The Java Memory Model: Operationally, Denotationally, Axiomatically

Pietro Cenciarelli[1], Alexander Knapp[2], and Eleonora Sibilio[1]

[1] Dipartimento di Informatica, Università di Roma “La Sapienza”
{cenciarelli,sibilio}@di.uniroma1.it
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
knapp@pst.ifi.lmu.de

Abstract. A semantics to a small fragment of Java capturing the new memory model (JMM) described in the Language Specification is given by combining operational, denotational and axiomatic techniques in a novel semantic framework. The operational steps (specified in the form of SOS) construct denotational models (configuration structures) and are constrained by the axioms of a configuration theory. The semantics is proven correct with respect to the Language Specification and shown to capture many common examples in the JMM literature.

1 Introduction

Two processes P and Q operating in parallel compete for a lock on shared data. The structure $\mathcal{A}$ shown in Fig. 1 models the parallel composition $P \mid Q$, where P executes *lock*; . . . *unlock*; and the same does Q. The identifiers *lock* and *lock*$'$ represent *events* occurring in computation, namely the execution of a “lock” action respectively by P and Q. Similarly for *unlock* and *unlock*$'$.

Sets of events, called *configurations* and depicted here as rounded squares surrounding their elements, represent consistent states of computation. The $\{$*unlock*, *lock*$\}$ configuration, for example, represents the state reached by the system after having performed a lock action *first* and then an unlock (while Q remains dormant). We know the lock came first because we see a $\{$*lock*$\}$ subconfiguration but not an $\{$*unlock*$\}$. Note that there is no configuration $\{$*lock*, *lock*$'\}$ and this represents the *mutual exclusion* of the two processes from the shared resource.

Structures as those depicted in Fig. 1 are called *configuration structures* [1], a denotational model introduced by Winskel as an alternative presentation of (prime) *event structures* [2]. Several closure conditions have been proposed over the years to make

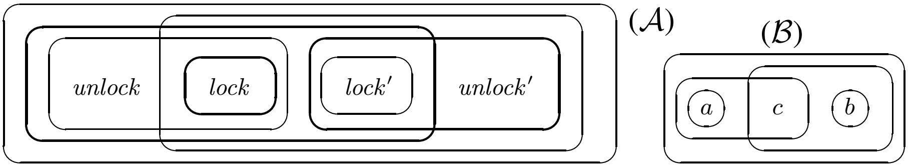

Fig. 1. Configuration structures

R. De Nicola (Ed.): ESOP 2007, LNCS 4421, pp. 331–346, 2007.

configuration structures mathematically tractable. In [3] van Glabbeek and Goltz characterise the class of configuration structures where the *causal dependency* between events can be faithfully represented by means of partial orders. Such *stable* structures are required to be closed under bounded unions and bounded intersections. Stable structures possess useful semantic properties; e.g., when a state C is part of the "history" of a state D, then D is reachable from C by a sequence of atomic steps of computation.

Unfortunately, many structures naturally arising in the semantics of concurrent systems are not stable; $\mathcal{A}$, for instance, is not. More general structures than the stable have been studied in the literature [4,5,6,7]. The *monotone* configuration structures of [6], e.g., (of which $\mathcal{A}$ is one) are those where causal dependency is preserved by inclusion of configurations, indeed a minimal requirement for monotonic reasoning about states of computation. However, consider an easy program where two threads both assign the value 42 to x (call a and b these events) while a third thread reads this value from x (event c). The corresponding structure, $\mathcal{B}$ in Fig. 1, is *not* monotone. So, a (provocative) question arises: *what are algebraically neat event-based models good for?*

The present paper advocates the usefulness of event based models by proposing a new semantic framework which combines denotational, operational and axiomatic techniques to challenge the *Java memory model*. The current definition of the Java memory model (JMM) [8] is still much driven by informal examples and, while the key ideas are understood within the community, there is a lack of rigour for mechanised reasoning. In our opinion, the reason of this is that, while the Java memory model and its run time semantics are largely independent, no formal account has been given as yet of their interplay. The notion of *execution*, introduced in the language specification as formal basis to the former, is not clearly related with the latter, in that executions may specify values being read or written which no single run of the program may be able to produce collectively. Hence, executions must be *validated* by a complicated procedure involving *tentative* executions, each validating the commitment of certain actions, but each relying on different assumptions as to the values being read or written by uncommitted actions. The connection with run time semantics is informally given by the statement that "executions should obey intra-thread consistency" [9, 4.4, clause 5].

In this paper we change perspective with respect to the language specification and propose an axiomatisation of the JMM based on the notion of *causality*, deriving from denotational semantics, rather than on the *happens-before* relation, upon which the abstract executions of [8] rely. We propose a formal framework where *structural operational semantics*, describing program evaluation, interacts with a *configuration theory*, describing the causal interplay of memory and threads.

Configuration theories were proposed in [6] as an axiomatic approach to the semantics of concurrent systems and are further developed here to capture mutual exclusion.

$$\mathit{lock}\ \ \mathit{lock}' \vdash \begin{array}{c} \mathit{lock}' \\ | \\ \mathit{unlock} \\ | \\ \mathit{lock} \end{array} \ , \ \begin{array}{c} \mathit{lock} \\ | \\ \mathit{unlock}' \\ | \\ \mathit{lock}' \end{array}$$

Fig. 2. Poset sequent for mutual exclusion

A configuration theory is a set of *poset sequents* which is closed under deduction. A poset sequent is made of partially ordered sets (posets) of events, where the order is interpreted as causal dependency. The sequent depicted in Fig. 2 (where order is represented by the vertical bars, with time pointing upward) spells roughly: "whenever two *lock* actions occur in a computation, they must occur sequentially, and moreover there must be an *unlock* action in between." As one would expect, this sequent is satisfied by structure $\mathcal{A}$, but not by the structure obtained by adding the configuration $\{lock, lock'\}$ to it, which violates mutual exclusion (see discussion in Sect. 3).

After developing the mathematics of configuration theories (Sect. 2 and 3), we present six poset sequents like the above axiomatising the JMM from the point of view of causal dependency (Sect. 4). The resulting configuration theory constrains the rules of a structural operational semantics for the minimal fragment of Java which is relevant for understanding the memory model (Sect. 5). Our semantics is then proven correct with respect to the Java language specification of [8, §17] (Sect. 6).

2 Stable Structures as Traces

A *set system* consists of a set E and a collection $\mathcal{A}$ of subsets of E [5]. If $A \in \mathcal{A}$ we write $sub(A)$ the set $\{B \in \mathcal{A} \mid B \subseteq A\}$. If $A, B \in sub(C)$ for some $C \in \mathcal{A}$ we say that A and B are *bound* in $\mathcal{A}$. The sets in a system $\mathcal{A}$ are called *configurations* when used for modelling a concurrent system, while the elements of the set $|\mathcal{A}| = \bigcup \mathcal{A}$ are called *events*. If $B \in \mathcal{A}$ and $A \in sub(B)$, then A is called a *subconfiguration* of B. A *labelled configuration structure* [5] is a structure $\mathcal{C}$ endowed by a labelling function $\lambda : |\mathcal{C}| \to Act$, where Act is a fixed set of labels called *actions*.

In [4] several closure conditions on the set of configurations of a structure $\mathcal{A}$ are given in order to get a precise match with *general event structures* (generalising those of [2]). They are: *finiteness* (if an event belongs to a configuration A, then it also belongs to a finite subconfiguration of A), *coincidence-freeness* (if two distinct events belong to a configuration A, then there exists a subconfiguration of A containing exactly one of them), closure under *bounded unions* and *non-emptiness* of $\mathcal{A}$. We call *configuration structures* (or just *structures*), and write them $\mathcal{C}, \mathcal{D}, \ldots$, the set systems satisfying all of the above requirements, *except* closure under bounded unions (this is not standard in literature). If $\mathcal{C} \subseteq \mathcal{D}$, we call $\mathcal{C}$ a *sub-structure* of $\mathcal{D}$, and $\mathcal{D}$ an *extension* of $\mathcal{C}$.

