Verifying a Compiler Optimization for Multi-Threaded Java

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

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

Abstract. The specification for the object-oriented concurrent language Java [3] is rather loose with respect to the interaction of shared memory and the local working memories of different threads. This allows maximal freedom in the language implementation. Such freedom is reflected in the semantics provided in [2], where threads-memory interaction is formalized in terms of structures called *event spaces*. Two kinds of memories are described in the Java specification: a "normal" memory and a more liberal one, where values can sometimes be stored even before they are produced as results of computation. Here we compare two structural operational semantics of a sublanguage of Java modelling the two types of memory. The two semantics share the same set of operational rules but put different requirements (expressed as first order theories) on the notion of event space. We prove a result which is informally stated in [3]: the two semantics coincide for properly synchronized programs. This shows the applicability of a new technique for combining structural operational semantics and first order specification of process behaviour.

1 Introduction

A concurrent program consists of multiple tasks that are or behave as if they were executed all at the same time. Such tasks can be implemented using *threads* (short for "threads of execution"), which are sequences of instructions that run independently within the encompassing program. The object-oriented language Java supports thread programming (see e.g. [1], [4]).

Java threads share a common memory, but keep working copies of shared variables in private working memories. It is only when leaving a synchronized block that *must* a thread copy the content of its working memory in the main memory. However, possible implementations of the run-time system *may* choose to update the value of a variable in the main memory as soon as a thread makes an assignment to its working copy of that variable. The Java language specification [3] leaves freedom to the implementation in that respect.

A particular implementation technique is also discussed in [3], where a value can be stored by a thread in the main memory before such value is produced by the computation. This is called a *prescient* store action [3, §17.8]. The only restriction is that between the prescient store and the matching assignment nothing "bad" happens, e.g. no other thread reads illegitimately the prescient value. Consider the following code example.

```
o.b = 2;
for(o.a = 1; o.a < 10; o.a = o.a + 1)
  o.b = o.a + o.b;
```

where o is (a reference to) an object with two attributes a and b of type int. When executed the value of o.a can only be stored after o.a has been assigned a new value, i.e. after it was incremented. If prescient store operations are permitted, then it would be also legal to store the value 10 for o.a in advance, i.e. before the loop is entered, excluding thereby that any other thread can load o.a before the end of the loop.

The rearrangement of store operations can be used to speed up programs when updating of variables is split into a thread action (called *Store*) and a memory action (*Write*). The global memory can concurrently provide the value of a pre-stored variable while a second thread waits for it. Re-grouping the store-operations might also optimize memory access itself.

In [2] we present a structural operational semantics (in the style of [5]) of a nontrivial sub-language of Java which includes dynamic creation of objects, blocks, and synchronization of threads. The notion of *event space* is introduced in that paper to formalize the communication protocol between shared memory and threads. Event spaces correspond roughly to *configurations* in Winskel's *event structures* [6], which are used for denotational semantics of concurrent languages.

Here we exploit the flexibility of the approach proposed in [2], where the operational semantics is given parametrically in the notion of event space, and compare two language implementations which share the same set of operational rules. The implementations are obtained by imposing different requirements on event spaces, so that prescient stores are possible in one case and impossible in the other. Such requirements are expressed in simple first order clauses. In this framework we prove that prescient and nonprescient semantics coincide for properly synchronized programs, that is programs where any two threads are not allowed to write a variable into the global memory without synchronization (*race conditions* [4]). This property was only informally stated in [3, §17.8]. We also provide an example where prescient store actions for non-properly synchronized programs lead to inconsistent memory contents.

The contribution of the paper is twofold: on the one hand it provides welcome formal confirmation of the intuitive correctness of certain compiler optimization techniques; on the other hand it shows the applicability of an innovative technique for combining structural operational semantics and first order axiomatisation of process behaviour.

The paper is organized as follows: Section 2 recapitulates the definition of event spaces from [2]. These are used in Section 3 for the SOS-rules. Next, the axiomatization of event spaces is changed (Section 4) in order to allow for prescient stores and prescient operational semantics is defined in Section 5. It is then proven, in Section 6, that for properly synchronized programs the extension is conservative w.r.t. the old semantics.

2 Event Spaces

The execution of a Java program comprises many *threads* of computation running in parallel. Threads exchange information by operating on values and objects residing in a shared *main memory*. As explained in the Java language specification [3], each thread also has 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. Moreover, there are rules which regulate the *locking* and *unlocking* of objects, by means of which threads synchronize with each other. These rules are given in [3, Chapter 17] and have been formalized in [2] as "well-formedness" conditions for structures called *event spaces*. We summarize their definition and usage.

