#459 The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency

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

PC conflicts

Ori Lahav
Radu Grigore

▼ Other conflicts

All (Depaul University)
All (HSE University)
All (University of Kent)
Anouk Paradis (ETH Zurich)
Brijesh Dongol (Surrey)
Marco Paviotti (Imperial)
Mikolás Janota (University of Lisbon)
Peter Sewell (Cambridge)
Philippa Gardner (Imperial)
Radu Grigore (Facebook)
Scott Owens (Facebook)
Stephen Kell (Kings College London)
Christopher Pulte (University of Cambridge)
Chung-Kil Hur
Conrad Watt (University of Cambridge)
Egor Namakonov (St. Petersburg University)
Evgenii Moiseenko (St. Petersburg University)
Guillaume Barbier
Jean Pichon-Pharabod (University of Cambridge)
Jonas Oberhauser (Huawei)
Minki Cho
Orestis Melkonian
Ori Lahav (Tel Avi University)
Shaked Flur (Google)
Shu-yu Guo
Soham Chakraborty
Stephen Dolan
Sung-Hwan Lee
Viktor Vafeiadis (MPI-SWS)

Submitted

Submission (544kB) 8 Jul 2021 5:38:12am AoE · 3d22a651

▼ Abstract

Program logics and semantics tell us that when executing $(S_1; S_2)$ starting in state s_0, we execute S_1 in s_0 to arrive at s_1, then execute S_2 in

▶ Authors (blind)

A. Jeffrey, J. Riely, M. Batty, S. Cooksey, A. Podkopaev [details]

In-person participation

s_1 to arrive at the final state s_2. This is, of course, an abstraction. Processors execute instructions out of order, due to pipelines and caches, and compilers reorder programs even more dramatically. All of this reordering is meant to be unobservable in single-threaded code, but is observable in multi-threaded code. A formal attempt to understand the resulting mess is known as a ``relaxed memory model.'' The relaxed memory models that have been proposed to date either fail to address sequential composition directly, overly restrict processors and compilers, or permit nonsense thin-air behaviors which are unobservable in practice.

To support sequential composition while targeting modern hardware, we propose using preconditions and families of predicate transformers. When composing $(S_1; S_2)$, the predicate transformers used to validate the preconditions of events in S_2 are chosen based on the semantic dependencies from events in S_1 to events in S_2. We apply this approach to two existing memory models: ``Modular Relaxed Dependencies'' for C11 and ``Pomsets with Preconditions.''

Yes, I will attend.

▼ **Topic**

Concurrency - Shared memory

	OveMer	RevExp
Review #459A	C	Y
Review #459B	C	X
Review #459C	B	X

1 Comment: *AuthRes Response* (J. Riely)

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

Edit submission **Edit AuthRes response**

Reviews and comments in plain text

Review #459A

Overall merit

C. Weak Reject

Reviewer expertise

Y. Knowledgeable

Paper summary

This paper proposes pomsets with predicate transformers (PwT), a new denotational model for relaxed-memory concurrency. PwT is built on pomsets with preconditions (PwP) proposed by Jagadeesan et al [OOPSLA 2020]. The key feature of PwT is that the denotation of sequential composition (S_1; S_2) is computed from the denotations of S_1 and S_2. The paper presents two versions of PwT for multicopy-atomic (MCA) architectures such as Arm8, and a version for C11. There is also a tool to automatically check whether an outcome is allowed for a litmus test.

Strengths

1) In PwT, the denotational semantics of sequential composition is compositional and associative. This looks beautiful as a denotational model.

2) PwT has been applied to both MCA and C11 models.

Weaknesses

1) The technical challenge is not clearly explained. Why is it challenging to give a denotational semantics to sequential composition?

2) It is unclear how to know the "correctness" of PwT. Appendix A.2 mentions that PwP has some errors. I'm wondering how the authors ensure that PwT does not have errors.

3) The paper is a bit dense. It lists a lot of definitions, but does not give much explanation on the intuition.

Comments for author

1) It is difficult for me to understand the challenges in giving a proper denotational semantics to sequential composition. The paper says that in earlier works [Paviotti et al 2020, Jagadeesan et al 2020], "adding an event requires perfect knowledge of the future" (l63), but it is unclear to me why they *require* knowledge of the future. In other words, if they do not have perfect knowledge of the future, what will be wrong?

   Sec 2.3 explains the reasons why [Jagadeesan et al 2020] requires knowledge of the future as folows:

   > their model uses prefixing, which requires that the model is built from right to left

   But it is still unclear to me why it is challenging to drop the requirement of "prefixing".

