Generative Operational Semantics for Relaxed Memory Models[⋆]

Radha Jagadeesan, Corin Pitcher, and James Riely

School of Computing, DePaul University

Abstract. The specification of the Java Memory Model (JMM) is phrased in terms of acceptors of execution sequences rather than the standard generative view of operational semantics. This creates a mismatch with language-based techniques, such as simulation arguments and proofs of type safety.

We describe a semantics for the JMM using standard programming language techniques that captures its full expressivity. For data-race-free programs, our model coincides with the JMM. For lockless programs, our model is more expressive than the JMM. The stratification properties required to avoid causality cycles are derived, rather than mandated in the style of the JMM.

The JMM is arguably non-canonical in its treatment of the interaction of data races and locks as it fails to validate roach-motel reorderings and various peephole optimizations. Our model differs from the JMM in these cases. We develop a theory of simulation and use it to validate the legality of the above optimizations in any program context.

1 Introduction

In the context of shared memory imperative programs, Sequential Consistency (SC) (Lamport 1979) enforces a global total order on memory operations that includes the program order of each individual thread in the program. SC may be realized by a traditional interleaving semantics where shared memory is represented as a map from locations to values. It has been observed that SC disables compiler optimizations such as reordering of independent statements. Despite arguments that SC does not impair efficiency (Kamil et al. 2005), this observation and others have motivated a body of work on relaxed memory models; Adve and Gharachorloo (1996) provide a tutorial introduction with detailed bibliography.

A first (conceptual, if not chronological) step in generalizing SC is to consider the Data Race Free (DRF) models. Informally, a program is DRF if no execution of the program leads to a state in which a write happens concurrently with another operation on the same location. A DRF *model* requires that the programmer view of computation coincides with SC for programs that are DRF. The DRF viewpoint is most strongly reflected in languages such as C++, where any program with data races is deemed erroneous, with undefined semantics (Boehm and Adve 2008).

[⋆] We gratefully acknowledge conversations with Alan Jeffrey and referee comments on an earlier version of this paper. Radha Jagadeesan and Corin Pitcher were supported by NSF 0916741. James Riely was supported by NSF Career 0347542.

Such an approach is at odds with the safety requirements of strongly typed languages that permit data races in well defined programs. Conceptually, this motivates the investigation of the Java Memory Model (JMM); see (Manson et al. 2005) for a detailed history. The JMM provides two key guarantees. First, it is a DRF model. Second, it disallows Thin Air Reads (no-TAR). In a configuration with multiple data races the JMM enforces a partial order on the resolution of these data races. Values that are written are justified by an execution of the program, and thus acyclicity of causality is maintained.

The formalization of the JMM is a technical tour-de-force. However, two criticisms are leveled at the JMM. First, the JMM is too complex. While simplicity is admittedly in the eyes of the beholder, some of the technical content of this criticism is that the JMM approach does not generate executions in the sense of traditional (structured) operational semantics (Saraswat 2004). Rather, it provides a means to test whether a given execution sequence is valid by providing criteria to establish the absence of causality cycles in the resolution of data races.

This is particularly problematic for standard tools-of-the-trade that often rely on a generative operational semantics. For example, proofs of type safety usually proceed by showing that each step of the execution of a program maintains the invariants provided in the type system. Similarly, (bi)simulation arguments proceed by showing that if two configurations are related by the candidate relation, and each takes an execution step(s), the resulting configurations are again related by the relation.

Second, the JMM impedes efficiency. As currently formalized, the JMM invalidates a variety of natural optimizations, such as reordering of independent statements (Cenciarelli et al. 2007). Sevcík and Aspinall (2008) show the incompatibility of JMM with roach-motel reordering (moving a read into the scope of a lock), redundant read after read elimination (reusing the results of a valid prior read) and some other peephole optimizations (such as eliminating a write that precedes another write to the same variable). As a result, the hotspot JVM has been non-compliant with the JMM (Sevcík 2008).

To address these issues, we describe a generative structured operational semantics for a concurrent object oriented language with a relaxed memory model. For DRF programs, our model coincides with the JMM. For lockless programs, our model allows every execution permitted by the JMM. Our model also allows executions that are forbidden by the JMM, but which are necessary to validate the peephole optimizations described above, such as redundant read after read elimination. For programs with both locks and data races, our model is better behaved than the JMM, for example, validating roach motel reorderings. Our model coincides with the JMM on the entire suite of causality test cases associated with the JMM (Pugh 2004).

We validate the utility of our operational semantics by establishing a theory of simulation. We use our study of simulation to validate several optimizations, including those mentioned above. Since simulation is a precongruence, our results show the legality of the transformations in any program context.

The rest of the paper is organized as follows. First, we discuss related work, then Section 3 provides an informal introduction to the basic ideas of the paper. The formalism follows in Section 4, with detailed examples in Section 5. We prove the DRF and lockless properties in Section 6. Section 7 defines simulation for a sub-language and shows the validity of some transformations.

2 Related Work

There is extensive research on memory models for hardware architectures, see (Steinke and Nutt 2004), (Luchangco 2001) and (Adve and Gharachorloo 1996) for surveys. This has led to research on (automated) verification of properties of memory models, e.g., see (Sarkar et al. 2009) for x86 and (Hangal et al. 2004) for Sparc TSO.

Our focus in this paper is on specifying the operational semantics for concurrent programming languages. The memory models for OpenMP (Bronevetsky and de Supinski 2007) and UPC (Yelick et al. 2004) deal with languages with weaker typing and pointer arithmetic and focus on synchronization primitives. These models may permit behaviors violating no-TAR (Boehm 2005). Saraswat (2004) provides a framework for operational semantics with relaxed memory models for typed languages. Saraswat et al. (2007) builds on this research and describes a collection of program transformations that are permitted in a relaxed memory model. In contrast to these papers, we capture the full expressiveness of the JMM for lockless programs, even while retaining DRF and no-TAR.

Our program of generative operational semantics using “true-concurrency” methods follows Cenciarelli et al. (2007) and Boudol and Petri (2009). While Cenciarelli et al. (2007) show that all their generated executions are permitted by the JMM, they do not discuss whether their theory is as expressive as the JMM. Boudol and Petri (2009) provide an operational model for write buffers and the ability for concurrent threads to snoop on the values in these buffers; causality test case 16 (Pugh 2004), discussed in Example 5, exemplifies the expressivity that is not captured.

In addition to eloquently articulating a collection of incisive examples, Aspinall and Sevcík (2007, 2008) formalize the Java DRF guarantee using theorem-provers and analyze several natural program transformations. Burckhardt et al. (2008) undertake the ambitious task of verifying concurrent programs in the presence of relaxed memory models, especially those associated with the CLR.

3 An informal introduction to our approach

We illustrate the key ideas underlying our approach using informal examples. We adopt the following notational conventions. Let `x`, `y` and `z` be thread-local variables. Let `f` and `g` be locations on the shared heap. Let `l` be a shared lock. Assume all heap locations and locks are initialized to `0`. Locks are initially free and a lock’s state increments on every action; thus even states are free and odd states are locked. Let `s`, `t` and `u` be thread identifiers. Write `s[`M`]` for the thread with identifier `s`, executing statement M, and write the parallel composition of threads A and B as $A \,|\, B$.

In the SC view, each location in memory remembers only the last write to each location. Therefore an SC execution makes it impossible for `t` to read `2` and then `1` from `f` in the following program.

`s[f=1; f=2; x=f;] | t[y=f; z=f;].` (Program A)

A relaxed memory model, such as the JMM, allows `t` to read `2` then `1` from `f`, even though the values are written by `s` in the reverse order. Rather than viewing memory as a map from locations to values, as in the SC model, we view memory as a sequence of *actions* which denote write and lock events; there are no read actions in our model. The action sequence generated by Program A is `s[f=1] s[f=2]`. A read can be assigned any value that is *visible*. In this case both values written by `s` are visible to the reads in `t`.

The order of statements in a program encodes the *program order* between actions of a single thread. A read can not see all of the values written by its own thread. In Program A, the read of `f` by `s` can only see `2`, since `2` is written after `1` in `s`.

To model compiler and memory hierarchy effects, one may permit dynamic transformations to the action sequence generated by a single thread, as long as this does not introduce new behaviors. For example, it is permitted to rewrite `s[f=1]s[f=2]` to `s[f=2]`, removing the value `1`, which may be visible to concurrent threads. The converse transformation is not sound, however, since it introduces the value `1` out of thin-air.

Due to nondeterminism, the program `s[f=1;]|t[g=1;]` may result in either the sequence `s[f=1]t[g=1]` or the sequence `t[g=1]s[f=1]`. The program `s[f=1;]|t[x=f; g=x;]` may produce `s[f=1]t[g=0]` or `s[f=1]t[g=1]` or `t[g=0]s[f=1]`. However, it can *not* produce `t[g=1]s[f=1]` due to the data dependency between the two threads.

Synchronization makes the program order of a thread visible to other threads, potentially hiding previously visible values. For example, in any execution of the program

```
s[l.acquire(); f=1; f=2; l.release();] |
t[l.acquire(); x=f; y=f; l.release();]
```

the two reads of `f` in `t` must see the same value, and therefore $\mathtt{x} = \mathtt{y}$.

Lock actions must be recorded in the memory, since they affect visibility. We write lock actions as `s[l:j]`, where `j` is an integer indicating the number of previous operations that have been performed on the lock. Thus, an even action corresponds to an `acquire` and an odd action to a `release`. In the example, if `s` executes first, we get the action sequence `s[l:0]s[f=1]s[f=2]s[l:1]t[l:2]t[l:3]`. Lock events in a memory induce a global *synchronization order*, which is used to define visibility.

Speculation. The approach sketched above can mimic the effects of write-buffers, cache-snooping and other non-SC executions. However it is insufficient to validate every behavior allowed by the JMM, such as the following (Manson et al. 2005, Fig 1).

```
s[x=g; f=1;] | t[y=f; g=2;]                (Program B)
```

In any SC execution, at least one of the threads must read `0`. The JMM allows the execution in which `s` reads `2` from `g` and `t` reads `1` from `f`, which can result from reordering independent statements in the two threads due to cache effects or optimization.

To accommodate such executions, we allow the execution to introduce *speculation*. Let A be the original pair of threads in Program B. Speculative execution reduces A to $(\top \Rightarrow A) \mathbin{[\,]} ((\mathtt{s}\langle\mathtt{f{=}1}\rangle\mathtt{t}\langle\mathtt{g{=}2}\rangle) \Rightarrow A)$, The reduction creates two copies of the original process, which are executed in separate universes with separate copies of the state. The left copy is called the *initial* process; the right, the *final* process. As indicated by the notation, the initial process may assume nothing, $\top$, whereas the final branch may assume the speculated writes, $\mathtt{s}\langle\mathtt{f{=}1}\rangle\mathtt{t}\langle\mathtt{g{=}2}\rangle$. A valid execution is one in which every speculation can be *finalized*, and therefore removed. When the speculation is removed, only the final process remains. The initial process is used only to justify the speculation. We rely on angelic nondeterminism to achieve a valid speculation, if possible.

The initial copy of Program B reads `0` in at least one of the threads and generates both writes. The final copy reads the speculative values and also generates both writes. Since the justifying writes are generated in both copies, the speculation can be finalized.

Unconstrained speculation can break both no-TAR and DRF. We constrain speculation so that it is *not self justifying*, but is *initial*, *consistent* and *timely*.

Self justifying computation allows a thread to see its own speculation, violating no-TAR. Consider the program `s[x=f; if(x==1){g=1;} f=1;]`. To produce the write `s[g=1]`, one might speculate `s⟨f=1⟩`. There is a later write which can justify the speculation. Our semantics forbids `s` from seeing its own speculation, however, thus ensuring that the conditional is false and `g` is not written.

Initiality requires that there is a computation that justifies the speculation without depending on the speculation. Consider the program `s[x=f; g=x;] | t[y=g; f=y;]` (Pugh 2004, §4). By speculating `s⟨g=1⟩t⟨f=1⟩`, both threads can read `1`, violating no-TAR. The final process can produce the necessary writes `s[g=1]t[f=1]`, but the initial process can only write `0`. Our semantics prevents the speculation from being finalized.