Event spaces will be included in the configurations of multi-threaded Java to constrain the applicability of certain operational rules. Additionally, they will be used to model the working memories of all threads. The main memory is modeled as an abstract store that can be thought of as mapping addresses of instance variables (left-values, from a semantic domain *LVal*) of objects (from a semantic domain *Obj*) to values or object references (right-values, from a semantic domain *RVal*).

In accord with [3], the terms *Use*, *Assign*, *Load*, *Store*, *Read*, *Write*, *Lock*, and *Unlock* are used here to name actions which describe the activity of the memories during the execution of a Java program. *Use* and *Assign* denote the above mentioned actions of the private working memory. *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 (as mentioned in the introduction).

We let the metavariable A (possibly indexed) stand for a generic action name. Moreover, we let B range over the set of thread actions and C over the set of memory actions, that is: $B \in \{Use, Assign, Load, Store, Lock, Unlock\}$, $C \in \{Read, Write, Lock, Unlock\}$.

Let *Thread_id* be a set of thread identifiers. An *action* is either a 4-tuple of the form (A, θ, l, v) where $A \in \{Assign, Store, Read\}$, $\theta \in Thread\_id$, $l \in LVal$ and $v \in RVal$, or a triple (A, θ, l), where θ and l are as above and $A \in \{Use, Load, Write\}$, or a triple (A, θ, o), where $A \in \{Lock, Unlock\}$ and $o \in Obj$.

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) and let a, b, c stand for either. Sometimes we omit components of an action or event: we may write e.g. $(Read, l)$ for $(Read, \theta, l, v)$ when θ and v are not relevant.

An *event space* is a poset of events (thought of as occurring in the given order) in which every chain can be enumerated monotonically with respect to the arithmetical ordering $0 \leq 1 \leq 2 \leq \ldots$ of natural numbers, and which satisfies the conditions (17.2.1–17.6.2') of Table 1. These conditions, which formalize directly the rules of [3, Chapter 17], are expressed by clauses of the form:

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

where $\boldsymbol{a}$ and $\boldsymbol{b}_i$ are lists of events, η is an event space 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 $\exists \boldsymbol{b} \in \eta \,.\, \Psi$). Such statements 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 $\forall \boldsymbol{a} \in \eta \,.\, (\mathit{true} \Rightarrow \dots)$ is written $\boldsymbol{a} \Rightarrow (\dots)$. The term $(A, \theta, x)_n$ denotes the n-th occurrence of (A, θ, x) in a given space, if such an event exists, and is undefined otherwise.

We include the origin of each rule from [3, Chapter 17] and refer to [3] and [2] for more detail. For instance, rule (17.2.1) says that actions performed by any thread are totally ordered and (17.2.2) that so are the actions performed by the main memory for any variable or object. Similarly, rules (17.6.2) and (17.6.2') say that a lock action acts as if it flushes all variables from the thread's working memory, i.e. before use they must be assigned or loaded from main memory.

A *complete event space* is an event space that additionally fulfills the axioms (17.2.6) and (17.2.7) such that any *Read* and *Store* events are "completed" by corresponding *Load* and *Write* events.

A new event $a = (A, \theta, x)$ is adjoined to an event space η by extending the execution order as follows: if A is a thread action, then $b \leq a$ for all instances b of (B, θ) in η; if a is a main memory action, then $c \leq a$ for all instances c of (C, x) in η. Moreover, if A is *Load* then $c \leq a$ for all instances c of $(\mathit{Read}, \theta, x)$ in η, and if A is *Write* then $c \leq a$ for all instances c of $(\mathit{Store}, \theta, x)$ in η. The term $\eta \oplus a$ denotes the space thus obtained, provided it obeys the above rules, and it is otherwise undefined. If it is defined then $\eta \oplus a\!\downarrow$ yields true and false otherwise. If η is an event space and $\boldsymbol{a} = (a_1, a_2, \dots a_n)$ is a sequence of events, we write $\eta \oplus \boldsymbol{a}$ for $\eta \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$ and analogously $\eta \oplus \boldsymbol{a}\!\downarrow$.

3 Operational Semantics

We briefly recapitulate the structural operational semantics of multi-threaded Java in [2]. We restrict ourselves to those parts that are relevant for the "prescient" compiler optimization.

