Java and the Java Memory Model — A Unified, Machine-Checked Formalisation*

Andreas Lochbihler

Karlsruher Institut für Technologie
andreas.lochbihler@kit.edu

Abstract. We present a machine-checked formalisation of the Java memory model and connect it to an operational semantics for Java source code and bytecode. This provides the link between sequential semantics and the memory model that has been missing in the literature. Our model extends previous formalisations by dynamic memory allocation, thread spawns and joins, infinite executions, the wait-notify mechanism and thread interruption. We prove the Java data race freedom guarantee for the complete formalisation in a modular way. This work makes the assumptions about the sequential semantics explicit and shows how to discharge them.

1 Introduction

A memory model (MM) specifies how shared memory behaves under concurrent programs. The most intuitive one is sequential consistency (SC) [15], which assumes interleaving semantics, i.e., threads execute one at a time and all threads immediately see all writes of all other threads. For efficiency reasons, modern hardware implements only MMs weaker than sequential consistency to allow for local caches and optimisations [1]. Similarly, many compiler optimisations that are correct for sequential code lead to unexpected results in concurrent code. Consider, e.g., the two threads in Fig. 1 that share locations x and y. Under sequential consistency, the result r1 == 2, r2 == 1 is impossible. However, if the compiler or the hardware reorders the independent statements in each thread — not being aware of the other thread — this outcome is in fact possible. Weak MMs relax interleaving semantics such that such optimisations become correct.

For the typical programmer, weak MMs like the Java Memory Model (JMM) [10,21] nevertheless provide intuitive SC semantics for an important class of programs – a property known as the data-race freedom (DRF) guarantee [2]: Two accesses to the same (non-volatile) location *conflict* if they originate from different threads and at least one is a write. A *data race* occurs if two conflicting accesses may happen concurrently, i.e., without synchronisation in between. Then, if no SC execution contains data races, the JMM promises that the program behaves like under SC semantics. In other words: If a programmer protects all accesses to shared data via locks or declares the locations volatile or in another way makes

* This work was partially funded by DFG grants Sn11/10-1,2.

H. Seidl (Ed.): ESOP 2012, LNCS 7211, pp. 497–517, 2012.
© Springer-Verlag Berlin Heidelberg 2012

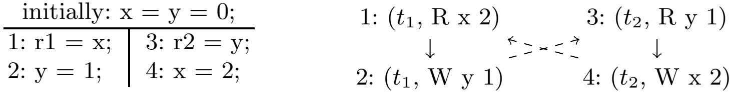


Fig. 1. Example program with data races from [21] (left) and (part of) its JMM execution for the result r1 == 2, r2 == 1 (right)

sure there are no data races, she can forget about the MM and assume interleaving semantics. In the above example, the read of x in l. 1 races with the write in l. 4 (and similarly l. 3 and l. 2 for y), i.e., the DRF guarantee does not apply.

In practice, the DRF guarantee is the most relevant part of the JMM. For type safety and security promises, the JMM also gives semantics to programs with data races, which is the main cause for its complexity. While the DRF guarantee is stated concisely and formally, only test cases [23] underpin these promises about type safety and security, and it is unclear whether the JMM actually provides the latter. Moreover, the JMM inadvertently and unnecessarily disallows certain program transformations that Java virtual machines (JVM) and the hardware regularly perform [9,26,29].[1] Hence, it fails to provide enough flexibility to compiler writers and implementors. Therefore, it is even more important that at least the DRF guarantee holds.

Given the technical complexity of the JMM and Java, it is crucial that all claims are mechanically checked – as a series of false claims about the JMM and their subsequent disproof demonstrates [21,9,26,29]. Moreover, such a JMM formalisation needs to be linked with a sequential semantics for Java, which several authors [4,9,11] have criticised as missing. Since the proof of the DRF guarantee makes assumptions about the sequential semantics, this is a prerequisite to show that Java actually provides it.

To that end, we extend our previous work JinjaThreads [16,17,20], a formalisation of multithreaded Java in the proof assistant Isabelle/HOL [22] as part of the Quis Custodiet project [25]. It models a substantial subset of multithreaded Java source code and bytecode, defines an interleaving semantics, and verifies a compiler from source code to bytecode — all assuming sequential consistency.

Contributions. In this work, we formalise the JMM in the proof assistant Isabelle/HOL [22], connect it to JinjaThreads, and prove the DRF guarantee. To our knowledge, this is the first unified, machine-checked model for Java and the JMM. All definitions and proofs have been checked by Isabelle and are available online in the Archive of Formal Proofs [18]. The accompanying technical report [19] contains high-level proofs for all theorems and further examples.

First, we present a *consistent* formalisation of the JMM based on the operational JinjaThreads semantics for Java source code and bytecode (§2). Our model covers dynamic memory allocation, thread spawns and joins, infinite

[1] It is inadvertent because the JMM's designers claimed that it allows such transformations [21], but were later proven wrong [9,26]. It is unnecessary as neither the DRF guarantee nor Java type safety nor its security promises would be broken.

executions, the wait-notify mechanism and interruption, all of which previous JMM formalisations [4,11] have omitted. Dynamic allocation and the special treatment of memory initialisation in the JMM force us to deal with infinite executions (see §1.1 for an informal JMM explanation).

Our model establishes a solid link between the semantics for sequential Java and the JMM by associating Java statements with their JMM inter-thread actions. In novel examples, we show that the Java Language Specification (JLS) [10] and the Java API define communication channels between threads that the JMM does not cover. Covert channels make the behaviour of one thread dependent on another thread's without synchronisation or memory access. We extend our model accordingly (§§2.1,2.2). Following [8,9], we interleave all threads and reconstruct the fundamental notions of the JMM a posteriori (§2.3).

Second, we prove the DRF guarantee (§3) for source code and bytecode. Our proof resolves the inconsistencies with initialisations of locations in previous proofs [21,11]. To bridge the gap between the axiomatic JMM and the operational semantics, we identify the assumptions of the DRF proof (§3.1) and prove that the semantics satisfies them (§3.2). Although these assumptions are intuitive, they surprisingly require a full subject reduction proof for sequentially consistent executions. In particular, we explicitly construct sequentially consistent executions for a given prefix by corecursion. Again, initialisations turn out to be the main complication in the proofs.

1.1 An Informal Introduction to the JMM

In this section, we informally explain the ideas of the JMM – see §2 for the formalisation. Aiming for independence from concrete hardware and implementations, the JMM [10, §17.4] consists of axiomatic rules that determine a posteriori whether a given execution is an allowed behaviour of a given program. To that end, it abstracts concrete thread operations to *(inter-thread) actions*:

- reading (R) from, writing (W) to and initialising (I) heap-based locations,
- locking (L) and unlocking (U) a monitor,
- thread start (S) and finish (F),
- interrupting (Ir) a thread and observing that it has been interrupted (Ird),
- spawning (Sp) of and joining (J) on a thread, and
- external actions (E) – for I/O, for example.