Coincidence-freeness endows each configuration C with a *canonical* partial order: $a \leq_C b$ iff, for all $D \in sub(C)$, $b \in D$ implies $a \in D$. This relation is called *causal dependency*. Two events $a, b \in C$ are said to be *concurrent* in C, written $a \diamond_C b$, when neither $a \leq_C b$ nor $b \leq_C a$ hold.

A structure $\mathcal{C}$ is called *connected* if, for all configurations $C \neq \emptyset$, there exists $a \in C$ such that $C \setminus \{a\} \in \mathcal{C}$. Clearly connectedness implies coincidence freeness and moreover, having assumed $\mathcal{C}$ nonempty and finitary, it also implies that $\emptyset \in \mathcal{C}$ (*rootedness*). Following [3] we call *stable* a configuration structure which is connected, closed under nonempty bounded unions and nonempty bounded intersections. Stability was introduced for *event structures* in [4]. Stable structures are precisely those where the order on a configuration determines its subconfigurations (see [3, Prop. 5.4 and Thm 5.2]). Below we establish a precise correspondence between certain stable configuration structures and *Mazurkiewicz traces*. The result motivates the use of stability as means for abstracting computations over concurrent actions.

Given a string s over a set S, we write $|s|$ the subset of elements of S occurring in s. A *path* over a set S is a string s of elements of S, none of which is repeated. If C is a configuration of a structure $\mathcal{C}$, we call *admissible* a path s over C such that $|u| \in \mathcal{C}$ for all prefixes u of s. We write $\simeq_C$ the smallest equivalence relation on the paths of C such that $uabv \simeq_C ubav$ if $a \diamond_C b$. A *trace* in C is an equivalence class of $\simeq_C$ in which all paths are admissible. The set of all traces $[s]_{\simeq_C}$ such that $|s| = C$ is denoted by $Tr(C)$. Note that the traces of all configurations in an *event structure* form a *Mazurkiewicz trace language* (see [10] for detail), and the construction can be shown to be the object map of an embedding (a *co-reflection*) of the category of event structures into that of trace languages [10, Cor. 39].

Theorem 1. *Let C be a configuration in a structure $\mathcal{C}$. There exists a one-to-one correspondence between the traces in $Tr(C)$ and the stable substructures $\mathcal{D}$ of $\mathcal{C}$ such that $C \in \mathcal{D} \subseteq sub(C)$, and moreover no other such substructure of $\mathcal{C}$ extends $\mathcal{D}$ properly.*

Proof. Let $[s]_{\simeq}$ be a trace in $Tr(C)$. We show that the set $\mathcal{D}$ of configurations of the form $|r|$, where r is a prefix of some path in $[s]_{\simeq}$, is stable. $\mathcal{D}$ is clearly rooted and connected. It is also closed under bounded unions. In fact, let $|u|$ and $|v|$ be configurations in $\mathcal{D}$, and let r_1 and r_2 be paths in $[s]_{\simeq}$, with u a prefix of r_1 and v of r_2. If v is empty the result holds trivially. Otherwise, let $v = av'$. Writing r_1 as waw', a must be independent of each event in w. Hence, $r_1 \simeq aww'$, and moreover the latter has a prefix u_1 such that $|u_1| = |u| \cup \{a\}$. By iterating the argument, all events in v can be pushed towards the front of r_1 to obtain a path in $[s]_{\simeq}$ with a prefix u_n such that $|u_n| = |u| \cup |v|$. Hence, $\mathcal{D}$ is stable, the argument for bounded intersections being similar to the above. Conversely, let $\mathcal{D}$ satisfy the stated conditions. It is easy to show that the set of paths r in C such that $|r| = C$ and $|u| \in \mathcal{D}$, for all prefixes u of r, is a trace in $Tr(C)$. This construction is inverse to the above. □

In view of the above result, we shall call *traces* of a configuration C in a structure $\mathcal{C}$ all the stable substructures of $\mathcal{C}$ satisfying the conditions of Thm. 1. The following result is used in Def. 2.

Proposition 1. *Let $\mathcal{D}$ and $\mathcal{E}$ be traces, respectively of D and E, in a structure, and let $\mathcal{D} \subseteq \mathcal{E}$. The inclusion map of D in E, written $D \hookrightarrow E$, is monotone with respect to the order induced by $\mathcal{D}$ and $\mathcal{E}$.*

Proof. Let $a \leq_D b$ and suppose $a \not\leq_E b$. There exists $A \in \mathcal{E}$ such that $b \in A \not\ni a$. Then $\mathcal{D} \not\ni D \cap A \in \mathcal{E}$. Clearly, $\{C \in \mathcal{E} \mid C \subseteq D\} \subseteq sub(D)$ is a stable substructure of $\mathcal{C}$ which includes $\mathcal{D}$ *properly* (as it contains $D \cap A$), and hence $\mathcal{D}$ is not maximal, against the assumptions. □

3 Sequents of Partial Maps

Notation. We write $f : A \rightharpoonup B$ to denote a *partial* function from A to B, and say that the expression $f(a)$ *denotes* (an element of B) when f is defined on $a \in A$. If e_1 and

e_2 are expressions as above involving partial functions, we write $e_1 = e_2$ when e_1 and e_2 denote the same element. When A and B are posets, we call $f : A \rightharpoonup B$ *monotone* if, when $f(a)$ and $f(b)$ both denote, $a \leq b$ implies $f(a) \leq f(b)$. (A different notion is usually adopted in domain theory, where the order represents approximation rather than causal dependency.) For partial maps f and g we write $f \sqsubseteq g$, if $f(x) = g(x)$ whenever $f(x)$ is defined. We use $\Gamma, \Delta, \ldots$ to denote sequences of posets, and write Γ_i the i-th component of Γ. The concatenation of two sequences Γ and Δ is written Γ, Δ. If $\Gamma = A_1, \ldots, A_m$ and $\Delta = B_1, \ldots, B_n$ are finite sequences of posets, we write $\rho : \Gamma \rightharpoonup \Delta$ to mean that ρ is an $m \times n$-matrix of monotone *injective* partial functions $\rho_{ij} : A_i \rightharpoonup B_j$. Given two matrices α and β of the form $\Gamma \rightharpoonup \Delta$, we write $\alpha \sqsubseteq \beta$ when $\alpha_{ij} \sqsubseteq \beta_{ij}$, for all i and j. Function composition is written in diagrammatical order.

Definition 1. *A* poset sequent *$\Gamma \vdash_\rho \Delta$ (or just* sequent*) consists of two finite sequences Γ and Δ of posets and a matrix $\rho : \Gamma \rightharpoonup \Delta$ of monotone injective partial functions.*

The posets in a sequent are meant to represent fragments of a configuration. The intuitive meaning of a sequent $\Gamma \vdash_\rho \Delta$ is that whenever a trace interprets *all* components of Γ, the interpretation extends along ρ to *at least one* component of Δ. Of course the Δ_i may include events that are not mentioned in Γ, thus specifying what is required to happen after, or must have happened before, a certain combination (Γ) of events. We write just ρ for a sequent $\Gamma \vdash_\rho \Delta$ when Γ and Δ are understood or not relevant. On the other hand, we may omit ρ when obvious from the labelling conventions.

Sequents predicate over traces. Let C be a configuration of a structure $\mathcal{C}$; by a slight abuse, we speak of a *trace* C to mean a trace $\mathcal{D}$ of C in $\mathcal{C}$. In such a case we intend C as endowed with the partial order induced by the configurations in $\mathcal{D}$. We call *interpretation* of a sequence Γ of m posets in a trace C an $m \times 1$-matrix $\Gamma \rightharpoonup C$ whose components are *total*.