Objects are kept in the main memory. We use a semantic domain *Store* that is abstractly given by the following five semantic functions: $\mathit{upd} : \mathit{LVal} \times \mathit{RVal} \times \mathit{Store} \rightarrow \mathit{Store}$ updates a given store; we write $\mu[l \mapsto v]$ for $\mathit{upd}(l, v, \mu)$. The function $\mathit{lval} : \mathit{Obj} \times \mathit{Identifier} \times \mathit{Store} \rightharpoonup \mathit{LVal}$ retrieves the left-value of an instance variable (such as of `o.a`). Analogously, $\mathit{rval} : \mathit{LVal} \times \mathit{Store} \rightharpoonup \mathit{RVal}$ retrieves the right-value of a left-value; we write $\mu(l)$ for $\mathit{rval}(l, \mu)$. New objects are allocated by $\mathit{new}_C : \mathit{Store} \rightarrow \mathit{Obj} \times \mathit{Store}$. This family of functions is indexed by class types $C \in \mathit{ClassType}$. For a given store μ, $\mathit{new}_C(\mu)$ yields an object o and a store μ' such that μ' extends μ where $\mu'(\mathit{lval}(o, i, \mu'))$ is defined for any identifier i ranging over all instance variables of the class C.

$$(B,\theta),(B',\theta) \Rightarrow (B,\theta) \leq (B',\theta) \vee (B',\theta) \leq (B,\theta) \tag{17.2.1}$$

$$(C,x),(C',x) \Rightarrow (C,x) \leq (C',x) \vee (C',x) \leq (C,x) \tag{17.2.2}$$

$$(\mathit{Assign},\theta,l) \leq (\mathit{Load},\theta,l) \Rightarrow (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l) \leq (\mathit{Load},\theta,l) \tag{17.3.2}$$

$$\begin{aligned}&(\mathit{Store},\theta,l)_m < (\mathit{Store},\theta,l)_n \Rightarrow \\ &\quad (\mathit{Store},\theta,l)_m \leq (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l)_n\end{aligned} \tag{17.3.3}$$

$$(\mathit{Use},\theta,l) \Rightarrow (\mathit{Assign},\theta,l) \leq (\mathit{Use},\theta,l) \vee (\mathit{Load},\theta,l) \leq (\mathit{Use},\theta,l) \tag{17.3.4}$$

$$(\mathit{Store},\theta,l) \Rightarrow (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l) \tag{17.3.5}$$

$$\begin{aligned}&(\mathit{Assign},\theta,l,v)_n \leq (\mathit{Store},\theta,l,v') \Rightarrow \\ &\quad v = v' \vee (\mathit{Assign},\theta,l,v)_n < (\mathit{Assign},\theta,l)_m \leq (\mathit{Store},\theta,l,v')\end{aligned} \tag{17.1}$$

$$(\mathit{Load},\theta,l)_n \Rightarrow (\mathit{Read},\theta,l)_n \leq (\mathit{Load},\theta,l)_n \tag{17.3.6}$$

$$(\mathit{Write},\theta,l)_n \Rightarrow (\mathit{Store},\theta,l)_n \leq (\mathit{Write},\theta,l)_n \tag{17.3.7}$$

$$(\mathit{Store},\theta,l)_m \leq (\mathit{Load},\theta,l)_n \Rightarrow (\mathit{Write},\theta,l)_m \leq (\mathit{Read},\theta,l)_n \tag{17.3.8}$$

$$(\mathit{Lock},\theta,o)_n \leq (\mathit{Lock},\theta',o) \wedge \theta \neq \theta' \Rightarrow (\mathit{Unlock},\theta,o)_n \leq (\mathit{Lock},\theta',o) \tag{17.5.1}$$

$$(\mathit{Unlock},\theta,o)_n \Rightarrow (\mathit{Lock},\theta,o)_n \leq (\mathit{Unlock},\theta,o)_n \tag{17.5.2}$$

$$\begin{aligned}&(\mathit{Assign},\theta,l) \leq (\mathit{Unlock},\theta) \Rightarrow \\ &\quad (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l)_n \leq (\mathit{Write},\theta,l)_n \leq (\mathit{Unlock},\theta)\end{aligned} \tag{17.6.1}$$

$$\begin{aligned}&(\mathit{Lock},\theta) \leq (\mathit{Use},\theta,l) \Rightarrow \\ &\quad (\mathit{Lock},\theta) \leq (\mathit{Assign},\theta,l) \leq (\mathit{Use},\theta,l) \vee \\ &\quad (\mathit{Lock},\theta) \leq (\mathit{Read},\theta,l)_n \leq (\mathit{Load},\theta,l)_n \leq (\mathit{Use},\theta,l)\end{aligned} \tag{17.6.2}$$

$$(\mathit{Lock},\theta) \leq (\mathit{Store},\theta,l) \Rightarrow (\mathit{Lock},\theta) \leq (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l) \tag{17.6.2'}$$

$$(\mathit{Read},\theta,l)_n \Rightarrow (\mathit{Load},\theta,l)_n \tag{17.2.6}$$

$$(\mathit{Store},\theta,l)_n \Rightarrow (\mathit{Write},\theta,l)_n \tag{17.2.7}$$