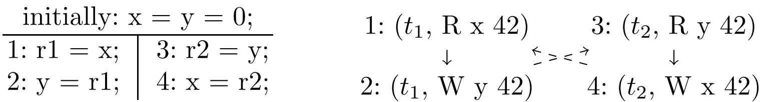
Actions in the JMM only deal with heap locations, i.e., object fields and array cells. Access to local variables, method parameters, and type information does not generate any inter-thread actions and is thus unaffected by the JMM.

In a given execution, the actions of a single thread are totally ordered by the sequence in which they would occur according to the intra-thread semantics, the so-called program order. Being consistent with this total order, the happens-before order provides a notion of time relative to a given action. It partitions the other actions into three groups: those that must have happened before it, those that must happen after it, and those that may happen concurrently. Synchronisation actions, which are all inter-thread actions except for external actions

and reads from and writes to non-volatile locations, introduce happens-before relationships between actions of different threads.

The right-hand side of Fig. 1 shows the essential part of the execution with the unexpected result using the following notation (see [19, Fig. 5] for the complete execution): Statements are abstracted to their actions labelled with the thread ID. The solid arrows represent program order, transitive relationships are not shown. Dotted arrows used in later examples denote synchronisation (synchronises-with relationships); as there is no synchronisation, happens-before coincides with program order. Hence, l. 1 and l. 2 may happen "concurrently" with l. 3 and l. 4. The dashed arrows denote the flow of values from writes to reads. An execution assigns to each read action the write action it sees, e.g., l. 1 sees the write from l. 4.

The JMM requires that the write must not happen after the read. However, if only happens-before determines visibility of write actions, values may appear *out of thin air*. Consider, e.g.,



The reads in ll. 1 and 3 may see the writes in ll. 4 and 2, resp., as they may happen concurrently. If both writes write 42, both reads may read 42. Since the program cannot normally produce 42, 42 appears out of thin air.

For type safety and security guarantees, it is vital that values *do not* appear out of thin air [24]. To preclude this, the JMM adds a causality condition: Reads that see concurrent writes must be committed, i.e., there must be a justifying execution that writes the same value, but the read action sees a write that happens before it. We omit the technical details in the presentation, as they are not relevant for understanding this work, but we have formalised them similarly to previous work [4,11]. In the above example, causality forbids r1 == 42 because no execution can produce the value 42 without performing both reads from concurrent writes. 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.

This is where initialisations come into play. The JMM assumes that all locations are initialised to their default value at the start of the execution. By definition, these initialisations happen before any other action. Thus, there is always at least one suitable write that happens before a given read, which ensures that such a basis for justifying executions exists.

The requirement that the JMM initialises heap locations at the start (instead of when the location is allocated) has been one of the main complications in our DRF proof – which previous formalisations have omitted. Since initialisation actions originate from dynamic allocation, we must consider complete executions, which may be infinite, instead of finite prefixes [4] – at least for the single-thread semantics. Consider, e.g., the program and one of its (legal) executions in Fig. 2. Note that the initialisation for the field f of the object created in l. 3 at location l happens before all other actions although the single-thread semantics executes

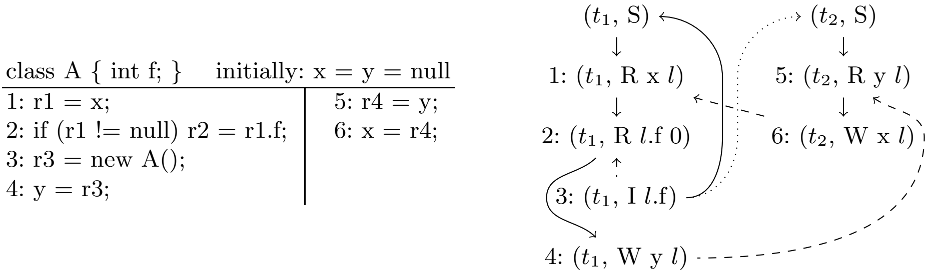


Fig. 2. Program with a legal execution in which a read sees the initialisation which occurs later in the program text

it only after ll. 1 and 2. Suppose we take the prefix of this execution up to l. 2. If (t_1, I l.f) is not part of the prefix, the prefix is an ill-formed execution because l. 2 sees no write. Hence, we must include the initialisation actions in prefixes. As the single-thread semantics produces initialisation actions only at allocations, we must run the program to completion, because we cannot decide at intermediate states whether we have collected all initialisation actions. Thus, our formalisation must deal with infinite executions.

1.2 Related Work

A lot of work has been devoted to hardware MMs, see [1] for an overview. Here, we focus on programming language MMs.

Huisman and Petri [11] have formalised the JMM and the proof of the DRF guarantee in 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 [4] have formalised parts of the JMM relevant for the DRF guarantee and proved the latter in Isabelle/HOL — which we have found very helpful in extending the DRF guarantee proof. Since they omit dynamic allocation, they need to consider only finite prefixes of executions, which considerably simplifies their proofs, as they do not need to assume that sequentially consistent continuations of executions exist. They do not provide an intra-thread semantics; 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 §§2.1, 2.2).

For a kernel language, Cenciarelli et al. [9] define an interleaving small-step semantics that generates configuration structures of actions which an axiomatic theory constrains. On paper, they show that they only generate behaviours that the JMM allows, but it is unknown if they produce every allowed behaviour.

Torlak et al. [29] developed a model checker for axiomatic memory models. Using whole-program analysis, they derive JMM executions from small Java programs that are restricted to a small (finite) number of heap locations and finite state; loops

are unrolled. Thus, their algorithm can compute all actions and memory allocations in advance. They focus on checking small test cases rather than providing a full semantics and proofs.

Jagadeesan et al. [13] define an operational semantics for weak MMs with speculative computations similar to the JMM. Instead of validating executions a posteriori, their semantics explicitly encodes permitted reorderings and speculation. Yet, their model is neither machine-checked nor comparable to the JMM for programs with data races and synchronisation.

Boyland [8] 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. 1,2, whereas our semantics deals with the full JMM.

The recent standard C++11 [12] considers programs with data races ill-formed and assigns undefined semantics to them, but it offers finer shades of synchronisation than Java. Boehm and Adve [7] describe the MM 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 accesses adjacent in any interleaving. For the JMM, this equivalence does not hold since threads can communicate without introducing happens-before relationships (§2.1). Batty et al. [6] have formalised the MM with a focus on rigorously defining the semantics, but do not report on any proofs.

Ševčík et al. [27] have verified the CompCert compiler backend with respect to the formal MM for x86 processors by Sewell et al. [28], which is the first formal correctness proof for an optimising compiler backend w.r.t. a weak MM. They expose the x86-TSO model in the programming language, which is considerably stronger than the JMM and also provides a DRF guarantee.

2 From Sequential Java to the Java Memory Model

This section introduces JinjaThreads (§2.1) and connects it (§2.2) to our JMM formalisation (§2.3). We discuss deviations from and suggestions for the original JMM in §2.4.

