An Event-Based Structural Operational Semantics of Multi-Threaded Java

Pietro Cenciarelli*, Alexander Knapp, Bernhard Reus, and Martin Wirsing

Ludwig–Maximilians–Universität München
{cenciare,knapp,reus,wirsing}@informatik.uni-muenchen.de

Abstract A structural operational semantics of a significant sublanguage of Java is presented, including the running and stopping of threads, thread interaction via shared memory, synchronization by monitoring and notification, and sequential control mechanisms such as exception handling and return statements. The operational semantics is parametric in the notion of "event space" [6], which formalizes the rules that threads and memory must obey in their interaction. Different computational models are obtained by modifying the well-formedness conditions on event spaces while leaving the operational rules untouched. In particular, we implement the *prescient stores* described in [10, §17.8] which allow certain intermediate code optimizations, and prove that such stores do not affect the semantics of properly synchronized programs.

1 Introduction

The object-oriented programming language Java offers simple and tightly integrated support for concurrent programming. In Java's model of concurrency multiple *threads of control* run in parallel and exchange information by operating on objects which reside in a shared main memory. A precise informal description of this model is given in the Java language specification [10]. Other notable references are [4] and [12].

This paper presents a formal semantics of a significant sublanguage of Java including the running and stopping of threads, thread interaction via shared memory, synchronization by monitoring and notification, and sequential control mechanisms such as exception handling and return statements. Here we focus on the *dynamic* semantics of Java and leave a detailed treatment of the static, type-related aspects of the language, e.g. class declarations, to a followup paper.

Our semantics is given in the style of Plotkin's structural operational semantics (SOS) [15]. In SOS, which has been used in the past for describing SML [13], evaluation is driven by the syntactic structure of programs. This allows a powerful proof technique for semantic analysis: *structural induction*. The idea inspiring the present work is that the semantics of real concurrent languages such as Java, with complex, interacting control features can be given in full detail by means of simple structural rules.

* Research partially supported by the HCM project CHRX-CT94-0591 "De Stijl."

One of the difficulties in modelling concurrent Java programs consists in capturing the complex interplay of memory and thread actions during execution. Each thread of control has, in Java, a private *working memory* in which it keeps its own working copy of variables that it must use or assign. As the thread executes a program, it operates on these working copies. The main memory contains the master copy of each variable. There are rules about when a thread is permitted or required to transfer the contents of its working copy of a variable into the master copy or vice versa. The process of copying is *asynchronous*. There are also rules which regulate the *locking* and *unlocking* of objects, by means of which threads synchronize with each other. All this is described precisely in [10, §17] in terms of eight kinds of low-level actions: *Use*, *Assign*, *Load*, *Store*, *Read*, *Write*, *Lock*, and *Unlock*. Here is an example of a rule from [10, §17.6, p. 407] involving locks and variables. Let T be a thread, V a variable and L a lock:

> "Between an *Assign* action by T on V and a subsequent *Unlock* action by T on L, a *Store* action by T on V must intervene; moreover, the *Write* action corresponding to that *Store* must precede the *Unlock* action, as seen by the main memory."

These rules impose constraints on any implementation of Java so as to allow a correct exchange of information among threads. On the other hand they intentionally leave much freedom to the implementor, thus permitting certain standard hardware and software techniques to improve the speed and efficiency of concurrent code. Therefore, it is *only* on the given rules that the programmer should rely to predict the possible behaviour of a concurrent program. Likewise, it is only the given rules that should constrain the possible execution traces generated by a correct operational semantics.

The above considerations led us to base our semantics on the notion of *event space*. These correspond roughly to *configurations* in Winskel's *event structures* [21] which are denotational, non-interleaving models of concurrent languages. The use of such structures in (interleaving) operational semantics is new. It allows us to give an abstract, "declarative" account of the Java thread model while retaining the virtues of a structural approach. This description is a straight formal paraphrase of the rules of [10]. Event spaces were introduced in [6], where we showed that their use in modelling multi-threading preserves the naive semantics of "sequential" computations (i.e. computations where one thread interacts synchronously with the memory).

Basing our description of Java on the finely grained notion of event allowed us to observe phenomena which may be not readily seen when more abstract approaches are taken. For example, we realized that the asynchrony of communication between main memory and working memories (viz. the *loose* coupling of *Read* and *Load* actions, and similarly of *Store* and *Write*) is actually *observable* in Java. Let threads θ_1 and θ_2, respectively running the code

```
(θ1)    synchronized(p) { p.y = 2; } a = p.x; b = p.y; c = p.y;
(θ2)    synchronized(p) { p.y = 3; p.y = 100; } p.x = 1;
```

share a main memory in which $\texttt{p.x} = \texttt{p.y} = 0$, and let their working memories be initially empty. No parallel execution of θ_1 and θ_2 in which main and working memories interact *synchronously* would possibly allow the values 1, 2 and 3 to be assigned respectively to `a`, `b` and `c`. Any model of execution not capable of producing a run with this assignment of values, indeed possible as we show in Section 2.3, provides maybe a correct implementation, but cannot be considered correct as semantics of Java.

The operational semantics presented below is *parametric* in the notion of event space. This allows different computational models to be obtained by modifying the well-formedness conditions on event spaces while leaving the operational rules untouched. To show the flexibility of this approach we study the "prescient" store actions introduced in [10, §17.8]. Such actions allow optimizing compilers to perform certain kinds of code rearrangements. A bisimulation is given to prove that such rearrangements preserve the semantics of properly synchronized programs (see also [17]).

Related work. Several other semantics of sublanguages of Java are available in the literature. Much work has also been done on the semantics of the Java Virtual Machine [7, 16, 18]; this is one half of a formal semantics of the language, the other half being a description of a Java-to-Virtual Machine bytecode compiler, not available to date.

In this volume Drossopoulou and Eisenbach [8] give a "small-step" structural operational semantics which covers roughly the sequential part of our sublanguage of Java; their work, which is mainly concerned with proving type soundness, has been formalized by Syme [19]. Von Oheimb and Nipkow [14] also deal with a sequential sublanguage of Java and give a formal proof of type safety. A noteworthy difference between [8] and [14] is that the latter follows a "big-step" approach. In [9] Flatt, Krishnamurthy and Felleisen investigate the semantics of operators for combining Java classes (so-called "mixins"). All these semantics focus on type soundness for a *sequential* portion of Java.

As for multi-threading, non-structural descriptions based on *abstract state machines* (see [11]) are given by Börger and Schulte [5], and by Wallace [20].

Synopsis. Section 2 describes and formalizes the Java memory-threads communication protocol. Section 3 presents our event-based, structural operational semantics of Java. Section 4 studies the notion of prescient store action. Loose ends and future research are discussed in Section 5.

2 Event Spaces

In this section we describe and formalize the memory-threads communication protocol of Java. This is done by writing the rules of [10, §17] as simple logical clauses (Section 2.2) and by adopting them as well-formedness conditions on structures called event spaces (Section 2.4). The latter are used in the operational judgements to constrain the applicability of some operational rules. An example of event space is given in Section 2.3, describing the "1-2-3" parallel run of the threads θ_1 and θ_2 introduced above.

2.1 Actions and events

A formal notion of event is given below in terms of five sets of entities:

- $\{Use, Assign, Load, Store, Read, Write, Lock, Unlock\}$, the action names;
- *Thread_id*, the thread identifiers;
- *Obj*, the objects;
- *LVal*, the left values (or "variables," following [10]) and
- *RVal*, the (right) values.

Intuitively, *Use* and *Assign* actions do just what their names suggest, operating on the private working memories. *Read* and *Load* are used for a loosely coupled copying of data from the main memory to a working memory and dually *Store* and *Write* are used for copying data from a working memory to the main memory. *Lock* and *Unlock* are for synchronizing the access to objects.

Formally, an *action* is either a triple (A, θ, o), where $A \in \{Lock, Unlock\}$, θ is a thread (identifier) and o is an object, or a 4-tuple of the form (A, θ, l, v), where $A \in \{Use, Assign, Load, Store, Read, Write\}$, l is a variable, v is a value and θ is as above. When $A \in \{Use, Assign, Load, Store\}$, the tuple (A, θ, l, v) records that the thread θ performs an A action on l with value v, while, if $A \in \{Read, Write\}$, it records that the main memory performs an A action on l with value v on behalf of θ. If A is *Lock* or *Unlock*, (A, θ, o) records that θ acquires, or respectively relinquishes, a lock on o. Actions with name *Use*, *Assign*, *Load*, *Store*, *Lock* and *Unlock* are called *thread actions*, while *Read*, *Write*, *Lock* and *Unlock* are *memory actions.*

Events are instances of actions, which we think of as happening at different times during execution. We use the same tuple notation for actions and their instances: the context clarifies which one is meant. When no confusion arises we may omit components of an action or event which are not immediately relevant in the context of discourse: so $(Read, l)$ stands for $(Read, \theta, l, v)$, for some θ and v. Given a thread θ, we write $\alpha(\theta)$ for a generic instance of a *thread* action performed by θ. Similarly, $\beta(x)$ indicates a generic instance of a *memory* action involving a location or object x.

2.2 The rules of interaction

Here we formalize the rules of [10, Chapter 17], to which we refer for a detailed discussion. These rules are translated into logical clauses describing the properties of a poset of events called the "poset of discourse." The events of such a poset, which are thought of as occurring in the given order, are meant to record the activity of memory and threads during the execution of a Java program. We assume that every chain of the poset of discourse can be counted monotonically: $a_0 \leq a_1 \leq a_2 \leq \ldots$. The clauses in our formalization have the form:

$$\forall \boldsymbol{a} \in \eta \,.\, (\Phi \Rightarrow ((\exists \boldsymbol{b}_1 \in \eta \,.\, \Psi_1) \vee (\exists \boldsymbol{b}_2 \in \eta \,.\, \Psi_2) \vee \ldots \ (\exists \boldsymbol{b}_n \in \eta \,.\, \Psi_n)))$$

where $\boldsymbol{a}$ and $\boldsymbol{b}_i$ are lists of events, η is the poset of discourse and $\forall \boldsymbol{a} \in \eta \,.\, \Phi$ means that Φ holds for all tuples of events in η matching the elements of $\boldsymbol{a}$ (and similarly for $\exists \boldsymbol{b}_i \in \eta \,.\, \Psi_i$). The clauses are abbreviated by adopting the following conventions: quantification over $\boldsymbol{a}$ is left implicit when all events in $\boldsymbol{a}$ appear in Φ; quantification over $\boldsymbol{b}_i$ is left implicit when all events in $\boldsymbol{b}_i$ appear in Ψ_i. Moreover, a rule of the form $\forall \boldsymbol{a} \in \eta \,.\, (\mathit{true} \Rightarrow \ldots)$ is written $\boldsymbol{a} \Rightarrow (\ldots)$. When the symbols θ and θ' appear in a rule, we always assume that $\theta \neq \theta'$. Similarly for values v and v', and for events a and a'.

The rules are the following: The actions performed by any one thread are totally ordered, and so are the actions performed by the main memory for any one variable or lock [10, §17.2, §17.5].

$$\alpha(\theta), \alpha'(\theta) \Rightarrow \alpha(\theta) \leq \alpha'(\theta) \vee \alpha'(\theta) \leq \alpha(\theta) \tag{1}$$

$$\beta(x), \beta'(x) \Rightarrow \beta(x) \leq \beta'(x) \vee \beta'(x) \leq \beta(x) \tag{2}$$

Hence, the occurrences of any action (A, θ, x) are totally ordered in the poset of discourse. We write $\eta(A, \theta, x)$ the subposet of η including only instances of (A, θ, x).

A *Store* action by θ on l must intervene between an *Assign* by θ of l and a subsequent *Load* by θ of l. Less formally, a thread is not permitted to lose its most recent assign [10, §17.3]:

$$(\mathit{Assign}, \theta, l) \leq (\mathit{Load}, \theta, l) \Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \leq (\mathit{Load}, \theta, l) \tag{3}$$

A thread is not permitted to write data from its working memory back to main memory for no reason [10, §17.3]:

$$(\mathit{Store}, \theta, l) \leq (\mathit{Store}, \theta, l)' \Rightarrow (\mathit{Store}, \theta, l) \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l)' \tag{4}$$

Threads start with an empty working memory and new variables are created only in main memory and are not initially in any thread's working memory [10, §17.3]:

$$(\mathit{Use}, \theta, l) \Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Use}, \theta, l) \vee (\mathit{Load}, \theta, l) \leq (\mathit{Use}, \theta, l) \tag{5}$$

$$(\mathit{Store}, \theta, l) \Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{6}$$

A *Use* action transfers the contents of the thread's working copy of a variable to the thread's execution engine [10, §17.1]:

$$\begin{aligned} &(\mathit{Assign}, \theta, l, v) \leq (\mathit{Use}, \theta, l, v') \Rightarrow \\ &\quad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Assign}, \theta, l)' \leq (\mathit{Use}, \theta, l, v') \vee \\ &\quad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Load}, \theta, l) \leq (\mathit{Use}, \theta, l, v') \end{aligned} \tag{7}$$

$$\begin{aligned} &(\mathit{Load}, \theta, l, v) \leq (\mathit{Use}, \theta, l, v') \Rightarrow \\ &\quad (\mathit{Load}, \theta, l, v) \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Use}, \theta, l, v') \vee \\ &\quad (\mathit{Load}, \theta, l, v) \leq (\mathit{Load}, \theta, l)' \leq (\mathit{Use}, \theta, l, v') \end{aligned} \tag{8}$$

A *Store* action transmits the contents of the thread's working copy of a variable to main memory [10, §17.1]:

$$
\begin{aligned}
&(Assign, \theta, l, v) \leq (Store, \theta, l, v') \Rightarrow \\
&\qquad (Assign, \theta, l, v) \leq (Assign, \theta, l)' \leq (Store, \theta, l, v')
\end{aligned}
\tag{9}
$$

The following rules require some events to be paired in the poset of discourse. Let A and B be posets, and let $f : A \rightleftharpoons B$ indicate that a function f is either a monotonic injection $A \to B$ with downward closed codomain or the *partial* inverse of a monotonic injection $B \to A$ with downward closed codomain. For every poset η satisfying (1) and (2), for every thread θ, left value l and object o, there exist unique functions

$$
\begin{aligned}
read\_of_{\eta,\theta,l} &: \eta(Load, \theta, l) \rightleftharpoons \eta(Read, \theta, l) \\
store\_of_{\eta,\theta,l} &: \eta(Write, \theta, l) \rightleftharpoons \eta(Store, \theta, l) \\
lock\_of_{\eta,\theta,o} &: \eta(Unlock, \theta, o) \rightleftharpoons \eta(Lock, \theta, o).
\end{aligned}
$$

These are called the "pairing" functions. Indices are omitted when understood. The function *read_of* matches the n-th occurrence of $(Load, \theta, l)$ in η with the n-th occurrence of $(Read, \theta, l)$ if such an event exists in η and is undefined otherwise. Similarly for *store_of* and *lock_of*.

Each *Load* or *Write* action is uniquely paired with a preceding *Read* or *Store* action respectively. Matching actions bear identical values [10, §17.2, §17.3]:

$$(Load, \theta, l, v) \Rightarrow (Read, \theta, l, v) = read\_of(Load, \theta, l, v) \leq (Load, \theta, l, v) \tag{10}$$

$$(Write, \theta, l, v) \Rightarrow (Store, \theta, l, v) = store\_of(Write, \theta, l, v) \leq (Write, \theta, l, v) \tag{11}$$

Rules (10) and (11) ensure that *read_of* and *store_of* are total. We call *load_of* and *write_of* their partial inverses.

The actions on the master copy of any given variable on behalf of a thread are performed by the main memory in exactly the order that the thread requested [10, §17.3]:

$$(Store, \theta, l) \leq (Load, \theta, l) \Rightarrow write\_of(Store, \theta, l) \leq read\_of(Load, \theta, l) \tag{12}$$

A thread is not permitted to unlock a lock it does not own [10, §17.5]:

$$(Unlock, \theta, o) \Rightarrow lock\_of(Unlock, \theta, o) \leq (Unlock, \theta, o) \tag{13}$$

Rule (13) ensures that *lock_of* is total. We write *unlock_of* its partial inverse.