Table 1. Event space axioms

The local variables of a block are kept in a stack of environments. *Environments*, denoted *Env*, are pairs (I, ρ) of declared identifiers $I \subseteq \mathit{Identifier} \cup \{\mathit{this}\}$ and a map $\rho : I \rightharpoonup \mathit{RVal}$ representing the values they (possibly) have. Environments are also used to store the information on which object's code is currently being executed ($\rho(\mathit{this})$). An environment ρ is updated as usual by $\rho[i \mapsto v]$. The empty environment is denoted by $\rho_\emptyset$. Let *S-Stack* be the domain of (single-threaded) *stacks of environments*. The empty stack is written $\sigma_\emptyset$. The operation *push* : $\mathit{Env} \times \mathit{S\text{-}Stack} \to \mathit{S\text{-}Stack}$ is the usual one on stacks. We use the operations $\sigma[i \mapsto v]$ for updating stacks, and $\sigma(i)$ for retrieving values. Each thread of execution of a Java program has its own stack. We call $\mathit{M\text{-}Stack} = \mathit{Thread\_id} \rightharpoonup \mathit{S\text{-}Stack}$ the domain of *multi-threaded stacks*, ranged over by σ. Given $\sigma \in \mathit{M\text{-}Stack}$, the multi-threaded stacks $\mathit{push}(\theta, \rho, \sigma)$, $\sigma[\theta, i \mapsto v]$ map θ' to $\sigma(\theta')$ when $\theta \neq \theta'$, and otherwise map θ respectively to $\mathit{push}(\rho, \sigma(\theta))$, $\sigma(\theta)[i \mapsto v]$. Note that an additional operation is necessary for extending (stacks of) environments when dealing with local variable declarations, but those are not addressed in this paper (cf. [2]). We also write $\sigma(\theta, i)$ instead of $\sigma(\theta)(i)$.

The operational semantics works on a set *M-Term* of *multi-threaded abstract terms* that contain *single-threaded abstract terms* from a set *S-Term*. We let the metavariable t range over *S-Term*. To each syntactic category of Java we associate a homonymous category of abstract terms. The well-typed terms of Java are mapped to abstract terms of corresponding category by a translation $(\_)^\circ$, which we leave implicit when no confusion arises. Abstract blocks are terms of the form $\{\, t\,\}_{(I,\rho)}$ where the source I of the environment (I,ρ) contains the local variables of the block. A multi-threaded abstract term T is a set of pairs (θ, t), where $\theta \in \mathit{Thread\_id}$ and $t \in \mathit{S\text{-}Term}$ and no distinct elements of T bear the same thread identifier. Multi-threaded abstract terms $\{(\theta_1, t_1), (\theta_2, t_2), \dots\}$ are written as lists $(\theta_1, t_1) \mid (\theta_2, t_2) \mid \dots$ and pairs (θ, t) are written t when θ is irrelevant.

The *configurations* of multi-threaded Java are 4-tuples (T, η, σ, μ) consisting of an M-term T, an event space η, an M-stack σ, and a store μ. The operational semantics is the binary relation $\longrightarrow$ on configurations inductively defined by the rules that follow. In the rule schemes in Tables 2–4, the metavariables range as follows: $i \in \mathit{Identifier}$, $k \in \mathit{Identifier} \cup \mathit{LVal}$, $l \in \mathit{LVal}$, $o \in \mathit{Obj}$, $e \in \mathit{Expression}$, $v \in \mathit{RVal}$, $s \in \mathit{Statement}$, $b \in \mathit{BlockStatement}$, $B \in \mathit{BlockStatement}^*$, $q \in \mathit{Block}$, $t \in \mathit{S\text{-}Term}$, and $T \in \mathit{M\text{-}Term}$. Stacks, event spaces, and stores are *omitted* when they are not relevant.

We write $\mathit{store}_\eta(\theta, l)$ for the oldest unwritten value of l stored by θ in η. More formally: let an event $(\mathit{Store}, \theta, l)_n$ in η be called *unwritten* if $(\mathit{Write}, \theta, l)_n$ is undefined in η; then, $\mathit{store}_\eta(\theta, l) = v$ if there exists an unwritten $(\mathit{Store}, \theta, l, v)_n$ such that for any unwritten $(\mathit{Store}, \theta, l)_m$ we have $n \le m$; if no such a *Store* event exists, $\mathit{store}_\eta(\theta, l)$ is undefined. Similarly, we write $\mathit{rval}_\eta(\theta, l)$ for the latest value of l assigned or loaded and read by θ in η.

