmers or is it too restrictive? Is the system sound? Does it slow down running programs, or do they have acceptable performance? Is the system just too cumbersome to use?

This paper makes a claim that their invariant system would provide positive answers to these questions. There is a soundness proof in the appendix. Soundness is non-trivial because of dynamic dispatch, non-determinism, IO, and exceptions. As an underlying idea for the authors' L42 protocol is not to check invariants when nothing could have changed, timings shouldn't be worse than systems that need to check more. Nonetheless having timings on a wide range of examples would have been re-assuring. The examples do not look onerous to write and the informal evidence that it is easier to use their L42 based system than Spec# is convincing. However, Spec# is more powerful and it would have been interesting to know what one can do in Spec# that isn't possible in L42.

Unfortunately, the paper does not provide convincing evidence that the annotations are expressive enough. This is because the paper only contains three case studies and they are all small with only a few invariants. Also, unfortunately, the three provided benchmarks are all synthetic and designed by the authors. More convincing would have been to do the examples previous protocols provided as exemplars. In addition, annotating real code would have made a much more convincing case, but that would have required translation into their language L42.

I also have a concern that the validity of an object relies on invariant methods within classes. This means that users of the class have to dig into the code of the class rather than only needing to understand its interface. Are the guarantees that executing the invariant methods really only available when the producer and the consumer of a class are written by the same programmer? The authors should discuss this.

The authors claim that the use of type modifiers and object capabilities ensures that potential mutation is tamed. Given the importance of type modifiers and object capabilities to L42, a more detailed introduction to them earlier on in the paper would be beneficial. I would also like to see explicitly why both are needed rather than one, and could there be any unintended interference between the two concepts?

On page 13 the authors say that Rust's relaxed support for TMs or OCs would make enforcement of their protocol harder or impossible. Why? Can you expand what the problem is?

Ending on a positive note – I found the examples throughout helped me understand your system and I thought the hamster in a cage staying on its path nice. Having an actual implementation as well as a soundness proof are both appreciated. The paper was straightforward to read.

Review #59B

Overall merit

B. Weak accept: I think this paper should be accepted but I will not champion it

Reviewer expertise

Y. Knowledgeable

Paper summary

This paper addresses the problem of checking representation invariants in object-oriented code. The key goal is to reduce the amount of invariant checks (i.e., calls to repOk/checkRep). The key idea is to combine guarantees of immutability and uniqueness types, particularly the type system by Gordon et al. presented in OOPSLA 2012, as well as certain ownership guarantees, to reduce the number of invariant checks needed to guarantee that the invariant holds. Basically, if there is no representation exposure, then it is safe to forgo dynamic checks on methods that do not mutate the object representation.

The paper compares the approach against two baseline approaches. One approach is to check the invariant at the end of every public method (what the authors call the “visible state semantics”). The second approach is based on the Boogie/Pack-Unpack methodology, which relies on similar immutability and ownership annotations to insert minimal number of invariant checks.

The paper presents several case studies. One case study, GUIs, is non-trivial, and the authors compare their approach to the “visible state semantics” approach, and the Boogie approach. The results show that the proposed new approach entails significantly fewer dynamic checks compared to “visible state semantics”, and while it entails the exact same number of checks as the Boogie approach, it is more useful, as it imposes a lighter annotation and special notation burden. The paper includes static semantics, dynamic semantics and soundness proofs.

Comments for author

Overall, this is a well-written and well-executed paper; it poses a real and difficult problem, and presents a novel solution. The paper is clearly written, with many examples, and just enough formalism, and everything appears sound. I think it will make a nice addition to Ecoop.

I have some minor comments for improvement. One is the use of terminology. I believe the established terminology for “invariant”, which dates back to Liskov and Guttag I believe, is “representation invariant” also known as “rep invariant”. The “invariant” method is usually called “repOK”, or “checkRep”. The “capsule” semantics captures and prevents what I’ve known as “representation exposure”. It took me a while to realize that the problem you define and solve, is the well-known problem of checking rep invariants.