Consistency requires that the initial and final computations agree on certain actions. It is necessary for DRF. Consider the following program.

```
s[l.acquire(); x=f; if(x==0){f=1;} l.release();] |
t[l.acquire(); y=f; if(y==0){f=2;} l.release();] |
u[l.acquire(); z=f; g=z; l.release();]
```

(Program C)

The program is DRF. In an SC execution, it is not possible that `f` is `1` and `g` is `2` after execution. Using speculation `t⟨f=2⟩`, however, the final process can achieve this result by scheduling order `u`, `s`, `t`, violating DRF. The initial process can produce the necessary write, but to do so it must schedule `t` before `s`. The inconsistent use of locks makes it impossible to finalize the speculation. Following the terminology of Manson et al. (2005), consistency prevents "bait" (in the initial process) and "switch" (in the final process), an intuition made precise in Example 6. Timeliness ensures that a speculation and its justifying write are in the same synchronization context. It is also necessary for DRF. Consider the following program.

```
s[l.acquire(); x=f; f=x+1; g=1; l.release();] |
t[l.acquire(); x=f; f=x+1; g=2; l.release();] |
u[l.acquire(); x=f; f=x+1; y=g; l.release();]
```

(Program D)

Again, the program is DRF. If `s` reads `0` from `f`, `t` reads `1` and `u` reads `2`, then the order of the threads is determined. Clearly it is unacceptable in this case for `u` to read `1` from `g`. In the execution which runs `s`, then speculates `s⟨g=1⟩`, then runs `t` and `u`, the memory after `t` runs is as follows.

```
s[l:0]s[f=1]s[g=1]s[l:1]s⟨g=1⟩t[l:2]t[f=2]t[g=2]t[l:3]
```

The speculation `s⟨g=1⟩` is "too late" with respect to its justifying write `s[g=1]` since the intervening release `s[l:1]` alters the synchronization context. In Section 5 we also discuss speculations which are "too early".

4 The Language

We develop the ideas of the previous section for an object oriented language with lock objects and thread parallelism. We do not explicitly treat volatile variables, final fields and several other features of the JMM. (From the synchronization perspective, a volatile write is similar to a lock release, a volatile read is similar to a lock acquire).

User Language. Let *bt* range over base type names, d over class names (including the reserved class `Lock`), f and g over field names, and m over method names (including the reserved method `start`). Types, T, include base types and classes ($T ::= bt \mid d$). Let $\vec{T}\ \vec{x}$ abbreviate $T_1\ x_1, \dots, T_n\ x_n$. Class declarations, $\mathscr{D}$, are then given as usual ($\mathscr{D} ::= \texttt{class}\ d\{\vec{T}\ \vec{f}\,;\vec{\mathscr{M}}\}$ where $\mathscr{M} ::= T\ m(\vec{T}\ \vec{x})\{M\}$). Fix a set of class declarations satisfying the well-formedness criteria of Igarashi et al. (2001). We assume, as there, an implicit constructor with arguments $\vec{T}\ \vec{f}$ for each $\texttt{class}\ d\{\vec{T}\ \vec{f}\,;\vec{\mathscr{M}}\}$. Define the partial functions *fields* and *mbody* so that $\mathit{fields}(d) = \vec{T}\,\vec{f}$; if the field declarations of d are $\vec{T}\,\vec{f}$; and $\mathit{mbody}(d.m) = \lambda\vec{x}.M$ if class d contains method $T\ m(\vec{T}\ \vec{x})\{M\}$ for some T and $\vec{T}$. The abstraction $\lambda\vec{x}.M$ is written $\lambda.M$ when $\vec{x}$ is the empty sequence. A class d is *runnable* if $\mathit{mbody}(d.\texttt{run}) = \lambda.M$ for some M. Both *fields* and *mbody* are undefined on the reserved class `Lock`.

We assume disjoint sets of base values, $bv \in BV$, variables, x, y, and object names, p, q, s, t, ℓ. Base values include integers and the constants `unit`, `true` and `false`, with operators (such as `==`, `+`, `&&`) ranged over by *op*. Variables include the reserved variable `this`. Each object name p is associated with a unique class $p.\texttt{class}$; a countable number of object names are associated with each class. By convention, we use name metavariables s, t for runnable objects and ℓ for lock objects. For any syntax category, let *fv* return the set of free variables and let *fn* return the set of free names.

A ground value is either an object name or a base value ($v, w, u ::= p \mid bv$). An open value may additionally be a variable ($V, W, U ::= p \mid bv \mid x$). The statement language is given in administrative normal form (Flanagan et al. 1993).

$$\begin{array}{rll}
M,N ::= & \texttt{val}\ x\ \texttt{=}\ \{M\}\ N & \text{(Stack frame statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ \texttt{new}\ d(\vec{V})\,;\ M & \text{(Creation statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ W.m(\vec{V})\,;\ M & \text{(Method statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ op(\vec{V})\,;\ M & \text{(Operator statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ V.f\,;\ M & \text{(Field read statement)}\\
\mid & V.f\ \texttt{=}\ W\,;\ M & \text{(Field write statement)}\\
\mid & \texttt{if}\ (V)\ \{M\}\ \texttt{else}\ \{N\} & \text{(Conditional statement)}\\
\mid & \texttt{return}\ V\,; & \text{(Return statement)}
\end{array}$$

As in Scala (Odersky et al. 2008), we use `val` to introduce local variables without requiring explicit type annotations. To make the examples shorter, we usually drop the `val`. We write $\uparrow V$ for "$\texttt{return}\ V\,;$" and $\uparrow(V, W)$ for "$\texttt{val}\ x\ \texttt{=}\ \texttt{new Pair}(V, W)\,;\ \texttt{return}\ x\,;$", where x is fresh. In examples, we also use complex expressions, use infix notation for operators and drop occurrences of "`return unit;`". Thus, "`y = a+b+c;`" should is sugar for "`val x = +(a,b); val y = +(x,c); return unit;`", where `x` is fresh. We write "$\texttt{val}\ x\ \texttt{=}\ \cdots\,;\ M$" as "$\cdots\,;\ M$" if x does not occur free in M. We write

“$\texttt{if(}V\texttt{)\{val}\ x\ \texttt{=}\cdots\texttt{;}\ M\texttt{\}}\ \texttt{else}\ \texttt{\{}M\texttt{\}}$” as “$\texttt{if (}V\texttt{) \{val}\ x\ \texttt{=}\ \cdots\texttt{;\}}\ M$” if x does not occur free in M; this notation extends to field write statements, conditional statements and sequences of statements in the obvious way.

We expect that stack frame statements do not occur in the user language; they are introduced by the dynamics. The variable x is bound with scope M in all statements of the form “$\texttt{val}\ x\ \texttt{=}\cdots\texttt{;}\ M$”. We identify syntax up to renaming of bound variables and names and write $M\{x := v\}$ for the capture avoiding substitution of v for x in M. We assume similar notation for substitution of names for names and for substitution over other syntax categories.

Actions and processes. Shared locations are assigned values via *actions*. Write, acquire and release actions are *committable* and so may be made visible at top-level. *Speculative* actions are introduced by the dynamics to explore possible future executions; they are not visible at top-level. The general class of actions include the evaluation context action $s[\![-]\!]$, belonging to thread s; this is used later to define justified reads and speculations.

$$\begin{aligned}
\alpha,\beta &::= s[p.f{=}v] \mid s[\ell\colon j] && \text{(Committable action)}\\
\phi,\psi &::= s\langle p.f{=}v\rangle && \text{(Speculative action)}\\
\sigma,\tau &::= \alpha \mid \phi \mid s[\![-]\!] && \text{(Actions)}
\end{aligned}$$

Write and speculative actions identify the writing thread. The write action $s[p.f{=}v]$ indicates a write by s to location $p.f$ with value v. The speculative write $s\langle p.f{=}v\rangle$ allows threads other than s to subsequently read v from location $p.f$.

The meaning of a lock action $s[\ell\colon j]$ depends on the parity of the natural number j. When j is even, the lock is free and the corresponding action is an acquire. When j is odd, the lock is busy and the corresponding action is a release. We write $s[\texttt{acq}\ \ell\colon j]$ to indicate that j is even, and $s[\texttt{rel}\ \ell\colon j]$ to indicate that j is odd.

Let $\mathit{thrd}(\sigma)$ return the unique thread associated with an action. For all actions other than the evaluation context action, define loc to return the location of the action as $\mathit{loc}(s[p.f{=}v]) = \mathit{loc}(s\langle p.f{=}v\rangle) = p.f$ and $\mathit{loc}(s[\ell\colon j]) = \ell$. Similarly, define val as $\mathit{val}(s[p.f{=}v]) = \mathit{val}(s\langle p.f{=}v\rangle) = v :$ and $\mathit{val}(s[\ell\colon j]) = j$. Write actions σ and τ *conflict* if $\mathit{loc}(\sigma) = \mathit{loc}(\tau)$; only two write actions can conflict.

The dynamics is defined using processes.

$$\begin{aligned}
A,B ::=\ & \texttt{free}\ p && \text{(Free object process)}\\
\mid\ & \texttt{runnable}\ p && \text{(Runnable object process)}\\
\mid\ & \texttt{lock}\ \ell\colon j && \text{(Lock process)}\\
\mid\ & s\,\texttt{[}M\texttt{]} && \text{(Thread process)}\\
\mid\ & A \mid B && \text{(Parallel process)}\\
\mid\ & (\nu p)A && \text{(Scope restriction process)}\\
\mid\ & \alpha\, A && \text{(Action process)}\\
\mid\ & \phi\, A && \text{(Guarded process)}\\
\mid\ & \top \Rightarrow A \mathbin{[\,]} \phi \Rightarrow B && \text{(Speculation process)}
\end{aligned}$$

The name p is bound with scope A in the process $(\nu p)A$. We identify processes up to renaming of bound names.

A *top-level* process contains no subterms that are guarded processes, $\phi\, A$, but may contain speculations. In speculation $\top\!\Rightarrow\! A \mathbin{[]} \phi \Rightarrow B$, we refer to A as the *initial* process and to B as the *final* process. We write $\top\!\Rightarrow\! A \mathbin{[]} \phi_1 \cdots \phi_n \Rightarrow B$ as shorthand for

$$\top\!\Rightarrow\! A \mathbin{[]} \phi_1 \Rightarrow (\top\!\Rightarrow\! A \mathbin{[]} \phi_2 \Rightarrow \cdots (\top\!\Rightarrow\! A \mathbin{[]} \phi_n \Rightarrow B)\cdots).$$

An *initial process* has no free names or variables and contains a single thread. Initial processes have the form $(\nu s)\, s\,\texttt{[}M\texttt{]}$.

We assume several well-formedness criteria, which are true of initial processes and preserved by structural order and reduction. Let *def* return the *defined* names of a process; for example $\mathit{def}(\texttt{free}\ p) = \mathit{def}(\texttt{runnable}\ p) = \mathit{def}(p\,\texttt{[}M\texttt{]}) = \mathit{def}(\texttt{lock}\ p\texttt{:}\,j) = \{p\}$. Let $\mathit{lockact}(A)$ return the lock actions in A with thread identifiers removed; for example $\mathit{lockact}(s[\ell\!:\!i]A) = \{[\ell : i]\} \cup \mathit{lockact}(A)$. A process is *well-formed* if (1) in any subprocess $A \,|\, B$, $\mathit{def}(A) \cap \mathit{def}(B) = \emptyset$, (2) in any subprocess $A \,|\, B$, $\mathit{lockact}(A) \cap \mathit{lockact}(B) = \emptyset$, (3) in any action $s[\ell\!:\!i]$, $\ell\texttt{.class} = \texttt{Lock}$, (4) in any action $s\langle p\,.f\texttt{=}v\rangle$ or $s[p\,.f\texttt{=}v]$, $p\texttt{.class} \neq \texttt{Lock}$, and (5) in any subprocess $\top\!\Rightarrow\! A \mathbin{[]} \phi \Rightarrow B$, $\mathit{thrd}(\phi) \in \mathit{def}(A)$. For the remainder of the paper, we consider only well-formed processes.

Evaluation contexts and justified reads. Evaluation contexts are defined as follows.

$$\mathbb{C} ::= [\![-]\!] \mid A \,|\, \mathbb{C} \mid \mathbb{C} \,|\, A \mid (\nu p)\mathbb{C} \mid \alpha\, \mathbb{C} \mid \phi\, \mathbb{C}$$

The name p is *not* bound in evaluation context $(\nu p)\mathbb{C}$. There is no evaluation context for speculation processes; these are treated specially in the semantics.