[assign1]	$\dfrac{e_1 \longrightarrow e_2}{e_1 = e \longrightarrow e_2 = e}$	[assign2]	$\dfrac{e_1 \longrightarrow e_2}{k = e_1 \longrightarrow k = e_2}$
[assign3']	$(\theta, l = v), \eta \longrightarrow (\theta, v), \eta \oplus (\mathit{Assign}, \theta, l, v)$		
[assign4']	$(\theta, i = v), \sigma \longrightarrow (\theta, v), \sigma[\theta, i \mapsto v]$		
[binop1]	$\dfrac{e_1 \longrightarrow e_2}{e_1 \ \mathtt{op}\ e \longrightarrow e_2 \ \mathtt{op}\ e}$	[binop2]	$\dfrac{e_1 \longrightarrow e_2}{v \ \mathtt{op}\ e_1 \longrightarrow v \ \mathtt{op}\ e_2}$
[binop3]	$v_1 \ \mathtt{op}\ v_2 \longrightarrow v_1 \ \mathit{op}\ v_2$	[pth]	$(e) \longrightarrow e$
[access1]	$\dfrac{e_1 \longrightarrow e_2}{e_1 \,.\, i \longrightarrow e_2 \,.\, i}$	[access2]	$o.i\,, \mu \longrightarrow \mathit{lval}(o, i, \mu), \mu$
[this]	$(\theta, \mathbf{this})\,, \sigma \longrightarrow (\theta, \sigma(\theta, \mathit{this})), \sigma$	[new]	$\mathbf{new}\ C\ (\,)\,, \mu \longrightarrow \mathit{new}_C(\mu)$
[val']	$(\theta, l), \eta \longrightarrow (\theta, \mathit{rval}_\eta(\theta, l)), \eta \oplus (\mathit{Use}, \theta, l)$		
[var']	$(\theta, i), \sigma \longrightarrow (\theta, \sigma(\theta, i)), \sigma$		

Table 2. Expressions

[statseq1]	$\dfrac{b_1 \longrightarrow b_2}{b_1\,B \longrightarrow b_2\,B}$	[statseq2]	$\dfrac{b,\mu_1 \longrightarrow \mu_2}{b\,B,\mu_1 \longrightarrow B,\mu_2}$
[expstat1]	$\dfrac{e_1 \longrightarrow e_2}{e_1\,; \longrightarrow e_2\,;}$	[expstat2]	$\dfrac{e,\mu_1 \longrightarrow v,\mu_2}{e\,;\,,\mu_1 \longrightarrow \mu_2}$
[skip]	$;\,,\sigma \longrightarrow \sigma$	[block1]	$\{\ \}_\rho\,,\sigma \longrightarrow \sigma$
[block2']	$\dfrac{(\theta,B_1),push(\theta,\rho_1,\sigma_1) \longrightarrow (\theta,B_2),push(\theta,\rho_2,\sigma_2)}{(\theta,\{B_1\}_{\rho_1}),\sigma_1 \longrightarrow (\theta,\{B_2\}_{\rho_2}),\sigma_2}$		
[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]	$\texttt{if}(\mathit{false})\ s,\mu \longrightarrow \mu$
[while]	$\texttt{while}(e)\ s \longrightarrow \texttt{if}(e)\ \{\ s\ \texttt{while}(e)\ s\ \}$		
[for]	$\texttt{for}(e_1;\ e_2;\ e_3)\ s \longrightarrow \{\ e_1;\ \texttt{while}(e_2)\ \{\ s\ e_3;\ \}\ \}$		

Table 3. Statements

[synchro1]	$\dfrac{e_1 \longrightarrow e_2}{\texttt{synchronized}(e_1)\ q \longrightarrow \texttt{synchronized}(e_2)\ q}$
[synchro2]	$\dfrac{q_1 \longrightarrow q_2}{\texttt{synchronized}(o)\ q_1 \longrightarrow \texttt{synchronized}(o)\ q_2}$
[lock]	$\dfrac{(\theta,e),\eta_1 \longrightarrow (\theta,o),\eta_2}{(\theta,\texttt{synchronized}(e)\ q),\eta_1 \longrightarrow (\theta,\texttt{synchronized}(o)\ q),\eta_2 \oplus (\mathit{Lock},\theta,o)}$
[unlock]	$(\theta,\texttt{synchronized}(o)\ \{\ \}_\rho),\eta \longrightarrow \eta \oplus (\mathit{Unlock},\theta,o)$
[read]	$T,\eta,\mu \longrightarrow T,\eta \oplus (\mathit{Read},\theta,l,\mu(l)),\mu$
[load]	$T,\eta \longrightarrow T,\eta \oplus (\mathit{Load},\theta,l)$
[store]	$T,\eta \longrightarrow T,\eta \oplus (\mathit{Store},\theta,l,v)$
[write]	$T,\eta,\mu \longrightarrow T,\eta \oplus (\mathit{Write},\theta,l),\mu[l \mapsto \mathit{store}_\eta(\theta,l)]$
[par]	$\dfrac{t_1 \longrightarrow t_2}{t_1 \mid T \longrightarrow t_2 \mid T}$

Table 4. Multi-threaded Java

The rules [assign3', val', lock, unlock, read, load, store, write] make use of the well-formedness conditions of event spaces via the $\oplus$. The rules [read, load, store, write] are spontaneous in the sense that they do not depend on T. The [store] rule additionally "guesses" the value of the last *Assign*; its correctness is ensured by axiom (17.1). *Synchronization*, i.e. mutual exclusion, is handled by [synchro1, synchro2, lock, unlock], by [par] sequential computations are lifted to multi-threaded ones.