Only one thread at a time is permitted to lay claim to a lock, and moreover a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of *Unlock* actions have been performed [10, §17.5]:

$$(Lock, \theta, o) \leq (Lock, \theta', o) \Rightarrow unlock\_of(Lock, \theta, o) \leq (Lock, \theta', o) \tag{14}$$

If a thread is to perform an *Unlock* action on any lock, it must first copy all assigned values in its working memory back out to main memory [10, §17.6] (this rule formalizes the quotation in the introduction):

$$\begin{aligned}&(Assign,\theta,l) \le (Unlock,\theta) \Rightarrow \\ &\qquad (Assign,\theta,l) \le store\_of(Write,\theta,l) \le (Write,\theta,l) \le (Unlock,\theta)\end{aligned} \tag{15}$$

A *Lock* action acts as if it flushes all variables from the thread's working memory; before use they must be assigned or loaded from main memory [10, §17.6]:

$$\begin{aligned}&(Lock,\theta) \le (Use,\theta,l) \Rightarrow \\ &\qquad (Lock,\theta) \le (Assign,\theta,l) \le (Use,\theta,l) \vee \\ &\qquad (Lock,\theta) \le read\_of(Load,\theta,l) \le (Load,\theta,l) \le (Use,\theta,l)\end{aligned} \tag{16}$$

$$(Lock,\theta) \le (Store,\theta,l) \Rightarrow (Lock,\theta) \le (Assign,\theta,l) \le (Store,\theta,l) \tag{17}$$

Discussion. Each of the above rules corresponds to one rule in [10]. Note that the language specification requires any *Read* action to be completed by a corresponding *Load* and similarly for *Store* and *Write*. The above theory does not include clauses expressing such requirements because it must capture "incomplete" program executions (see Section 4). Except for read and store completion, any rule in [10] which we have not included above can be derived in our axiomatization. In particular,

$$(Load,\theta,l) \le (Store,\theta,l) \Rightarrow (Load,\theta,l) \le (Assign,\theta,l) \le (Store,\theta,l) \tag{$*$}$$

of [10, §17.3] holds in any model of the axioms. In fact, by (6) there must be some *Assign* action before the *Store*; moreover, one of such *Assign* must intervene in between the *Load* and the *Store*, because otherwise, from (1) and (3), there would be a chain $(Store,\theta,l) \le (Load,\theta,l) \le (Store,\theta,l)$ with no *Assign* in between, which contradicts (4). Similarly, the following rule of [10, §17.3] derives from (10) and (11):

$$(Load,\theta,l) \le (Store,\theta,l) \Rightarrow read\_of(Load,\theta,l) \le write\_of(Store,\theta,l)$$

Clauses (6) and (17) simplify the corresponding rules of [10, §17.3, §17.6] which include a condition $(Load,\theta,l) \le (Store,\theta,l)$ to the right of the implication. This would be redundant because of $(*)$.

2.3 Example

We briefly illustrate the above formal rules on the example given in the introduction, where two threads

(θ_1) `synchronized(p) { p.y = 2; } a = p.x; b = p.y; c = p.y;`
(θ_2) `synchronized(p) { p.y = 3; p.y = 100; } p.x = 1;`

start with a main memory where both instance variables `p.x` and `p.y` have value 0, and with empty working memories, and interact so that the values 1, 2 and 3 are eventually assigned to `a`, `b`, and `c` respectively. We shall run part of this example through our operational rules in Section 3.7. Figure 1 describes this run as a poset of events, whose ordering is represented by the arrows. The actions of the two threads and of the main memory on the two instance variables `p.x` and `p.y` are aligned vertically in four columns. We let o be the object denoted by `p`, while x and y stand for the left values of `p.x` and `p.y` respectively.

Since all actions performed by the same thread and by the memory on the same variable must be totally ordered, each column of Figure 1 is a chain. Moreover, some memory actions must occur before or after some thread actions. For example, a $(\mathit{Write}, \theta_1, y, 2)$ must come after $(\mathit{Assign}, \theta_1, y, 2)$ because, as dictated by the structure of the program, an *Unlock* follows the assignment `p.y` $= 2$, and hence, by (15), θ_1's working copy of y must be written in main memory before the *Unlock* and after a corresponding *Store*. Note that not all the assigned values must be stored in main memory. For example, it would have been legal to omit $(\mathit{Store}, \theta_2, y, 3)$ and $(\mathit{Write}, \theta_2, y, 3)$; in this case, however, the value 3 would have never been passed to θ_1. Similarly, not all the values used by a thread must be first loaded from main memory: in the example no $(\mathit{Load}, \theta_1, y, 2)$ precedes $(\mathit{Use}, \theta_1, y, 2)$.

As stated in the introduction, the above assignments to `a`, `b` and `c` would not be possible if communication between main and working memories where "synchronous," that is if no other event were allowed to happen between a *Read* and a corresponding *Load* or, equivalently, if these two actions were executed as a single atomic step (and similarly for *Store* and *Write*). Assume in fact that there is a synchronous run producing `a` $= 1$, `b` $= 2$, and `c` $= 3$. Since 3 must be assigned to `c`, an action $(\mathit{Read}, \theta_1, y, 3)$ must occur, and moreover it must be after θ_2 writes 3 and before it writes 100 in the master copy of y. Hence, by (15), $(\mathit{Read}, \theta_1, y, 3)$ must occur while θ_2 is executing the synchronized block. Again by (15), a $(\mathit{Store}, \theta_1, y, 2)$ must occur before θ_1 exits its synchronized block; moreover this *Store* must occur before $(\mathit{Read}, \theta_1, y, 3)$, otherwise the value 3 would be lost, and therefore θ_1 must enter its synchronized block before θ_2. Then, in order to get the value 1 for `a`, the assignment `a` $=$ `p.x` must occur after θ_2 has left the block, it has assigned, stored and written 1 in x, and after θ_1 has read and loaded such value in its working copy of x. However, by the time θ_1 can load 1 in x, the value of y in its working memory must already be 3, because a $(\mathit{Read}, \theta_1, y, 3)$ occured while θ_2 was executing the synchronized block. Therefore, to assign 2 to `b`, θ_1 can neither rely on the content of it's working copy of y, nor on the master copy in main memory, which, by now, must contain 100.

2.4 Event spaces

An *event space* is a poset of events every chain of which can be counted monotonically ($a_0 \leq a_1 \leq a_2 \leq \ldots$) and satisfying conditions (1) to (17) of Section 2.2.

Event spaces serve two purposes in our operational semantics: On the one hand they provide all the information needed to reconstruct the working memories (which in fact do not appear in the operational judgements). On the other

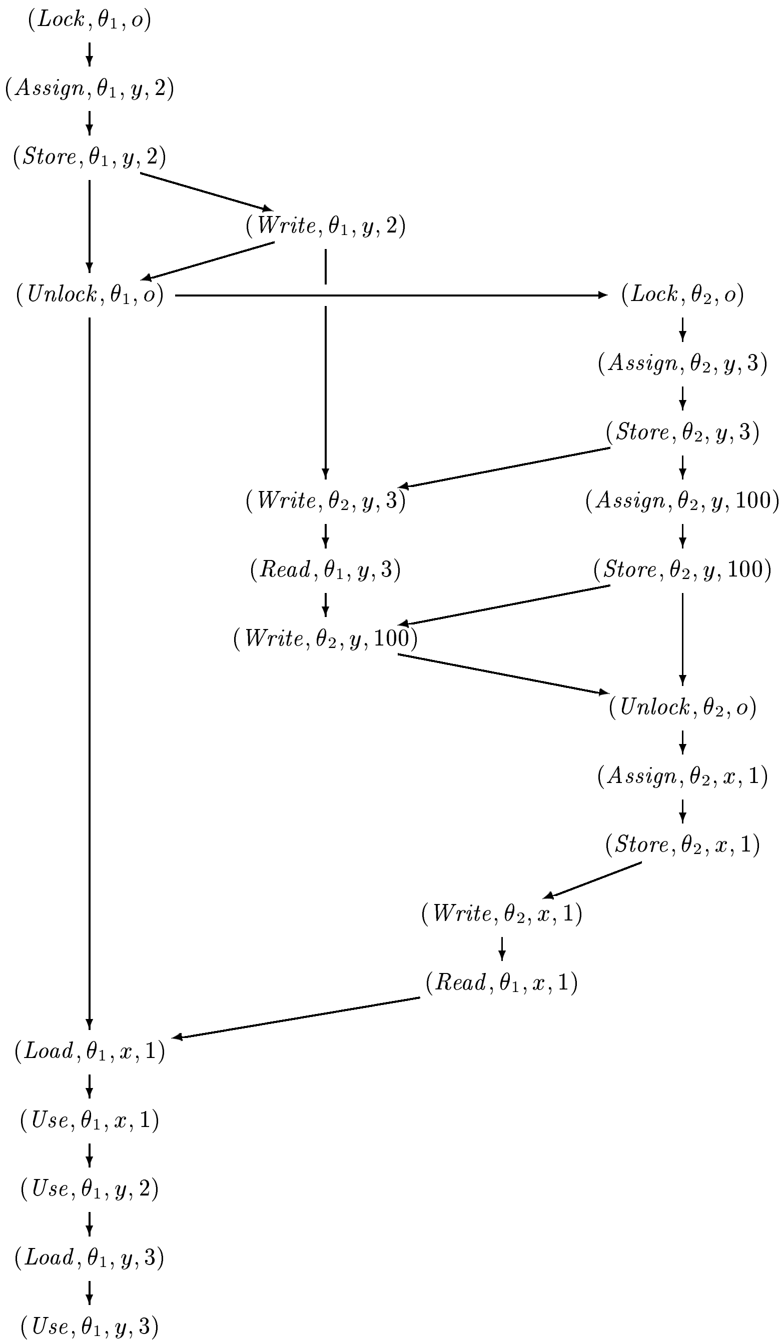

Figure 1. An event space for Example 2.3

hand event spaces record the "historical" information on the computation which constrains the execution of certain actions according to the language specification, and hence the applicability of certain operational rules (see Section 3.4).

Given two event spaces $(X, \leq_X)$ and $(Y, \leq_Y)$, we say that $(X, \leq_X)$ is a *conservative extension* of $(Y, \leq_Y)$ when $Y \subseteq X$ and $\leq_Y \subseteq \leq_X$ and, for all $a, b \in Y$, $a \leq_X b$ implies $a \leq_Y b$.

To adjoin a new event a to an event space $\eta = (X, \leq_X)$, we use an operation $\oplus$ defined as follows: $\eta \oplus a$ denotes nondeterministically an event space $\eta' = (Y, \leq_Y)$ such that:

- η' is a conservative extension of η, with $Y = X \cup \{a\}$;
- if $a = \alpha(\theta)$ is a thread action performed by θ, then $a' \leq_Y a$ for all thread actions $a' = \alpha'(\theta)$ by θ in η';
- if $a = \beta(x)$ is a memory action on x, then $a' \leq a$ for all memory actions $a' = \beta'(x)$ on x in η'.

If no event space η' exists satisfying these conditions, then $\eta \oplus a$ is undefined. For example, by (5), the term $\eta \oplus (\mathit{Use}, \theta, l)$ is defined only if a suitable $(\mathit{Assign}, \theta, l)$ or $(\mathit{Load}, \theta, l)$ occurs in η. If η is an event space and $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$ is a sequence of events, we write $\eta \oplus \boldsymbol{a}$ for $\eta \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$.

As little ordering may be added to an event space by the operation $\oplus$ as is required by the rules of interaction: indeed two expressions $\eta \oplus a \oplus b$ and $\eta \oplus b \oplus a$ may denote the same event space. This reflects the fact that the same concurrent activity may be described by different sequences of interleaved events. *More* ordering can also be introduced than strictly dictated by the rules. For example, the expression $(\mathit{Read}, \theta, o) \oplus (\mathit{Lock}, \theta, l, v) \oplus (\mathit{Load}, \theta, l, v)$ may produce an event space $\{(\mathit{Lock}, \theta, o) \leq (\mathit{Read}, \theta, l, v) \leq (\mathit{Load}, \theta, l, v)\}$: although no rule enforces that $(\mathit{Lock}, \theta, o) \leq (\mathit{Read}, \theta, l, v)$, it better be so in view of rule (16) if a $(\mathit{Use}, \theta, l)$ is to be further added to the space.

3 Operational Semantics

The present paper focuses on the dynamic semantics of Java. Of course, the behaviour of a program may depend on type information obtained from static analysis. Part of this information we assume is retrievable at run-time from the main memory (see Section 3.1), part goes to enrich the syntactic terms upon which the operational semantics operates (see Section 3.2).

In Java every variable and every expression has a type which is known at compile-time. The type limits the possible values that the variable can hold or expression can produce at run-time. Adopting the terminology of [10], every object *belongs* to a class (the class *of* the object, the one which is mentioned when the object is created). Moreover, the values contained by a variable or produced by an expression should, by the design of the language, be *compatible* with the type of the variable or expression. A value of primitive type (such as booleans) is only compatible with that type (`boolean`), while a reference to an object is compatible with any class type which is a superclass of the object's class [10, §4.5.5]. We do not implement run-time compatibility checks in our semantics (they can be added straightforwardly). For example, like in Java, we do not check that the object produced by evaluating the expression e in `throw` e; is compatible with `Throwable`. However, we do use type information wherever it is needed to drive computation. An example is the execution of a `try-catch` statement (see Section 3.8).

Java's *modifiers* are not treated in the present paper. For example, we do not consider *static* fields; these would require minor changes of the semantic machinery. Similarly, *synchronized* methods can be easily implemented by using synchronized statements (see Section 3.7), as remarked in [10, §8.4.3.5].

After introducing in Section 3.1 semantic domains such as stores and environments, we describe a "compilation" function translating Java programs into semantically enriched *abstract syntax* (Section 3.2). Next, we define operational judgements (Section 3.3) and give the SOS rules which generate them. These are presented in homogeneous groups (expressions, statements, exceptions, etc.) in Section 3.4 to 3.10.

3.1 Semantic Domains

Primitive semantic domains. These are the building blocks of our operational semantics, and nothing is assumed on the structure of their elements.

We call *RVal* the primitive domain of (right) values. These are produced by the evaluation of expressions and can be assigned to variables. A distinguished subset *Obj* of *RVal* is also given as primitive; we call its elements (references to) *objects*. In particular, since threads are objects in Java, we choose the domain *Thread_id* of the previous section to be *Obj*. Right values come equipped with a primitive function *value* mapping literals to the corresponding values.

$$\mathit{value} : \mathit{Literal} \to \mathit{RVal}$$

In particular, *null* is the reference to the null object denoted by the literal `null`, that is: $\mathit{null} = \mathit{value}(\texttt{null})$. Similarly, $\mathit{true} = \mathit{value}(\texttt{true})$ and so on.

In Java the object denoted by an expression e may contain several fields with the same name i; then, the type of e decides on which field is actually accessed by the expression $e.i$. An identifier together with a type are therefore a non-ambiguous name for field access. We call *FieldIdentifier*, ranged over by f, the set of such pairs (see Table 1). The domain of left-values introduced in the previous section is *not* primitive: an instance variable is addressed by a non-null object reference o together with a field identifier f, and written $o.f$.

$$\mathit{LVal} = (\mathit{Obj} \setminus \{\mathit{null}\}) \times \mathit{FieldIdentifier}$$

Store is the primitive domain of stores ranged over by μ. This domain comes equipped with the following primitive semantic functions, where *ClassType* is as in Appendix A:

$$\mathit{new} : \mathit{ClassType} \times \mathit{Store} \to \mathit{Obj} \times \mathit{Store}$$

$$\begin{aligned} \mathit{upd} &: \mathit{LVal} \times \mathit{RVal} \times \mathit{Store} \rightharpoonup \mathit{Store} \\ \mathit{rval} &: \mathit{LVal} \times \mathit{Store} \rightharpoonup \mathit{RVal}. \end{aligned}$$

Besides providing storage for variables, stores are assumed to contain information produced by the static analysis of a program; typically: the names and types of fields and methods for each class, the initial values of fields, the subclass relation, and so on. This information does not change during execution and it could alternatively be kept separate from stores.

Given a class type C and a store μ, the function *new* produces a new object of type C with suitably initialized instance variables, and returns it in output together with μ updated with the new object. We write:

$$o \in_\mu C,$$

dropping μ when understood, to mean that o is a reference to an object in μ of a class type which is *compatible* with C. We also assume that the partial function $\mathit{init} : \mathit{FieldIdentifier} \times \mathit{Store} \rightharpoonup \mathit{RVal}$ returns the initial values for an object's fields. The domain of this function is the set of pairs (f, μ) where $f = (i, C)$ and i is an appropriate field for C in μ.

The function *upd* updates a store, while *rval* gets the right-value associated in a store with a given left-value. These functions are partial: they are undefined on the left-values $o.f$ where f is not an appropriate field for o in the given store. We write $\mu[l \mapsto v]$ and $\mu(l)$ for $\mathit{upd}(l, v, \mu)$ and $\mathit{rval}(l, \mu)$ respectively.

A rather weak axiomatization of stores is given below by using a binary predicate $\preceq$ (written infix). The meaning of $e_1 \preceq e_2$ is that if e_1 is defined, then so is e_2 and they denote the same value. By $e_1 \simeq e_2$ we mean that both $e_1 \preceq e_2$ and $e_2 \preceq e_1$ hold.

$$\begin{aligned} \mu(l) &\preceq \mu'(l) && \text{where } \mathit{new}(C, \mu) = (o, \mu') \\ \mathit{init}((i, C), \mu) &\preceq \mu'(o.(i, C)) && \text{where } \mathit{new}(C, \mu) = (o, \mu') \\ \mu[l \mapsto v](l) &\preceq v \\ \mu[l' \mapsto v](l) &\simeq \mu(l) && \text{if } l \neq l' \\ \mu[l \mapsto v'][l \mapsto v] &\preceq \mu[l \mapsto v] \\ \mu[l' \mapsto v'][l \mapsto v] &\simeq \mu[l \mapsto v][l' \mapsto v'] && \text{if } l \neq l' \\ \mu[l \mapsto \mu(l)] &\preceq \mu \end{aligned}$$

Finally, *Throws* is the primitive domain of exceptional results. Upon occurrence of an exception, Java allows objects to be passed to handlers as "reasons" for the exception. The primitive function

$$\mathit{throw} : \mathit{Obj} \to \mathit{Throws}$$

turns an object into an exception $\mathit{throw}(o)$ "with reason o." Note that elements of *Throws* are not right values.