Another minor criticism is that the paper (without considering the appendix) is not self-contained. It relies on certain semantics of reference and object immutability, and ownership, and from what I know of the area, different type systems give different flavors of immutability and ownership. For example, it is unclear to me what the capsule, i.e., ownership semantics is, exactly, and how it relates to the well-known owners-as-dominators, and owners-as-modifiers semantics. For example, what does it mean “immutable references can be freely shared across capsule boundaries”? How does the underlying immutability system guarantee object immutability? For example, suppose we have

imm x = new X();

There are no mut arguments. But constructor X() can still leak implicit parameter “this” either through a static field, or through something like “y = this; x = this; y.f = x;”, thus creating references other than “x” to the immutable object. How does the underlying type system keep track of such references, or prevent their creation?

It was also unclear what the importance of object capabilities is. The gist of the paper is really about how ownership and immutability guarantees can allow to safely forgo lots of dynamic repOK checks. “Object capabilities” appears in the title, suggesting they are somehow essential, yet only a short paragraph discusses them.

Yet another point is that the method presented in this paper, in its essence, is not very different from the Boogie/Spec# methodology, which also relies on ownership and immutability guarantees. The two methods appear to be equivalent, as they do insert the exact same dynamic checks, and the Boogie/Spec# does not appear to rely or object immutability guarantees. (I find object immutability confusing and difficult to reason about.)

So I have a concrete question for the authors: Is my understanding correct? Does Boogie/Spec# rely on object immutability guarantees as well, or does it rely solely on standard method purity and standard owners-as-dominators ownership?. I do agree that the methodology presented in this paper is more programmer-friendly, and therefore, more likely to have impact in practice (provided that the programmer is familiar with the underlying object immutability and ownership semantics).

I have one last minor point. Why do we care for reducing repOK checks (“invariant” checks in your terminology)? I think it is customary to add repOK checks to every method, including pure methods, during development/debugging, then remove all those checks during production, since repOK checks in practice tend to be inherently expensive.

Review #59C

Overall merit

A. Accept: I will champion this paper

Reviewer expertise

Y. Knowledgeable

Paper summary

The paper proposes an approach for relatively efficient run-time checking of class invariants in object-oriented programs. The approach is based on a type system for (some kind of) uniqueness and immutability, and an object capabilities regime for controlling I/O and nondeterminism. The object capabilities regime means global variables and static I/O-performing methods are not allowed and instead, objects encapsulating I/O capabilities must be passed around. In the proposed approach, class invariants are specified as parameterless instance methods called "invariant", whose receiver is typed "read", meaning no mutation (or I/O or nondeterministic operations) can be performed through this reference. Furthermore, a class invariant can use "this" only to read fields of type "imm" or "capsule", holding a reference to an immutable object graph, or the sole external (mutation-allowing)

reference into a mutable object graph. This means that an object's class invariant can be violated only by mutating its own fields or by mutating fields of objects reachable through its capsule fields. The approach inserts run-time checks of an object's class invariant after each update of one of the object's fields and at the end of a call of a capsule mutator method on the object, which is a method that uses its receiver only once, to read a capsule field, and furthermore takes only "imm" and "capsule" arguments (so that the external uniqueness property is preserved).

The authors introduce and motivate the approach informally, provide a formal soundness proof (axiomatizing the properties guaranteed by the type system), and evaluate the usability of the programming model by implementing a small GUI program with an interesting widget hierarchy, where composite widgets have a class invariant and encapsulate an object graph that has cycles between widgets and event handlers for those widgets. They also report on an extensive comparison with Spec#.

Comments for author

Arguments in favor:

+ Verifying class invariants is useful.
+ The approach seems to be technically sound and strikes me as quite elegant.
+ The approach is presented reasonably clearly.
+ The reported case study indicates that the approach may be usable in practice. Arguments against:
- More and larger case studies are necessary in future work to gain assurance of the practical usability of the programming model.
- The "box" and "transformer" patterns, which the programming model essentially imposes, are unusual. Hopefully, it will be possible in future work to reduce the programming overhead.
- The presentation is a bit uneven. In particular, I would have liked a slightly deeper and more rigorous introduction to the type system. Specifically, the typing rules for "capsule" fields are not entirely clear to me. At the same time, some sections belabour the obvious a bit; I found Secs. 5 and 6 unnecessary.

