A

Making the Java Memory Model Safe

ANDREAS LOCHBIHLER, Karlsruhe Institute of Technology

This work presents a machine-checked formalisation of the Java memory model and connects it to an operational semantics for Java and Java bytecode. For the whole model, I prove the data race freedom guarantee and type safety. The model extends previous formalisations by dynamic memory allocation, thread spawns and joins, infinite executions, the wait-notify mechanism, and thread interruption, all of which interact in subtle ways with the memory model. The formalisation resulted in numerous clarifications of and fixes to the existing JMM specification.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Operational Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent Programming Structures*

General Terms: Languages, Theory

Additional Key Words and Phrases: data race freedom, Java memory model, operational semantics, type safety

ACM Reference Format:
Lochbihler A. YYYY Making the Java Memory Model Safe ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 65 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Type safety and the Java security architecture distinguish the Java programming language from other mainstream programming languages like C and C++. Another distinctive feature of Java is its built-in support for multithreading and its memory model for executing threads in parallel [Gosling et al. 2005, §17]. To enable optimisations in compilers and hardware, the Java memory model (JMM) allows more behaviours than interleaving semantics. It therefore non-trivially interacts with type safety and Java's security guarantees. Although this is well-known [Gosling et al. 2005; Pugh 2000], their combination has never been considered formally.

Here, I present a machine-checked model of Java and the JMM called JinjaThreads for both Java source code and bytecode, and investigate the impact of the Java memory model on type safety and Java's security guarantees. In particular, my contributions are the following:

First, I present a *unified formalisation* of the axiomatic JMM based on the operational JinjaThreads semantics for Java source code and bytecode (§2). This provides the first rigorous link between a Java and the JMM, which several authors have cri-

Preliminary versions of §1.1, §2.2 to §2.4, §2.7, §3, and §7 have appeared in [Lochbihler 2012a]. The author's PhD thesis [Lochbihler 2012b] contains preliminary versions of the same sections plus §4 and §5.1. This work has been partially supported by the Deutsche Forschungsgemeinschaft under grants Sn11/10-1,2.
Author's address: A. Lochbihler, Institute of Information Security, ETH Zurich, 8092 Zurich, Switzerland.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0164-0925/YYYY/01-ARTA $15.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

tised as missing [Aspinall and Ševčík 2007a; Cenciarelli et al. 2007; Huisman and Petri 2007]. My model of Java builds on 15 years of formalised Java semantics, from Nipkow and von Oheimb [1998] to Lochbihler [2012b]. It features dynamic memory allocation, thread spawns and joins, the wait-notify mechanism, interruption, and infinite executions. Each of these is well-understood and has been formalised before, e.g., [Liu and Moore 2003; Petri and Huisman 2008; Farzan et al. 2004a; 2004b]. But their combination with the memory model is novel and results in subtle interactions, which previous JMM formalisations have missed [Aspinall and Ševčík 2007a; Cenciarelli et al. 2007; Huisman and Petri 2007; Boyland 2009]. I illustrate these cases with new examples and show how to deal with them. Dynamic allocation and the special treatment of memory initialisation in the JMM are the main complication in the definitions and proofs. In particular, it does not suffice to consider only finite (prefixes of infinite) executions. Coinductive definitions and proofs fortunately deal with terminating and infinite executions uniformly (§1.2 introduces the relevant concepts).

Second, I clarify the existing JMM specification and fix it where it is inadequate (see §6 for a summary). This draws on two sources: On the one hand, carefully working through all the subtle interactions between Java and the JMM exposes unclarities and inconsistencies. On the other hand, the safety properties below hold only after some fixes derived from analysing failed proof attempts.

Third, I formally prove the *data race freedom (DRF) guarantee* (§3): For correctly synchronised programs, the JMM guarantees interleaving semantics, which is also known as sequential consistency (SC) [Lamport 1979].[1] In other words: If a programmer makes sure that there are no data races (e.g., by using locks), then she can forget about the JMM and assume interleaving semantics.

In this work, I resolve the inconsistencies with initialisations of memory allocations in previous proofs [Manson et al. 2005; Huisman and Petri 2007]. By fixing the definition of data race [Jacobs 2005], I strengthen the formal guarantee such that it now holds also for programs that synchronise via volatiles, see Figs. 4 and 22 for examples. Moreover, I bridge the gap between the axiomatic style of the JMM and the operational JinjaThreads semantics. To that end, I identify the assumptions of the proof about the single-threaded semantics and discharge them using the above link. In particular, I explicitly construct sequentially consistent executions for a given prefix by corecursion. Hence, I am the first to prove the DRF guarantee unconditionally.

Forth, I prove that the JMM allows every execution that interleaving semantics produces (§4). This is the converse of the DRF guarantee, but also holds for programs with data races. Hence, despite its technical complexity, the JMM specification is consistent in that it defines some behaviour for every program, not just for correctly synchronised ones. This is non-trivial in the presence of data races and, to my knowledge, has not been proven before.

Fifth, I show *type safety* (§5), independent of correct synchronisation. Unfortunately, the axiomatic nature of the JMM is not suited for standard proofs using subject reduction, as Goto et al. [2012] have observed. Direct proofs must deal with reads retrieving type-incorrect values from the shared heap, because the JMM matches reads with writes a posteriori. To avoid the induced complications, I rather follow a two-step approach. I prove that each value read from memory during any legal execution is of the expected type (§5.2). Hence, the usual progress and preservation theorems [Wright and Felleisen 1994] may assume type-correct reads and, therefore, do not depend on

[1] A data race occurs when two confliciting accesses may happen concurrently, i.e., without synchronisation in between; two accesses to the same (non-volatile) location conflict if they originate from different threads and at least one writes. A program is correctly synchronised iff no SC execution contains a data race.

the memory model. In fact, I reuse the existing type-safety proofs from previous work [Lochbihler 2008] that were ignorant of the memory model.

Surprisingly, a weakness in the JMM specification allows pointer forgery. This breaks type safety (§5.1) when the JMM is combined with the standard run-time type system for Java [Drossopoulou and Eisenbach 1999], which stores type information inside the objects on the heap. I show that encoding type information directly in the references themselves rescues type safety. Thus, Java with the JMM is type safe and allows pointer forgery at the same time, which normally exclude each other. However, pointer forgery allows behaviours that break Java's security architecture (§5.4), independent of how type information is stored. This shows that the JMM really needs to be revised in some form; yet, it is unclear whether a quick fix is possible at all, as my examples suggest that the flaw is inherent to the committing style of the JMM.

Sixth, the JMM is very technical and subtle, and so are the proofs; machine support is therefore essential – as a series of false claims about the JMM and their subsequent disproof demonstrates [Manson et al. 2005; Cenciarelli et al. 2007; Ševčík and Aspinall 2008; Torlak et al. 2010]. All my definitions and proofs have been checked mechanically by the proof assistant Isabelle/HOL [Nipkow et al. 2002]. But this is not just the mechanisation. I have designed the proof structure carefully to be as abstract as possible and reasonable: an interface of signatures and assumptions connects the concurrent semantics and proofs to the single-threaded ones. This yields a tractable concurrency model that does do not rely on the specifics of the concrete language. I demonstrate this by specialising the results to both Java source code and bytecode, but they apply to other languages and memory models as well. For example, Boehm and Adve's proof [2008] of the DRF guarantee for C++ also postulates sequentially consistent completions; my corecursive construction fits there, too. Hence, the challenging proofs about the concurrent semantics have to be done just once. Moreover, the assumptions on the single-threaded semantics have the usual format of invariants and preservation. Thus, ordinary inductions suffice. This is a major improvement over the axiomatic style of memory model specifications.

The presentation focuses on the relevant parts of the concurrent semantics, omits most of the single-threaded semantics, and illustrates the subtleties rather with examples than formal definitions. A detailled description of the technical details can be found in the author's PhD thesis [Lochbihler 2012b]; the formalisation with all the ugly details of mechanised proofs is available online in the Archive of Formal Proofs [Lochbihler 2007]. In §1.1, I informally explain the JMM; the appendix summarises Java's concurrency primitives.

Throughout the article, I contrast my approach with others' and discuss the advantages and drawbacks using examples. Moreover, I thoroughly review the existing literature on the JMM (§7). Hence, this work can serve as a reference of the JMM and its relation to the semantics, as it collects the strengths and weaknesses of the current JMM. For the next revision of the JMM, this will be a valuable resource of what must not be missed.

1.1. Informal Introduction to the Java Memory Model

The Java memory model [Manson et al. 2005; Gosling et al. 2005, §17.4] specifies how shared memory behaves under concurrent accesses. This section sketches the main ideas behind the JMM.

1.1.1. Motivation. The program in Fig. 1a has two threads, each of which reads one of `C`'s static fields `x` and `y` into a local variable and subsequently sets the other to `1`. Figure 1b shows all possible interleavings of the two threads, and for each schedule, the final values for the threads' local variables `r1` and `r2`. All these schedules assume

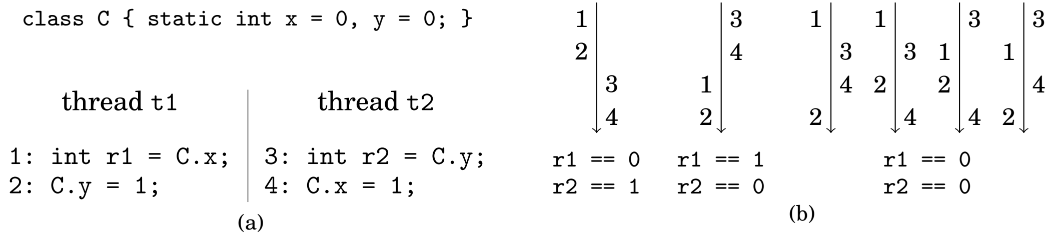


Fig. 1: Program with two threads (a) and all of its sequentially consistent schedules (b)

sequential consistency (SC) [Lamport 1979], which is considered the most intuitive memory model [Hill 1998]: There is a global notion of time, one thread executes at a time, and every write to a memory location immediately becomes visible to all threads. In particular, the result `r1 == r2 == 1` is impossible under SC as the following argument shows. Suppose it was possible. Then, l. 1 executes after l. 4 and l. 3 after l. 2. As l. 1 and l. 3 literally precede l. 2 and l. 4, respectively, one obtains the contradiction that l. 1 executes after l. 4 after l. 3 after l. 2 after l. 1.

For efficiency reasons, modern hardware implements memory models that are weaker than SC to allow for local caches and optimisations [Adve and Gharachorloo 1996; Sorin et al. 2011]. For example, if the threads `t1` and `t2` execute on different cores of a processor, the reads in ll. 1 and 3 might still be waiting for memory to return the values, when the writes in ll. 2 and 4 execute. If intra-processor cache communication is faster than memory, both pending reads may return the written values, i.e., 1. This results in `r1 == r2 == 1`, which is not possible under SC. Similarly, compiler optimisations might reorder the independent statements in each thread. Then, `r1 == r2 == 1` is possible for the transformed program even under SC. Therefore, a correct implementation of SC must take extra precautions: As the code does not provide any clues about how threads communicate via shared memory, it must either conservatively disable such optimisations in *all* code or laboriously analyse whether they are allowed [Sura et al. 2005]. To avoid the ensuing slow-down, the JMM relaxes SC and allows the outcome `r1 == r2 == 1` in the example.

Nevertheless, the JMM provides the intuitive SC semantics under additional assumptions – known as the data-race freedom guarantee [Adve and Hill 1990]. Two accesses to the same location *conflict* if

(1) they originate from different threads,
(2) at least one is a write, and
(3) the location is not explicitly declared as `volatile`.

A *data race* occurs if two conflicting accesses to a location may happen concurrently, i.e., without synchronisation in between. If the program contains no data races, the JMM promises that it behaves like under SC. In other words: If a programmer protects all accesses to shared data via locks or declares the fields as `volatile`, she can forget about the JMM and assume interleaving semantics, i.e., SC.

In the above example, there are two data races: the write of `C.y` in l. 2 races with the read in l. 3 and similarly l. 4 and l. 1 for `C.x`, i.e., the DRF guarantee does not apply. To eliminate these data races, one can use Java's synchronisation mechanisms, e.g., wrapping every line in its own `synchronized` block on `C`'s class object. Alternatively,

one can declare `C`'s static fields `x` and `y` as `volatile`, because accesses to such fields never conflict.[2]

1.1.2. Components of a JMM Execution. Since the JMM must ensure that compilers can implement Java on a variety of hardware with different MMs efficiently, it reduces concrete thread operations to events, which are called inter-thread actions in JMM terminology:

— reading (*Read*) from, writing (*Write*) to and initialising (*Alloc*)[3] a location on the heap,
— locking (*Lock*) and unlocking (*Unlock*) a monitor,
— interrupting (*Intr*) a thread and observing that it has been interrupted (*Intrd*),[4]
— spawning (*Spawn*) of and joining (*Join*) on a thread, and
— external actions (*IO*) – for input and output, for example, and
— *Start* and *Finish* to mark mark the start and termination of a thread, respectively.

This way, the JMM is independent from syntax and implementation techniques. It nevertheless gets a global view on how a given program works algorithmically and on how its threads interact, and uses this to determine the set of legal behaviours.

A JMM execution consists of a sequence of such events for each thread, three orders on these events (program order, happens-before order, and synchronisation order), and a function that assigns writes to reads.

Program order (notation $\leq_{\text{po}}$) totally orders the events of each thread according to their occurrence in the thread's sequence, but does not relate events from different threads.

The (partial) *happens-before order* $\leq_{\text{hb}}$ provides a notion of time relative to a given event α. As Fig. 2 illustrates, it partitions the other events of the execution into three groups: those that must have happened before it ($_ \leq_{\text{hb}} \alpha \wedge \alpha \not\leq_{\text{hb}} _$), those that must happen after it ($\alpha \leq_{\text{hb}} _ \wedge _ \not\leq_{\text{hb}} \alpha$), and those that may happen concurrently ($_ \not\leq_{\text{hb}} \alpha \wedge \alpha \not\leq_{\text{hb}} _$).[5] Since α's thread knows that all of its events prior to α must have happened and all posterior to α must not yet have happened, $\leq_{\text{hb}}$ always includes $\leq_{\text{po}}$. Additionally, synchronisation events, which are all events except for external actions and reads from and writes to non-volatile locations, introduce happens-before relationships between events of different threads. For example, spawning a thread t_1 (event *Spawn* t_1 _) synchronises with t_1's start event *Start*. In combination with program order, all events of the spawning thread before the spawning happen before any of t_1's events. Similarly, a write to a volatile location synchronises with a read that sees it, because the reading thread then knows that the write must have occurred before. The

[2]When a thread reads from a volatile field, it synchronises with all other threads that have written previously to that field. Hence, the reading thread can be sure that everything that should have happened in the other threads prior to their writes in fact has happened prior to its read. For the formal semantics, see §2.4.
[3]Technically, the JMM defines initialisation actions each of which initialises only a single location. JinjaThreads uses one event per memory allocation that initialises *all* members of the allocated object or array. This way, allocation events keep track of allocated addresses whereas JMM initialisation actions would not if the allocated object or array contains no members, e.g., an array of length 0. The special treatment of allocations in the JMM (see below) ensures that this deviation does not matter semantically. For clarity, I write *Init* `x` for initialising global variables in examples. Formally, a bootstrapping thread initialises such variables with allocations as explained in §2.3.
[4]The JMM list of inter-thread actions does not mention *Intr* and *Intrd* [Gosling et al. 2005, §17.4.2], but the definition of synchronisation points [Gosling et al. 2005, §17.4.4] includes interrupts and observing an interruption. My events *Intr* and *Intrd* model these two points.
[5]For technical reasons, the JMM's happens-before order $\leq_{\text{hb}}$ is reflexive, although "happens-before" would intuitively correspond to an irreflexive order. Since an event is never write and read action at the same time, this detail does not affect the semantics.

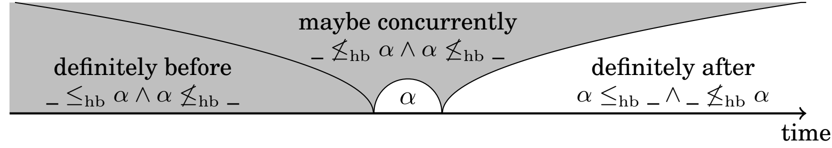


Fig. 2: Happens-before provides a notion of time relative to a given event α. If α is a *Read*, it may see *Write* events in the grey area.

Fig. 3: Program from Fig. 1a in simpler syntax (a) and its JMM execution for the result r1 == r2 == 1 (b).

synchronises-with order $\leq_{\mathrm{sw}}$ captures these relationships between events of different threads, see §2.4 for the formal definition.

Whenever α's thread cannot deduce – using only allowed means of synchronisation – that an event β of another thread must have happened before or will happen after α, then α and β may happen concurrently. In particular, the thread must not make any assumption about the relative order of two events that happen concurrently. This permits compilers and hardware to freely reorder independent statements of a thread without synchronisation in between.

Finally, the synchronisation order $\leq_{\mathrm{so}}$ totally orders all synchronisation events and must be consistent with happens-before. It models a global time on synchronisation events on which all threads must agree.

Since the JMM is independent from a concrete language and sequential semantics, it is custom to write examples such as in Fig. 1a in a simple imperative language (Fig. 3a) rather than to obfuscate the point by irrelevant Java details. In this language, thread-local variables start with "r", e.g., `r1`, `r2`, whereas `x`, `y`, etc. denote shared locations. In examples, vertical rules separate the threads, and the thread in column i has ID t_i. Above the threads, the initial values of shared locations and any necessary declarations are given.

Fig. 3b shows how executions are depicted. The threads are abstracted to events – labelled with the thread ID – and orders. Solid arrows $\longrightarrow$ represent program order, transitive relationships are not shown. The dashed arrows $\dashrightarrow$ denote the flow of values from writes to reads; the *write-seen function* of an execution assigns to each read event the write event it sees. Dotted arrows $\cdots\rightarrow$ used in later examples denote synchronisation (synchronises-with relationships).

The JMM requires that the write-seen function *respects happens-before* in the following sense: A read α may see a write β that happens before or may happen concurrently (grey area in Fig. 2), but the write must not happen after the read. Moreover, there must not be another write γ to that location that is known to happen between α and β, i.e., $\beta \leq_{\mathrm{hb}} \gamma \leq_{\mathrm{hb}} \alpha$.

The execution shown in Fig. 3b results in `r1 == r2 == 1`, which SC does not allow. As there is no synchronisation, happens-before coincides with program order. Hence, ll. 1 and 2 may happen concurrently with ll. 3 and 4. Therefore, l. 1 and l. 3 are allowed to see the writes from l. 4 and l. 2, respectively. In particular, the thread on the left

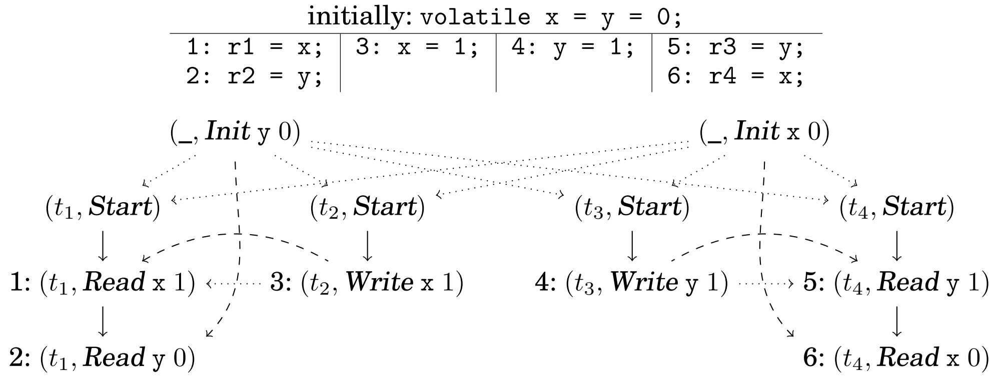


Fig. 4: The classic independent reads of independent writes (IRIW) example

must not deduce that l. 3 must have already executed from the fact that l. 1 reads the value 1 from l. 4, because there is no synchronisation involved.

These constraints alone, which happens-before imposes on the visibility of writes, are insufficient to enforce global time. Figure 4 shows the classic independent reads of independent writes (IRIW) example [Boehm and Adve 2008]. In the execution depicted, t_1 perceives l. 3 to execute *before* l. 4, because its reads see the former write, but y's initialisation instead of the latter write. Conversely, t_4's reads see l. 4 and x's initialisation, but not l. 3. Hence, it appears to t_4 as if l. 3 executes *after* l. 4. Such a result is possible if, e.g., t_2 and t_3 execute simultaneously on different cores such that t_2's core propagates the write to t_1's core faster than to t_4's and conversely t_3's core communicates faster with t_4's than with t_1's (e.g., because they share their caches). However, no SC execution can produce such a result. By the DRF guarantee, the JMM must not allow this result either, because there are no data races (all shared locations `x` and `y` are marked as volatile). This is why volatile reads must also respect the synchronisation order analogously to happens-before. In Fig. 4, there is no such synchronisation order, because either l. 3 or l. 4 would be an intervening write between the initialisation of `x` or `y` and the read in l. 2 or l. 6, respectively.

1.1.3. Values Out of Thin Air. The constraints from happens-before and synchronisation order capture the JMM notion of a *well-formed execution*. However, they are still too weak for programs with data races. Consider, e.g., the program in Fig. 5a. In this program, the threads merely copy `x` to `y` and vice versa; the thread-local assignment in l. 3 has no effect and could well be removed. So one would expect `r1 == r2 == 0` to be the only possible result. However, a queer compiler might eliminate the local variable `r2` in the thread on the right: `x = 1; x = y;` is a correct implementation in a sequential setting, but in parallel with the thread on the left, the result `r1 == 1` becomes possible even under interleaving semantics (e.g., schedule `x = 1; r1 = x; y = r1; x = y;`). As the original program cannot normally produce 1, 1 appears *out of thin air*. Yet, the constraints mentioned so far do not forbid this behaviour even for the original program. The reads in ll. 1 and 3 may see the writes in ll. 4 and 2, respectively, as they may happen concurrently and there is no synchronisation at all. Thus, Fig. 5b shows a well-formed execution for Fig. 5a.

For type safety and Java's security guarantees, it is vital that values do not appear out of thin air [Pugh 2000]. Otherwise, malicious code could exploit this to forge a pointer to an object to which it must not gain access or which it can then access in a type-unsafe fashion. For example, if l. 3 in Fig. 5a stored in `r2` such a pointer instead

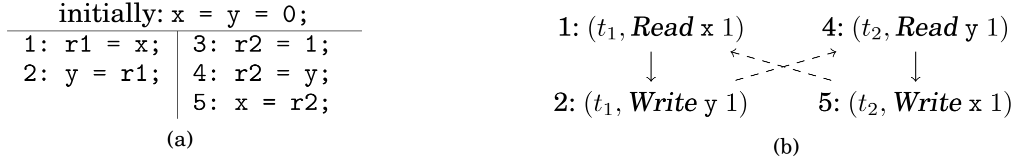


Fig. 5: JMM causality test case 4 with l. 3 added: the value 1 appears out of thin air

of the value 1, such optimisations would enable the thread on the left to gain access to the pointer. Therefore, the JMM must forbid values appearing out of thin air. But note that the executions in Figs. 3b and 5b are identical when viewed in isolation. Yet, the JMM only allows the former. Thus, one cannot decide whether an execution is allowed by looking solely at the execution. In fact, being allowed is a second-order property.

To ban self-justifying speculations (and distinguish Fig. 3b from Fig. 5b), the JMM adds a causality condition called *legality*: Reads that see concurrent writes must be committed, i.e., there must be a justifying JMM execution that produces the same value, but the read event α sees a write event β that happens before it ($\beta \leq_{\mathrm{hb}} \alpha$). This causality condition distinguishes the JMM from memory models of other languages like C++, where concurrent reads and writes immediately result in undefined behaviour. For Fig. 5a, causality forbids `r1 == r2 == 1`, because no execution can produce the value 1 without having both reads see the concurrent writes. In contrast, it accepts Fig. 3b, because Fig. 3a has another execution in which ll. 2 and 4 write 1 even if ll. 1 and 3 see the initialisations. The important thing to note is that at the basis of any sequence of justifying executions, there is one in which all reads see writes that happen before them. Such an execution is called *well-behaved*.

This is where memory initialisations come into play; the JMM assumes that all locations are initialised to their default value. To ensure that such a basis for justifying executions always exists, these initialisation events are defined to happen before any other event, i.e., conceptually at the start of the execution. Thus, there is always at least one suitable write that happens before any given read.

The details of the causality condition are the most complex part of the JMM. In fact, there are two versions: Aspinall and Ševčík [2007a] weakened the original condition such that more optimisations are possible while maintaining the DRF guarantee. I have formalised both and the theorems hold for both.

1.2. A Note on Coinduction

My formalisation heavily uses coinductive definitions and proofs. They provide an elegant way to handle finite and infinite executions uniformly. To make the formalisation and proofs more accessible, I now introduce the coinductive concepts that are used and compare them to their inductive counterparts, using a simple example. Readers familiar with coinduction may skip this section.

1.2.1. Coinductive Definitions. Like an inductive definition, a coinductive definition is given by inference rules; I use double horizontal bars to distinguish them from inductive ones. Formally, the rules are interpreted as a fixed point of the associated (monotone) functional $\mathcal{F}$: the least for inductive ones and the greatest fixed point (gfp) for coinductive ones.

The least fixed point (lfp) yields the *smallest* set that is closed under the rules. Hence, for each element, there is a finite derivation tree using only the introduction rules. In this sense, inductive definitions contain only finite elements. Therefore, one can use

induction to prove that all elements satisfy a property – abstractly, the inductive proof shows that the property is closed under the rules, too.

For example, given a labelled transition system (LTS), the following defines the family $(\Pi_s)_s$ of sets of all maximal paths that start in a given state s.

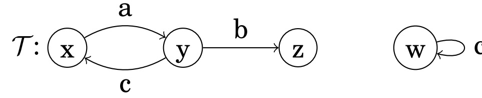
$$\mathrm{P}_1: \frac{s \not\rightarrow}{[] \in \Pi_s} \qquad\qquad \mathrm{P}_2: \frac{s \xrightarrow{l} s' \qquad \pi \in \Pi_{s'}}{l \cdot \pi \in \Pi_s}$$

Here, $s \xrightarrow{l} s'$ denotes that state s can transition to s' with label l. A path is modelled as a word of labels; it is maximal iff it is infinite or it leads to a state s from which no transition is possible (notation $s \not\rightarrow$). This example simplifies the actual construction I use for connecting Java with the JMM (§2.3): the LTS corresponds to the interleaved small-step semantics and the paths to traces of an execution. The monotone functional $\mathcal{F}$ associated with P_1 and P_2 transforms a family $A = (A_s)_s$ of sets to the family $(\mathcal{F}(A)_s)_s$ as follows:

$$\mathcal{F}(A)_s = \begin{cases} \{\, [] \,\} & \text{if } s \not\rightarrow \\ \{\, l \cdot\ \pi.\ \exists s'.\ s \xrightarrow{l} s' \wedge \pi \in A_{s'} \,\} & \text{otherwise} \end{cases}$$

When P_1 and P_2 are interpreted inductively, $\Pi_s = \mathit{lfp}\ \mathcal{F}$ consists of all *finite* maximal paths that start in s. The paths are finite, because each rule application (i.e., every step in $\mathcal{F}$'s fixed point iteration) adds just one label to the path. Hence, the inductive interpretation covers only finite, terminating executions.

For the following transition system $\mathcal{T}$, e.g., Π_x consists of the paths $(\mathrm{ac})^n\mathrm{ab}$ where n is any natural number. Here, I have omitted the cons operator $\cdot$ and list brackets; π^n denotes n repetitions of π and π^∞ infinitely many. Inductively, one however obtains $\Pi_\mathrm{w} = \emptyset$, because the only maximal path c^∞ from w is infinite.



In contrast, the greatest fixed point $\mathit{gfp}\ \mathcal{F}$, i.e., the coinductive interpretation, denotes the *greatest* set all of whose elements are justified by one of the inference rules – Pierce [2002] calls such sets consistent. It excludes only elements for which one can prove (in a finite derivation) that they do not belong to the set. Of course, the gfp contains all the finite elements with finite derivation trees. In addition, it typically includes infinite elements backed by infinite derivation trees. Hence, there is no rule for (structural) induction.

1.2.2. Coinduction and Infinite Derivation Trees. To show membership of an infinite element x, one uses coinduction: find a set A with $x \in A$ such that A is consistent, i.e., all of A's elements are justified by one of the rules (formally $A \subseteq \mathcal{F}(A)$). This proof principle is derived from greatestness: the gfp is the greatest consistent set and therefore a superset of A; in particular, it contains x. For example, $\mathrm{c}^\infty \in \Pi_\mathrm{w}$ under the coinductive interpretation of P_1 and P_2: Choose $A_\mathrm{w} = \{\, \mathrm{c}^\infty \,\}$ and $A_s = \emptyset$ for $s \neq \mathrm{w}$. I must show that whenever $\pi \in A_s$, either $s \not\rightarrow$ and $\pi = []$, or π has the form $l \cdot \pi'$ such that $s \xrightarrow{l} s'$ and $\pi' \in A_{s'}$ for some s'. The latter clearly holds for the chosen $(A_s)_s$: $\mathrm{c}^\infty = \mathrm{c} \cdot \mathrm{c}^\infty$ and $\mathrm{w} \xrightarrow{\mathrm{c}} \mathrm{w}$. Similarly, Π_x contains also $(\mathrm{ac})^\infty$ under the coinductive interpretation (provable with a different choice for $(A_s)_s$).

Note that (in)finiteness describes the derivation trees, not the elements themselves. In the example, both rules add one constructor. So, a path is finite iff it has a finite

derivation. But this does not hold in general. Suppose, e.g., that the label c signifies an internal computation that should not show up in a trace. The following attempt defines the family $(\Pi'_s)_s$ of sets of all maximal paths from s from in which the label c has been purged. It splits the rule P_2 into P'_2 and P'_{D}. If the next transition has label c, P'_{D} does not cons the label. This breaks the correspondence between constructors and derivation steps.

$$\mathrm{P}'_1: \frac{s \not\to}{[] \in \Pi'_s} \qquad \mathrm{P}'_2: \frac{s \xrightarrow{l} s' \qquad \pi \in \Pi'_{s'} \qquad l \neq \mathrm{c}}{l \cdot \pi \in \Pi'_s} \qquad \mathrm{P}'_{\mathrm{D}}: \frac{s \xrightarrow{\mathrm{c}} s' \qquad \pi \in \Pi'_{s'}}{\pi \in \Pi'_s}$$

Now, $[] \in \Pi'_{\mathrm{w}}$ is a path of finite length, but its (only) derivation is infinite, namely infinitely many applications of P'_{D}. Formally, this is again shown by coinduction: choose $A_{\mathrm{w}} = \{ [] \}$ and $A_s = \emptyset$ for $s \neq \mathrm{w}$; justify $[] \in A_{\mathrm{w}}$ with P'_{D}: $\mathrm{w} \xrightarrow{\mathrm{c}} \mathrm{w}$ and $[] \in A_{\mathrm{w}}$. Computationally, it is sensible to consider $[]$ as infinite, because it models a non-terminating computation with only internal steps.