Definition 2. *A structure $\mathcal{C}$ is said to* satisfy *a sequent $\Gamma \vdash_\rho \Delta$ when, for any trace C in $\mathcal{C}$ and interpretation $\pi : \Gamma \rightarrow C$, there exist a trace D extending C, a component $\Delta_k \in \Delta$ and a monotone injective total function $q : \Delta_k \rightarrow D$ such that $\rho_{ik} q \sqsubseteq \pi_i u$ for all i, where $u : C \hookrightarrow D$ is the inclusion.*

A *labelled sequent* ρ is one in which the elements of posets are assigned labels from Act and the maps in ρ preserve them. Definition 2 extends to labelled sequents and structures by requiring that interpretation maps preserve labels.

A pathological kind of sequent is $\vdash$, which features empty sequences as antecedent and succedent, and is decorated by the empty matrix. Under the assumption that structures are not empty, this sequents denotes the *absurd*. A sequent of the form $\vdash A$ is satisfied by structures in which every trace is bound to produce a configuration matching A. Similarly the sequent $A \vdash$ is satisfied by structures in which no configuration ever matches A.

The formal system of poset sequents introduced in [6] featured inference rules mimicking the structural rules of Gentzen's sequent calculus. The differences with the present work are in the kind of maps decorating the sequents (total in [6], partial here) and in the notion of interpretation (quantifying over configurations vs. traces). Partial maps yield a stronger system, in which the old rules are derivable. The sequent $a \;\vdash\; a\, b$, for example, is now derivable from $a \vdash \begin{smallmatrix} b \\ | \\ a \end{smallmatrix}$, while it was previously not, although the former holds in any structure satisfying the latter. The metatheory is also more compact, featuring four rules against ten, and a general *cut* rule, which was previously split into left and right rules. On the other hand, interpreting over traces allows us to axiomatise *mutual exclusion*, as with the lock/unlock example, which could not be captured in the old system. In fact, consider the labelled structure $\mathcal{A}$ in Fig. 1, where we assume $\lambda(\mathit{lock}) = \lambda(\mathit{lock}')$ and $\lambda(\mathit{unlock}) = \lambda(\mathit{unlock}')$, and let $\mathcal{A}'$ be the structure obtained from $\mathcal{A}$ by adding the configuration $\{\mathit{lock}, \mathit{lock}'\}$ (no mutual exclusion!). In both structures the configuration $C = \{\mathit{lock}, \mathit{unlock}, \mathit{lock}', \mathit{unlock}'\}$ is endowed with the ordering $\mathit{lock} \leq \mathit{unlock}$, $\mathit{lock}' \leq \mathit{unlock}'$. Hence, had we defined satisfaction by quantifying over configurations rather than on traces, the axiom in Fig. 2 would be satisfied by neither structures. However, while $\mathcal{A}'$ only has one trace on C (viz. $\mathcal{A}'$ itself), featuring the same order as above, $\mathcal{A}$ has two: $\{\mathit{lock} \leq \mathit{unlock} \leq \mathit{lock}' \leq \mathit{unlock}'\}$ and $\{\mathit{lock}' \leq \mathit{unlock}' \leq \mathit{lock} \leq \mathit{unlock}\}$. Hence, in the current development, $\mathcal{A}$ satisfies the axiom while $\mathcal{A}'$ does not, as expected.

The following lemmas are used to prove the soundness of our inference system of poset sequents (Fig. 3).

Let $\Gamma = \Gamma_1, \ldots, \Gamma_n$ and $\Delta = \Delta_1, \ldots, \Delta_m$ be vectors of posets; a *covariant map* from Γ to Δ consists of a function $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$ on indices, and a family of (total) monos $\psi_i : \Gamma_i \rightarrowtail \Delta_{f(i)}$. We write $(f, \psi) : \Gamma \overset{>}{\longmapsto} \Delta$ such a map, shortening (f, ψ) as f when no confusion arises. A *contravariant map* $(f, \psi) : \Gamma \overset{<}{\longmapsto} \Delta$ is defined just as above, except for $f : \{1, \ldots, m\} \to \{1, \ldots, n\}$ mapping the indices of Δ to those of Γ, and the ψ_i being of the form $\Gamma_{f(i)} \rightarrowtail \Delta_i$. A matrix $\sigma : \Gamma \rightharpoonup \Sigma$ is called *right extension* of a matrix $\rho : \Gamma \rightharpoonup \Delta$ when there exists a contravariant map $(f, \psi) : \Sigma \overset{<}{\longmapsto} \Delta$ such that $\sigma_{jf(i)}\psi_i \sqsubseteq \rho_{ji}$, for all i, j; in such a case we write $\sigma \in \mathit{rex}(\rho)$.

Lemma 1. *Let $\sigma \in \mathit{rex}(\rho)$; if a structure satisfies ρ, then it satisfies σ.*

Proof. Let a structure $\mathcal{C}$ satisfy $\rho : \Gamma \rightharpoonup \Delta$, let $\sigma : \Gamma \rightharpoonup \Sigma$ be in $\mathit{rex}(\rho)$ by $(f, \psi) : \Sigma \overset{<}{\longmapsto} \Delta$, and let $\pi : \Gamma \to C \in \mathcal{C}$ be an interpretation of Γ in $\mathcal{C}$. Since $\mathcal{C}$ satisfies ρ there exists an inclusion $u : C \hookrightarrow D$ of C in a configuration D and, for some k, a map $q : \Delta_k \to D$ such that $\rho_{ik} q \sqsubseteq \pi_i u$, for all i. Then, $\sigma_{if(k)}\psi_k q \sqsubseteq \rho_{ik} q \sqsubseteq \pi_i u$. □

The left composition of a matrix $\sigma : \Sigma \rightharpoonup \Delta$ with a covariant map $(f, \psi) : \Gamma \overset{>}{\longmapsto} \Sigma$ is the matrix $f\sigma : \Gamma \rightharpoonup \Delta$ where $(f\sigma)_{ij}(a) = \sigma_{f(i)j}(\psi_i(a))$. A *left Kan extension* of a matrix $\rho : \Gamma \rightharpoonup \Delta$ *along a covariant map* $(f, \psi) : \Gamma \overset{>}{\longmapsto} \Sigma$ is a matrix $\hat{\rho} : \Sigma \rightharpoonup \Delta$ such that $\rho \sqsubseteq f\hat{\rho}$, and moreover $\hat{\rho} \sqsubseteq \sigma$ holds for all $\sigma : \Sigma \rightharpoonup \Delta$ such that $\rho \sqsubseteq f\sigma$. It is easy to check that, when the ψ_i are *strong*, such a $\hat{\rho}$ exists iff, whenever $f(i) = f(j)$, $\psi_i(a') = \psi_j(a'')$ iff $\rho_{ik}(a') = \rho_{jk}(a'')$. In such a case $\hat{\rho}_{hk}(a)$ is $\rho_{jk}(a')$ when j and a' exist such that $h = f(j)$ and $a = \psi_j(a')$; otherwise $\hat{\rho}_{hk}(a)$ is undefined. Note that the above definition of $\hat{\rho}$ does correspond to the categorical notion of left Kan extension [11, 10.3] in a precise sense. A matrix $\sigma : \Sigma \rightharpoonup \Delta$ is called *left extension* of a matrix $\rho : \Gamma \rightharpoonup \Delta$ when ρ has a left Kan extension $\hat{\rho}$ along some map $\Gamma \overset{>}{\longmapsto} \Sigma$ and $\sigma \sqsubseteq \hat{\rho}$; in such a case we write $\sigma \in \mathit{lex}(\rho)$.