Detailed comments:

- p. 2: when first mentioning Spec#, cite a Spec# reference
- p. 9: "Note that these restrictions do not apply ...": I do not understand this sentence.
- p. 9: Footnote 12: to me, it would make sense that the runtime semantics (in particular: the invariant checking semantics) of checked exceptions would be inconsistent with that of unchecked semantics.
- p. 17: line 684: "cs" -> "c"
- p. 19: "using an inner Box object is a common pattern in static verification": I thought I knew a bit about static verification, but I've never heard of the Box pattern or seen anything like it.
- p. 23: "Dafny ... requires objects to be newly allocated (or cloned) before another object's invariant may depend on it.": I thought Dafny implemented the dynamic frames approach, which is very flexible? An invariant is just a pure function, which can have a "reads" clause mentioning any arbitrary set of memory locations? I don't think Dafny implements the visible state semantics. Invariants are not even built into the language; they are just a pattern of using pure functions in preconditions and postconditions?
- p. 24: The suggestion at the bottom of p. 24, "One interesting avenue", strikes me as a bit silly and impractical. Indeed: a precondition would have to hold throughout the method's execution, which is generally not the case.

English: There are many English errors; most, however, fall into the following two categories:

- Using a comma to join two related sentences instead of a semicolon. For example: p. 2: "may change, this is done" -> "may change; this is done". Similarly: "can next be used, we then inject": "," -> ";". Further, I detected one occurrence of this error on p. 4, one on p. 5, one on p. 7, two on p. 8, three on p. 9, one on p. 10, one on p. 11, one on p. 16, one on p. 17, one on p. 18, one on p. 19, one on p. 20, one on p. 22, three on p. 23, two on p. 24.
- Missing hyphens. For example: in the abstract: "non determinism" -> "non-determinism" (or "nondeterminism") (many occurrences throughout the text); "pre existing" -> "pre-existing". The general rule is that prefixes such as "pre", "post", "re", "sub" and "non", which are not words by themselves, cannot be written as separate words. (This means also: "pre and post conditions" -> "pre- and postconditions", "re establish" -> "re-establish", "sub object" -> "subobject".) Another category of these is "object oriented" (p. 5) -> "object-oriented" and "well formedness" -> "well-formedness". Other:
- p. 1: "complicated with" -> "... by"
- p. 2 and elsewhere: "et. al." -> "et al." (or "\emph{et al.}")
- p. 2: "that that"
- p. 3: "their receivers ROG" -> "... receivers' ..."
- p. 4: "visible state semantic" -> "... semantics"
- p. 4: "and section 4" -> "and Section 4"
- p. 5: "allow statically reasoning" -> "allow reasoning statically" or "allow static reasoning"
- p. 6: "in literature" -> "in the literature"
- p. 9: "the fields ROG" -> "the field's ROG"
- p. 9: "it's invariant" -> "its invariant"
- p. 9: "need check" -> "need to check"
- p. 13: "all Persons fields" -> "all Persons' fields"
- p. 16: "the children ... is a list" -> "the children ... are stored as a list"
- p. 17: "(is" -> "(which is"
- p. 18: "Widgets ... methods" -> "Widget's ..."
- p. 18: "times;in" -> "times; in"
- p. 19: "we had to in L42"
- p. 20: "Gordon" -> "Gordon et al."
- p. 20: "self contained" -> "self-contained"
- p. 20: "leverage on" -> "leverage"
- p. 21: "fine grained", "reference based", "class based": insert hyphen
- p. 21: "objects public methods" -> "objects' ..."
- p. 21: "whose semantic" -> "whose semantics"
- p. 22: "stronger then" -> "... than"
- p. 22: "aspect oriented" -> "aspect-oriented"
- p. 22: "a roll back approach" -> "a rollback approach"
- p. 22: "has broken" -> "has been broken"
- p. 22: "user facing" -> "user-facing"
- p. 23: "real world system" -> "real-world system"
- p. 23: "user defined" -> "user-defined"
- p. 23: "program language" -> "programming language"