2.1 Single-Thread Semantics

JinjaThreads is a complex model of Java that supports a broad spectrum of concepts: local variables, objects and fields, inheritance, dynamic dispatch, recursion, arrays and exception handling; for details see [14,16,17]. It uses a stack of small-step semantics to give meaning to programs (Fig. 3). As source code and bytecode share the same program structure except for method bodies, they share most of the levels. The stack falls into two parts: the multithreaded semantics at levels 4 to 6, which we defer to §2.2, and the sequential small-step semantics at levels 1 to 3. Source code and bytecode differ only on level 2, which defines the

semantics of the language primitives. For bytecode, this level consists of three sub-levels 2a through 2c.

Before we look at the individual levels, we discuss the general form $t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle$ of the single-thread semantics. Local states of thread t are denoted by x and x', and T, T' are the (global) type information that all threads share (see §2.2). The multithreaded semantics abstracts from the concrete steps of the single-thread semantics and uses only lists as of inter-thread actions. Reductions can generate multiple actions in one step. When the `wait` method suspends the thread to the wait set, e.g., it also tests for the monitor lock and for not being interrupted. Reductions without actions, i.e., $as = []$, are called *τ-moves.*

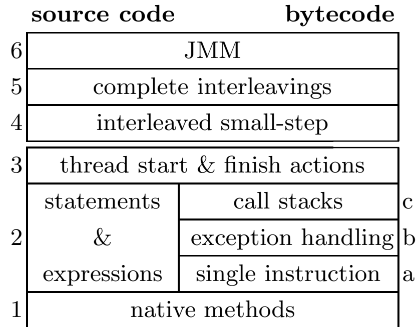


Fig. 3. Stack of JinjaThreads source code and bytecode semantics

Unfortunately, the actions from §1.1 are insufficient to correctly implement the JLS, because the JLS (and the Java API) introduce other communication channels between threads. Consider, e.g., the following program in which two threads race for spawning the same thread:

initially: x = null;

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

(P1)

Suppose both reads in ll. 3 and 5 see the write at l.2. Then, either l. 4 or l. 6 must throw an `IllegalThreadStateException`, 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 action 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.*[2]

Therefore, we introduce the following additional inter-thread actions: (i) Detect that a thread has already been started (TS), (ii) wait in a monitor (Wait), (iii) notification (N, NA), (iv) clearing an interrupt (CIr), (v) testing for a thread not being interrupted (NIrd), and (vi) test whether the current thread does (not) hold a lock (HL, NL). Technically, the last group is only a convenience, because this way, a thread need not remember in its local state which locks it holds.

Now, let us return to the single-thread semantics. Most of Java concurrency hides in (native) library methods, in particular in classes `Thread` and `Object`.

[2] For `start`, the JMM specifies synchronisation only between a successful call and the first action of the spawned thread [10, §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.

$$t \vdash \langle (ad, \texttt{start}),\, T\rangle \xrightarrow{[\text{Sp}\ ad\ (method\ C\ \ run)]} \langle unit,\, T\rangle \qquad \textsc{Spawn}$$

$$t \vdash \langle (ad, \texttt{start}),\, T\rangle \xrightarrow{[\text{TS}\ ad]} \langle throw\ \texttt{IllegalThreadStateException},\, T\rangle \qquad \textsc{SpawnF}$$

Fig. 4. Semantics of the methods `start` and `isInterrupted` for class Thread. All rules have the preconditions $typeof\ T\ ad = \lfloor C \rfloor$ and $C :\leq Thread$, which have been omitted.

Hence, we provide at the bottom of the stack a hard-wired semantics for some native methods. We focus on concurrency-related methods such as `wait`, `notify`, `notifyAll` in `Object` or `start`, `join`, `interrupt` in `Thread`, but also include ordinary methods like `hashCode`.

Figure 4 gives a flavour of the semantics rules; the full definition can be found online [18]. If address *ad* has type C (notation $typeof\ T\ ad = \lfloor C \rfloor$) and C is a subtype of *Thread* (notation $C :\leq Thread$), calling `start` on *ad* either (i) successfully spawns the new thread *ad* whose initial state becomes C's `run` method Spawn, or (ii) fails with an `IllegalThreadStateException` SpawnF. The single-thread semantics is non-deterministic here, but the reductions generate different actions; the concurrent semantics ensures which of these actions can actually happen. In particular, the new action TS in SpawnF is necessary.

The second level specifies the semantics for the language primitives. In source code, this is a standard small-step semantics. In bytecode, sub-level 2a executes single instructions, calls to native methods are delegated to level 1. Sub-level 2b adds exception handling, 2c joins everything together into the semantics of a single thread.

All actions originate on level 2 except for thread start and finish actions and those generated by native methods. For example, `synchronized` blocks or `monitorenter` and `monitorexit` instructions generate lock and unlock actions, field accesses via `getfield` and `putfield` produce read and write actions. Field read expressions and instructions such as `getfield` non-deterministically read any value, irrespective of the dynamic location type. Primitives like `instanceof` that do not produce any action yield τ-moves. The shared type information grows when objects and arrays are allocated and remains unchanged otherwise.

On level 3, we add artificial start and finish actions to each thread. This ensures that the start action of a thread precedes all its other actions.

The semantics on level 3 defines the sequential small-step semantics, on which levels 4 to 6 build. In the remainder, we use $\rightarrow$ to refer to either source code or bytecode semantics of level 3.

2.2 Complete Interleavings

In this section, we build the multithreaded semantics on top of the sequential (levels 4 and 5 in Fig. 3).

The JMM is only concerned about values, not types and array lengths. 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 [10, §17.4.5].[3] However, since objects and arrays are dynamically allocated, the type of an object at a given address (or the type and length of an array at that address) is determined only after allocation. For types and array lengths, we adopt sequential consistency, i.e., allocations immediately update type information of all threads. This directly solves a problem pointed out by Aspinall and Ševčík [4]: What does it mean for an address being fresh for memory allocation?

Technically, the global type information T is like a global shared state that contains only type information and array lengths, but no data values. Then, an address is fresh in state T iff T contains no type information for it. Java's type safety then ensures that it has not yet been used in any thread, so we can safely use it when allocating new memory.

Threads also communicate via types and array lengths – unnoticed by the JMM. For example,

initially: x = 0; y = null;

1: r1 = x; 2: r2 = (r1 == 0 ? new A() : new B()); 3: y = r2;	4: x = 1;	5: r3 = y; 6: r4 = r3.f();

(P2)

Suppose that classes A and B inherit from an interface I which declares a method f() and that their objects may be allocated at the same address. Then, dynamic dispatch at l. 6 tells the thread on the right about the left thread's local variable r1. However, from the JMM point of view, the thread on the right only reads an address (in fact the same value in both cases), but behaves differently. An analogous problem occurs if we use array lengths instead of types or declare x and y as volatile.

Hence, threads cannot execute in isolation, as the JMM suggests. Instead, we compute their interleavings with type information as shared state, which guarantees sequential consistency. Our interleaving semantics also takes care of mutual exclusion for locks and manages the monitor wait sets and notifications.

In the rest of this section, we formally define the interleaved semantics (level 4) and complete interleavings (level 5). Remember that we must consider complete interleavings because the JMM treats initialisations specially (see §1.1). Since threads in the single-thread semantics can only communicate via type information or inter-thread actions, the following is independent of the concrete single-thread semantics.