   Sec 2.3 also shows an example:

> For example, Jagadeesan et al. state the equivalence allowing reordering independent writes as follows, [[x := M; y := N; S]] = [[y := N; x := M; S]] if x <> y

But it is unclear to me why S cannot be removed. Apparently, S can be arbitrary in this equivalence (since there are no constraints on S), and hence removing S should not affect the equivalence.

Sec 3 says that [Paviotti et al 2020] gives the semantics of sequential composition in continuation passing style. Why is it challenging to turn the semantics to a "direct style"?

Besides, I'm wondering why adding predicate transformers to PwP is crucial for addressing the challenges. Is the idea of predicate transformers also necessary for defining prefixing in PwP? If not, what makes the idea of predicate transformers necessary for general sequential composition? Why can the approach of predicate transformers address the challenges?

Due to the above issues, I am not convinced that this work makes a strong theoretical contribution.

2) To enhance the confidence in the correctness of a new model, in addition to testing, one may also want to formally prove the equivalence between the new model and the existing ones. Appendix A.2 says that PwP has some errors, so it seems that PwT is not a conservative extension of PwP. That is, [[S]]_PwT may not be equivalent to [[S]]_PwP for some program S written in the language of PwP. However, I am still wondering about a formal relationship between PwT and PwP. It is even unclear to me whether the set of events and the preconditions in [[S]]_PwT are the same as those in [[S]]_PwP. Is there any theorem formally showing the relationship between PwT and PwP?

   From Fig 1, I see that the precondition of an event in S_2 might be changed when S_2 is sequentially after some S_1 (see (s3b)). In order to know the precondition of an event e, one has to know the predicate transformer for the events "before" e. Does this mean that the precondition of an event cannot be locally determined? Then, what is the meaning of the precondition of an event in PwT? Is its meaning the same as in PwP?

   Besides, I am also wondering how PwT-C11 is related to MRD-C11 [Paviotti et al 2020]. Is there any theorem formally showing the equivalence between PwT-C11 and MRD-C11?

3) Presentation issues and typos:

- l377. It seems that \tau^{\downarrow e} has the type \Phi -> \Phi, the same as \tau in Definition 4.2. This is a bit confusing, since the type of the symbol \tau in \tau^{\downarrow e} is no longer \Phi -> \Phi. Please clarify. In addition, what is d < e?

- Fig 1. Please explain why (R4b) is defined in this way. In addition, why can the set E of events be empty for read and write operations?

- l534. Should (W5) be (W4)?
- l685. Why should associativity hold for incomplete pomsets?
- It is difficult to understand the last paragraph of Sec 4. The paragraph says that the attempt to define predicate transformers using substitution fails because (\forall r)\psi does not distribute through disjunction. I don't understand how substitution is related to (\forall r)\psi.
- l702. What does "optimal lowering" mean?
- It is difficult to understand Definition 7.1. In particular, please explain the intuition of \pi, e.g. what it represents, why it is necessary, and why it is defined in this way.
- l952. I didn't see the definition of PwT-C11. How is it related to PwT-PO?
- l1223. is ubiquitous is -> is ubiquitous in

Review #459B

Overall merit
C. Weak Reject

Reviewer expertise
X. Expert

Paper summary
This paper proposes a relaxed memory model that supports sequential composition while flexible enough for hardware design. It combines Pomsets with families of predicate transformers. The authors demonstrate how their methods can be used on various MCA hardware models and C11. The authors develop a prototype tool to evaluate litmus tests.

Strengths
- The roadmap of this work, Pomsets with Preconditions + predicate transformer sets, separates itself from most existing relaxed memory models and can have a big impact if proved useful.
- This work contributes to the important discussion of fixing OOTA behavior in the C11 memory model.

Weaknesses
- The paper is difficult to approach due to relatively poor presentation.
- I am doubtful whether the proposed memory model can work beyond simple litmus tests and whether it is practical in real hardware and compiler design.

Comments for author

This work is built on the idea that, according to Jagadeesan et al. [2020], “logic is better than syntax to capture dependencies”. This is a very interesting direction. My major concern is that syntactic dependencies are easier to analyze for compilers and easier to implement for processors. On the other hand, to analyze logical/arithmetic dependencies, more information has to be tracked along execution paths, and operators like remainder can pose a challenge. By proposing predicate transformers tuned for relaxed memory, this work alleviates the problem but still does not solve it.

I would be more convinced if the authors can test their model on slightly larger pieces of code beyond litmus tests. For example, Promising-ARM/RISC-V [Pulte et al. 2019] is evaluated on a concurrent queue implemented in 215 lines of C++ code. Does PwT-c11 work at this scale?