$$[\text{true}]\ \frac{}{\vdash \emptyset} \qquad\qquad [\text{incl}]\ \frac{}{A \vdash_{\phi^{-1}} B}\ \ (\phi : B \rightarrowtail A \text{ is strong})$$

$$[\text{sub}]\ \frac{\mathit{\Gamma} \vdash_{\rho} \mathit{\Delta}}{\mathit{\Sigma} \vdash_{\sigma} \mathit{\Pi}}\ \ \sigma \leq \rho \qquad\qquad [\text{cut}]\ \frac{\mathit{\Gamma} \vdash_{\tau,\rho} A, \mathit{\Delta} \qquad \mathit{\Sigma}, A \vdash_{\sigma;\pi} \mathit{\Pi}}{\mathit{\Gamma}, \mathit{\Sigma} \vdash_{(\rho;\emptyset),(\tau\pi;\sigma)} \mathit{\Delta}, \mathit{\Pi}}$$

Fig. 3. Inference rules

Lemma 2. *Let $\sigma \in \mathit{lex}(\rho)$; if a structure satisfies ρ, then it satisfies σ.*

Proof. Let structure $\mathcal{C}$ satisfy $\rho : \mathit{\Gamma} \rightharpoonup \mathit{\Delta}$, let $\hat{\rho}$ be a Kan extension of ρ along $(f, \psi) : \mathit{\Gamma} \overset{>}{\longmapsto} \mathit{\Sigma}$, let $\sigma \sqsubseteq \hat{\rho}$ and let $\pi : \mathit{\Sigma} \to C \in \mathcal{C}$ be an interpretation of $\mathit{\Sigma}$ in $\mathcal{C}$. The interpretation $f\pi$ yields a configuration $C \subseteq D \in \mathcal{C}$ and a map $q : \mathit{\Delta}_k \to D$ such that $\rho_{ik} q \sqsubseteq \psi_i \pi_{f(i)k} u$, where $u : C \to D$ is the inclusion. Then, $\sigma \sqsubseteq \hat{\rho}$ yields $\sigma q \sqsubseteq \pi u$. □

Figure 3 shows rule schemes for deriving poset sequents. Rule [sub] makes use of a preorder $\leq$ over sequents defined to be the smallest transitive relation where $\sigma \leq \rho$ when σ is either in $\mathit{lex}(\rho)$ or in $\mathit{rex}(\rho)$. In the [cut] rule two operations (comma and semi-colon) are used to compose matrices. If ρ and σ are matrices of size $m \times n$ and $r \times n$ respectively, we write $(\rho; \sigma)$ for the $(m + r) \times n$ matrix obtained by "placing ρ above σ": the ij-component of $(\rho; \sigma)$ is ρ_{ij} for $i \leq m$, while it is $\sigma_{(i-m)j}$ when $i > m$. Similarly, if ρ and σ are of size $m \times n$ and $m \times r$, we write (ρ, σ) for the $m \times (n + r)$ matrix obtained by "placing ρ to the left of σ": the ij-component of (ρ, σ) is ρ_{ij} for $j \leq n$, while it is $\sigma_{i(j-n)}$ when $j > n$. Finally, let τ and π be respectively a $n \times 1$ column vector and a $1 \times m$ row vector. Then, $\tau\pi$ stands for the $n \times m$ matricial *product* of the two, where $(\tau\pi)_{ij}$ is the composite map $\mathit{\Gamma}_i \xrightarrow{\tau_i} A \xrightarrow{\pi_j} \mathit{\Pi}_j$. By $\emptyset$ we mean a matrix (of suitable size) whose components are the always undefined partial functions.

Definition 3. *A* configuration theory *is a set of sequents which is closed under the rule schemes of Fig. 3.*

Theorem 2. *The rules of Fig. 3 are sound.*

The proof is almost immediate for all the rules except for [sub], where it follows from Lemmas 1 and 2. Completeness can also be obtained by adjoining to the rules of Fig. 3 the [extend] rule of [6, 5]. This is however out of the scope of the present paper.

4 A Configuration Theory of Java

We present a configuration theory specifying the rules by which events of a Java computation may depend on each other.

Let *Var*, *Mon* and *Tid* denote disjoint countable sets, respectively of program variables (ranged over by $x, y, \ldots$), monitors ($m, \ldots$) and thread identifiers ($\theta, \zeta, \xi, \ldots$). The *actions* of the theory of Java are either of the form (H, θ, x, v), where $H \in \{R, W\}$ and v is a value, or of the form (K, θ, m), with $K \in \{L, U\}$. Actions (H, θ, x, v), called *memory actions*, represent the *reading* (R) of a value v from the variable x by a thread θ, or the assignment (W for *writing*) of v to x by θ, while actions of the form (K, θ, m),

$$1)\quad a\ b \vdash \begin{array}{c} a \\ \mid \\ b \end{array},\ \begin{array}{c} b \\ \mid \\ a \end{array} \qquad \begin{array}{ll} \text{1a)} & a = (\theta, x, v),\ b = (\theta, x, w) \\ \text{1b)} & a = (\theta, x, v),\ b = (\theta, m) \\ \text{1c)} & a = (\zeta, m),\ b = (\theta, m) \end{array}$$

$$2)\quad (R,\theta,x,v) \vdash \begin{array}{c} (R,\theta,x,v) \\ \mid \\ (W,\zeta,x,v) \end{array} \qquad 3)^{\star *}\ \begin{array}{c} (R,\theta,x,v) \\ \mid \\ (W,\theta,x,w) \end{array} \vdash \begin{array}{c} (R,\theta,x,v) \\ \mid \\ (W,\theta,x,v) \\ \mid \\ (W,\theta,x,w) \end{array},\ \begin{array}{c} (R,\theta,x,v) \\ \mid \\ (W,\zeta,x,v) \end{array}$$

$$4)^{\star *}\ \begin{array}{c} (R,\theta,x,v) \\ \mid\ \dots\ \mid \\ A_1 \qquad A_n \end{array} \vdash B_1, \dots, B_n,\ \begin{array}{c} (R,\theta,x,v) \\ \mid \\ (W,\xi,x,v) \end{array} \ \text{where}\ A_i = \begin{array}{c} (L,\theta,m_i) \\ \mid \\ (U,\zeta_i,m_i) \\ \mid \\ (W,\zeta_i,x,w_i) \end{array} \ \text{and}\ B_i = \begin{array}{c} (R,\theta,x,v) \\ \mid \\ (W,\zeta_i,x,v) \\ \mid \\ (W,\zeta_i,x,w_i) \end{array}$$

$$5)\quad (U,\theta,m)^n \vdash \begin{array}{c} (U,\theta,m)^n \\ \mid \\ (L,\theta,m)^n \end{array} \qquad 6)^{*}\ \begin{array}{c} (L,\theta,m) \\ \mid \\ (L,\zeta,m)^n \end{array} \vdash \begin{array}{c} (L,\theta,m) \\ \mid \\ (U,\zeta,m)^n \end{array}$$

$(\star)$ $v \neq w, w_i$ for all i
$(*)$ $\theta \neq \zeta, \zeta_i$ for all i

Fig. 4. The configuration theory of Java

called *synchronisations*, represent the *locking* (L) or the *unlocking* (U) of a monitor m by θ. When H and K are irrelevant, (H, θ, x, v) and (K, θ, m) are shortened respectively as (θ, x, v) and (θ, m). Other action component may be similarly omitted when not relevant. Events are labelled by actions. We write $e : l$ to mean that event e has label l. When no confusion arises, we use actions to denote the events of which they are labels. We do so in Fig. 4.

Figure 4 shows the axiom schemes of our configuration theory of Java. The ρ in a sequent $\Gamma \vdash_\rho \Delta$ is left implicit by convening that an event $e : A$ in Γ_i is mapped by ρ_{ij} to one with the same label A in Δ_j, in lack of which $\rho_{ij}(e)$ is undefined.

Scheme 1 describes how the different kinds of actions are to be ordered in legal program executions, according to the Java memory model [8, §17]. All memory actions of one thread over a same variable must be totally ordered (1a), while all synchronisations of a thread over a monitor must be ordered with the memory actions of that thread (1b) and with the synchronisations of other threads over the same monitor (1c).