We define the notion $\mathbb{C}$ *justifies read* $p\,.f\texttt{=}v$ *by* s to mean that context $\mathbb{C}$ contains a visible write $t\,[p\,.f\texttt{=}v]$ or speculation $t'\langle p\,.f\texttt{=}v\rangle$, where $t' \neq s$). The notion $\mathbb{C}$ *justifies speculation* ϕ is defined similarly.

To begin, define $\mathit{act}_s(\mathbb{C})$ to return the sequence of labeled actions occurring before the hole in $\mathbb{C}$.

$$\begin{array}{lll} \mathit{act}_s([\![-]\!]) = s[\![-]\!] & \mathit{act}_s(A \,|\, \mathbb{C}) = \mathit{act}_s(\mathbb{C}) & \mathit{act}_s(\alpha\, \mathbb{C}) = \alpha\, \mathit{act}_s(\mathbb{C}) \\ \mathit{act}_s((\nu q)\mathbb{C}) = \mathit{act}_s(\mathbb{C}) & \mathit{act}_s(\mathbb{C} \,|\, A) = \mathit{act}_s(\mathbb{C}) & \mathit{act}_s(\phi\, \mathbb{C}) = \phi\, \mathit{act}_s(\mathbb{C}) \end{array}$$

Note that it is not possible for the hole to happen before any action. Given action sequence $\vec{\sigma}$ define *program order* $(<^{\vec{\sigma}}_{po})$ and *synchronizes-with* $(<^{\vec{\sigma}}_{sw})$ as follows.

$$\begin{array}{lll} i <^{\vec{\sigma}}_{po} j & \text{iff} & i < j \text{ and } \mathit{thrd}(\sigma_i) = \mathit{thrd}(\sigma_j) \\ i <^{\vec{\sigma}}_{sw} j & \text{iff} & \sigma_i = s[\texttt{rel}\ \ell\!:\!k] \text{ and } \sigma_j = t[\texttt{acq}\ \ell\!:\!k{+}1] \text{for some } s,\, t,\, \ell \text{ and odd } k \end{array}$$

Note that $(<^{\vec{\sigma}}_{sw}) = \emptyset$ if $\vec{\sigma}$ contains no lock actions. Define happens-before order $(<^{\vec{\sigma}}_{hb})$ to be the transitive closure of the union of program order and synchronizes-with.

Definition 1 (Intervening write and justified read). We say that there is *no intervening write between i and k in* $\vec{\sigma}$ if for every j such that σ_j is a write action and $i <^{\vec{\sigma}}_{hb} j <^{\vec{\sigma}}_{hb} k$, we have that $\mathit{loc}(\sigma_j) \neq \mathit{loc}(\sigma_i)$.

Let $\vec{\sigma} = \mathit{act}_s(\mathbb{C})$. Let k be the index of $s[\![-]\!]$ in $\vec{\sigma}$. We say that $\mathbb{C}$ *justifies* read $p\,.f\texttt{=}v$ (by s) if there exists some i, with no intervening write between i and k in $\vec{\sigma}$, such that $\sigma_i = t\,[p\,.f\texttt{=}v]$, for some t (possibly equal to s), or $\sigma_i = t'\langle p\,.f\texttt{=}v\rangle$, for some $t' \neq s$. □

For the purpose of reading, speculations are “transparent” in the sense that they do not obscure the prior writes. Both writes and speculations (of other threads) can be used to justify reads. Only writes can be used to justify speculations.

Definition 2 (Intervening release and justified speculation). We say that there is *no intervening release between i and k in* $\vec{\sigma}$ if for every j such that σ_j is a release action and $i <_{hb}^{\vec{\sigma}} j <_{hb}^{\vec{\sigma}} k$, we have that $\mathit{thrd}(\sigma_j) \neq \mathit{thrd}(\sigma_i)$.

Let $\vec{\sigma} = \mathit{acts}_s(\mathbb{C})$. Let k be the index of $s[\![-]\!]$ in $\vec{\sigma}$. We say that $\mathbb{C}$ *justifies* speculation $s\langle p\,.f{=}v\rangle$ if there exists some i, with no intervening write nor intervening release between i and k in $\vec{\sigma}$, such that $\sigma_i = s[p\,.f{=}v]$. □

The requirement that there be no intervening release between a write and the speculation that it justifies is motivated by Program D (Section 3). Since any synchronization edge originates from a release action, the absence of intervening releases ensures that a speculation and the write justifying it occupy the same position in the synchronization order and the happens-before relation.

Single-threaded action reordering and structural order. We define $\rhd$ as a relation on single-threaded action sequences. That is $\vec{\sigma} \rhd \vec{\tau}$ is defined only if $\mathit{thrd}(\vec{\sigma}) = \mathit{thrd}(\vec{\tau}) = \{s\}$, for some s.

Definition 3. Let $\rhd$ be the least precongruence ($\vec{\sigma}\vec{\tau} \rhd \vec{\sigma}'\vec{\tau}'$ whenever $\vec{\sigma} \rhd \vec{\sigma}'$ and $\vec{\tau} \rhd \vec{\tau}'$) on single-threaded action sequences that satisfies all instances of the following axiom schemata, where $\vec{\sigma} \mathrel{\dot{\bowtie}} \vec{\tau}$ abbreviates the axiom schemata $\vec{\sigma} \rhd \vec{\tau}$ and $\vec{\tau} \rhd \vec{\sigma}$.

(A-NONLOCK) If σ and τ are nonlock actions that do not conflict then $\sigma\tau \mathrel{\dot{\bowtie}} \tau\sigma$.
(A-ACQUIRE) If σ is a write and τ is an acquire then $\sigma\tau \rhd \tau\sigma$.
(A-RELEASE) If σ is a release and τ is a write then $\sigma\tau \rhd \tau\sigma$.
(A-ABSORPTION1) If σ is a write then $\sigma \rhd \sigma\sigma$.
(A-ABSORPTION2) If σ and τ are conflicting writes then $\tau\sigma \rhd \sigma$.
(A-ABSORPTION3) If σ, τ and τ' are conflicting writes then $\tau\tau'\sigma \mathrel{\dot{\bowtie}} \tau'\tau\sigma$. □

If $\vec{\sigma} \rhd \vec{\tau}$ then $\vec{\sigma}$ “simulates” $\vec{\tau}$; that is, all reads permitted by $\vec{\tau}$ are also permitted by $\vec{\sigma}$. This can be viewed as an adaptation of Lea’s (2008) cookbook to our memory actions.

A-NONLOCK allows write actions and speculative actions in the same thread to commute. A-ACQUIRE and A-RELEASE permit enlarging the scope of locks. These rules are necessary to validate roach motel (Example 10). Were we to allow speculations to commute with lock actions, DRF would fail (Example 8).

In an SC model, later writes completely overwrite earlier writes to the same location. The absorption laws reflect approximations that are available in our relaxed memory model. The first rule allows identical writes to be copied. The second rule allows any write to be eliminated when there is a subsequent “protecting” write to the same location. The third rule allows reordering behind a protecting write. Thus, we get:

Lemma 4. *If* $\vec{\sigma}$ *is a single-threaded sequence of write actions then* $\vec{\sigma} \rhd \vec{\sigma}\vec{\sigma}$.

PROOF. Use the first absorption law to make multiple adjacent copies of each action. Use the remaining laws to rearrange them into the required order. □

$$\textsc{(s-par)}\ \frac{}{A\,|\,A' \mathrel{\dot{\equiv}} A'\,|\,A} \qquad \textsc{(s-free)}\ \frac{}{A \mathrel{\dot{\equiv}} A\,|\,(\nu p)\,(\texttt{free}\ p)} \qquad \textsc{(s-nu-nu)}\ \frac{}{(\nu p)\,(\nu p')A \mathrel{\dot{\equiv}} (\nu p')\,(\nu p)A} \qquad \textsc{(s-par-par)}\ \frac{}{B\,|\,(A\,|\,A') \mathrel{\dot{\equiv}} (B\,|\,A)\,|\,A'}$$

$$\textsc{(s-par-prefix)}\ \frac{}{B\,|\,(\sigma A) \geqq \sigma(B\,|\,A)} \qquad \textsc{(s-par-nu)}\ \frac{p \notin \mathit{fn}(B)}{B\,|\,((\nu p)A) \mathrel{\dot{\equiv}} (\nu p)\,(B\,|\,A)} \qquad \textsc{(s-par-speculation)}\ \frac{\mathit{thrd}(\phi) \notin \mathit{def}(B)}{B\,|\,(\top \Rightarrow A \,[\!]\, \phi \Rightarrow A') \geqq \top \Rightarrow (B\,|\,A) \,[\!]\, \phi \Rightarrow (B\,|\,A')}$$

$$\textsc{(s-nu-prefix)}\ \frac{p \notin \mathit{fn}(\sigma)}{(\nu p)\,\sigma A \mathrel{\dot{\equiv}} \sigma(\nu p)A} \qquad \textsc{(s-nu-speculation)}\ \frac{p \notin \mathit{fn}(\phi)}{(\nu p)\,(\top \Rightarrow A \,[\!]\, \phi \Rightarrow A') \mathrel{\dot{\equiv}} \top \Rightarrow ((\nu p)A) \,[\!]\, \phi \Rightarrow ((\nu p)A')}$$

$$\textsc{(s-prefix-prefix)}\ \frac{\sigma\tau \triangleright \tau\sigma}{\sigma\tau A \geqq \tau\sigma A} \qquad \textsc{(s-speculation-prefix)}\ \frac{(\mathit{thrd}(\sigma) \neq \mathit{thrd}(\phi)) \vee (\phi\sigma \triangleright \sigma\phi)}{\top \Rightarrow (\sigma A) \,[\!]\, \phi \Rightarrow (\sigma A') \ \geqq\ \sigma\,(\top \Rightarrow A \,[\!]\, \phi \Rightarrow A')}$$

Fig. 1. Structural order ($A \geqq B$)

We define $A \geqq B$ to be the smallest precongruence on processes that satisfies the axioms in Figure 1 (where $A \mathrel{\dot{\equiv}} B$ abbreviates the two axioms $A \geqq B$ and $B \geqq A$). Many of the rules follow Milner (1991). We discuss the exceptions.

In order to allow speculation about objects that are not yet initialized, we separate object allocation and initialization. The structural rule S-FREE allows object names to be in scope before the corresponding call to the constructor.

The prefix and speculation rules are ordered so that parallel components can go under action prefixes and speculations, but can not come out. S-PAR-PREFIX effectively fixes the order of operations between threads once those operations become visible to other threads. The S-PREFIX-PREFIX and S-SPECULATION-PREFIX rules are induced by the single-threaded commutation rules. In the S-SPECULATION-PREFIX rule, the required order condition on actions holds for all the branches of speculation; so, it is appropriate to think of this as a "forall" speculation rule, in contrast to the "exists" speculation rule in the forthcoming reduction semantics.

The rules do not allow speculations to commute with each other; adding this rule would not affect contextual equivalence, but would affect the (finer) simulation relation introduced later, which is sensitive to the order of speculations.

In the remainder of the paper, let $\equiv$ denote the kernel of $\geqq$.

Reduction. Process reduction is defined as the least relation satisfying the rules and axioms given in Figure 2. $\twoheadrightarrow$ is the reflexive and transitive closure of $(\geqq) \cup (\rightarrow)$.

Again, many of the rules are standard. The built-in operators R-OPERATOR and the conditionals, R-IF-TRUE and R-IF-FALSE, carry no surprises. Method calls are implemented as usual by R-METHOD, R-FRAME and R-RETURN. The assumption of well-formedness guarantees that there is at most one thread for each object s, and therefore R-FRAME introduces no nondeterminism. Frames are deleted when a function returns.

The reserved methods `acquire` and `release` update the shared global counter associated with the appropriate lock object. As in Java, the reserved method `start` starts method `run` under the thread identity of the receiving object. As per Java semantics, this is a synchronization event, which we enforce using a fresh "dummy" lock.