In terms of paper writing, Section 4 and 5 are too dense with definitions and proofs. I get lost multiple times. I would suggest in Section 2 you give one example concurrent program, one valid behavior, and one invalid behavior. Then use your methods to run through the example and show why they are valid/invalid.

If the problems can be solved, I think this can be good work. I skipped some definitions and all proofs, but they look reasonable. PwT-C11 looks like a better solution than RC11 for OOTA behaviors. This can potentially make a real impact.

Detailed comments:

- Section 2.2, the authors should point out all memory locations are initialized with zero. Otherwise, the examples (especially out-of-thin-air) do not hold.
- Page 3 bottom, the blue arrow is not defined. Same with the dashed line on Page 5.
- Please define “pomsets” in the Overview.
- The links in Sections 4 and 5 pose a lot of pain. For example, at Line 552, one has to scroll back and forth twice. This makes hard-to-understand texts even harder. Eventually, I gave up on the formal model and read exclusively the later examples, which, thanks to the authors, are mostly understandable.
- Page 11, “In a complete pomset, c3 requires that every precondition k(e) is a tautology.” This is crucial in understanding how your model works and should be stated much earlier.
- Section 4.6, the authors should emphasize that example separates your semantic dependency model from the syntactic dependency models widely used today.
- What is π^{-1} in Lemma 5.3?
- Section 8 is extremely short. Have the authors tested PwTer on any benchmarks? How many programs have been tested? I would like to see some runtime numbers here, especially for larger code. I think the authors would agree their memory model is more

complicated than most existing ones. So some experiments are needed to ensure the overhead is acceptable.

- In the Conclusion, “efficiently compiled to modern CPUs” is not demonstrated in the paper.

Review #459C

Overall merit
B. Weak Accept

Reviewer expertise
X. Expert

Paper summary
This paper presents a denotational semantics for weak-memory concurrency that supports *both* of the following things for the first time: (1) semantic dependencies (rather than just syntactic overapproximations), and (2) sequential composition (where previous works that supported (1) only managed "prefixing").

The semantics is based on "pomsets with predicate transformers". Executions are represented as pomsets, whose vertices represent events. Each vertex is associated with a precondition; these preconditions are discharged by earlier read events as the executions are "built up"; an execution for a complete program should have "true" for all of its preconditions. Each vertex is also associated with a set of predicate transformers. When executions are sequentially composed, the appropriate predicate transformer is picked according to which events are causal precedents.

The semantics is instantiated to two versions of Armv8 with slightly different tradeoffs in terms of which compiler optimisations are validated, and also a version for C11. A prototype tool for exploring the model will be open-sourced upon publication.

Strengths
- Skillfully and engagingly written paper. Barely any typos.
- The model is quite complicated but also kinda makes sense. It's not obvious to me that it could be much simpler.
- Very much on topic for POPL -- this is very much a paper that engages with *principles* of programming languages.

Weaknesses
- The key motivator for the work is to extend "pomsets with preconditions" (Jagadeesan et al. OOPSLA 2020) from "prefixing" to general sequential composition. The advantage of sequential composition is that equivalences like `[x:=M;y:=N;S] = [y:=N;x:=M;S]` can become `[x:=M;y:=N] = [y:=N;x:=M]`. My worry is: is this a big enough "win" to be worth all this effort?

- I felt that there were a few gaps in the explanations, which I've tried to point out below. These are easily fixed though.
- Not mechanised in a theorem prover, which isn't a disaster on its own... but I did notice that the appendix refers to an error made in a similar recent paper that also wasn't mechanised (Jagadeesan et al. OOPSLA 2020) where a stated result doesn't actually hold. So I think it's fair to say that confidence in the results would be increased if the model were mechanised.

Comments for author

This is a very appealing paper and I enjoyed reading it. Here are some remarks...

1: The title is quite amusing, with the play on "leaky colon".

85: "Introductory programmers" doesn't make sense. (Perhaps you are thinking of "introductory programming courses"?) I suggest "Novice".

126: I think "extends naturally" would read slightly better.

132: I think it would be worth clarifying that this execution is just one of many possible executions. (This is in contrast to the program on line 123 which only has one execution.)

145: The blue arrow hasn't been explained.

169: The parentheses are italicised but their contents aren't -- looks odd.

170: Around here I think it would be good to have a couple of sentences giving the intuition about how these preconditions work. I mean, the reader may be trying to imagine "executing" one of your pomsets -- so, what do I do when I meet a precondition? If the precondition is false do I come back to that event later or just throw it away?