Schemes 2, 3 and 4 specify how threads are allowed to read values from the shared memory. Any value being read by a thread θ from a variable x must have been previously assigned to x by a *possibly* different thread (2). If θ reads its own assignment, then it must be the most recent one (3), while, if it is a value assigned by another thread ζ, it must be the most recent only if θ and ζ synchronised over the same monitor (4).

Schemes 5 and 6 describe synchronisation. By a^n we mean a poset of n a-labelled events $a_1, \dots, a_n$, with the discrete ordering, while $\begin{array}{c} b^n \\ \mid \\ a^n \end{array}$ denotes the poset $a^n \cup b^n$ where $a_i \leq b_i$, for all i. Then, scheme 5 says that any unlock action must be paired with a preceding lock by the same thread, while 6 guarantees, in combination with 5, that locks are granted to one thread at a time.

5 An Event-Based Semantics of Java

The axioms are used to constrain the applicability of the operational rules: semantic configurations of events, labelled as in Sect. 4, are included as part of the *operational* configurations, and each time the semantics reduces a Java term an event is added to (and causal dependencies recorded in) the current semantic configuration, *provided* this complies with the specified theory. Thus, operational semantics builds a denotational model of the program (see discussion in Sect. 7). However, events may also be added to the semantic configurations *presciently* (by rule [pre] in Tab. 1), that is before the corresponding reduction is performed, and only later *fulfilled* by the execution engine. Hence, semantic configurations are also equipped with a *fulfilment predicate* $(\_)!$ on write events. Intuition is that $(W)!$ holds in η precisely when (W) has been fulfilled by program evaluation. More formally: configurations of events are called *event spaces* (and ranged over by $\eta, \zeta, \ldots$) when viewed as part of operational configurations. Mathematically an event space is just a poset equipped with a fulfilment predicate and satisfying the axioms of Fig. 4. By that we mean that it does when viewed as the (stable) structure whose configurations are its downward closed subsets.

By using prescient actions, threads may read values from the shared memory which have not yet been assigned to the corresponding variable. As predicated in the Java specification [8], this allows the language implementation to apply compiler optimisation techniques (such as swapping statements, extracting assignments from the branches of an `if` ...) without violating the legal executions of a program.

Dependencies. A *syntactic dependency set* is a set of read events. Given syntactic dependency sets δ_1 and δ_2, we write $\delta_1\delta_2$ for $\delta_1 \cup \delta_2$, while $\delta\, e$ stands for $\delta \cup \{e\}$. Syntactic dependencies are attached to statements during evaluation. Intuitively, if x is assigned the value 7 by a statement $x = y + 2$, the corresponding write action must depend on some event labelled by $(R, y, 5)$. When fulfilling the assignment, the operational semantics checks that its syntactic dependencies do correspond to causal dependencies in the current event space.

An event e is adjoined to an event space η by an operation $\oplus$. More precisely, let η and η' be event spaces; we write $\eta' \in \eta \oplus e$ when:

- $|\eta'| = |\eta| \cup \{e\}$ and the order in η' extends that of η conservatively;
- fulfilment in η' extends that of η conservatively, with e unfulfilled if $e : (W)$;
- if e is labelled by (R, θ, x), then $d!$ holds for all $d : (W, \theta, x) < e$;
- if $e : (\theta) < d : (\theta)$, then d is an unfulfilled write.

We write $\eta \oplus e$ to denote *any* $\eta' \in \eta \oplus e$. If no such η' exists, then $\eta \oplus e$ is undefined. Given an event space η, a dependency set δ and a write action (W, θ, x, v), the expression $\eta \downarrow_\delta (W, \theta, x, v)$ is defined if there exists an *unfulfilled* event $e : (W, \theta, x, v)$ in η such that $d!$ holds for all $d : (W, \theta, x) < e$, and moreover $d' < e$ in η for all $d' \in \delta$. Noting that such an e is necessarily unique, we let $\eta \downarrow_\delta (W, \theta, x, v)$, when defined, denote the event space η with the new fulfilment $e!$.

Syntax. We use the following simple fragment of Java.

$$\begin{array}{rcl@{\qquad}rcl}
\textit{D-Term} & ::= & \textit{D-Stm} \mid \textit{D-Expr} & \textit{Stm} & ::= & \texttt{;} \mid \textit{Var}\ \texttt{=}\ \textit{D-Expr}\ \texttt{;} \mid \textit{D-Stm}\ \textit{D-Stm} \\
\textit{D-Stm} & ::= & \textit{Stm Dep} & & \mid & \texttt{if (}\ \textit{D-Expr}\ \texttt{)}\ \textit{D-Stm}\ \texttt{else}\ \textit{D-Stm} \\
\textit{D-Expr} & ::= & \textit{Expr Dep} & & \mid & \texttt{synchronized (}\ \textit{Mon}\ \texttt{)}\ \textit{D-Stm} \\
 & & & & \mid & \textit{synchronized}\ \texttt{(}\ \textit{Mon}\ \texttt{)}\ \textit{D-Stm} \\
 & & & \textit{Expr} & ::= & \textit{Lit} \mid \textit{Var} \mid \textit{Expr Op Expr}
\end{array}$$

Here, *Lit* is the syntactic domain of *literals*, which we identify with the domain of values and where we assume suitable functions $op : \textit{Lit} \times \textit{Lit} \to \textit{Lit}$ corresponding to the syntactic binary operators $\texttt{op} \in \textit{Op}$. *Dep* stands for the domain of syntactic dependency sets. A "conventional" Java term like `x = 1;` is turned into a *D-Term* (*dependent* term) by filling in empty dependency sets, i.e., $(\texttt{x = }(\texttt{1})_\emptyset\ \texttt{;})_\emptyset$, and we omit empty dependency sets in our examples.

Operational configurations. An operational configuration represents the state of execution of a multi-threaded Java program; therefore, it may include several dependent terms, one for each thread of execution. We call *multiterm* a partial map from thread identifiers to dependent terms. We let the metavariable T range over multiterms: $T : \textit{Tid} \rightharpoonup \textit{D-Term}$. When we assume that θ is not in the domain of T we write $T \parallel (\theta, t)$ for the multiterm T' such that $T'(\theta) = t$ and $T'(\theta') \simeq T(\theta')$ for $\theta' \neq \theta$; where $h \simeq h'$ means that if h is defined so is h', and vice versa.

An *operational configuration* is a pair (T, η) consisting of a multiterm T and an event space η. In writing operational configurations, we generally drop the parentheses and all parts that are not immediately relevant in the context of discourse; for example, we may write just "t, η" to mean some configuration $(T \parallel (\theta, t), \eta)$. Operational configurations are ranged over by γ.

Rule conventions. In writing an axiom $\gamma_1 \to \gamma_2$ we focus only on the relevant parts of the configurations involved, and understand that whatever is omitted from γ_1 remains unchanged in γ_2. For example, we understand that the axiom $\texttt{;}\ p \to p$ stands for $T \parallel (\theta, \texttt{;}\ p), \eta \to T \parallel (\theta, p), \eta$. On the other hand, rules with a premise are read by assuming that whatever changes occur in the omitted parts of the premise also occur in the conclusion. For example, we understand that:

$$\frac{e_1 \to e_2}{e_1\ \texttt{op}\ e \to e_2\ \texttt{op}\ e} \quad \text{means} \quad \frac{T_1 \parallel (\theta, (e_1)_{\delta_1}), \eta_1 \to T_2 \parallel (\theta, (e_2)_{\delta_2}), \eta_2}{T_1 \parallel (\theta, (e_1\ \texttt{op}\ e)_{\delta_1}), \eta_1 \to T_2 \parallel (\theta, (e_2\ \texttt{op}\ e)_{\delta_2}), \eta_2}\,.$$

Operational rules. The operational rules are given in Tab. 1. The metavariables used (in variously decorated form) in the rule schemes range as follows: $u, v \in \textit{Lit}$, $x \in \textit{Var}$, $m \in \textit{Mon}$, $d, e \in \textit{Expr}$, $s \in \textit{Stm}$, $p, q \in \textit{D-Stm}$, $\delta, \epsilon \in \textit{Dep}$.