A *thread pool* ts is a finite map from a thread's ID to its local state x, the multiset L of locks it holds, its interrupt status i, and its wait set status w (none, waiting in a monitor, notified, interrupted, reacquiring the locks). We define the *interleaved small-step semantics* $\langle ts, T\rangle \xRightarrow{(t,\, as)} \langle ts', T'\rangle$ as

$$\frac{ts(t) = \lfloor(x, L, i, w)\rfloor \qquad t \vdash \langle x, T\rangle \xrightarrow{as} \langle x', T'\rangle \qquad ts \vdash_t as\surd \qquad ts \xmapsto{t,\, as,\, x'} ts'}{\langle ts, T\rangle \xRightarrow{(t,\, as)} \langle ts', T'\rangle}$$

[3] Although the JLS specifies that every array has a final field `length` [10, §6.4.5] that stores its length, the JMM treats array lengths specially [10, §17.4.5].

where $\lfloor _ \rfloor$ denotes definedness of a finite map. The predicate $ts \vdash_t as\surd$ checks whether t may perform all actions in as in the current system state ts. It implements the wait-notify and interruption mechanism, and ensures mutual exclusion for locks and that each thread is spawned at most once. $ts \xmapsto{t,\, as,\, x'} ts'$ inserts all threads spawned in as into ts and updates t's locks, wait set status, and local state to x', which yields ts'. For details, see [16,17].

A *complete interleaving* E is a potentially infinite list of pairs of thread ID and inter-thread action. The relation $\langle ts,\ T\rangle \Downarrow E$ characterises all complete interleavings E that start in $\langle ts,\ T\rangle$, which we define as

$$\langle ts,\ T\rangle \Downarrow E :\Leftrightarrow \exists E'.\ \langle ts,\ T\rangle \downarrow E' \wedge E = concat(E') \quad (1)$$

where $concat(E')$ concatenates all lists in E' and $\langle ts,\ T\rangle \downarrow E'$ (defined coinductively)[4] collects the list of lists of inter-thread actions.

$$\frac{\langle ts,\, T\rangle \not\Rightarrow}{\langle ts,\ T\rangle \downarrow []}\ \textsc{Stop} \qquad \frac{\langle ts,\, T\rangle \xRightarrow{(t,\, as)} \langle ts',\, T'\rangle \qquad \langle ts',\ T'\rangle \downarrow E'}{\langle ts,\ T\rangle \downarrow obs_t(as) : E'}\ \textsc{Step}$$

where $\langle _,\ _\rangle \not\Rightarrow$ characterises stuck configurations and $obs_t(as)$ collects all JMM inter-thread actions in as (as defined in §1.1) and pairs them with the thread ID t. That is, it removes the additional actions from above, as they are irrelevant for the JMM.

Note that the detour via a list of action lists is necessary. If we had defined $\langle ts,\ T\rangle \Downarrow E$ directly with the above coinductive rules Stop and Step (i.e., prepending $obs_t(as)$ to E instead of consing), we could have derived every trace E for a state $\langle ts,\, T\rangle$ that can perform an infinite sequence of τ-moves, because $obs_t(as) = []$ for all τ-moves. Our approach works fine since $obs_t(as) : E$ is productive and concatenating the infinite list of empty lists yields [].

The initial state $\langle ts_0,\, T_0\rangle$ for a program is specified by a class, a method name, and the list of parameters it takes. Its thread pool ts_0 consists of a single thread t_0 that holds no locks and is about to execute the specified method with the given parameters. T_0 has pre-allocated the t_0 `Thread` object and certain system exceptions. The list as_0 of start-up actions contains t_0's start action and initialisations for the fields of the pre-allocated objects.

For the JMM, we identify a program with the set $\mathcal{E}$ of complete interleavings that start in the initial state, prefixed with as_0. Formally:

$$\mathcal{E} = \{ obs_{t_0}(as_0) +\!+ E \mid \langle ts_0,\ T_0\rangle \Downarrow E \}$$

where ++ concatenates two lists. $\mathcal{E}$ contains many ill-formed executions, because read operations may read arbitrary values (see §2.1), even not type-conforming ones that no write operation of the program can ever produce. Since they have no write-seen function, the JMM on level 6 discards them.

[4] We use double bars to distinguish coinductive definitions from inductive ones.

2.3 The Java Memory Model

In this section, we formally derive the orders of the JMM (level 6) from a complete interleaving $E \in \mathcal{E}$. For the intuition behind them, see [21,10,11,4]. The JMM notions of well-formed and legal executions are standard [4,10], we only explain them informally; [19] shows their formal definitions.

Since an action can occur multiple times in E, we use the index in E to assign a unique identifier to an action. In the following, we identify an action with its index, i.e., $\mathcal{A}_E = \{a \in \mathbb{N} \mid a < |E|\}$ denotes the set of actions for E. This already provides the *induced total order* $\leq_E = \leq|_{\mathcal{A}_E}$ over $\mathcal{A}_E$, where $R|_A$ restricts the binary relation R to elements from A. Since the JMM requires initialisation actions[5] to be ordered before the threads' start actions, we introduce the (total) *execution order* $\leq^E_{\text{eo}}$ on $\mathcal{A}_E$:

$$a \leq^E_{\text{eo}} a' :\Leftrightarrow \mathsf{if}\ \mathit{init}_E\, a\ \mathsf{then}\ \neg\, \mathit{init}_E\, a' \vee a \leq_E a'\ \mathsf{else}\ \neg\, \mathit{init}_E\, a' \wedge a \leq_E a'$$

where $\mathit{init}_E\, a$ predicates that a is an initialisation action in E.

The *program order* $\leq^E_{\text{po}}$ restricts $\leq^E_{\text{eo}}$ to actions of the same thread. The *synchronisation order* $\leq^E_{\text{so}}$ restricts $\leq^E_{\text{eo}}$ to synchronisation actions. *Synchronisation actions* are all initialisation actions, reads from and writes to volatile locations, locking and unlocking, thread spawns and joins, thread start and finish actions, and the interruption actions Ir and Ird. The *synchronises-with order* $\leq^E_{\text{sw}}$ restricts $\leq^E_{\text{so}}$ to release-acquire pairs of actions. (a, a') is a *release-acquire pair* iff

1. a unlocks monitor m and a' locks m,
2. a writes to a volatile location that a' reads,
3. a spawns a thread whose start action is a',
4. a is a thread's finish action on which a' joins,
5. a is an initialisation action and a' is a thread start action, or
6. a interrupts a thread t and a' observes that t has been interrupted.

The *happens-before order* $\leq^E_{\text{hb}}$ is the transitive closure of $\leq^E_{\text{po}}$ and $\leq^E_{\text{sw}}$. $\mathcal{V}_E\, a$ denotes the value that the write action $a \in \mathcal{A}_E$ writes – initialisation actions write default values (0, false, or null, resp.); for normal write actions, E contains the value written.