For both $\mathrm{c}^\infty \in \Pi_{\mathrm{w}}$ and $[] \in \Pi'_{\mathrm{w}}$, the justifying element in $A_{s'}$ was c^∞ and $[]$, respectively. That is, they justified themselves. For $(\Pi_s)_s$, self-justification is unproblematic, because all rules add constructors. Hence, the infinite derivation traverses all of c^∞ – it is a mere coincidence that c^∞ is equal to its tail. For $[] \in \Pi'_{\mathrm{w}}$, this argument fails, because P'_{D} applies infinitely often without changing the path, i.e., the infinite derivation completely ignores the structure of $[]$. In fact, Π_{w} contains junk: coinduction similarly shows that any path π is a member of Π'_{w}, not just $\pi = []$. Thus, P'_1, P'_2, and P'_{D} are wrong (although they would be fine for an inductive definition).

In fact, it is hard to characterise $(\Pi'_s)_s$ coinductively, but there is an elegant way out: Define $\Pi'_s = \{ \mathit{purge}\ \pi.\ \pi \in \Pi_s \}$ where the function *purge* (defined corecursively below) removes all occurrences of c from a list. Single-valuedness ensures that no junk is introduced. Hence, $\Pi'_{\mathrm{w}} = \{ [] \}$ as desired. In §2.3, I use a similar approach to remove unobservable transitions in traces. (To indicate that a variable is a list, I often draw a bar across the variable like in $\overline{x}$; list variables $\overline{x}$ are distinct from those without bars, e.g., x, which usually denote an element of such a list.)

$$\begin{aligned} \mathit{purge}\ \overline{x} &= \mathit{purge}'\ (\mathit{dropWhile}\ (\lambda y.\ y = \mathrm{c})\ \overline{x}) \\ \mathit{purge}'\ [] &= [] \\ \mathit{purge}'\ (x \cdot \overline{x}) &= x \cdot \mathit{purge}'\ (\mathit{dropWhile}\ (\lambda y.\ y = \mathrm{c})\ \overline{x}) \end{aligned}$$

As this definition steps out of the language of coinduction, coinduction as proof principle no longer works out of the box. For example, *no-c* $\overline{x}$ defined by NC_1 and NC_2 formalises that the list $\overline{x}$ does not contain c. Let me show that all paths in Π'_s in fact do not contain c, i.e., that *no-c* (*purge* π) holds for any $\pi \in \Pi_s$.[6]

$$\mathrm{NC}_1: \frac{}{\textit{no-c}\ []} \qquad \mathrm{NC}_2: \frac{\textit{no-c}\ \overline{x} \qquad x \neq \mathrm{c}}{\textit{no-c}\ (x \cdot \overline{x})}$$

I must use coinduction, as π and *purge* π may be infinite, i.e., induction is not available. Note that in the coinductive step, every justification with NC_2 must produce one label of the (purged) path. However, *dropWhile* in *purge*'s definition may gobble arbitrarily many labels of π. Hence, such a justification requires unbounded search for the state that produces the next label other than c in π.

[6]Actually, *no-c* (*purge* $\overline{x}$) holds for any list $\overline{x}$, not just paths $\pi \in \Pi_s$. Nevertheless, to illustrate the idea of termination parameters, I thread the premise $\pi \in \Pi_s$ through the coinductive proof.

1.2.3. Termination Parameters. To avoid complicated proofs, I introduce a termination parameter a which is taken from a well-founded order $(R, \prec)$. This way, I can delay the justification in the coinductive step provided that a decreases in $\prec$.[7] The parametrised *no-c*$_a$ adds to *no-c*'s rules the delay rule NC'_D. Its premise $a' \prec a$ and well-foundedness ensure that any derivation tree applies the rule NC'_D only finitely many times in a row – this avoids the junk due to self-justifications. As the other recursive rule NC'_2 adds a constructor, the termination parameter may be chosen anew after each application.

$$\mathrm{NC}'_1\colon \frac{}{\textit{no-c}_a\ []} \qquad \mathrm{NC}'_2\colon \frac{\textit{no-c}_{a'}\ \overline{x} \qquad x \neq \mathsf{c}}{\textit{no-c}_a\ (x \cdot \overline{x})} \qquad \mathrm{NC}'_D\colon \frac{\textit{no-c}_{a'}\ \overline{x} \qquad a' \prec a}{\textit{no-c}_a\ \overline{x}}$$

Then, *no-c* $\overline{x}$ iff *no-c*$_a$ $\overline{x}$ for some $a \in R$. Thus, I can use *no-c*$_a$'s coinduction principle to prove that $\pi \in \Pi'_s$ does not contain c. For $\prec$, I take as measure the length of π's prefix of c labels. Hence, when the first label in π is c (and I normally would have to start an unbounded search to gobble all c labels), I merely drop the first label and I use NC'_D for justification, as the measure decreases.

Formally, Isabelle/HOL represents the coinduction principle as an ordinary higher-order theorem. Hence, I do not need to define *no-c*$_a$ explicitly and prove equivalence to *no-c*. It suffices to prove the new coinduction principle with delay as a theorem like it would be generated for *no-c*$_a$. This proof is a trivial induction on well-foundedness and can be easily automated. In this work, I use delayed coinduction when building sequentially consistent completions (Lem. 3.20) and well-behaved executions.

2. LINKING JAVA WITH THE JAVA MEMORY MODEL

This section describes how to formally connect Java and Java bytecode with the Java memory model. I build on the JinjaThreads model of Java and Java bytecode, which I have described in detail in previous work [Lochbihler 2007; 2008; 2010; 2012b; Lochbihler and Bulwahn 2011]. It covers a substantial subset of sequential Java and Java bytecode, e.g., local variables and assignments, objects and fields, inheritance, dynamic dispatch, recursion, arrays, exception handling, standard control structures, and native methods. It models the following multithreading features according to the Java language specification (JLS) [Gosling et al. 2005]: arbitrary thread creation, synchronisation through monitors, the wait-notify mechanism, joining on threads, thread interruption, and volatile fields. JinjaThreads is executable [Lochbihler and Bulwahn 2011] and has been validated by running Java test programs [Lochbihler 2012b].

To favor reuse and sharing, I have organised the semantics in a stack (Fig. 6). It falls into two parts: Layers 1 to 4 define the single-threaded semantics and layers 5 to 7 the multithreaded ones. This separation nicely disentangles all intra-thread issues from concurrency; both parts only communicate using events à la JMM inter-thread actions. Similarly, each layer connects to the layer above through an interface of signatures and assumptions, on which the definitions and proofs rely. Moreover, this structure favors reuse between the source code and bytecode: their formalisations differ only in layer 3, which defines the semantics of the language primitives.

The presentation omits some uninteresting details of the semantics. Instead, I focus on the following challenges:

[7] These termination parameters are orthogonal to up-to techniques that are common in bisimulation proofs [Sangiorgi 1998; Hur et al. 2013]. Up-to techniques allow to pick the justification of elements in the coinduction invariant from a larger set, but still require that the elements are immediately justified according to one of the rules. In contrast, my parametrisation over a well-founded order allows to defer *before* committing to one rule. In the following *no-c* example, up-to techniques do not help to prove c being absent in paths taken from Π'_s, because *dropWhile* delays the decision between NC_1 and NC_2 for arbitrarily many steps. Without parametrisation, the coinduction proof requires a nested induction.

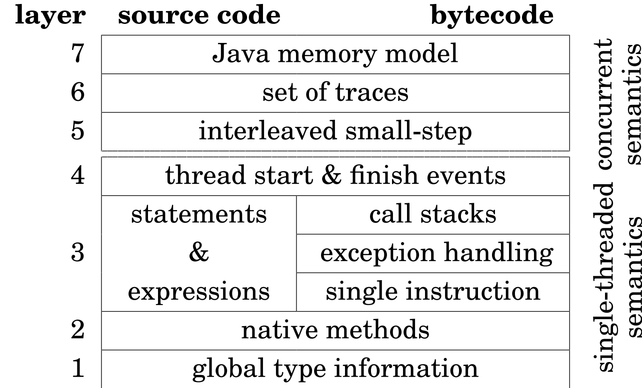


Fig. 6: JinjaThreads stack of semantics

— storing runtime type information and allocating fresh memory (§2.1, §2.6, §2.7),
— deriving event traces from the single-threaded semantics (§2.2, §2.3),
— formalising the JMM (§2.4)
— determining the identity of an event (§2.5),
— dealing with non-termination (§2.8), and
— modelling the wait-notify mechanism and interruption (§2.9, §2.10).

2.1. Type Information and Fresh Addresses

In the informal introduction §1.1, I have only mentioned values in heap locations. Yet, a Java implementation must also exchange runtime type information, array lengths,[8] and freshness of addresses between the threads. According to the JLS [Gosling et al. 2005, §17.4.5], checked type casts, virtual method calls, and reading the length of an array are not part of the inter-thread actions and thus not affected by the JMM; reading types and array lengths must always return the correct data. Therefore, the JMM provides the strong model of sequential consistency for type information, and array lengths.

To enforce SC for types, I implement allocation and type information as a global state. The multithreaded semantics passes this global state from one thread to the other during execution like in interleaving semantics. Moreover, I define abstract operations to query and update the global state and specify their properties. Hence, I am able to state clearly what assumptions type safety and the DRF guarantee rely on. Along the way, this addresses a question pointed out by Aspinall and Ševčík [2007a]: What does it mean for an address being fresh for memory allocation?

The interface to the shared state consists of three operations:

(1) The partial function *typeof-addr* σ a extracts from the state σ the type information for address a, i.e., the type of the object at a and possibly its array length.
(2) *empty-*σ denotes the initial state when no objects have been allocated.
(3) The non-deterministic function *alloc* σ T allocates a new object or array of type T[9] and returns its fresh address and the updated state. To model non-determinism, *alloc* returns a set of pairs of addresses and state such that the higher levels can

[8]Although the JLS specifies that every array has a final field `length` [Gosling et al. 2005, §6.4.5] that stores its length, the JMM treats array lengths specially [Gosling et al. 2005, §17.4.5].
[9]I assume that the type of an array also specifies its length.

```
class A { void m() {} }   initially: x = y = null;
1: r1 = x;                      | 5: r4 = y;
2: if (r1 != null) r1.m();      | 6: x = r4;
3: r3 = new A();                |
4: y = r3;                      |
```

Fig. 7: Dynamic dispatch in l. 2 requires type information which is not yet available

propagate the non-determinism – I discuss the need for non-determinism in §2.6. If all addresses are already allocated, the set is empty.

The implementation must ensure two properties:

(A1) *alloc* is correct, i.e., if $(\sigma', a) \in$ ***alloc*** $\sigma\ T$, then σ' stores type T for a, i.e., ***typeof-addr*** $\sigma'\ a = \lfloor T \rfloor$ (where $\lfloor_\rfloor$ denotes definedness).
(A2) *alloc* only extends the type information in σ, i.e., if ***typeof-addr*** $\sigma\ a = \lfloor T \rfloor$ and $(\sigma', a') \in$ ***alloc*** $\sigma\ T'$, then ***typeof-addr*** $\sigma'\ a = \lfloor T \rfloor$.

In the following, I implement this specification *twice* (§2.1.1 and §2.1.2).

2.1.1. Dynamic Type Information. The first implementation stores the type of objects in the objects themselves. This follows standard practice in formalising Java [Alves-Foss 1999; Stärk et al. 2001; Liu and Moore 2003; Klein and Nipkow 2006; Farzan et al. 2004b; Petri and Huisman 2008]. The global type information σ is like a shared heap (a map) except that it stores for every allocated address instead of the object itself only its type and possibly the array length, but no field values. Field values are irrelevant, because the JMM determines them. I implement the operations as follows: ***typeof-addr*** $\sigma\ a$ merely returns the information that σ stores for a; ***empty-***σ is the everywhere undefined map; and ***alloc*** $\sigma\ T$ returns the set of all addresses fresh in σ and pairs each such address a with σ updated to store T for a. An address is fresh in state σ iff σ contains no type information for it. This implementation is straightforward and satisfies the above specification (A1 and A2).

However, programs with data races may have type-unsafe executions. In the following two examples, the program gets stuck or subject reduction fails. They both exploit that the type of an address is only determined upon allocation, but not when it is first used. Nevertheless, the DRF guarantee applies (§3) and ensures that programs without data races are type safe like under SC.

First, when the type of an address is not yet known, the defensive (source code) semantics can get stuck and the aggressive virtual machine (VM) behaves in an undefined way. The example in Fig. 7 has data races on `x` and `y`. The JMM allows that l. 1 reads the address a of the object allocated in l. 3 via the detour of the second thread, because an optimising compiler might move ll. 3 and 4 before l. 1. However, the semantics does not anticipate such optimisations, but executes the program as it is. Hence, when l. 2 calls `m` on a, the defensive source code semantics gets stuck and the aggressive VM calls an unspecified method, because a's type information used for dynamic dispatch is still undefined.

A temptingly simple measure would be to restrict reading such that only allocated addresses may be read from memory. However, the semantics then misses some legal JMM behaviours, because this restriction prohibits reordering with memory allocations. In Fig. 7, e.g., the JMM allows `r1 == y` in the final state, because compilers are allowed to move l. 1 past the independent statements in ll. 3 and 4. However, the restricted semantics cannot produce this result because the read in l. 1 always executes before the allocation in l. 3, i.e., it can never return the address to be allocated.

```
            class C {}   class D {}
   initially: b = false; x = y = null;
1: r1 = y; | 3: r2 = x;           | 9: b = true;
2: x = r1; | 4: if (b)            |
           | 5:   r3 = new C();   |
           | 6: else              |
           | 7:   r2 = new D();   |
           | 8: y = r2;           |
```

Fig. 8: A program with a legal execution where `r2` of type `D` references a `C` object

Second, reading an address before its type is determined may also compromise subject reduction. The program in Fig. 8 is type correct if it declares `x`, `y` and `r2` of type `D`. However, it has a legal execution where they reference a `C` object (for a detailled derivation, see Fig. 27). The problem here is that the type of an address may vary across justifying executions. The allocation operation *alloc* may pick the *same* address a for the allocations in ll. 5 and 7, because only one of them occurs in any one execution. Hence, it is l. 4 that decides on the type to allocate – based on whether it sees the initialisation `b = false` or the concurrent write `b = true`. However, l. 3 may have already read a and stored it in `r2` of static type `D`. Then, if l. 5 allocates a `C` object, type safety is broken.

2.1.2. Static Types for Addresses. To tackle the above type safety issues, I define the following alternative implementation of the operations. It is motivated by the insight that the core of the above problems is that the type of an address is only determined upon allocation, but not when it is first used.

Now, it is the address itself that stores type information for the address. Hence, an address $a = (T, n)$ consists of its type information T and a sequence number n to distinguish objects of the same type. The type and array length of an address is the information stored in the address, i.e., *typeof-addr* $\sigma\ (T, n) = \lfloor T \rfloor$. In particular, type information for every address is available from the start. Hence, the programs in Figs. 7 and 8 are unproblematic, because with this implementation,

(1) dynamic dispatch only requires correct type information, which is now available independent of allocations, and
(2) the allocations in ll. 5 and 7 always return different addresses.

Now that I have stripped type information off the shared state, it only needs to remember which addresses are fresh for allocation. Hence, the state maps type information to the set of sequence numbers of objects of that type that have already been allocated. To allocate an object of type T in state σ, allocation produces a pair $(\sigma', (T, n))$ for every sequence number $n \notin \sigma\ T$ where σ' updates σ to store $\sigma\ T - \{n\}$ at T. Obviously, this implementation satisfies the specification A1 and A2, too.

Apart from supporting type safety, this implementation exposes a hidden communication channel via type information from which the previous suffered. For example,

```
       class A implements I { int f() { return 0; } }
       class B implements I { int f() { return 1; } }
   interface I { int f(); }     initially: x = 0; y = null;
1: r1 = x;                              | 4: x = 1; | 5: r3 = y;
2: r2 = (r1 == 0 ? new A() : new B());  |           | 6: r4 = r3.f();
3: y = r2;                              |           |
```
(P1)

Classes `A` and `B` inherit method `f()` from their common interface `I`. When l. 5 sees l. 3, dynamic dispatch at l. 6 tells the thread on the right about the left thread's local variable `r1`, although there is no synchronisation involved. In the previous implementation, the allocation in l. 2 returns the same address value, no matter whether `A` or `B` is allocated. Hence, from the point of view of events, the thread on the right only reads an address (in fact the same value in both cases), but behaves differently. In contrast, the second implementation allocates `A`'s objects at different addresses than `B`'s. Hence, the value that l. 5 reads completely determines the call target in l. 6. Analogously, threads can communicate through array lengths instead of types, see Fig. 25 for an example. This is why I treat array lengths as part of the type information – arrays of different lengths have distinct addresses.

However, there is also a disadvantage over the previous implementation. Since type information partitions the address space, each read or write of an address value not only transfers a pointer value as on standard hardware, but simultaneously does so for the complete run-time type information of the object it references. From an implementation point of view, this is unrealistic.

2.2. Thread Management Actions

As the global state needs to be passed between the threads, threads cannot execute in isolation, as the JMM suggests [Gosling et al. 2005, §17.4]. Instead, I compute their interleavings, which guarantees sequential consistency for the shared type information. My interleaving semantics also takes care of mutual exclusion for locks and manages the monitor wait sets, notifications, spawns of and joining on threads, and thread interruption.

The single-threaded semantics has the form $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$. Local states of thread t are denoted by x and x', and σ, σ' are the (global) type information that all threads share. Source code defines a small-step semantics, the VM for bytecode uses a functional style. Both semantics are standard except for three aspects:

(1) They can access only the local state of the current thread and the shared state of type information, but not, e.g., the thread pool, the wait sets, or other thread's local states. Instead, they use a list $\overline{\alpha}$ of events to communicate with the interleaving semantics. This separates the concurrency features from the sequential aspects such that I can use the same interleaving semantics for Java source code and bytecode. Figure 9 shows the source code reduction rules for calls to *Thread*'s native methods *start*, *interrupt*, and *isInterrupted*. As can be seen, these rules merely translate the method calls to appropriate events for the multithreaded semantics. I will discuss the specific events below.

(2) As there is no shared state that stores the values of object fields and array cells, the thread does not know what value should be read. Non-determinism solves this. When a thread needs to read a field or array cell, there is a separate transition for every value that could be read, even for type-incorrect ones. Although this produces many impossible reductions, the JMM well-formedness and legality constraints will later select the right ones. To that end, read and write operations are recorded in the list $\overline{\alpha}$, too. For example, the rules for reading and writing field $C{::}F$[10] of the object at address a are as follows. Note that the shared state σ does not constrain value v at all.

$$\text{FACC:}\quad t \vdash (\textit{\textbf{addr}}\ a.C{::}F, \sigma) -[\textit{\textbf{Read}}\ (a, C{::}F)\ v]\rightarrow (\textit{\textbf{Val}}\ v, \sigma)$$

$$\text{FASS:}\quad t \vdash (\textit{\textbf{addr}}\ a.C{::}F := \textit{\textbf{Val}}\ v, \sigma) -[\textit{\textbf{Write}}\ (a, C{::}F)\ v]\rightarrow (\textbf{unit}, \sigma)$$

[10]To distinguish hidden fields, a field is labelled with the class that declares it, e.g., $C{::}F$ denotes F declared in C.

SP: $t \vdash (\textit{addr } a.\textit{start}([]), \sigma) -[\textit{Spawn } a\ (C, \textit{run}, a)]\rightarrow (\textit{unit}, \sigma)$

SPF: $t \vdash (\textit{addr } a.\textit{start}([]), \sigma) -[\textit{ThreadEx } a\ \textit{True}]\rightarrow (\textit{throw IllegalThreadState}, \sigma)$

INTR: $t \vdash (\textit{addr } a.\textit{interrupt}([]), \sigma) -[\textit{ThreadEx } a\ \textit{True}, \textit{WakeUp } a, \textit{Intr } a]\rightarrow (\textit{unit}, \sigma)$

INTRINEX: $t \vdash (\textit{addr } a.\textit{interrupt}([]), \sigma) -[\textit{ThreadEx } a\ \textit{False}]\rightarrow (\textit{unit}, \sigma)$

ISINTRDT: $t \vdash (\textit{addr } a.\textit{isInterrupted}([]), \sigma) -[\textit{Intrd } a]\rightarrow (\textit{true}, \sigma)$

ISINTRDF: $t \vdash (\textit{addr } a.\textit{isInterrupted}([]), \sigma) -[\textit{NotIntrd } a]\rightarrow (\textit{false}, \sigma)$

Fig. 9: Semantics of methods *start* and *isInterrupted* for class *Thread*. All rules have the preconditions *typeof-addr* $\sigma\ a = \lfloor T \rfloor$ and that T inherits the called method from class *Thread*.

(3) The semantics ensure that the first transition of every thread generates the *Start* event and – if the thread terminates – the last transition the *Finish* event. Layer 4 in the stack of semantics takes care of this.

Just like the single-threaded semantics does not know anything about the other threads, the multithreaded semantics is oblivious of the thread-local states and the shared type information. The multithreaded state stores the state of all locks, the wait sets and notifications, the pending interrupts and the threads' local states.

When the interleaving semantics interleaves the reductions of the individual threads (notation $s -(t, \overline{\alpha})\rightarrow s'$), it checks that the events $\overline{\alpha}$ of t's reduction are in line with the current multithreaded state s. For example, if $\overline{\alpha}$ contains a *Lock* a event, then the lock a must not currently be held by any thread other than t. And in case of a *Spawn* a _ event, the thread identified by object a must not have been spawned yet. Implementing all this is tedious – see [Lochbihler 2012b, Ch. 3] for the definitions –, but the full details are not relevant for the rest of this paper. I write $s -\overline{t\overline{\alpha}}\rightarrow^* s'$ for the reflexive and transitive closure of _ −_→ _, where $\overline{t\overline{\alpha}}$ is a list of pairs $(t, \overline{\alpha})$ of thread ID and event list $\overline{\alpha}$.

Unfortunately, the events from §1.1 are insufficient to correctly implement the JLS. Therefore, I introduce the following additional events:

(1) Detect whether a thread has already been spawned (*ThreadEx*),
(2) wait in a monitor (*Suspend*) and notification (*Notify*, *NotifyAll*, *WakeUp*),
(3) clearing an interrupt (*ClearIntr*) and testing for a thread not being interrupted (*NotIntrd*), and
(4) test whether the current thread does (not) hold a lock (*HasLock*, *NoLock*).

Technically, the last group is only a convenience, because this way, a thread need not remember in its local state which locks it is holding. The others, however, are necessary as the examples P2, P3, P4, P5, and Fig. 20 will show.

For *ThreadEx*, consider program P2 of two threads that race for spawning the same thread:

initially: volatile t = null;

`1: r1 = new Thread();`	`3: r2 = t;`	`5: r3 = t;`
`2: t = r1;`	`4: r2.start();`	`6: r3.start();`

(P2)

Suppose both reads in ll. 3 and 5 see the write at l. 2, i.e., they read the address of the allocated *Thread* object. Then, either l. 4 or l. 6 must throw an *IllegalThreadState* exception, but not both. Hence, both l. 4 and l. 6 must be allowed to fail in some executions. Thus, the two right-most threads may just start, read the address of the

Thread object (then fail with the exception, but the JMM has no event for that), and then finish. Hence, if each thread were run in isolation, they both would be allowed to fail, too. Since this contradicts the specification of the *start* method, there is a covert communication channel.[11]

For the new interruption events, consider the following program.[12] Can we have `r == 0` at the end?

initially: `volatile v = 0;`

`1: t2.interrupt();`
`2: while (t2.isInterrupted());`
`3: r = v;`
	`4: while (!t2.isInterrupted());`
	`5: v = 1;`
	`6: Thread.interrupted();`
		(P3)

Intra-thread consistency requires that l. 3 executes only after the loop in l. 2 has terminated. According to the API specification, *isInterrupted*() returns true as long as t_2 is interrupted. As only l. 6 can clear the interrupt, l. 3 must execute after ll. 5 and 6. As `v` is volatile, the read in l. 3 must see the most recent write, i.e., consistency allows only `r == 1`. Without the new events *ClearIntr* and *NotIntrd*, however, one could not express this dependency at all.

2.3. Traces

Given the interleaving semantics, the next step is to construct the execution candidates (called traces) from which the JMM rules will select the well-formed and legal ones. A trace ξ is a possibly infinite list of events labelled by the thread that generated them. The relation $s \Downarrow \xi$ characterises all traces ξ that start in the state s, which I define as

$$s \Downarrow \xi \longleftrightarrow (\exists \overline{\xi}.\ s \downarrow \overline{\xi} \wedge \xi = \textit{concat}\ \overline{\xi}) \tag{1}$$

where *concat* $\overline{\xi}$ concatenates all lists in $\overline{\xi}$ and $s \downarrow \overline{\xi}$ (defined coinductively) collects the list of lists of events labelled with the thread ID as follows:

$$\frac{s \not\rightarrow}{s \downarrow []}\ \textsc{Stop} \qquad\qquad \frac{s -(t, \overline{\alpha})\rightarrow s' \qquad s' \downarrow \overline{\xi}}{s \downarrow (\textit{events}\ t\ \overline{\alpha}) \cdot \overline{\xi}}\ \textsc{Step}$$

where $_ \not\rightarrow$ characterises stuck states in the interleaving semantics and *events* $t\ \overline{\alpha}$ retains only the original JMM inter-thread actions (as defined in §1.1) from $\overline{\alpha}$ and pairs each with the thread ID t. Hence, every trace is a complete list of the JMM events produced by running the program.

Note that the detour via a list of lists of events is necessary. If I defined $s \Downarrow \xi$ directly with the above coinductive rules STOP and STEP (i.e., prepending *events* $t\ \overline{\alpha}$ to ξ instead of consing), I could derive every trace ξ for a state s that can perform an infinite sequence of transitions without events, i.e., *events* $t\ \overline{\alpha} = []$, because it would be impossible to prove that ξ was not a trace (STEP would be applicable infinitely often as discussed in §1.2). The above approach works fine since $(\textit{events}\ t\ \overline{\alpha}) \cdot \overline{\xi}$ adds a constructor and concatenating the infinite list of empty lists yields the empty list [].

For the JMM, a program always comes with a fixed start state *start-state*. It is specified by a class, a method name, and the list of parameters it takes. It contains only a

[11]For *start*, the JMM specifies synchronisation only between a successful call and the first action of the spawned thread [Gosling et al. 2005, §17.4.4]. A JVM implementation might add more synchronisation, but our semantics must not, since this might eliminate data races from programs, i.e., it could wrongly certify programs with data races as DRF: for an example, see the program P8 and its data race between ll. 1 and 5 in §6.

[12]The method `isInterrupted` returns whether the receiver thread has been interrupted. The static method `interrupted` returns the interrupt status of the current thread and atomically clears the interrupt.

```
class T0 {
  public static void main(String[] args) {
    C c = new C(); Thread t1 = new T1(c); Thread t2 = new T2(c);
    t1.start(); t2.start(); } }

class T1 extends Thread {
  C c;
  T1(C c) { this.c = c; }
  public void run() { C c = this.c; int r1 = c.x; c.y = 1; } }

class T2 extends Thread {
  C c;
  T2(C c) { this.c = c; }
  public void run() { C c = this.c; int r2 = c.y; c.x = 1; } }

class C { int x, y; }
```

Fig. 10: Java implementation for the example in Fig. 3

single thread *start-tID* that holds no locks and is about to execute the specified method with the given parameters. All wait sets are empty and there are no pending interrupts. The shared type information has pre-allocated the *start-tID* `Thread` object and certain system exceptions. The list *start-events* of start-up events contains *start-tID*'s *Start* event and allocations for the preallocated objects.

Then, the JMM identifies a program with the set $\mathcal{E}$ of complete traces that start in *start-state*, prefixed with *start-events*. Formally ($++$ concatenates two lists):

$$\mathcal{E} = \{\, \textit{start-events} \mathbin{++} \xi \mid \textit{start-state} \Downarrow \xi \,\}$$

$\mathcal{E}$ contains many ill-formed traces, because read operations may read arbitrary values, even not type-conforming ones that no write operation of the program can ever produce. Since such traces have no write-seen function, the JMM well-formedness rules discard them.

Let me now present an example of $\mathcal{E}$ in detail. Fig. 10 shows a Java implementation of the example in Figs. 1 and 3. There is a bootstrapping thread t_0 that creates and spawns the two threads t_1 and t_2 whose *run* methods contain the code from the example. Since JinjaThreads does not model static fields, the shared locations `x` and `y` are represented by the fields of a container class `C`.

All traces start with the following events that represent t_0's `main` method up to the first call to *start*, abbreviated as *up-to-spawn*:

$$\begin{aligned}\textit{start-events} \mathbin{++} [&(t_0, \textit{Alloc}\ a_0\ \texttt{C}), (t_0, \textit{Alloc}\ t_1\ \texttt{T1}), (t_0, \textit{Write}\ (t_1, \texttt{T1::c})\ a_0),\\ &(t_0, \textit{Alloc}\ t_2\ \texttt{T2}), (t_0, \textit{Write}\ (t_2\ \texttt{T2::c})\ a_0), (t_0, \textit{Spawn}\ t_1\ _)]\end{aligned}$$

i.e., t_0 allocates the objects for the container and the two threads at locations a_0, t_1 and t_2[13] and executes `T1`'s and `T2`'s constructors. Remember that the allocations initialise the fields with default values, i.e., 0 for `x` and `y` declared in `C`, and *Null* for `c` declared in `T1` and `T2`.

Thread t_1 has three structurally different traces depending on the kind of value that reading `this.c` stores in the local variable `c`, namely (the label t_1 is omitted):

(1) $[\textit{Start}, \textit{Read}\ (a_1, \texttt{T1::c})\ (\textit{Addr}\ a), \textit{Read}\ (a, \texttt{C::x})\ v, \textit{Write}\ (a, \texttt{C::y})\ (\textit{Intg}\ 1), \textit{Finish}]$,

[13]For simplicity, I assume in examples that threads are identified by the address of their associated object. If not mentioned otherwise, all examples abstract from the non-determinism in the allocator by using location names a_0, t_1, and t_2 instead of concrete addresses.