The JMM axioms (Fig. 4) constrain the operational rules. This is because the latter rely on $\oplus$ producing a legal event space. For example, an attempt by a thread θ to use [syn1] for acquiring a lock on m would fail if m is detained by a different thread in the current state η, because the expression $\eta \oplus (L, \theta, m)$ would then denote no event space satisfying the axioms for locks. Similarly, the value v read by θ in x through rule [var] is forced to comply with the model by the requirement that $\eta \oplus (R, \theta, x, v)$ be defined.

Table 1. Operational rules

$$\text{[binop1]}\quad \frac{d \to e}{d \;\texttt{op}\; e' \to e \;\texttt{op}\; e'} \qquad \text{[binop2]}\quad \frac{d \to e}{v \;\texttt{op}\; d \to v \;\texttt{op}\; e}$$

$$\text{[binop3]}\quad u \;\texttt{op}\; v \to \mathit{op}(u,v) \qquad \text{[var]}\quad \theta : x, \eta \to \theta : v_{(R,\theta,x,v)}, \eta \oplus (R,\theta,x,v)$$

$$\text{[assign1]}\quad \frac{d \to e}{x \;\texttt{=}\; d\texttt{;} \to x \;\texttt{=}\; e\texttt{;}} \qquad \text{[assign2]}\quad \theta : x \;\texttt{=}\; v_{\epsilon}\,\texttt{;}_{\delta}, \eta \to \theta : \texttt{;}_{\delta}, \eta \downarrow_{\delta\epsilon} (W,\theta,x,v)$$

$$\text{[if1]}\quad \frac{d \to e}{\texttt{if (}\, d \,\texttt{)}\; p \;\texttt{else}\; q \to \texttt{if (}\, e \,\texttt{)}\; p \;\texttt{else}\; q}$$

$$\text{[if2]}\quad (\texttt{if (}\, \mathit{true}_{\epsilon} \,\texttt{)}\; p \;\texttt{else}\; q)_{\delta} \to p_{\delta\epsilon}$$

$$\text{[if3]}\quad (\texttt{if (}\, \mathit{false}_{\epsilon} \,\texttt{)}\; p \;\texttt{else}\; q)_{\delta} \to q_{\delta\epsilon}$$

$$\text{[if4]}\quad \frac{p_{\delta}, \eta \to p'_{\delta}, \eta' \quad q_{\delta}, \eta \to q'_{\delta}, \eta'}{(\texttt{if (}\, v \,\texttt{)}\; p \;\texttt{else}\; q)_{\delta}, \eta \to (\texttt{if (}\, v \,\texttt{)}\; p' \;\texttt{else}\; q')_{\delta}, \eta'}$$

$$\text{[syn1]}\quad \theta : \texttt{synchronized (}\, m \,\texttt{)}\; p, \eta \to \theta : \mathit{synchronized}\;\texttt{(}\, m \,\texttt{)}\; p, \eta \oplus (L,\theta,m)$$

$$\text{[syn2]}\quad \frac{p_{\delta} \to q_{\delta}}{(\mathit{synchronized}\;\texttt{(}\, m \,\texttt{)}\; p)_{\delta} \to (\mathit{synchronized}\;\texttt{(}\, m \,\texttt{)}\; q)_{\delta}}$$

$$\text{[syn3]}\quad \theta : \mathit{synchronized}\;\texttt{(}\, m \,\texttt{)}\;\texttt{;}, \eta \to \theta : \texttt{;}, \eta \oplus (U,\theta,m)$$

$$\text{[skip]}\quad \texttt{;}\; p \to p \qquad \text{[seq]}\quad \frac{p_{\delta} \to p'_{\delta}}{(p\; q)_{\delta} \to (p'\; q)_{\delta}} \qquad \text{[pre]}\quad T, \eta \to T, \eta \oplus (W)$$

Examples. We show that an execution of the sample program in Fig. 5, top-left, started with all variables initialised to zero can result in `r1` and `r2` set to 1, as predicated in [9]. Using rule [pre], the operational semantics may first "guess" that `x` and `y` will eventually be set to 1 and that these settings do not causally depend on any previously read value. In fact, this will be fulfilled by execution according to the operational semantics, and thus the Java trace (writing $a \to b$ for $a \leq b$) in Fig. 5, top-right, can be produced:

$$\begin{array}{ll}
\texttt{r1=x; y=1;} \parallel \texttt{r2=y; x=1;}, \emptyset & \xrightarrow{\text{[pre]}} \\
\texttt{r1=x; y=1;} \parallel \texttt{r2=y; x=1;}, \{c'\} & \xrightarrow{\text{[assign1, var]}} \\
\texttt{r1=}\texttt{1}_{a}\texttt{; y=1;} \parallel \texttt{r2=y; x=1;}, \{c' < a\} & \xrightarrow{\text{[pre]}} \\
\texttt{r1=}\texttt{1}_{a}\texttt{; y=1;} \parallel \texttt{r2=y; x=1;}, \{c' < a < b\} & \xrightarrow{\text{[assign2]}} \\
\texttt{; y=1;} \parallel \texttt{r2=y; x=1;}, \{c' < a < b!\} & \xrightarrow{\text{[skip]}} \\
\texttt{y=1;} \parallel \texttt{r2=y; x=1;}, \{c' < a < b!\} & \xrightarrow{\text{[pre]}} \\
\texttt{y=1;} \parallel \texttt{r2=y; x=1;}, \{c' < a < b!, c\} & \xrightarrow{\text{[assign2]}} \\
\texttt{;} \parallel \texttt{r2=y; x=1;}, \{c' < a < b!, c!\} & \xrightarrow{\text{[assign1, var]}} \\
\texttt{;} \parallel \texttt{r2=}\texttt{1}_{a'}\texttt{; x=1;}, \{c' < a < b!, c! < a'\} & \xrightarrow{\text{[pre]}} \\
\texttt{;} \parallel \texttt{r2=}\texttt{1}_{a'}\texttt{; x=1;}, \{c' < a < b!, c! < a' < b'\} & \xrightarrow{\text{[assign2]}} \\
\texttt{;} \parallel \texttt{; x=1;}, \{c' < a < b!, c! < a' < b'!\} & \xrightarrow{\text{[skip]}} \\
\texttt{;} \parallel \texttt{x=1;}, \{c' < a < b!, c! < a' < b'!\} & \xrightarrow{\text{[assign2]}} \\
\texttt{;} \parallel \texttt{;}, \{c'! < a < b!, c! < a' < b'!\} &
\end{array}$$

where the terms for the threads θ_1 and θ_2 are shown left and right to $\parallel$.