- p. 23: "short circuit semantics" -> "short-circuit semantics"
- p. 23: "sets. Whereas" -> "sets, whereas"
- p. 24: "a methods" -> "a method's"
- p. 25: "parsed" -> "passed"

Review #59D

Overall merit

D. Reject: I will strongly argue for rejection of this paper

Reviewer expertise

X. Expert

Paper summary

This paper proposes a new way of using reference capabilities (of the reference immutability variety) to control where dynamic invariant checks are required. The core idea is to constrain invariants to only address immutable and fully-encapsulated (externally-unique) representation state, and use that simplification to guide the (dynamic) injection of monitoring calls checking the invariant of a mutated object.

The paper outlines some of the background material on capabilities, describes dynamic semantics for invariant monitoring based on a mutability control system in the style of Gordon et al./Clebsch et al./Servetto et al.'s work. Some of the general problems with soundly checking invariants any time a change may occur that must be handled by the approach are outlined, along with explanations of how this particular system handles them. One case study is shown (implementing a simple GUI program) to argue that this approach improves on prior approaches in either annotation burden (vs. Spec#) or number of invariant checks required (vs. D and Eiffel).

Positives:

+ Revisiting "old" problems with new technology, which is always interesting
+ Genuinely novel approach to guiding monitor insertion with less explicit guidance than other low-overhead approaches
+ Some attempt to extract general knowledge about idiomatic use (Box, Transform)

Negatives:

- Some parts are not clearly explained, or details of examples contradict explanation "by reference to prior work"
- Concerns about soundness
- Approach seems very restrictive
- Very weak evaluation, that does not seem to support the strength or generality of some claims

Comments for author

I think this paper is tackling a very interesting problem, with an interesting general approach. The family of type systems this builds on has shown some value in a few systems, and being able to get more benefits out of such a system is a great thing. I think something very good can come of this work.

But I have concerns about the limitations of this paper's approach, the quality of the exposition, and find the current evaluation highly incomplete. I also have some mild concerns about soundness of some parts of the system.

Concerns with Approach: Limitations and Costs

This paper advocates checking the invariant of an object after *every single field update*. This seems prohibitively expensive for general software. Avoiding this is a major advantage of the visible state semantics or inhale/exahale: they batch invariant checking in order to do it less often. The (one) case study shows this being competitive with Spec#, but there are issues with that case study (more below). Notably, the semantics add an invariant check after every field write or reciever-mut method call on a class with a capsule field. This can lead to redundant invariant checks (e.g., such a method consists of one field write, which leads to an immediate invariant check, followed by another invariant check from the method call itself).

The motivation for this particular checking protocol is also unclear. The paper is not very explicit about it, but seems to have as a goal that it is not possible to call a method of an object whose invariant does not hold. This is nice in theory, but it's quite restrictive; there's no shortage of code that knowingly violates invariants for short periods before restoring them, and checking invariants there is undesirable. This is well-known in static verification, and is one of the reasons for the "batching" in other contract systems. The paper doesn't address this important concern except for one sentence I don't fully understand (see exposition section).