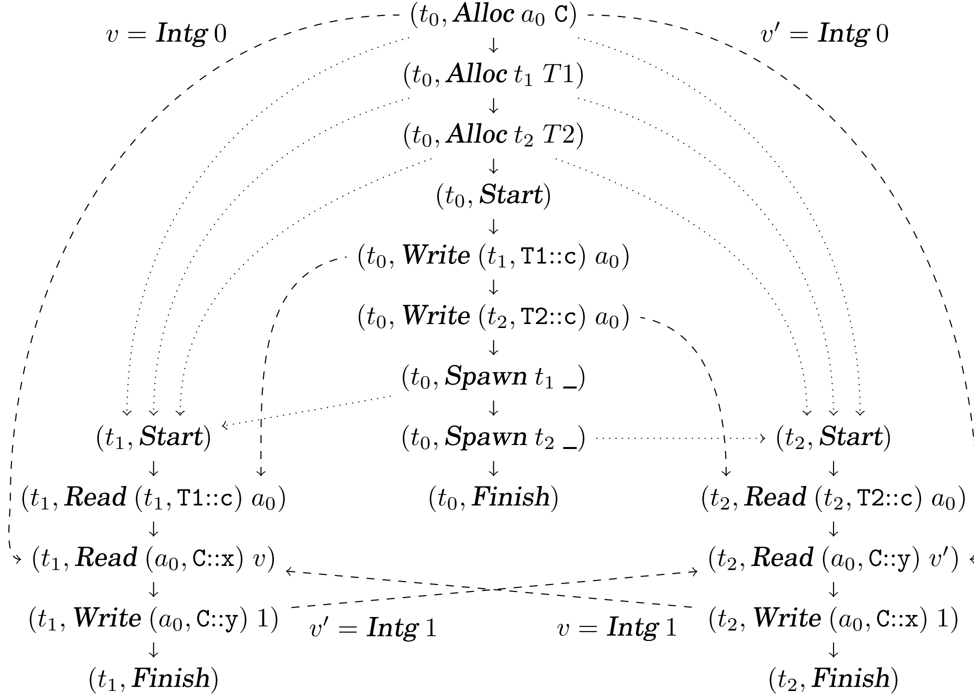


Fig. 11: Well-formed executions for the program in Fig. 10

(2) $[\textit{Start}, \textit{Read}\ (a_1, \texttt{T1::c})\ \textit{Null}, \textit{Finish}]$, and
(3) $[\textit{Start}, \textit{Read}\ (a_1, \texttt{T1::c})\ w]$,

where a is an arbitrary address, v is an arbitrary value, and w is any value other than an address or the null pointer *Null*. In the first form, the address a from the *Read* is then used to access the fields x and y of the referenced container. In the second, t_1 reads *Null*, so the subsequent field access raises the preallocated *NullPointer* exception and the thread immediately terminates. In the last case, w is type-incorrect, so the semantics gets stuck upon the next field access, i.e., there is no *Finish*. Thread t_2 has the same traces with x and y exchanged.

The traces in $\mathcal{E}$ for this program all start with *up-to-spawn* and then interleave the threads. Of these, the JMM considers only those as well-formed that originate from the first form with $a = a_0$ and $v, v' \in \{\ \textit{Intg}\ 0, \textit{Intg}\ 1\ \}$ (*Intg* injects 32 bit integers into the type of values). In particular, the unexpected behaviour from Fig. 3 is well-formed. In terms of the JMM, all these interleavings collapse to four well-formed executions as shown in Fig. 11 (where I have omitted the bootstrap events for clarity – except for t_0's *Start* event that is relevant for determining what happens before what). The write-seen arrows are labelled with conditions for which they apply. All well-formed executions are legal in this example.

2.4. Formal Definition of the Java Memory Model

Now, I formalise the JMM and connect it with the set of traces $\mathcal{E}$. First, I introduce some JMM terminology (§2.4.1) and I derive the orders of the JMM from a trace (§2.4.2), which abstract from the concrete interleaving as explained in §2.3. Then, the

formal definitions of well-formed and legal execution builds on these orders (§2.4.3 and §2.4.4).

2.4.1. JMM Terminology. Most JMM definitions depend on a trace ξ, which I usually attach as a sub- or superscript. To simplify the notation, I drop the sub- and superscript, when ξ is obvious from the context.

Since an event can occur multiple times in ξ, I use the index in ξ (a natural number) to assign a unique identifier to an event. Formally, an event identifier α is just an index whose interpretation $\xi_{[\alpha]}$ depends on the execution ξ. In the following, I usually blur this distinction: I use the variable α for both indices and events, and – when it is clear from the context – I write α instead of the interpretation $\xi_{[\alpha]}$. Hence, $\mathcal{A}_\xi = \{\, \alpha \mid \alpha < |\xi| \,\}$ denotes the set of events (more precisely, event identifiers) for ξ, where $|\xi|$ denotes the length of ξ.

A read event is an event of the form $(t, \mathit{Read}\ (a,l)\ v)$, it reads from location (a,l) the value v; $\mathcal{R}_\xi$ denotes the set of read events of ξ. A write event is either a write $(t, \mathit{Write}\ (a,l)\ v)$ or an allocation $(t, \mathit{Alloc}\ a\ T)$, $\mathcal{W}_\xi$ denotes the set of write events in ξ. A write event $\alpha \in \mathcal{W}_\xi$ writes to location (a,l) (is a write to (a,l)) iff $\alpha = (_, \mathit{Write}\ (a,l)\ _)$, or $\alpha = (_, \mathit{Alloc}\ a\ T)$ and l is a member of T (notation $l \in \mathit{memb}\ T$). The members of a class C are all fields of C and its super-classes, and the members of an array type $T[n]$ are the fields of *Object* and the array cells $[0], \dots, [n-1]$. An allocation $\mathit{Alloc}\ a\ T$ initialises all locations (a,l) for $l \in \mathit{memb}\ T$.

I say that α accesses location (a,l) iff α is a read or write event that reads from or writes to (a,l), respectively. $\mathit{locs}\ \alpha$ denotes the set of locations that $\alpha \in \mathcal{R} \cup \mathcal{W}$ accesses; $\mathit{locs}\ \alpha = \emptyset$ for $\alpha \in \mathcal{A} - (\mathcal{R} \cup \mathcal{W})$.

For $\alpha \in \mathcal{W}$, $\mathit{vw}\ \alpha\ (a,l)$ denotes the value that α writes to location (a,l) – allocation events $(_, \mathit{Alloc}\ _\ T)$ write default values (0, *False*, and *Null*, respectively) for all members of T; normal writes $\mathit{Write}\ (a,l)\ v$ store the value v written themselves; $\mathit{vw}\ \alpha\ (a,l)$ is unspecified if α does not write to (a,l).

A member l is volatile (written $\mathit{is\text{-}volatile}\ l$) iff l is a field $C{::}F$ and class C declares F as volatile. A read or write α is volatile iff α reads from or writes to a volatile member of a location. In particular, array cells are never volatile by definition [Gosling et al. 2005, §8.3.1.4].

2.4.2. From Traces to Orders. A trace ξ already provides the *induced total order* $\preceq^\xi = \leq|_{\mathcal{A}_\xi}$ over $\mathcal{A}_\xi$, where $R|_A$ restricts the binary relation R to elements from A and $\leq$ is the standard order on natural numbers.

Since the JMM requires initialisation events (i.e. *Alloc*) to be ordered before the threads' initial events, I introduce the (total) *execution order* $\leq^\xi_{\mathrm{eo}}$ on $\mathcal{A}_\xi$:

$$\alpha \leq^\xi_{\mathrm{eo}} \alpha' \longleftrightarrow (\mathbf{if}\ \mathit{init}_\xi\ \alpha\ \mathbf{then}\ \neg \mathit{init}_\xi\ \alpha' \vee \alpha \preceq^\xi \alpha'\ \mathbf{else}\ \neg \mathit{init}_\xi\ \alpha' \wedge \alpha \preceq^\xi \alpha') \tag{2}$$

where $\mathit{init}_\xi\ \alpha$ predicates that α is an allocation event in ξ, i.e., $\xi_{[\alpha]} = (_, \mathit{Alloc}\ _\ _)$.

The program order $\leq^\xi_{\mathrm{po}}$ restricts $\leq^\xi_{\mathrm{eo}}$ to events of the same thread. The synchronisation order $\leq^\xi_{\mathrm{so}}$ restricts $\leq^\xi_{\mathrm{eo}}$ to synchronisation events. Synchronisation events are allocations (*Alloc*), reads from and writes to volatile locations, locking (*Lock*) and unlocking (*Unlock*), thread spawns (*Spawn*) and joins (*Join*), thread start (*Start*) and finish events (*Finish*), and the interruption events *Intr* and *Intrd*. The synchronises-with order $\leq^\xi_{\mathrm{sw}}$ restricts $\leq^\xi_{\mathrm{so}}$ to release-acquire pairs of events. (α, α') is a release-acquire pair (notation $\alpha \leadsto \alpha'$, definition in Fig. 12) iff

— α unlocks a monitor and α' locks the same monitor,
— α spawns a thread whose start action is α',
— α is the finish event of the thread on which α' joins,

$$
\begin{array}{rlr}
(t, \textit{Unlock}\ a) \dashrightarrow (t', \textit{Lock}\ a) & (t, \textit{Alloc}\ a\ T) \dashrightarrow (t', \textit{Start}) & \\
(t, \textit{Spawn}\ t'\ _) \dashrightarrow (t', \textit{Start}) & (t, \textit{Write}\ (a,l)\ v) \dashrightarrow (t', \textit{Read}\ (a,l)\ v') & \\
(t, \textit{Finish}) \dashrightarrow (t', \textit{Join}\ t) & (t, \textit{Alloc}\ a\ T) \dashrightarrow (t', \textit{Read}\ (a,l')\ v) & \text{if } l' \in \textit{memb}\ T \\
(t, \textit{Intr}\ t'') \dashrightarrow (t', \textit{Intrd}\ t'') & &
\end{array}
$$

Fig. 12: Release-acquire pairs

— α interrupts a thread t and α' observes that t has been interrupted,
— α is an allocation event and α' is a thread start event,[14] or
— α writes to a location that α' reads.[15]

The happens-before order $\leq^{\xi}_{\text{hb}}$ is the transitive closure of $\leq^{\xi}_{\text{po}}$ and $\leq^{\xi}_{\text{sw}}$. This concludes the construction of orders from traces.

2.4.3. Well-formed Executions. An execution (ξ, ws) consists of a trace ξ and a write-seen function ws that assigns to every read event in $\mathcal{R}_\xi$ the write event it sees. This yields the JMM notion of an execution [Gosling et al. 2005, §17.4.6] as $(\mathcal{E}, \mathcal{A}, \leq_{\text{po}}, \leq_{\text{so}}, ws, \textit{vw}, \leq_{\text{sw}}, \leq_{\text{hb}})$.

An execution is well-formed (written $(\xi, ws)\surd$) iff every thread has a thread start event that $\preceq^{\xi}$-precedes its other events except for allocation events (denoted $\xi\surd_{\text{Start}}$) and for all read events $\alpha \in \mathcal{R}$ to some location (a, l),

(W1) $ws\ \alpha$ writes to (a, l), i.e., $ws\ \alpha \in \mathcal{W}$ and $(a, l) \in \textit{locs}\ (ws\ \alpha)$,
(W2) α reads the value $\textit{vw}\ (ws\ \alpha)\ (a, l)$,
(W3) $\alpha \not\leq_{\text{hb}} ws\ \alpha$,
(W4) for all write events β to (a, l), if $ws\ \alpha \leq_{\text{hb}} \beta \leq_{\text{hb}} \alpha$, then $\beta = ws\ \alpha$, and
(W5) if α is a volatile read, then $\alpha \not\leq_{\text{so}} ws\ \alpha$ and for all write events β to (a, l), if $ws\ \alpha \leq_{\text{so}} \beta \leq_{\text{so}} \alpha$, then $\beta = ws\ \alpha$.

ξ is well-formed iff $(\xi, ws)\surd$ for some ws.

These conditions correspond to the JMM well-formedness conditions 1 (each read sees a write to the same location), 4 ($\leq_{\text{hb}}$ consistency) and 5 ($\leq_{\text{so}}$ consistency for volatiles) in [Gosling et al. 2005, §17.4.7]. (ξ, ws) meets conditions 2 ($\leq_{\text{hb}}$ is a partial order) and 3 (intra-thread consistency) by construction. Moreover, layer 4 of the single-threaded semantics ensures that all traces $\xi \in \mathcal{E}$ satisfy $\xi\surd_{\text{Start}}$.

Note that conditions W3 and W4 do not imply condition W5, because $\leq_{\text{hb}}$ contains only $\leq_{\text{sw}}$, but not $\leq_{\text{so}}$. For example, the execution in Fig. 4 satisfies all well-formedness conditions except W5. Remember that it is $\leq_{\text{so}}$ that forces all threads to agree on a global order of synchronisation events.

Reconsider the program in Fig. 5. Since there is a trace for every possible value (cf. §2.2), $\mathcal{E}$ contains a trace

$$\xi = [\ldots, (t_1, \textit{Read}\ \texttt{x}\ 1), (t_2, \textit{Read}\ \texttt{y}\ 1), (t_1, \textit{Write}\ \texttt{y}\ 1), (t_2, \textit{Write}\ \texttt{x}\ 1), \ldots]$$

The omitted events include those necessary to allocate and to spawn the threads – similar to the example in §2.3, but they do not constrain the write-seen function on the events mentioned. In particular, each read may see the respective write, because they may happen concurrently. Therefore, the execution in Fig. 5b is well-formed.

[14]There has been some unclarity whether allocations of objects without volatile fields should synchronise with thread start events [Aspinall and Ševčík 2007a, 2007b]. I assume that all allocations synchronise with *Start* events, because the DRF guarantee (§3) and consistency (§4) rely on this.
[15]I do not need to restrict writes and reads to volatiles explicitly like the JMM does [Gosling et al. 2005, §17.4.4], because the synchronisation order already imposes this.

2.4.4. Legal executions. Still, the JMM disallows the execution by imposing additional legality constraints. As the following presentation is very technical, readers unfamiliar with the JMM may want to first look at [Aspinall and Ševčík 2007a; Huisman and Petri 2007] for a warm-up.

A *legal* execution is a well-formed execution (ξ, ws) that is justified by a sequence of justifying executions $(\xi_i, ws_i, C_i, \varphi_i)_i$, where C_i are the sets of committed events and the event renaming functions φ_i inject the committed events of ξ_i into ξ's events. The legality constraints ensure that a read α in ξ sees a possibly concurrent write $ws\ \alpha$ only if the read and writes are committed in a justifying execution i such that the previous justifying execution $i-1$ contains a write to the same location with the value that α reads such that α may see that write and the write happens-before α in $i-1$. This makes the execution in Fig. 5b illegal, because any well-formed execution one of whose reads does not see the $\leq_{\text{hb}}$-unrelated write (i.e., it sees the initialisation with 0), all values written to x or y are 0. Hence, no execution can produce the write of 1 necessary for justification.

Conversely, the execution in Fig. 3b is legal by the following argument: Start with an execution in which each read sees the respective initialisation and commit the writes, which write the value 1. Complete the justification by committing the reads and have them see the committed, but $\leq_{\text{hb}}$-unreleated write.

The formal definition of justification uses the following notation: f ‘ A denotes the image of the set A under the function f; *inj-on* $f\ A$ expresses that f is injective on A; for a binary relation R, $R|_C$ restricts R to elements from C and $\alpha\ \varphi_i^{-1}(R)\ \alpha'$ iff $(\varphi_i\ \alpha)\ R\ (\varphi_i\ \alpha')$; $\varphi_i^{-1}(\alpha)$ denotes some α' such that $\alpha' \in C_i$ and $\varphi_i\ \alpha' = \alpha$; $(t, e) \simeq (t', e')$ iff $t = t'$ and the events e and e' are identical except for the values they write or read.

A sequence $J = (\xi_i, ws_i, C_i, \varphi_i)_i$ ***justifies*** (ξ, ws) (notation (ξ, ws) ***justified-by*** J) iff all of the following hold for all i:

(L1) well-formedness: $(\xi_i, ws_i)\surd$ and $\xi_i \in \mathcal{E}$
(L2) commit sequence: $C_0 = \emptyset$, and φ_i ‘ $C_i \subseteq \varphi_{i+1}$ ‘ C_{i+1} for all i, and $\mathcal{A}_\xi = \bigcup_j \varphi_j$ ‘ C_j
(L3) commit only actions: $C_i \subseteq \mathcal{A}_{\xi_i}$
(L4) happens-before order: $\leq_{\text{hb}}^{\xi_i}\Big|_{C_i} = \varphi_i^{-1}(\leq_{\text{hb}}^{\xi})\Big|_{C_i}$
(L5) synchronisation order: $\leq_{\text{so}}^{\xi_i}\big|_{C_i} = \varphi_i^{-1}(\leq_{\text{so}}^{\xi})\big|_{C_i}$
(L6) value written:
$\textbf{\textit{vw}}\ \xi_{i[\alpha]}\ (a, l) = \textbf{\textit{vw}}\ \xi_{[\varphi_i\ \alpha]}\ (a, l)$ for all $\alpha \in \mathcal{W}_{\xi_i} \cap C_i$ and $(a, l) \in \textbf{\textit{locs}}\ \xi_{[\varphi_i\ \alpha]}$
(L7) write-seen: $\varphi_{i+1}\ (ws_{i+1}\ (\varphi_{i+1}^{-1}\ (\varphi_i\ \alpha))) = ws\ (\varphi_i\ \alpha)$ for all $\alpha \in \mathcal{R}_{\xi_i} \cap C_i$
(L8) uncommitted reads: if $\varphi_{i+1}\ \alpha \notin \varphi_i$ ‘ C_i, then $ws_{i+1}\ \alpha \leq_{\text{hb}}^{\xi_{i+1}} \alpha$ for all $\alpha \in \mathcal{R}_{\xi_{i+1}}$
(L9) newly committed reads: for all $\alpha \in \mathcal{R}_{\xi_{i+1}} \cap C_{i+1}$, if $\varphi_{i+1}\ \alpha \notin \varphi_i$ ‘ C_i, then $\varphi_{i+1}\ (ws_{i+1}\ \alpha) \in \varphi_i$ ‘ C_i and $ws\ (\varphi_{i+1}\ \alpha) \in \varphi_i$ ‘ C_i
(L10) external events: for all external actions $\alpha \in \mathcal{A}_{\xi_i}$ and all $\alpha' \in C_i$, if $\alpha \leq_{\text{hb}}^{\xi_i} \alpha'$, then $\alpha \in C_i$
(L11) event renaming: *inj-on* $\varphi_i\ \mathcal{A}_{\xi_i}$ and for all $\alpha \in C_i$, $\xi_{i[\alpha]} \simeq \xi_{[\varphi_i\ \alpha]}$

Constraints L1 and L2 ensure that all executions are well-formed, committed events remain committed in subsequent justifying executions, and eventually all events are committed. L3 to L10 formalise the JMM legality conditions 1 to 7 and 9 [Gosling et al. 2005, §17.4.8] augmented with explicit renaming of events. I omit condition 8 for two reasons: First, it relies on the transitive reduction of $\leq_{\text{hb}}$, which need not exists for infinite executions [Aspinall and Ševčík 2007b]. Second, Torlak et al. [2010] have shown that it is irrelevant for all JMM test cases [Pugh and Manson 2004].

The novel constraint L11 deals with renaming of events. As the above discussion of Figs. 5b and 3b has shown, legality requires to identify events across executions. The renaming function φ_i injectively maps events from $\mathcal{A}_{\xi_i}$, i.e., natural numbers, to natural numbers. L2 ensures that the image $\varphi_i\ \alpha$ of a committed event α is again an event of $\mathcal{A}_\xi$, and L11 demands that the original event $\xi_{i[\alpha]}$ and the image $\xi_{[\varphi_i\ \alpha]}$ be identical except for values they read or write. Renamings identify two events α and β as follows:

— If α originates from the execution to be justified and β from i-th justifying execution, they are identical iff $\varphi_i\ \beta = \alpha$.
— If α and β come from the i-th and j-th justifying execution, respectively, they are identical iff $\varphi_i\ \alpha = \varphi_j\ \beta$.

Note that I require φ_i to be injective on all events of the justifying execution, not only on the committed ones. This way, L8 can express that α has not been committed in a previous justifying execution as $\varphi_{i+1}\ \alpha \notin \varphi_i\ `\ C_i$. I will discuss the issue of event identification in more detail in §2.5.

Cenciarelli et al. [2007] noted that the legality constraints disallow some desirable compiler optimisations. To fix this, Aspinall and Ševčík [2007a] suggested to weaken the constraints as follows:[16] They drop L5 and replace L4 and L9 with

(L4') for all $\alpha \in \mathcal{R}_{\xi_i} \cap C_i$, it is the case that $ws\ (\varphi_i\ \alpha) \leq^{\xi}_{\mathrm{hb}} \varphi_i\ \alpha$ iff $\varphi_i^{-1}(ws\ (\varphi_i\ \alpha)) \leq^{\xi_i}_{\mathrm{hb}} \alpha$, and $\alpha \not\leq^{\xi_i}_{\mathrm{hb}} \varphi_i^{-1}(ws\ (\varphi_i\ \alpha))$
(L9') for all $\alpha \in \mathcal{R}_{\xi_{i+1}} \cap C_{i+1}$, if $\varphi_{i+1}\ \alpha \notin \varphi_i\ `\ C_i$, then $ws\ (\varphi_{i+1}\ \alpha) \in \varphi_i\ `\ C_i$

This yields *weak justification sequences* and *weak legality*. Note that every justification sequence also satisfies L4' and L9', i.e., legality implies weak legality. Aspinall and Ševčík [2007a] showed that weak legality suffices for the DRF guarantee. In §5.2, I will show that weak legality also suffices for type safety.

2.5. Identity of Events Across Executions

The JLS assumes that every event carries a unique identifier, but justification requires to identify events across executions. Previous JMM formalisations [Aspinall and Ševčík 2007a; Cenciarelli et al. 2007; Huisman and Petri 2007] assume that comparable events across executions are identical and thus avoid the problem of identification. However, for a mechanically-checked link between operational semantics and the JMM, I must explicitly construct the events from the program and therefore define identity. As the JLS does not tell how to assign identifiers, I judge whether an identification scheme captures the JMM intent by studying its implications on the allowed behaviours.

In the following, I discuss previous attempts and compare them to my approach with renaming functions. They fall in two groups: In the first group, identity depends only on the event and the execution in which the event occurs. I show that each in this group forbids some desirable behaviour (§2.5.1). The second group, to which my renaming functions belong, assigns identity based on the whole justification sequence. I discuss the additional expressiveness that they offer (§2.5.2).

[16] In [Aspinall and Ševčík 2007a], they also omit L10 as they only consider the constraints relevant for the DRF guarantee, but in [Aspinall and Ševčík 2007b], they keep L10. As noted by Torlak et al. [2010], they change L10 in [Ševčík and Aspinall 2008; Ševčík 2008]. I keep the original formulation, because their change seems inadvertent: It renders L10 vacuous, and they do not mention the change. Rather, their informal description fits the original L10.

initially: `x = y = 0;`

`1: r1 = x;`	`3: r2 = y;`
`2: if (r1 == 1) y = 1;`	`4: if (r2 == 1) x = 1;`
	`5: synchronized (new Object()) {}`
	`6: if (r2 == 0) x = 1;`

Fig. 13: JMM test case 6 [Pugh and Manson 2004] with additional synchronisation (l. 5): the result `r1 == r2 == 1` is allowed.

2.5.1. Identity Relative to an Execution. Polyakov and Schuster [2006] first noted the problem of event identity. Their approach relies on *lexical scoping*; they identify two events if the same program statement produces them in the same iteration of any surrounding loop. However, they note that their formalisation fails JMM test case 6 (Fig. 13 without l. 5), in which the writes in l. 4 and l. 6 must be considered identical.[17] They suggest that writes without synchronisation in between could be merged once they have been determined to produce the same value. Still, this fails for the example in Fig. 13, which adds redundant synchronisation to JMM test case 6. A compiler can easily determine that the monitor cannot escape t_2 and that the synchronisation is therefore redundant [Choi et al. 1999; Ruf 2000]. As the JMM always allows to remove such synchronisation [Manson et al. 2005], Fig. 13 is equivalent to JMM test case 6. Therefore, the JMM allows the result `r1 == r2 == 1`, too.

Besides identification by lexical scope, Jin et al. [2012] suggest two further approaches. *Occurrence* identifies two events of the same thread in two executions if they access the same location and the thread has previously accessed the location the same number of times. *Occurrence-val* additionally distinguishes events by the value that is read or written.

For Fig. 14, e.g., the JMM should allow `r1 == r2 == r3 == 1`, because a compiler can move ll. 6 and 7 before l. 4; then, the schedule 3, 6, 7, 1, 2, 4, 5 yields the result even under SC. However, neither occurrence nor occurrence-val allows it. Note that one must commit the data race on `z` first, i.e., ll. 3 and 6; otherwise, no write to `x` executes and we are stuck with all being 0. Then, l. 7 executes `x = 1`, and we can commit the race with l. 1. Since causality constraint L4 fixes the happens-before relationships between committed events, the read of `z` must happen before t_3's *first* write `x = 1`. But if l. 4 reads 1 from l. 2, the write from l. 5 seizes l. 7's identity, which illegally reverses the happens-before relationship with the read of `z`.

Note that both occurrence and occurrence-val do allow the result under weak legality. However, it fails for the more intricate example in Fig. 15 to allow `r1 == r2 == r3 == r4 == 1`. Let me first show how to justify the result under legality – in all executions, I assume that l. 3 reads from l. 8, i.e., ll. 6 to 8 happen before ll. 3 to 5 because `v` is volatile. First, commit the race on `z` such that l. 4 sees l. 9. This makes l. 5 write 1, so commit the race with l. 1, such that l. 2 writes 1, too. Finally, commit the race on `y` such that l. 6 reads 1 from l. 2 instead of the initial value 0. The last step inserts the write

[17] To justify `r1 == r2 == 1`, start with the well-behaved execution in which all reads see the initialisations, i.e., l. 6 writes 1 to `x`. Then, commit the race on `x` between l. 1 and l. 6 such that `1` sees l. 7, i.e., l. 2 now writes 1 to `7`. In the next step, commit the race on ll. 2 and 3 such that l. 3 sees l. 2. This completes the justification. Note that it is now l. 4 instead of l. 6 that writes 1 to `x`. As l. 6 has already been committed, their events must be identified, but the lexical approach distinguishes the two writes are distinct. Hence, their model does not allow this justification; and neither the result `r1 == r2 == 1`, because the above justification sequence is the only one: To obtain `r2 == 1`, l. 3 must see the concurrent write in l. 2, i.e., one first must have `r1 == 1` in a justifying execution. Hence, one must first commit the data race on `x` such that l. 1 can see the write of 1 to `x`. As l. 3 can only see `y`'s initialisation, only the write to `x` in l. 6 executes. However, once l. 3 sees l. 2, it is l. 4 that generates the write to `x`.

initially: `x = y = z = 0;`

1: r1 = x;	3: z = 1;	4: r2 = y;
2: y = r1;		5: if (r2 == 1) x = 1;
		6: r3 = z;
		7: if (r3 == 1) x = 1;

Fig. 14: Occurrence and occurrence-val forbid the result `r1 == r2 == r3 == 1`

initially: `x = y = z = 0; volatile v = 0;`

1: r1 = x;	3: r2 = v;	6: r4 = y;
2: y = r1;	4: r3 = z;	7: if (r4 == 1) z = 1;
	5: x = r3;	8: v = 1;
		9: z = 1;

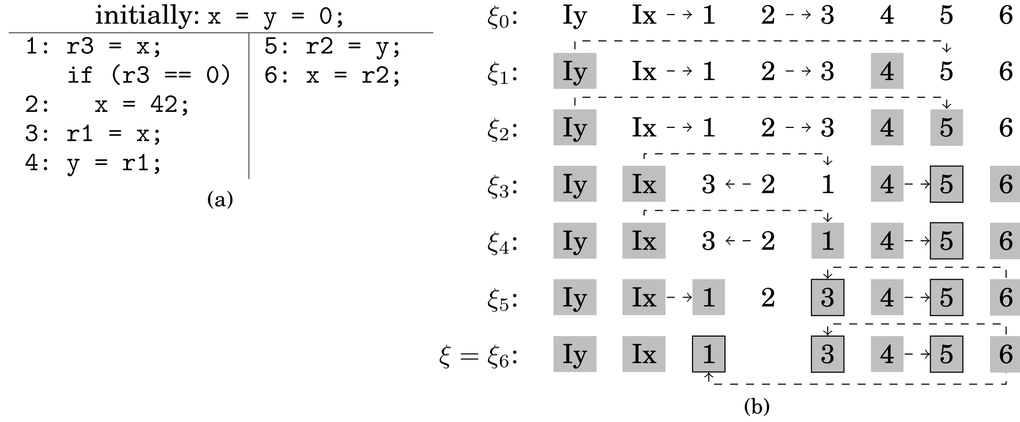
Fig. 15: Occurrence and occurrence-val forbid the result `r1 == r2 == r3 == r4 == 1` even under weak legality

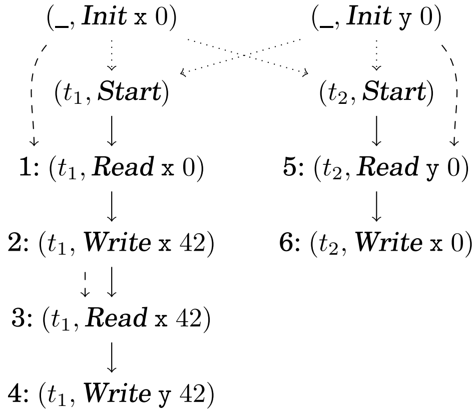
to z from l. 7. Under occurrence-val, this write seizes the identity of the former write from l. 9. However, l. 7 happens before l. 4 whereas l. 9 may happen concurrently with l. 4, i.e., L4' is violated. As l. 7 seizes the identity of l. 9 in every justification sequence for the execution, both occurrence and occurrence-val forbid the result. Note that the above sequence does justify the result, if the writes in ll. 7 and 9 are not identified.

2.5.2. Identity Relative to a Justification Sequence. The identification scheme by Torlak et al. [2010] belongs to the second group. For small Java programs with finite state (loops are unrolled), they use whole-program analysis to compute in advance identifiers for all events. Although they do not provide a declarative description, their model checking algorithm assigns event identities on a per-justification basis. This achieves the same effect as my renaming functions and constraint L11: Their precomputation of sharing enforces $\xi_{i\,[\alpha]} \simeq \xi_{[\varphi_i\ \alpha]}$ and injectivity is added as a separate constraint for the model checker. Renaming functions have the advantage that I do not need to construct a universe of event identifiers. This simplifies my formalisation considerably.

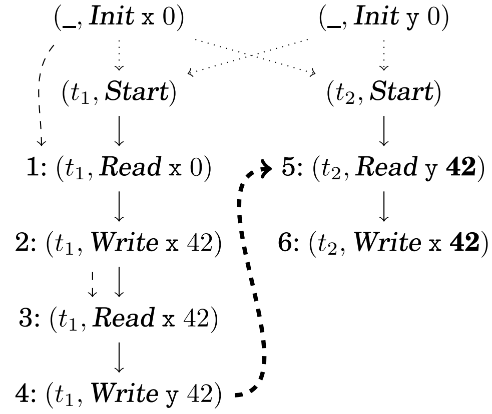
To appreciate the power of event renaming, consider the JMM causality test case 18 in Fig. 16a [Pugh and Manson 2004]. The original justification sequence by Pugh and Manson [2004] for the result `r1 == r2 == r3 == 42` is incorrect, and [Ševčík and Aspinall 2008] claimed that the JMM did not allow it at all. MemSAT by Torlak et al. [2010] found a justification sequence, though (Fig. 16b) [personal communication, Oct. 2012]. It relies on different statements performing the same event in different executions.