4 Prescient Event Spaces

The *prescient* store actions are introduced in [3, 17.8] as follows: " ... the *store* [of variable V by thread T] action [is allowed] to instead occur before the *assign* action, if the following rule restrictions are obeyed:

- If the *store* 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*: ... "

The specification above seems to assume that it is known which *Store* events are prescient and which prescient *Store* event is matched by which *Assign* event. We do not assume such knowledge but adopt a more general approach introducing so-called complete labellings. These labellings are not necessarily unique but it is always possible to infer a complete labelling at run time. It will turn out, however, that the semantics is independent of the choice of complete labellings, see Corollary 6.

In order to define the new *prescient event spaces* we proceed as follows:

First, we have to add new relations (cf. axioms (17.2.1), (17.2.2)) between certain actions of different threads in order to be able to formalize the preconditions of the second, third, and fourth requirement above. *Assign*, *Load* or *Store* actions for the same variable and *Lock* actions must be comparable. To this end let $D = \{Assign, Load, Store\}$, then we stipulate:

$$(D,\theta,l),(D,\theta',l) \Rightarrow (D,\theta,l) \leq (D,\theta',l) \vee (D,\theta',l) \leq (D,\theta,l)$$

Since *Store* and *Lock* events are already comparable, by transitivity also *Lock* and D actions are comparable.

Second, rules (17.3.3), (17.3.5), (17.1), and (17.6.2') are now used for the definition of a predicate *prescient* on event spaces and *Store* events yielding true iff a *Store* is necessarily prescient. We define $prescient_\eta((Store,\theta,l)_n)$ to be valid if one of the rules in Table 5 holds. Note that η is usually omitted if it is clear from the context.

Rules (P1–P4) simply tell that a *Store* event which does not obey old rules (17.3.3), (17.3.5), (17.1), or (17.6.2') is necessarily prescient. Rule (P5) is sound because if there is only one $(Assign,\theta,l,v)$ 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.

Third, keep rules (17.2.1), (17.2.2), (17.3.4), (17.3.6), (17.3.7), (17.3.8), (17.5.1), (17.5.2), and (17.6.2).

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

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

$$\begin{aligned}&(\mathit{Assign},\theta,l,v')_m \leq (\mathit{Store},\theta,l,v)_n \not\Rightarrow\\ &\quad v = v' \lor (\mathit{Assign},\theta,l,v')_m \leq (\mathit{Assign},\theta,l)_k \leq (\mathit{Store},\theta,l,v)_n\end{aligned} \tag{P3}$$

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

$$\begin{aligned}&(\mathit{Store},\theta,l)_m \leq (\mathit{Assign},\theta,l)_k \leq (\mathit{Assign},\theta,l)_{k'} \leq (\mathit{Store},\theta,l)_n \land\\ &\quad \mathit{prescient}((\mathit{Store},\theta,l)_m) \Rightarrow k = k'\end{aligned} \tag{P5}$$

Table 5. Rules for *prescient*

Fourth, adapt rule (17.3.2) as follows, allowing prescient *Stores* on the right hand side of an implication:

$$\begin{aligned}&(\mathit{Assign},\theta,l,v) \leq (\mathit{Load},\theta,l) \Rightarrow\\ &\quad ((\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l) \leq (\mathit{Load},\theta,l)) \lor\\ &\quad ((\mathit{Store},\theta,l,v) \leq (\mathit{Assign},\theta,l,v) \leq (\mathit{Load},\theta,l) \land \mathit{prescient}(\mathit{Store},\theta,l,v))\end{aligned}$$

and rule (17.6.1) as follows:

$$\begin{aligned}&(\mathit{Assign},\theta,l,v) \leq (\mathit{Unlock},\theta) \Rightarrow\\ &\quad (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l)_n \leq (\mathit{Write},\theta,l)_n \leq (\mathit{Unlock},\theta)) \lor\\ &\quad \big((\mathit{Store},\theta,l,v)_n \leq (\mathit{Assign},\theta,l,v) \leq (\mathit{Unlock},\theta) \land\\ &\quad\ (\mathit{Write},\theta,l)_n \leq (\mathit{Unlock},\theta) \land \mathit{prescient}((\mathit{Store},\theta,l,v)_n)\big)\end{aligned}$$