171: You mention data dependencies and control dependencies here; am I to understand that the example on line 174 has both a data dependency (r*s = 0) *and* a control dependency (s < 1)? If so, would be good to clarify.

179: There's a case for making the vertical bar fill the entire height of the TikZ node. It would help clarify the precedence between `==>` and `|`, for instance.

203: "pomset" -> "vertex in the pomset" ?

240: "sanity check" is a slightly old-fashioned term these days.

244: Would help if this graph used the same layout as the one on line 238.

284: "thin-air free" -> "thin-air-free"

316: You want a `\citet` here. (And in a few other places too.)

319: "syntax sugar" -> "syntactic sugar"

327: I suggest dropping the $= \{...\}$ here because it's really weird until you read the appendix and realise that you're mapping events to registers. Just say that there's a set of registers that do not appear in programs.

380: I wasn't sure why it's $D \subseteq \mathcal{E}$ here not $D \subseteq E$.

381: I think abbreviating τ^E as τ is a bad idea. Yes you save a bit of ink but the confusion is a high price to pay -- see how (M4) gives the type of τ and then (M5a) immediately seems to violate that type because it uses the abbreviation.

402: Remind the reader here that you're not including parallel composition and fences.

407: I struggled with the concept of a "termination condition". Can you add a sentence or two of intuition here? I wonder: what if you renamed it to "completion condition"? Once it's met, the pomset is deemed "complete". That would unify your terminology a bit, right? By the way, I note that PwP doesn't use these termination conditions, and there's no explanation in the paper of why this part of the model has changed, even in the appendix -- could you add it?

423: Item (c) has been lost. And items (i) and (j) could be combined into a single line if you want.

Fig 1: I reckon you could probably merge (S3a) and (S3b) and (S3c) together. If I read it right, the only reason for splitting them up is to make sure that you don't call κ_1 or κ_2 outside of their domains of definition. But if you just arrange that $\kappa(e)$ is $false$ if e is not in E, then everything should just work, no? Then you can just have (S3): $\kappa(e) = \kappa_1(e) \vee \kappa_2(e)$, I think?

494: This seems wrong: you're saying that [x:=1] is not a non-strict superset of [if(M){x:=1}]. But they're surely equal if M is true?

500: "be become" -> "become"

515: "true" -> "taken" ("true branch" connotes "then branch" for me)

547: "example" -> "example,"

614: "CPU" doesn't really need small-caps.

620: Notation suggestion: how about $<e$ instead of $\downarrow e$? That would be consistent with line 354.

670: Dodgy kerning around the bang.

700: Not sure you defined the "MCA" acronym.

701: You might clarify what "lowering" means.

705: This could be clearer about how the models are different. *How* exactly are the models different when a thread reads its own write?

722: (P4) looks weird to me. Why doesn't τ_2 feature here?

817: Should c' be c?

852: "It's" -> "Its"

1163: I'm not too sure what "If-closure" means exactly. Could you clarify?

AuthRes Response

Author [James Riely] 7 Sep 2021 **1118 words**

We are thankful to reviewers for their work.

First, let us address one of the main concerns of referees A and C:

Coq formalization

Since submission we have formalized this work in Coq. This development is available for referees at https://fpl.cs.depaul.edu/jriely/PwT.zip (approximately 11,000 lines of code). In particular, we have formally verified lemma 4.5, which states that SEQ is associative (SeqAssoc.v) and that SKIP is left and right unit (SeqSkipId.v).

Overall comments

We reiterate the broader context of our work: Thin air is a problem with the semantics of all optimized concurrent languages. There are only a handful of solutions and all have deficiencies. We focus on denotations with semantic dependency (SDEP) and sequential composition for the following reasons:

- Using an SDEP relation is the only approach compatible with the existing C++ concurrency definition.
- Sequential composition is necessary for scalable reasoning about big chunks of code. No existing model captures it.
- Existing models do not get at the essence of the problem. By analogy: one can use Landin's SECD machine to understand iteration, but Hoare logic and Scott–Strachey semantics provide much more insight.

Reviewers A and C. Why is a denotational semantics for sequential composition challenging and worthwhile?

Both referees asked about the motivation for a compositional model of sequential composition, why not consider S1 and S2 equal if, for every continuation S0, we have [[S1 ; S0]] = [[S2 ; S0]]? The problem is that this requires a quantification over all continuations S0. This quantification is problematic, both from a theoretical point of view (the syntax of programs is now mentioned in the

definition of the semantics) and in practice (tools cannot quantify over infinite sets. This is a related problem to contextual equivalence, full abstraction and the CIU theorem.

In addition, referee C asks: *My worry is: is this a big enough "win" to be worth all this effort?* We would argue yes, that having a model behind peephole compiler optimizations is worth it, and that this requires a compositional treatment of sequential composition.