The justification table in Fig. 16b uses the following notation: Each row lists the events of an execution, they are represented by the line number of the statement that produces them; Ix and Iy represent the initialisations of x and y, respectively. Uninteresting events like *Start* and *Finish* have been omitted. As before, dashed arrows denote the flow of values, i.e., the read at the tip sees the write at the origin. To visualise the renaming functions φ_i, I have not depicted the events in execution order. Rather, they are in the column of their image under φ_i. For example, for execution ξ_4, φ_4 maps the read event of l. 1 to the read event of l. 3 in the final execution ξ. Shaded events are committed, and a box around a read event indicates that the write it sees is fixed. Remember that the execution *after* the one that commits a read α is the last (in the justification sequence) to change the write that α sees (L7). For clarity, the justifying executions are shown in Figs. 16c to 16f; differences to the previous picture are set in bold.

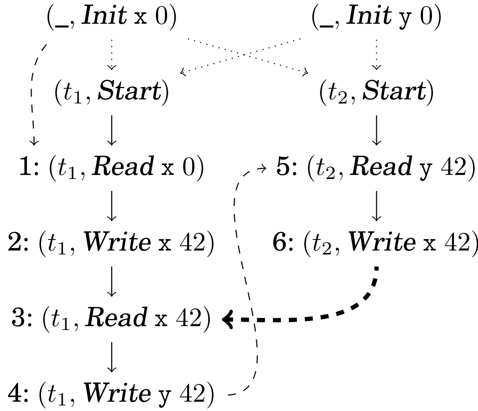




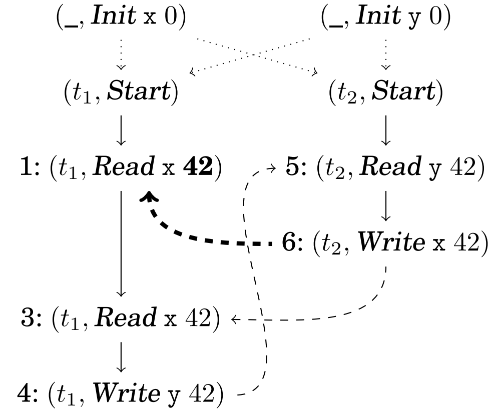
(c) Justifying executions ξ_0, ξ_1, and ξ_2



(d) Justifiying executions ξ_3 and ξ_4



(e) Justifying execution ξ_5



(f) Justifying execution ξ_6

Fig. 16: Causality test case 18 (a), justification for `r1 == r2 == r3 == 42` (b), and justifying executions (c) to (f).

initially: `x = null; b = false;`

```
1: r1 = null;                 | 6: if (x != null)
2: if (b)                     | 7:   b = true;
3:   r1 = new Object();       |
4: r2 = new Object();         |
5: x = r2;                    |
```

Fig. 17: Reordering of allocations allows `r1 != null`.

The executions ξ_0 to ξ_3 commit the data race on y between ll. 4 and 5. Most interesting is the transition from ξ_4 to ξ_5. ξ_4 commits the read from l. 1. Note, however, that φ_4 identifies this read with the read from l. 3 in ξ. In contrast, φ_5 maps the read from l. 3 to the read from l. 3, i.e., l. 3 generates the committed read from l. 1. Although ξ_4 has already committed the read from l. 1, ξ_5 commits it a second time, because event renaming has decided that they are different events in the final execution. This change between φ_4 and φ_5 is allowed, because we have not yet committed any events that happen between between ll. 1 and 3. Otherwise, L4 would be violated. The rest of the justification is standard, ξ_6 merely changes l. 1 to see l. 6 like in the final execution.

In fact, justification of `r1 == r2 == r3 == 42` requires event renaming: Any justification sequence must first commit the race on y to get 42 into r2 and r3. Hence, ξ_0 to ξ_3 are canonical in any such justification. Now, suppose we commit l. 1 in ξ_4 and set its write from ξ to l. 6 in ξ_5, but $\varphi_4\ 1 = \varphi_5\ 1$. Then, l. 2 would no longer execute in ξ_5, i.e., the uncommitted read in l. 5 would have to see the initialisation and the committed l. 6 could not produce the value 42 any more as required by L6. Alternatively, we could try to commit l. 3 before l. 1 such that l. 3 reads from l. 6. However, L9 requires that we commit l. 2 before changing the write-seen to an $\leq_{\mathrm{hb}}$-unrelated write, but we must not commit l. 2 because it has no match in the final execution. With weak legality, the latter also yields a justification, because L9' does not require l. 2 to be committed.

2.6. Non-deterministic allocators

Identity of events across execution also influences the identity of values and addresses. Legality condition L6 requires that committed writes write identical values – and a value may be an address. In my model, two addresses are identical iff they have the same concrete representation. Then, however, the allocator must be non-deterministic; otherwise, the compiler may not reorder allocations. Figure 17 shows a counterexample for any deterministic allocator that picks the next fresh address based only on its internal state σ.[18] To get the result `r1 != null`, l. 3 has to execute, i.e., l. 2 reads `true` from b and therefore sees the write in l. 7. Thus, the read in l. 6 has to see the write from l. 5 – otherwise, l. 7 does not execute at all. This result should be allowed, because a compiler can move ll. 4 and 5 before l. 2.

This execution ξ has two data races that we must justify, namely on x (ll. 5 and 6) and b (ll. 2 and 7). The justification sequence starts with the well-behaved execution ξ_0 of ll. 1, 2, 4, 5, 6. First (execution ξ_1), we commit the write to x in l. 5, say of the address a_0. By L6, the final execution ξ has to write the identical address a_0 in l. 5. Note that in execution ξ_1, l. 2 sees b's initialisation, so l. 4 is the only allocation. From this, we deduce that the allocator always returns a_0 as the first fresh address for an object of class `Object`. In ξ, however, the first such allocation occurs in l. 3, so `r1 ==` a_0 in ξ. As the allocator always returns a fresh address (see assumption A4 in §3.2 below), it returns a different address a_1 for l. 4 in ξ, which contradicts that l. 5 writes

[18] Allocators whose strategy depends on external data such as lexical scoping cause problems like scope-based event identities in §2.5.1 do.

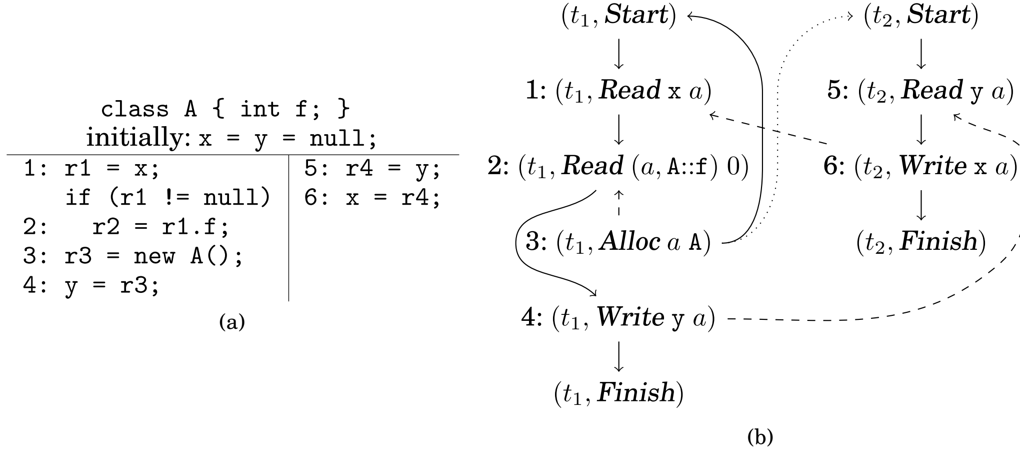


Fig. 18: Program with an execution in which the read in l. 2 sees the initialisation from l. 3, which occurs later in the program text

a_0. Thus, a deterministic allocator does not allow the result `r1 != null`, i.e., it does not allow reordering of allocations in general. In contrast, the non-deterministic allocators from §2.1 allow `r1 != null`, because they are free to pick another address a_1 for the allocation in l. 3.

Alternatively, I could have introduced address renaming functions in L6, similar to the event renaming functions. Then, allocators could be deterministic, but it is not clear what properties the address renaming functions should have. Note that non-deterministic allocators in the model do not require non-deterministic allocators in a real implementation. Just like a VM does not have to produce all behaviours that the JMM allows, the allocation strategy of the implementation merely must be predicted by the model.

2.7. Initialisations

Apart from identifying events across executions, justification also complicates memory initialisation. Previous machine-checked formalisations have omitted the initialisation business [Aspinall and Ševčík 2007a; Huisman and Petri 2007]. Ševčík [2008] assumes that a special thread initialises all necessary locations and terminates before the normal threads start. Unfortunately, this does not work for infinite executions that allocate infinitely many objects, because the initialising thread does not terminate. Moreover, a special initialisation thread conflicts with the final field semantics extension to the JMM, which requires to know which thread created which object [Gosling et al. 2005, §17.5.1].

Therefore, in my model, it is dynamic memory allocation which produces the initialisation writes (*Alloc*). Hence, they occur in the trace of a thread when it allocates the object. Figure 18 shows an example, a variation of Fig. 7; thread t_1 generates the events as shown from top to bottom. For the JMM, the execution order orders allocations before all other events (Eq. 2). Hence, in any well-formed execution ξ, allocations happen before all other events, because allocations synchronise with *Start*s and every thread produces its *Start* event $\preceq$-before all of its other events. In Fig. 18b, e.g., the allocation in l. 4 precedes all others in program order and happens-before order.

```
      class A { int f; }    class B { int g; }
           initially: x = y = z = w = null;
    while (true) {                 |     while (true) {
1:    r1 = x;                      |  9:   r7 = y;
      if (r1 != null) {            | 10:   x = r7;
2:      r2 = r1.f;                 | 11:   r8 = w;
3:      r3 = new B();              | 12:   z = r8; }
4:      w = r3; }                  |
5:    r4 = z;                      |
6:    if (r4 != null) r5 = r3.g;   |
7:    r6 = new A();                |
8:    y = r6; }                    |
```

Fig. 19: A program that has an infinite execution whose finite prefixes are eventually all ill-formed.

This ensures that every program has a well-behaved execution, i.e., each read sees a write that happens before it, and consequently also at least one legal execution. However, this also complicates the model: In Fig. 18, the read in l. 2 sees the allocation which always executes after l. 2. In general, such subsequent allocations might depend on the values read. As initialisations are considered to having taken place at the start (instead of when allocations generate them), the JMM legality constraints cannot catch such causality cycles. Consequently, the proofs have to take care of that themselves, see §3.4 and §5.2 for examples.

2.8. Non-termination

Since allocations are special, I must consider complete executions, which may be infinite. I show that considering only finite prefixes of executions does not work, as the single-threaded semantics generates the initialisation events only at allocations. First, reconsider Fig. 18 to illustrate the idea. Recall that the allocation event $(t_1, \textit{Alloc}\ a\ \texttt{A})$ initialises a's field `f` and happens before all other shown events, although the single-threaded semantics generates it after ll. 1 and 2. Take the prefix of this execution up to l. 2. The prefix is ill-formed, because $(t_1, \textit{Alloc}\ a\ \texttt{A})$ is not part of it and l. 2, therefore, sees no write.

For Fig. 18, one could extend the affected prefix a bit (e.g., with the allocation event) and obtain a well-formed prefix again. However, for Fig. 19, this is impossible. The program intertwines two copies of Fig. 18a and repeats them infinitely often. Consider the infinite execution in which each instance of l. 1 reads the address via ll. 9 and 10 of the object allocated in l. 7 of the same loop iteration and each instance of l. 5 reads via ll. 11 and 12 the address of the object allocated in l. 3 of the *next* iteration. This execution is legal. However, all finite prefixes that include at least one instance of l. 2 are ill-formed: Suppose such a prefix was well-formed. Let α_2 denote the last instance of l. 2 in the prefix and i the corresponding loop iteration (α_2 exists because the prefix is finite). By assumption, α_2 reads the default value that the allocation from the i-th instance of l. 7 writes, i.e., the prefix also contains this allocation. Hence, the prefix also contains the i-th instance of l. 5 – which reads the address of the object that l. 3 will allocate in the $i + 1$-th iteration – and therefore also the read is l. 6 in the i-th loop iteration. As the prefix is well-formed and the read can only see the allocation, the prefix also contains the $i + 1$-th instance of l. 3 – and thus also l. 2's $i + 1$-th instance. This contradicts α_2 being the last instance of l. 2, i.e., no such prefix is well-formed.

The program in Fig. 19 produces neither a final state nor intermediate output, so this execution is not observable from the outside. However, a simple extension remedies

this: At the end of each iteration, let t_1 output the values of `r1`, `r3`, and `r4` from the *previous* iteration – this requires another three local variables to save their values till the next iteration, but this shift does not generate additional events. Then, the execution is observable. Note that the loop must not output these local variables in the current iteration, because the write to `w` in the next iteration must be committed before the read of `z` can read its value (via ll. 11 and 12). As L10 requires to commit the output no later than the write, the justification can no longer change the read value. Shifting output by one iteration lets the output happen after the write to be committed and L10 is easy to abide by.

In this example, the trace is infinite. Yet, some programs, e.g., `while (true) ;`, also run forever without producing any events, i.e., diverge. The JMM covers this case with thread divergence events. They "model how a thread may cause all other threads to stall and fail to make progress" [Gosling et al. 2005]. Thread divergence events and deadlock can be observed as a "hang" action.

My approach with the coinductive trace definition does not need thread divergence events. If a thread can diverge in state s, an infinite derivation with STEP yields $s \downarrow$ *repeat* [], where *repeat* [] denotes the infinite list of empty lists. It is coinductiveness that allows such infinite derivations. Since *concat* (*repeat* []) = [], the trace of the diverging program does not contain any events. In particular, I need not include infinitely many thread divergence events to stop other threads from executing. Neither do I require explicit hang events. If a finite trace does not contain *Finish* events for all spawned threads, the program has either diverged or deadlocked – or the semantics got stuck, but type safety shows that this is not possible (§5).

Moreover, the coinductive definitions (and coinduction as proof principle) allows to treat finite and infinite executions uniformly – as can be seen in §3. This is important because the above example shows that one must be very careful in approximating infinite executions by their finite prefixes.

2.9. Spurious Wake-Ups and Deadlock

The JLS allows, but does not require JVMs to perform "spurious wake-ups" [Gosling et al. 2005, §17.8.1], i.e., return from a call to *wait* without interruption and notification. Spurious wake-ups are a delicate matter, because they affect when programs are correctly synchronized. For example, the program in Fig. 20a is correctly synchronized iff spurious wake-ups are not allowed. Only two accesses conflict, namely the write in l. 1 and the read in l. 8. If spurious wake-ups were not allowed, there is only one sequentially consistent execution that contains l. 8 (Fig. 20b). Line 8 can only execute after l. 6 has terminated normally, i.e, t_2 has been notified. However, only t_1 can do so and the write precedes the notification in t_1's `synchronized` block. When t_2 returns from `wait`, it reacquires `m`'s lock, which t_1's unlocking synchronises with. Hence, l. 1 happens before l. 8. Without spurious wake-ups, this is the only SC execution with conflicting accesses, so Fig. 20a would be correctly synchronized.

However, Java allows spurious wake-ups, and with spurious wake-ups, the execution in Fig. 20c is also SC. Now, t_2 spuriously wakes up before the notification and leaves the `synchronized` block. Hence, ll. 1 and 8 may happen in parallel, i.e., there is a data race, so Fig. 20a is *not* correctly synchronised.

In my semantics, threads can wake up spuriously, but they do not have to. Two rules apply for a thread t calling *wait* on monitor m provided that t is not interrupted and has locked m. The one releases the lock on m and suspends t to m's wait set; t then waits until it is notified or interrupted. The other only releases the lock on m, i.e., t instantaneously wakes up spuriously. Since I do not model a specific scheduler, t may postpone to reacquire m's lock as long as other threads make progress. Hence,

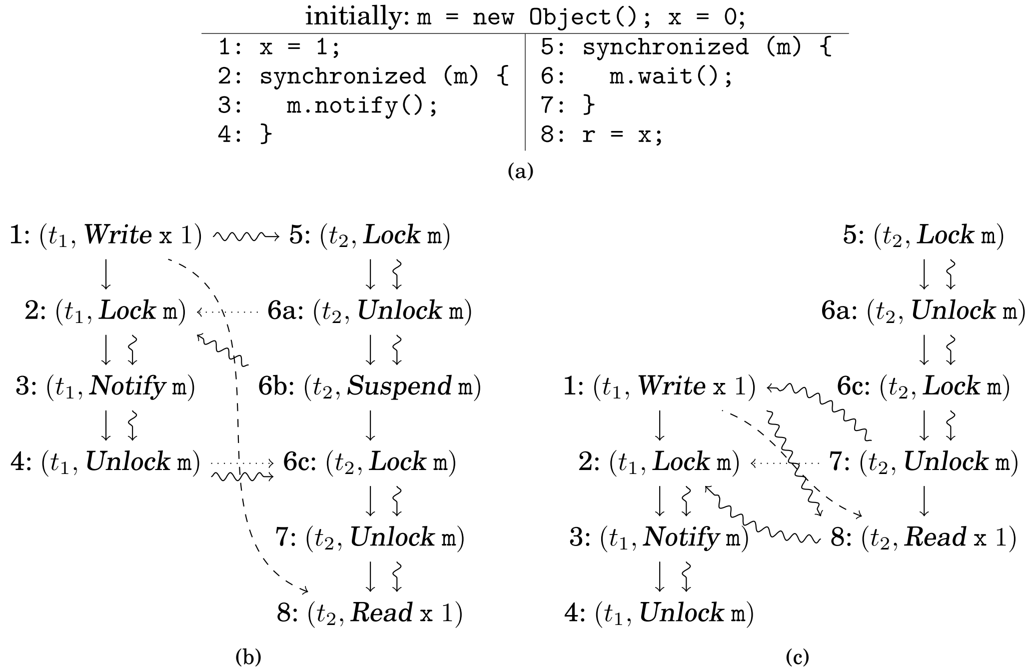


Fig. 20: Two sequentially consistent executions for the given program which contain l. 8. $\rightsquigarrow$ denotes the global SC order.

instantaneous wake-ups cover all spurious wake-ups, because other threads cannot tell whether another thread has woken up spontaneously.

A simpler model for spurious wake-ups would be to include a rule without preconditions that removes any thread from any wait set any time. However, this does not work well with considering complete traces as discussed in §2.8: If there is at least only one possibility for reduction, one such will be taken. This assumption is also the basis for type safety proofs via progress and preservation. However, this can enforce spurious wake-ups (rather than discouraging them as the JLS does). Consider, e.g.,

```
m = new Object(); synchronized (m) { m.wait(); } print "X";        (P4)
```

When run with Oracle's Hotspot and OpenJDK VMs, this program with only one thread deadlocks, because the thread waits forever for being notified or interrupted, but there is no thread to do so. Hence, it never prints `X`.

With the simpler model for spurious wake-ups, such a deadlock could not occur. The semantics would wake-up the thread and run it to completion, i.e., the program would always print `X` and terminate. My semantics produces both behaviours: If the call to wait chooses to instantaneously wake up, the program prints `X` and terminates. Otherwise, it chooses to wait in the wait set and deadlocks.

However, my model requires additional inter-thread events to implement wait sets. In particular, *notify* and *notifyAll* are more than no-ops and spurious wake-ups as the JMM suggests. For example, P5 always terminates in Java, as the following reasoning shows. Note that the program is correctly synchronised, i.e., by the DRF guarantee, we only need to consider SC executions. If t_1 runs first, it enters m's wait set, but t_2 notifies it and notifications must not get lost, so both terminate. Otherwise, t_2 runs first; the notification has no effect, but t_1 does not call *wait* either. Yet, if I implemented *notify* as

initially: `m = new Object();`

`1: synchronized (m) {`	`4: synchronized (m) {`	`8: synchronized (m) {`
`2:   m.wait();`	`5:   t1.interrupt();`	`9:   m.wait();`
`3: }`	`6:   m.notify();`	`10: }`
	`7: }`	

Fig. 21: Program for which Oracle's HotSpot VM determines that notification causes t_1 to return from `m`'s wait set

a no-op (hoping that the thread to notify wakes up spuriously), P5 could also deadlock, if the call to wait chooses not to spuriously wake up instantaneously.

initially: `m = new Object(); x = 0;`

`1: synchronized (m) {`	`5: synchronized (m) {`
`2:   if (x == 0)`	`6:   m.notify();`
`3:     m.wait();`	`7:   x = 1;`
`4: }`	`8: }`

(P5)

2.10. Consistency of Interruption and Notification

While a thread t is in a wait set, i.e., it has called *wait*, but the call has not yet returned, it may be notified and interrupted simultaneously. The JLS demands that t determine an order over these causes and behave accordingly. That is, if the notification comes first, t's interrupt flag is set and the call returns normally. If the interrupt comes first, t's call throws an *InterruptedException*, but the notification must not be lost, i.e., another thread in the wait set must be notified, if there is any.

My semantics implements these requirements as follows. In case of an interrupt, the call to *interrupt* atomically removes t from the wait set (event *WakeUp* in INTR) and the interleaving semantics remembers that t has left the wait set due to an interrupt. After t has reacquired its lock, it throws the exception. Similarly, *notify* removes one thread from the wait set and the interleaving semantics ensures that it will continue normally. For details of the implementation, see [Lochbihler 2012b]. This also ensures that the order of causes is consistent with the other orders of the JMM, in particular $\leq_{\mathrm{hb}}$.

However, the JLS does not require this order to be consistent. In fact, Oracle's HotSpot 6 and 7 sometimes choose an inconsistent order, e.g., for the program in Fig. 21. When t_1 and t_3 both enter m's wait set before t_2 acquires m's monitor, t_1 returns normally with its interrupt flag set and t_3 is deadlocked in the call to wait. Hence, t_1 determines that the notification preceded the interrupt, although the interrupt happens before the notification in t_2. Note that t_1 does not wake up spuriously, because otherwise, the notification would have to reach t_3 and t_3 therefore would have to complete normally.

My semantics cannot produce this behaviour, because its ordering of causes is always consistent. In principle, it would be easy to adjust my semantics to model HotSpot's behaviour, but this would not solve the problem. Actually, the semantics should predict *all* allowed behaviours, but the current architecture does not support this. I leave this as future work.

3. THE DATA RACE FREEDOM GUARANTEE

The JMM promises that correctly synchronised programs behave as if they were executed under sequential consistency. In this section, I recapitulate the definitions and identify the assumptions of this guarantee (§3.1). Then, I show that source code and bytecode indeed satisfy these assumptions (§3.2 to §3.6).

The proof of the DRF guarantee extends over all layers of the semantics stack (Fig. 6). Hence, the challenge consists of adequately decomposing the proof and distributing it over the layers such that each proof is as abstract as possible. This way, I prove the DRF guarantee for both implementations of shared type information (§2.1) and for both source code and bytecode (Thm. 3.24) almost simultaneously. To that end, I develop the assumptions that each layer makes about the lower ones. It is crucial that these assumptions respect the abstraction of the layer, i.e., they only refer to notions of the current layer or of layers below, but not above. For example, assumptions about the shared type information must not mention JMM executions.

Therefore, I focus on identifying and formalising these assumptions. The transition from the global behaviour (executions and traces) to the individual steps of the small-step semantics is the most difficult one, because it must translate global notions into state invariants. Sometimes, it is better to strengthen assumpions: For example, levels 5 and below generalise the happens-before order to the execution order, because happens-before is hard to express as a state invariant. To derive and motivate the low-level assumptions from those on higher levels, the presentation starts with the proofs on the JMM level and then descends the stack of semantics, similar to backward-style reasoning.

On the JMM level, some assumptions state that there must be some execution with some property such as SC, others require that some event must occur in a given execution. From a modelling perspective, the two kinds are entirely different. The former kind views the JMM as a specification, i.e., a set of legal behaviours from which implementations may pick any (non-empty) subset – in particular, the implementation need not meet the assumption. When I use them to prove safety properties (DRF guarantee and type safety), every implementation also enjoys these properties, even if the implementation does not meet the assumption. For example, one could imagine a weird, but correct processor and VM that cannot produce an SC execution for some program with data races; but the DRF guarantee and type safety still hold. In contrast, the latter kind extends down to the implementation level: every implementation must also satisfy the assumption, if the implementation can produce the given execution.

3.1. The DRF Guarantee

In this section, I formally state the DRF guarantee and prove it. Two events of an execution *conflict* if they are read or write events to the same non-volatile location with at least one being a write event. Two conflicting events constitute a *data race* if they are not ordered by happens-before, i.e., may happen concurrently.

An execution (ξ, ws) is *sequentially consistent (SC)* iff every read event $\alpha \in \mathcal{R}$ sees the most recent write event, i.e., $ws\ \alpha \leq_{\mathrm{eo}} \alpha$, and $\beta \leq_{\mathrm{eo}} ws\ \alpha$ or $\alpha \leq_{\mathrm{eo}} \beta$ for all write events β to the location that α reads from.[19]

A program is *correctly synchronised* (*data race free*) iff none of its SC executions contains a data race. Formally: Whenever $\xi \in \mathcal{E}$, $(\xi, ws)\surd$ and (ξ, ws) is SC, then $\alpha \leq_{\mathrm{hb}} \alpha'$ or $\alpha' \leq_{\mathrm{hb}} \alpha$ for all conflicting events $\alpha, \alpha' \in \mathcal{A}$.

THEOREM 3.1 (DRF GUARANTEE). *Let the program be correctly synchronised. If (ξ, ws) is a (weakly) legal execution, then (ξ, ws) is SC.*

Thanks to the DRF guarante, a programmer can forget about the JMM and legality in particular; she only has to avoid all data races in the program. To that end, it is

[19]The JMM only requires that $\leq_{\mathrm{po}}$ is extended to a total order over all events of all threads to determine most recent writes [Gosling et al. 2005, §17.4.3]. Aspinall and Ševčík [Aspinall and Ševčík 2007a] showed that, to respect mutual exclusion of locks, the total order must also extend $\leq_{\mathrm{so}}$. My execution order $\leq_{\mathrm{eo}}$ extends both by construction.

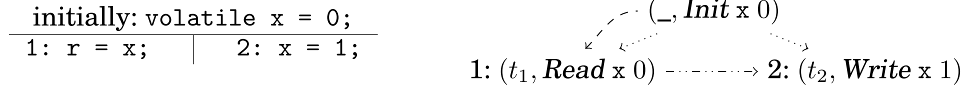


Fig. 22: A volatile read need not happen before a $\leq_{\mathrm{so}}$-later write. ⇢ represents $\leq_{\mathrm{so}}$-relationships that are not in $\leq_{\mathrm{hb}}$.

important that only SC executions must not contain a data race. Otherwise, she would have to understand the whole JMM to see whether her program is correctly synchronised and the DRF guarantee applies to it.

Moreover, it is crucial that accesses to volatile fields never race by definition. The JLS [Gosling et al. 2005, §17.4.5] and previous proofs of the DRF guarantee [Manson et al. 2005; Aspinall and Ševčík 2007a; Huisman and Petri 2007; Ševčík 2008] do not explicitly exclude volatile fields from data races. Jacobs [2005] has already observed that this prevents applying the DRF theorem to programs that use volatile fields for synchronisation. Recall from Fig. 12 that for volatile memory accesses, only writes synchronise with later reads, but not vice versa. This asymmetry can lead to the situation that a read does not happen before a $\leq_{\mathrm{so}}$-later write to the same volatile, i.e., a data race according to the JLS definition. In Fig. 22, e.g., l. 1 executes (in $\leq_{\mathrm{so}}$) before l. 2, but l. 1's read event happens neither before nor after l. 2's write event. Hence, the program would not be correctly synchronised in the JLS sense.

To my knowledge, I present the first proof of the DRF guarantee which addresses this flaw. As my notion of data race is stronger, correct synchronisation is *weaker*, i.e., more programs, e.g., Figs. 4 and 22, are correctly synchronised.

On the level of the JMM, my proof of the DRF guarantee (Thm. 3.1) follows in structure the others' [Aspinall and Ševčík 2007a; Huisman and Petri 2007; Manson et al. 2005]. The key idea in all of them is that in a DRF program, a well-formed execution (ξ, ws) is SC if it is well-behaved, i.e, every read sees a write that happens before it (Lem. 3.2). From this, the DRF guarantee (Thm. 3.1) follows. I omit the latter proof, as it closely follows Aspinall and Ševčík's [2007a, Thm. 1].

LEMMA 3.2 (DRF LEMMA). *Let $\mathcal{E}$ be correctly synchronised and $\xi \in \mathcal{E}$ such that $(\xi, ws)\surd$. If $ws\ \alpha \leq_{\mathrm{hb}} \alpha$ for every read α in $\mathcal{R}$, then (ξ, ws) is sequentially consistent.*

Let me outline the proof idea, see below for the technical details. To exploit correct synchronisation in a proof by contradiction, one first obtains a SC execution (ξ', ws') from (ξ, ws) as follows: ξ' starts like ξ until the first non-SC read α in ξ and continues sequentially consistently from there on. Then, it suffices to find a data race between α, $ws\ \alpha$, and $ws'\ \alpha$ in ξ'. For the latter, I use Lem. 3.4 (see below) to transfer happens-before relationships between ξ and ξ' on their common prefix. Thus, the proof rests on two assumptions on the set of traces $\mathcal{E}$:

(D1) For every sequentially consistent prefix of a well-formed execution (ξ, ws) with $\xi \in \mathcal{E}$, there is a trace $\xi' \in \mathcal{E}$ with the same prefix and a write seen-function ws' such that $(\xi', ws')\surd$ and (ξ', ws') is SC. If ξ immediately continues with a read after the prefix, ξ' also continues with a read from the same location.

(D2) Every execution $\xi \in \mathcal{E}$ initialises every location at most once.

The first assumption ensures that ξ' as required in the proof of Lem. 3.2 does exist; this is not trivial as prefixes of well-formed executions need not be well-formed (§2.8). This assumption is implicit in the original proof by Manson et al. [2005], as Huisman and Petri [2007] have pointed out. Aspinall and Ševčík [2007a] get away with a simpler "cut-and-update" property (§3.6), because they consider only finite prefixes,

which causes other problems (§2.8). The second assumption is merely a standard well-formedness condition, but nevertheless essential (for a discussion about "at most once" vs. "exactly once", see §5.3). Later, I show that source code and bytecode satisfy these. But now, let me prove Lem. 3.2. I start with two lemmata about happens-before:

LEMMA 3.3. *Let $\xi \surd_{\mathit{Start}}$ and $\alpha,\, \alpha' \in \mathcal{A}$ with init α and $\neg$init α'. Then $\alpha \leq_{\mathrm{hb}} \alpha'$.*

PROOF. Let ι be the event *Start* of α''s thread in ξ. By definition, $\alpha \leq_{\mathrm{sw}} \iota \leq_{\mathrm{po}} \alpha'$. □

LEMMA 3.4 (HAPPENS-BEFORE PREFIX LEMMA). *Let ξ and ξ' be two traces such that their first n events differ only in the values read or written, and let $\alpha, \alpha' < n$. If $\xi' \surd_{\mathit{Start}}$ and $\alpha \leq^{\xi}_{\mathrm{hb}} \alpha'$, then $\alpha \leq^{\xi'}_{\mathrm{hb}} \alpha'$.*

PROOF. By induction on $\alpha \leq^{\xi}_{\mathrm{hb}} \alpha'$, which is the transitive closure of $\leq^{\xi}_{\mathrm{po}}$ and $\leq^{\xi}_{\mathrm{sw}}$. In the base case, $\alpha \leq^{\xi}_{\mathrm{po}} \alpha'$ or $\alpha \leq^{\xi}_{\mathrm{sw}} \alpha'$. By unfolding the definitions, $\alpha \leq^{\xi'}_{\mathrm{po}} \alpha'$ or $\alpha \leq^{\xi'}_{\mathrm{sw}} \alpha'$ follows from $\xi_{[\alpha]} \simeq \xi'_{[\alpha]}$ and $\xi_{[\alpha']} \simeq \xi'_{[\alpha']}$. Hence, $\alpha \leq^{\xi'}_{\mathrm{hb}} \alpha'$.