Environments and stacks. *Environments* are pairs (I, ρ) where I is a subset of $\mathit{Identifier} \cup \{\mathit{this}\}$ and ρ is a partial function from I to right values.

$$\mathcal{I} = \mathit{Identifier} \cup \{\mathit{this}\}$$
$$\mathit{Env} = \textstyle\sum_{I \subseteq \mathcal{I}} (I \rightharpoonup \mathit{RVal})$$

The component I of an environment (I, ρ), called the *source* of ρ, is meant to contain the local variables of a block and the formal parameters of a method body or of an exception handler. Environments are also used to store the information on which object's code is currently being executed: $\rho(\mathit{this})$. By abuse of notation, we write ρ for an environment (I, ρ) and indicate with $\mathit{src}(\rho)$ its source I. In particular, we understand that $\rho_\emptyset$ is an empty environment $(I, \rho_\emptyset)$ such that $\rho_\emptyset(i)$ is undefined for all $i \in I$. As usual, $\rho[i \mapsto v](j) = v$ if $i = j$ and $\rho[i \mapsto v](j) \simeq \rho(j)$ otherwise.

Let *Stack* be the domain of stacks of environments, and let the metavariable σ range over this domain. The empty stack is written $\sigma_\emptyset$. The operation $\mathit{push} : \mathit{Env} \times \mathit{Stack} \to \mathit{Stack}$ is the usual one on stacks. An instance variable declaration $i = v$ binds v to i in the topmost environment of a stack σ; we write $\sigma[i = v]$ the result of this operation. The result of assigning v to i in the first environment (I, ρ) of σ such that $i \in I$ is written $\sigma[i \mapsto v]$. The value associated with i in such an environment is denoted by $\sigma(i)$. More precisely:

$$\sigma[i = v] = \begin{cases} \mathit{push}(\rho[i \mapsto v], \sigma') & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \in \mathit{src}(\rho) \\ \mathit{undefined} & \text{otherwise;} \end{cases}$$

$$\sigma[i \mapsto v] = \begin{cases} \mathit{push}(\rho[i \mapsto v], \sigma') & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \in \mathit{src}(\rho) \\ \mathit{push}(\rho, \sigma'[i \mapsto v]) & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \notin \mathit{src}(\rho) \\ \mathit{undefined} & \text{otherwise;} \end{cases}$$

$$\sigma(i) = \begin{cases} \rho(i) & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \in \mathit{src}(\rho) \\ \sigma'(i) & \text{if } \sigma = \mathit{push}(\rho, \sigma') \text{ and } i \notin \mathit{src}(\rho) \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

3.2 Abstract terms

The operational semantics presented below does not work directly on the Java syntax of Appendix A, which we call *concrete*, but on the *abstract* terms produced by the grammar of Table 1. We call *A-Term* the set of abstract terms and let t range over this set. Concrete and abstract syntax share the clauses defining *Identifier*, *Literal*, *ReturnType* and *ClassInstanceCreationExpression*.

Some of the abstract terms, those which cannot be further evaluated, play the role of *results* in our operational semantics. There are operational rules which only apply when a result is produced ([assign4] for example). Some of the results are called *abrupt* (see Section 3.8), as specified by the following grammar:

$$\mathit{Results} ::= * \mid \mathit{RVal} \mid \mathit{AbruptResults}$$
$$\mathit{AbruptResults} ::= \mathit{Throws} \mid \mathit{return}\ \mathit{RVal} \mid \mathit{return}$$

The terms *return* v and *return* are results produced by evaluating return statements, respectively with and without a return value.

In most cases, abstract terms look just like their concrete counterparts. Some abstract terms, however, are enriched with semantic information produced by the static analysis of the Java program. For example, abstract blocks, which we write $\{S\}_\rho$, have two components: a sequence S of (abstract) statements and an environment ρ containing the local variables of the block. We leave ρ implicit when irrelevant.

Unlike with field identifiers, the method invoked by a method call $e.i(\dots)$ is only known at run-time, because it depends not only on the static type C of e but on the dynamic class type of the object denoted by e. At compile-time, however, a "most specific compile-time declaration" is chosen for i among the methods of C and of its superclasses. The class where this declaration is found, the types of the parameters and the return type are attached by the compiler to i for later run-time usage (see [10, §15.11] for more detail). This motivates the introduction of the domain *MethodIdentifier* in the abstract syntax. When the rest is understood, we write just the identifier of a method identifier.

A recursive function $(\_)^\circ$ translates concrete into abstract syntax. Terms of the shared domains are translated into themselves. The concrete list-like syntactic domains, such as *BlockStatements*, are translated in the obvious way into abstract domains of the form $\mathcal{K}^*$ and $\mathcal{K}^+$, where:

$$\begin{aligned}\mathcal{K}^* &::= () \mid \mathcal{K}\,\mathcal{K}^* \\ \mathcal{K}^+ &::= \mathcal{K} \mid \mathcal{K}\,\mathcal{K}^* \ .\end{aligned}$$

Lists that are *optional* in a concrete term are translated into the empty list () when missing. In writing abstract terms we often omit the empty list.

The translation is generally trivial. For example: $(\texttt{throw}\,(e))^\circ = \texttt{throw}\,(e^\circ)$. All non-trivial cases are listed in Table 2. We understand that a "declaration environment" is implicitly carried along during translation, recording the static information collected from processing class declarations. We express that an expression e has declared type τ (in the current declaration environment) by writing $e : \tau$.

Every syntactic domain A-$\mathcal{K}$ of the abstract syntax corresponds to a concrete domain $\mathcal{K}$, and the translation is such that $t \in \mathcal{K}$ whenever $t^\circ \in A$-$\mathcal{K}$. There are syntactic categories in the abstract syntax which have no counterpart in the concrete; these are: *Obj*, *RVal*, *Throws*, *FieldIdentifier*, *MethodIdentifier* and *ActivationFrame*. Of these only the latter is still to be discussed, which we do in Section 3.5.

3.3 Operational Judgements

Configurations. A configuration represents the state of execution of a multi-threaded Java program; therefore, it may include several abstract terms, one for each thread of execution. Each thread has an associated stack. We call *M-term*

$$
\begin{array}{rcl}
\textit{A-Statement} & ::= & * \mid \,;\, \mid \textit{A-Block} \mid \textit{A-StatementExpression}\,; \\
 & & \mid \texttt{synchronized}\ (\,\textit{A-Expression}\,)\ \textit{A-Block} \\
 & & \mid \textit{A-IfThenStatement} \mid \textit{AbruptResults} \\
 & & \mid \texttt{throw}\ \textit{A-Expression}\,; \mid \textit{A-TryStatement} \\
 & & \mid \texttt{return}\,; \mid \texttt{return}\ \textit{A-Expression}\,; \\
\textit{A-Block} & ::= & \{\ \textit{A-BlockStatement}^*\ \}\ \textit{Env} \\
\textit{A-BlockStatement} & ::= & \textit{A-LocalVariableDeclaration}\,; \mid \textit{A-Statement} \\
\textit{A-LocalVariableDeclaration} & ::= & \textit{Type}\ \textit{A-VariableDeclarator}^+ \\
\textit{A-VariableDeclarator} & ::= & \textit{Identifier}\ =\ \textit{A-Expression} \\
\textit{A-Expression} & ::= & \textit{RVal} \mid \textit{Throws} \mid \textit{Literal} \mid \textit{Identifier} \mid \texttt{this} \\
 & & \mid \textit{A-FieldAccess} \mid \textit{ClassInstanceCreationExpression} \\
 & & \mid \textit{A-MethodInvocation} \mid \textit{ActivationFrame} \\
 & & \mid \textit{A-Assignment} \mid \textit{UnaryOperator}\ \textit{A-Expression} \\
 & & \mid \textit{A-Expression}\ \textit{BinaryOperator}\ \textit{A-Expression} \\
\textit{A-FieldAccess} & ::= & \textit{A-Expression}\,.\,\textit{FieldIdentifier} \\
\textit{FieldIdentifier} & ::= & (\textit{Identifier},\ \textit{ClassType}) \\
\textit{A-MethodInvocation} & ::= & \textit{A-Expression}\,.\,\textit{MethodIdentifier}\,(\ \textit{A-Expression}^*\ ) \\
\textit{MethodIdentifier} & ::= & (\textit{Identifier},\ \textit{ClassType},\ \textit{Type}^*,\ \textit{ResultType}) \\
\textit{ActivationFrame} & ::= & (\textit{MethodIdentifier},\ \textit{A-Block}) \\
\textit{A-Assignment} & ::= & \textit{A-LeftHandSide}\ =\ \textit{A-Expression} \\
\textit{A-LeftHandSide} & ::= & \textit{Identifier} \mid \textit{A-FieldAccess} \\
\textit{A-StatementExpression} & ::= & \textit{A-Assignment} \mid \textit{ClassInstanceCreationExpression} \\
 & & \mid \textit{A-MethodInvocation} \mid \textit{ActivationFrame} \\
\textit{A-TryStatement} & ::= & \texttt{try}\ \textit{A-Block}\ \textit{A-CatchClause}^+ \\
 & & \mid \texttt{try}\ \textit{A-Block}\ \textit{A-CatchClause}^*\ \texttt{finally}\ \textit{A-Block} \\
\textit{A-CatchClause} & ::= & \texttt{catch}\ (\ \textit{Type}\ \textit{Identifier}\,)\ \textit{A-Block} \\
\textit{A-IfThenStatement} & ::= & \texttt{if}\ (\ \textit{A-Expression}\ )\ \textit{A-Statement}
\end{array}
$$

Table 1. Abstract syntax

$$
\begin{array}{rcl}
\texttt{\{}\ S\ \texttt{\}}^\circ & = & \{\ S^\circ\ \}_{(I,\rho_\emptyset)} \ \ \text{where } I \text{ is the set of local variables declared in } S \\
(\texttt{catch (}\tau\ i\texttt{)}\ b)^\circ & = & \texttt{catch}\ (\tau\ i)\ \{\ S\ \}_{(I\cup\{i\},\rho_\emptyset)} \ \text{where } \{\ S\ \}_{(I,\rho_\emptyset)} = b^\circ \\
(\texttt{(}e\texttt{)})^\circ & = & e^\circ \\
(e\,\texttt{.}\,i)^\circ & = & e^\circ.f \ \text{where } e : \tau \text{ and } f = (i,\tau) \\
(e\,\texttt{.}\,i\texttt{(}E\texttt{)})^\circ & = & e^\circ.\,m(E^\circ) \ \ \text{where } m = (i, C, \mathcal{T}, \tau) \text{ and the ``compile-time declaration'' of } i \text{ is found in } C \text{ and has signature } \mathcal{T} \to \tau
\end{array}
$$

$$
i^\circ = \begin{cases} i & \text{if } i \text{ appears in the scope of a local variable declaration with that name;} \\ \texttt{this}.f & \text{otherwise, where } \texttt{this} : \tau \text{ and } f = (i,\tau). \end{cases}
$$

Table 2. Translation to abstract syntax

a partial map from thread identifiers to pairs (t, σ), where t is an abstract term and σ is a stack. We let the metavariable T range over M-terms:

$$T : \mathit{Thread\_id} \rightharpoonup \mathit{A\text{-}Term} \times \mathit{Stack}.$$

When we assume that θ is not in the domain of T we write $T \mid (\theta, t, \sigma)$ for the M-term T' such that $T'(\theta) = (t, \sigma)$ and $T'(\theta') \simeq T(\theta')$ for $\theta' \neq \theta$, where $\simeq$ is as in Section 3.1.

A *configuration* of the operational semantics is a triple (T, η, μ) consisting of an M-term T, an event space η and a store μ. In writing 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, σ, η" to mean some configuration $(T \mid (\theta, t, \sigma), \eta, \mu)$. Configurations are ranged over by γ.

Operational semantics. The *operational semantics* is the smallest binary relation $\longrightarrow$ on configurations which is closed under the rules of Section 3.4 to 3.10. These are, in fact, rule *schemes*, whose instances are obtained by replacing the metavariables with suitable semantic objects. Rules with no premise are called *axioms*. Related pairs of configurations are written $\gamma_1 \longrightarrow \gamma_2$ and called *operational judgements* or *transitions*.

Rule conventions. In writing an axiom $\gamma_1 \longrightarrow \gamma_2$ 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 $; \longrightarrow *$ stands for $T \mid (\theta, ;, \sigma), \eta, \mu \longrightarrow T \mid (\theta, *, \sigma), \eta, \mu$. On the other hand, rules with a premise are read by assuming that whatever changes occur in the omitted parts of the premise (besides thread identifiers) also occur in the conclusion (unless otherwise specified). For example, we understand that:

$$\frac{e_1 \longrightarrow e_2}{e_1\,; \longrightarrow e_2\,;} \quad \text{stands for} \quad \frac{T_1 \mid (\theta, e_1, \sigma_1), \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2, \sigma_2), \eta_2, \mu_2}{T_1 \mid (\theta, e_1\,;, \sigma_1), \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2\,;, \sigma_2), \eta_2, \mu_2}\,.$$

Metavariable convention. The metavariables used below (in variously decorated form) in the rule schemes range as follows: $k \in$ *Literal*, $i \in$ *Identifier*, $f \in$ *FieldIdentifier*, $m \in$ *MethodIdentifier*, $o \in$ *Obj*, $l \in$ *LVal*, $v \in$ *RVal*, $V \in$ *RVal**, $e \in$ *A-Expression*, $E \in$ *A-Expression**, $\tau \in$ *Type*, $C \in$ *ClassType*, $d \in$ *A-VariableDeclarator*, $D \in$ *A-VariableDeclarator**, $s \in$ *A-BlockStatement*, $S \in$ *A-BlockStatement**, $b \in$ *A-Block*, $h \in$ *A-CatchClause*, $H \in$ *A-CatchClause**, $c \in$ *Results*, and $q \in$ *AbruptResults*.

3.4 "Silent" actions

We call *Load*, *Store*, *Read* and *Write* the "silent" actions because they may spontaneously occur during the execution of a Java program without the intervention of any thread's execution engine (no term evaluation). In some cases such an occurrence is subject to the previous occurrence of other actions. In the operational semantics, the relevant "historical" information is recorded in a configuration's event space. Note that, given an event space η and an action a, only if $\eta \oplus a$ is defined, and hence the occurrence of a in η complies with the requirements of the language specification, can a rule $\eta \longrightarrow \eta \oplus a$ be fired. This point is crucial for a correct understanding of the rules [read, load, store, write] for silent actions given in Table 3, as well as [assign5, access3] of Table 4 and [syn2, syn4] of Table 8.

The same argument explains how is the [store] rule able to "guess" the right value to be stored: the axioms (6) and (9) of Section 2 guarantee that the apparently arbitrary value v in $\eta \longrightarrow \eta \oplus (Store, \theta, l, v)$ is in fact the latest value assigned by θ to l. In Section 4, changing the event space axioms, we let [store] make a *real* guess on v, by looking "presciently" into the future.

[read][1]	$T, \eta, \mu \longrightarrow T, \eta \oplus (Read, \theta, l, \mu(l)), \mu$
[load][1]	$T, \eta \longrightarrow T, \eta \oplus (Load, \theta, l, v)$
[store][1]	$T, \eta \longrightarrow T, \eta \oplus (Store, \theta, l, v)$
[write][1]	$T, \eta, \mu \longrightarrow T, \eta \oplus (Write, \theta, l, v), \mu[l \mapsto v]$

[1] if $T(\theta)$ is defined

Table 3. "Silent" actions

3.5 Expressions

Table 4 contains the rules for expressions.

To evaluate the assignment to an instance variable successfully, the left hand side is evaluated first by repeatedly applying [assign1], until a left value is produced. Then the right hand side is evaluated by [assign3], and the assignment of the resulting value is recorded in the event space by [assign5]. Note that [assign1] does not apply to an assignment $e_1 = e$ when e_1 is a left value l because, even though l may further evaluate to a right value v by [access3], $v = e$ would not be a legal abstract term. The same argument applies below to rules such as [syn3] and so forth. Note that evaluating *null.f* to $throw(o)$ in rule [access2] would not allow exceptions thrown to the left hand side of an assignment to propagate outward in the structure of the program (see Section 3.8). To wit, $throw(o)$ is an expression result while $throw(o).f$ can be viewed as an "*A-LeftHandSide* result."

The rules [assign2] and [assign4] deal with assignments to local variables. In the present semantics an attempt to access a field of the *null* object raises a `NullPointerException` [access2]. A more elaborate treatment is required when *static* fields are considered (see [10, §15.10.1]).