Thread θ_1	Thread θ_2
`r1 = x;`	`r2 = y;`
`y = 1;`	`x = 1;`

Thread θ_1	Thread θ_2
`r1 = x;`	`r2 = y;`
`if (r1 == 1)`	`if (r2 == 1)`
`  y = 1;`	`  x = 1;`
	`else`
	`  x = 1;`

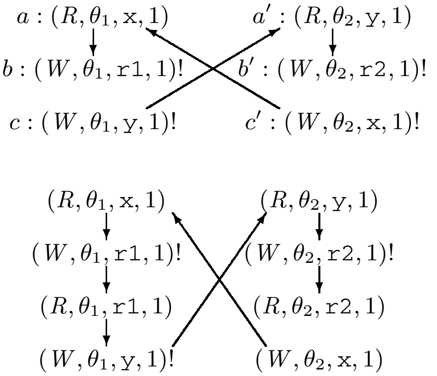
$a : (R, \theta_1, \mathtt{x}, 1)$
$a' : (R, \theta_2, \mathtt{y}, 1)$
$b : (W, \theta_1, \mathtt{r1}, 1)!$
$b' : (W, \theta_2, \mathtt{r2}, 1)!$
$c : (W, \theta_1, \mathtt{y}, 1)!$
$c' : (W, \theta_2, \mathtt{x}, 1)!$
$(R, \theta_1, \mathtt{x}, 1)$
$(R, \theta_2, \mathtt{y}, 1)$
$(W, \theta_1, \mathtt{r1}, 1)!$
$(W, \theta_2, \mathtt{r2}, 1)!$
$(R, \theta_1, \mathtt{r1}, 1)$
$(R, \theta_2, \mathtt{r2}, 1)$
$(W, \theta_1, \mathtt{y}, 1)!$
$(W, \theta_2, \mathtt{x}, 1)$

Fig. 5. Examples of Java programs and resulting Java configurations

In contrast, in the program

$$\theta_1 : \mathtt{r1 = x;\ if\ (r1 == 1)\ y = 1;} \parallel \theta_2 : \mathtt{r2 = y;\ if\ (r2 == 1)\ x = 1;}$$

the write action for `y` and `x` do depend on the values previously read from `r1` and `r2`, respectively. Consequently, a poset like the one depicted in Fig. 5, bottom-right, in which $(W, \theta_2, \mathtt{x}, 1)$ does not extend to a fulfilled execution. But, in fact, this Java configuration with this event being fulfilled is the possible outcome of the program in Fig. 5, bottom-left, where a single write to `x` not depending on `r2` suffices.

6 Correctness

The JMM [8, §17] is based on a notion of "happens-before". This notion subsumes on the one hand the *program order po*, a thread-wise total order of actions as dictated by sequentially executing each thread according to the Java language specification; on the other hand, it is based on the *synchronisation order so*, the total order of all lock and unlock actions in a program run. Then the *happens-before order hb*, which must be a partial order, is defined to include the transitive closure of *po* with the *synchronises-with order sw* which restricts *so* to lock and unlock actions on the same monitor.

The action description of the JMM differs from our notion of Java actions with respect to the values, which we included into the actions: In the JMM, two functions V and W are used where V gives for a write action the *value written* of this write and W references for a read action the *write seen* by this read. The write-seen function must be compatible with the happens-before order in the sense that no write can be seen by a read which actually happens after it, and no read can see a write that happened before it but has been overwritten in the happens-before order. Finally, the JMM requires that all variables of a program are properly initialised and that these initialisations can be seen by all threads. For this purpose it strengthens the synchronises-with order to include the initialising writes and the first action of each thread.

A (well-formed) *execution* of a program P with an action set A now, according to the JMM, is a tuple $(P, A, \mathit{po}, \mathit{so}, W, V, \mathit{sw}, \mathit{hb})$ fulfilling the description above. It has to be stressed that the JMM description [8, §17] does not define the connection between the program P and the actions A and the various orderings and functions. In fact, the actions actually executed in a program run will, in general, depend on W and V, and their precise connection would be mutually recursive.

The notion of happens-before alone does not suffice to capture causally legal executions, as it would allow "out-of-thin-air" results to be produced. Thus, the JMM predicates that an execution X has to be *validated* by a sequence of other executions $(X_i)_i$ of the same program *committing* subsequently all actions of X in an increasing sequence $(C_i)_i$. The process of commitments must be such that the happens-before orders and the value-written functions of X and X_i coincide on already committed actions in C_i; the writes-seen of X_i, however, need not coincide on C_i, but only on C_{i-1}, with the additional requirement that every new read action in X_i has to see a write that happened-before in X_i and, if it is committed in C_i, then the write-seen must be in C_{i-1}. Finally, synchronisation actions immediately following each other in X_i below a committed action in C_i must persist in the validation process.

In order to prove that our semantics is correct with respect to the JMM, we have to show that a run of the operational semantics on a multiterm T such that the final Java trace is fulfilled indeed gives rise to an execution X for T that can be validated by a sequence $(X_i, C_i)_i$ of executions and commitments. We assume in the following that the operational semantics starts with an initial Java trace η_T that show initialisations for all variables of P and that η_T will be extended during computation in such a way that all subsequent events depend on the initialisations.

Let T be a multiterm and let $\vec{\gamma}$ be a computation $\gamma_0 \to \cdots \to \gamma_n$, with $\gamma_0 = (T, \eta_T)$, $\gamma_i = (T_i, \eta_i)$, and η_n totally fulfilled. For the first task, producing an execution, we observe that the computation $\vec{\gamma}$ induces a total order on the events in η_n by assigning to each $e \in |\eta_n|$ the index of the computational step in which either it was added, if $e : (R)$, or $e : (L)$, $e : (U)$, or it was fulfilled, if $e : (W)$. We construct an execution

$$\mathit{exec}(\vec{\gamma}) = (T, |\eta_n|, \mathit{po}(\vec{\gamma}), \mathit{so}(\vec{\gamma}), W(\vec{\gamma}), V(\vec{\gamma}), \mathit{sw}(\vec{\gamma}), \mathit{hb}(\vec{\gamma}))$$

as follows: Constraining the total order of events to each thread and to all synchronisation actions, we obtain a program order $\mathit{po}(\vec{\gamma})$ and a synchronisation order $\mathit{so}(\vec{\gamma})$, respectively; this also induces a happens-before order $\mathit{hb}(\vec{\gamma})$ and a synchronises-with order $\mathit{sw}(\vec{\gamma})$. We define the value-written function $V(\vec{\gamma})$ by setting $V(\vec{\gamma})(e) = v$ if $e : (W, v) \in \eta_n$, and a write-seen function $W(\vec{\gamma})$ by setting $W(\vec{\gamma})(e)$ to that $e' \in \eta_n$ which satisfies $e' : (W, v) \leq e : (R, v)$ in η_n and has the minimum distance of indices assigned to e and e'.

Lemma 3. *$\mathit{exec}(\vec{\gamma})$ is a well-formed execution of T.*

Proof. By construction, $hb(\vec{\gamma})$ is a partial order. $W(\vec{\gamma})$ conforms to the requirements of the JMM as, although there may be several writes of the desired value for a read that can be seen by the read, there will be at least one valid for $W(\vec{\gamma})$ by axioms (2–4) on Java configurations. □

For the second task, validating an execution $exec(\vec{\gamma})$, we construct a sequence of executions and commitments $(X(\vec{\gamma})_i, C(\vec{\gamma})_i)$ inductively as follows: $X(\vec{\gamma})_0$ and $C(\vec{\gamma})_0$ are empty. Assuming $X(\vec{\gamma})_k$ and $C(\vec{\gamma})_k$ to have been defined already for a $0 < k < n$, we let e_{k+1} be a minimal element of $\eta_n \setminus C_k$. Then there is a computation $\vec{\gamma}^{(k)} = \gamma_0^{(k)} \rightarrow \cdots \rightarrow \gamma_l^{(k)}$, with $\gamma_0^{(k)} = (T, \eta_T)$, $\eta_l^{(k)}$ fulfilled, $\eta_n \restriction C(\vec{\gamma})_k = \eta_l^{(k)}$, and e_{k+1} maximal in $\eta_l^{(k)}$, which uses the [pre] rule only for events in C_k. Indeed, using $exec(\vec{\gamma})$ as the guide for executing which statement and action, no rule execution can be prohibited, but it may produce a different value for the read and write actions. In fact, having chosen e_{k+1} to be minimal in $\eta_n \setminus C(\vec{\gamma})_k$ all events in the $\eta_l^{(i)}$ only depend on actions having been committed in C_k and thus, in particular, for e_{k+1} the same value as in η will be produced. As $\vec{\gamma}^{(k)}$ is a computation, it induces an execution $X(\vec{\gamma})_{k+1} = exec(\vec{\gamma}^{(k)})$ by Lem. 3; we also set $C(\vec{\gamma})_{k+1} = C(\vec{\gamma})_k \cup \{e_{k+1}\}$.