To determine whether a dependency is present in some fragment of code, MRD and PwP require that all of following code is evaluated. Suppose that you are writing system call code and you wish to know if you can reorder a couple of statements. Using MRD or PwP, you cannot tell whether this is possible without having the calling code. This is what we mean when we say that MRD and PwP require perfect knowledge of the future.

We have attempted to capture semantic dependencies in a meaningful way that admits sequential composition. PwP/MRD only got halfway there: No one would have ever been happy with Hoare logic if it failed to capture the meaning of sequential composition.

With PwT, the presence or absence of a dependency can be understood in isolation. In practice, this enables future applications where PwT can be used to modularly validate assumptions about program dependencies in larger blocks of code incrementally -- rather than the approach of MRD/PwP where evaluation must be done totally.

Further, PwT-C11 is only the second semantics to interoperate with C++ through a semantic dependency relation, and the first one to be fully compositional. Semantic dependency is a worthwhile goal: a restriction of $acyclic(SDEP \cup RF)$ is a statement which is compatible with the existing C++ standard, subject to a good definition of SDEP. With the exception of MRD, other thin-air free programming language memory models do not distil dependencies down to a relation compatible with the existing C++ standard.

Reviewers A and C. Correctness of the model, relating models.

As noted above, the work has been formalized in Coq.

We agree that a new memory model needs to be positioned against existing models. The usual result here is a compilation correctness to hardware memory models. For PwT-MCA, we address this by showing compilation result for Armv8 model (§5).

Comparing software models, however, is unsatisfying: they are all incomparable, i.e., there are examples which are allowed by one/disallowed by the other and vice versa.

Morally, our model sits between the strong models (exemplified by RC11 [Lahav-al:PLDI17]) and the speculative models (exemplified by the promising semantics [Kang-al:POPL17]). As we argue in §1:

- The strong models require too much synchronization.
- The speculative models allow thin air behaviors.

Looking at the details, however, PwT-MCA is incomparable to both RC11 and promising semantics. RC11 allows non-MCA behaviors that PwT-MCA disallows. PwT-MCA has a weaker notion of coherence than the promising semantics.

Some differences reflect an attempt to fix a bug. For example, Weakestmo [Chakraborty-Vafeiadis:POPL19] purposefully disallows *thin-air-like* behaviors of the promising semantics.

Other differences reflect a different balance between allowed optimizations and reasoning principles. There are fundamental conflicts, for example:

- between Common Subexpression Elimination (CSE) and read-read coherence (§D.1)
- between if-introduction (aka, case analysis, if-closure) and java-style final-field semantics. (If-introduction requires that address dependencies and control dependencies are the same. Final-fields require that they be different.)

Reviewer B. Syntactic vs. semantic dependencies and their usage in compilers

Yes, tracking semantic dependencies is intrinsically harder than syntactic ones, and hardware memory models stick with syntax. However, we cannot settle with syntactic dependencies for models of programming languages since common compiler optimizations used in almost all real compilers (including GCC and Clang) may remove syntactic dependencies (unlike semantic ones). That is, a programming language memory model which supports compiler optimizations and disallows OOTA (unlike the C/C++ memory model) has to track semantic dependencies in one way or another. All other proposed solutions to this problem do that (Promising [Kang-al:POPL17], Weakestmo [Chakraborty-Vafeiadis:POPL19], MRD [Paviotti-al:ESOP20], PwP [Jagadeesan-al:OOPSLA20]).

A compiler does not have to use the proposed semantics directly, i.e., for calculating dependencies. Instead, the semantics is meant to be a wrapper that validates some reasonable set of compiler optimizations. A compiler may make more conservative assumptions about dependencies than the semantics. This is explicitly allowed by Lemma 4.8 (augment closure).

Reviewer B. Does the model scale?

> Promising-ARM/RISC-V [Pulte et al. 2019] is evaluated on a concurrent queue implemented in 215 lines of C++ code. Does PwT-C11 work at this scale?

We have not attempted to run PwTer on programs as large as the Michael-Scott queue. In the theory itself, there is no issue, but as [Pulte et al. 2019] point out, exploring such a large combinatorial state space is challenging. The task is simplified in a hardware model, such as Promising-ARM/RISC-V. In a software model such as PwT-C11 or PwT-MCA, the state space is larger, and therefore the task is that much more difficult.

Reviewers A and B: Presentation

We have tried to make §§1-4 readable, as these communicate the basic idea. It is true that §§5-9 are mostly definitions and results, but this seems inevitable given the space constraints.