[assign1]	$\dfrac{e_1 \longrightarrow e_2}{e_1 = e \longrightarrow e_2 = e}$	[assign2]	$\dfrac{e_1 \longrightarrow e_2}{i = e_1 \longrightarrow i = e_2}$
[assign3]	$\dfrac{e_1 \longrightarrow e_2}{l = e_1 \longrightarrow l = e_2}$	[assign4]	$i = v\,, \sigma \longrightarrow v\,, \sigma[i \mapsto v]$
[assign5]	$(\theta, l = v), \eta \longrightarrow (\theta, v), \eta \oplus (Assign, \theta, l, v)$		
[access1]	$\dfrac{e_1 \longrightarrow e_2}{e_1\,.\,f \longrightarrow e_2\,.\,f}$	[access2][1]	$\mathit{null}\,.\,f, \mu \longrightarrow \mathit{throw}(o)\,.\,f, \mu'$
[access3]	$(\theta, l), \eta \longrightarrow (\theta, v), \eta \oplus (Use, \theta, l, v)$		
[this]	$\texttt{this}\,, \sigma \longrightarrow \sigma(\mathit{this})\,, \sigma$	[var]	$i\,, \sigma \longrightarrow \sigma(i)\,, \sigma$
[new]	$\texttt{new}\ C\ \texttt{()}, \mu \longrightarrow \mathit{new}(C, \mu)$	[lit]	$k \longrightarrow \mathit{value}(k)$
[unop1]	$\dfrac{e_1 \longrightarrow e_2}{\texttt{op}\ e_1 \longrightarrow \texttt{op}\ e_2}$	[unop2]	$\texttt{op}\ v \longrightarrow \mathit{op}(v)$
[binop1]	$\dfrac{e_1 \longrightarrow e_2}{e_1\ \texttt{bop}\ e \longrightarrow e_2\ \texttt{bop}\ e}$	[binop2]	$\dfrac{e_1 \longrightarrow e_2}{v\ \texttt{bop}\ e_1 \longrightarrow v\ \texttt{bop}\ e_2}$
[binop3]	$v_1\ \texttt{bop}\ v_2 \longrightarrow \mathit{bop}(v_1, v_2)$		
[parseq1]	$\dfrac{e_1 \longrightarrow e_2}{e_1\ E \longrightarrow e_2\ E}$	[parseq2]	$\dfrac{E_1 \longrightarrow E_2}{v\ E_1 \longrightarrow v\ E_2}$
[call1]	$\dfrac{e_1 \longrightarrow e_2}{e_1.m(E) \longrightarrow e_2.m(E)}$	[call2]	$\dfrac{E_1 \longrightarrow E_2}{o.m(E_1) \longrightarrow o.m(E_2)}$
[call3]	$o.m(V) \longrightarrow \mathit{frame}(o, m, V)$	[call4][1]	$\mathit{null}.m(V), \mu \longrightarrow \mathit{throw}(o), \mu'$

[1] where $(o, \mu') = \mathit{new}(\texttt{NullPointerException}, \mu)$

Table 4. Expressions

[frame]	$\dfrac{b_1 \longrightarrow b_2}{(m, b_1) \longrightarrow (m, b_2)}$	[exit1]	$(m, \{\,\})\,; \longrightarrow *$
[exit2]	$(m, \{\ \mathit{return}\ S\,\})\,; \longrightarrow *$	[exit3]	$(m, \{\ \mathit{return}\ v\ S\,\}) \longrightarrow v$

Table 5. Activation frames

The evaluation of a method invocation $e.m(e_1, \ldots, e_k)$ is done in three steps: First e, $e_1, \ldots, e_k$ are evaluated (in this order). If evaluation is successful, the actual method to be invoked is then determined from m and from the type of the object denoted by e. We deal with non-successful evaluations in Section 3.8. Finally, the actual method call is performed. We assume that the run-time retrieval of methods is performed by a function

$$\mathit{methodBody} : \mathit{ClassType} \times \mathit{MethodIdentifier} \times \mathit{Store} \rightharpoonup \mathit{A\text{-}Block} \times \mathit{Identifier}^*$$

which receives in input the class of the object for which the method is being invoked, a method identifier m and a store (containing the class declarations), and returns, together with the body of m, the list of its formal parameters. This function is partial: $\mathit{methodBody}(C, m, \mu)$, where $m = (i, C', \mathcal{T}, \tau)$, is undefined if no *user-defined* method i with signature $\mathcal{T} \rightarrow \tau$ can be found in μ, inspecting the classes which lie between C and C' in the class hierarchy. In that case m could still be a Java *built-in* method, like `start` or `stop`, otherwise a compile time error would have occured. Separate operational rules are provided for built-in methods (see Table 12 for example). Note that all such rules are subject to the condition that *methodBody* is undefined (which it must be for *final* methods), thus implementing method overriding.

Method calls produce *activation frames*, the elements of *ActivationFrame* in Table 1. The block of a frame represents the body of the invoked method. Activation frames are produced at run-time by the function

$$\mathit{frame} : \mathit{Obj} \times \mathit{MethodIdentifier} \times \mathit{RVal}^* \times \mathit{Store} \rightharpoonup \mathit{ActivationFrame}$$

defined as follows: $\mathit{frame}(o, m, V, \mu) = (m, \{S\}_{\rho[\mathit{this} \mapsto o][I \mapsto V]})$, for an object o of type C, if $\mathit{methodBody}(C, m, \mu) = (\{S\}_\rho, I)$; otherwise it is undefined. Note that, since the type of the null object has no name (see [10, §4.1]), *frame* is always undefined when applied to *null*. Since it is the "static" information contained in μ which is used by *frame*, we generally leave this parameter implicit. The operational rules for evaluating activation frames are given in Table 5.

Start configuration. Let C be the only class in a program called P to be *public*, and let the compilation of P produce an initial store $\mu_\emptyset$ recording all relevant type information. Let C have a method `main` with a string parameter (this is a simplifying assumption: Java requires an *array* of strings, but arrays are not treated in this paper). We understand that a command line "`java` P *arg*" given as input to the computer produces a start configuration

$$(\theta, (\mathtt{main}, \{\, S \,\}_{\rho[i \mapsto v]}), \sigma_\emptyset), \emptyset, \mu$$

where $\emptyset$ is the empty event space, $(\theta, \mu) = \mathit{new}(\mathtt{Thread}, \mu_\emptyset)$, $v = \mathit{value}(\mathit{arg})$, and $\mathit{methodBody}(C, \mathtt{main}, \mu_\emptyset) = (\{\, S \,\}_\rho, i)$.

3.6 Local Variable Declarations

The rules for local variable declarations are given in Table 6.

[decl]	$\dfrac{e_1 \longrightarrow e_2}{\tau\ i = e_1\ D;\ \longrightarrow \tau\ i = e_2\ D;}$
[locvardecl1]	$\tau\ i = v\, d\, D;\ ,\ \sigma \longrightarrow \tau\ d\, D;\ ,\ \sigma[i = v]$
[locvardecl2]	$\tau\ i = v\ ;\ ,\ \sigma \longrightarrow *\ ,\ \sigma[i = v]$

Table 6. Local variable declarations

[expstat1]	$\dfrac{e_1 \longrightarrow e_2}{e_1\ ;\ \longrightarrow e_2\ ;}$	[expstat2]	$v\ ; \longrightarrow *$
[skip]	$; \longrightarrow *$	[if1]	$\dfrac{e_1 \longrightarrow e_2}{\texttt{if}\,(e_1)\ s \longrightarrow \texttt{if}\,(e_2)\ s}$
[if2]	$\texttt{if}\,(\mathit{true})\ s \longrightarrow s$	[if3]	$\texttt{if}\,(\mathit{false})\ s \longrightarrow *$

Table 7. Expression statements, skip and conditional

[statseq]	$\dfrac{s_1 \longrightarrow s_2}{s_1\, S \longrightarrow s_2\, S}$	[$*$]	$*\, S \longrightarrow S$
[block1]	$\{\ \} \longrightarrow *$		
[block2]	$\dfrac{S_1, \mathit{push}(\rho_1, \sigma_1) \longrightarrow S_2, \mathit{push}(\rho_2, \sigma_2)}{\{S_1\}_{\rho_1}, \sigma_1 \longrightarrow \{S_2\}_{\rho_2}, \sigma_2}$		
[syn1][1]	$\dfrac{e_1 \longrightarrow e_2}{\texttt{synchronized}\ (e_1)\ b \longrightarrow \texttt{synchronized}\ (e_2)\ b}$		
[syn2]	$\dfrac{e, \eta_1 \longrightarrow o, \eta_2}{(\theta, \texttt{synchronized}\ (e)\ b), \eta_1 \longrightarrow \texttt{synchronized}\ (o)\ b, \eta_2 \oplus (\mathit{Lock}, \theta, o)}$		
[syn3]	$\dfrac{b_1 \longrightarrow b_2}{\texttt{synchronized}\ (o)\ b_1 \longrightarrow \texttt{synchronized}\ (o)\ b_2}$		
[syn4]	$\dfrac{b, \eta_1 \longrightarrow c, \eta_2}{(\theta, \texttt{synchronized}\ (o)\ b), \eta_1 \longrightarrow c, \eta_2 \oplus (\mathit{Unlock}, \theta, o)}$		

[1] if $e_2 \notin \mathit{RVal}$

Table 8. Blocks and synchronization

3.7 Statements

Table 7 contains the rules for expression statements, skip and conditional statements. Table 8 contains the rules for blocks and synchronization. The statements for control manipulation (`return` and exception `try`) are treated in Section 3.8.

Example. Consider the two threads θ_1 and θ_2 of Example 2.3 running in parallel with initially empty working memories, empty event space $\emptyset$, and stacks mapping the local variable `p` to o. We write t_2 the portion of program run by θ_2. In the example θ_1 enters its synchronized block first. Its evaluation is described in Figure 2, where stacks are omitted.

$$
\begin{array}{c}
(\theta_1,\ \texttt{synchronized}\,(\texttt{p})\,\{\,\texttt{p.y} = 2;\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \emptyset, \mu \\
\text{by [statseq,} \quad \big\downarrow \quad \text{syn2, var]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\,\texttt{p.y} = 2;\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{(\mathit{Lock}, \theta_1, o)\}, \mu \\
\text{by [statseq, syn3, block2, statseq,} \quad \big\downarrow \quad \text{expstat1, assign1, access1, var]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\,o.\texttt{y} = 2;\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{(\mathit{Lock}, \theta_1, o)\}, \mu \\
\text{by [statseq, syn3, block2, statseq,} \quad \big\downarrow \quad \text{expstat1, assign3, lit]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\,o.\texttt{y} = 2;\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{(\mathit{Lock}, \theta_1, o)\}, \mu \\
\text{by [statseq, syn3, block2, statseq,} \quad \big\downarrow \quad \text{expstat1, assign5]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\,2;\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Assign}, \theta_1, o.\texttt{y}, 2)\}, \mu \\
\text{by [statseq, syn3, block2, statseq} \quad \big\downarrow \quad \text{expstat2]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\,*\,\}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Assign}, \theta_1, o.\texttt{y}, 2)\}, \mu \\
\text{by [statseq, syn3, block2,} \quad \big\downarrow \quad *] \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Assign}, \theta_1, o.\texttt{y}, 2)\}, \mu \\
\big\downarrow \quad \text{by [store]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Store}, \theta_1, o.\texttt{y}, 2)\}, \mu \\
\big\downarrow \quad \text{by [write]} \\
(\theta_1,\ \texttt{synchronized}\,(o)\,\{\ \}\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Write}, \theta_1, o.\texttt{y}, 2)\}, \mu[o.\texttt{y} \mapsto 2] \\
\text{by [statseq,} \quad \big\downarrow \quad \text{syn4, block1]} \\
(\theta_1,\ *\,\texttt{a} = \texttt{p.x};\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Unlock}, \theta_1, o)\}, \mu[o.\texttt{y} \mapsto 2] \\
\big\downarrow \quad \text{by } [*] \\
(\theta_1,\ \texttt{a} = \texttt{p.x};\ \ldots) \mid (\theta_2,\ t_2),\ \{\cdots \leq (\mathit{Unlock}, \theta_1, o)\}, \mu[o.\texttt{y} \mapsto 2] \\
(\ldots)
\end{array}
$$

Figure 2. Run of Example 2.3

3.8 Control mechanisms

In Java, the evaluation of expressions and statements may have a *normal* or an *abrupt* completion. Abrupt completion may be caused by the occurrence of an exceptional situation during execution, such as an attempt to divide an integer by 0; it can also be forced by the program by means of a `throw` or a `return` statement. For example, the execution of `throw` e;, where the expression e evaluates to some object o, throws an exception "with reason o" to be caught by the nearest dynamically-enclosing `catch` clause of a `try` statement (see [10, §11.3]). Similarly, the execution of `return` e; returns control, together with the value of e, to the nearest dynamically-enclosing activation frame.

The interactions between these two mechanisms are described in [10, §14.15, §14.16, §14.18], to which we refer for more detail. The rules for exception handling are given in Table 9 and Table 11. Uncaught exceptions are not treated in the present paper.

Some of the rules for the `try` statement include a `finally` clause written in square brackets, to be regarded as "optional:" the brackets indicate that the clause should be ignored if the statement at hand has no `finally` block. A similar convention is adopted for the return statements and results, where `return` $[v]$; accounts for both cases where some value v is and is not returned (and similarly for the results).

Table 10 contains a grammar of syntactic contexts which pop control out upon occurrence of an abrupt evaluation result, with no further ado. Contexts of the form $\vartheta[\_]$, called "pop-out" contexts, are used in the rule scheme [pop] to propagate abrupt evaluation results outwards through the structure of a program. All syntactic constructs which are not represented in a pop-out context respond to such results with some computational action described by a separate semantic rule. Examples of such constructs are the `synchronized` and the `try` statements.

3.9 Starting and stopping threads

The notion of configuration introduced in Section 3.3 is extended here to include a set Θ of thread identifiers, whose elements identify threads which are bound to stop. We write $\Theta \mid \theta$ for $\Theta \cup \{\theta\}$ when we assume that θ is not in Θ. A *configuration* is now redefined to be a 4-tuple of the form:

$$(T, \Theta, \eta, \mu).$$

All operational rules introduced so far have no interaction with the mechanism for stopping threads; in view of the conventions introduced in Section 3.3, by which parts of a configuration may be left implicit when not directly involved in the evaluation, the rules of the previous sections can be read with no editing in the new operational setting with Θ.

Table 12 presents the rules for the methods `start()` and `stop()` of the class `Thread`. The interplay of stopping threads and Java's notification system is discussed in Section 3.10.

$$\text{[pop]}^{1} \qquad \vartheta\,[q] \longrightarrow q$$

$$\text{[exit4]} \qquad (m, \{\, \mathit{throw}(o)\ S\,\}) \longrightarrow \mathit{throw}(o)$$

$$\text{[ret1]} \qquad \frac{e_1 \longrightarrow e_2}{\texttt{return}\ e_1\,; \longrightarrow \texttt{return}\ e_2\,;}$$

$$\text{[ret2]} \qquad \texttt{return}\ [v]\,; \longrightarrow \mathit{return}\,[v]$$

$$\text{[throw1]} \qquad \frac{e_1 \longrightarrow e_2}{\texttt{throw}\ e_1\,; \longrightarrow \texttt{throw}\ e_2\,;}$$

$$\text{[throw2]} \qquad \texttt{throw}\ o\,; \longrightarrow \mathit{throw}\,(o)$$

$$\text{[try1]} \qquad \texttt{try}\ \{\ \}\ H \longrightarrow *$$

$$\text{[try2]} \qquad \texttt{try}\ \{\, \mathit{return}\,[v]\ S\,\}\ H\ [\texttt{finally}\ \{\ \}] \longrightarrow \mathit{return}\,[v]$$

$$\text{[try3]}^{2} \qquad \frac{b_1 \longrightarrow b_2}{\texttt{try}\ b_1\ H\ [\texttt{finally}\ b] \longrightarrow \texttt{try}\ b_2\ H\ [\texttt{finally}\ b]}$$

$$\text{[try4]}^{3} \qquad \frac{b \longrightarrow \{\, \mathit{throw}(o)\ S\,\}}{\begin{array}{l}\texttt{try}\ b\ \texttt{catch}\ (\tau\ i)\ \{\, S'\,\}_{\rho}\ H\ [\texttt{finally}\ b'] \longrightarrow \\ \texttt{try}\ \{\, \mathit{throw}(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\, S'\,\}_{\rho[i\mapsto o]}\ H\ [\texttt{finally}\ b']\end{array}}$$

$$\text{[try5]}^{3} \qquad \frac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b_1\ H\ [\texttt{finally}\ b] \longrightarrow \\ \texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b_2\ H\ [\texttt{finally}\ b]\end{array}}$$

$$\text{[try6]}^{3} \qquad \frac{b \longrightarrow c}{\texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b\ H \longrightarrow c}$$

$$\text{[try7]}^{4} \qquad \frac{b \longrightarrow \{\, \mathit{throw}\,(o)\ S\,\}}{\begin{array}{l}\texttt{try}\ b\ \texttt{catch}\ (\tau\ i)\ b' \longrightarrow \\ \texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b'\end{array}}$$

$$\text{[try8]}^{4} \qquad \frac{\texttt{try}\ b\ H_1\ [\texttt{finally}\ b_1] \longrightarrow \texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ H_2\ [\texttt{finally}\ b_2]}{\begin{array}{l}\texttt{try}\ b\ \texttt{catch}\ (\tau\ i)\ b'\ H_1\ [\texttt{finally}\ b_1] \longrightarrow \\ \texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b'\ H_2\ [\texttt{finally}\ b_2]\end{array}}$$

$$\text{[try9]}^{4} \qquad \texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b\ [\texttt{finally}\ \{\ \}] \longrightarrow \mathit{throw}\,(o)$$

$$\text{[try10]}^{4} \qquad \frac{\texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ H\ [\texttt{finally}\ b] \longrightarrow c}{\texttt{try}\ \{\, \mathit{throw}\,(o)\ S\,\}\ \texttt{catch}\ (\tau\ i)\ b\ H\ [\texttt{finally}\ b] \longrightarrow c}$$