In the induction step, assume $\alpha, \alpha'' < n$, and $\alpha \leq^{\xi}_{\mathrm{hb}} \alpha'$, and $\alpha' \leq^{\xi}_{\mathrm{po}} \alpha''$ or $\alpha' \leq^{\xi}_{\mathrm{sw}} \alpha''$, and the induction hypothesis if $\alpha' < n$, then $\alpha \leq^{\xi'}_{\mathrm{hb}} \alpha'$. I must show that $\alpha \leq^{\xi'}_{\mathrm{hb}} \alpha''$.

If $\neg\mathit{init}_{\xi}\ \alpha'$ or $\mathit{init}_{\xi}\ \alpha''$, then $\alpha' \preceq^{\xi} \alpha''$ by definition of $\leq^{\xi}_{\mathrm{eo}}$, because $\alpha' \leq^{\xi}_{\mathrm{eo}} \alpha''$ follows from either $\alpha' \leq^{\xi}_{\mathrm{po}} \alpha''$ or $\alpha' \leq^{\xi}_{\mathrm{sw}} \alpha''$. Since $\alpha'' < n$, also $\alpha' < n'$ and the induction hypothesis applies. Moreover, $\alpha' \leq^{\xi'}_{\mathrm{po}} \alpha''$ or $\alpha' \leq^{\xi'}_{\mathrm{sw}} \alpha''$ follow from $\alpha' \leq^{\xi}_{\mathrm{po}} \alpha''$ or $\alpha' \leq^{\xi}_{\mathrm{sw}} \alpha''$ as in the base case. Therefore, $\alpha \leq^{\xi'}_{\mathrm{hb}} \alpha''$.

Otherwise, I have $\mathit{init}_{\xi}\ \alpha'$ and $\neg\mathit{init}_{\xi}\ \alpha''$. Then, $\mathit{init}_{\xi}\ \alpha$ follows from $\mathit{init}_{\xi}\ \alpha'$ by induction on $\alpha \leq^{\xi}_{\mathrm{hb}} \alpha'$. Since $\alpha, \alpha'' < n$ and ξ's and ξ''s first n actions only differ in the values read or written, $\mathit{init}_{\xi'}\ \alpha$ and $\neg\mathit{init}_{\xi'}\ \alpha''$, too. Hence $\alpha \leq^{\xi'}_{\mathrm{hb}} \alpha''$ by Lem. 3.3. □

PROOF OF LEM. 3.2. By contradiction. Suppose that (ξ, ws) is not SC. Note that $\leq^{\xi}_{\mathrm{eo}}$ is well-founded by construction. Let $\alpha \in \mathcal{R}_{\xi}$ be the $\leq^{\xi}_{\mathrm{eo}}$-minimal read event from some location (a, l) such that $ws\ \alpha$ is not the most recent write for α in ξ. By assumption, $ws\ \alpha \leq^{\xi}_{\mathrm{hb}} \alpha$. Then, there is another write event $\beta \in \mathcal{W}_{\xi}$ to (a, l) such that $\beta \not\leq^{\xi}_{\mathrm{hb}} ws\ \alpha$ and $\beta \not\leq^{\xi}_{\mathrm{so}} ws\ \alpha$ and $\alpha \not\leq^{\xi}_{\mathrm{hb}} \beta$ and $\alpha \not\leq^{\xi}_{\mathrm{so}} \beta$ and $\beta \leq^{\xi}_{\mathrm{eo}} \alpha$ – otherwise, $ws\ \alpha$ would be the most recent write for α. First, the member l is not volatile: Assume it was. Then, α, β, and $ws\ \alpha$ are synchronisation events. Since $\leq^{\xi}_{\mathrm{so}}$ is total on synchronisation events, $ws\ \alpha \leq^{\xi}_{\mathrm{so}} \beta \leq^{\xi}_{\mathrm{so}} \alpha$ follows from $\beta \not\leq^{\xi}_{\mathrm{so}} ws\ \alpha$ and $\alpha \not\leq^{\xi}_{\mathrm{so}} \beta$. This contradicts the sychronisation order consistency condition W5, as $\beta \neq ws\ \alpha$. Second, $\neg\mathit{init}_{\xi}\ \beta$, as otherwise $\neg\mathit{init}_{\xi}\ (ws\ \alpha)$, because ξ initialises every location at most once (D2), and therefore $\beta \leq^{\xi}_{\mathrm{hb}} ws\ \alpha$ by Lem. 3.3, which contradicts $\beta \not\leq^{\xi}_{\mathrm{hb}} ws\ \alpha$. With $\beta \leq^{\xi}_{\mathrm{eo}} \alpha$, it follows by definition of $\leq_{\mathrm{eo}}$ that β occurs before α in ξ, i.e., $\beta \preceq^{\xi} \alpha$.

By requirement D1, obtain a well-formed execution (ξ', ws') that starts with ξ up to α and continues SC, with α being a read from (a, l) in ξ'. Then, $\alpha, \beta \in \mathcal{A}_{\xi'}$ conflict and (ξ', ws') is SC, so $\alpha \leq^{\xi'}_{\mathrm{hb}} \beta$ or $\beta \leq^{\xi'}_{\mathrm{hb}} \alpha$ by correct synchronisation. By Lem. 3.4, $\alpha \leq^{\xi}_{\mathrm{hb}} \beta$ or $\beta \leq^{\xi}_{\mathrm{hb}} \alpha$, too; but $\alpha \not\leq^{\xi}_{\mathrm{hb}} \beta$, so $\beta \leq^{\xi}_{\mathrm{hb}} \alpha$.

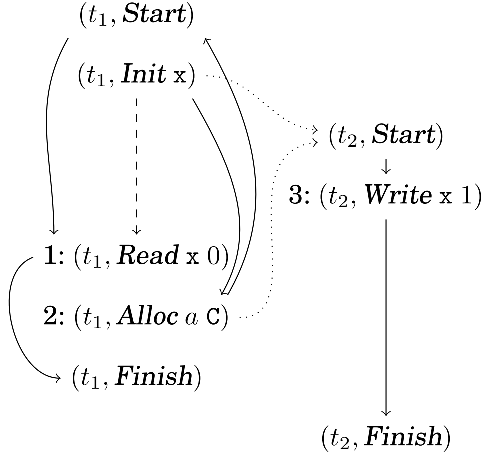
Now, it suffices to show that $ws\ \alpha \leq^{\xi}_{\mathrm{hb}} \beta$, as this violates the happens-before consistency condition W4. If $\mathit{init}_{\xi}\ (ws\ \alpha)$, then $ws\ \alpha \leq^{\xi}_{\mathrm{hb}} \beta$ by Lem. 3.3. So, suppose $\neg\mathit{init}_{\xi}\ (ws\ \alpha)$. As $ws\ \alpha \leq^{\xi}_{\mathrm{hb}} \alpha$, $ws\ \alpha$ also occurs before α in ξ, i.e., $ws\ \alpha \preceq^{\xi} \alpha$. Hence, $\beta, ws\ \alpha \in \mathcal{A}_{\xi'}$ conflict. By correct synchronisation, $\beta \leq^{\xi'}_{\mathrm{hb}} ws\ \alpha$ or $ws\ \alpha \leq^{\xi'}_{\mathrm{hb}} \beta$. Again

```
class C { volatile int v; }
```
t_1 initialises `x = 0;`

`1: r1 = x;` `2: if (r1 == 0) r2 = new C();`	`3: x = 1;`

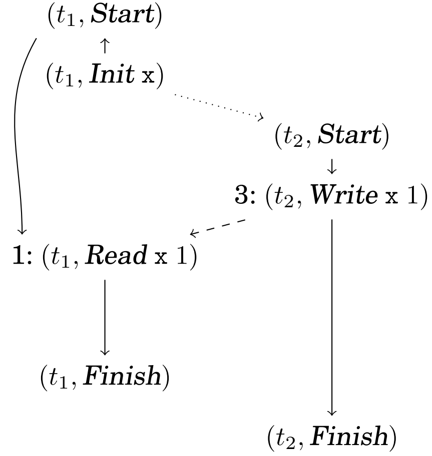
(a) Program with a data race on x

$\xi_b = [(t_1, \textit{Start}), (t_1, \textit{Init}\ \mathtt{x}), (t_2, \textit{Start}), (t_2, \textit{Write}\ \mathtt{x}\ 1), (t_1, \textit{Read}\ \mathtt{x}\ 0), (t_1, \textit{Alloc}\ a\ \mathtt{C}), (t_1, \textit{Finish}), (t_2, \textit{Finish})]$



(b) A well-behaved execution that is not SC

$\xi_c = [(t_1, \textit{Start}), (t_1, \textit{Init}\ \mathtt{x}), (t_2, \textit{Start}), (t_2, \textit{Write}\ \mathtt{x}\ 1), (t_1, \textit{Read}\ \mathtt{x}\ 1), (t_1, \textit{Finish}), (t_2, \textit{Finish})]$



(c) An SC execution that is not well-behaved

Fig. 23: Two well-formed traces and their executions for the program at the top

with Lem. 3.4, these relationships hold also in ξ: $\beta \leq^{\xi}_{\text{hb}} ws\ \alpha$ or $ws\ \alpha \leq^{\xi}_{\text{hb}} \beta$, but $\beta \not\leq^{\xi}_{\text{hb}} ws\ \alpha$. □

It is worth noting that none of the previous proofs of the DRF lemma [Manson et al. 2005; Huisman and Petri 2007; Aspinall and Ševčík 2007a] depended on the synchronisation order $\leq_{\text{so}}$ at all. This suggested that $\leq_{\text{so}}$ was redundant from a DRF perspective. Yet, Fig. 4 shows that only $\leq_{\text{so}}$ guarantees global time. One can also see this directly in the proof: Since volatile locations never participate in data races by definition, I prove that l is not volatile ("First, ...") such that the events β and $ws\ \alpha$ conflict. This step requires $\leq^{\xi}_{\text{so}}$ being total and consistent (W5). In fact, without either of them, the non-SC execution in Fig. 4 would satisfy the assumptions of the DRF lemma, but not the conclusion.

Initialisations complicate the proof: Prefixes of traces need not be closed under happens-before, because the corresponding allocation event may occur in the trace after the prefix, see Fig. 18 for an example. Consequently, the proofs of Lems. 3.4 and 3.2 treat the case of initialisations specially. In particular, I require at most one initialisations per location (D2) (i.e., at most one allocation per address) and that *all* allocation events synchronise with *Start* events, i.e., they are synchronisation events.

Figure 23 shows that Lem. 3.4 and the proof of Lem. 3.2 require the latter. In Fig. 23b, we have exactly the situation as in the proof of Lem. 3.2: Each reads sees a write that happens before, but l. 1 is the first (and only) read that does not see the

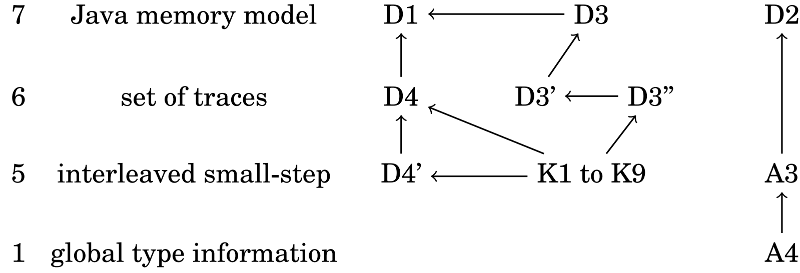


Fig. 24: Assumptions of the DRF guarantee and their decomposition over the stack of semantics

most recent write. The proof goes on by changing l. 1 to see the most recent write from l. 3 and completing sequentially consistently, e.g., as in Fig. 23c. This makes t_1 no longer allocate the `C` object in l. 2, which initialises the volatile member `v`.

Both traces share the prefix $[(t_1, \textit{Start}), (t_1, \textit{Init}\ \texttt{x}), (t_2, \textit{Start}), (t_2, \textit{Write}\ \texttt{x}\ 1)]$ followed by t_1 reading from `x`. If only initialisations for volatile locations synchronised with *Start* (as suggested by Manson [2007]), $(t_1, \textit{Init}\ \texttt{x})$ would not synchronize with $(t_2, \textit{Start})$, i.e., there would not be such a dotted arrow in Figs. 23b and 23c. For ξ_b, we would still get $(t_1, \textit{Init}\ \texttt{x}) \leq_{\text{hb}}^{\xi_b} (t_2, \textit{Write}\ \texttt{x}\ 1)$, because

$$(t_1, \textit{Init}\ \texttt{x}) \leq_{\text{po}}^{\xi_b} (t_1, \textit{Alloc}\ a\ \texttt{C}) \leq_{\text{sw}}^{\xi_b} (t_2, \textit{Start}) \leq_{\text{po}}^{\xi_b} (t_2, \textit{Write}\ \texttt{x}\ 1)$$

and the allocation $(t_1, \textit{Alloc}\ a\ \texttt{C})$ writes (initialises) the volatile member `v` from class `C`. However, for ξ_c, we would have $(t_1, \textit{Init}\ \texttt{x}) \not\leq_{\text{hb}}^{\xi_c} (t_2, \textit{Write}\ \texttt{x}\ 1)$.

3.2. At Most One Initialisation

In §3.1, I have shown the DRF guarantee under two assumptions on the set $\mathcal{E}$ of traces. In the remainder of this section, I discharge them for source code and bytecode by descending the stack of semantics (Fig. 6) and adapting the assumptions. They act like an interface between the levels and ensure that I can share the proofs for all layers that source code and bytecode share.

To help the reader follow the proofs, Fig. 24 shows how the assumptions D1 and D2 evolve and assigns them to the level of the semantics. The arrows are stylised implications, i.e., the assumption at the source discharges the one at the target. Where multiple arrows point to one assumption, the conjunction of the sources imply the target. For example, assumptions D4 and D3 together discharge D1. Layer 4 is not shown, because layer 4 is irrelevant for the assumptions that level 5 makes. Layer 3 and 2 are the core of the semantics where the crucial parts of the assumptions are discharged. Their assumptions about the shared state are listed at layer 1.

I start with assumption D2 that every execution initialises a location at most once. Remember that allocation events initialise locations. When an allocation returns an address, it was fresh before, but afterwards, it is allocated, i.e., not fresh. Hence, it suffices to prove that the semantics correctly keeps track of all memory allocations in the events.

To discharge D2 on the level of interleaving semantics, I extend the specification of the shared state with an operation *allocated* that returns the set of allocated addresses. I require that the single-threaded semantics changes the shared state only in accordance with *allocated* – assumption A3:

(A3) For every single-thread reduction $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$, $\overline{\alpha}$ contains exactly one allocation event *Alloc* a T iff a is newly allocated. Formally:

$$\textit{allocated}\ \sigma \subseteq \textit{allocated}\ \sigma' \tag{A3a}$$

$$\forall a.\ (a \in \textit{allocated}\ \sigma' - \textit{allocated}\ \sigma \longleftrightarrow (\exists T.\ \textit{Alloc}\ a\ T \in \overline{\alpha})) \tag{A3b}$$

$$\forall i, j < |\overline{\alpha}|.\ \forall a\ T\ T'.\ \overline{\alpha}_{[i]} = \textit{Alloc}\ a\ T \wedge \overline{\alpha}_{[j]} = \textit{Alloc}\ a\ T' \Longrightarrow i = j \tag{A3c}$$

The concept of allocated addresses reduces the global property of at most one initialisation to a property of single reductions.

LEMMA 3.5. *If start-events contains exactly one allocation event for every preallocated object in start-state and the single-threaded semantics satisfies A3, then every* $\xi \in \mathcal{E}$ *contains for every address* a *at most one event* $(_, \textit{Alloc}\ a\ _)$.

PROOF. By contradiction. Suppose $\xi \in \mathcal{E}$ contains two such allocation events α and β for address a. Without loss of generality, we may assume $\alpha < \beta$. By assumption, $|\textit{start-events}| \leq \beta$, because *start-events* contains at most one allocation event for a. Consider the sequence of reductions from *start-state* to the one (exclusive) that produces β. Let σ' denote the shared state after this sequence. If $|\textit{start-events}| \leq \alpha$, too, this sequence contains a reduction that produces α, because α and β cannot originate from the same reduction by assumption A3c. Denote the shared state after α's reduction with σ. From A3b and A3a, I get $a \in \textit{allocated}\ \sigma$ and $\textit{allocated}\ \sigma \subseteq \textit{allocated}\ \sigma'$, respectively. This contradicts $a \notin \textit{allocated}\ \sigma'$, which A3b requires for β's reduction. Otherwise, a is already allocated in *start-state* by assumption and the argument is analogous. □

Next, I also specify *allocated* with respect to the other operations on the shared state:

(A4) The empty heap has no allocated addresses. If successful, allocation updates the state σ to σ' such that the returned address is allocated in σ', but not in σ. In any case, the other addresses' allocation status remains unchanged. Formally:

$$\textit{allocated empty-}\sigma = \emptyset$$

$$(\sigma', a) \in \textit{alloc}\ \sigma \Longrightarrow \textit{allocated}\ \sigma' = \{\, a \,\} \cup \textit{allocated}\ \sigma \wedge a \notin \textit{allocated}\ \sigma$$

Under these assumptions, it is routine to show that *start-events* exactly records the pre-allocations of the *start-state* and that the single-threaded semantics of source code and bytecode meet assumption A3. Thanks to the abstract specifications, these proofs do not depend on the implementation of the shared state.

The implementations of the shared state implement *allocated* σ as

$$\textit{dom}\ (\textit{typeof-addr}\ \sigma) \qquad \text{and} \qquad \{\, (T, n) \mid n \in \sigma\ T \,\},$$

respectively. Both meet assumption A4. This concludes the proof of D2.

LEMMA 3.6 (ASSUMPTION D2).
Every execution $\xi \in \mathcal{E}$ *initialises every location at most once.* □

3.3. Sequential Consistency Coinductively

For the DRF guarantee, assumption D1 remains to be shown. However, the JMM definition of SC is not amenable to the coinductive definition of $_ \Downarrow _$ as it relies on the notions of write-seen function and most recent write, which are only defined for traces. Therefore, I introduce a coinductive version of SC and prove that it adequately models SC.

A snapshot of a sequentially consistent heap (*snapshot heap*) H is a map from locations to values. The empty heap is written *empty*. The function *mrw* H α updates the snapshot heap H if α is a write or initialisation event, else leaves H unchanged.

The function *mrws* folds *mrw* over finite lists of events. An event list $\overline{\alpha}$ is sequentially consistent (SC') for the snapshot heap H (denoted $H \vdash \overline{\alpha}\surd_{\text{sc}}$) iff

$$\frac{}{H \vdash []\surd_{\text{sc}}} \qquad \frac{\textit{mrw}\ H\ \alpha \vdash \overline{\alpha}\surd_{\text{sc}} \qquad \alpha = \textit{Read}\ (a,l)\ v \implies H\ (a,l) = \lfloor v \rfloor}{H \vdash \alpha \cdot \overline{\alpha}\surd_{\text{sc}}}$$

i.e., the empty list is SC' for all snapshot heaps, and $\alpha \cdot \overline{\alpha}$ is SC' for H iff $\overline{\alpha}$ is SC' for the updated snapshot heap *mrw* H α and if α reads the value v from a location (a, l), then the snapshot heap H must store v at (a, l).

The next theorem shows that *empty* $\vdash _\surd_{\text{sc}}$ and sequential consistency are equivalent under the following assumption:

(D3) Initialisations precede reads in ξ. If $\alpha \in \mathcal{R}$ reads from some location (a, l), then there is a write event $\beta \in \mathcal{W}$ such that $\beta \preceq \alpha$ and *init* β and $(a, l) \in$ *locs* β.

Thus, I can use coinduction to show an execution being SC. Again, coinductivity permits to handle finite and infinite executions uniformly.

THEOREM 3.7 (EQUIVALENCE OF SC AND SC').

(*a*) *If ξ initialises every location at most once (assumption D2) and empty $\vdash \xi\surd_{\text{sc}}$ and $\xi\surd_{\textit{Start}}$, then there is a ws such that $(\xi, ws)\surd$ and (ξ, ws) is SC.*

(*b*) *If initialisations precede reads in ξ (assumption D3) and $(\xi, ws)\surd$ and (ξ, ws) is SC, then empty $\vdash \xi\surd_{\text{sc}}$.*

PROOF. (a) Set $ws\ \alpha$ to be the most recent write for $\alpha \in \mathcal{R}$ to location (a, l). The condition *empty* $\vdash \xi\surd_{\text{sc}}$ ensures that there is a write event for every read, D2 guarantees the existence of the most recent one.[20] Then, (ξ, ws) is SC by definition. For $(\xi, ws)\surd$, only condition W2 of well-formedness, i.e., α reads *vw* $(ws\ \alpha)$ (a, l), is interesting. Let $\overline{\alpha}$ be the prefix of ξ up to α. From *empty* $\vdash \xi\surd_{\text{sc}}$, I obtain that *mrws empty* $\overline{\alpha}$ $(a, l) = \lfloor v \rfloor$ and α reads the value v. Since $ws\ \alpha$ is the most recent write for α in ξ, assumption D2 and *empty* $\vdash \xi\surd_{\text{sc}}$ ensure that $ws\ \alpha < \alpha$. Hence, *vw* $(ws\ \alpha)$ $(a, l) = v$ holds.

(b) Suppose $(\xi, ws)\surd$ and (ξ, ws) is SC. Let $\alpha \in \mathcal{R}_\xi$ read v from location (a, l), and let $\overline{\alpha}$ denote the prefix of ξ up to α. Since initialisations precede reads in ξ, the most recent write $ws\ \alpha$ precedes α, i.e. $ws\ \alpha \preceq \alpha$. Well-formedness condition W2 of $(\xi, ws)\surd$ yields that $v =$ *vw* $(ws\ \alpha)$ (a, l). Since $ws\ \alpha$ is the most recent write for α and $ws\ \alpha \preceq \alpha$, I also have *mrws empty* $\overline{\alpha}$ $(a, l) = \lfloor v \rfloor$. As this holds for all reads α, *empty* $\vdash \xi\surd_{\text{sc}}$ follows by coinduction. □

COROLLARY 3.8. *Let unique initialisations precede reads in ξ (assumptions D2 and D3) and $\xi\surd_{\textit{Start}}$. Then, empty $\vdash \xi\surd_{\text{sc}}$ iff there is a ws such that $(\xi, ws)\surd$ and (ξ, ws) is SC.* □

This equivalence holds only if the initialisation of every location (a, l) occurs before the first read from (a, l) in the trace. For example, the ficticious trace

$$\xi = [(t, \textit{Start}), (t, \textit{Read}\ a[0]\ 0), (t, \textit{Alloc}\ a\ \textit{Integer}[1])]$$

is SC for $ws\ 1 = 2$, but not SC', i.e. $\neg$ *empty* $\vdash \xi\surd_{\text{sc}}$. Such problematic traces do occur in the JMM: Figure 18 shows a (non-SC) execution of a type-correct program that violates assumption D3: The initialisation of $(a, \texttt{A::f})$ in l. 3 occurs after the read in l. 2. Thus, in order to exploit this equivalence, I show that initialisations precede reads in SC' prefixes of a trace:

[20]Assumption D2 is necessary. For example, suppose that ξ initialises some location (a, l) infinitely often, but there is no *Write* (a, l) _. Then, a read of the default value in ξ from (a, l) would be SC', but not SC, because none of the initialisations is most recent.

(D3') If a trace $\xi \in \mathcal{E}$ has an SC' prefix $\overline{\alpha}$ followed by a read from (a, l), $\overline{\alpha}$ initialises (a, l).

LEMMA 3.9. *Assumption D3' implies D3 for all $\xi \in \mathcal{E}$ that are SC'.* □

3.4. Initialisations Precede Reads

Next, I tackle D3' by decomposing it into smaller assumptions that no longer refer to traces, but only to single reductions in the small-step semantics – similar to what I did for assumption D2 above. At the same time, I prove it for a more general class of prefixes such that I can reuse this assumption when proving consistency in §4.

A heap record $\mathcal{H}$ is a function from locations to sets of values – it records all values that have been written to a location. Similar to *mrw*, if α is a write or initialisation event, the function *uhr* $\mathcal{H}$ α adds the written value(s) to the heap record $\mathcal{H}$, else it leaves $\mathcal{H}$ unchanged. The function *uhrs* folds *uhr* over finite lists of events. An event list $\overline{\alpha}$ is *non-speculative* with respect to the heap record $\mathcal{H}$ (denoted $\mathcal{H} \vdash \overline{\alpha}\surd_{\mathrm{ns}}$) iff for any read event α in $\overline{\alpha}$ from any location (a, l), α reads a value that has been written to (a, l) in $\overline{\alpha}$ before α or that has already been in $\mathcal{H}$ (a, l). Formally:

$$\frac{}{\mathcal{H} \vdash []\surd_{\mathrm{ns}}} \qquad \frac{\textit{uhr}\ \mathcal{H}\ \alpha \vdash \overline{\alpha}\surd_{\mathrm{ns}} \qquad \alpha = \textit{Read}\ (a, l)\ v \Longrightarrow v \in \mathcal{H}\ (a, l)}{\mathcal{H} \vdash \alpha \cdot \overline{\alpha}\surd_{\mathrm{ns}}}$$

A prefix of a trace is non-speculative iff its list of events is non-speculative with respect to the empty heap record $\lambda_.\ \emptyset$.

A snapshot heap H *fits to* a heap record $\mathcal{H}$ iff whenever H stores a value for a location (a, l), then $\mathcal{H}$ (a, l) contains that value.

LEMMA 3.10. *If $\overline{\alpha}$ is SC' for H and H fits to $\mathcal{H}$, then $\overline{\alpha}$ is non-speculative with respect to $\mathcal{H}$.* □

COROLLARY 3.11. *An SC' prefix of a trace is non-speculative.* □

This corollary shows that it suffices to prove the following assumption D3" instead of D3'.

(D3") If a trace $\xi \in \mathcal{E}$ has a non-speculative prefix $\overline{\alpha}$ followed by a read from (a, l), then $\overline{\alpha}$ initialises (a, l).

To discharge D3", it suffices to show that a thread cannot make up addresses (§3.4.1) and it accesses only the declared fields of objects and cells within the bounds of the array (§3.4.2). Then, since the prefix does not speculate, the read can only access an existing member of an address that has been allocated before. Since allocation initialises all fields and array cells, the member therefore must have been initialised. In §3.4.3, I make this reasoning formal.

3.4.1. Addresses are not Made Up. To show that addresses are never made up, I introduce the concept of *known addresses*. Let *ka* t x denote the set of addresses that thread t with local state x knows about. This set consists of all addresses stored in the local state x, e.g., in local variables, of t's associated *Thread* object, and of the addresses of the preallocated system exceptions. A thread *learns an address* a in an event list $\overline{\alpha}$ iff $\overline{\alpha}$ contains a read event *Read* _ (*Addr* a) or an allocation *Alloc* a _, i.e., it either reads the address from some location or allocates it. The function *learns* $\overline{\alpha}$ computes the set of learnt addresses from $\overline{\alpha}$. The single-threaded semantics *does not invent addresses* iff *ka* satisfies all of the following for any reduction $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$:

(K1) *ka* t $x' \subseteq$ *ka* t $x \cup$ *learns* $\overline{\alpha}$, i.e., after the reduction step, t knows only addresses that it has known before or has learnt in this step.

(K2) Whenever *Spawn* t' x'' $\in \overline{\alpha}$, then *ka* t' $x'' \subseteq$ *ka* t x, i.e., a spawned thread may only know those addresses that the spawning thread knows.
(K3) Whenever *Read* $(a, _)$ $_ \in \overline{\alpha}$, then $a \in$ *ka* t x, i.e., t may only read from members of known addresses.
(K4) If $\overline{\alpha}_{[i]}$ = *Write* _ (*Addr* a) for $i < |\overline{\alpha}|$, then $a \in$ *ka* t $x \cup$ *learns* (*take* i $\overline{\alpha}$), i.e., if t writes an address a into memory, it must have known a before or just learnt (*take* i $\overline{\alpha}$ denotes the first i events from $\overline{\alpha}$).

Note the asymmetry between reads and writes. It suffices to restrict the reads to members at known addresses; the writes may write to any address, as threads cannot read from such a location if they do not know it. Conversely, if an address value is written to memory, the address must not be made up. I explicitly allow writing an address that has been being learnt before in the same reduction, because this allows to easily implement atomic operations. For example, "read-don't-modify-write" [Boehm 2012] translates to [*Read* (a, l) v, *Write* (a, l) v]; the interleaving semantics ensures that they are executed atomically. Technically, it would be fine to immediately read from a learnt address, too, but this would unnecessarily complicate the proofs.

Proving conditions K1 to K4 is standard: Induction on the small-step semantics and case analysis on the executed instruction, respectively.

LEMMA 3.12.
The single-threaded semantics for source code and bytecode do not invent addresses. □

The concept of known addresses naturally extends to multithreaded states and the interleaving semantics. I write *kas* s for the set of addresses that the state s knows; it is the union of the known addresses of all threads in s.

LEMMA 3.13. *Let* $s -(t, \overline{\alpha})\rightarrow s'$. *Then* *kas* $s' \subseteq$ *kas* $s \cup$ *learns* $\overline{\alpha}$. *If Read* $(a, _)$ $_ \in \overline{\alpha}$*, then* $a \in$ *kas* s. □

Let the *recorded addresses addrs* $\mathcal{H}$ be the set of all addresses in the heap record $\mathcal{H}$, i.e., *addrs* $\mathcal{H} = \{ a \mid \exists a'\ l.\ \mathit{Addr}\ a \in \mathcal{H}\ (a', l) \}$. Then, the interleaving semantics preserves the invariant that all known or recorded addresses are allocated for non-speculative executions (Lem. 3.14). The proof goes by case analysis of the reduction and induction on the prefixes of $\overline{\alpha}$; *shr* s extracts the shared state from s:

LEMMA 3.14. *Let* $s -(t, \overline{\alpha})\rightarrow s'$ *such that* $\mathcal{H} \vdash \overline{\alpha}\surd_{\mathrm{ns}}$. *If kas* $s \cup$ *addrs* $\mathcal{H} \subseteq$ *allocated* (*shr* s)*, then kas* $s' \cup$ *addrs* (*uhrs* $\mathcal{H}$ $\overline{\alpha}$) $\subseteq$ *allocated* (*shr* s'). □

Now, I can prove that non-speculative prefixes allocate addresses before they read from their locations, which is the first part of proving D3". Let *start-*$\mathcal{H}$ denote the start heap record *uhrs* ($\lambda_.\ \emptyset$) *start-events*, let *start-*σ = *shr start-state* denote the initial shared state, and let *events'* $(t, \overline{\alpha})$ denote the list of original JMM inter-thread actions in α, i.e., *events'* is like *events* except for not pairing the events with the thread ID t.