(R-FRAME)

$$\frac{\mathbb{C}\llbracket s\texttt{[}N\texttt{]}\rrbracket \to \mathbb{C}'\llbracket s\texttt{[}N'\texttt{]}\rrbracket}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[val }x\texttt{ = \{}N\texttt{\}}M\texttt{]}\rrbracket \\ \to \mathbb{C}'\llbracket s\texttt{[val }x\texttt{ = \{}N'\texttt{\}}M\texttt{]}\rrbracket\end{array}}$$

(R-RETURN)

$$\frac{}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[val }x\texttt{ = \{return }v\texttt{;\}}M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket s\texttt{[}M\{x{:=}v\}\texttt{]}\rrbracket\end{array}}$$

(R-IF-TRUE)

$$\frac{}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[if (true) \{}M\texttt{\} else \{}N\texttt{\}]}\rrbracket \\ \to \mathbb{C}\llbracket s\texttt{[}M\texttt{]}\rrbracket\end{array}}$$

(R-IF-FALSE)

$$\frac{}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[if (false) \{}M\texttt{\} else \{}N\texttt{\}]}\rrbracket \\ \to \mathbb{C}\llbracket s\texttt{[}N\texttt{]}\rrbracket\end{array}}$$

(R-NEW)

$$\frac{p\texttt{.class} = d \quad \mathit{fields}(d) = \vec{T}\,\vec{f}}{\begin{array}{l}\quad\mathbb{C}\llbracket \texttt{free } p \qquad \texttt{|}\ s\texttt{[val }x\texttt{ = new }d\texttt{(}\vec{v}\texttt{); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \texttt{runnable } p\ \texttt{|}\ s[p\,.\vec{f}{=}\vec{v}]\ s\texttt{[}M\{x{:=}p\}\texttt{]}\rrbracket\end{array}}$$

(R-NEW-LOCK)

$$\frac{}{\begin{array}{l}\quad\mathbb{C}\llbracket \texttt{free } \ell \quad \texttt{|}\ s\texttt{[val }x\texttt{ = new Lock(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \texttt{lock } \ell\texttt{:0}\ \texttt{|}\ s\texttt{[}M\{x{:=}\ell\}\texttt{]}\rrbracket\end{array}}$$

(R-METHOD)

$$\frac{p\texttt{.class} = d \quad \mathit{mbody}(d.m) = \lambda\vec{y}.N}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[val }x\texttt{ = }p\texttt{.}m\texttt{(}\vec{v}\texttt{); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket s\texttt{[val }x\texttt{ = \{}N\{\texttt{this}{:=}p\}\{\vec{y}{:=}\vec{v}\}\texttt{\}}M\texttt{]}\rrbracket\end{array}}$$

(R-OPERATOR)

$$\frac{w \text{ is the result of applying } \mathit{op} \text{ to } \vec{v}}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[val }x\texttt{ = }\mathit{op}\texttt{(}\vec{v}\texttt{); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket s\texttt{[}M\{x{:=}w\}\texttt{]}\rrbracket\end{array}}$$

(R-METHOD-START)

$$\frac{p\texttt{.class} = d \quad \mathit{mbody}(d.\texttt{run}) = \lambda\vec{y}.N}{\begin{array}{l}\quad\mathbb{C}\llbracket \texttt{free } \ell\ \texttt{|}\ \texttt{runnable } t\ \texttt{|}\ s\texttt{[val }x\texttt{ = }t\texttt{.start(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket t[\ell\texttt{:1}]\ t\texttt{[}N\{\texttt{this}{:=}t\}\texttt{]}\ \texttt{|}\ s[\ell\texttt{:0}]\ s\texttt{[}M\{x{:=}\texttt{unit}\}\texttt{]}\rrbracket\end{array}}$$

(R-METHOD-ACQUIRE)

$$\frac{j \text{ is even}}{\begin{array}{l}\quad\mathbb{C}\llbracket \texttt{lock } \ell\texttt{:}\,j \qquad \texttt{|}\ s\texttt{[val }x\texttt{ = }\ell\texttt{.acquire(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \texttt{lock } \ell\texttt{:}\,j{+}1\ \texttt{|}\ s[\ell\texttt{:}\,j]\ s\texttt{[}M\{x{:=}\texttt{unit}\}\texttt{]}\rrbracket\end{array}}$$

(R-METHOD-RELEASE)

$$\frac{j \text{ is odd}}{\begin{array}{l}\quad\mathbb{C}\llbracket \texttt{lock } \ell\texttt{:}\,j \qquad \texttt{|}\ s\texttt{[val }x\texttt{ = }\ell\texttt{.release(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \texttt{lock } \ell\texttt{:}\,j{+}1\ \texttt{|}\ s[\ell\texttt{:}\,j]\ s\texttt{[}M\{x{:=}\texttt{unit}\}\texttt{]}\rrbracket\end{array}}$$

(R-FIELD-WRITE)

$$\frac{}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[}p\,.f\texttt{ = }v\texttt{; }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket s[p\,.f{=}v]\ s\texttt{[}M\texttt{]}\rrbracket\end{array}}$$

(R-SPECULATION-BEGIN)

$$\frac{\mathit{thrd}(\phi) \in \mathit{def}(A)}{\begin{array}{l}\quad\mathbb{C}\llbracket A\rrbracket \\ \to \mathbb{C}\llbracket \top \Rightarrow A \,[]\, \phi \Rightarrow A\rrbracket\end{array}}$$

(R-SPECULATION-END)

$$\frac{\mathbb{C} \text{ justifies speculation } \phi}{\begin{array}{l}\quad\mathbb{C}\llbracket \top \Rightarrow A \,[]\, \phi \Rightarrow B\rrbracket \\ \to \mathbb{C}\llbracket B\rrbracket\end{array}}$$

(R-FIELD-READ)

$$\frac{\mathbb{C} \text{ justifies read } p\,.f{=}v \text{ by } s}{\begin{array}{l}\quad\mathbb{C}\llbracket s\texttt{[val }x\texttt{ = }p\,.f\texttt{; }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket s\texttt{[}M\{x{:=}v\}\texttt{]}\rrbracket\end{array}}$$

(R-SPECULATION-CONTEXT1)

$$\frac{\mathbb{C}\llbracket A\rrbracket \to \mathbb{C}\llbracket A'\rrbracket}{\begin{array}{l}\quad\mathbb{C}\llbracket \top \Rightarrow A \,[]\, \phi \Rightarrow B\rrbracket \\ \to \mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \phi \Rightarrow B\rrbracket\end{array}}$$

(R-SPECULATION-CONTEXT2)

$$\frac{\mathbb{C}\llbracket \phi B\rrbracket \to \mathbb{C}\llbracket \phi B'\rrbracket}{\begin{array}{l}\quad\mathbb{C}\llbracket \top \Rightarrow A \,[]\, \phi \Rightarrow B\rrbracket \\ \to \mathbb{C}\llbracket \top \Rightarrow A \,[]\, \phi \Rightarrow B'\rrbracket\end{array}}$$

Fig. 2. Reduction ($A \to B$)

Non-locks are initialized via R-NEW, consuming a free name of the appropriate class and initializing the fields using write actions by the initializing thread. We ignore types as much as possible, therefore all non-locks are runnable once initialized.

New lock creation is addressed separately in rule R-NEW-LOCK. The state of the lock is stored as an integer counter, which enforces sequential consistency on lock actions. Locks with even state may be acquired, and those with odd state released. (Both *fields* and *mbody* are undefined on the reserved class `Lock`.)

R-FIELD-WRITE describes field writes. This is a relaxed memory model, so the field writes become actions that float into the evaluation context, rather than updating a shared location. Field reads, as described in rule R-FIELD-READ, may take any value that is justified by the evaluation context. In a program with data races or locks, this could be nondeterministic.

Speculation can occur at any point, using R-SPECULATION-BEGIN. The initial branch has guard $\top$, indicating that this branch may make no additional assumptions. The final branch has a speculative action as its guard. The final branch may use the speculation to justify reads. R-SPECULATION-CONTEXT lets each branch of speculation evolve independently. This typically happens by using the structural rules of Figure 1 to bring parallel threads and locks into the speculation to enable computation. Results from an active speculation can only leak to the outside world via S-SPECULATION-PREFIX. If all branches produce an action, it can potentially float out into the surrounding environment. This is significant, since only actions that manage to make it outside of a speculation may be used to finalize it R-SPECULATION-END.

5 Examples

In the following examples, we assume an initialization thread which sets the initial state and starts the threads. We assume a single object `p`, with four fields, `f`, `g`, `h`, and `e`. To make the examples shorter, we elide the object name from field references, writing `p.f` as `f`. All fields are initially set to `0`. (Further examples may be found in Appendix D.)

Example 5 (Pugh (2004) §16). Consider the following variation of Program B from Section 3, which uses a single field: $\texttt{s[x=f;f=1;}\uparrow\texttt{x] | t[y=f;f=2;}\uparrow\texttt{y]}$. As in the JMM, the outcome $\texttt{s[}\uparrow\texttt{2]|t[}\uparrow\texttt{1]}$ is possible. In our semantics, one may speculate $\texttt{s}\langle\texttt{f=1}\rangle$ and $\texttt{t}\langle\texttt{f=2}\rangle$, resulting in the following reduction.

$$\begin{array}{lll} \twoheadrightarrow & \top\Rightarrow & (\texttt{s[x=f;f=1;}\uparrow\texttt{x] | t[y=f;f=2;}\uparrow\texttt{y]}) \\ & [\!]\ \texttt{s}\langle\texttt{f=1}\rangle\texttt{t}\langle\texttt{f=2}\rangle\Rightarrow & (\texttt{s[x=f;f=1;}\uparrow\texttt{x] | t[y=f;f=2;}\uparrow\texttt{y]}) \end{array}$$

The read actions from each thread may now read any justifiable value. In the final branch, the value read may come from the speculation, as below.

$$\begin{array}{lll} \twoheadrightarrow & \top\Rightarrow & (\texttt{s[f=1;}\uparrow\texttt{0] | t[f=2;}\uparrow\texttt{0]}) \\ & [\!]\ \texttt{s}\langle\texttt{f=1}\rangle\texttt{t}\langle\texttt{f=2}\rangle\Rightarrow & (\texttt{s[f=1;}\uparrow\texttt{2] | t[f=2;}\uparrow\texttt{1]}) \end{array}$$

The write actions can then be performed. Because the same writes actions are performed in each branch, the write actions may leave the speculation using the structural order (S-PAR-PREFIX and S-SPECULATION-PREFIX).

```
↠  ⊤⇒             (s[f=1]s[↑0] | t[f=2]t[↑0])
   [] s⟨f=1⟩t⟨f=2⟩⇒(s[f=1]s[↑2] | t[f=2]t[↑1])
≧  s[f=1]t[f=2](⊤⇒             (s[↑0] | t[↑0])
                [] s⟨f=1⟩t⟨f=2⟩⇒(s[↑2] | t[↑1]) )
→  s[f=1]t[f=2](s[↑2] | t[↑1])
```

The speculation is justified, allowing us to use R-SPECULATION-END. □

Most of the examples deal with integer fields because this is the typical style in the literature. Given that our semantics separates name binding from object initialization, as runtime systems do, dealing with object fields is no more complicated. For example, in `s[x=f;f=new d();↑x] | t[y=f;f=new d();↑y] | free q | free r`, reduction can proceed as above. In this case we speculate `s⟨f=q⟩` and `t⟨f=r⟩`, resulting in `s[f=q]t[f=r] (s[↑r] | t[↑q] | runnable q | runnable r)`

Before getting negative, we present two more "positive" examples, which are consistent with the JMM. Example 6 discusses inlining. Example 7 discusses nested speculation. Inlining can reduce the number of concurrent reads available, but can also add flexibility in reordering writes if there are data or control dependencies between threads that prevent reordering.

Example 6 (Manson et al. (2005) figures 11 and 12). Consider the following.

```
s[x=f; if(x==0){f=1;} y=f; g=y; ↑(x,y)] |
u[z=g; f=z; ↑z]
```

The outcome `s[↑(1,1)]|u[↑1]` is possible. Speculate `s⟨g=1⟩ u⟨f=1⟩`. The initial branch can produce `s[f=1] s[g=1] u[f=1]` in that order. Note that the write by `u` must follow the `s`'s write to `g`, but is not dependent on the `s`'s write to `f`. The semantics can therefore reorder the writes by `s` before making them visible to `u`, resulting in the sequence `s[g=1] u[f=1] s[f=1]`. The final branch can produce `s[g=1]` and `u[f=1]`, in any order, but can not produce `s[f=1]`. We can therefore reach the following state.

```
   ⊤⇒           s[g=1]u[f=1]s[f=1](s[↑(0,1)]|u[↑1])
   [] s⟨g=1⟩u⟨f=1⟩⇒(s[g=1]u[f=1](s[↑(1,1)]|u[↑1]))
↠  s[g=1]u[f=1](⊤⇒          s[f=1](s[↑(0,1)]|u[↑1])
                [] s⟨g=1⟩u⟨f=1⟩⇒(s[↑(1,1)]|u[↑1]))
↠  s[g=1]u[f=1](s[↑(1,1)]|u[↑1])
```

Thus, the result is possible.