[1] where $\vartheta\,[\,\_\,]$ is a "pop-out" context
[2] if $b_2 \neq \{\, \mathit{throw}(o)\ S\,\}$
[3] if $o \in \tau$
[4] if $o \notin \tau$

Table 9. Exceptions and `return`

$$
\begin{array}{lll}
\vartheta\,[\_] & ::= & [\_]\,.f = e \quad | \quad i = [\_] \quad | \quad l = [\_] \\
 & & |\ \texttt{op}\ [\_] \quad | \quad [\_]\ \texttt{bop}\ e \quad | \quad v\ \texttt{bop}\ [\_] \\
 & & |\ [\_]\,.f \quad | \quad [\_]\,.\,m(E) \quad | \quad o.\,m\,(\xi\,[\_]) \\
 & & |\ \tau\ i = [\_]\,D\,; \quad | \quad [\_]\,; \quad | \quad \{\,[\_]\,S\,\} \\
 & & |\ \texttt{if}\ ([\_])\ s \quad | \quad \texttt{return}\ [\_]\,; \\
 & & |\ \texttt{throw}\ [\_]\,; \quad | \quad \texttt{synchronized}\,([\_])\ b \\
 & & |\ \texttt{try}\ \{\ \}\ H\ \texttt{finally}\ \{\,[\_]\,S\,\} \\
 & & |\ \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{finally}\ \{\,[\_]\,S'\,\} \\
 & & |\ \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,[\_]\,S'\,\}\ H\ \texttt{finally}\ \{\,\}\quad \text{if } o \in \tau \\
 & & |\ \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,q'\,S'\,\}\ H\ \texttt{finally}\ \{\,[\_]\ S''\,\}\quad \text{if } o \in \tau \\
 & & |\ \texttt{try}\ \{\,\mathit{return}\,[v]\ S\,\}\ H\ \texttt{finally}\ \{\,[\_]\,S'\,\} \\
\xi\,[\_] & ::= & [\_]\ E \mid v\ \xi\,[\_]
\end{array}
$$

Table 10. "Pop-out" contexts

[fin1]
$$\frac{b_1 \longrightarrow b_2}{\texttt{try}\ \{\ \}\ H\ \texttt{finally}\ b_1 \longrightarrow \texttt{try}\ \{\ \}\ H\ \texttt{finally}\ b_2}$$

[fin2]
$$\frac{b \longrightarrow c}{\texttt{try}\ \{\ \}\ H\ \texttt{finally}\ b \longrightarrow c}$$

[fin3]
$$\frac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{return}\ [v]\ S\,\}\ H\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{return}\ [v]\ S\,\}\ H\ \texttt{finally}\ b_2\end{array}}$$

[fin4][1]
$$\frac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\ \}\ H\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\ \}\ H\ \texttt{finally}\ b_2\end{array}}$$

[fin5][1]
$$\frac{b \longrightarrow c}{\texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\ \}\ H\ \texttt{finally}\ b \longrightarrow c}$$

[fin6][1]
$$\frac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,q\,S'\,\}\ H\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{catch}\ (\tau\ i)\ \{\,q\,S'\,\}\ H\ \texttt{finally}\ b_2\end{array}}$$

[fin7]
$$\frac{b \longrightarrow \{\,\mathit{throw}\,(o)\,S\,\}}{\texttt{try}\ b\ \texttt{finally}\ b' \longrightarrow \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{finally}\ b'}$$

[fin8]
$$\frac{b_1 \longrightarrow b_2}{\begin{array}{l}\texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{finally}\ b_1 \longrightarrow \\ \texttt{try}\ \{\,\mathit{throw}\,(o)\,S\,\}\ \texttt{finally}\ b_2\end{array}}$$

[1] if $o \in \tau$

Table 11. `finally`

$$[\text{start1}]^{1,2} \qquad \theta.\texttt{start}();, \Theta \longrightarrow * \mid (\theta, \mathit{frame}(\theta, \texttt{run}, ());, \sigma_\emptyset), \Theta$$

$$[\text{start2}]^{1,3} \qquad \theta.\texttt{start}();, \Theta \longrightarrow * \mid (\theta, *, \sigma_\emptyset), \Theta \setminus \{\theta\}$$

$$[\text{start3}]^{1,4,5} \qquad T \mid (\theta', \theta.\texttt{start}();), \mu \longrightarrow T \mid (\theta', \mathit{throw}(o)), \mu'$$

$$[\text{stop1}] \qquad \theta.\texttt{stop}();, \Theta \longrightarrow *, \Theta \cup \{\theta\}$$

$$[\text{stop2}]^{6} \qquad (\theta, t), \Theta \mid \theta, \mu \longrightarrow (\theta, \mathit{throw}(o)), \Theta, \mu'$$

1 if $\mathit{frame}(\theta, \texttt{start}, ())$ is undefined
2 if $\theta \notin \Theta$
3 if $\theta \in \Theta$ or $\mathit{frame}(\theta, \texttt{run}, ())$ is undefined
4 if $T(\theta)$ is defined or $\theta = \theta'$
5 where $(o, \mu') = \mathit{new}(\texttt{IllegalThreadStateException}, \mu)$
6 where t is a redex and $(o, \mu') = \mathit{new}(\texttt{ThreadDeath}, \mu)$

Table 12. `start()` and `stop()`

The rules [start1], [start2] and [start3] can only be applied if the method `start()` has not been overloaded, that is if $\mathit{frame}(\theta, \texttt{start}, ())$ is undefined. Since `stop()` is declared as *final* in class `Thread` and thus cannot be redefined, no analogous side condition is required in the rules for `stop()`. The rule [start1] only applies if no thread with the same identifier as the one to be started is currently running; this is implicit in the use of "|". If such a thread identifier exists an `IllegalThreadStateException` is thrown by [start3].

If a thread θ is started and $\mathit{frame}(\theta, \texttt{run}, ())$ is undefined, the built-in `run` method of the class `Thread` is invoked. The latter simply calls the `run` method of θ's *run object*, that is the runnable object given as argument to the expression that created θ [10, §20.20], if such an object exists, and do nothing otherwise [10, §20.20.13]. Since, for simplicity, we only consider class instance creation expressions with empty parameter list, and hence have no run objects associated with threads, θ does nothing when started if $\mathit{frame}(\theta, \texttt{run}, ())$ is undefined. This explains [start2]. This rule also captures the case of a thread which has been stopped before having ever been started (indeed possible in Java [10, §20.20.15]). If the thread is eventually started, it will immediately terminate and its name removed from Θ.

As a result of the invocation of a `stop` method of class `Thread` an *asynchronous* exception is thrown. Java allows a small but bounded amount of execution to occur between the method call and the actual throw of the exception [10, §11.3.2]. We allow such execution to be arbitrarily long: at any time during execution a thread whose `stop` method has been invoked (by [stop1]) may decide that the time has come to throw a `ThreadDeath` exception. The exception is thrown by [stop2] as deep inside the structure of the program as is necessary to allow a catch by a possibly enclosing `try-catch` statement. This is ensured by the side condition that t is a *redex*. These are the terms of the form:

$$\begin{aligned} \mathit{Redex} ::= \;& i = v \mid l = v \mid \mathit{null}.f \mid \mathit{null}.f = e \mid l \mid \texttt{this} \mid i \mid \texttt{new}\, C\,()\,; \mid k \\ & \mid \texttt{op}\, v \mid v_1\; \texttt{bop}\; v_2 \mid o.m\,(V) \mid \tau\; i = v\; d\, D\,; \mid \tau\; i = v\,; \mid v\,; \mid\, ; \\ & \mid \texttt{if}\,(v)\, s \mid \{\,\} \mid (m, \{\,\}) \mid \texttt{throw}\,(o)\,; \mid \texttt{return}\,[v]\,; \mid \texttt{try}\,\{\,\}\, H \\ & \mid \theta.\texttt{start}\,()\,; \mid \theta.\texttt{stop}\,()\,; \mid o.\texttt{wait}\,()\,; \mid o.\texttt{notify}\,()\,; \end{aligned}$$

As *throw* v and *return* $[v]$ are not contained in this list of redices, a thread cannot stop as long as it is performing a transfer of control, i.e. performing pop-out rules.

A more committed policy for stopping threads may be adopted either by requiring *fairness* on [stop2] or by enforcing such a condition by means of a counter binding the amount of execution steps allowed before this rule is applied.

No rule removes threads from a configuration: when they finish execution, threads keep dwelling in an M-term together with the result that they produced.

3.10 Wait and notification

In Java every object has a "wait set." A thread θ who owns at least one, say n locks on an object o can add itself on that object's wait set by invoking $o.\texttt{wait}()$. This thread would then lose all its locks on o and lie dormant until some other thread wakes it up by invoking $o.\texttt{notify}()$. Before resuming computation, θ must get its n locks back, possibly competing with other threads in the usual manner. When a thread goes to sleep in a wait set it is said to change its *state* from *running* to *waiting*. When notified, such a thread changes its state from *waiting* to *notified*, and finally from *notified* to *running* when it obtains its locks back.

Let the letters R, W and N stand respectively for running, waiting and notified. The notion of M-term introduced in Section 3.3 is extended here by endowing each thread with a *record* of its state. The record of a running thread consists just of the identifier R. The record of a thread which is waiting or notified consists of a triple (X, o, n), where X is the identifier W or N, o is the object on whose wait set the thread is waiting and n is the number of locks that the thread acquired on that object.

An M-term is now redefined to be a partial function mapping thread identifiers to triples (t, ϵ, σ), where t and σ are as before and ϵ is a state record. The notation $T \mid (\theta, t, \epsilon, \sigma)$ extends that of Section 3.3 in the obvious way. When ϵ is a triple (X, o, n) we write $T \mid (\theta, t, X, o, n, \sigma)$ for $T \mid (\theta, t, (X, o, n), \sigma)$ and omit the parts that are not immediately relevant as usual when no confusion arises.

The operational rules introduced so far apply to M-terms of the new form by agreeing that, unless otherwise specified, evaluation applies to running threads (which can nevertheless change state when evaluated). More precisely: if the state record of a thread is omitted in the left hand side of a judgement, then it is understood to be R. For example, [expstat1] is now read

$$\frac{T_1 \mid (\theta, e_1, R, \sigma_1), \Theta_1, \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2, \epsilon, \sigma_2), \Theta_2, \eta_2, \mu_2}{T_1 \mid (\theta, e_1\,;, R, \sigma_1), \Theta_1, \eta_1, \mu_1 \longrightarrow T_2 \mid (\theta, e_2\,;, \epsilon, \sigma_2), \Theta_2, \eta_2, \mu_2}$$

while [skip] is now read $T \mid (\theta, ;, R, \sigma), \Theta, \eta, \mu \longrightarrow T \mid (\theta, *, R, \sigma), \Theta, \eta, \mu$. Moreover, silent actions only apply to running threads; more precisely: the side condition in Table 3 changes now to "if $T(\theta) = (t, R)$." Finally, threads run when started, that is: the state of θ in the right hand side of [start1] and [start2] is R.

[wait1][1]	$(\theta, o.\mathtt{wait}();), \eta, \mu \longrightarrow (\theta, \mathit{throw}(o')), \eta, \mu'$
[wait2][2]	$(\theta, o.\mathtt{wait}();, R), \eta \longrightarrow (\theta, *, W, o, n), \eta \oplus (\mathit{Unlock}, \theta, o)^n$
[notify1][1]	$(\theta, o.\mathtt{notify}();), \eta, \mu \longrightarrow (\theta, \mathit{throw}(o')), \eta, \mu'$
[notify2][2]	$(\theta, o.\mathtt{notify}();) \mid (t, W, o), \eta \longrightarrow (\theta, *) \mid (t, N, o), \eta$
[notify3][3]	$T \mid o.\mathtt{notify}(); \longrightarrow T \mid *$
[ready]	$(\theta, t, N, o, n), \eta \longrightarrow (\theta, t, R), \eta \oplus (\mathit{Lock}, \theta, o)^n$
[stop3]	$(\theta, t, W), \Theta \mid \theta \longrightarrow (\theta, t, N), \Theta \mid \theta$

[1] if $\mathit{locks}(\theta, o, \eta) = 0$ and $(o', \mu') = \mathit{new}(\mathtt{IllegalMonitorException}, \mu)$
[2] if $\mathit{locks}(\theta, o, \eta) = n > 0$
[3] if $T(\theta) \neq (W, o)$ for all θ

Table 13. Wait sets and notification

The rules for the notification system are given in Table 13.

By the rules [wait1] and [notify1], an appropriate exception is thrown if a thread attempts to operate on the wait set of an object on which it possesses no locks. The expression $\mathit{locks}(\theta, o, \eta)$ denotes the number of locks that a thread θ possesses on o in an event space η (the number of events $(\mathit{Lock}, \theta, o)$ with no matching *Unlock*). By the rule [wait2] a thread θ can put itself in the wait set of an object o. This step involves the release by θ of all its locks on o. Rule [notify2] notifies a thread waiting in the wait set of an object o. Such a thread, however, cannot run until all its locks on o are restored. This is done by [ready].

Any notification on an object whose wait set is empty has no effect ([notify3]). A waiting thread which has been stopped is woken up by [stop3].

Example. Figure 3 illustrates the interaction of the rules for wait and notification. Consider the M-term

$$(\theta, \texttt{synchronized(p)} \; \{ \; \texttt{if(c)} \; \texttt{p.wait();} \; \}, \sigma) \mid$$
$$(\theta', \texttt{synchronized(p)} \; \{ \; \texttt{p.notify();} \; \}, \sigma').$$

Let $t = \texttt{synchronized}(o) \; \{ * \}$ and $t' = \texttt{synchronized(p)} \; \{ \; \texttt{p.notify();} \; \}$, let $\emptyset$ be the empty event space and $\eta = \{(\mathit{Lock}, \theta, o) \leq (\mathit{Unlock}, \theta, o)\}$; let σ and σ' be stacks with $\sigma(\texttt{p}) = \sigma'(\texttt{p}) = o$ and $\sigma(\texttt{c}) = \mathit{true}$. The stacks, which do not change during execution, are omitted in the figure.

4 Prescient Event Spaces

The aim of this section is the formalization of the so-called "prescient stores" of [10, §17.8] in our event space semantics. The specification claims that the "prescient" semantics is conservative for "properly synchronized" programs. We also formalize the intuitive notion of "proper synchronization" and prove this claim.

The *prescient* store actions are introduced in [10, §17.8, p. 408] as follows:

> " ... the *store* action [of variable V by thread T is allowed] to instead occur before the *assign* action, if the following restrictions are obeyed:
> - If the *store* action occurs, the *assign* is bound to occur. ...
> - No *lock* action intervenes between the relocated *store* and the *assign*.
> - No *load* of V intervenes between the relocated *store* and the *assign*.
> - No other *store* of V intervenes between the relocated *store* and the *assign*.
> - The *store* action sends to the main memory the value that the *assign* action will put into the working memory of thread T.
>
> The last property inspires us to call such an early *store* action *prescient*: ... "

This section is an improved and corrected version of [17].

4.1 Prescient Event Space Rules

The specification of prescient stores [10, §17.8] seems to assume that it is known which *Store* events are prescient and which prescient *Store* event is matched by which *Assign* event (as if they would be e.g. re-arrangements of *Store* actions in the old sense). We do not assume such knowledge but adopt a more general approach introducing so-called labellings that allow us to use the "old" *Store* and *Assign* events as introduced in Section 2.1 with an additional "labelling" that states whether they are prescient or not. These labellings are not necessarily unique but it is always possible to infer a labelling at run time. It will turn out, however, that the semantics is independent of the choice of labellings, see Corollary 7.