LEMMA 3.15. *Let start-state* $-\overline{t\overline{\alpha}}\rightarrow^{*} s$ *and* $s -(t, \overline{\alpha})\rightarrow s'$ *with Read* $(a, _)$ $_ \in \overline{\alpha}$ *such that* $\lambda_.\ \emptyset \vdash$ *start-events ++ concat* (*map events'* $\overline{t\overline{\alpha}}$)$\surd_{\mathrm{ns}}$. *Suppose further that kas start-state* $\subseteq$ *allocated start-*σ. *Then, either*

(*a*) a *is preallocated, i.e., Alloc* a $T \in$ *start-events for some* T*, or*
(*b*) *some reduction has allocated* a*, i.e., there are* t'*,* $\overline{\alpha}'$*, and* T *such that* $(t', \overline{\alpha}') \in \overline{t\overline{\alpha}}$ *and Alloc* a $T \in \overline{\alpha}'$.

PROOF. If $a \in$ *allocated start-*σ, then (a) holds by construction of *start-*σ and *start-events*. So suppose $a \notin$ *allocated start-*σ. Let $\overline{\alpha}^{*}$ abbreviate *concat* (*map events'* $\overline{t\overline{\alpha}}$). Since allocations write default values, which are never addresses, *addrs start-*$\mathcal{H} = \emptyset$ by construction of *start-events*. Therefore, *kas* $s \cup$

addrs (*uhrs start-*$\mathcal{H}$ $\overline{\alpha}^*$) $\subseteq$ *allocated* (*shr* s) by Lem. 3.14 and induction on *start-state* $-\overline{t\overline{\alpha}}\rightarrow^*$ s. In particular, I have $a \in$ *allocated* (*shr* s), because $a \in$ *kas* s by Lem. 3.13. Since $a \notin$ *allocated start-*σ, induction on *start-state* $-\overline{t\overline{\alpha}}\rightarrow^*$ s yields (b) using A3b.[21] □

3.4.2. Reads Access Only Declared Locations. I now turn to the second part of assumption D3”, namely to show that the location being read is a declared field or a cell within the bounds of an array. The proof approach combines known addresses with conformance and monotonicity of type information. Let me first introduce types. The *type of a value* v in shared state σ is for primitive values the corresponding primitive type, e.g., *Integer*, or for the null reference the null type, or *typeof-addr* σ a if $v =$ *Addr* a is an address. If l is a member of a's type, the *type of the location* (a, l) is defined as follows: If l denotes a field $A{::}f$, then it is the type declared for field f in class A. If l denotes an array cell, then it is the type of the elements of the array. A value *conforms* to a type (written $\sigma \vdash v :\leq T$) if v's type is defined and a subtype of T. A heap record $\mathcal{H}$ conforms to σ (notation $\sigma \vdash \mathcal{H}\surd$) iff all values in $\mathcal{H}$ conform to the location's type, i.e., whenever $v \in \mathcal{H}$ (a, l), then *typeof-addr* σ $a = \lfloor T \rfloor$ for some T such that l is a member of T and v conforms to (a, l)'s type.

Further, let Q denote the invariant for the subject reduction proof of the single-threaded semantics. For source code, it consists of run-time typeability and further conformance notions; for bytecode, it is bytecode conformance – see [Lochbihler 2012b] for details. The standard subject reduction proof (for non-speculative reductions) shows the following relationships between reductions and Q: Let $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$ such that Q t x σ and $\sigma \vdash \mathcal{H}\surd$ and $\mathcal{H} \vdash \overline{\alpha}\surd_{\text{ns}}$.

(K5) Type information grows monotonically under non-speculative reductions: Whenever *typeof-addr* σ $a = \lfloor T \rfloor$, then also *typeof-addr* σ' $a = \lfloor T \rfloor$.
(K6) Non-speculative reductions preserve Q: Q t x' σ' holds. Moreover, Q t'' x'' σ' holds for all spawned threads in $\overline{\alpha}$, i.e., *Spawn* t'' $x'' \in \overline{\alpha}$, and whenever Q t'' x'' σ, then Q t'' x'' σ', too.
(K7) Non-speculative reductions preserve conformance: $\sigma' \vdash$ *uhrs* $\mathcal{H}$ $\overline{\alpha}\surd$ holds.

Property K5 formalises that allocated objects never change their class. The other two correspond to the standard subject reduction theorems: K6 ensures that the subject reduction invariant Q holds for all threads in the successor state if all threads satisfy Q in the original state. K7 represents preservation of heap conformance in a classical type-safety proof. $\sigma' \vdash$ *uhrs* $\mathcal{H}$ $\overline{\alpha}\surd$ checks that every value that a write in $\overline{\alpha}$ writes conforms to the type of the respective location.

The novel thing is to restrict reductions to non-speculative ones. Remember that at all layers below the JMM, nothing constrains the values that are being read from shared locations. Therefore, I must restrict my attention to reductions that contain no junk like type-incorrect reads. For example, consider `int r = new A().f;` where `class A { int f; }`. When the read of `f` executes, there are reductions for every possible value including `0` and `null`. While `0` is type correct, reading `null` would result in a thread-local state that stores the null reference in a local variable of primitive type `int`. This would immediately break the type safety invariant. I could restrict reads to type-correct values, but this causes problems when type information is dynamic (cf. §2.1.1). Fortunately, I am still proving the DRF guarantee and therefore, I only have to deal with (prefixes of) SC’ executions. Non-speculative ones generalise them such that the general type-safety proof (§5.2) can reuse this development.

[21] The attentive reader may have noticed that the proof of Lem. 3.5 requires A3b only in the direction from right to left. This proof uses the other direction.

Moreover, I need to make two further assumptions about the single-threaded semantics. In contrast to K5 to K7, they constrain all reductions from states satisfying Q, not only non-speculative ones. This is fine as they only restrict those parts that the single-threaded semantics can control themselves, i.e., they do not depend on which values are read. Let $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$ such that $Q\ t\ x\ \sigma$.

(K8) Reductions of conforming states read only type-correct locations. Whenever *Read* (a, l) _ $\in \overline{\alpha}$, then there is a T such that *typeof-addr* $\sigma = \lfloor T \rfloor$ and $l \in$ *memb* T.
(K9) Allocation events record the correct type, i.e., whenever *Alloc* $a\ T \in \overline{\alpha}$, then *typeof-addr* $\sigma'\ a = \lfloor T \rfloor$.

LEMMA 3.16 (ASSUMPTIONS K5 TO K9). *The single-threaded semantics for source code and bytecode satisfy assumptions K5 to K9.*

PROOF. As discussed above, K5 to K7 are reformulations of a standard subject reduction theorem, and so are their proofs. K8 and K9 are by induction on the semantics or case analysis of the executed instruction. The provision that Q holds ensures that no undefined behaviour occurs. □

3.4.3. Initialisations Precede Reads. Finally, I am ready to discharge assumption D3” using K1 to K9 (Lem. 3.17; *snd* $(t, \alpha) = \alpha$ is the second projection for pairs). Hence, initialisations precede reads in non-speculative prefixes of traces and Thm. 3.7 and Cor. 3.8 are applicable.

LEMMA 3.17 (ASSUMPTION D3”). *Let Q hold for the bootstrapping thread in start-state, and kas start-state $\subseteq$ allocated start-σ, and start-$\sigma \vdash$ start-$\mathcal{H}\surd$. If $\xi \in \mathcal{E}$, $\xi_{[\alpha]} =$ Read $(a, l)\ v$ with $\alpha < |\xi|$, and $\lambda_.\ \emptyset \vdash$ take α (map snd $\xi)\surd_{\mathrm{ns}}$, then there is a $\beta < \alpha$ such that $\xi_{[\beta]}$ initialises (a, l).*

PROOF. Since $\xi \in \mathcal{E}$, there is $\overline{\xi}$ such that *start-state* $\Downarrow \overline{\xi}$ and $\xi =$ *start-events* ++ *concat* $\overline{\xi}$. Since $i < |\xi|$, I can split $\overline{\xi}$ such that $\overline{\xi} = \overline{\xi}'$ ++ *events* $(t, \overline{\alpha})$ ++ $\overline{\xi}''$, and *Read* $(a, l)\ v \in \overline{\alpha}$, and $|\overline{\xi}'| < i$. By definition of $\Downarrow$, $\overline{\xi}'$ is of the form *map events* $\overline{t\overline{\alpha}}$ for some $\overline{t\overline{\alpha}}$ such that there are states s and s' with *start-state* $-\overline{t\overline{\alpha}}\rightarrow^* s$, and $s -(t, \overline{\alpha})\rightarrow s'$. Since non-speculative prefixes of executions preserve conformance (K6 and K7), (a, l) is type correct in *shr* s (K8), i.e., *typeof-addr* (*shr* s) $(a, l) = \lfloor T \rfloor$ and $l \in$ *memb* T for some T. It suffices to show that there is an event *Alloc* $a\ T$ in *start-events* or $\overline{\xi}'$. This event initialises (a, l), since $l \in$ *memb* T.

By Lem. 3.15, the address a has been allocated before, i.e., either in *start-events* or in $\overline{\xi}'$. If *Alloc* $a\ T' \in$ *start-events* for some T', then *typeof-addr start-σ* $a = \lfloor T' \rfloor$ by construction of *start-events*, and therefore *typeof-addr* (*shr* s) $a = \lfloor T' \rfloor$ (K5 with induction on *start-state* $-\overline{t\overline{\alpha}}\rightarrow^* s$, and K6 and K7 for preserving the conformance invariant), i.e., $T = T'$. Otherwise, $\overline{t\overline{\alpha}} = \overline{t\overline{\alpha}}^*$ ++ $(t^*, \overline{\alpha}^*) \cdot \overline{t\overline{\alpha}}^{**}$ for some $\overline{t\overline{\alpha}}^*$, t^*, $\overline{\alpha}^*$, and $\overline{t\overline{\alpha}}^{**}$ such that *Alloc* $a\ T' \in \overline{\alpha}^*$ for some T'. Hence, there are states s^* and s^{**} such that *start-state* $-\overline{t\overline{\alpha}^*}\rightarrow^* s^*$, $s^* -(t^*, \overline{\alpha}^*)\rightarrow s^{**}$, and $s^{**} -\overline{t\overline{\alpha}}^{**}\rightarrow^* s$. By K9, I obtain *typeof-addr* (*shr* s^{**}) $= \lfloor T' \rfloor$ and therefore *typeof-addr* (*shr* s) $a = \lfloor T' \rfloor$ (K5 and induction as before). Hence $T = T'$. □

LEMMA 3.18 (START STATE). *The start state satisfies the assumptions of Lem. 3.17, i.e., Q holds for the bootstrapping thread in start-state, and kas start-state $\subseteq$ allocated start-σ, and start-$\sigma \vdash$ start-$\mathcal{H}\surd$. Further, start-events contains no read events.* □

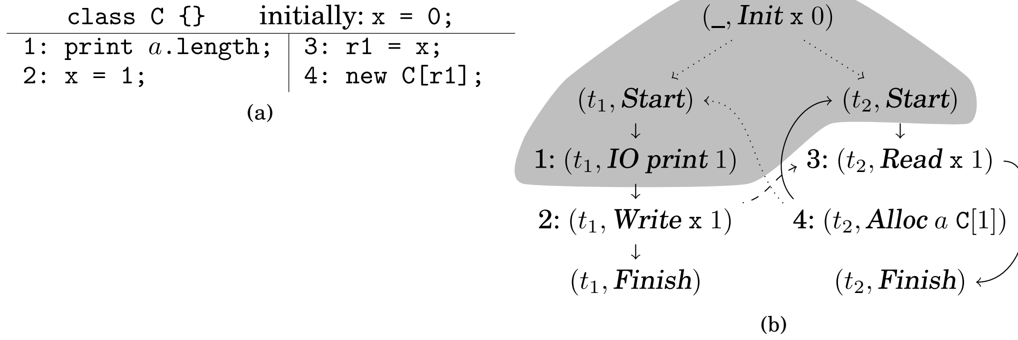


Fig. 25: An ill-formed program (a) and its execution (b) with a sequentially consistent prefix (grey area) followed by a read (l. 3) that cannot be cut, updated, and completed sequentially consistently.

3.5. Sequentially Consistent Completions

Assumption D1 still remains to be shown, i.e., sequentially consistent prefixes of well-formed executions can be completed sequentially consistently. Although this sounds trivial, it is formally not obvious, because allocations complicate things again. The program in Fig. 25a demonstrates that ill-formed programs can have sequentially consistent prefixes of executions which cannot be cut, updated, and completed sequentially consistently. Note that the program is ill-formed only because it literally contains the address a. However, such a program could well occur as an intermediate state while executing a valid Java program.

In the execution in Fig. 25b, the read in l. 3 sees the write from l. 2, but the most recent write would be the initialisation of location x. Suppose that l. 3 is scheduled after l. 1, but before l. 2, as the grey area indicates. Suppose further that a is also the address that the array allocation in l. 4 returns. Then, the prefix up to l. 1 is sequentially consistent, but has no sequentially consistent completion when l. 3 executes next. If l. 3 is updated to read the initial value 0, then l. 4 allocates an array of length 0 at address a, but l. 1 has already output a's array length as 1. This violates the JLS that array lengths are always correct [Gosling et al. 2005, §17.4.5]. Note that this example depends on the implemenation of the shared state: The issue arises only for dynamic type information (§2.1.1), because static types for addresses (§2.1.2) uses different addresses for arrays of different lengths, i.e., l. 4 cannot return the same address a for both allocations.

In this example, the problem is that t_1 literally contains the address a that the allocation of the other thread t_2 returns. This violates that a thread only knows an address if it has allocated it itself or it has read it from memory, which I have formalised in K1 and K2 and the condition *kas start-state* $\subseteq$ *allocated start-*σ (see Lem. 3.18 above).

I construct a *sequentially consistent completion* *scc* s H that starts with a multithreaded state s and a snapshot heap H. I define *scc* by corecursion as follows:

$$\begin{aligned} \textit{scc}\ s\ H = (\ & \textbf{\textit{if}}\ \exists t\ \overline{\alpha}\ s'.\ s -(t,\overline{\alpha})\rightarrow s' \\ & \textbf{\textit{then let}}\ (t,\overline{\alpha},s') = \varepsilon(t,\overline{\alpha},s').\ s -(t,\overline{\alpha})\rightarrow s' \wedge H \vdash \overline{\alpha}\surd_{\mathrm{sc}} \\ & \quad\ \textbf{\textit{in}}\ (t,\overline{\alpha}) \cdot \textit{scc}\ s'\ (\textit{mrws}\ H\ \overline{\alpha}) \\ & \textbf{\textit{else}}\ [\,]) \end{aligned} \tag{3}$$

Hilbert's ε-operator (indefinite description operator) $\varepsilon x.\ P\ x$ denotes one (fixed, but underspecified) x such that $P\ x$ holds, provided P is satisfiable at all. Otherwise, it is

unspecified. Hence, in order to prove anything about *scc* s H, I must make sure that the predicate to the ε-operator is satisfiable for all reachable configurations. Thus, I presume the following:

(D4) The interleaving semantics satisfies the cut-and-update property for the start state *start-state* and the start snapshot heap *start-H*, where *start-H* = *mrws empty* (*map snd start-events*).

The *cut-and-update property* (C&U) for s and H (denoted *C&U* s H) denotes the following. Let the state s' be reachable from s via an SC' event list, say $s -\overline{t\overline{\alpha}}\rightarrow^* s'$ such that $H \vdash$ *concat* (*map events'* $\overline{t\overline{\alpha}}$)$\surd_{\text{sc}}$, and let H' denote the updated snapshot heap *mrws* H (*concat* (*map events'* $\overline{t\overline{\alpha}}$)). Then, for every reduction $s' -(t', \overline{\alpha}')\rightarrow s''$ from s', there are $\overline{\alpha}''$, s''' such that

(a) $s' -(t', \overline{\alpha}'')\rightarrow s'''$,
(b) $H' \vdash \overline{\alpha}''\surd_{\text{sc}}$, and
(c) $H' \vdash \overline{\alpha}' \approx \overline{\alpha}''$ (to be explained in a moment).

Conditions (a) and (b) predicate that all reachable, non-stuck states can reduce with events $\overline{\alpha}''$ that are SC' w.r.t. the current snapshot heap H'; they suffice to prove that *scc* does compute an SC' interleaving (Lem. 3.20). In condition (c), $H' \vdash \overline{\alpha}' \approx \overline{\alpha}''$ denotes that two event lists $\overline{\alpha}'$ and $\overline{\alpha}''$ consist of the same events up to the first SC'-inconsistent read in $\overline{\alpha}'$ (if any) and $\overline{\alpha}''$ continues with a read from the same location. With condition (c), given a trace that is SC' up to a read α, I can cut the interleaving after α, replace α with a read of the most recent value, and continue the interleaving SC'.

LEMMA 3.19 (PRESERVATION OF C&U).
If C&U s H, $s -(t, \overline{\alpha})\rightarrow s'$, and $H \vdash \overline{\alpha}\surd_{\text{sc}}$, then C&U s' (mrws H $\overline{\alpha}$).

PROOF. This holds by definition of C&U because every state that is reachable via SC' reductions from s' is also reachable via SC' reductions from s by prefixing the SC' reduction $s -(t, \overline{\alpha})\rightarrow s'$. □

Under assumption D4, *scc* computes an SC' execution (Lem. 3.20). By the equivalence of SC and SC' (Thm. 3.7), I then discharge the main assumption of the DRF proof (Thm. 3.21).

LEMMA 3.20.
If C&U s H, then $s \downarrow$ scc s H and $H \vdash$ concat (map events' (scc s H))$\surd_{\text{sc}}$.

THEOREM 3.21 (SC COMPLETION). *Let $\xi \in \mathcal{E}$, $(\xi, ws)\surd$, (ξ, ws) be SC up to a read event $(t,$ **Read** $(a,l)\ v)$, say $\xi = \xi_1 +\!\!+ (t,$ **Read** $(a,l)\ v) \cdot \xi_2$ with $ws\ \alpha$ being the most recent write for all reads $\alpha \in \mathcal{R}_{\xi_1}$. Then, there are ξ_3, v', and ws' such that $\xi^* := \xi_1 +\!\!+ (t,$ **Read** $(a,l)\ v') \cdot \xi_3 \in \mathcal{E}$, and $(\xi^*, ws')\surd$, and (ξ^*, ws') is SC.*

PROOF OF LEM. 3.20. I show $s \downarrow$ *scc* s H by coinduction with *C&U* s H as the coinduction invariant. If s is stuck, then *scc* s $H = []$ and I am done by STOP. Otherwise, conditions (a) and (b) of C&U ensure that the predicate to Hilbert's choice in (3) is satisfiable. Hence, it does pick an SC' reduction step $s -(t, \overline{\alpha})\rightarrow s'$ and updates H to $H' :=$ *mrws* H $\overline{\alpha}$. Note how this mimics STEP. Since SC' reductions preserve *C&U* (Lem. 3.19), and the reduction is SC', *C&U* s' H' holds, too. This concludes the coinductive step.

For $H \vdash$ *concat* (*map events'* (*scc* s H))$\surd_{\text{sc}}$, the standard coinduction rule is too weak because *concat* is unproductive for any number of consecutive reductions without events. Hence, I use a coinduction rule for $_ \vdash _\surd_{\text{sc}}$, which allows to defer the next step if one decreases in a well-founded relation, as described in §1.2.3. Taking as measure

the length of the maximal prefix of *scc* s H for whose elements *events'* returns the empty lists, I show $H \vdash$ *concat* (*map events'* (*scc* s H))$\surd_{\text{sc}}$ with the invariant ***C&U*** s H like above. □

PROOF OF THM. 3.21. Construct ξ_3 as follows: First, identify the reduction $s -(t, \overline{\alpha})\rightarrow s'$ that generates $(t,$ ***Read*** $(a,l)\ v)$. Let ξ_1' be the prefix of ξ up to *events* $(t, \overline{\alpha})$ exclusively, which is also a prefix of ξ_1. Since all reads in ξ_1 (and thus ξ_1') see the most recent write, ξ_1' is SC' by Thm. 3.7. Since C&U holds for the start state and the start snapshot heap and SC' reductions preserve C&U (Lem. 3.19), C&U holds for s and H_1 = ***mrws empty*** ξ_1', too. Hence, by C&U, there are $\overline{\alpha}'$ and s'' such that $s -(t, \overline{\alpha}')\rightarrow s''$, $H_1 \vdash \overline{\alpha}' \surd_{\text{sc}}$, and $H_1 \vdash \overline{\alpha} \approx \overline{\alpha}'$. From the latter, I know that $\overline{\alpha}$ and $\overline{\alpha}'$ are the same up to the read *Read* $(a,l)\ v$ in $\overline{\alpha}$ (exclusively), which is ***Read*** $(a,l)\ v'$ in $\overline{\alpha}'$ for the SC'-correct value v'. Now, choose ξ_3 to be the rest of $\overline{\alpha}'$ followed by ***concat*** (***map events*** (***scc*** s (***mrws*** $H_1\ \overline{\alpha}'$))).

With Lem. 3.20, I get that ξ^* is SC' and $\xi^* \in \mathcal{E}$. Thm. 3.7 yields the required ws'. □

COROLLARY 3.22.
Every program has a well-formed, sequentially consistent execution.

PROOF. Set ξ = ***start-events*** ++ ***concat*** (***map events*** (***scc*** s H)). Then, $\xi \in \mathcal{E}$ and ***empty*** $\vdash \xi \surd_{\text{sc}}$ by Lem. 3.20 and definition of $\mathcal{E}$ and Lem. 3.18. By Thm. 3.7, there is a ws such that $(\xi, ws)\surd$ and (ξ, ws) is SC. □

3.6. Cut and Update

For the DRF guarantee, it remains to show that the interleaving semantics satisfies C&U for the start state (assumption D4). Similar to initialisations preceding reads, I generalise C&U to non-speculative prefixes and reading any previously written value, not only the most recent one. This way, I can reuse the proof for consistency in §4.

Formally, the sequential semantics has the *generalised cut-and-update property* (*gC&U*) for a state s and heap record $\mathcal{H}$ iff for all states s' reachable from s in the interleaving semantics with non-speculative events $\overline{\alpha}$ and any reduction $s -(t, \overline{\alpha}')\rightarrow s'$, whenever $\overline{\alpha}'_{[i]}$ = ***Read*** $(a,l)\ v$ for some $i < |\overline{\alpha}'|$ such that ***uhrs*** $\mathcal{H}\ \overline{\alpha} \vdash$ ***take*** $i\ \overline{\alpha}' \surd_{\text{ns}}$, then for any value $v' \in$ ***uhrs*** $\mathcal{H}$ $(\overline{\alpha}$ ++ ***take*** $i\ \overline{\alpha}')$, there is a reduction $s -(t, \overline{\alpha}'')\rightarrow s''$ such that $i < |\overline{\alpha}''| \leq |\overline{\alpha}'|$ and $\overline{\alpha}''_{[i]}$ = ***Read*** $(a,l)\ v'$ and ***take*** $i\ \overline{\alpha}'$ = ***take*** $i\ \overline{\alpha}''$.

Intuitively, gC&U allows to cut a trace at any read event in its non-speculative prefix and replace it with a read from the same location that reads any value which has previously been written to that location. This might seem overly complicated, but I must ensure that the semantics stays within defined behaviour. Speculative event lists such as those leading to the well-formed execution in Fig. 5 may read values out of thin air. Although the JMM legality dismisses such traces later, I must deal with them on the level of interleaving semantics, because being illegal is hard to characterise at this level. Since non-speculative executions preserve conformance (assumptions K6 and K7) and generalise sequential consistency (Lem. 3.10), they provide an adequate universe of trace prefixes. In §4 and §5, I will show that non-speculative events also contain further prefixes of interest. In particular, I restrict the updated values to previously written ones to maintain non-speculativity.

LEMMA 3.23 (GC&U IMPLIES C&U). *If the interleaving semantics satisfies gC&U for start-state and start-$\mathcal{H}$ and the bootstrapping thread initially satisfies Q, then it also satisfies C&U for start-state and start-H.*

PROOF. Suppose ***start-state*** $-\overline{t\overline{\alpha}}\rightarrow^* s$ and $s -(t, \overline{\alpha}')\rightarrow s'$ such that ***start-H*** $\vdash \overline{\alpha} \surd_{\text{sc}}$ where $\overline{\alpha}$ abbreviates ***concat*** (***map events'*** $\overline{t\overline{\alpha}}$). Let H = ***mrws start-H*** $\overline{\alpha}$. I must show that there are $\overline{\alpha}''$ and s'' such that $s -(t, \overline{\alpha}'')\rightarrow s''$, and $H \vdash \overline{\alpha}'' \surd_{\text{sc}}$, and $H \vdash \overline{\alpha}' \approx \overline{\alpha}''$.

By Lem. 3.10, *start-*$\mathcal{H} \vdash \overline{\alpha}\surd_{\mathrm{ns}}$, i.e., gC&U allows to cut and update all reductions from state s. Construct $\overline{\alpha}''$ iteratively as follows: Start with $\overline{\alpha}'' = \overline{\alpha}'$ and consider the first event in $\overline{\alpha}''$. If it is an event reading not the most recently written value (according to the snapshot heap H), change the reduction to the most recently written value using gC&U, then continue with the new reduction for the next event. Otherwise, update the snapshot heap H for the event and consider the next event. This process terminates after at most $|\overline{\alpha}'|$ iterations because gC&U bounds the length of the replacement events $\overline{\alpha}''$ to that length. The reduction thus obtained serves as witness.

The key step in the iteration is to show that H stores a most recently written value at all. I show similar to Lem. 3.17 that *start-state*$-\overline{t\overline{\alpha}}\rightarrow^* s$ initialises the location. This ensures that H does store some value v for the location and $\mathcal{H} =$ *uhrs start-*$\mathcal{H}\ \overline{\alpha}$ has recorded v, too, as H fits to $\mathcal{H}$. Hence, gC&U ensures that I can cut and update the reduction as described. □

Now, it remains to show that both source code and bytecode satisfy gC&U, i.e.,

(D4') The interleaving semantics satisfies gC&U for *start-state* and *start-*$\mathcal{H}$.

Although D4' is tedious to prove for the layers 2 to 5, these proofs are not particularly interesting. I avoid having to reason about which values have previously been written by generalising it further to arbitrary type-conforming values. Since gC&U involves only non-speculative prefixes of executions, assumptions K6 and K7 ensure that these preserve conformance. In particular, all written values conform, too. Therefore, any value that gC&U requires to be read conforms to its type.

Thus, I am finally able to conclude that the DRF guarantee holds for source code and bytecode.

THEOREM 3.24. *The DRF guarantee holds for source code and bytecode. If the program is correctly synchronised, then every legal execution is SC.* □

4. CONSISTENCY

In the previous section, I have shown that the JMM allows solely sequentially consistent behaviour for correctly synchronised programs. The DRF guarantee shows that the JMM is strong enough to disallow certain undesired behaviours. Conversely, consistency requires that the JMM be not too strong: the JMM does allow some legal behaviour for *every* valid Java program. Now, I prove consistency, even for incorrectly synchronised programs. In particular, I show that any sequentially consistent execution is legal.[22] This is not trivial, because in programs with data races, the most recent write for a read need not happen before it. Hence, these data races must be justified.

THEOREM 4.1 (CONSISTENCY). *Every source code and bytecode program has a sequentially consistent execution. Every sequentially consistent execution is legal.*

Combining consistency with the DRF guarantee, I obtain the following:

— For correctly synchronised programs, the JMM is exactly sequential consistency.
— For programs with data races, the JMM is strictly weaker than sequential consistency.

Like in the previous section, I have identified assumptions on the interfaces between the different layers of the semantics such that I can conduct the proofs as abstractly as possible. In fact, this section only relies on the properties of the single-threaded

[22]Note that this property need not extend to concrete Java implementations, because it is not a safety property. For example, one could imagine a weird, but correct processor and VM that cannot produce an SC result for some program with data races.

semantics from the previous section. All theorems are on the layer of the interleaving semantics or on higher ones. Like in §3, I start at the JMM layer with assumptions about traces and then discharge these assumptions in the layers below.

At the JMM layer, the assumptions are now:

(C1) For every sequentially consistent prefix of a well-formed execution (ξ, ws) with $\xi \in \mathcal{E}$, there is a trace $\xi' \in \mathcal{E}$ with the same prefix and a write seen function ws' such that
  (a) $(\xi', ws')\surd$,
  (b) for all read events $\alpha \in \mathcal{A}_{\xi'}$, if α is in the prefix, then $ws'\ \alpha = ws\ \alpha$ else $ws'\ \alpha \leq^{\xi'}_{\text{hb}} \alpha$, and
  (c) if ξ continues with an event β directly after the prefix, ξ' continues with the same β, except that if β is a read, it may read a different value.

(C2) If a well-formed execution has an SC prefix $\overline{\alpha}$ followed by a read from (a, l), $\overline{\alpha}$ initialises (a, l).

(D2) Every execution $\xi \in \mathcal{E}$ initialises every location at most once.

Assumption C1 expresses that I can cut any execution after an SC prefix and continue such that every read in the continuation sees a write that happens before the read. I dub this happens-before consistent completion in analogy to sequentially consistent completions for the DRF guarante (D1). The second assumption C2 is similar to D3' with SC' replaced by SC. Assumptions C2 and D2 ensure that for well-formed executions with an SC prefix followed by a read α to location (a, l),

— a most recent write α' exists for α with $\alpha' < \alpha$, and
— if a write α^* to (a, l) happens before α, then $\alpha^* < \alpha$, too.

THEOREM 4.2. *Under assumptions C1, C2, and D2, every SC execution is legal.*

PROOF. Let $\xi \in \mathcal{E}$ such that $(\xi, ws)\surd$ and (ξ, ws) is SC. I must justify (ξ, ws) by a justifying sequence $(\xi_i, ws_i, C_i, \varphi_i)_i$. For $i \leq |\xi|$, choose some (ξ_i, ws_i) with the following properties:

— $\xi_i \in \mathcal{E}$
— $(\xi_i, ws_i)\surd$
— $i \leq |\xi_i|$
— ***take*** $(i - 1)\ \xi =$ ***take*** $(i - 1)\ \xi_i$
— Suppose $i > 0$. If $\xi_{[i-1]} =$ ***Read*** $(a, l)\ v$, then $\xi_{i[i-1]} =$ ***Read*** $(a, l)\ v'$ for some v', else $\xi_{[i-1]} = \xi_{i[i-1]}$.
— For all read events $\alpha \in \mathcal{A}_{\xi_i}$, if $\alpha < i - 1$ then $ws_i\ \alpha = ws\ \alpha$, else $ws_i\ \alpha \leq^{\xi_i}_{\text{hb}} \alpha$.

Assumption C1 ensures that such (ξ_i, ws_i) exist. Set $C_i = \{\, \alpha \mid \alpha < i \,\}$, i.e., (ξ_i, ws_i) commits the first i events.