The system imposes some very severe restrictions, whose impact is never discussed, and which seem to make many desirable things very difficult:

- The restriction that invariants can only access imm and capsule fields is very severe! I see the point: this ensures the state affecting the invariant's truth can be narrowly identified with capsule fields and mutable local pointers to immutable data, making it "easy" to decided when checking is sensible, and reduces the possibility of passing broken receivers around.
  + However, the methodology does not claim to actually prevent any clear class of these pathologies (invariants can still dereference null by passing something inappropriate to a method called in an invariant, for example), and no evidence is given that this type of pathology is actually a serious problem in practice (to the degree that a vague "reduction in likelihood" would be worth striving for).
  + Section 6 sort of touches on a justification for this, but really only shows that it is one way of solving a representation exposure problem, not that it's really necessary or not-problematic. This paper is using reference immutability: why not use that in some way to control rep exposure?
- Requiring that "capsule mutators" only use "this" once is extremely restrictive. For example, this seems to make it hard to implement a binary tree with a sortedness invariant in this system. If one used mut fields for the children, they couldn't be part of an invariant (such as sortedness). But if one used capsule for the children in order to specify the invariant, many binary tree operations (notably deletion) could simply not be implemented because they require accessing *multiple* (capsule) fields. One possible work-around I could think of is to define the tree with mut fields, then use a separate "wrapper" object with a capsule TreeNode reference that checked sortedness as its invariant. Does this work?

- The restrictions on invariant bodies, as currently described, are a brittle syntactic check on fields accessed by (statically dispatched) receiver methods called from the invariant. That only works as-is in L42 because the prototype currently lacks inheritance! The paper writes this off as a minor fix for the future, but it's not quite so easy, and introduces new headaches like breaking naive separate compilation: if an override of a known method called by an invariant accesses a non-imm/capsule field, this might be easy to check locally by tagging, but if "this" escapes as a parameter to another call, the system must track which fields are transitively accessed by that call... this gets out of hand quickly. (Consider Chandra et al.'s OOPSLA'16 paper on type inference for JavaScript.)

The case study examines a case where the encapsulation is particularly natural. This is always a red flag in an evaluation: if the limitations of a system aren't encountered in an evaluation, why not? Most code in the class of type systems considered uses read and mut much more heavily than those permitted in invariants here; Gordon et al. reported that over half of their annotations were the equivalent of `read` here (it's unclear how much of this was fields vs. parameters).

Exposition

This paper relies on the interaction of a number of non-trivial systems, which is always a difficult sort of system to explain. This paper does fairly well, but falls short in a few critical places. The general explanation of "type modifiers" is adequately clear, at least to someone with background in that space (though note discrepancies between that space and this system below). But the other areas of background: protocols for invariant checking and the details of strong exception safety were both inadequately explained. I still do not understand the claims about how SES ensures an object with a violated invariant is garbage when the exception is caught. For that matter, I don't fully understand the explanation of what SES is/provides. The protocols for invariant checking are explained a bit better, but never explained as a whole even though the paper is peppered with references to specific details.

The exposition of the main system itself feels hand-wavy and incomplete. The "Invariants" paragraph on page 9 says "this" can only be used to access capsule and imm fields. This suggests to me that method calls on those fields would be disallowed, but later examples make use of them. If the static checking were written down somewhere formally I could check this there, but most of the system's checks are specified only in English, so I can only guess whether this is actually meant to be included, or an implicit L42 extension.

The immediately following discussion of Capsule Mutators is similarly imprecise. On lines 370-372, how does this allow an invariant to be violated for a non-trivial period of time before being checked again? On lines 372-374, why would it otherwise be possible to "leak out a mut alias to the capsule field?" Even doing this locally violates the soundness semantics for every reference immutability system I've seen, including Gordon et al.'s system, Clebsch et al.'s Pony, and earlier L42 papers' type systems. In particular: if a mut alias is created to a capsule field, what then stops the next few bits of the program from applying recovery to the capsule field itself to make it immutable, thereby leaving a mut alias to an imm object graph?

Later the paper allows read aliases to the internals of capsule data, which is less obviously problematic in the absence of parallelism, though still inconsistent with the systems I've seen. If this paper is working with a type modifier system that differs in such important ways from other published approaches, that needs to be explicitly explained in this paper. If it's relying on a published variant with these properties, the paper should be clear that these differences exist (currently the writing lumps L42, Pony, and the Microsoft system together as roughly the same, which is mostly correct aside from these aliases to capsule data), and point at exactly where further details can be found.

On lines 374-376: I just don't understand what this sentence is suggesting.