Prescient event spaces are defined, on the one hand, by a relaxation of the event space rules: All rules which forbid prescient stores are cancelled and used instead to define inductively a predicate that tells whether a *Store* event is

$$(\theta,\ \texttt{synchronized(p)}\ \{\ \texttt{if(c)}\ \texttt{p.wait}();\ \}, R) \mid (\theta',\ t', R), \emptyset$$

$$\Big\downarrow\ \text{by [syn2, var]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ \texttt{if(c)}\ \texttt{p.wait}();\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,}\ \Big\downarrow\ \text{if1, var]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ \texttt{if}(\mathit{true})\ \texttt{p.wait}();\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,}\ \Big\downarrow\ \text{if2]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ \texttt{p.wait}();\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,}\ \Big\downarrow\ \text{expstat1, call1, var]}$$

$$(\theta,\ \texttt{synchronized}(o)\ \{\ o\texttt{.wait}();\ \}, R) \mid (\theta',\ t', R), \{(\mathit{Lock}, \theta, o)\}$$

$$\text{by [syn3, block2, statseq,}\ \Big\downarrow\ \text{wait2]}$$

$$(\theta'\ t', R) \mid (\theta, \texttt{syn'd}(o)\ \{\ *\ \}, W, o, 1), \{(\mathit{Lock}, \theta, o) \leq (\mathit{Unlock}, \theta, o)\}$$

$$= \Big|$$

$$(\theta',\ \texttt{synchronized(p)}\ \{\ \texttt{p.notify}();\ \}) \mid (\theta, t, W, o, 1), \eta$$

$$\Big\downarrow\ \text{by [syn2, var]}$$

$$(\theta',\ \texttt{synchronized}(o)\ \{\ \texttt{p.notify}();\ \}, R) \mid (\theta, t, W, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\text{by [syn3, block2, statseq,}\ \Big\downarrow\ \text{expstat1, call1, var]}$$

$$(\theta',\ \texttt{synchronized}(o)\ \{\ o\texttt{.notify}();\ \}, R) \mid (\theta, t, W, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\text{by [syn3, block2, statseq,}\ \Big\downarrow\ \text{notify2]}$$

$$(\theta',\ \texttt{syn'd}(o)\ \{\ *\ \}, R) \mid (\theta, \texttt{syn'd}(o)\ \{\ *\ \}, N, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\text{by [syn3, block2,}\ \Big\downarrow\ *]$$

$$(\theta',\ \texttt{syn'd}(o)\ \{\ \}, R) \mid (\theta, \texttt{syn'd}(o)\ \{\ *\ \}, N, o, 1), \eta \oplus (\mathit{Lock}, \theta', o)$$

$$\Big\downarrow\ \text{by [syn4, block1]}$$

$$(\theta',\ *, R) \mid (\theta, \texttt{synchronized}(o)\ \{\ *\ \}, N, o, 1), \eta \oplus \cdots \oplus (\mathit{Unlock}, \theta', o)$$

$$\Big\downarrow\ \text{by [ready]}$$

$$(\theta',\ *, R) \mid (\theta,\ \texttt{synchronized}(o)\ \{\ *\ \}, R), \eta \oplus \cdots \oplus (\mathit{Lock}, \theta, o)$$

$$\text{by [syn3, block2,}\ \Big\downarrow\ \text{by } *]$$

$$(\theta',\ *, R) \mid (\theta,\ \texttt{synchronized}(o)\ \{\ \}, R), \eta \oplus \cdots \oplus (\mathit{Lock}, \theta, o)$$

$$\Big\downarrow\ \text{by [syn4, block1]}$$

$$(\theta',\ *, R) \mid (\theta,\ *, R), W, \eta \oplus \cdots \oplus (\mathit{Unlock}, \theta, o)$$

Figure 3. Interaction of `wait()` and `notify()`

necessarily prescient. But, on the other hand, we have to add some rules to ensure that a prescient *Store* corresponds to a relocated *Store* that obeyes the old event space rules.

First, we define an abbreviation for the maximal event of type (A, θ, l), irrelevant of its fourth component, occurring before some other event a, and thus write $(A, \theta, l) \leq_L a$ if

$$(A, \theta, l) \leq a \wedge \big((A, \theta, l)' \leq a \Rightarrow (A, \theta, l)' \leq (A, \theta, l)\big)$$

If we write, however, $(\mathit{Store}, \theta, l, v) \leq_L (\mathit{Assign}, \theta, l, v)$, i.e. both events are written with their values and those are *identical*, then we mean the maximal $(\mathit{Store}, \theta, l, v)$ event *with value* v before $(\mathit{Assign}, \theta, l, v)$.

We define $\mathit{prescient}_\eta(\mathit{Store}, \theta, l)$ to be valid if one of the rules (P1–P7) below holds. The subscript η is usually omitted if it is clear from the context. Note that $\Phi \not\Rightarrow \Psi$ abbreviates $\neg(\Phi \Rightarrow \Psi)$ where we use the conventions of Section 2.2, i.e. $\neg(\Phi \Rightarrow \Psi)$ is short for $\neg\forall \boldsymbol{a} \,.\, (\Phi \Rightarrow \exists \boldsymbol{b} \,.\, \Psi)$ where $\boldsymbol{a}$ and $\boldsymbol{b}$ are lists of events and $\boldsymbol{a}$ contains precisely all events occurring in Φ except the bound $(\mathit{Store}, \theta, l)$ event.

$$(\mathit{Store}, \theta, l)' \leq (\mathit{Store}, \theta, l) \not\Rightarrow (\mathit{Store}, \theta, l)' \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{P1}$$

$$(\mathit{Store}, \theta, l) \not\Rightarrow (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{P2}$$

$$\begin{aligned}&(\mathit{Assign}, \theta, l, v') \leq (\mathit{Store}, \theta, l, v) \not\Rightarrow \\ &\qquad (\mathit{Assign}, \theta, l, v') \leq (\mathit{Assign}, \theta, l)' \leq (\mathit{Store}, \theta, l, v)\end{aligned} \tag{P3}$$

$$(\mathit{Lock}, \theta) \leq (\mathit{Store}, \theta, l) \not\Rightarrow (\mathit{Lock}, \theta) \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Store}, \theta, l) \tag{P4}$$

$$\begin{aligned}&(\mathit{Store}, \theta, l)' \leq (\mathit{Store}, \theta, l) \wedge \mathit{prescient}((\mathit{Store}, \theta, l)') \not\Rightarrow \\ &\qquad (\mathit{Store}, \theta, l)' \leq (\mathit{Assign}, \theta, l) \leq (\mathit{Assign}, \theta, l)' \leq (\mathit{Store}, \theta, l)\end{aligned} \tag{P5}$$

$$\begin{aligned}&(\mathit{Store}, \theta, l, v) \leq_L (\mathit{Assign}, \theta, l, v) \leq_L (\mathit{Load}, \theta, l) \not\Rightarrow \\ &\qquad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Store}, \theta, l, v)' \leq (\mathit{Load}, \theta, l)\end{aligned} \tag{P6}$$

$$\begin{aligned}&(\mathit{Store}, \theta, l, v) \leq_L (\mathit{Assign}, \theta, l, v) \leq_L (\mathit{Unlock}, \theta) \not\Rightarrow \\ &\qquad (\mathit{Assign}, \theta, l, v) \leq (\mathit{Store}, \theta, l, v)' \leq \mathit{write\_of}((\mathit{Store}, \theta, l, v)') \\ &\qquad\quad \leq (\mathit{Unlock}, \theta) \;\; \wedge \neg\mathit{prescient}((\mathit{Store}, \theta, l, v)')\end{aligned} \tag{P7}$$

Rules (P1–P4) are the negations of (4), (6), (9), and (17), respectively, that forbid prescient *Store* events. Rule (P5) is sound because if there is only one $(\mathit{Assign}, \theta, l, v)$ between two stores and the first is *prescient*, then by re-arranging the prescient *Store* two *Store* events would follow each other without a triggering *Assign* in between, which contradicts the old semantics. Rules (P6–P7) ensure that in cases where old event space rules (3) and (15) are violated, still a relocated (i.e. *prescient*) *Store* exists which is responsible for storing the right value. So e.g. (P6) states that if any *Store* between the last *Assign* before a *Load* and the *Load* itself is necessarily prescient, then the last *Store* before the *Assign* must also be prescient and thus responsible for fulfilling old (3) when relocated. Note that it is sufficient to consider the last *Assign* before the *Load* (and the *Unlock*, respectively).

With respect to the other (old) event space laws, we keep rules (1), (2), (5), (7–8), (10–14), and (16).

Rule (3) has to be adapted as follows, allowing prescient *Store*s on the right hand side of an implication:

$$\begin{aligned}&(\mathit{Assign},\theta,l,v) \le_L (\mathit{Load},\theta,l) \Rightarrow\\ &\quad\big((\mathit{Assign},\theta,l,v) \le (\mathit{Store},\theta,l,v) \le (\mathit{Load},\theta,l)\big) \vee\\ &\quad\big((\mathit{Store},\theta,l,v) \le_L (\mathit{Assign},\theta,l,v) \le_L (\mathit{Load},\theta,l)\big)\end{aligned} \tag{3'}$$

and rule (15) analogously:

$$\begin{aligned}&(\mathit{Assign},\theta,l,v) \le_L (\mathit{Unlock},\theta) \Rightarrow\\ &\quad\big((\mathit{Assign},\theta,l,v) \le (\mathit{Store},\theta,l,v) \le \mathit{write\_of}(\mathit{Store},\theta,l,v)\\ &\qquad \le (\mathit{Unlock},\theta) \wedge \neg\mathit{prescient}(\mathit{Store},\theta,l)\big) \quad \vee\\ &\quad\big((\mathit{Store},\theta,l,v) \le_L (\mathit{Assign},\theta,l,v) \le_L (\mathit{Unlock},\theta) \;\wedge\\ &\qquad \mathit{write\_of}(\mathit{Store},\theta,l,v) \le (\mathit{Unlock},\theta)\big)\end{aligned} \tag{15'}$$

Both rules are used in cooperation with (P6–P7). Note that in the left branch of the the disjunction in the conclusion of (3') it is unnecessary to stipulate $\neg\mathit{prescient}(\mathit{Store},\theta,l)$ since this will follow from (NP3) and (18) that will be defined below.

We can also infer which *Store* events are necessarily *not* prescient: We define the predicate $\mathit{non\_prescient}(\mathit{Store},\theta,l)$ on the given event space η to be true if one of the rules (NP1–NP3) is fulfilled.

$$\begin{aligned}&\forall a \in \{(\mathit{Lock}),(\mathit{Load},l),(\mathit{Store},l)\}\,.\,(\mathit{Store},\theta,l,v) < a \not\Rightarrow\\ &\quad(\mathit{Store},\theta,l,v) \le (\mathit{Assign},\theta,l,v) < a\end{aligned} \tag{NP1}$$

$$\begin{aligned}&(\mathit{Store},\theta,l) \le (\mathit{Store},\theta,l)' \wedge \mathit{non\_prescient}((\mathit{Store},\theta,l)') \not\Rightarrow\\ &\quad(\mathit{Store},\theta,l) \le (\mathit{Assign},\theta,l) \le (\mathit{Assign},\theta,l)' \le (\mathit{Store},\theta,l)'\end{aligned} \tag{NP2}$$

$$\begin{aligned}&\big((\mathit{Assign},\theta,l,v) \le_L (\mathit{Unlock},\theta) \wedge\\ &\quad(\mathit{Assign},\theta,l,v) \le (\mathit{Store},\theta,l,v) \le (\mathit{Unlock},\theta)\big) \not\Rightarrow\\ &\quad\big((\mathit{Assign},\theta,l,v) \le (\mathit{Store},\theta,l,v)' \le \mathit{write\_of}((\mathit{Store},\theta,l,v)')\\ &\qquad \le (\mathit{Unlock},\theta) \wedge \neg\mathit{prescient}((\mathit{Store},\theta,l,v)')\big) \vee\\ &\quad\big((\mathit{Store},\theta,l,v)'' \le_L (\mathit{Assign},\theta,l,v) \le_L (\mathit{Unlock},\theta) \wedge\\ &\qquad \neg\mathit{non\_prescient}((\mathit{Store},\theta,l,v)'')\big)\end{aligned} \tag{NP3}$$

Rule (NP1) corresponds to the second, third, and fourth requirement in [10, §17.8] (see top of Section 4), (NP2) to (P5), and (NP3) to (P7). If *Assign* $\le_L$ *Unlock*, such that the *Assign* is the last one before the *Unlock*, then (NP3) says that if all *Store*s in between are prescient but one, then this one is necessarily *non_prescient* if the following holds: There is no matching *Store* before the *Assign* or the last such is *non_prescient*. This is a sound rule, because if the *Store* of discourse were not non-prescient, then one might choose it to be prescient, but then no last matching *Store* would occur before the *Assign* that could be chosen prescient. In such a case the *Assign* would not have been stored before the *Unlock*—not even by a prescient store—and hence the old semantics is not preserved.

Notice that the predicate *prescient* propagates from past to present with the exception of (P6–P7) which in some case needs to look back to the last non-prescient *Store*, whereas *non_prescient* is computed in the opposite direction. Also observe that $\neg \mathit{non\_prescient}(s)$ is *not* equivalent to $\mathit{prescient}(s)$ for a *Store* event s and hence also $\mathit{prescient}(s) \vee \mathit{non\_prescient}(s)$ does not always hold.

Finally, we add the new rule

$$(\mathit{Store}, \theta, l) \Rightarrow \neg\big(\mathit{prescient}(\mathit{Store}, \theta, l) \wedge \mathit{non\_prescient}(\mathit{Store}, \theta, l)\big) \qquad (18)$$

according to the specification of prescient *Store* events. This rule in cooperation with (NP1–NP3) prohibits that prescient *Store*s occur at places ruled out by the specification.

Summing up, a *prescient event space* is a poset of events every chain of which can be counted monotonically and satisfying conditions (1), (2), (3'), (5), (7–8), (10–14), (15'), (16), and (18).

The non-deterministic operation $\oplus$ of Section 2.4 also works for prescient event spaces (the only difference being that it defines a predicate on event spaces that are prescient).

An event space is called *complete* if for all *Read* and *Store* events corresponding *Load* and *Write* events exist (all *load_of* and *write_of* functions are total; see the discussion at the end of Section 2.2). A prescient event space η is called *complete* if additionally for any necessarily prescient $(\mathit{Store}, \theta, l, v)$ there is a subsequent $(\mathit{Assign}, \theta, l, v)$. Note that it makes sense only for the final event space of a reduction sequence to be complete. During execution, the matching *Assign* for a prescient *Store* might not have happened. A complete prescient event space fulfills the first and last requirement in [10, §17.8] (see top of Section 4). A prescient event space Γ is called *completable* if there is a sequence of events $\boldsymbol{a}$ such that $\Gamma \oplus \boldsymbol{a}$ is complete.

4.2 Labellings

According to the definitions above even for complete prescient event spaces there might be a *Store* event s in a given event space for which neither $\mathit{prescient}(s)$ nor $\mathit{non\_prescient}(s)$ is derivable. We define so-called *labellings* which allow to choose to a certain extent which *Store* shall be considered prescient and which not.

For a complete prescient event space η a labelling is a predicate ℓ on *Store*s that obeys rules (L1–L4) below together with a corresponding matching function

$$\mathit{passign\_of}^{\ell}_{\eta,\theta,l} : \{(\mathit{Store}, \theta, l) \in \eta \mid \ell(\mathit{Store}, \theta, l)\} \rightleftharpoons \eta(\mathit{Assign}, \theta, l)$$

that fulfills the axioms (M1–M5). Note that rule (M1) ensures that *passign_of* is total.

$$prescient(s) \Rightarrow \ell(s) \tag{L1}$$

$$non\_prescient(s) \Rightarrow \neg\ell(s) \tag{L2}$$

$$(\mathrm{P5})\,[\ell/prescient] \Rightarrow \ell(Store, \theta, l, v) \tag{L3}$$

$$(\mathrm{NP3})\,[\ell/prescient, \neg\ell/non\_prescient] \Rightarrow \neg\ell(Store, \theta, l, v) \tag{L4}$$

$$\begin{aligned}&(Store, \theta, l, v) \wedge \ell(Store, \theta, l, v) \Rightarrow \\ &\qquad (Store, \theta, l, v) \leq passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)\end{aligned} \tag{M1}$$

$$\begin{aligned}&\forall a \in \{(Lock), (Load, l), (Store, l)\} \,.\, (Store, \theta, l) < a \wedge \ell(Store, \theta, l) \Rightarrow \\ &\qquad passign\_of^{\ell}(Store, \theta, l) \leq a\end{aligned} \tag{M2}$$

$$\begin{aligned}&passign\_of^{\ell}(Store, \theta, l) \leq (Store, \theta, l)' \wedge \neg\ell((Store, \theta, l)') \Rightarrow \\ &\qquad passign\_of^{\ell}(Store, \theta, l) \leq (Assign, \theta, l)' \leq (Store, \theta, l)'\end{aligned} \tag{M3}$$

$$\begin{aligned}&\big(\ (Store, \theta, l, v) \leq_L (Assign, \theta, l, v) \leq_L (Load, \theta, l)\ \wedge \ell(Store, \theta, l, v) \not\Rightarrow \\ &\ (Assign, \theta, l, v) \leq (Store, \theta, l, v)' \leq (Load, \theta, l)\,\big) \\ &\qquad\qquad \Rightarrow passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)\end{aligned} \tag{M4}$$

$$\begin{aligned}&\big(\ (Store, \theta, l, v) \leq_L (Assign, \theta, l, v) \leq_L (Unlock, \theta)\ \wedge \ell(Store, \theta, l, v) \not\Rightarrow \\ &\qquad (Assign, \theta, l, v) \leq (Store, \theta, l, v)' \leq write\_of((Store, \theta, l, v)') \\ &\qquad\qquad \leq (Unlock, \theta)\ \ \wedge \neg\ell((Store, \theta, l, v)')\ \big) \\ &\qquad\qquad\qquad \Rightarrow passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)\end{aligned} \tag{M5}$$

In rule (L3) we use "(P5) [ℓ/*prescient*]" to abbreviate the axiom (P5) where *prescient* is syntactically replaced by ℓ and the (bound) event $(Store, \theta, l, v)$ of (P5) coincides with the one in the conclusion of (L3). The analogous convention applies for (L4). Rule (L3) is necessary to propagate ℓ (as *prescient*) according to (P5), and rule (L4) to propagate $\neg\ell$ (as *non_prescient*) according to (NP3). Observe that one does not need similar rules in order to propagate (P7) and (NP2), since those are already covered by (the contraposition of) rules (L4) and (L3), respectively.

By rule (M3) one can never choose an *Assign* event as matching when its rearrangement would lead to a situation forbidden by the old event space rule (4), i.e. where two *Store* events would follow each other. Rules (M4) and (M5) fix the matching for the prescient *Store* in situations where rules (3') and (15') apply but only the right disjunct in their conclusion is fulfilled. Note that for (M4–M5) the nested implication

$$(\Phi \not\Rightarrow \Psi) \Rightarrow passign\_of^{\ell}(Store, \theta, l, v) = (Assign, \theta, l, v)$$

is read with the usual conventions for $\not\Rightarrow$ but $(Store, \theta, l, v)$ and $(Assign, \theta, l, v)$ are obviously universally quantified outermost.