An *execution* (E, ws) consists of a complete interleaving E and a *write-seen function* ws on $\mathcal{A}_E$ that assigns to every read action in $\mathcal{A}_E$ the write action it sees. This yields the JMM notion of an execution [10, §17.4.6] as $(\mathcal{E}, \mathcal{A}_E, \leq^E_{\text{po}}, \leq^E_{\text{so}}, ws, \mathcal{V}_E, \leq^E_{\text{sw}}, \leq^E_{\text{hb}})$.

An execution is *well-formed* (written $\vdash (E, ws)\surd$) iff every thread has a thread start action that $\leq_E$-precedes its other actions except for initialisation actions (denoted $E\surd_{\text{start}}$) and ws is a proper write-seen functions for all reads in E as specified by the JMM well-formedness conditions 1 (each read sees a write to the same location), 4 ($\leq^E_{\text{hb}}$ consistency) and 5 ($\leq^E_{\text{so}}$ consistency for volatiles) in [10, §17.4.7]. (E, ws) meets conditions 2 ($\leq^E_{\text{hb}}$ is a partial order) and 3 (intra-thread consistency) by construction. E is *well-formed* iff $\vdash (E, ws)\surd$ for some ws.

[5] When the single-thread semantics allocates memory, it produces initialisation actions for the new locations. This records that the executing thread has generated them.

A *legal* execution is a well-formed execution (E, ws) that is justified by a sequence of justifying executions (E_i, ws_i). As §1.1 explains, it serves to ban values appearing out of thin air. The concrete definition is tedious, but uninteresting for the rest of this work. It can be found in [19].

2.4 Discussion of Our JMM Formalisation

Our formalisation shows how to connect a Java semantics with the JMM, which has been missing in the literature [4,9,11]. The main insight is that action traces of isolated threads do not suffice to obey the JLS and Java API. The examples (P1) and (P2) present hidden communication channels between threads that the JMM inter-thread actions do not capture – although they only use Java features that the JMM mentions. To expose these channels, we have introduced new actions – and our semantics shows that they suffice for the features that JinjaThreads models except for type information and array lengths. We conjecture that further actions for allocations would also lift this restriction (see below).

Most obviously, the JMM misses actions for thread interrupts. It predicates that `Thread.interrupt` "synchronises-with the point where any other thread [...] determines that [the thread] has been interrupted" [10, §17.4.4], but there are no designated actions for neither thread interruption nor "that point". Hence, we have added the synchronisation actions Ir and Ird (§1.1), and their duals for non-interruption CIr and NIrd (§2.1). Similarly, the API of class `Thread` requires new actions to query a thread's state, e.g., TS predicates that it has been started. Previous JMM formalisations [4,8,11] did without these new actions, because they omitted interruption and wait sets, but a realistic formalisation cannot.

The interesting question is which of these new actions should participate in synchronisation and happens-before order. We follow the original JMM in that only Ir synchronises with Ird; $obs_t(_)$ removes the others. In particular, the others do not synchronise with any action and need not be committed or justified. Hence, they do not affect the writes that a read may see. We consider this sensible, because we have found it very hard to construct programs that can exploit such additional synchronisation to avoid data races (see, e.g., [19, P3]). Typically, other schedules exhibit races in such programs. Counter-intuitively, this may also disallow some behaviours, since adding synchronisation may allow new behaviours for programs with data races [3,21].

We do not use actions to broadcast type information, but interleave the execution to obtain sequential consistency for types. This also solves the problem of finding a fresh address for memory allocation, as the shared type information stores which addresses are fresh. Although complete interleavings introduce a global notion of time, we do not use it to constrain the write that each read sees, because the JMM order relations abstract from it.

However, we see two approaches to avoid the interleaving. One could include actions for producing and querying type information for locations and array lengths. In a well-formed execution, these actions have to be matched, but they do not interact with other thread actions. Alternatively, one could partition the address space by type and array length like in [13]. Then, however, every read

of a reference value would implicitly transfer all type information associated with it, which is unrealistic for implementations. In either approach, allocation actions subsume initialisation actions such that allocation returns an arbitrary address and the JMM ensures that every address is allocated at most once.

There are also a few technical changes to the JMM that we briefly review: First, for the DRF guarantee, *all* initialisation actions must be synchronisation actions, not only those for volatile locations, which follows Aspinall and Ševčík [4]. In contrast to them [4], we do not need a special initialisation thread (which might run infinitely in the case of an infinite execution), but assign initialisation actions the thread's ID which created the object. This change is relevant for the final field semantics extension to the JMM, which requires to know which thread created which object [10, §17.5.1].

Second, happens-before for the `wait` method arises not only from the associated unlock and lock actions [10, §17.4.5], but also calling `interrupt` on the waiting thread synchronises with throwing the `InterruptedException`. When a thread in a wait set is both interrupted and notified, our semantics always respects happens-before, although the JLS does not require this [10, §17.8.1].

Third, we do not model thread divergence actions. The JMM introduces them to "model how a thread may cause all other threads to stall and fail to make progress" [10, §17.4.2]. Our construction achieves the same via the coinductive trace definition (Stop, Step), which then gets filtered for τ-moves (1). Thus, it handles terminating, non-terminating and diverging executions uniformly.

Finally, JinjaThreads models neither final fields nor garbage collection and finalisation. Hence, we do not model that part of the JMM [10, §17.5].

3 The Data Race Freedom Guarantee

The JMM promises that correctly synchronised programs exhibit only sequentially consistent behaviours. First, we recapitulate the definitions and identify the assumptions of this guarantee (§3.1). Next, we show that source code and bytecode indeed satisfy these assumptions (§3.2); the proofs can be found [19]. In §3.3, we discuss our formalisation and its implications.

3.1 The DRF Guarantee

In this section, we formally state the DRF guarantee and prove it. Two actions of an execution are *conflicting* if they are read or write actions to the same location with at least one being a write. Two conflicting actions constitute a *data race* if they are not ordered by happens-before.[6]

[6] As the happens-before order approximates time, it serves to identify data races. More intuitively, two conflicting actions race iff they can happen "concurrently" in an execution, i.e., they are adjacent in an interleaving and the location is not marked volatile. For simple models of happens-before, these are equivalent [7], but not for Java with implicit communication channels between threads, see, e.g., [19, P3]. Still, every data race in the latter sense is also one in the happens-before sense.

An execution (E, ws) is *sequentially consistent (SC)* iff every read action $a \in \mathcal{A}_E$ sees the most recent write action, i.e., $ws(a) \leq_{\mathrm{eo}}^E a$, and $a' \leq_{\mathrm{eo}}^E ws(a)$ or $a \leq_{\mathrm{eo}}^E a'$ for all write actions a' to the location that a reads from.[7]

The program $\mathcal{E}$ is *correctly synchronised* (data race free, DRF) iff no SC execution in $\mathcal{E}$ contains a data race. Formally: Whenever $E \in \mathcal{E}, \vdash (E, ws)\surd$ and (E, ws) is SC, then $a \leq_{\mathrm{hb}}^E a'$ or $a' \leq_{\mathrm{hb}}^E a$ for all conflicting actions $a, a' \in \mathcal{A}_E$.