Finally, we need an additional rule corresponding to the second, third, and fourth requirements in the citation at top of Section 4. We add a new rule scheme: for any $a \in \{(\mathit{Lock}), (\mathit{Load},l), (\mathit{Store},l)\}$:

$$\begin{aligned}&(\mathit{Store},\theta,l,v)_n < a \land \mathit{prescient}((\mathit{Store},\theta,l,v)_n) \Rightarrow\\ &\quad (\mathit{Store},\theta,l,v)_n \leq (\mathit{Assign},\theta,l,v) \leq a\end{aligned} \tag{17.8}$$

Next, we redefine the operation $\oplus$ on *prescient event spaces*: A new event a is adjoined to a prescient event space η as in the case for old event spaces, but one additional condition. Let $A \in \{\mathit{Store}, \mathit{Assign}, \mathit{Load}\}$. If $a = (A,\theta,l)$ and $b = (A,\theta',l) \in \eta$ then $b \leq a$. Also, the term $\eta \oplus a$ denotes the space thus obtained, provided it obeys the above rules for prescient event spaces, and it is otherwise undefined.

Analogously to the predicate *prescient* one can also define a predicate *non_prescient* which contains only *Stores* that are necessarily non-prescient. We define $\mathit{non\_prescient}((\mathit{Store},\theta,l)_m)$ on an (implicitly) given event space to be true if one of the rules of Table 6 is fulfilled.

Rule (NP2) is the dual of (P5). Moreover, rule (NP2) is raised by (17.3.3) and (NP3) by new rule (17.8). Observe also that the predicate *prescient* propagates from past to present whereas *non_prescient* is computed in the opposite direction. Note that $\neg\, \mathit{non\_prescient}(B)$ is *not* equivalent to $\mathit{prescient}(B)$

$$\neg\exists(Assign,\theta,l,v)\,.\,(Store,\theta,l)_m \le (Assign,\theta,l,v) \qquad \text{(NP1)}$$

$$(Store,\theta,l)_m \le (Assign,\theta,l)_k \le (Assign,\theta,l)_{k'} \le (Store,\theta,l)_n \wedge \\ non\_prescient((Store,\theta,l)_n) \Rightarrow k = k' \qquad \text{(NP2)}$$

$$\forall a \in \{(Lock),(Load,l),(Store,l)\}\,.\,(Store,\theta,l,v)_m < a \Rightarrow \\ \neg\exists(Assign,\theta,l,v)\,.\,(Store,\theta,l,v)_m \le (Assign,\theta,l,v) \le a \qquad \text{(NP3)}$$

Table 6. Rules for *non_prescient*

and hence also $prescient(B) \vee non\_prescient(B)$ does not always hold and $prescient(B) \wedge non\_prescient(B)$ is not always false.

A prescient event space η is called *consistently complete* if it is complete and for no instance of a *Store*, say s, we have that $prescient_\eta(s) \wedge non\_prescient_\eta(s)$. Note that it makes only sense for the final event space of a reduction sequence to be consistently complete (as for complete). During execution, the matching *Assign* for a prescient *Store* might not have happened and therefore the corresponding *Store* would be considered *non_prescient*, which might lead to a contradiction. A consistently complete event space fulfills the first and last requirement in [3, §17.8] (see top of Section 4), because a prescient *Store* would otherwise have no matching *Assign* and hence by rule (NP1) contradict consistently completeness.

There might be a *Store* event s in a given event space for which neither $prescient(s)$ nor $non\_prescient(s)$ is derivable. In this case one needs a "labelling" of *Store* events, i.e. a predicate fixing whether a *Store* shall be considered prescient or not. More formally, a *labelling* for a prescient event space is a predicate ℓ on *Store* events such that it obeys rules (L1–L3) in Table 7.

$$prescient(s) \Rightarrow \ell(s) \qquad \text{(L1)}$$

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

$$\big((Store,\theta,l)_m \le (Assign,\theta,l)_k \le (Assign,\theta,l)_{k'} \le (Store,\theta,l)_n \Rightarrow k = k'\big) \Rightarrow \\ \big(\ell((Store,\theta,l)_m) \Rightarrow \ell((Store,\theta,l)_n)\big) \qquad \text{(L3)}$$

$$prescient(s) \Rightarrow p^*(s) \qquad \text{(PC1)}$$

$$p(s) \Rightarrow p^*(s) \qquad \text{(PC2)}$$

$$\big((Store,\theta,l)_m \le (Assign,\theta,l)_k \le (Assign,\theta,l)_{k'} \le (Store,\theta,l)_n \Rightarrow k = k'\big) \Rightarrow \\ \big(p^*((Store,\theta,l)_m) \Rightarrow p^*((Store,\theta,l)_n)\big) \qquad \text{(PC3)}$$

Table 7. Rules for labelling and prescient closure

Rule (L3) implies that $\neg\ell$ is closed under (NP2).

