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.

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 conflicting 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

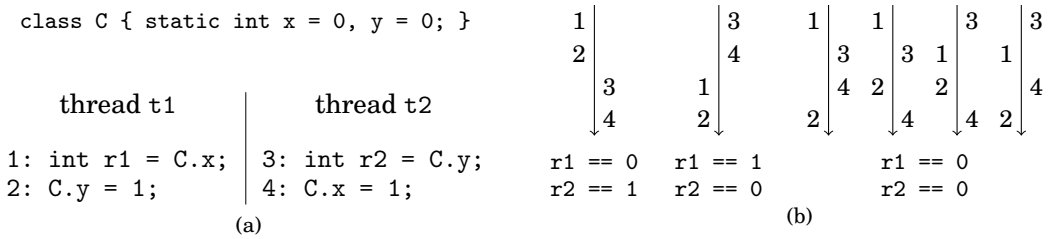


Fig. 1: Program with two threads (a) and all of its sequentially consistent schedules (b)

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 $\leq_{\mathrm{po}}$) 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* $\leq_{\mathrm{hb}}$ 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 ($\_ \leq_{\mathrm{hb}} \alpha \wedge \alpha \not\leq_{\mathrm{hb}} \_$), those that must happen after it ($\alpha \leq_{\mathrm{hb}} \_ \wedge \_ \not\leq_{\mathrm{hb}} \alpha$), and those that may happen concurrently ($\_ \not\leq_{\mathrm{hb}} \alpha \wedge \alpha \not\leq_{\mathrm{hb}} \_$).[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, $\leq_{\mathrm{hb}}$ always includes $\leq_{\mathrm{po}}$. 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 $\leq_{\mathrm{hb}}$ 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.

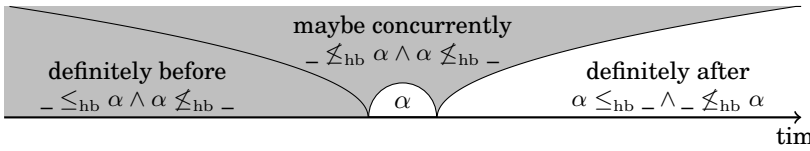
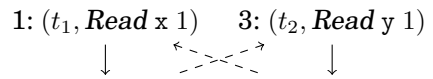


Fig. 2: Happens-before provides a notion of time relative to a given event α. If α is a *Read*, it may see *Write* events in the grey area.

initially: `x = y = 0;`

`1: r1 = x;`	`3: r2 = y;`
`2: y = 1;`	`4: x = 1;`

(a)



(b)

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

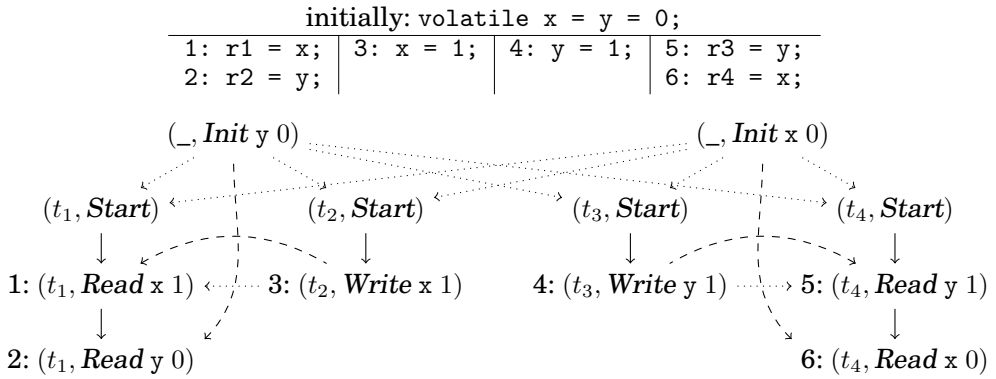


Fig. 4: The classic independent reads of independent writes (IRIW) example

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

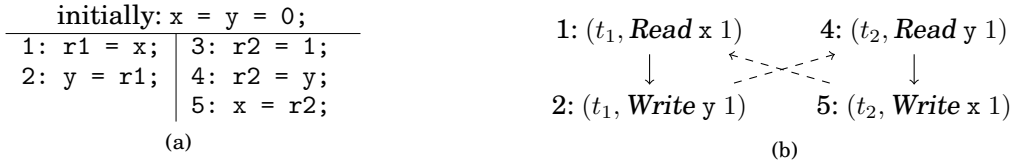


Fig. 5: JMM causality test case 4 with l. 3 added: the value 1 appears out of thin air

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