Lemma 1. *For any complete prescient event space one can give a labelling.*

Proof. We choose $\ell := \mathit{prescient}$ and show that it fulfills the labelling rules: (L1) holds by definition of ℓ, (L2) follows from (18), (L3) follows from (P5), and (L4) can be shown by contradiction employing (15'), (P7), and (18).

For any $(\mathit{Store}, \theta, l)$ in the event space with $\ell(\mathit{Store}, \theta, l)$ there is a following matching $(\mathit{Assign}, \theta, l)$ event as the event space of discourse is complete. So for $\mathit{passign\_of}^{\ell}$ we can choose the function which maps any labelled (Store, l) to the last following matching Assign before the first following event $a \in \{\, (\mathit{Load}, l), (\mathit{Lock}), (\mathit{Unlock}), (\mathit{Store}, l)\,\}$, unless $a = (\mathit{Store}, l)$ and $\neg\ell(\mathit{Store}, l)$ when the last but one such Assign is chosen which exists by (NP2). Then $\mathit{passign\_of}^{\ell}$ is a matching function by definition.

4.3 Prescient Operational Semantics

We obtain the prescient operational semantics from the old semantics of Section 3 just by switching from the event spaces of Section 2 to the prescient event spaces of Section 4 keeping the operational rules untouched.

For the prescient operational semantics we write $\longrightarrow\!\!\triangleright$. Moreover, let $\mathit{Conf}_{\triangleright}$ denote the set of configurations with prescient event spaces, and $\mathit{Conf}_{\blacktriangleright}$ those according to the definition of $\longrightarrow\!\!\blacktriangleright$ of Section 3.

Lemma 2. *Any event space η (obeying the old rules) is also a prescient event space, thus any old configuration is a new configuration, i.e., $\mathit{Conf}_{\blacktriangleright} \subseteq \mathit{Conf}_{\triangleright}$, and any reduction $\Gamma \longrightarrow\!\!\blacktriangleright \Gamma'$ is also a prescient one, i.e. $\Gamma \longrightarrow\!\!\triangleright \Gamma'$ holds as well.*

Proof. Assume η is an event space satisfying the old rules. By a simple induction, $\mathit{prescient}(s)$ never holds for any Store event s in η. Thus η is a prescient event space because the new rules form a subset of the old rules. Since the configurations only differ in the event space definition and the rules of the semantics are not changed at all, the other claims of the lemma now hold trivially.

Since we use labellings our operational semantics is very liberal. It accepts reductions using *Store* events even if it is not clear during execution whether this *Store* event is meant to be prescient or not. In such a case, however, the prescient *Store* is not done as early as possible. Therefore, in practical cases, any *Store* which is not recognized by the rules (P1–P7) can be considered *non_prescient*. This corresponds to choose the labelling to be simply *prescient* (cf. Lemma 1). As a consequence, the labelling can be computed at run time. Due to (P6–P7), however, it is not always possible to detect immediately whether a *Store* is *prescient*, sometimes one has to wait for a *Load*- or *Lock* event to happen. Also the matching can be computed at run-time with a little lookahead, cf. (M4–M5).

By the proof of Lemma 1, however, labellings only exist for complete prescient event spaces, hence, in the rest of the paper, any prescient event space Γ is supposed to be completable. Any completion of Γ has a labelling and though its restriction to Γ does not necessarily give a labelling, because (M1) obviously need not be valid, it is easily checked that all the other rules for labellings still hold. Thus for any completable prescient event space there exists a “partial” labelling, which fulfills only (L1–L4) and (M2–M5). Therefore we can assume that any completable prescient event space is endowed with a fixed (partial) labelling ℓ that, for the sake of simplicity, will be exhibited in form of special action names: $pStore$ and $pAssign$. If $\ell(Store,\theta,l,v)$ holds then $(Store,\theta,l,v)$ is denoted $(pStore,\theta,l,v)$ and analogously for the corresponding $Assign$ we use $pAssign$. This notation contains implicitly all information given by the matching function, since by monotonicity of $passign\_of$ for every $(pStore,\theta,l,v)$ the first subsequent $(pAssign,\theta,l,v)$ must be the matching one.

4.4 Prescient Semantics is conservative

The relation between the “normal” and the “prescient” semantics is described in [10, §17.8, p. 408] as follows:

> “The purpose of this relaxation is to allow optimizing Java compilers to perform certain kinds of code rearrangements that preserve the semantics of properly synchronized programs but might be caught in the act of performing memory actions out of order by programs that are not properly synchronized.”

This has to be formalized in the sequel. The following notation, exemplified for $\longrightarrow$ only, will be used analogously for all kinds of arrows: $\xrightarrow{r}$ denotes a one-step reduction with rule r; if $e = (r_1, \ldots, r_n)$ is a list of rules then $\xrightarrow{e}$ denotes $\xrightarrow{r_1} \ldots \xrightarrow{r_n}$; if the list is irrelevant we write $\longrightarrow^*$. For rules that change the event space we often decorate arrows with actions instead of rule names as the latter are ambiguous.

First, we observe that $\rightarrowtriangle$ and $\longrightarrow$ can not be bisimilar by definition since $\rightarrowtriangle$ permits (prescient) $Store$-actions where $\longrightarrow$ does not. But $\rightarrowtriangle$ cannot even be bisimilar to the reflexive closure of $\longrightarrow$, since simulating a $(pStore,\theta,l)$ and the following $Write$s by void steps leads to inequivalent configurations (since the main memories will contain different values for l).

As a prerequisite for a simulation relation of type $\mathit{Conf}_{\blacktriangleright} \times \mathit{Conf}_{\triangleright}$, we define an equivalence on prescient configurations $\sim \subseteq \mathit{Conf}_{\triangleright} \times \mathit{Conf}_{\triangleright}$ as follows:

$$(T,\eta,\mu) \sim (T',\eta',\mu') \iff T = T' \wedge (T,\eta,\sigma,\mu) \downarrow (T',\eta',\sigma',\mu')$$
$$(T,\eta,\mu) \downarrow (T',\eta',\mu') \iff \forall \boldsymbol{a} \,.\, \eta \oplus \boldsymbol{a}{\downarrow} \Leftrightarrow \eta' \oplus \boldsymbol{a}{\downarrow} \;\wedge$$
$$\forall e.\, (T,\eta,\mu) \overset{e}{\rightarrowtriangle}{}^{c} (T_1,\eta_1,\mu_1) \wedge (T',\eta',\mu') \overset{e}{\rightarrowtriangle}{}^{c} (T_2,\eta_2,\mu_2) \Rightarrow \mu_1 = \mu_2$$

where $\boldsymbol{a}$ is any sequence of actions, e is a sequence of rules and $(T,\eta,\mu) \rightarrowtriangle^{c} (T',\eta',\mu')$ if $(T,\eta,\mu) \rightarrowtriangle^{*} (T',\eta',\mu')$ such that η' is complete. (For the sake of simplicity we do not consider the extended configurations of Section 3.10.)

This equivalence relation is obviously preserved by the rules of the semantics:

Lemma 3. *The relation $\sim$ is an equivalence relation such that if $\Gamma_1 \sim \Gamma_2$ then $\Gamma_1 \overset{r}{\rightarrowtriangle} \Gamma_1'$ iff $\Gamma_2 \overset{r}{\rightarrowtriangle} \Gamma_2'$ for any rule r, and if such a reduction r exists then $\Gamma_1' \sim \Gamma_2'$ holds.*

In order to establish a bisimulation result, we must delay all the operations which are possible due to a $(pStore, \theta, l, v)$ until the matching $pAssign$ event.

But that will not work for all kinds of programs. Consider the following example:

$$(\theta,\ \{\ \texttt{synchronized(p)}\ \{\ \texttt{p.x} = 1;\}\ \},\ \sigma)\ \mid\ (\theta',\ \{\ \texttt{p.x} = 2;\},\ \sigma')$$

with $\sigma(\texttt{p}) = \sigma'(\texttt{p}) = o$ and $l = o.\texttt{x}$. Its execution may give rise to a sequence of computation steps which contains the following complete subsequence of actions:

$$\begin{aligned}&(Lock, \theta, o), (Assign, \theta, l, 1), (Store, \theta, l, 1), (pStore, \theta', l, 2),\\ &\quad (Write, \theta', l, 2), (Write, \theta, l, 1), (Unlock, \theta, o), (pAssign, \theta', l, 2)\end{aligned}$$

In a simulation the $(pStore, \theta', l, 2)$ is illegal w.r.t. to the old event space definition and can only be simulated by a void (i.e. delaying) step as well as the following *Write*. Now the $(Write, \theta, l, 1)$ and the corresponding $(Store, \theta, l, 1)$ are bound to occur before the *Unlock*. Finally, after the *pAssign* we must recover the pending prescient $(Store, \theta', l, 2)$ and its corresponding $(Write, \theta', l, 2)$. According to this simulation, l has value 2 in the global memory but the reduction via $\longrightarrow\!\!\!\triangleright$ yields 1 for l. Thus, both end-configurations are not equivalent.

Consequently, we have to restrict ourselves to "properly synchronized" programs. A multi-threaded program T is called *properly synchronized* if any (prescient) event space in any possible configuration occurring during reduction fulfills the following axiom:

$$\begin{aligned}&(Assign, \theta, l), (Assign, \theta', l) \Rightarrow\\ &\quad (Assign, \theta, l) \leq (Unlock, \theta, o) \leq (Lock, \theta', o) \leq (Assign, \theta', l)\end{aligned} \tag{19}$$

where the *Assign*s may correspond to prescient *Store* actions. Analogously, an event space is called properly synchronized if it fulfills (19). A sufficient condition for "properly synchronizedness" is obviously the syntactic criterion that in a program *shared variables* may only be assigned in synchronized blocks.

Proper synchronization guarantees that between a prescient *Store* event and its corresponding *pAssign* event no other thread can change the main memory:

Lemma 4. *Let Γ be a properly synchronized complete prescient event space. If $\theta \neq \theta'$ the following holds:*

$$(pStore, \theta, l) \leq (Write, \theta', l) \Rightarrow passign\_of(pStore, \theta, l) \leq (Write, \theta', l)$$

Proof. Let $(pStore, \theta, l) \leq (Write, \theta', l)$ with $\theta \neq \theta'$ and let $(pAssign, \theta, l) = passign\_of(pStore, \theta, l)$.

First, assume that $store\_of(Write, \theta', l) = (Store, \theta', l) \leq (Write, \theta', l)$, for a non-prescient $(Store, \theta', l)$ such that we have $(Assign, \theta', l) \leq (Store, \theta', l)$ by the negation of (P2). There is a maximal non-prescient $(Assign, \theta', l) \leq_L (Store, \theta', l)$ such that by (P3) the fourth (value-)components of $(Assign, \theta', l)$

and $(Store, \theta', l)$ are equal. Moreover, by (M3) no $(pAssign, \theta', l)$ whatsoever can occur between those two. If now

$$(pAssign, \theta, l) \leq (Unlock, \theta, o) \leq (Lock, \theta', o) \leq (Assign, \theta', l)$$

we obviously have $(pAssign, \theta, l) \leq (Write, \theta', l)$. Otherwise, from properly synchronization, i.e. (19), it follows

$$(Assign, \theta', l) \leq (Unlock, \theta', o) \leq (Lock, \theta, o) \leq (pAssign, \theta, l) \qquad (*)$$

for a suitable $(Unlock, \theta')$–$(Lock, \theta)$ pair. We show that even

$$(Assign, \theta', l) \leq (Store, \theta', l) \leq write\_of(Store, \theta', l) \leq (Unlock, \theta', o) \qquad (**)$$

which proves the lemma since, by the negation of (NP3), we also have

$$(Lock, \theta, o) \leq (pStore, \theta, l) \leq (pAssign, \theta, l)$$

which together with $(*)$ leads to a contradiction to our assumption that $(pStore, \theta, l) \leq (Write, \theta', l)$.

In order to prove $(**)$, first note that

$$\begin{aligned}&(Store, \theta', l) \leq (Unlock, \theta') \Rightarrow \\ &\qquad (Store, \theta', l) \leq write\_of(Store, \theta', l) \leq (Unlock, \theta')\end{aligned}$$

holds in arbitrary prescient event spaces. To see this, it is sufficient to consider the maximal $(Store, \theta', l) \leq_L (Unlock, \theta')$ by monotonicity of $write\_of$. By (P7) and (M4) it is then impossible that there is also another $(Assign, \theta', l)$ or $(pAssign, \theta', l)$ after $(Store, \theta', l)$. There is a maximal $(Assign, \theta', l) \leq_L (Store, \theta', l)$. Between those two events no $(pAssign, \theta', l)$ can occur due to (M3), hence (15') is applicable and we are done.

For a proof of $(**)$ by contradiction, assume that

$$(Assign, \theta', l) \leq (Unlock, \theta', o) \leq (Store, \theta', l)$$

such that $(Assign, \theta', l) \leq_L (Unlock, \theta', o)$ follows. Then by (15') we have

$$(Assign, \theta', l) \leq (Store, \theta', l)' \leq write\_of((Store, \theta', l)') \leq (Unlock, \theta')$$

since if we only had

$$(pStore, \theta', l, v) \leq_L (Assign, \theta', l, v) \leq (Unlock, \theta', o) \leq (Store, \theta', l)$$

the matching rule (M5) would be violated. By (P1), however, there must exist a $(pAssign, \theta', l)$ event such that

$$(Store, \theta', l)' \leq (pAssign, \theta', l) \leq (Store, \theta', l).$$

which contradicts the assumed maximality of $(Assign, \theta', l)$.

The second case that $store\_of(Write, \theta', l) = (pStore, \theta', l) \leq (Write, \theta', l)$ is treated analogously.

In the rest of this subsection we formalize the already sketched simulation idea. To that end, in the sequel Δ (possibly with annotations) stands for configurations in $\mathit{Conf}_{\blacktriangleright}$ and Γ for new configurations in $\mathit{Conf}_{\triangleright}$. Recall that any old configuration is also a valid one in the new sense by Lemma 2. According to the observations above, we define a new reduction relation $\triangleright\!\!\rightarrow\; : (\mathit{Conf}_{\blacktriangleright} \times E^*) \times (\mathit{Conf}_{\blacktriangleright} \times E^*)$ where $E = \{(\mathit{pStore}), (\mathit{Write}), (\mathit{Read})\}$ by the rules of $(\mathrm{red_s})$–$(\mathrm{red_d})$ below. Note that we do not need to treat (Load) events (cf. rule (NP3)). The corresponding $\triangleright\!\!\rightarrow$-configurations (Δ, e) consist of an old configuration $\Delta \in \mathit{Conf}_{\blacktriangleright}$ plus a list of "pending" events e. Appending an event a at the end of a list e is written $e \circ a$. An additional operation $\mathit{split}_{\theta,l}(e)$ is needed. Given a list of events e it yields a pair of lists (e_l, e') where both are sublists of e; the sublist e_l is obtained from e by extracting all $(\mathit{pStore}, \theta, l)$, $(\mathit{Write}, \theta, l)$ and $(\mathit{Read}, \theta', l)$ events and simultaneously changing a $(\mathit{pStore}, \theta, l)$ into $(\mathit{Store}, \theta, l)$; e' is e_l's complement w.r.t. e.

$$(\Delta, e) \xrightarrow{(\mathit{pStore},\theta,l,v)}\!\!\triangleright\; (\Delta, e \circ (\mathit{pStore}, \theta, l, v)) \qquad (\mathrm{red_s})$$

$$\begin{aligned}(\Delta, e) \xrightarrow{(\mathit{Write},\theta,l)}\!\!\triangleright\; (\Delta, e \circ (\mathit{Write}, \theta, l, v)) \quad \text{if} \quad & (\mathit{pStore}, \theta, l, v) \in e \wedge \\ & \mathit{write\_of}(\mathit{pStore}, \theta, l, v) = (\mathit{Write}, \theta, l, v)\end{aligned} \qquad (\mathrm{red_w})$$

$$(\Delta, e) \xrightarrow{(\mathit{Read},\theta',l,v)}\!\!\triangleright\; (\Delta, e \circ (\mathit{Read}, \theta', l, v)) \quad \text{if} \quad (\mathit{Write}, \theta, l) \in e \qquad (\mathrm{red_r})$$

$$\begin{aligned}(\Delta, e) \xrightarrow{(\mathit{pAssign},\theta,l,v)}\!\!\triangleright\; (\Delta', e') \quad \text{if} \quad & \mathit{split}_{\theta,l}(e) = (e_l, e') \wedge \\ & \Delta \xrightarrow{(\mathit{Assign},\theta,l,v)} \Delta_1 \xrightarrow{e_l} \Delta'\end{aligned} \qquad (\mathrm{red_a})$$

$$(\Delta, e) \xrightarrow{r}\!\!\triangleright\; (\Delta', e) \text{ for any other case } r \text{ if } \Delta \xrightarrow{r} \Delta' \qquad (\mathrm{red_d})$$

To relate configurations of $\longrightarrow\!\!\!\triangleright$ and $\triangleright\!\!\rightarrow$ reductions the simulation relation $\approx\, \subseteq \mathit{Conf}_{\triangleright} \times (\mathit{Conf}_{\blacktriangleright} \times E^*)$ is defined as follows:

$$\Gamma \approx (\Delta, e) \quad \text{if, and only if,} \quad (\Delta, e)\!\downarrow \;\wedge\; \Delta \overset{e}{\longrightarrow\!\!\!\triangleright} \Gamma_\Delta \wedge \Gamma_\Delta \sim \Gamma$$

where

$$(\Delta, e)\!\downarrow \quad \text{if, and only if,} \quad \exists \Delta'.\, (\Delta', \varepsilon) \triangleright\!\!\rightarrow^* (\Delta, e)$$

i.e. Γ is equivalent to (Δ, e) if e is obtained correctly by means of $\triangleright\!\!\rightarrow$ and Γ is equivalent to the completion of Δ, usually called Γ_Δ, by executing the pending events in e. Note that $\longrightarrow\!\!\!\triangleright$ is used here for the sequence of events e, as e may contain prescient *Store* events.