The situation changes, however, if we split thread `s` as follows. In this case, the result `s[↑1]|t[↑1]|u[↑1]` is impossible.

```
s[x=f; if(x==0){f=1;} ↑x] | t[y=f; g=y; ↑y] |
u[z=g; f=z; ↑z]
```

The dependency between `s[f=1]` and `t[g=1]` now crosses two threads, and therefore `s[f=1]` must be ordered before any subsequent actions. We reach the following state.

$$
\begin{array}{lll}
 & \top\!\Rightarrow \texttt{s[f=1](s[}\uparrow\texttt{0]|(t[g=1](t[}\uparrow\texttt{1]|u[f=1]u[}\uparrow\texttt{1])))} & \\
 & \square\cdots & \\
\geqq & \top\!\Rightarrow \quad \texttt{s[f=1]t[g=1]u[f=1](s[}\uparrow\texttt{0]|t[}\uparrow\texttt{1]|u[}\uparrow\texttt{1])} & \\
 & \square\cdots\Rightarrow \qquad\quad \texttt{t[g=1]u[f=1](s[}\uparrow\texttt{1]|t[}\uparrow\texttt{1]|u[}\uparrow\texttt{1])} &
\end{array}
$$

In this case, however, we can not move the writes by `t` or `u` through to justify the speculation since they are blocked by `s[f=1]` in the initial branch and this write can not be matched by the final branch. □

Example 7 (Pugh (2004) §11). Consider `s[x=h;e=x;y=f;g=y;↑(x,y)] | t[w=e; z=g;h=z;f=1;↑(w,z)]`. To get the result `s[↑(1,1)]|t[↑(1,1)]`,we first speculate `t⟨f=1⟩s⟨g=1⟩`, then `t⟨h=1⟩`, and then `s⟨e=1⟩`. These speculations result in a four-hole context.

$$
\begin{array}{l}
\top\!\Rightarrow [\![-]\!]_1 \\
\square\ \texttt{t}\langle\texttt{f=1}\rangle\texttt{s}\langle\texttt{g=1}\rangle \Rightarrow \top\!\Rightarrow [\![-]\!]_2 \\
\qquad\qquad\qquad\qquad\quad \square\ \texttt{t}\langle\texttt{h=1}\rangle \Rightarrow \top\!\Rightarrow [\![-]\!]_3 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \square\ \texttt{s}\langle\texttt{e=1}\rangle \Rightarrow [\![-]\!]_4
\end{array}
$$

Placing the term into this context creates four copies of the initial process, which we will refer to by number. Process 1 justifies the outer speculation, each subsequent process justifies the next speculation, and process 4 is the final process. To succeed, all processes must generate `t[f=1]` and `s[g=1]`, processes 2–4 must generate `t[h=1]`, and processes 3 and 4 must generate `s[e=1]`.

Process 1 can perform the writes `t[h=0]t[f=1]` and `s[e=0]s[g=1]`. The second write of `s` is only possible after the second write of `t`. The semantics can reorder the writes of `t`, keeping the first write private, and likewise for `s`. The other processes can reduce without any dependencies between threads and can therefore perform the same reordering. Thus we can get the following processes.

```
1: t[f=1]s[g=1](t[h=0]t[↑(0,0)]|s[e=0]s[↑(0,1)])
2: t[f=1]s[g=1]t[h=1](t[↑(0,1)]|s[e=0]s[↑(0,1)])
3: t[f=1]s[g=1]t[h=1]s[e=1](t[↑(0,1)]|s[↑(1,1)])
4: t[f=1]s[g=1]t[h=1]s[e=1](t[↑(1,1)]|s[↑(1,1)])
```

Using this stratification, the speculations can be discharged and the result is allowed.

The multiple nesting of speculations is necessary. While the write `s[e=1]` is already possible in process 2, this write can only happen after the write to `h` in `t`. This dependency makes it impossible for process 2 to publish `s[g=1]` without the necessarily preceding `t[h=1]`. This in turn prohibits the outer speculation from finalizing because process 1 can not match `t[h=1]`. □

The examples above demonstrate out-of-order reads, a hallmark of relaxed memory models. These examples argue informally that the model is "relaxed enough". We now revisit the examples given in Section 3, to argue that it is not "too relaxed".

The program `s[x=f; if(x==1){g=1;} f=1; y=g; ↑y]` should not be allowed to produce `s[↑1]`. Such *self justifying* executions are prevented by our semantics. Since only `s` can produce writes, only speculations by `s` can be finalized (via R-SPECULATION-END and Definition 2); yet reads by `s` can not be justified by its own speculations (Definition 1). Speculation is useless in single-threaded programs, as it should be.

Initiality prevents the program `s[x=f;g=x;↑x] | t[y=g;f=y;↑y]` from producing the outcome `t[↑1]`. The initial branch can not write anything but `0`; therefore no useful speculations can be finalized via R-SPECULATION-END.

Consistency prevents Program C (Section 3) from producing the illegal execution reported there. S-SPECULATION-PREFIX prevents such executions by requiring that the initial and final branch of a speculation must execute the same actions in the *same order*. The actual requirement is slightly weaker, since S-SPECULATION-PAR and Definition 3 allow some reordering; but no reordering is allowed on lock actions.

Timeliness prevents Program D (Section 3) from producing the illegal execution reported there. This example motivates the "no intervening release" clause of Definition 2, which ensures that the speculation can not be finalized. Whereas Program D describes a speculation that occurs too late with respect to its justifying write, Example 8 discusses one that occurs too early.

Example 8. Consider the following program.

```
s[l.acquire(); x=f; f=x+1; g=1; l.release(); ↑x] |
u[l.acquire(); x=f; f=x+1; y=g; l.release(); ↑(x,y)]
```

Clearly `s[↑1]|u[↑(0,1)]` is unacceptable. If we attempt to get this result by first allowing `u` to acquire the lock, then speculating `s⟨g=1⟩`, we arrive at

```
u[l:0]
    ⊤⇒          u[f=1]u[l:1]s[l:2]s[f=2]s[g=1]s[l:3](s[↑1]|u[↑(0,0)])
    [] s⟨g=1⟩ ⇒u[f=1]u[l:1]s[l:2]s[f=2]s[g=1]s[l:3](s[↑1]|u[↑(0,1)]).
```

The actions of `u` can commute with the speculation since they belong to a different thread, but the actions of `s` can not, since `s⟨g=1⟩s[l:2]` ⊁ `s[l:2]s⟨g=1⟩`; clause A-ACQUIRE of Definition 3 applies to write actions, but not speculations. Thus the speculation can not be finalized. □

The final two examples demonstrate areas where our model differs from the JMM. Example 9 shows that our model allows executions of lockless programs that are not allowed by the JMM. Example 10 shows that our model is incomparable to the JMM for programs with both locks and data races. In both cases, our model validates optimizations that are disallowed by the JMM. See Section 7 for more general results.

Example 9 (Sevcík (2008) §5.3.2). This example discusses redundant read after read elimination. Consider the following program.

```
s[x=f; g=x;] |
t[y=g; if(y==1){z=g; f=z;} else {f=1;}; ↑y]
```

The outcome `t[↑1]` is allowed using the speculation `s⟨g=1⟩`. Both initial and final branches produce the actions `t[f=1]s[g=1]`. The same behavior is allowed, with the same speculation, if the boxed statement pair is replaced by "`f=y;`". Our semantics validates the transformation. The JMM disallows the behavior for the original program, but allows it for the transformed one (Sevcík 2008), thus invalidating the transformation.

Conversely, Sevcík (2008, §5.3.4) demonstrates a behavior that is allowed by the JMM, but invalidated by an irrelevant read introduction. Again, our semantics allows the behavior both before and after the transformation. (See Example 20.) □

Example 10 (Sevcík (2008) §5.3.3). (Roach motel optimization). Consider whether the following program.

```
s[l.acquire(); f=2; l.release();] |
t[l.acquire(); f=1; l.release();] |
u[x=f; l.acquire();
    y=h; if(x==2){g=1;} else {g=y;}
  l.release(); ↑(x,y)] |
v[z=g; h=z; ↑z]
```

The outcome `u[↑(1,1)]|v[↑1]` is possible using the speculation v⟨h=1⟩. The initial branch schedules as follows: `s`, `u`'s initial read, `t`, `u`'s acquire and write, `v`, then `u`'s release. This allows the initial branch to reduce to the following.

```
s[l:0]s[f=2]s[l:1]t[l:2]t[f=1]t[l:3]
  u[l:4]u[g=1]v[h=1]u[l:5](u[↑(2,0)]|v[↑1])
```

The final branch performs the same schedule, except that `t` executes before `u`'s initial read, with the speculation occurring after `u`'s acquire. Using the false case of the conditional, it produces the same action sequence, but with the desired result.

This execution become impossible after reversing the order of the statements in the boxed term so that the lock is acquired before the read: "`l.acquire(); x=f;`". Now the actions of threads `s`, `t` and `u` are now totally ordered and therefore the relation of `t` and `u`'s initial read must be consistent in the initial and final branches. If the initial branch reads `1` from `f`, then it must write `v[h=0]`. If the initial branch reads `2` from `f`, then the final branch must also read `2` and therefore can not produce the desired result.

Our semantics validates the transformation; the JMM does not. In a reversal of our results, the JMM disallows the first execution, but allows the second (Sevcík 2008). □

6 Analysis

Informally, one can see that the speculation construct can not create thin air reads because it enjoys initiality (there is a computation justifying the speculation that does not use the speculation) and consistency (the only way in which results from an active speculation can leak to the outside world is via the S-SPECULATION-PREFIX rules). Thus, any speculation is validated by an execution consistent with the final execution.

Every valid JMM execution of a lockless program can be mimicked by the system in this paper. See Appendix B for proof sketch. We now show that our semantics coincides with SC (and therefore with the JMM) for DRF programs. As shown by Example 10, our semantics is incomparable to the JMM for programs with both data-races and locks.

Our model does not record read actions. In order to define read-write data races, we use a modified reduction relation, which introduces a *read actions* into the process, notation $s\ulcorner p\,.f{=}v\urcorner$. A read write data race occurs whenever there is a race between a read and a write. Define $\mapsto$ as in Figure 2, but for the rule R-FIELD-READ, which becomes

$$\frac{\mathbb{C}\text{ justifies } s\langle p\,.f{=}v\rangle}{\mathbb{C}[\![s\texttt{[val } x = p\,.f\texttt{; } M\texttt{]}]\!] \mapsto \mathbb{C}[\![\ s\ulcorner p\,.f{=}v\urcorner\ s\texttt{[}M\{x{:=}v\}\texttt{]}]\!]}\,.$$

Define the partial function $act_s(A) = act_s(\mathbb{C})$, if $A = \mathbb{C}[\![s\,\texttt{[}M\texttt{]}]\!]$ for some $\mathbb{C}$ and M. We say that A has a *read-write data race* if $\vec{\sigma} = act_s(\mathbb{C})$ and there exists i and j such that $\sigma_i = t\ulcorner p\,.f\texttt{=}v\urcorner$ and $\sigma_j = s[p\,.f\texttt{=}w]$ such that $i \not<^{\vec{\sigma}}_{hb} j$ and $j \not<^{\vec{\sigma}}_{hb} i$. Define a *write-write data race* similarly.

A process A is *speculation-free* if it has no subterm that is a speculation process. Write $A_0 \to \cdots \to A_n$ to abbreviate $A_0\ (\to \cup \geqq) \cdots (\to \cup \geqq)\ A_n$, and similarly for $\mapsto$. A reduction sequence $A_0 \to \cdots \to A_n$ is *top-level* if A_0 and A_n are speculation-free.