For $i > |\xi|$, set $\xi_i = E$ and $ws_i = ws$ and $C_i = \mathcal{A}_\xi$. Then, the sequence $(\xi_i, ws_i, C_i, \varphi_i)_i$ justifies (ξ, ws) where all renamings φ_i are the identity.

To illustrate how $J = (\xi_i, ws_i, C_i, \varphi_i)_i$ justifies reads which see writes that do not happen before, consider the following program P6 where the write in l. 1 races with the read in l. 2. It differs from Fig. 22 in x not being volatile.

initially: `x = 0;`	
`1: x = 1;`	`2: r = x;`

(P6)

Fig. 26 shows the executions for the two traces where thread t_1 on the left executes before the one on the right (t_2). I wish to justify the SC execution shown in Fig. 26a.

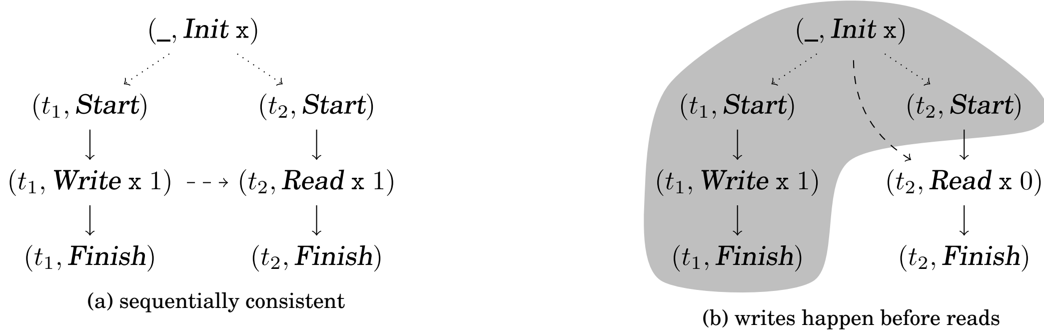


Fig. 26: Two executions of the traces $[(_, \mathit{Init}\ \mathrm{x}), (t_1, \mathit{Start}), (t_1, \mathit{Write}\ \mathrm{x}\ 1), (t_1, \mathit{Finish}), (t_2, \mathit{Start}), (t_2, \mathit{Read}\ \mathrm{x}\ v), (t_2, \mathit{Finish})]$ for (a) $v = 1$ and (b) $v = 0$ of the program P6

Suppose that we are about to commit the read event, i.e., $i = 5$. Fig. 26b shows (ξ_4, ws_4) where the grey area contains all committed events.

The JMM justification rules allow the write-seen function (dashed arrows) to change only for reads that the previous justifying execution has committed for the first time. Since (ξ_5, ws_5) commits the read, it must still see the allocation as there are no other writes that happen before the read. Thus, Fig. 26b also shows (ξ_5, ws_5). In the next step, (ξ_6, ws_6) may change the read such that it sees the write, i.e., Fig. 26a shows (ξ_6, ws_6). At the same time, it also commits the last event (t_2, Finish).

This offset explains why the specification of (ξ_i, ws_i) mostly refers to $i - 1$. However, one cannot shift the whole sequence by one because the JMM requires that (ξ_0, ws_0) has not yet committed any events.

The proof that $(\xi_i, ws_i)_i$ justifies (ξ, ws) is tedious and largely uninteresting, except for the case when the $i-1$-th event in ξ reads from a write that does not happen before it. In that case, (ξ_i, ws_i) changes the write from $ws_{i-1}\ (i-1)$ to $ws\ (i-1)$. Legality condition L9 require that (ξ_{i-1}, ws_{i-1}) has already commited both of them, i.e., $ws_{i-1}\ (i-1) < i-1$ and $ws\ (i-1) < i-1$. As noted above, assumptions C2 and D2 ensure this, because $ws\ (i-1)$ is the most recent write for and $ws_{i-1}\ (i-1)$ happens before $i-1$. □

COROLLARY 4.3. *Under assumptions D2, D3', D4, C1, and C2, every program has a (weakly) legal execution.*

PROOF. By Cor. 3.22, it has a well-formed SC execution. By Thm. 4.2, this execution is legal. Legality implies weak legality. □

Next, I show that source code and bytecode satisfy assumptions C1 and C2. Note that the latter is equivalent to D3' by Thm. 3.7, i.e., Lem. 3.17 discharges it.

Assumption C1 is structured similarly to D1. Thus, I construct a witness execution by corecursion similar to *scc* in (3), but choose $\overline{\alpha}$ such that reads in $\overline{\alpha}$ see writes that happen before them. Since assumptions C2 and D2 ensure that such writes precede the reads in the execution, the prefix up to the read is non-speculative and thus gC&U ensures that such a witness exists. Here, the crucial step is to show that such a write exists. Note that the initialisation exists by assumption D3" and happens before the read by Lem. 3.3. Then, the $\leq_{\mathrm{eo}}$-maximal write to the location that happens before the read serves as witness. Since the proof structure is similar to sequentially consistent completions (Lems. 3.19 and 3.20, Thm. 3.21), I omit the details.

5. TYPE SAFETY

Type safety has been one of the motivations for the JMM defining semantics for data races, but to date, no proof of type safety for the JMM has been published. Here, I prove that type safety holds for *all* Java programs, even in the presence of data races, subject to a few side conditions (§5.2). I discuss the type safety statement (§5.3) and its relation to the out-of-thin-air guarantee (§5.4).

5.1. Type-Unsafe Executions

Before I delve into the details, I present the difficulties and challenges in proving type safety for the JMM. Traditionally, subject reduction is used to show type safety, i.e., one identifies an invariant Q that all reductions preserve [Wright and Felleisen 1994]. Typically, Q includes type-correctness of the statement and conformance of the store. For the DRF guarantee, I have already shown preservation for non-speculative reductions (assumption K6), i.e., for programs without data races. Yet, speculation is a core feature of the JMM, and in general, preservation does not hold.

In Fig. 5, e.g., both reads speculate to read the value 1, which is still type-correct. However, they could have speculated *null* as well, and storing *null* in the local variable `r1` of type `int` violates conformance, i.e., breaks preservation. Legality dismisses both executions, but this only happens in layer 7; the subject reduction proof, however, operates below – mainly in layer 3 – and, therefore, this has to be addressed.

Of course, one can restrict traces to read only type-conforming values, but this would be cheating: This renders the semantics type-safe by construction and therefore cannot show that the legality constraints do ban out-of-thin-air values. Moreover, as discussed in §2.1.1, this restriction can exclude some legal behaviours.

Moreover, I have claimed that dynamic type information (§2.1.1) leads to unsoundness. Figure 27 shows the justification for the type-unsafe execution ξ that I described in §2.1.1 for Fig. 8. It uses the notation from Fig. 16b, but omits the events *Start* and *Finish* as they are irrelevant. The trick is to justify the address of the `D` object allocated in l. 7 as an out-of-thin-air value for the data races on `x` and `y`. These races have the same pattern as in Fig. 5, where the JMM is sufficiently strong to disallow out-of-thin-air values. However, in Fig. 8, this cycle occurs only if the then branch (l. 5) executes. The justification in Fig. 27 first executes the else branch (l. 7) until both data races on `x` and `y` are committed (ξ_0 to ξ_3). Then, ξ_4 switches the branches and the address a keeps being passed between the two data races as an out-of-thin-air value. The then branch could then do almost anything – in the example, it allocates a `C` object. Although `r1`, `r2`, `x`, and `y` already contain the address a, it has not yet been allocated in ξ_4 and is thus fresh.[23] Consequently, the allocation strategy of §2.1.1 uses a for the allocation in l. 5, which is a `C` object. Hence, the locations `x` and `y` of type `D` now refer to a `C` object, which is type-unsafe.

This problem is not specific to the allocation strategy of §2.1.1. Similar examples can be conceived for other strategies that allow to allocate objects of different types at the same address. Static type information (§2.1.2) circumvents this issue, because `C` and `D` objects have distinct address spaces.

5.2. Proof of Type Safety

In the following, I prove type safety for the JMM. I assume *typeof-addr* and derived notions like conformance do not depend on the shared state. Static type information as in §2.1.2 meets this assumpion. Consequently, the type of an address, a field, cell, or

[23]Note that allocation sometimes has to return an address that is already stored in some location or variable, e.g., in Fig. 18.

ξ_0: Ix ⇢ 1 8 Iy ⇢ 3 2 Ib ⇢ 4 9 7

ξ_1: Ix ⇢ 1 8 Iy ⇢ 3 2 Ib ⇢ 4 9 7

ξ_2: Ix 1 ⇠ 8 Iy ⇢ 3 2 Ib ⇢ 4 9 7

ξ_3: Ix 1 ⇠ 8 Iy 3 ⇠ 2 Ib ⇢ 4 9 7

$\xi = \xi_4$: Ix 1 ⇠ 8 Iy 3 ⇠ 2 Ib 4 ⇠ 9 5

Fig. 27: Justification for a type-unsafe execution of Fig. 8

value makes sense without the context of a particular execution. In particular, I now omit the shared state σ for conformance, e.g., $\vdash v :\leq T$ instead of $\sigma \vdash v :\leq T$.

A read or write event to location (a, l) of value v is *type-correct* iff a has some type T such that l is a member of T and v conforms to (a, l)'s type. Type safety for the JMM means that in any legal execution (ξ, ws), all read events read are type-correct. Thus, standard type safety proofs for the language extend to the JMM.

THEOREM 5.1 (JMM TYPE SAFETY). *Let (ξ, ws) be a (weakly) legal execution of a source code or bytecode program, and α be any read event in ξ. Then, α is type-correct.*

Fortunately, the machinery for the DRF guarantee almost suffices to prove the assumptions of the JMM type safety proof about the single-threaded semantics. This might surprise at first, because in §3, I have paid close attention that I only talk about non-speculative lists of events, but Thm. 5.1 also holds for speculative executions. The key is that I can use $\mathcal{H} \vdash \overline{\alpha}\surd_{\text{ns}}$ to also characterise event lists of type-safe executions. Instead of *start-$\mathcal{H}$*, I use the largest conforming heap record $\widehat{\mathcal{H}}$. If the address a has some type T such that $l \in$ *memb* T, then $\widehat{\mathcal{H}}\ (a, l)$ contains all values that conform to (a, l)'s type. Otherwise, $\widehat{\mathcal{H}}\ (a, l) = \emptyset$. Clearly, $\vdash \widehat{\mathcal{H}}\surd$. Then, $\widehat{\mathcal{H}} \vdash \overline{\alpha}\surd_{\text{ns}}$ expresses that any read α in $\overline{\alpha}$ is type correct provided that all writes prior to α are type correct, too. Conversely, if all reads in $\overline{\alpha}$ are type correct, then $\widehat{\mathcal{H}} \vdash \overline{\alpha}\surd_{\text{ns}}$, because $\widehat{\mathcal{H}}$ already contains all the values that are read.

The assumptions for type safety are K6, K8, and the following two: T1 strengthens K7 to prefixes; and T2 generalises K9 by dropping conformance $Q\ t\ x\ \sigma$ and non-speculativity.

(T1) Non-speculative prefixes preserve conformance:
Let $Q\ t\ x\ \sigma$, and $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$, and $\vdash \mathcal{H}\surd$. For all i, if $\mathcal{H} \vdash$ *take* $i\ \overline{\alpha}\surd_{\text{ns}}$, then $\vdash$ ***uhrs*** $\mathcal{H}$ (*take* $i\ \overline{\alpha}$)$\surd$.

(T2) If $t \vdash (x, \sigma) -\overline{\alpha}\rightarrow (x', \sigma')$, and *Alloc* $a\ T \in \overline{\alpha}$, and *typeof-addr* a is defined, then *typeof-addr* $a = \lfloor T \rfloor$.

Since these assumptions are closely related to K7 and K9, discharging them is also similar. Thus:

LEMMA 5.2.
The single-threaded semantics for source code and bytecode satisfy T1 and T2. □

Before I prove type safety, I first show two lemmas about when writes and reads are type-correct. The actual proof of Thm. 5.1 then lifts these to weak justification sequences and legality.

LEMMA 5.3 (TYPE-CORRECT WRITES). *Suppose that K6 and T1 hold. Let* $\xi \in \mathcal{E}$ *and* $\alpha \in \mathcal{W}$ *such that* $\widehat{\mathcal{H}} \vdash$ *map snd* (*take* α ξ)$\surd_{\text{ns}}$. *Then,* α *is type-correct for all locations that it writes to. More precisely, for all* $(a, l) \in$ *locs* α, (a, l) *has a type* T *such that* $\vdash$ *vw* α $(a, l) :\leq T$.

PROOF. If α originates from *start-events*, it allocates one of the pre-allocated objects. By construction of *start-state*, the initialisation values conform to the locations' types. In particular, *uhrs* $\widehat{\mathcal{H}}$ *start-events* $= \widehat{\mathcal{H}}$. So, suppose that α is produced by some reduction $t \vdash (x, \sigma) -\overline{\alpha}'\rightarrow (x', \sigma')$ such that *start-state* $-\overline{t\overline{\alpha}}\rightarrow^*$ s, and s contains thread t with local state x and shared state σ, and $\overline{\alpha}$ = *start-events* ++ *concat* (*map events* $\overline{t\overline{\alpha}}$) ++ *events* $(t, \overline{\alpha}')$ ++ $\overline{\xi}$ for some rest $\overline{\xi}$. Let $\overline{\alpha}^* =$ *concat* (*map events'* $\overline{t\overline{\alpha}}$) and $\mathcal{H} =$ *uhrs* $\widehat{\mathcal{H}}$ $\overline{\alpha}^*$ and let i denote α's index in $\overline{\alpha}$. As *start-state* conforms (Lem. 3.18), $\vdash \widehat{\mathcal{H}}\surd$, and $\widehat{\mathcal{H}} \vdash \overline{\alpha}^* \surd_{\text{ns}}$, the assumptions K6 and T1 ensure that Q t x σ and $\vdash \mathcal{H}\surd$ hold (by induction on *start-state* $-\overline{t\overline{\alpha}}\rightarrow^*$ s). Applying T1 for t's reduction, I get that $\vdash$ *uhrs* $\mathcal{H}$ (*take* $(i + 1)$ $\overline{\alpha}'$)$\surd$, too. Moreover *vw* α $(a, l) \in$ *uhrs* $\mathcal{H}$ (*take* $(i + 1)$ $\overline{\alpha}'$) by construction of *uhrs*. By definition of heap record conformance, the claim follows. $\square$

LEMMA 5.4 (TYPE-CORRECT READS). *Suppose that K6, K8, T1, and T2 hold. Let* $\xi \in \mathcal{E}$, *and* $(\xi, ws)\surd$. *Further suppose that for any read* $\beta \in \mathcal{R}$, *if* ws $\beta \not\leq_{\text{hb}} \beta$, *then* β *is type correct. Then, all reads* $\alpha \in \mathcal{R}$ *are type correct.*

PROOF. By induction on α (which is a non-negative integer). If ws $\alpha \not\leq_{\text{hb}} \alpha$, I am done by assumption. So, suppose ws $\alpha \leq_{\text{hb}} \alpha$. Unfortunately, I cannot deduce ws $\alpha < \alpha$ like I did for consistency (§4), because C2 does not apply. In fact, Fig. 18 shows a counterexample with allocation.

Nevertheless, suppose for now that ws $\alpha < \alpha$; I will deal with $\alpha \leq ws$ α below. Then, the prefix $\xi_1 =$ *take* $(ws$ $\alpha)$ ξ of ξ up to ws α is non-speculative for $\widehat{\mathcal{H}}$, because by the inductive hypothesis, all reads $\alpha' \in \mathcal{R}$ with $\alpha' < \alpha$ (thus $\alpha' < ws$ α) are type-correct. Thus, Lem. 5.3 applies and ws α is type-correct, too – in particular for the location (a, l) that α reads from. Well-formedness W2 gives that α reads the same value that ws α writes, i.e., α is type-correct, too.

Now, consider the missing case $\alpha \leq ws$ α. Then, $\alpha < ws$ α, because reads are never writes. As $\leq_{\text{hb}}$ is consistent with the total execution order $\leq_{\text{eo}}$, ws $\alpha \leq_{\text{hb}} \alpha$ implies ws $\alpha \leq_{\text{eo}} \alpha$. By definition of $\leq_{\text{eo}}$ and $\preceq$, this is only possible if ws α is an allocation, say *Alloc* a' T. By W1, α reads from some location (a, l) with $(a, l) \in$ *locs* $(ws$ $\alpha)$, i.e., $a' = a$ and $l \in$ *memb* T. It suffices to show that *typeof-addr* $a = \lfloor T \rfloor$, because then, the location (a, l) also has a type, say T' and α reads the default value for T', which always conforms to T'. Yet, this is not obvious, because a's type and T in *Alloc* a T need not fit together. In particular, assumption K9 does not apply, because I cannot deduce that the thread's state satisfies Q when it produces ws α.

For example, consider the following scenario: α speculates a type-incorrect value, say `false` instead of `null`. A subsequent method call with `false` as receiver does not raise a null pointer exception as expected, but behaves undefinedly. And it is the undefined behaviour that generates the allocation that α sees; but as the behaviour is undefined, a's type need not be T. For example, suppose that a is drawn from the addresses for arrays of `Object`, but T is `bool[1]`, i.e., an array of primitive booleans. Hence, α is type-incorrect, but α itself causes the type-incorrect write that it sees. The legality constraints do not catch this causal cycle, because allocations happen before the start of the program.

Assumption T2 bans this scenario, as the following proof shows. Some reduction $s -(t, \overline{\alpha})\rightarrow s'$ generates the read α such that *start-state* $-\overline{t\overline{\alpha}}\rightarrow^*$ s, i.e., *start-events* ++

concat (*map events* $\overline{t\overline{\alpha}}$) is ξ's prefix before $\overline{\alpha}$. This prefix is non-speculative for $\widehat{\mathcal{H}}$, using the induction hypothesis analogously to ξ_1 in $ws\ \alpha < \alpha$ above. Hence, t's thread-local state in s satisfies Q and by assumption K8, a has some type, i.e., *typeof-addr* a is defined. Then, *typeof-addr* $a = \lfloor T \rfloor$ using T2, because some reduction generates $ws\ \alpha$.[24] □

The next lemma shows that if a read is committed in a (weak) justification sequence, then the write that is sees in the justified execution has already been committed. Aspinall and Ševčík [2007a] have already used – but not mentioned – this lemma for their proof of the DRF guarantee. It follows from L2, L9', and L11 by induction on i.

LEMMA 5.5. *Let* $(\xi_i, ws_i, C_i, \varphi_i)_i$ *weakly justify* (ξ, ws). *If* $\alpha \in \mathcal{R}_{\xi_i} \cap C_i$, *then* $ws\ (\varphi_i\ \alpha) \in \varphi_i\ `\ C_i$. □

Now, I am ready to prove type safety.

PROOF OF THM. 5.1. It suffices to show the statement for weak legality, because weak legality implies legality (§2.4). Let $J = (\xi_i, ws_i, C_i, \varphi_i)_i$ be the weak justification sequence for (ξ, ws).

First, I prove that for all $i \geq 1$, all reads $\alpha \in \mathcal{R}_{\xi_i}$ are type-correct,[25] by induction on i. As induction hypothesis, I assume that all reads in $\mathcal{R}_{\xi_{i-1}}$ are type-correct if $i \geq 2$. I show that all reads in $\alpha \in \mathcal{R}_{\xi_i}$ are type-correct, too. By Lem. 5.4, it suffices to consider only the case $ws\ \alpha \not\leq^{\xi_i}_{\mathrm{hb}} \alpha$. Suppose that α reads value v from location (a, l) in ξ_i. By L8, α must have already been committed in C_{i-1}, i.e., there is some $\alpha' \in C_{i-1}$ such that $\varphi_{i-1}\ \alpha' = \varphi_i\ \alpha$. Hence, α sees in ξ_i the same write as in ξ by L7, i.e., $ws\ (\varphi_i\ \alpha) = \varphi_i\ (ws_i\ \alpha)$. Moreover, as $\alpha \in C_i$, $ws_i\ \alpha \in C_i$, too (Lem. 5.5). By L6, $ws\ (\varphi_i\ \alpha)$ $(= \varphi_i\ (ws_i\ \alpha))$ writes the same as $ws_i\ \alpha$ to (a, l), namely v.

Note that $i \geq 2$, because $C_0 = \emptyset$ (L2) and $\alpha' \in C_{i-1}$. The induction hypothesis therefore applies for $i - 1$: all reads in ξ_{i-1} are type-correct. Thus, (every prefix of) ξ_{i-1} is non-speculative for $\widehat{\mathcal{H}}$, i.e., Lem. 5.3 applies and all writes in ξ_{i-1} are type-correct. Thus, it suffices to find a write in $\mathcal{W}_{\xi_{i-1}}$ that writes v to the location (a, l), as this ensures that α is type-correct, too.

Finding this write depends on whether α has already been committed in C_{i-2}. If so, there is some $\alpha'' \in C_{i-2}$ such that $\varphi_{i-2}\ \alpha'' = \varphi_i\ \alpha$. Hence, $ws\ (\varphi_{i-1}\ \alpha') = \varphi_{i-1}\ (ws_{i-1}\ \alpha')$ by L7. As $\varphi_{i-1}\ \alpha' = \varphi_i\ \alpha$ and $ws\ (\varphi_i\ \alpha) = \varphi_i\ (ws_i\ \alpha)$, the renaming functions identify $ws_i\ \alpha$ and $ws_{i-1}\ \alpha'$. Hence, they both write to (a, l) (L6), but not necessarily the same value, because renamings do not consider values read or written. As $\alpha' \in C_{i-1}$, the write $ws\ (\varphi_{i-1}\ \alpha')$, which α' sees in ξ, has already been committed in C_{i-1} by Lem. 5.5. Hence, $ws_{i-1}\ \alpha' \in C_{i-1}$, because $ws\ (\varphi_{i-1}\ \alpha') = \varphi_{i-1}\ (ws_{i-1}\ \alpha')$ and φ_{i-1} is injective on C_{i-1} (L11). By L11, α' reads from (a, l), because α does. So, $ws_{i-1}\ \alpha'$ writes to (a, l) by W1. Since $ws_{i-1}\ \alpha'$ is committed, it writes the same value as $\varphi_{i-1}\ (ws_{i-1}\ \alpha')$ $(= ws\ (\varphi_i\ \alpha))$ by L6, namely v. Hence, I have found $ws_{i-1}\ \alpha'$.

[24] The attentive reader might wonder why T2 comes with the premise that *typeof-addr* a be defined; without it, this proof would be much simpler. Unfortunately, the single-threaded semantics would not satisfy T2 without this premise. Note that T2 does not (and must not) assume conformance of the thread-local state $(Q\ t\ x\ \sigma)$. Hence, it also holds for non-conforming states which may yield undefined behaviour. Fortunately, the structure of JinjaThreads source code and bytecode semantics satisfies T2 even for non-conforming states with the premise.

[25] For $i = 0$, the legality constraints do not ensure that (ξ_0, ws_0) is well-behaved. Consequently, type-unsafe executions are allowed. However, this does not matter, because justifications are existentially quantified and (ξ_0, ws_0) can always be changed to a type-safe one: Whenever $(\xi_i, ws_i, C_i, \varphi_i)_i$ (weakly) justifies (ξ, ws), so does $(\xi'_i, ws'_i, C'_i, \varphi'_i)_i$ where $\xi'_0 = \xi_1$, $ws'_0 = ws_1$, $C'_0 = \emptyset$, $\varphi'_0 = \varphi_1$, and $\xi'_i = \xi_i$, $ws'_i = ws_i$, $C'_i = C_i$, $\varphi'_i = \varphi_i$ for $i \geq 1$, i.e., (ξ_0, ws_0) is replaced by (ξ_1, ws_1), but without committing any events.

```
      class X extends Exception { int f; }
        initially: b = false; x = y = null;
----------------------------------------------------
1: r1 = y; | 3: r2 = x;          | 8: b = true;
2: x = r1; | 4: if (!b)          |
           | 5:   r2 = new X();  |
           | 6: y = r2;          |
           | 7: throw r2;        |
```

Fig. 28: A program that the JMM allows to terminate with a raised, but unallocated exception

Otherwise, α' has been newly committed in C_{i-1}. Hence, L9' yields that $ws\ (\varphi_i\ \alpha)$ has been committed in C_{i-1}. So, there is some $\beta \in C_{i-1}$ such that $\varphi_{i-1}\ \beta = ws\ (\varphi_i\ \alpha)$. β is the write I am looking for: L11 ensures that β writes to (a, l), too; by L6, β and $ws\ (\varphi_i\ \alpha)$ write the same value to (a, l), namely v. □

COROLLARY 5.6. *Every read in every legal execution is type correct. Justifications need only consider type-safe executions.* □

5.3. Discussion

I have just proved type safety for the JMM. Now, I discuss whether its implications are satisfactory.

Thm. 5.1 does not mention progress or subject reduction, as is typical for type safety proofs [Wright and Felleisen 1994]. But one should not expect this, because the JMM layer is too abstract to express these notions. Hence, Thm. 5.1 should not talk about this, either. Rather, it enables using these well-established techniques for the single-threaded semantics. It shows that one may assume that reading a location always returns a type-correct value, provided that addresses have static types as in §2.1.2. This suffices for typical subject reduction proofs. Hence, these proofs can completely forget about the memory model issues. In fact, I have done so, see [Lochbihler 2007, 2012b] for details.

Still, there are some issues with memory allocation: type safety does not express that everything has been allocated. Consider, e.g., the program in Fig. 28, which is a variation on Fig. 8. There is a legal execution in which t_2 terminates with an uncaught exception whose object has never been allocated. The justification is similar to Fig. 27: First, commit both data races on `y` and `x`; then, flip the `if` condition such that l. 5 does not execute. Still, `y` and `x` refer to the exception object that l. 5 would allocate, and l. 7 throws it.

Nevertheless, the JMM ensures that uninitialised memory cannot be read. Replace l. 7 in Fig. 28 with `r3 = r2.f;`. Then, one cannot find a legal execution where l. 7 accesses the field `f`, although l. 5 has not executed, i.e., the object was not allocated: One can still commit the races on `y` and `x` as before, but as soon as the `if` condition is flipped, the execution becomes ill-formed: there is no write that the read in l. 7 can see, as the allocation in l. 5 does not execute. Hence, well-formedness dismisses this execution; it is illegal. By the same argument, programs in general never read uninitialised memory.

Yet, the JMM does allow writes to uninitialised memory locations. If a read of the same location happens after such a write, the read may see the write, i.e., the write acts like an initialisation except that does not happen before everything else. If I replace l. 7 in the running example with `r2.f = 0; r3 = r2.f;`, it is a legal execution to skip l. 5 and to have the new write and read access the field `f` of the unallocated `X` object.

Recall that assumption D2 demands that every execution initialise every (accessed) location *at most* once and that assumption D3" requires "*exactly* once" for non-speculative executions. In fact, "exactly once" need not hold for speculative ones, as the execution from the last paragraph demonstrates: the `X` object is never allocated and therefore its field `f` never initialised, although it is accessed. In this sense, this execution violates the JLS requirement that all variables be initialised at the start [Gosling et al. 2005, §17.4.4]. The cause is memory allocation generating the initalisations. Given that the JLS requirement is inconsistent [Aspinall and Ševčík 2007a], I argue that one can neglect this deviation – especially as the legality conditions should not allow the execution in the first place (see §5.4).

5.4. Out-of-Thin-Air Values and the Java Security Architecture

Type safety and the Java security architecture [Gong 2003] have been the main motivation for the (complicated) legality constraints, because they require to that programs with data races have defined semantics. As I have explained in §1.1, they rely on values not appearing out of thin air. Banning such values has been an important concern during the last decade – Pugh [2000] first noticed the need to ban them and Manson et al. [2005] expand on the issue. The recent C++11 standard bans out-of-thin-air values, too, although informally [ISO JTC1/SC22/WG21 2011, §29.3.9]; Boehm [2007] explains the main ideas and problems. Nevertheless, it is still unclear what actually constitutes an out-of-thin-air value and no formal definition has been found to date.However, one can narrow down this notion from its motivation: type safety and the security architecture.

Ševčík [2008] showed a weak form of out-of-thin-air guarantee for a class of program transformations: If a program has no means to generate a certain value, then no transformation of the program can output that value. For example, a program without arithmetic only outputs numbers that literally appear in the program. His guarantee supports neither type safety nor the SA, because they require that values do not appear at locations other than they are meant for. My type-safety proof therefore is a stronger form of out-of-thin-air guarantee.

Now, let me approach this notion from the security point of view. According to the JLS, "many security features of the Java programming language depend upon `Strings` being perceived as truly immutable, even if malicious code is using data races to pass `String` references between threads" [Gosling et al. 2005, §17.5]. To that end, the string contents are stored in a `char` array whose reference must not escape the `String` class. The out-of-thin-air guarantee should prevent malicious code from forging a pointer to the `char` array – otherwise, it can mutate the string contents and break the security architecture.

Now, consider Fig. 8 again, but replace both `new` expressions (in ll. 5 and 7) with `new char[2];`. Imagine that l. 5 allocates the `char` array that is to store the contents of a `String` object. Now, both ll. 5 and 7 allocate an object of the same type, so it is very likely that the allocations return the same address – even when addresses carry static type information – because only one of them can execute in one execution. Nevertheless, Fig. 27 justifies an execution where `r2` references the same array as `r3`. Hence, data races *can* forge pointers under the JMM.

This example shows that the out-of-thin-air guarantee is too weak to support the security architecture. However, it is only a theoretical example, because I do not know of any optimisation in a compiler, a JVM, nor in hardware that could lead to such behaviour. Hence, this should be considered a deficiency of the JMM specification. Static type information for addresses rescues type safety, but does not solve the fundamental problem, which appears to be inherent to committing semantics. A real solution is still missing.

6. CLARIFICATIONS OF AND FIXES TO THE JMM

The following list summarises my clarifications and changes to the JMM beyond Aspinall's and Ševčík's [2007a].[26] The letters on the right indicate whether this is necessary to model Java (M) adequately or to ensure the properties consistency (C), DRF guarantee (D), and type safety (T), or that this simplifies (S) the model. Below, I review them in the given order with references to the previous sections.

(1) Additional events needed for thread existence, wait-notify and interruption. M
(2) All initialisation events are synchronisation events. D
(3) Synchronisation order is an ω-order on non-initialisation events. CS
(4) Memory allocation generates initialisation events. M
(5) Interleaved traces render thread divergence events obsolete. S
(6) The type of an object must be computable from its address alone. T
(7) The identity of an event is relative to a justification sequence. MS

Additional events needed for thread existence, wait-notify and interruption. I add the events *ThreadEx*, *Suspend*, *Notify*, *NotifyAll*, *WakeUp*, *ClearIntr*, and *NotIntrd*; the examples P2, P3, P4, P5, and Fig. 20 show that they are necessary to implement all behaviours the JLS allows. However, they *are not and should not* be synchronisation events, i.e., they do not affect $\leq_{\mathrm{hb}}$. My model makes this explicit by removing the additional events from the traces (STEP), but this is not necessary.