Ultimately, I feel like I still do not fully understand all of the restrictions. Part of this is that everything is presented in English, even subtle technical points. Part of it is also that the paper cycles through the system there are proofs for, L42's own relaxations, and further speculated relaxations. These are all fine to have in the paper, but cycling between them makes it difficult to keep straight what's actually part of the proof, what's implemented, and what's speculative. I'd move the speculation to the very end of the paper, and include a brief overview of the gap between L42 and the formal system just before the evaluation. Localizing those deltas would make it easier to consider the variations separately.

Soundness

Beyond my questions above about aliasing of capsule data:

- Well-Formedness criteria in Section 4 *assume* an important correctness property, rather than proving it is true. Specifically, the paper restricts the grammar of expressions such that in $try^{\sigma}\{...M(l;\_;\_)...\}\_, l \notin \sigma$. This shouldn't be a syntactic restriction. I can see why this is desirable: it means if an invariant fails, it's for an object that's not visible outside the try, so this is presumably related to SES. But I see no reason why this makes sense to assume true of the syntax. The runtime semantics in Figure 2 would have no issue reducing $\sigma; try(l.f = v)catch\_$ to $\sigma[l.f = v]; try^{\sigma}(M(l;l;l.invariant()))catch\_$ and violating this assumption. So this *cannot* be a syntactic restriction. It must be a property *proven* true of program executions under the specific type system. This isn't something that would just follow from from of the SES work, either (at least not the papers cited specifically for SES), because it relates to the monitoring new to this work.

Case Study Issues

The case study is very narrow: one program, which happens to avoid encountering the many restrictions of the approach. Even then, based on the description of the paper and familiarity with prior work, I'm not sure the example in the paper should type check.

In particular, I'm looking at children() method of the "Java-ized" L42 code on page 17. The children() method returns state from *inside* capsule box. As noted earlier, this should be type-incorrect in all of the related systems cited, to the best of my knowledge. Those systems would reject line 678, because as-is there's nothing stopping a caller of `children()` from storing that `read` result in the heap somewhere, leading to a violation of encapsulation. I checked the implementation, which allows the same. This is the same code that was apparently difficult to implement in Spec#. Is there a bug in the implementation? If not, why is this permitted? This is related to the speculation on page 10 that read-only access to capsule fields is permissible, which is not the case in the prior work I'm aware of, but perhaps L42 has made different choices (which then should be explained here, given all the comparisons to Gordon et al. and Pony). This doesn't invalidate recovery to immutable, but means the system can no longer be used for safe concurrency; fine if L42 has none, but again if this is deviating from prior work this must be discussed, and if it's relying on the details of a particular

system, some detail of that specific system/paper should be given to contrast with the broad-stokes view of Gordon et al./Clebsch et al./Servetto et al.

Note: this example also seems to run afoul of the speculation on page 10 that allowing public fields can work, as long as capsule fields are never accessed except directly from the receiver. Line 685 appears to do exactly that problematic thing.

The evidence from this case study is not very strong. It certainly shows feasibility, which is good. But the D and Eiffel approaches are known to have high overhead. The L42 prototype performs the same number of invariant checks as the Spec# version, which is good. But it's not clear how good.

Certainly the check locations fall out of using a simpler type system in L42. But since L42 also injects checks after literally every field write, a block of code that writes to n fields of the same object back to back can lead to n invariant checks. But Spec# can (with user guidance) check only once, after all updates. This is partly due to intended differences in granularity of invariant checking, but again that difference is never motivated. It may be that this protocol is simply the result of taking a qualifier-driven approach like this, which is fine, but there should still be an argument that this is sensible. At the same time: would a different annotation of the Spec# version lead to soundly checking the invariant with fewer invocations of the invariant method?

The case study considers 1 execution of 1 program written by the authors of the technique. This is a very weak evaluation. Consider my earlier question about the granularity of invariant checking: would a different program or a variation of this one run into problems where an invariant check fired too early, such as moving two adjacent pieces "simultaneously" in a simulation step. If they were programmatically moved one by one, this technique might inject invariant checks when they were overlapping, but only because one had been moved towards the other, but the other had not yet been moved out of the way *for the same time step*. This seems much more likely with this kind of protocol than with the earlier approaches. Maybe this is okay, but one example is not enough to support that.