The speculation-free assumption on top-level processes is reasonable because user programs do not have speculations; speculations are only created by the operational semantics. Speculation transitions are redundant in read-write data race free reduction sequences.

Definition 11. Let A'_i be derived from A_i by replacing each speculation $(\top{\Rightarrow}A \,[\!]\, \phi \Rightarrow B)$ by the final branch (B). By induction on n, such an A'_i exists for each A_i. A top-level reduction sequence $A_0 \mapsto \cdots \mapsto A_n$ is *read-write data race free*, if none of the A'_i, so defined, has a read-write data race. □

Lemma 12. *Let the top-level reduction sequence $A_0 \to \cdots \to A_n$ be read-write data race free. Then, there is a reduction sequence $A_0 = B_0 \to \cdots \to B_n = A_n$, such that for all $j \in \{1,\ldots,n\}$, B_j is speculation-free.*

PROOF. See Appendix A. □

For processes that are also write-write data race free, each read is matched by a unique write. Thus, the memory may be treated as a map from locations to values without any change to the possible reductions, ensuring that DRF programs can be executed in standard SC fashion.

7 Simulation

The goal of this section is to define a simulation relation that is a precongruence and that validates interesting examples. We are not concerned if the relation is finer than orders based on testing or contextual equivalence. For simplicity, we restrict our attention in this section to processes that do not contain name binders, object initialization or method calls other than `acquire` and `release`. For this class of processes, we impose the following additional well-formedness criterion: in any subprocess $\top{\Rightarrow}A \,[\!]\, \phi \Rightarrow B$, $def(A) = def(B)$.

Intuitively, A simulates B if A and B have the same memory and whenever B reduces, then A can reduce to a matching process. The definition is complicated by the possible interleaving of actions and speculations, and the various ways that a context can interact with an environment. Rather than comparing memories, we compare *environment contexts*: $\mathbb{E} ::= [\![-]\!] \mid \alpha\,\mathbb{E} \mid \phi\,\mathbb{E} \mid s\,\texttt{[}{\uparrow}v\texttt{]} \,|\, \mathbb{E}$. The environment context $s\,\texttt{[}{\uparrow}v\texttt{]} \,|\, \mathbb{E}$ contains a placeholder for environment actions performed by thread s, in parallel with the rest of the context.

For a set of thread names S, the context $\mathbb{E}$ is *complete* iff for every $\sigma \in \mathbb{E}$ such that $s = thrd(\sigma) \notin S$, it is the case that $s\,\texttt{[}{\uparrow}v\texttt{]}$ occurs in $\mathbb{E}$ after σ.

In the remainder of this section, we use S to refer to the set of non-environment threads. Threads not in S can be used by the environment.

Definition 13. Given a set S of thread names and a binary relation $\mathscr{R}$ on well-formed processes, we define $S \vdash A\ \mathscr{F}(\mathscr{R})\ B$ to hold iff the following conditions are satisfied.

(Threads) $\mathit{def}(A) = \mathit{def}(B)$ and $S \subseteq \mathit{thrds}(A)$ and for all $s \in \mathit{thrds}(A) \setminus S$, if $s[M]$ occurs in A or B then $M = \uparrow \mathtt{unit}$.
(Well-formed) For all $\mathbb{C}$, $\mathbb{C}[\![A]\!]$ is well-formed iff $\mathbb{C}[\![B]\!]$ is well-formed.
(Reduction) For all B', if $B \rightarrow B'$ then there exists A' such that $A \twoheadrightarrow A'$ and $S \vdash A'\ \mathscr{R}\ B'$.
(Structural order) For all B' if $B \geqq B'$ then there exists A' such that $A \twoheadrightarrow A'$ and $S \vdash A'\ \mathscr{R}\ B'$.
(Equivalent top-level choices) For all B', ϕ, B'', if $B = \mathbb{E}[\![\top \Rightarrow B' \,[\!]\, \phi \Rightarrow B'']\!]$ then there exists A', ψ, A'' such that (1) $A = \mathbb{E}[\![\top \Rightarrow A' \,[\!]\, \psi \Rightarrow A'']\!]$, (2) $S \vdash \mathbb{E}[\![A']\!]\ \mathscr{R}\ \mathbb{E}[\![B']\!]$, and (3) $S \vdash \mathbb{E}[\![\phi A'']\!]\ \mathscr{R}\ \mathbb{E}[\![\psi B'']\!]$.
(Equivalent actions/guards/returns) For all $\mathbb{E}$, B' if $B = \mathbb{E}[\![B']\!]$ then there exists A' such that $A = \mathbb{E}[\![A']\!]$.
(Environment writes) For each $s \in \mathit{thrds}(A) \setminus S$ if B' is obtained from B by replacing every occurrence of $s\mathtt{[}{\uparrow}\mathtt{unit]}$ with $s[p\,.f{=}v]s\mathtt{[}{\uparrow}\mathtt{unit]}$ and similarly for A' obtained from A, then $S \vdash A'\ \mathscr{R}\ B'$.
(Top-level lock removal) For all $\vec{\sigma}$, B' and for all ℓ in the fixed set of lock names if $B = \vec{\sigma}\mathtt{(lock}\ \ell\mathtt{:}\,j\,|\,B'\mathtt{)}$ then there exists A' such that $A = \vec{\sigma}\mathtt{(lock}\ \ell\mathtt{:}\,j\,|\,A'\mathtt{)}$ and $S \vdash \vec{\sigma}A'\ \mathscr{R}\ \vec{\sigma}B'$.
(Top-level lock addition) For all ℓ in the fixed set of lock names if $\mathtt{(lock}\ \ell\mathtt{:}\,j\,|\,B\mathtt{)}$ is well-formed then $S \vdash \mathtt{(lock}\ \ell\mathtt{:}\,j\,|\,A\mathtt{)}\ \mathscr{R}\ \mathtt{(lock}\ \ell\mathtt{:}\,j\,|\,B\mathtt{)}$.
(Environment locks) For each s in $\mathit{thrds}(A) \setminus S$ if

- the occurrences of $\mathtt{lock}\ \ell\mathtt{:}\,j\,|\,s\mathtt{[}{\uparrow}\mathtt{unit]}$ in B account for all occurrences of $s\mathtt{[}{\uparrow}\mathtt{unit]}$ in B, and
- B' is obtained from B by replacing all occurrences of $\mathtt{lock}\ \ell\mathtt{:}\,j\,|\,s\mathtt{[}{\uparrow}\mathtt{unit]}$ with $\mathtt{lock}\ \ell\mathtt{:}\,j{+}1\,|\,s[\ell\mathtt{:}\,j]s\mathtt{[}{\uparrow}\mathtt{unit]}$,

then

- the occurrences of $\mathtt{lock}\ \ell\mathtt{:}\,j\,|\,s\mathtt{[}{\uparrow}\mathtt{unit]}$ in A account for all occurrences of $s\mathtt{[}{\uparrow}\mathtt{unit]}$ in A,
- A' is obtained from A by replacing all occurrences of $\mathtt{lock}\ \ell\mathtt{:}\,j\,|\,s\mathtt{[}{\uparrow}\mathtt{unit]}$ with $\mathtt{lock}\ \ell\mathtt{:}\,j{+}1\,|\,s[\ell\mathtt{:}\,j]s\mathtt{[}{\uparrow}\mathtt{unit]}$, and
- $S \vdash A'\ \mathscr{R}\ B'$.

Define $S \vdash A \gtrsim B$ to be the largest relation such that $S \vdash A \gtrsim B$ implies $S \vdash A\ \mathscr{F}(\gtrsim)\ B$. Define the order $A \gtrsim B$ iff for all complete $\mathbb{E}$ such that $\mathbb{E}[\![A]\!]$ and $\mathbb{E}[\![B]\!]$ are well-formed, we have $\mathit{thrds}(A) \vdash \mathbb{E}[\![A]\!] \gtrsim \mathbb{E}[\![B]\!]$.

Consider terms M and N with no free variables but perhaps free names. Define the order $M \gtrsim N$ iff there exists t such that $t[M] \gtrsim t[N]$ The choice of t is irrelevant in this definition. □

Proposition 14. $\gtrsim$ *is a precongruence on processes and on terms.* □

We now use the theory of simulation to validate several optimizations. The first inequality shows that writes can be reordered. The second demonstrates roach motel reordering. The third demonstrates redundant read after read elimination. Since simulation is a precongruence, the transformations are valid in any program context.

Proposition 15. *The following inequivalences hold.*

$$\begin{aligned}
p\texttt{.}f\texttt{=1;}p\texttt{.}g\texttt{=1;}\uparrow\texttt{unit} &\gtrsim p\texttt{.}g\texttt{=1;}p\texttt{.}f\texttt{=1;}\uparrow\texttt{unit}\\
p\texttt{.}f\texttt{=1;}\ell\texttt{.acquire();}\uparrow\texttt{unit} &\gtrsim \ell\texttt{.acquire();}p\texttt{.}f\texttt{=1;}\uparrow\texttt{unit}\\
\texttt{val }x\texttt{=}p\texttt{.}f\texttt{;val }y\texttt{=}p\texttt{.}f\texttt{;}M &\gtrsim \texttt{val }x\texttt{=}p\texttt{.}f\texttt{;}M\{y\texttt{:=}x\}
\end{aligned}$$

PROOF. See Appendix C. □

8 Conclusion

This paper follows the research program of Cenciarelli et al. (2007) and Boudol and Petri (2009) in attempting to fit relaxed memory models into generative structured operational semantics. The technical novelty is manifest in the "speculation" construct. We show that the basic properties of the JMM hold in our setting. Our contributions advance the state-of-the-art in two ways. (1) We expand the expressivity of these methods to include full JMM behaviors for lockless programs and general object-oriented programs. (2) We describe simulation methods and precongruence results for the sublanguage that corresponds to the first-order imperative shared-memory computing.

Our treatment of programs with both data races and locks provides a technically robust variation on JMM ideas. For example, our methods validate expected roach-motel reordering laws and related peephole optimizations.

Bibliography

S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

D. Aspinall and J. Sevcík. Formalising Java's data race free guarantee. In *TPHOLs*, volume 4732 of *LNCS*, pages 22–37. Springer, 2007.

H.-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05*, pages 261–268. ACM, 2005.

H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08*, pages 68–78, 2008.

G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *POPL*, pages 392–403, 2009.

G. Bronevetsky and B. R. de Supinski. Complete formal specification of the OpenMP memory model. *Int. J. Parallel Program.*, 35(4):335–392, 2007.

S. Burckhardt, M. Musuvath, and V. Singh. Verifying compiler transformations for concurrent programs. MSR-TR-2008-171, 2008.

P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, volume 4421 of *LNCS*, pages 331–346. Springer, 2007.

C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI'93*, volume 28(6), pages 237–247. ACM Press, 1993.

S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu. TSOtool: A program for verifying memory systems using the memory consistency model. In *ISCA '04*, page 114. IEEE, 2004.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Programming Languages and Systems*, 23(3):396–450, 2001.

A. Kamil, J. Su, and K. A. Yelick. Making sequential consistency practical in Titanium. In *SC*, page 15. IEEE, 2005.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

D. Lea. The JSR-133 cookbook for compiler writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html, 2008. Last modified: Apr 2008.

V. M. Luchangco. *Memory consistency models for high-performance distributed computing*. PhD thesis, MIT, 2001.

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05*, pages 378–391. ACM, 2005.

R. Milner. The polyadic pi-calculus: a tutorial. Technical report, Logic and Algebra of Specification, 1991.

M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima, 2008.

W. Pugh. Causality test cases. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html, 2004.

V. A. Saraswat. Concurrent constraint-based memory machines: A framework for Java memory models. In *ASIAN*, volume 3321 of *LNCS*, pages 494–508. Springer, 2004.

V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *PPOPP*, pages 161–172. ACM, 2007.

S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, pages 379–391, 2009.

J. Sevcík. *Program Transformations in Weak Memory Models*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2008.

J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, volume 5142 of *LNCS*, pages 27–51. Springer, 2008.

R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

K. Yelick, D. Bonachea, and C. Wallace. A proposal for a UPC memory consistency model, v1.1. Technical Report LBNL-54983, Lawrence Berkeley National Lab, 2004.

A Proof of Lemma 12

By induction on the length of the transition sequence.