Lemma 4. *$exec(\vec{\gamma})$ is validated by the sequence $(X(\vec{\gamma})_i, C(\vec{\gamma})_i)_i$.*

Proof. By construction, the happens-before order of $exec(\vec{\gamma})$ is preserved on each $C(\vec{\gamma})_i$ and all read actions either use a happens-before value in $X(\vec{\gamma})_i$, as the [pre] rule must not be used for uncommitted actions, or see a happens-before write. □

It is worth noting that we have resolved the dilemma of the mutually dependent definitions of program actions and the values seen and written by these actions in the JMM by restricting the use of prescient write actions in our construction of a validation sequence.

7 Conclusions and Further Research

We presented a structural operational semantics of a small fragment of Java including much of what is needed to understand the JMM. The semantics was proven correct with respect to the language specification of [8]. The specification of the memory model (Fig. 4) is separate from the run time semantics (Tab. 1) and yet connected in a single formal framework which gives unambiguous account of their interplay. We believe this has been missing in the literature as yet. Moreover, the theoretical foundations of the proposed framework, combining denotational, operational and axiomatical semantics, support formal reasoning about programs, specifically for proving correctness of optimisation techniques.

There are, e.g., obvious compiler optimisations that the current JMM does *not* support. An example is the following program where threads θ_1 and θ_2 run in parallel:

θ_1 : `r1 = x; r2 = y; if (r1 == 1 && r2 == 1) z = 1;`
θ_2 : `r3 = z; if (r3 == 1) { x = 1; y = 1; } else { y = 1; x = 1; }`

After reordering the independent statements in the `else` branch, a compiler may execute assignments `x = 1;` and `y = 1;` *early*, so that `r1`, `r2`, `r3` can all be assigned 1. However, such a behaviour is not legal according to the current JMM, as it violates the condition that the happens-before orders during validation be consistent with the final happens-before on the committed actions. In fact, the latter will have the write to `x` before the write to `y`, but during validation the write to `y` happens before the write to `x`.

This is indeed a counterexample to the claim by Manson, Pugh, and Adve [9, Thm. 1] that in the JMM all independent program statements can be reordered; it seems that the happens-before order would have to be relaxed, not requiring, e.g., the ordering of independent program actions. In our framework, such a compiler optimisation can be included by a simple editing of rule [if4]. The theory of reorderings developed by Saraswat et al. [12] takes into account also more complicated code rearrangements, but, like the JMM, is not connected to a language semantics.

On a more theoretical side, we notice that our axiomatisation of the JMM has only been used to constrain the operational rules by *local* checks on fragments of a configuration structure, the event spaces. What the *whole* structure is, which represents the full program denotationally, can also be made explicit. (The following construction extends easily to possibly infinite computations, e.g. when including `while` loops.)

Let $\eta_0, \ldots, \eta_n$ be the sequence of event spaces of a computation $\vec{\gamma}$. We write $\eta_{\vec{\gamma}}$ to denote the last event space η_n in $\vec{\gamma}$. A computation $\vec{\gamma}$ is called *accomplished* if all write actions in $\eta_{\vec{\gamma}}$ are fulfilled and moreover, if T_n is its last multiterm, then $T_n(\theta)$ is `;`, when defined, for all threads θ. We write $\underline{x}$ to denote a specific occurrence of a variable x in a program T, and similarly for monitors. Let E_T be the set whose elements are either pairs $(\underline{x}, v)$, where x is a variable and v a value, or pairs $(\underline{m}, K)$, where m is a monitor and $K \in \{L, U\}$. Viewing the elements of E_T as events, we construct a denotational model of T by assuming that operational semantics adjoins events to the current trace according to the following protocol:

- [var] adds $(\underline{x}, v) : (R, x, v)$ if v is the value read at $\underline{x}$;
- [pre] adds $(\underline{x}, v) : (W, x, v)$ if v is the value written in $\underline{x}$;
- [syn1] adds $(\underline{m}, L) : (L, m)$ when evaluating `synchronized (` $\underline{m}$ `)` p;
- [syn3] adds $(\underline{m}, U) : (U, m)$ when evaluating *synchronized* `(` $\underline{m}$ `) ;`;

Given a program T, we let $[\![T]\!]$ be the structure whose configurations are sets $C \subseteq E_T$ such that there exists an accomplished computation $\vec{\gamma}$ of T and C is a downward closed subset of $\eta_{\vec{\gamma}}$. Note that the causal dependency relation associated with such a C in $[\![T]\!]$ is included in, but may not coincide with, the partial order of $\eta_{\vec{\gamma}}$ restricted to C.

Proposition 2. *$[\![T]\!]$ satisfies the Java axioms.*

Proof. Suppose $[\![T]\!]$ does not satisfy an axiom $\Gamma \vdash_\rho \Delta$. There must exist a trace C in $[\![T]\!]$ and an interpretation $\pi : \Gamma \to C$ violating the conditions of Def. 2. By definition, $|C|$ is a downward closed subset of some $\eta_{\vec{\gamma}}$, and there exists an event space η in $\vec{\gamma}$ (hence satisfying the axioms) which contains all events in C. By an easy argument, η satisfies ρ iff so does C, against the assumptions. □

By the arguments developed in Sect. 1, we know that $[\![T]\!]$ is neither stable nor monotone. What the algebraic properties of such structures are is still under investigation, and we believe that such a denotational understanding may provide valuable tools for formal proofs of program properties.

Acknowledgements. We would like to thank Florian Lasinger for pointing us to some problems in the JMM.

References

1. Winskel, G.: Event Structure Semantics of CCS and Related Languages. In Nielsen, M., Schmidt, E.M., eds.: Proc. 9th Int. Coll. Automata, Languages and Programming (ICALP'82). Volume 140 of Lect. Notes Comp. Sci., Springer, Berlin (1982) 561–576
2. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri Nets, Event Structures and Domains: Part I. Theo. Comp. Sci. **13** (1981) 85–108
3. van Glabbeek, R.J., Goltz, U.: Refinement of Actions and Equivalence Notions for Concurrent Systems. Acta Informatica **37** (2001) 229–327
4. Winskel, G.: Event Structures. In Brauer, W., Reisig, W., Rozenberg, G., eds.: Advances in Petri Nets 1986, Part II. Number 255 in Lect. Notes Comp. Sci., Springer, Berlin (1987)
5. van Glabbeek, R.J., Plotkin, G.D.: Configuration Structures. In: Proc. 10th IEEE Symp. Logics in Computer Science (LICS'95), San Diego, IEEE Press (1995) 199–209
6. Cenciarelli, P.: Configuration Theories. In Bradfield, J.C., ed.: Proc. 16th Int. Wsh. Computer Science Logic (CSL'02). Volume 2471 of Lect. Notes Comp. Sci., Springer, Berlin (2002) 200–215
7. van Glabbeek, R.J., Plotkin, G.D.: Event Structures for Resolvable Conflicts. In Fiala, J., Koubek, V., Kratochvíl, J., eds.: Proc. 29th Int. Symp. Mathematical Foundation of Computer Science (MFCS'04). Volume 3153 of Lect. Notes Comp. Sci., Springer, Berlin (2004) 550–561
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
9. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: Proc. 32nd ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL'05), ACM Press (2005) 378–391
10. Winskel, G., Nielsen, M.: Models of Concurrency. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: Handbook of Logic in Computer Science. Vol. 4: Semantic Modelling. Oxford University Press, Oxford (1995) 1–148
11. MacLane, S.: Categories for the Working Mathematician. Springer, New York (1971)
12. Saraswat, V., Jagadeesan, R., Michael, M., von Praun, C.: A Theory of Memory Models (2006) `http://www.saraswat.org/raofull.pdf`$^{(06/12/28)}$.