For the DRF guarantee, it is important that we only have to check that SC executions do not contain a data race. Otherwise, it would fail its purpose because the programmer would have to understand the whole JMM to see whether his program is correctly synchronised and the DRF guarantee applies to it.

Our proof of the DRF guarantee (Thm. 1) adapts the others' [21,4,11] to deal with memory allocation and initialisations (see §3.3 for a discussion). The key idea in all of them is that in a DRF program, a well-formed execution (E, ws) is SC if every read sees a write that happens before it (Lem. 1) – which includes program order. Then, the legality constraints ensure that all legal executions are SC.

Lemma 1. *Let $\mathcal{E}$ be correctly synchronised, $E \in \mathcal{E}$ such that $\vdash (E, ws)\surd$. If $ws(a) \leq_{hb}^E a$ for every read a in $\mathcal{A}_E$, then (E, ws) is sequentially consistent.*

To exploit correct synchronisation in a proof of Lem. 1 by contradiction, one first obtains a SC execution (E', ws') from (E, ws) as follows: E' starts like E until the first non-SC read a in E and continues SC from there on. Then, it suffices to find a data race between a, $ws(a)$, and $ws'(a)$ in E', and for this, we use Lem. 2 to transfer happens-before relationships between E and E' on their common prefix.

Lemma 2 ($\leq_{\mathbf{hb}}$-prefix lemma). *Let E and E' be two complete interleavings such that their first n actions differ only in the values read or written, and let $a, a' < n$. If $E'\surd_{start}$ and $a \leq_{hb}^E a'$, then $a \leq_{hb}^{E'} a'$.*

Theorem 1 (DRF guarantee). *If the program P is correctly synchronised and (E, ws) a legal execution, then (E, ws) is sequentially consistent.*

The proof closely follows [4, Thm. 1], it uses Lem. 1. Both Thm. 1 and Lem. 1 implicitly rely on two assumptions about $\mathcal{E}$:

- A1 For every sequentially-consistent prefix of a well-formed execution (E, ws) with $E \in \mathcal{E}$, there is a well-formed complete interleaving $E' \in \mathcal{E}$ with the same prefix and a write seen-function ws' such that (E', ws') is SC. If E immediately continues with a read after the prefix, E' also continues with a read from the same location.
- A2 Every execution initialises every location at most once.

[7] The JMM only requires that $\leq_{\mathrm{po}}$ is extended to a total order over all actions to determine most recent writes [10, §17.4.3]. Aspinall and Ševčík [4] showed that, to respect mutual exclusivity of locks, the total order must also extend $\leq_{\mathrm{so}}$. Our execution order $\leq_{\mathrm{eo}}$ extends both by construction.

The first assumption ensures that E' as required in the proof of Lem. 1 does exist, the second is a standard well-formedness condition. In §3.2, we prove that JinjaThreads source code and bytecode satisfy these by explicitly constructing SC executions. Moreover, Lem. 2 requires that *all* initialisation actions synchronise with thread start actions [19, P4], i.e., they are synchronisation actions.

3.2 Sequentially Consistent Completions

In the previous section, we have shown the DRF guarantee under two assumptions on the set $\mathcal{E}$ of complete interleavings. Now, we discharge them for source code and bytecode by descending the stack of semantics (Fig. 3) and adapting the assumptions. They act like an interface between the levels and ensure that we can share the proofs for all levels that source code and bytecode share.

We start with complete interleavings. The JMM definition of SC is not amenable to the coinductive definition of $\langle _, _\rangle \Downarrow _$ as it relies on the notions of write-seen function and most recent write, which are only defined for complete interleavings. Therefore, we introduce a coinductive version of SC.

Let h denote a snapshot of a sequentially consistent heap, i.e., a finite map from locations to values. The function $mrw(h, a)$ updates the heap h if a is a write or initialisation action, else leaves h unchanged. The function $mrws$ folds mrw over action lists. An action list as is *sequentially consistent* (SC') for the heap h (denoted $h \vdash as\surd_{sc}$) iff

$$\overline{\overline{h \vdash [] \surd_{sc}}} \qquad \frac{mrw(h,a) \vdash as\surd_{sc} \qquad a = \mathrm{R}\ l\ v \implies h(l) = \lfloor v \rfloor}{h \vdash a : as\surd_{sc}}$$

i.e., the empty list is SC' for all heaps, and $a : as$ is SC' for h iff as is SC' for the updated heap $mrw(h, a)$ and if a reads the value v from location l, then the heap h must store v at l.

The next theorem shows that $\emptyset \vdash _\surd_{sc}$ adequately models SC, where $\emptyset$ denotes the empty map. Thus, we can use coinduction to show an execution being SC.

Theorem 2. *Let initialisations $\leq_E$-precede reads and $E\surd_{start}$. Then, $\emptyset \vdash E\surd_{sc}$ iff there is a ws such that $\vdash (E, ws)\surd$ and (E, ws) is SC.*

This equivalence holds only if the initialisation of any location l occurs before the first read from l in the complete interleaving. For example, the execution $[(t_1, \mathrm{S}), (t_1, \mathrm{R}\ l.\mathrm{x}\ 0), (t_1, \mathrm{I}\ l.\mathrm{x}\ 0)]$ is SC for $ws(t_1, \mathrm{R}\ l.\mathrm{x}\ 0) = (t_1, \mathrm{I}\ l.\mathrm{x}\ 0)$, but not SC'. The problem is real: Figure 2 shows a (non-SC) execution of a type-correct program that violates this assumption. In order to exploit this equivalence, we must show that initialisations $\leq_E$-precede reads in SC prefixes of complete interleaving (see below).

Prior to this, we construct a sequentially consistent completion $scc(\langle ts, T\rangle, h)$ that starts with a thread pool ts, global type information T, and a heap h. We define scc by corecursion as follows, where ε denotes Hilbert's choice operator.

$scc(\langle ts,\, T\rangle, h) :=$
$\textsf{if } \langle ts,\, T\rangle \not\Rightarrow \textsf{ then } []$
$\textsf{else let } (t, as, ts', T') = \varepsilon(t, as, ts', T').\ \langle ts,\, T\rangle \xRightarrow{(t,\ as)} \langle ts',\, T'\rangle \wedge h \vdash as\surd_{sc}$
$\quad \textsf{in } \ obs_t(as) : scc(\langle ts',\, T'\rangle, mrws(h, as))$

In order to prove anything about $scc(\langle ts,\, T\rangle, h)$, we must make sure that the predicate to the ε-operator is satisfiable for all reachable configurations. Hence, we presume for now that the interleaving semantics satisfies the *cut-and-update property* (C&U), namely whenever $\langle ts,\, T\rangle \xRightarrow{(t,as')} \langle ts',\, T'\rangle$ and $wf(\langle ts,\, T\rangle, h)$, then there are as'', ts'', and T'' such that (i) $\langle ts,\, T\rangle \xRightarrow{(t,as'')} \langle ts'',\, T''\rangle$, (ii) $h \vdash as''\surd_{sc}$, and (iii) $h \vdash as' \approx as''$. The predicate wf ensures well-formedness of the state and conformance of heap; for source code and bytecode, we define wf below and prove that their semantics satisfy C&U. Conditions (i) and (ii) predicate that non-stuck states always have a reduction with actions as that are SC' w.r.t. the current heap h; they suffice to prove that scc does compute an SC' interleaving (Lem. 3). Condition (iii) denotes that as' and as'' consist of the same actions upto the first SC' inconsistent read in as' and as'' continues with a read from the same location. With this condition, given a complete interleaving that is SC' up to a read r, we can cut the interleaving after r, change r to read the most recent value, and continue the interleaving SC'.