In the inductive step, the key case is R-SPECULATION-BEGIN when $A_0 = \mathbb{C}\left[\!\left[B\right]\!\right]$ and $A_1 = \mathbb{C}\left[\!\left[\top \Rightarrow B \,[\!]\, \phi \Rightarrow B\right]\!\right]$. Since A_n is speculation-free without loss of generality assume that the speculation end rule is invoked in the transition to A_n. Let $thrd(\phi) = t_{wr}$.

Recall that the sole use of speculative actions is in the read rule R-FIELD-READ. The possibilities are as follows.

(1) There is no use of R-FIELD-READ in s that uses ϕ. In this case, the speculation is redundant, and there is a reduction from A_0 to A_n that does not use this speculation and with at least one less reduction.

(2) There is a use of R-FIELD-READ in s that uses ϕ. Let $\mathbb{C}'[\![s\texttt{[val } x \texttt{ = } p.f\texttt{; } M\texttt{]}]\!]$ be the read in question. (By inspection of the semantic rules, there can only be one use of R-FIELD-READ in deducing a single transition.) We show that the read can produce the same value without considering the speculation.

Let $\beta_{\mathtt{w}r}$ be the write action that justifies the closing of the speculation in the reduction sequence to A_n. Let $\vec{\tau} = acts_s(A_n)$. Let k be the index of $t[\![-]\!]$ in $\vec{\tau}$. Let the position of $\beta_{\mathtt{w}r}$ in $\vec{\tau}$ be $i_{\mathtt{w}r}$. By Definition 2, $thrd(\beta_{\mathtt{wr}}) = t_{\mathtt{wr}}$. Let the position of the marked action (corresponding to the read) in $\vec{\tau}$ be $i_{\mathtt{r}d}$.

Since there is no read-write data race, the read and write must have one of the following relations.

(2)(a) The write happens-before the read ($i_{\mathtt{w}r} <_{hb}^{\vec{\tau}} i_{\mathtt{r}d}$). We show that the speculative action is not used in this case by showing that $\beta_{\mathtt{w}r}$ is visible to the read[1]. It suffices to show that there is no intervening write between $i_{\mathtt{wr}}$ and $i_{\mathtt{r}d}$. We prove by showing that that the presence of an intervening write leads to a contradiction.

There are two cases to consider depending on the thread of the intervening write.

(2)(a)(i) Intervening write in thread different from $t_{\mathtt{w}r}$: Since $\vec{\tau}$ justifies ϕ. So, there is no intervening release between $i_{\mathtt{wr}}$ and k, implying that there is no intervening release between $i_{\mathtt{wr}}$ and $i_{\mathtt{r}d}$. Thus, there can not be a synchronization edge between positions $i_{\mathtt{wr}}$ and $i_{\mathtt{r}d}$ in $\vec{\tau}$. So, there can not be an intervening write between $i_{\mathtt{wr}}$ and $i_{\mathtt{r}d}$ in $\vec{\tau}$ in a thread different from $t_{\mathtt{wr}}$. (Compare Example 8.)

(2)(a)(ii) Intervening write in thread $t_{\mathtt{w}r}$: In this case there is an intervening write between $i_{\mathtt{wr}}$ and k contradicting our assumption that $\beta_{\mathtt{wr}}$ justifies closing of the speculation.

(2)(b) The read happens-before the write ($i_{\mathtt{r}d} <_{hb}^{\vec{\tau}} i_{\mathtt{wr}}$). We show that $\beta_{\mathtt{wr}}$ can not move outside the scope of the speculation and thus can not justify the closing of the speculation. Thus, this case leads to a contradiction.

There are two subcases.

(2)(b)(i) The read is program-order before the write. In this case, $t_{\mathtt{wr}} = s$ and Definition 1 does not permit the speculation ϕ to be used to justify the read. So, this is not possible.

(2)(b)(ii) The read is synchronization-order before the write. That is, there is a release of some lock by s after the read that enables an acquire in $thrd(\beta_{\mathtt{wr}})$ before $\beta_{\mathtt{wr}}$.

[1] Thanks to an anonymous referee who detected a bug in this portion of the proof in an earlier version of this paper.

The following non-commutations are in play:

$$\begin{gathered}\phi\ t_{\mathtt{w}r}[\mathtt{acq}\ \ell{:}j] \not\triangleright t_{\mathtt{w}r}[\mathtt{acq}\ \ell{:}j]\ \phi \\ t_{\mathtt{w}r}[\mathtt{acq}\ \ell{:}j]\ \beta_{\mathtt{w}r} \not\triangleright \beta_{\mathtt{w}r}\ t_{\mathtt{w}r}[\mathtt{acq}\ \ell{:}j]\end{gathered}$$

So, there there is no way to use the S-SPECULATION-PREFIX rules to move $\beta_{\mathtt{w}r}$ outside the scope of the speculation. So, this is not possible. (Compare Program D, Section 3.)

B Lockless programs

For lockless programs, Example 9 showed that our model allows executions that are not allowed by the JMM. We now show that our model is strictly more expressive than the JMM for lockless programs. Perhaps surprisingly, the absorption laws of Definition 3 are used in this proof.

As in Section 5, we assume an initialization thread which sets the initial state and starts the threads. The adjective "lockless" applies to the threads so started — our rules for starting threads already forces the initialization actions of the initialization thread to "happen-before" any action in the thread. As in the JMM literature, we assume only integer shared locations, whose values are initially set to `0` in the initialization thread.

Consider a lockless program with data races. We use the JMM version of Definition 5.9 (page 100) of Sevcík (2008) as our starting point. Thus, we are using the JMM versions of the conditions (2,6). In this extended abstract, we highlight the constraints that are relevant to the proof sketch. For a valid execution, this definition provides a stratification via a collection of executions E_i, for $0 \leq i \leq n-1$. Execution E_i commits actions C_{i+1}, and the definition constrains $C_1 \subseteq \cdots \subseteq C_n$. In this lockless case, the idea is that an uncommitted read in execution E_i can only be connected to a write that precedes it in the program order of E_i.

Wlog, we assume that $C_{i+1} \setminus C_i$ contains atmost one write that we denote α_{i+1}.

For a read action $rd \in C_i$, there is an associated write action denoted by $W(rd)$ whose value is given by $V(W(rd))$. Consider the set

$$\begin{aligned}\{s\langle p\,.f{=}v\rangle \mid (\exists rd)\ rd \in C_i \text{ and } s = \mathit{thrd}(W(rd)) \text{ and}\\ p\,.f = \mathit{loc}(rd) \text{ and } v = V(W(rd))\\ \text{and } \mathit{thrd}(W(rd)) \neq \mathit{thrd}(rd)\}\end{aligned}$$

and let $\vec{\phi}_i$ be any sequence containing all the elements of this set in an order.

We build on intuitions from Example 7. Given an initial process B, we define the initial speculative process A_0 as follows.

$$\begin{aligned}A_0 &= (\top \Rightarrow B \,[\!]\, \vec{\phi}_1 \Rightarrow A_1)\\ A_j &= (\top \Rightarrow B \,[\!]\, \vec{\phi}_{j+1} \Rightarrow A_{j+1})\\ A_{n-1} &= B\end{aligned}$$

The intention is that the jth copy of B will execute as per the execution E_j. The required assumptions for this execution, namely $\vec{\phi}_j$, are present because of the surrounding speculation context.

For $j \in \{1,\ldots,n-1\}$, define $\overline{D_j} = \alpha_1 \ldots \alpha_j$. Fix an arbitrary j. We first show that E_j, in the context of speculation $\vec{\phi}_j$, produces the write sequence $\overline{D_{j+1}}$.

(1) Let $\overline{D_{j+1}}|s)$ be the restriction of $\overline{D_{j+1}}$ to a particular thread s. By assumption on E_j, s can produce the set of writes in $\overline{D_{j+1}}|s$. However, these writes may not be in the order indicated by $\overline{D_{j+1}}|s$ and the sequence of writes by s as per E_j could potentially also include other writes not included in $\overline{D_{j+1}}$. However, since there are no locks, we can use the absorption laws of Definition 3 to add prefix $\overline{D_{j+1}}|s$ to the actual write sequence produced by s under execution E_j. The proof of this follows that of lemma 4.

(2) By definition, every read in E_j is fulfilled either by the speculative context $\vec{\phi}_j$ or by a write that happens before it. Thus, no read in a thread s of E_j relies on writes in a different thread $t \neq s$, and the write actions produced by different threads in E_j are fully commutable by the dynamics. Therefore, the result of step (1) above for individual threads lifts to E_j that (potentially) consists of several threads, to yield the write sequence $\overline{D_{j+1}}$.

Given this, we now reason informally that all levels of this nested speculation can be successfully closed to show that computation E_{n-1} can be achieved as the result of executing A_0. The first step of our argument is that A_j produces the write sequence $\overline{D_{j+1}}$. This proceeds by a reverse induction from $n-1$ down to 0. The base case is by assumption on E_{n-1}. By the inductive step, A_{j+1} produces the writes $\overline{D_{j+2}}$ and hence the prefix $\overline{D_{j+1}}$. By assumption, E_j produces the write sequence $\overline{D_{j+1}}$. Therefore the two branches of the speculation are consistent up to $\overline{D_{j+1}}$. Thus, the complete commutation of writes with speculations embodied in Definition 3 and rule S-SPECULATION-PREFIX of Figure 1 enables A_j to produce the write sequence $\overline{D_{j+1}}$.

The second step of our argument is that the speculation of A_j can be closed successfully. This proceeds by a forward induction from 0 up to $n-1$. From the preceding step, A_0 produces $\overline{D_1}$ that suffices for justifying $\vec{\phi}_1$. Therefore the base case follows. For showing the inductive case, use the inductive hypothesis to get that all the speculations above A_j have been closed. From the above step, A_j produces $\overline{D_{j+1}}$ and hence all the writes needed to justify $\vec{\phi}_{j+1}$. Therefore the branch for A_{j+1} can be chosen in A_j.

C Simulation proof sketches

In proving simulations, it is useful to define *speculation contexts*, where context holes are labelled with natural numbers. Each natural number must occur on at most one context hole in a $\mathbb{D}$ context. We write $\mathit{labels}(\mathbb{D})$ for the finite set of natural numbers labelling context holes in $\mathbb{D}$.

$$\begin{aligned}\mathbb{D} ::= {} & [\![-]\!]_n \mid \alpha\,\mathbb{D} \mid \phi\,\mathbb{D} \mid t\,\texttt{[}{\uparrow}v\texttt{]} \\ & \mid \texttt{lock}\;\ell\texttt{:}\,j \mid \mathbb{D}\,\texttt{|}\,\mathbb{D} \mid \top \Rightarrow \mathbb{D} \,[\,]\, \phi \Rightarrow \mathbb{D}\end{aligned}$$

Here $[\![-]\!]_n$ is a context hole labelled with natural number n.

Consider a speculation context $\mathbb{D}$. Given a map $\mathscr{P}$ from $\mathit{labels}(\mathbb{D})$ to processes, define $\mathbb{D}[\![\mathscr{P}]\!]$ to be the process obtained by replacing context hole $[\![-]\!]_n$ with $\mathscr{P}(n)$ for all n in $\mathit{labels}(\mathbb{D})$.

Given a map $\mathscr{G}$ from $\mathit{labels}(\mathbb{D})$ to pairs of processes, define $\mathbb{D}[\![\mathscr{G}]\!]$ to be the pair of processes $(\mathbb{D}[\![\mathscr{G} \circ \mathit{fst}]\!], \mathbb{D}[\![\mathscr{G} \circ \mathit{snd}]\!])$.