Otherwise, if spawns, e.g., synchronised with *ThreadEx*, too, a call to `Thread.start` has release-acquire semantics instead of just release. This forbids a compiler to move a non-volatile read before the call, as the following program P7, a variation of P2, illustrates:[27]

initially: `t = new Thread(); x = 0;`

```
1: x = 1;       | 3: try { r1 = 0;
2: t.start();   | 4:   t.start();
                | 5: } catch (IllegalThreadStateException e) {
                | 6:   r1 = 1; }
                | 7: r2 = x;
```
(P7)

Can we have `r1 == 1` and `r2 == 0`? To get `r1 == 1`, we must execute l. 6, but l. 6 executes only if the thread on the left has spawned `t` before, which executes after the write to `x` in l. 1. So, if the successful call of `start` in l. 2 synchronised with the failing one in l. 4, l. 1 would happen before l. 7 and therefore l. 7 would have to read `1` from l. 1, i.e., `r2 == 0` would be forbidden. In this case, a compiler or VM must not move l. 7 before l. 3, because even interleaving semantics allows the result for the transformed program. In contrast, the JMM semantics permits the reordering: it allows the result already for P7, as l. 1 does not happen before l. 7 and therefore l. 7 may read the initial value `0`.

All initialisation events are synchronisation events. This has been unclear in the JLS: either them being synchronisation events or not leads to inconsistencies in the specification [Aspinall and Ševčík 2007b]. The DRF proof provides evidence that treating allocations as synchronisation events is a good choice (§3.1). Moreover, as *all* initialisation events synchronise with *Start* events, I may combine the initialisation events for all fields of one object in a single allocation event. Otherwise, I would have

[26] As discussed in §2.4.4, I also drop causality condition 8. As Aspinall and Ševčík [2007a] have already omitted this condition, I do not count it as a clarification of mine.

[27] In [Demange et al. 2013], a successful call to `start` synchronises also with the failing ones. Therefore, their semantics forbids this optimisation.

had to separate the event for initialising ordinary members from the one for volatiles. This would have complicated the model and the proofs – for example, splitting the proof of assumption D3” into two parts (§3.4.1 and §3.4.2) relies on a single allocation event.

Synchronisation order is an ω-order on non-initialisation events. The JMM constrains $\leq_{so}$ to be an ω-order in well-formed executions. As Aspinall and Ševčík [2007a] already noted, in an infinitely running program, infinitely many allocation events for volatile fields synchronise with thread start events – this violates this constraint, so the JMM would allow no behaviour at all. As I derive the orders from traces, $\leq_{so}$ is of order at most ω *on non-initialisation events* by construction. This restriction appears sensible, because it (i) resolves the inconsistencies with initialisation events being synchronisation events and (ii) still ensures a global time for synchronisation events. Note that my model need not explicitly impose this restriction, because all traces satisfy it.

Memory allocation generates initialisation events. The JMM leaves the source of initialisation events open, as Aspinall and Ševčík [2007a] have noted. I demonstrate that memory allocation is a reasonable choice (§2.7). Although this complicates proofs (§3.2 to §3.4), infinite executions and final fields cause no problems – unlike a special initialisation thread suggested by Aspinall and Ševčík [2007a].

Interleaved traces render thread divergence events obsolete. My model omits thread divergence events and hang actions (§2.8). This simplification is only possible because the threads' steps of execution are interleaved coinductively. Similarly, interleaving provides some properties for free, such as mutual exclusion for locks (§2.2), freshness of addresses (§2.1), and the total order for sequential consistency (Footnote 19). Moreover, corecursion and coinduction become available.

Nevertheless, interleaving can be eliminated if desired; then, additional well-formedness constraints have to enforce these properties (e.g., in the style of Aspinall and Ševčík [2007a]), and an additional event has to distinguigh divergence from non-termination. Note that freshness of addresses is no issue, either: as allocation events participate in the linear synchronisation order, this order allows to keep track of freshness, see Footnote 3 (in my model, the shared state already takes care of that).

The type of an object must be computable from its address alone. Regarding type information and array lengths, the JLS only states that the JMM does not affect them – this is too vague. As discussed in §5.1, type safety requires that all objects that one concrete address references have the same type across all executions of a justification. In this work, I enforce this by including the type information in the address itself. In the model, this includes the full class name; but a more light-weight approach like encoding a type number in the lower bits of a pointer could also work.

The identity of an event is relative to a justification sequence. The JMM does not tell how to assign identifiers to events. In §2.5, I show that the identity of an event has to include the justification sequence in which it occurs; previous declarative approaches failed to do so. I introduce event renaming and adapt the legality conditions (§2.4.4) to avoid such complicated constructions. Then, an execution-based identity such as a plain sequence number suffices.

7. RELATED WORK

A lot of work has been devoted to hardware MMs, see [Adve and Gharachorloo 1996; Steinke and Nutt 2004; Boudol and Petri 2009] for an overview. Here, I focus on programming language MMs, which are looser than hardware MMs (and therefore harder to design), because they should be efficiently implementable on various hardware and

allow as many compiler optimisations as possible, but nevertheless be defined unambiguously.

Huisman and Petri [2007] have formalised the JMM and the proof of the DRF guarantee in the proof assistant Coq. They have already noted that initialisations break the proof, but added an axiom to avoid the problem. They set out at the abstract level of threads in isolation, without connection to an operational semantics.

Aspinall and Ševčík [2007a] have formalised parts of the JMM relevant for the DRF guarantee and proved the latter in Isabelle/HOL — which I have found very helpful in extending the DRF guarantee proof. Since they omit dynamic allocation, they need to consider only finite prefixes of executions. This simplifies their proofs considerably, as they do not need to assume that sequentially consistent completions of executions exist. They do not provide an intra-thread semantics, either. Instead, they model a program as an unspecified predicate that checks whether a trace of memory accesses and synchronisation operations represents a valid execution of the thread. This does not suffice to model the hidden communication channels between threads that the JLS specifies, see the examples P1 and P2.

They have also studied which compiler transformations the JMM allows [Ševčík and Aspinall 2008]. They show that weak legality improves on legality in that it always allows to reorder memory accesses to distinct, non-volatile locations, but they have found counterexamples to several other optimisations [Aspinall and Ševčík 2007b].

Jagadeesan et al. [2010] define an operational semantics for weak MMs with speculative computations similar to the JMM, which also provides the DRF guarantee. Instead of validating executions a posteriori, their semantics explicitly encodes permitted reorderings and speculation. Similar to §2.1.2, their semantics computes the type of an object only from its address. Yet, their model is neither machine-checked nor comparable to the JMM for programs with data races and synchronisation. They show that their semantics validates many of Aspinall's and Ševčík's counterexamples [2007b] to local optimisations in the JMM. However, their semantics does not allow reordering with memory allocation as the JMM does, which required a major effort in this work; for example, they do not allow the execution in Fig. 18. Like the JMM, they claim that their model bans out-of-thin-air values, but their semantics allows the same out-of-thin-air values as mine, e.g., the one from §5.4 that could compromise Java's security architecture.

Goto et al. [2012] have extended this semantics with correspondence assertions and a type system for which they show type safety. Yet, their type safety statement is buggy, because the definition of being stuck misses some cases, e.g., multiple spawns of one thread as in P2. Once more, this shows that machine-support is essential in dealing with memory models. They face a similar problem of typing speculative reads of non-conforming values, which they call shape traps and ignore during type checking. As initialisations are treated like ordinary writes, their proofs do not have to deal with the additional complications such as assumptions D2, D3, C2, and T2. Their monolithic semantics and type system lead to large and complicated proofs. In contrast, my stack of semantics leads to modular proofs with explicit assumptions at the interfaces.

Boyland [2009] formalises in Twelf a semantics for a simple language with allocation, synchronisation, volatiles, thread spawns and joins, which may raise an error upon a data race. He shows that a program never raises such errors iff it is data-race free in the JMM sense. For programs with data races, the semantics misses many behaviours that the JMM allows, e.g., reorderings as in Figs. 3 and 18, whereas my semantics deals with the full JMM.

For a kernel language, Cenciarelli et al. [2007] define an interleaving small-step semantics that generates configuration structures of events which an axiomatic theory

constrains. On paper, they show that they only generate behaviours that the JMM allows, but it is unknown whether they produce every allowed behaviour and they do not consider the DRF guarantee.

Some model checkers are aware of the JMM [De et al. 2008; Torlak et al. 2010; Jin et al. 2012]. Each of them comes with an identification scheme for events, but none of them presents a thorough analysis of its implications like I did in §2.5. Only the model checker by Torlak et al. exactly captures the JMM; De et al.'s under-approximates it (misses, e.g., Fig. 3b), Jin et al.'s over-approximates it (e.g., allows the forbidden result for JMM test case 5 [Pugh and Manson 2004]). Using whole-program analysis, Torlak et al.'s algorithm computes all events and memory allocations in advance. They focus on checking small test cases with finite state rather than providing a full semantics and proofs.

The recent C++11 standard [ISO JTC1/SC22/WG21 2011] considers programs with data races ill-formed and assigns undefined semantics to them, but offers finer shades of synchronisation than Java. Relaxed atomics provide visibility guarantees similar to ordinary fields in Java, although there are subtle differences [Boehm 2007]. Boehm and Adve [Boehm and Adve 2008] describe a preliminiary version of the memory model and prove the DRF guarantee for programs which use only strong synchronisation primitives. They show that such programs are characterised more intuitively as never having conflicting events adjacent in any interleaving. For the JMM, this equivalence does not hold, as threads can communicate without introducing happens-before relationships.[28] Similar to the Java DRF guarantee, their proof assumes that sequentially consistent executions exist, but they do not construct them formally. Batty et al. [2011] have formalised the memory model with a focus on rigorously defining the semantics, and proved correct compilation schemes for synchronisation primitives. In [Batty et al. 2012], they formally proved the DRF guarantee along the lines of [Boehm and Adve 2008]. Surprisingly, they found that the intuitive characterisation of data races does not hold for C++11 in general, because C++11 treats initialisations of atomics specially, namely as non-atomic writes.

Ševčík et al. [2011] have verified the CompCert compiler backend with respect to the formal memory model for x86 processors by Sewell et al. [2010], which is the first formal correctness proof for an optimising compiler backend with respect to a weak MM. They expose the total store order (TSO) model in a C-like programming language, which is considerably stronger than the JMM and also provides a DRF guarantee.

Demange et al. [2013] have extended the TSO model with synchronisation primitives (re-entrant monitors, volatiles, thread spawns and joins, but not interruption and the wait-notify mechanism) and connected it with a Java bytecode semantics. Their model BMM is strictly stronger than the JMM and thus inherits the DRF guarantee from Java. They want to integrate the model in a verified compiler infrastructure targeted to TSO hardware. Coq type-checked their definitions of the model, but they have not formalised the proofs. Whether their model supports proofs mechanisation well, therefore, remains open.

[28] Consider the following program P8, a variation of P7.

initially: `t = new Thread(); x = 0; y = 0;`	
`1: x = 1;`	`4: try { t.start();`
`2: y = 1;`	`   } catch (IllegalThreadStateException e) {`
`3: t.start();`	`5:   r = x; }`

(P8)

As discussed for P7 in §6, l. 1 does not happen before l. 5 according to the JMM (and my semantics), i.e., the program is not correctly synchronised. However, there is no sequentially consistent interleaving with adjacent conflicting actions, because the write in l. 2 always separates l. 1 and l. 5.

Verbrugge et al. [2011] observe that the ban on out-of-thin-air values disallows aggressive compiler optimisations. For example, a compiler must not reverse the iteration order of a loop that sums up the elements of an array and stores the intermediate results in a shared location. Although such a transformation is functionally equivalent, the intermediate results are different and therefore appear out of thin air. They suggest that threads share only volatile fields, non-volatile locations are thread-local. This breaks common programming idioms like safe publication through volatiles [Manson et al. 2005, Fig. 3] and the performance penalty is unknown. Nevertheless, I agree that to racy programs, the JMM should give as weak semantics as possible that still supports type safety and the security architecture. The latter is the hard part, because type safety could be constructed into the model. Unfortunately, C++'s (informal) ban on out-of-thin-air values does not provide the DRF guarantee [Boehm 2007].

8. CONCLUSION AND FUTURE WORK

My machine-checked model of multithreaded Java spans from a realistic subset of Java source code and bytecode to the Java memory model. I have proven that it provides the DRF guarantee and is type safe. Beyond that, JinjaThreads features a compiler, which I have not presented in this article. But it is worth to note that its verification [Lochbihler 2010] relies on type safety of the bytecode language and that I have extended this verification to the JMM using the type safety result.

JinjaThreads is part of the Quis Custodiet project,[29] which aims at formally verifying an infrastructure for information flow control (IFC). The DRF guarantee and type safety are essential: the IFC algorithm assumes interleaving semantics [Giffhorn 2012], i.e., it is only sound for correctly synchronised programs. And establishing the absence of data races relies on type safety, because intermediate representations like control flow graphs statically approximate dynamic dispatch based on types.

To achieve this goal, my formalisation stays close to Java and tries not to palliate its ugly corners as simple core languages often do. This article consequently cannot describe all the formal details; it rather presents most of the model informally. Nevertheless, remember that all this has been mechanised in Isabelle/HOL [Lochbihler 2007]. By exploring the dark corners, the formalisation lead to several clarifications of and fixes to the JMM. As a byproduct, I have found numerous new examples of corner cases that illustrate unexpected interactions between different features and the memory model.

In total, JinjaThreads consists of 85k lines of Isabelle definitions and proofs. Of these, 11k are alotted to defining the JMM and to conducting the proofs abstractly at the multithreaded level. Discharging the assumptions on the single-threaded semantics requires 1.5k for source code and 1.6k for bytecode plus 1.0k shared between the two. This demonstrates that separating the memory model from the single-threaded semantics yields huge savings.

Despite its size and complexity, JinjaThreads does not yet cover full sequential Java. From the concurrency perspective, the following three features are particularly interesting and investigating them in the future seems worthwhile:

(1) *Class initialisation.* Classes must be initialised at most once even if multiple threads trigger initialisation concurrently. The initialisation procedure specified in the JLS [Gosling et al. 2005, §12.4.2] uses locks to synchronise such threads. However, the JLS allows compilers and the JVM to remove "unnecessary synchronisation" for class initialisation [Gosling et al. 2005, §12.4.3]. It is unclear how to model these optimisa-

[29] `http://pp.ipd.kit.edu/projects/quis-custodiet/`

tions and their consequences on threads that synchronise through class initialisation (see [Lochbihler 2012b, §7.4] for an example).

(2) *Final fields.* For final fields, the JMM provides stronger guarantees than for ordinary ones [Gosling et al. 2005, §17.5]. I expect that extending the JMM formalisation with final fields does not pose any major difficulties. However, final fields are designed to be used without synchronisation; in particular, writes to final fields need not happen before reads – otherwise, final fields would have synchronisation semantics like volatiles. As a single race on a final field expels the program from the DRF guarantee, one should also revisit the guarantee. Ideally, one would show that data races on final fields can only occur if there is another data race on some non-final field. Such a result can only hold if the program adheres to the coding discipline of not writing “a reference to the object being constructed in a place where another thread can see it before the object’s constructor is finished” [Gosling et al. 2005, §17.5], but it is open whether this is also sufficient.

(3) *Garbage collection and finalisation.* JinjaThreads omits garbage collection (GC), although in practice, GC is needed for infinite executions that keep allocating fresh objects – and GC triggers object finalisation that runs concurrently in separate finalisation threads.[30] In the JLS [Gosling et al. 2005, §12.6.1.1], the visibility restrictions for finalisers assume that objects have a unique identity (with respect to a single execution) and that read and write events refer to these identities. Therefore, unique initialisations of locations, on which the DRF guarantee and type safety rely, does not constrict GC: in my formalisations, addresses are the unique identifiers, they must not be confused with physical addresses. That is, a VM model with GC must merely use distinct identifiers for accesses to memory that has been reclaimed in between. In particular, a correct garbage collector does not affect type safety, although physical addresses can be re-used multiple times.
The visibility constraints for finalisation, however, pose two challenges: First, they group all events of an execution into epochs such that the synchronisation events of an epoch form an interval of the synchronisation order. Assignment of non-volatile reads and writes to epochs, though, need not respect happens-before. Similar to §2.10, it is unclear how to model such relaxed consistency requirements in interleaved traces. Second, these constraints are supposed not to restrict the visibility of writes outside of finalisers such that compilers and VMs can ignore finalisation when they optimise other methods, but it is open whether this holds and how to formally state this.

Regarding the JMM itself, initialisations and the special way the JMM handles them caused most complications in the proofs. In the future, I want to investigate simpler ways of initialising locations. My DRF guarantee proof shows that the special treatment is irrelevant for correctly synchronised programs. I am therefore not constrained when searching for better options. Moreover, it is unsatisfactory that values can appear out of thin air. However, one first needs a good understanding of what an out-of-thin-air value actually should be and what restrictions the ban imposes on compilers. In the long term, this might lead to another revision of the JMM that tackles the open problems.

APPENDIX

In the following, I give a quick tour of the concurrency features of Java 6 as specified in the Java language specification [Gosling et al. 2005].

[30]The VM runs the method `finalize` of an object somewhen between the object becoming unreachable from ordinary threads and the GC reclaiming the memory. Such finalisers are used to free resources that are not under the control of the GC, e.g., file descriptors.

Threads are parallel strands of execution with shared memory. A program controls a thread through its associated object of (a subclass of) class *Thread*. To spawn a new thread, one allocates a new object of class *Thread* (or any subclass thereof) and invokes its *start* method. The new thread will then execute the *run* method of the object, in parallel with all other threads. Each thread must be spawned at most once, every further call to *start* raises an *IllegalThreadState* exception. The thread terminates when *run* terminates, either normally or abruptly due to an exception. The static method *currentThread* in class *Thread* returns the object associated with the executing thread.

Java offers four kinds of synchronisation between threads: locks, wait sets, joining, and interrupts. The package `java.util.concurrent` in the Java API builds sophisticated forms of synchronisation from these primitives and atomic compare-and-set operations.

Every object (and array) has an associated monitor with a lock and a wait set. Locks are mutually-exclusive, i.e., at most one thread can hold a lock at a time, but re-entrant, i.e., a thread can acquire a lock multiple times. For locking, Java uses `synchronized` blocks that take a reference to an object or array. A thread must acquire the object's lock before it executes the block's body, and releases the lock afterwards. If another thread already holds the lock, the executing thread must wait until the other thread has released it. Thus, `synchronized` blocks on the *same* object never execute in parallel.

To avoid busy waiting, a thread can suspend itself to the wait set of an object by calling the object's method *wait*, which class *Object* declares. To enter the wait set, the thread must have locked the object's monitor and must not be interrupted; otherwise, an *IllegalMonitorState* exception or *InterruptedException*, respectively, is thrown. If successful, the call also releases the monitor's lock completely. The thread remains in the wait set until (a) another thread interrupts or notifies it, or (b) if *wait* is called with a time limit, the specified amount of time has elapsed, or (c) it wakes up spuriously. After having been removed, the thread reacquires the lock on the monitor before its execution proceeds normally or, in case of interruption, by raising an *InterruptedException*. The methods *notify* and *notifyAll* remove one unspecified or all threads from the wait set of the call's receiver object. Like for *wait*, the calling thread must hold the lock on the monitor. Thus, the notified thread continues its execution only after the notifying thread has released the lock.

When a thread calls *join* on another thread, it blocks until (a) the thread that the receiver object identifies has terminated, or (b) another thread interrupts the joining thread, or (c) an optionally-specified amount of time has elapsed. In the second case, the call raises an *InterruptedException*; otherwise, it returns normally.

Interruption provides asynchronous communication between threads. Calling the *interrupt* method of a thread sets its interrupt status. If the interrupted thread is waiting or joining, it aborts that, raises an *InterruptedException*, and clears its interrupt status. Otherwise, interruption has no immediate effect on the interrupted thread. Instead, class *Thread* implements two methods to observe the interrupt status. First, the static method *interrupt* returns and resets the interrupt status of the *executing* thread. Second, the method *interrupted* returns the interrupt status of the receiver object's thread without changing it.

Apart from that, class *Thread* also declares the methods `yield` and `sleep`. They instruct the scheduler to prefer other threads and cease execution for the specified time, respectively. Since these are only recommendations to the scheduler, they cannot be used for synchronisation.

Additionally, the `java.util.concurrent` library [Lea 2004] provides advanced synchronisation primitives for high-performance applications. Their implementations by-

pass Java and the JMM by invoking native methods from `sun.misc.Unsafe` such as "compare and swap". Consequently, this work does not cover these extensions.

ACKNOWLEDGMENTS

I would like to thank Joachim Breitner, Martin Hecker, Denis Lohner, Martin Mohr, and Gregor Snelting for valuable discussions about this work; their comments as well as the anonymous reviewers' helped to focus and clarify the presentation. I also thank Emina Torlak and Julian Dolby for contributing the justification for JMM test case 18 (Fig. 16b).

REFERENCES

ADVE, S. V. AND GHARACHORLOO, K. 1996. Shared memory consistency models: A tutorial. *Computer 29,* 12, 66–76.

ADVE, S. V. AND HILL, M. D. 1990. Weak ordering — a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA 1990)*. ACM, 2–14.

ALVES-FOSS, J., Ed. 1999. *Formal Syntax and Semantics of Java*. Lecture Notes in Computer Science Series, vol. 1523. Springer.

ASPINALL, D. AND ŠEVČÍK, J. 2007a. Formalising Java's data-race-free guarantee. In *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, K. Schneider and J. Brandt, Eds. Lecture Notes in Computer Science Series, vol. 4732. Springer, 22–37.

ASPINALL, D. AND ŠEVČÍK, J. 2007b. Java memory model examples: Good, bad and ugly. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP 2007)*. technical report ICIS-R07021. University of Nijmegen, 66–80.

BATTY, M., MEMARIAN, K., OWENS, S., SARKAR, S., AND SEWELL, P. 2012. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*. ACM, 509–520.

BATTY, M., OWENS, S., SARKAR, S., SEWELL, P., AND WEBER, T. 2011. Mathematizing C++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)*. ACM, 55–66.

BOEHM, H.-J. 2007. Memory model rationales. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.html`. ISO JTC1/SC22/WG21 document no. WG21/N2176.

BOEHM, H.-J. 2012. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2012)*. ACM, New York, NY, USA, 12–20.

BOEHM, H.-J. AND ADVE, S. V. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2008)*. ACM, 68–78.

BOUDOL, G. AND PETRI, G. 2009. Relaxed memory models: an operational approach. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2009)*. ACM, New York, NY, USA, 392–403.

BOYLAND, J. 2009. An operational semantics including "volatile" for safe concurrency. *Journal of Object Technology 8,* 4, 33–53. Formal Techniques for Java Programs 2008.

CENCIARELLI, P., KNAPP, A., AND SIBILIO, E. 2007. The Java memory model: Operationally, denotationally, axiomatically. In *Programming Languages and Systems (ESOP 2007)*, R. De Nicola, Ed. Lecture Notes in Computer Science Series, vol. 4421. Springer, 331–346.

CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1999)*. ACM, New York, NY, USA, 1–19.

DE, A., ROYCHOUDHURY, A., AND D'SOUZA, D. 2008. Java memory model aware software validation. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE 2008)*. ACM, New York, NY, USA, 8–14.

DEMANGE, D., LAPORTE, V., ZHAO1, L., JAGANNATHAN, S., PICHARDIE, D., AND VITEK, J. 2013. Plan B: A buffered memory model for Java. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2013)*. ACM, 329–341.

DROSSOPOULOU, S. AND EISENBACH, S. 1999. Describing the semantics of Java and proving type soundness. See Alves-Foss [1999], 542–542.

FARZAN, A., CHEN, F., MESEGUER, J., AND ROŞU, G. 2004a. Formal analysis of Java programs in JavaFAN. In *Computer Aided Verification (CAV 2004)*, R. Alur and D. Peled, Eds. Lecture Notes in Computer Science Series, vol. 3114. Springer, 501–505.

FARZAN, A., MESEGUER, J., AND ROŞU, G. 2004b. Formal JVM code analysis in JavaFAN. In *Algebraic Methodology and Software Technology (AMAST 2004)*, C. Rattray, S. Maharaj, and C. Shankland, Eds. Lecture Notes in Computer Science Series, vol. 3116. Springer, 132–147.

GIFFHORN, D. 2012. Slicing of concurrent programs and its application to information flow control. Ph.D. thesis, Fakultät für Informatik, Karlsruher Institut für Technologie.

GONG, L. 2003. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation* 2nd Ed. The Java Series. Addison-Wesley.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The Java Language Specification, Third Edition*. Addison-Wesley.

GOTO, M., JAGADEESAN, R., PITCHER, C., AND RIELY, J. 2012. Types for relaxed memory models. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation (TLDI 2012)*. ACM, New York, NY, USA, 25–38.

HILL, M. D. 1998. Multiprocessors should support simple memory-consistency models. *IEEE Computer 31,* 8, 28–34.

HUISMAN, M. AND PETRI, G. 2007. The Java Memory Model: a formal explanation. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP 2007)*. technical report ICIS-R07021. University of Nijmegen, 81–96.

HUR, C.-K., NEIS, G., DREYER, D., AND VAFEIADIS, V. 2013. The power of parameterization in coinductive proof. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2013)*. ACM, 193–205.

ISO JTC1/SC22/WG21. 2011. International standard ISO/IEC 14882:2011. information technology – programming languages – C++. International Organization for Standardization.

JACOBS, B. 2005. JLS3 contains glitch concerning volatiles? Java Memory Model mailing list, post 2477.

JAGADEESAN, R., PITCHER, C., AND RIELY, J. 2010. Generative operational semantics for relaxed memory models. In *Programming Languages and Systems (ESOP 2010)*, A. D. Gordon, Ed. Lecture Notes in Computer Science Series, vol. 6012. Springer, 307–326.

JIN, H., YAVUZ-KAHVECI, T., AND SANDERS, B. A. 2012. Java memory model-aware model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2012)*, C. Flanagan and B. König, Eds. Lecture Notes in Computer Science Series, vol. 7214. Springer, 220–236.

KLEIN, G. AND NIPKOW, T. 2006. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems 28,* 4, 619–695.

LAMPORT, L. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers 28,* 9, 690–691.

LEA, D. 2004. JSR 166: Concurrency utilities. `http://jcp.org/en/jsr/detail?id=166`.

LIU, H. AND MOORE, J. S. 2003. Executable JVM model for analytical reasoning: A study. In *Proceedings of the 2003 workshop on Interpreters, Virtual Machines and Emulators (IVME 2003)*. ACM, 15–23.

LOCHBIHLER, A. 2007. Jinja with threads. *Archive of Formal Proofs*. `http://afp.sf.net/entries/JinjaThreads.shtml`, Formal proof development.

LOCHBIHLER, A. 2008. Type safe nondeterminism - a formal semantics of Java threads. In *Proceedings of the 2008 International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*.

LOCHBIHLER, A. 2010. Verifying a compiler for Java threads. In *Programming Languages and Systems (ESOP 2010)*, A. D. Gordon, Ed. Lecture Notes in Computer Science Series, vol. 6012. Springer, 427–447.

LOCHBIHLER, A. 2012a. Java and the Java memory model – a unified, machine-checked formalisation. In *Programming Languages and Systems (ESOP 2012)*, H. Seidl, Ed. Lecture Notes in Computer Science Series, vol. 7211. Springer, 497–517.

LOCHBIHLER, A. 2012b. A machine-checked, type-safe model of Java concurrency – language, virtual machine, memory model and verified compiler. Ph.D. thesis, Fakultät für Informatik, Karlsruher Institut für Technologie.

LOCHBIHLER, A. AND BULWAHN, L. 2011. Animating the formalised semantics of a Java-like language. In *Interactive Theorem Proving (ITP 2011)*, M. van Eekelen, H. Geuvers, J. Schmalz, and F. Wiedijk, Eds. Lecture Notes in Computer Science Series, vol. 6898. Springer, 216–232.

MANSON, J. 2007. The proof of DRF guarantee and initialization. Java memory model mailing list, post 62.

MANSON, J., PUGH, W., AND ADVE, S. V. 2005. The Java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2005)*. ACM, 378–391.

NIPKOW, T., PAULSON, L. C., AND WENZEL, M. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science Series, vol. 2283. Springer.

NIPKOW, T. AND VON OHEIMB, D. 1998. Java$_{\ell ight}$ is type-safe — definitely. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1998)*. ACM, 161–170.

PETRI, G. AND HUISMAN, M. 2008. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *Bytecode semantics, Verification, Analysis and Transformation (BYTECODE 2008)*. Electronic Notes in Theoretical Computer Science.

PIERCE, B. C. 2002. *Types and Programming Languages*. The MIT Press.

POLYAKOV, S. AND SCHUSTER, A. 2006. Verification of the Java causality requirements. In *Hardware and Software, Verification and Testing (HVC 2005)*, S. Ur, E. Bin, and Y. Wolfsthal, Eds. Lecture Notes in Computer Science Series, vol. 3875. Springer, Berlin, Heidelberg, 224–246.

PUGH, W. 2000. The Java memory model is fatally flawed. *Concurrency: Practice and Experience 12*, 445–455.

PUGH, W. AND MANSON, J. 2004. Causality test cases for the Java memory model. `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`.

RUF, E. 2000. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI 2000)*. ACM, New York, NY, USA, 208–218.

SANGIORGI, D. 1998. On the bisimulation proof method. *Mathematical Structures in Computer Science 8,* 5, 447–479.

ŠEVČÍK, J. 2008. Program transformations in weak memory models. Ph.D. thesis, Laboratory for Foundations of Computer Science, School of Informatics, University of Edinburgh.

ŠEVČÍK, J. AND ASPINALL, D. 2008. On validity of program transformations in the Java memory model. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, J. Vitek, Ed. Lecture Notes in Computer Science Series, vol. 5142. Springer, 27–51.

ŠEVČÍK, J., VAFEIADIS, V., NARDELLI, F., JAGANNATHAN, S., AND SEWELL, P. 2011. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2011)*. ACM, 43–54.

SEWELL, P., SARKAR, S., OWENS, S., NARDELLI, F. Z., AND MYREEN, M. O. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53*, 89–97.

SORIN, D. J., HILL, M. D., AND WOOD, D. A. 2011. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool.

STÄRK, R., SCHMID, J., AND BÖRGER, E. 2001. *Java and the Java Virtual Machine*. Springer.

STEINKE, R. C. AND NUTT, G. J. 2004. A unified theory of shared memory consistency. *Journal of the ACM 51,* 5, 800–849.

SURA, Z., FANG, X., WONG, C.-L., MIDKIFF, S. P., LEE, J., AND PADUA, D. 2005. Compiler techniques for high performance sequentially consistent Java programs. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP 2005)*. ACM, New York, NY, USA, 2–13.

TORLAK, E., VAZIRI, M., AND DOLBY, J. 2010. MemSAT: checking axiomatic specifications of memory models. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2010)*. ACM, 341–350.

VERBRUGGE, C., KIELSTRA, A., AND ZHANG, Y. 2011. There is nothing wrong with out-of-thin-air: Compiler optimization and memory models. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2011)*. ACM, New York, NY, USA, 1–6.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A Syntactic Approach to Type Soundness. *Information and Computation 115,* 1, 38–94.

Received XXX; revised YYY; accepted ZZZ