Let us further assume that $wf(\langle ts,\, T\rangle, h)$ holds for the initial state (ts_0, T_0) with the initial heap $h_0 := mrws(\emptyset, as_0)$, and is preserved by all SC' reductions. Then, scc computes an SC' execution (Lem. 3). By the equivalence of SC and SC' (Thm. 2), we can then discharge the main assumption of the DRF proof (Thm. 3).

Lemma 3.
If $wf(\langle ts,\, T\rangle, h)$, then $\langle ts,\ T\rangle \!\downarrow\! scc(\langle ts,\, T\rangle, h)$ and $h \vdash concat(scc(\langle ts,\, T\rangle, h))\surd_{sc}$.

Theorem 3 (SC completion). *Let $E \in \mathcal{E}$, $\vdash (E, ws)\surd$, (E, ws) be SC up to a read action $(t, R\ l\ v)$, say $E = E_1 +\!\!+ (t, R\ l\ v) : E_2$ with $ws(r)$ being the most recent write for all reads $r \in \mathcal{A}_{E_1}$. Then, there are E_3, v', and ws' such that $E^* := E_1 +\!\!+ (t, R\ l\ v') : E_3 \in \mathcal{E}$, $\vdash (E^*, ws')\surd$, and (E^*, ws') is SC.*

We have now replaced the assumptions A1 and A2 of §3.1 by the following, which are simpler and no longer refer to JMM notions.

- B1 Every execution initialises every location at most once.
- B2 If a complete interleaving has an SC' prefix as followed by a read from l, as must initialise l.
- B3 wf is preserved by SC' reductions and $wf(\langle ts_0,\, T_0\rangle, h_0)$ holds.
- B4 The interleaving semantics satisfies C&U.

Next, we tackle these proof obligations. They naturally translate to the levels below the interleaving semantics, so we do not expand on them in detail. The actual proofs decompose the semantics on levels 4 down to 1, perform induction on the

semantics (source code) or case analysis on the individual instructions (bytecode), resp., and lift everything back to level 4. Here, we only present the main arguments.

For B1, remember that only memory allocations generate initialisation actions. 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 inter-thread actions, as initialisation actions refer to the address.

For B2, not only must we show that the program cannot make up arbitrary addresses, but also that it accesses only the declared fields of objects. To that end, we define the well-formedness predicate $\mathit{wf}(\langle ts, T\rangle, h)$ to denote that

(i) for all allocated addresses a, T contains type information and h contains type-conforming values for all fields and array cells of a,
(ii) all addresses in thread-local states of ts and in h's range are allocated, and
(iii) all thread-local states in ts are language-specifically well-formed.

For source code, the latter states that all values in the local store are of correct type and the statement is runtime-typeable. For bytecode, the operand stack and registers must conform to the well-typing as computed by the bytecode verifier. Type correctness ensures that the semantics stays within the safe state space, e.g., it does not get unexpectedly stuck or yields undefined behaviour about which nothing can be proven.

Preservation for *wf* (assumption B3) relies on the JinjaThreads type safety proofs [16,17,14]. The subject reduction proofs require that reads only return type-conforming values. This holds because the semantics correctly keeps track of all reads in the inter-thread actions, which are by assumption SC', and the heap contains only type-conforming values. By the type safety proofs, all values written to the memory are type-conforming, too. Moreover, we show that the single-thread semantics cannot generate new addresses other than via memory allocation. Hence, *wf* ensures that addresses cannot appear out of thin air in an SC' execution. For the initial state, $\mathit{wf}(\langle ts_0, T_0\rangle, h_0)$ holds by construction.

From this, B2 follows. By preservation, *wf* holds for the state after the prefix as. Hence, type safety ensures that the read accesses an allocated location.

Finally, showing that the semantics satisfies C&U (assumption B4) is tedious, but uninteresting because reads may return arbitrary values.

Theorem 4. *The JMM DRF guarantee holds for source code and bytecode.*

This follows from Thm. 1, 3 by the above argument that their assumptions hold.

3.3 Discussion

The DRF guarantee for Java (§3.1) has been formalised before [4,11] – in fact, we employ the same key ideas in §3.1. The novel aspects are that (i) our JMM formalisation covers dynamic allocation with explicit initialisation actions and infinite executions, and (ii) we identify the assumptions of the DRF guarantee on the sequential semantics and discharge them for source code and bytecode. The key insights are the following:

1. Our new actions and different kinds of synchronisation do not affect the DRF proof. This suggests that other means of synchronisation that we do not cover, e.g., atomics in `java.util.concurrent`, do not affect it either.
2. We must find better ways to handle initialisations, as the JMM way severely complicates the proofs.
3. Our proofs show that the treatment of initialisations is irrelevant for the DRF guarantee, i.e., we are not constrained when searching for better ways.

Insight 3 a posteriori justifies Aspinall's and Ševčík's simpler approach of considering finite prefixes for the purpose of formalising the DRF guarantee. However, it is still insufficient when dealing with the full JMM. For example, the JMM allows the execution in Fig. 2, but not some of its prefixes.

Similarly, our DRF proof shows that it would be safe to restrict read operations to type-conforming values – for correctly synchronised programs. Subject reduction and preservation proofs would become significantly easier. However, it would disallow some legal executions of programs with data races such as Fig. 2.

Technical Considerations. Our work in §3.1 differs from [4,11,21] mainly in the proof of the key Lem. 1. We adapt the others' in two respects to deal with initialisation actions. First, the others topologically sort $\leq_{\mathrm{po}}^{E}$ [21] or $\leq_{\mathrm{hb}}$ [4,11] first to obtain $\leq_{\mathrm{eo}}$, and then take $\{a \mid a \leq_{\mathrm{eo}}^{E} r\}$ as the prefix for the SC execution. Instead of through sorting, we obtain the induced total order $\leq_E$ from the complete interleaving, which does not move initialisation actions to the program start.

Second, Manson et al. [21] and Huisman and Petri [11] require a sequentially consistent completion E'; so do we. However, the former ignore that different initialisation actions in the suffix might change the $\leq_{\mathrm{hb}}$ relation on the prefix. The latter note this problem, but add an axiom that $\leq_{\mathrm{hb}}$ remain unchanged. We solve the issue by using $\leq_E$ instead of $\leq_{\mathrm{eo}}^{E}$. Hence, $\leq_{\mathrm{hb}}$ on the prefix becomes independent of later initialisations (Lem. 2). Aspinall and Ševčík [4] completely avoid it by restricting their model to finite prefixes of executions – which causes problems when dynamic allocation creates initialisations (§1.1).