Below we use the following notation of a commuting diagram

$$\begin{tikzcd} \Gamma \arrow[r] \arrow[d] \arrow[dr, phantom, "\sim"] & \Gamma_1 \arrow[d] \\ \Gamma_3 \arrow[r] & \Gamma_2 \end{tikzcd}$$

stating that $\Gamma \longrightarrow \Gamma_1 \longrightarrow \Gamma_2$ and $\Gamma \longrightarrow \Gamma_3 \longrightarrow \Gamma_2'$ and $\Gamma_2 \sim \Gamma_2'$. This notation is also used for any other kind of arrows.

Lemma 5. *If $\Gamma \approx (\Delta, e)$ and $\Gamma \overset{r}{\longrightarrow\!\!\!\triangleright} \Gamma'$, where r is as in case* (red$_\mathrm{d}$) *and Γ stems from a properly synchronized program, then $\Delta \xrightarrow{r} \Delta'$ and the diagram*

$$\begin{tikzcd} \Delta \arrow[r, "e"] \arrow[d, "r"] & \Gamma_\Delta \arrow[d, "r"'] & \sim & \Gamma \arrow[d, "r"] \\ \Delta' \arrow[r, "e"] & \Gamma'_\Delta & \sim & \Gamma' \end{tikzcd}$$

commutes; thus $\Gamma \approx (\Delta, e) \overset{r}{\triangleright\!\!\!\longrightarrow} (\Delta', e) \approx \Gamma'$ holds.

Proof. (sketched) First note that if the left square commutes, then the whole diagram commutes by Lemma 3.

Next, observe that r can be executed also before e. For a proof of this check that r does not depend on e by inspecting the relevant laws for event spaces: Rules (5), (16) refer to in-between-events which are not possible in $e \in E^*$. Rules (10) and (3') are impossible since corresponding *Load*s are ruled out by (NP1) and (18). Rule (11) is not relevant as matching *Write*s are treated in (red$_\mathrm{w}$). Thus, we are left with (15'). Suppose $r = (\mathit{Unlock}, \theta)$ and that $(\mathit{pAssign}, \theta, l, v) \leq r$ is ensured via rule (15') by a preceding *Store only* (i.e. the right branch of the disjunction in (15') holds exclusively), then the last of those preceding $(\mathit{Store}, \theta, l, v)$ events is prescient, i.e. $\ell(\mathit{Store}, \theta, l, v)$ holds by (P6). Therefore, $(\mathit{pAssign}, \theta, l, v) = \mathit{passign\_of}^{\ell}(\mathit{Store}, \theta, l, v)$ by (M4) such that e can not contain the *Store* anymore as it is obtained via $\triangleright\!\!\!\longrightarrow^*$.

To prove that the diagram commutes it suffices, by definition of $\sim$, to show that the same actions are executed, but maybe in different order. We have to ensure that *Write* events of the same variable from different threads are not re-ordered. Consider some $(\mathit{Write}, \theta, l)$ of e. By Lemma 4 *Write* events of a different thread θ' can not occur in the completion of Δ, so neither in Γ_Δ and hence neither in e. But e can also never contain two $(\mathit{Write}, \theta, l)$ events, since the first would be the matching *Write* event for the starting *pStore*; the second *Write* event's matching *Store* (maybe prescient) would have to intervene between the starting *pStore* and its corresponding *pAssign* event by the monotonicity of *store_of*, thus contradicting (M2).

Theorem 6. *For properly synchronized programs the relation $\approx$ is a simulation relation of $\longrightarrow\!\!\!\triangleright$ and $\triangleright\!\!\!\longrightarrow$, i.e. if $\Gamma \overset{r}{\longrightarrow\!\!\!\triangleright} \Gamma'$ during the execution of such a program and $\Gamma \approx (\Delta, e)$ then there is a (Δ', e') such that $(\Delta, e) \overset{r}{\triangleright\!\!\!\longrightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e')$.*

Proof. Assume $\Gamma \approx (\Delta, e)$, i.e. $\Delta \overset{e}{\longrightarrow\!\!\!\triangleright} \Gamma_\Delta \sim \Gamma$. We do a case analysis for $\Gamma \overset{r}{\longrightarrow\!\!\!\triangleright} \Gamma'$:

Case $r = (\mathit{Write})$: If $(\mathit{pStore}, \theta, l) \in e$ then it holds that $(\Delta, e) \overset{r}{\triangleright\!\!\!\longrightarrow} (\Delta, e \circ r)$ by (red$_\mathrm{w}$). Moreover, by Lemma 3, $\Gamma' \approx (\Delta, e \circ r)$.

If $(\mathit{pStore}, \theta, l) \notin e$ then by Lemma 5, $(\Delta, e) \overset{r}{\triangleright\!\!\!\longrightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e)$.

Case $r = (pAssign)$: Let $split_{\theta,l}(e) = (e_l, e')$. Since an *Assign* is always possible, assume that $\Delta \xrightarrow{(Assign,\theta,l,v)} \Delta_1$. Now every action in e_l becomes legal for the old semantics, so we can further assume $\Delta_1 \xrightarrow{e_l} \Delta'$, such that $(\Delta, e) \rhd\!\!\!\xrightarrow{r} (\Delta', e')$. One can prove analogously to Lemma 5 that the left rectangle in

$$\begin{tikzcd}
\Delta \arrow[r, "e"] \arrow[d, "{(Assign,\theta,l,v)}"'] & \Gamma_\Delta \arrow[dd, "r"] \arrow[r, phantom, "\sim"] & \Gamma \arrow[dd, "r"] \\
\Delta_1 \arrow[d, "e_l"'] \arrow[r, phantom, "\sim"] & & \\
\Delta' \arrow[r, "e'"] & \Gamma'_\Delta \arrow[r, phantom, "\sim"] & \Gamma'
\end{tikzcd}$$

commutes; the right rectangle commutes by Lemma 3, thus $(\Delta, e) \rhd\!\!\!\xrightarrow{r} (\Delta', e')$ and $\Gamma' \approx (\Delta', e')$.

For *pStore* and *Read* one proceeds as for *Write*, all other cases follow from Lemma 5.

The main result of Section 4 is the following corollary which states that the prescient semantics is conservative, i.e. any prescient execution sequence of a properly synchronized program can be simulated by a "normal" execution of Java.

Corollary 7. *Given $\Gamma \in Conf_{\rhd}$ from a properly synchronized program and $\Delta \in Conf_{\blacktriangleright}$, if $\Gamma \sim \Delta$ and $\Gamma \longrightarrow\!\!\rhd^* \Gamma'$ such that the event space $\eta_{\Gamma'}$ of Γ' is complete, then for any labelling of $\eta_{\Gamma'}$ there is a reduction sequence $\Delta \longrightarrow^* \Delta'$ such that $\Gamma' \sim \Delta'$.*

Moreover, if two different labellings yield two different reduction sequences $\Delta \longrightarrow^* \Delta'_1$ and $\Delta \longrightarrow^* \Delta'_2$, then still $\Delta'_1 \sim \Delta'_2$ holds.

Proof. First, observe that if $\Gamma \sim \Delta$ then $\Gamma \approx (\Delta, \varepsilon)$. By a simple induction on the length of the derivation by Theorem 6, we get $(\Delta, \varepsilon) \rhd\!\!\!\longrightarrow^* (\Delta', e)$ and $\Gamma' \approx (\Delta', e)$. Now $e = \varepsilon$ follows from the fact that Γ' is complete which entails that all prescient stores are matched by an *Assign* such that e must be empty in the end. From $e = \varepsilon$ we immediately get $\Gamma' \sim \Delta'$. Also from $(\Delta, \varepsilon) \rhd\!\!\!\longrightarrow^* (\Delta', \varepsilon)$ we can strip off a derivation $\Delta \longrightarrow^* \Delta'$ by definition of $\rhd\!\!\!\longrightarrow$.

The second claim follows just by transitivity of $\sim$ as $\Delta'_1 \sim \Gamma' \sim \Delta'_2$.

5 Conclusions and Future Work

In this paper we presented a structural operational semantics of concurrent Java and showed its flexibility by proving a non-trivial result relating two memory implementations. Our semantics covers a substantial part of the dynamic behaviour of the language, and we expect it to combine easily with the type system developed in [8]. A further ambitious step is to include in the semantics practical features like input/output, garbage collection, distributed applications via sockets or remote method invocation, and applets.

Event spaces are not necessarily "complete," that is, no matching *Load* must necessarily occur after a *Read* action or *Write* after a *Store*. In fact, there are well-formed event spaces which are not completable, and this complicates the metatheory of the semantics. However, it is conceivable that completability may be axiomatized by means of "local" conditions such as the rules of Section 2.2.

It might also be worthwhile to study stronger notions of "proper synchronization" (for example, by taking into account *Use* actions). This might simplify the simulation of prescient semantics and allow a synchronous treatment of *Read* and *Load*.

The proofs of semantical properties (like Lemma 4 or Theorem 6) are combinatorial in nature; this is a typical situation where proof checkers or automated theorem provers can be usefully employed.

Finally, we intend to investigate operationally based notions of program equivalence, which may serve as foundations for program logics. Abadi and Leino [2] have provided an axiomatic semantics, in Hoare style, for one of the (sequential) object calculi of [1] and proved that the logic is sound with respect to the operational semantics of the object calculus in use. The development of such a logic for a real concurrent object-oriented language like Java remains a challenge.

Acknowledgements. We thank Doug Lea for useful comments and some inspiration regarding future work.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, New York, 1996.
2. Martín Abadi and K. Rustan M. Leino. A Logic of Object-Oriented Programs. In Michel Bidoit and Max Dauchet, editors, *Proc. 7^{th} Int. Conf. Theory and Practice of Software*, volume 1214 of *Lect. Notes Comp. Sci.*, pages 682–696, Berlin, 1997. Springer.
3. Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java*. Lect. Notes Comp. Sci. Springer, Berlin, 1998. This volume.
4. Ken Arnold and James Gosling. *The Java Programming Language*. Addison–Wesley, Reading, Mass., 1996.
5. Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition for the Semantics of Java. In Alves-Foss [3]. This volume.
6. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In Michael Johnson, editor, *Proc. 6^{th} Int. Conf. Algebraic Methodology and Software Technology*, volume 1349 of *Lect. Notes Comp. Sci.*, pages 75–90, Berlin, 1997. Springer.
7. Richard Cohen. The Defensive Java Virtual Machine Specification. Technical report, Computational Logic Inc., 1997. Draft at `http://www.cli.com`.
8. Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In Alves-Foss [3]. This volume.

9. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In Alves-Foss [3]. This volume.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison–Wesley, Reading, Mass., 1996.
11. Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
12. Doug Lea. *Concurrent Programming in Java*. Addison–Wesley, Reading, Mass., 1997.
13. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.
14. Tobias Nipkow and David von Oheimb. Machine-checking the Java Specification: Proving Type-Saftey. In Alves-Foss [3]. This volume.
15. Gordon D. Plotkin. A Structural Approach to Operational Semantics (Lecture notes). Technical Report DAIMI FN–19, Aarhus University, 1981 (repr. 1991).
16. Zhenyu Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. In Alves-Foss [3]. This volume.
17. Bernhard Reus, Alexander Knapp, Pietro Cenciarelli, and Martin Wirsing. Verifying a Compiler Optimization for Multi-Threaded Java. In Francesco Parisi Presicce, editor, *Sel. Papers 12^{th} Int. Wsh. Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lect. Notes Comp. Sci.*, pages 402–417, Berlin, 1998. Springer.
18. Raymie Stata and Martín Abadi. A Type System for Java Bytecode Subroutines. In *Proc. 25^{th} ACM Symp. Principles of Programming Languages*, 1998. To appear.
19. David Syme. Proving Java Type Soundness. In Alves-Foss [3]. This volume.
20. Charles Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, University of Michigan, 1997.
21. Glynn Winskel. An Introduction to Event Structures. In Jacobus W. de Bakker, Willem P. de Roever, and Grzegorg Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes Comp. Sci.*, pages 364–397. Springer, Berlin, 1988.

A Syntax

$$
\begin{array}{rcl}
\mathit{Statement} & ::= & \texttt{;} \mid \mathit{Block} \mid \mathit{StatementExpression}\ \texttt{;} \\
 & & \mid \texttt{synchronized(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Block} \\
 & & \mid \texttt{throw}\ \mathit{Expression}\ \texttt{;} \mid \mathit{TryStatement} \\
 & & \mid \texttt{return}\ \mathit{Expression}_{opt}\ \texttt{;} \\
 & & \mid \mathit{IfThenStatement} \\
\mathit{Block} & ::= & \texttt{\{}\ \mathit{BlockStatements}_{opt}\ \texttt{\}} \\
\mathit{BlockStatements} & ::= & \mathit{BlockStatement} \mid \mathit{BlockStatements}\ \mathit{BlockStatement} \\
\mathit{BlockStatement} & ::= & \mathit{LocalVariableDeclaration}\ \texttt{;} \mid \mathit{Statement} \\
\mathit{LocalVariableDeclaration} & ::= & \mathit{Type}\ \mathit{VariableDeclarators} \\
\mathit{ReturnType} & ::= & \mathit{Type} \mid \texttt{void} \\
\mathit{Type} & ::= & \mathit{PrimitiveType} \mid \mathit{ClassType} \\
\mathit{PrimitiveType} & ::= & \texttt{boolean} \mid \texttt{int} \mid \ldots \\
\mathit{ClassType} & ::= & \mathit{Identifier} \\
\mathit{VariableDeclarators} & ::= & \mathit{VariableDeclarator} \\
 & & \mid \mathit{VariableDeclarators}\ \texttt{,}\ \mathit{VariableDeclarator} \\
\mathit{VariableDeclarator} & ::= & \mathit{Identifier}\ \texttt{=}\ \mathit{Expression} \\
\mathit{Expression} & ::= & \mathit{AssignmentExpression} \\
\mathit{AssignmentExpression} & ::= & \mathit{Assignment} \mid \mathit{BinaryExpression} \\
\mathit{Assignment} & ::= & \mathit{LeftHandSide}\ \texttt{=}\ \mathit{AssignmentExpression} \\
\mathit{LeftHandSide} & ::= & \mathit{Name} \mid \mathit{FieldAccess} \\
\mathit{Name} & ::= & \mathit{Identifier} \mid \mathit{Name}\ \texttt{.}\ \mathit{Identifier} \\
\mathit{FieldAccess} & ::= & \mathit{Primary}\ \texttt{.}\ \mathit{Identifier} \\
\mathit{Primary} & ::= & \mathit{Literal} \mid \texttt{this} \mid \mathit{FieldAccess} \mid \texttt{(}\ \mathit{Expression}\ \texttt{)} \\
 & & \mid \mathit{ClassInstanceCreationExpression} \\
 & & \mid \mathit{MethodInvocation} \\
\mathit{ClassInstanceCreationExpression} & ::= & \texttt{new}\ \mathit{ClassType}\ \texttt{()} \\
\mathit{MethodInvocation} & ::= & \mathit{Primary}\ \texttt{.}\ \mathit{Identifier}\texttt{(}\ \mathit{ArgumentList}_{opt}\ \texttt{)} \\
\mathit{ArgumentList} & ::= & \mathit{Expression} \mid \mathit{ArgumentList}\ \texttt{,}\ \mathit{Expression} \\
\mathit{BinaryExpression} & ::= & \mathit{UnaryExpression} \\
 & & \mid \mathit{BinaryExpression}\ \mathit{BinaryOperator} \\
 & & \quad \mathit{UnaryExpression} \\
\mathit{UnaryExpression} & ::= & \mathit{UnaryOperator}\ \mathit{UnaryExpression} \\
 & & \mid \mathit{Primary} \mid \mathit{Name} \\
\mathit{StatementExpression} & ::= & \mathit{Assignment} \mid \mathit{ClassInstanceCreationExpression} \\
 & & \mid \mathit{MethodInvocation} \\
\mathit{TryStatement} & ::= & \texttt{try}\ \mathit{Block}\ \mathit{Catches} \\
 & & \mid \texttt{try}\ \mathit{Block}\ \mathit{Catches}_{opt}\ \texttt{finally}\ \mathit{Block} \\
\mathit{Catches} & ::= & \mathit{CatchClause} \mid \mathit{CatchClauses}\ \mathit{CatchClause} \\
\mathit{CatchClause} & ::= & \texttt{catch(}\ \mathit{Type}\ \mathit{Identifier}\ \texttt{)}\ \mathit{Block} \\
\mathit{IfThenStatement} & ::= & \texttt{if(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Statement}
\end{array}
$$