We sketch a proof for the first case of Proposition 15, by defining a simulation candidate relation and then showing that it is indeed a simulation. First, define the following processes.

$$\begin{aligned}
A_1 &= t\,[p\,.\,g\texttt{=1;}\,p\,.f\texttt{=1;}\uparrow\texttt{unit]} \\
A_2 &= t\,[p\,.\,g\texttt{=1}]\,t\,\texttt{[}\uparrow\texttt{unit]} \\
A_3 &= t\,\texttt{[}\uparrow\texttt{unit]} \\
B_1 &= t\,[p\,.f\texttt{=1;}\,p\,.\,g\texttt{=1;}\uparrow\texttt{unit]} \\
B_2 &= t\,[p\,.\,g\texttt{=1;}\uparrow\texttt{unit]} \\
B_3 &= t\,\texttt{[}\uparrow\texttt{unit]}
\end{aligned}$$

Note that the following hold.

$$\begin{array}{ll}
A_1 \rightarrow\rightarrow\geqq t\,[p\,.f\texttt{=1}]A_2 & \qquad A_2 = t\,[p\,.\,g\texttt{=1}]A_3 \\
B_1 \rightarrow t\,[p\,.f\texttt{=1}]B_2 & \qquad B_2 \rightarrow t\,[p\,.\,g\texttt{=1}]B_3
\end{array}$$

For the result, by definition, we must show the following.

$$A_1 = t\,[p\,.f\texttt{=1;}\,p\,.\,g\texttt{=1;}\uparrow\texttt{unit]} \gtrsim t\,[p\,.\,g\texttt{=1;}\,p\,.\,g\texttt{=1;}\uparrow\texttt{unit]} = B_1$$

That is, for all complete $\mathbb{E}$ such that $\mathbb{E}[\![A_1]\!]$ and $\mathbb{E}[\![B_1]\!]$ are well-formed, we must show $\{t\} \vdash \mathbb{E}[\![A_1]\!] \gtrsim \mathbb{E}[\![B_1]\!]$. We construct a larger relation that is a simulation.

First, define the relation R on processes as follows.

$$R = \{(A_1,B_1),(A_2,B_2),(A_3,B_3)\}$$

Next, define the simulation candidate relation.

$$\mathscr{R} = \{\mathbb{D}[\![\mathscr{G}]\!] \mid \mathscr{G} \text{ is a map from } \mathit{labels}(\mathbb{D}) \text{ to } R\}$$

The $\mathscr{R}$ simulation candidate is closed under $\mathbb{E}$ since $\mathbb{D}$ contexts are more general than $\mathbb{E}$ contexts, so establishing that $\mathscr{R}$ is a simulation is sufficient.

We require that for all processes A and B,

$$\{t\} \vdash A \mathrel{\mathscr{R}} B \text{ implies } \{t\} \vdash A \gtrsim B.$$

We prove this by coinduction, i.e., we must show for all processes A and B,

$$\{t\} \vdash A \mathrel{\mathscr{R}} B \text{ implies } \{t\} \vdash AF(\mathscr{R})B.$$

We briefly comment on some of the cases, assuming the following $\mathscr{R}$-related processes,

$$\{t\} \vdash (\mathbb{D}[\![\mathscr{P}_1]\!]) \mathrel{\mathscr{R}} (\mathbb{D}[\![\mathscr{P}_2]\!])$$

where $\mathscr{P}_1 = \mathscr{G} \circ \mathit{fst}$ and $\mathscr{P}_2 = \mathscr{G} \circ \mathit{snd}$, for some $\mathscr{G}$ mapping from $\mathit{labels}(\mathbb{D})$ to R.

- *Threads* and *well-formed* are immediate from the definition of $\mathscr{R}$.
- *Reduction.* Consider reductions in $\mathbb{D}[\![\mathscr{P}_2]\!]$ that involve an occurrence of B_1 or B_2 from $\mathscr{P}_2$ in the context hole $[\![-]\!]_n$. The R-FIELD-WRITE rule, perhaps using R-SPECULATION-CONTEXT, is applied as follows.

- If $\mathscr{P}_2(n) = B_1$ then

  $$\mathbb{D}\llbracket \mathscr{P}_2 \rrbracket \rightarrow \mathbb{D}\llbracket \mathscr{P}_2 + \{n \mapsto t\,[p\,.f\texttt{=1}]B_2\}\rrbracket$$

  where $\mathscr{P}_2 + \{n \mapsto \cdots\}$ denotes the update of function $\mathscr{P}_2$ at domain element n. In this case, $\mathscr{P}_1(n) = A_1$, so

  $$\mathbb{D}\llbracket \mathscr{P}_1 \rrbracket (\rightarrow\rightarrow\geqq) \mathbb{D}\llbracket \mathscr{P}_1 + \{n \mapsto t\,[p\,.f\texttt{=1}]A_2\}\rrbracket.$$

  We obtain a new choice context $\mathbb{D}'$ by replacing $\llbracket - \rrbracket_n$ with $t\,[p\,.f\texttt{=1}]\,\llbracket - \rrbracket_n$, and thus get back to $\mathscr{R}$-related processes.

- If $\mathscr{P}_2(n) = B_2$ then

  $$\mathbb{D}\llbracket \mathscr{P}_2 \rrbracket \rightarrow \mathbb{D}\llbracket \mathscr{P}_2 + \{n \mapsto t\,[p\,.\,g\texttt{=1}]B_3\}\rrbracket$$

  In this case, $\mathscr{P}_1(n) = A_2$, so

  $$\mathbb{D}\llbracket \mathscr{P}_1 \rrbracket = \mathbb{D}\llbracket \mathscr{P}_1 + \{n \mapsto t\,[p\,.\,g\texttt{=1}]A_3\}\rrbracket.$$

  We obtain a new choice context $\mathbb{D}'$ by replacing $\llbracket - \rrbracket_n$ with $t\,[p\,.\,g\texttt{=1}]\,\llbracket - \rrbracket_n$, and thus get back to $\mathscr{R}$-related processes.

The only other possible reductions for $\mathbb{D}\llbracket \mathscr{P}_2 \rrbracket$ are R-SPECULATION-BEGIN and R-SPECULATION-END. Since B_1, B_2, B_3 are thread processes, such reductions occur entirely within $\mathbb{D}$, and can be applied to both $\mathbb{D}\llbracket \mathscr{P}_1 \rrbracket$ and $\mathbb{D}\llbracket \mathscr{P}_2 \rrbracket$ equally. By introducing new context holes for context holes duplicated in R-SPECULATION-BEGIN, we stay in $\mathscr{R}$.

– *Structural order.* Only thread processes are substituted into the RHS $\mathbb{D}\llbracket \mathscr{P}_2 \rrbracket$. Thus any structural reordering is confined to $\mathbb{D}$ being reordered to $\mathbb{D}'$ (subject to the processes substituted into $\mathbb{D}$ and $\mathbb{D}'$ having the same defined threads, which they do here), and thus there is a matching reordering on the LHS.

– *Equivalent top-level choices* and *equivalent actions/guards/returns.* Observe that B_3 has the only return on the RHS, and it is paired with A_3 which also has a return. These two cases follow immediately from the fact that the same $\mathbb{D}$ is used on both sides of $\mathbb{D}\llbracket \mathscr{P}_1 \rrbracket$ and $\mathbb{D}\llbracket \mathscr{P}_2 \rrbracket$, and there are no actions in the RHS processes B_1, B_2, B_3.

– *Environment writes.* Writes in other threads occur in $\mathbb{D}$ and thus apply to both LHS and RHS immediately.

– *Top-level lock removal.* None of the processes in R contain lock processes, so if a lock process is available to be removed, it must be a subprocess of $\mathbb{D}$ only (on both the LHS and RHS) and can be removed from both and stay in $\mathscr{R}$.

– *Top-level lock addition.* The set of $\mathbb{D}$ is closed under parallel composition with lock processes.

– *Environment locks.* For a thread $s \neq t$, any occurrence of $\texttt{lock}\ \ell\colon j \,|\, s\texttt{[}{\uparrow}\texttt{unit]}$ must be a subprocess of $\mathbb{D}$ only (so on both the LHS and RHS). Replacing occurrences of $\texttt{lock}\ \ell\colon j \,|\, s\texttt{[}{\uparrow}\texttt{unit]}$ in $\mathbb{D}$ with $\texttt{lock}\ \ell\colon j{+}1 \,|\, s[\ell\colon j]s\texttt{[}{\uparrow}\texttt{unit]}$ yields $\mathbb{D}'$ which yields a well-formed process after substitution whenever $\mathbb{D}$ yields a well-formed process after the same substitution.

D Further examples

Example 16. Consider the following program.

```
s[x=f; if(x){g=1;} ↑x] |
t[y=g; if(y){f=1;} ↑y]
```

The program is DRF. There is no execution that sets either f or g to 1 and therefore the outcome s[↑1]|t[↑1] is impossible.

To see how our semantics disallows the behavior, let us try to get the result by speculating that s⟨g=1⟩, where we show only t.

```
↠ ···
   ⊤⇒       t[y=g; if(y){f=1;} ↑y]
   [] s⟨g=1⟩⇒t[y=g; if(y){f=1;} ↑y]
↠ ···
   ⊤⇒       t[if(0){f=1;} ↑0]
   [] s⟨g=1⟩⇒t[if(1){f=1;} ↑1]
↠ ···
   ⊤⇒ t[↑0] [] t⟨g=1⟩⇒t[f=1]t[↑1]
```

Since the initial branch is not capable of generating the write t[f=1], the write from the final branch can not leave the speculation; therefore it is not visible to the other thread. If the write were visible then the initial branch could also produce the write, and the justifying write could then escape and finalize the speculation, causing a thin air read.

Note that if thread s is copied into the speculation, then the thin air reads can occur in the speculation; however, the speculation can never be finalized since the writes do not occur in the initial branch. □

Example 17 (Pugh (2004) §8). Consider the following program.

```
s[x=f; y=1+x*x-x; g=y; ↑(x,y)] |
t[z=g; f=z; ↑z]
```

The result s[↑(1,1)]|t[↑1] is possible via the speculation t⟨f=1⟩ s⟨g=1⟩. The initial process produces the writes s[g=1] t[f=1], in order. These writes can be matched by the final process allowing the speculation to be removed and giving the desired result. Pugh (2004) §1 and §2, among others, are similar. □

Example 18 (Pugh (2004) §5). Consider the following program.

```
s[x=f; g=x; ↑x] | u[h=1;] |
t[y=g; f=y; ↑y] | v[z=h; f=z; ↑z]
```

The result s[↑1]|t[↑1]|v[↑0] is not allowed by the JMM, nor our semantics. Any attempt to get the result must speculate that f or g is 1. Suppose we speculate t⟨f=1⟩ (the case for the speculation v⟨f=1⟩ is similar and is omitted). The initial branch can only achieve this by scheduling u[h=1], v[f=1], s[g=1] and t[f=1] in that order.

For `v` to produce the correct write, it must read `u[h=1]`. Since there is a cross-thread data dependency between each write and the next, the first three writes must be outside the parallel composition and therefore their relative order is fixed. This means that the final branch must match at least the first two action, in order. But this is not possible, since the final branch must write `v[f=0]` in order to get the desired return value for `v[↑0]`. As a result, the speculation can not be finalized with the desired return values. The same reasoning applies to Pugh (2004) §10 which uses control dependencies rather than data dependencies. □

Example 19 (Pugh (2004) §17). Consider the following program.

```
s[z=f; if(z!=1){f=1;} x=f; g=x; ↑(z,x)] |
t[y=g; f=y; ↑y]
```

The result `s[↑(1,1)]|t[↑1]` follows if we speculate `t⟨f=1⟩`.

The initial process can perform writes `s[f=1]` `s[g=1]` `t[f=1]`. The write `t[f=1]` depends on `s[g=1]`, but not on `s[f=1]`. Therefore the initial process may reach state `s[g=1]t[f=1](s[f=1]s[↑(0,1)]|t[↑1])`.

The final process can not produce the write `s[f=1]`, but it can reach state `s[g=1]` `t[f=1](s[↑(1,1)]|t[↑1])`. Since the initial and final branches agree, the writes can leave the speculation and the speculation can be finalized.

(Note that Pugh (2004) §19, which divides thread `s` in two, fails here as it does for the JMM (Sevcík 2008). The reasons are the same as in Example 6.) □

Example 20 (Sevcík (2008) §5.3.4). This example discusses irrelevant read introduction. Consider the following program.

```
s[x=h; if(x==0){z=f; if(z==1){g=1;}}
             else{w=f; g=x;} ↑x] |
t[f=1; y=g; h=y; ↑y]
```

The outcome `s[↑1]|t[↑1]` is allowed using nested speculation `s⟨g=1⟩` then `t⟨h=1⟩`. There are three terms in the nested speculation, all of which perform prefixes of the action sequence `t[f=1]s[g=1]t[h=1]`. The two initial branches both execute the true case of the conditional, resulting in `s[↑0]`. The final branch executes the false case, giving the desired result.

Removing the boxed statement has no effect on the execution. The result is also validated by our semantics when the irrelevant read is introduced.

The JMM allows the behavior with the boxed statement, but disallows it without (Sevcík 2008). □