Initialisations also complicate the construction of sequentially consistent completions. We failed to construct them directly, as due to the special treatment of initialisations, ill-formed programs might not have such, see, e.g., [19, P5]. Hence, we would need appropriate constraints that the semantics preserves, but the JMM notion of execution is unsuitable for preservation proofs. Instead, we proved that sequential consistency w.r.t. happens-before is the same as for interleaving semantics – if initialisations do not interfere (Thm. 2).[8] Being operational, interleaving semantics is much more amenable to reduction invariants and their preservation proofs than the JMM. While it is still challenging to show properties about *scc*, most proofs follow the well-known pattern of preservation.

[8] Interestingly, Batty et al. [5, §4] found that initialisations of atomics cause problems in the DRF proof for C++11, too.

Faithfulness of the Semantics. Aspinall and Ševčík [4] suggested to weaken legality to enable more optimisation without sacrificing the DRF guarantee. Since our proof follows theirs, our proof also works for their weaker notion of legality. We have not formally checked that our semantics validates all JMM test cases by Pugh et. al. [23]. Torlak et. al. [29] have shown that the original JMM does not validate test cases 19 and 20, but the fix by Aspinall and Ševčík [4] does. Since none of the test cases uses dynamic allocation, spawning nor interruption of threads, nor `wait` and `notify`, our formalisation should perform equivalent to the original JMM. With the fix by Aspinall and Ševčík, our formalisation also validates test cases 19 and 20.

4 Conclusion and Future Work

Our machine-checked model of multithreaded Java spans from a realistic subset of Java source and bytecode via statement and instruction-level operational semantics to the axiomatic JMM. We have proven that our semantics provides the DRF guarantee, the most important property of the JMM for programmers.

Our DRF result is not limited to Java. The key lemma 1 plays a similar role in other DRF guarantee proofs, e.g., [2,7]. They all postulate sequentially consistent completions of prefixes, which we have constructed formally for a realistic language. For Java, this surprisingly requires a full subject reduction theorem, but this need not be a restriction for other languages. C and C++, e.g., assign such type-unsafe programs undefined semantics and exclude them from the guarantee.

For this work, it was essential to separate the MM from the operational semantics. This way, we were able to define the JMM and prove the DRF guarantee on the abstract level of complete interleavings in about 2.5kLoc of definitions and proof scripts. Similarly, this clear interface allows to reuse the same JMM formalisation for both source code and bytecode. Still, connecting the operational semantics to the JMM and discharging the DRF assumptions was tedious (7.2kLoc), since every lemma must be lifted over the whole stack of semantics. In particular, the complete interleavings from §2.2 turned out very unwieldy as they connect operational semantics with inductive and coinductive definition and proof principles to the world of abstract orders. Consequently, we achieved only little proof automation there; it was much better for the interleaving semantics and the abstract JMM specification.

Initialisations and the special way the JMM handles them caused most complications in our proofs. In this work, we willingly stayed as close to the JMM as possible, but we will investigate simpler ways of initialising locations. Moreover, we have shown type safety only for SC executions, i.e., correctly synchronised programs. Since the JinjaThreads compiler correctness proof relies on type safety, we hope to show that every legal execution is type safe. Type safety of the MM, when no explicit constraints trivially enforce it, is a necessary condition for the absence of out-of-thin-air values. This will hopefully provide a better

understanding of this notion, which has so far only been illustrated by examples. Ultimately, it will be interesting to explore the tension between the safety guarantees that a MM provides and the compiler transformations it allows.

Acknowledgements. We thank M. Hecker, D. Lohner, and G. Snelting and the anonymous referees for valuable comments on earlier drafts.

References

1. Adve, S., Gharachaorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29(12), 66–76 (1996)
2. Adve, S., Hill, M.D.: Weak ordering - a new definition. In: ISCA 1990, pp. 2–14. ACM (1990)
3. Adve, S.V., Boehm, H.J.: Memory models: A case for rethinking parallel languages and hardware. Commun. ACM 53, 90–101 (2010)
4. Aspinall, D., Ševčík, J.: Formalising Java's Data Race Free Guarantee. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 22–37. Springer, Heidelberg (2007)
5. Batty, M., Memarian, K., Owens, S., Sarkar, S., Sewell, P.: Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In: POPL 2012, pp. 509–520. ACM (2012)
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: POPL 2011, pp. 55–66. ACM (2011)
7. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI 2008, pp. 68–78. ACM (2008)
8. Boyland, J.: An operational semantics including "volatile" for safe concurrency. Journal of Object Technology 8(4), 33–53 (2009); FTfJP 2008
9. Cenciarelli, P., Knapp, A., Sibilio, E.: The Java Memory Model: Operationally, Denotationally, Axiomatically. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 331–346. Springer, Heidelberg (2007)
10. Gosling, J., Joy, B., Stelle, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley (2005)
11. Huisman, M., Petri, G.: The Java Memory Model: a formal explanation. In: VAMP 2007, pp. 81–96, Tech. Rep. ICIS-R07021, University of Nijmegen (2007)
12. International standard ISO/IEC 14882:2011. programming languages – C++. International Organization for Standardization (2011)
13. Jagadeesan, R., Pitcher, C., Riely, J.: Generative Operational Semantics for Relaxed Memory Models. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 307–326. Springer, Heidelberg (2010)
14. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM TOPLAS 28(4), 619–695 (2006)
15. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28, 690–691 (1979)
16. Lochbihler, A.: Type safe nondeterminism - a formal semantics of Java threads. In: Foundations of Object-Oriented Languages, FOOL 2008 (2008)
17. Lochbihler, A.: Verifying a Compiler for Java Threads. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 427–447. Springer, Heidelberg (2010)
18. Lochbihler, A.: Jinja with threads. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (2011), `http://afp.sourceforge.net/entries/JinjaThreads.shtml`, formal proof development

19. Lochbihler, A.: A unified, machine-checked formalisation of Java and the Java Memory Model. Tech. Rep. 2011-34, Karlsruhe Reports in Informatics (2011)
20. Lochbihler, A., Bulwahn, L.: Animating the Formalised Semantics of a Java-Like Language. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 216–232. Springer, Heidelberg (2011)
21. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: POPL 2005, pp. 378–391. ACM (2005)
22. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
23. Causality test cases for the Java memory model, `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`
24. Pugh, W.: The Java memory model is fatally flawed. Concurrency: Practice and Experience 12, 445–455 (2000)
25. Quis custodiet, `http://pp.info.uni-karlsruhe.de/project.php?id=31`
26. Ševčík, J., Aspinall, D.: On Validity of Program Transformations in the Java Memory Model. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 27–51. Springer, Heidelberg (2008)
27. Ševčík, J., Vafeiadis, V., Nardelli, F., Jagannathan, S., Sewell, P.: Relaxed-memory concurrency and verified compilation. In: POPL 2011. pp. 43–54. ACM (2011)
28. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53, 89–97 (2010)
29. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: PLDI 2010. pp. 341–350. ACM (2010)