A proper evaluation would give a broader view of more algorithms, with discussion of the trade-offs involved. For example: there's a fair amount of Spec# code, why not port some to L42? This would also relieve general concerns of bias from only comparing on code L42 was known to be reasonably good at. Similarly, just covering a range of structures and drawing inferences from those conversions would be good. What about binary trees? A stronger evaluation for this work would have more discussion of idioms like Box and Transform, but based on a more diverse and less biased set of programs. I think asking for hundreds of examples or tens of thousands of lines of code is clearly too much for a paper about a new idea, but this one program is not enough to convince me.

Overall

Overall, I think this paper is on to some very cool ideas, and down the line I'd like very much to see an improved version of this. But given that I have concerns about the soundness of the proposed technique, serious doubts about its applicability (the impact of the many restrictions) and concerns about the case study, I cannot recommend acceptance of this paper in its current form.

My strong reject is fairly firm, mostly on the grounds of not fully understanding the system's restrictions despite trying to read very closely, lack of discussion of those restrictions, and a very incomplete evaluation. The evaluation can't be fixed at this stage, but some things that might soften my stance:

- convince me the restrictions are not so severe or I have misunderstood them
- explain why the children call in the GUI case study type checks, and what this discussion about mut aliases to capsule references is about
- convince me there's more generality to the GUI case study than I'm seeing
- convince me that the details of the static reasoning bits that lie outside "established" work on reference immutability (e.g., the syntactic checks done for invariant bodies) can be both clearly explained and extended cleanly to a system with real inheritance hierarchies and similar extensions discussed in the paper.

Other General Concerns

I'm not usually one to quibble over terminology, but this use of "Type Modifiers" terminology is problematic. The paper defines "type modifiers" to mean what prior work called "reference capabilities." Arguing for change in terminology when appropriate can be a very good thing, but generally breeds confusion unless there's a good reason for it. This paper doesn't actually provide such a reason: it merely claims in a footnote that the new terminology is to avoid confusion between reference capabilities and object capabilities. Why is this even a risk? Reference capabilities are a static variant of object capabilities where instead of all references to an object granting the same abilities to modify the object, different references grant different abilities. Prior work seemed to explain this clearly, so this seems a weak reason to sow confusion by breaking with prior work's language. At the same time, the particular choice "type modifiers" sounds much more general than "reference capabilities," to the point where I assumed initially it was a stylistic variant of "type qualifiers.

Miscellaneous Comments

- According to Gordon's dissertation (Ch. 3.5.5), the Microsoft system also was used for enforcing purity of design-by-contract constructs, though they do not specify the checking protocol. (This is apparently only in his dissertation, not [38].)
- page 13: The definition of "trusted" is missing details on the inner evaluation context in the second case, which presumably needs to have a hole somewhere
- Sections 5 and 6 contain good discussion that should come earlier, but are also full of details that really belong in an appendix; I'd move some material earlier, some of the pickier bits to the appendices, and use the extra space to improve the explanation of the background material.
- page 15: This discussion of shipping lists and items explains a problem that this system's restrictions solve, but it really needs to argue for why this solution is *good*, not that it just works. Fundamentally, the first example in Section 6 is about rep exposure; why not use reference immuatbility to address that too? (I'm not saying it's easy to do so, only that it should be discussed, given that reference immutability was proposed in part to address just that issue.)
- page 16: allowing 'this' to be used more than once without restriction does cause problems, but are there no intermediate restrictions that work? This one is drastic.
- page 19: Counting tokens and characters in annotations is not useful, and is not something we should be counting. Please remove those, which would set a bad precedent for others. The annotation counts and categories of course are fine.
- page 28: References 45 and 46 are the same paper.

Response 0 words

+ Add Response

HotCRP