Let p be any binary predicate on event spaces and *Store* events (where we usually omit the event space argument). Then we define the *prescient closure* of p, the binary predicate p^*, inductively by rules (PC1–PC3) of Table 7.

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

Proof. Choose a p such that $\neg(p(s) \wedge \mathit{non\_prescient}(s))$ holds for any *Store* event s. This is possible since the event space is consistently complete; for example, $p = \mathit{prescient}$ (or equivalently $p = \mathit{false}$) will do. It remains to prove that p^* is a labelling: rules (L1) and (L3) hold by (PC1) and (PC3), respectively. In order to show rule (L2) prove by induction on the derivation of $p^*(s)$ that $\mathit{non\_prescient}(s) \wedge p^*(s)$ leads to a contradiction. In the (PC1)-case one needs consistently completeness and in the (PC2)-case the assumption on p.

For a consistently complete prescient event space with a labelling the *Assign* events matching the prescient *Store*s can also be singled out as follows: Let ℓ be a labelling on an prescient event space η. A *matching* (labelling of *Assigns*) on ℓ and η, m_ℓ, is a predicate on the *Assign* events of η fulfilling the three axioms in Table 8.

$$\begin{aligned}
&\forall a \in \{(Lock), (Load, l), (Store, l)\} \,.\, (Store, \theta, l, v) < a \wedge \ell((Store, \theta, l, v)) \Rightarrow \\
&\qquad (Store, \theta, l, v) \leq (Assign, \theta, l, v) \leq a \wedge m_\ell((Assign, \theta, l, v)) \\
&(Store, \theta, l, v) \wedge \ell((Store, \theta, l, v)) \Rightarrow \\
&\qquad (Store, \theta, l, v) \leq (Assign, \theta, l, v) \wedge m_\ell((Assign, \theta, l, v)) \\
&(Store, \theta, l, v)_k \leq (Assign, \theta, l, v)_m < (Assign, \theta, l, v)_n \wedge \\
&\qquad\qquad \ell((Store, \theta, l, v)_k) \wedge m_\ell((Assign, \theta, l, v)_m) \wedge m_\ell((Assign, \theta, l, v)_n) \Rightarrow \\
&\qquad (Store, \theta, l, v)_k \leq (Assign, \theta, l, v)_m \leq (Store, \theta, l, v)_{k'} \leq (Assign, \theta, l, v)_n \wedge \\
&\qquad\qquad\qquad\qquad\qquad\qquad \ell((Store, \theta, l, v)_{k'})
\end{aligned}$$

Table 8. Rules for matching

It is easily checked that the following predicate fulfills the axioms for matchings.

$$\begin{aligned}
&\hat{m}_\ell((Assign, \theta, l, v)_m) \Leftrightarrow \\
&\qquad \exists (Store, \theta, l, v) . (Store, \theta, l, v) \leq (Assign, \theta, l, v)_m \wedge \ell((Store, \theta, l, v)) \wedge \\
&\qquad \neg\exists (Assign, \theta, l, v)_n \,.\, (Store, \theta, l, v) \leq (Assign, \theta, l, v)_n < (Assign, \theta, l, v)_m
\end{aligned}$$

A *complete labelling* is a pair consisting of a labelling and a matching for this labelling.

For the sake of simplicity we assume in the rest of the paper that a complete labelling is always given and exhibited in form of special action names, i.e. *pStore* and *pAssign*. If *prescient*$(Store, \theta, l, v)$ holds then $(Store, \theta, l, v)$ is denoted $(pStore, \theta, l, v)$ and analogously for the matching *Assign* we use *pAssign*.

5 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 $\longrightarrow\!\!\blacktriangleright$ of Section 2.

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. $Conf_{\blacktriangleright} \subseteq Conf_{\triangleright}$, and any reduction $\Gamma \to_{\blacktriangleright} \Gamma'$ is also a prescient one, i.e. $\Gamma \to_{\triangleright} \Gamma'$ holds as well.*

Proof. Assume η is an event space satisfying the old rules. By a simple induction, $prescient_{\eta}(s)$ 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 immediately recognized by the rules (P1–P5) can be considered nonprescient. This corresponds to the prescient closure $false^{*}$ (cf. Lemma 1) meaning that the labelling is computed at run time. By definition also $\hat{m}_{false^{*}}$ is computable at run time, thus a complete labelling is, too.

6 Prescient Semantics is conservative

The relation between the "normal" and the "prescient" semantics is described in [3, §17.8] 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 $\to_{\blacktriangleright}$ only, will be used analogously for all kinds of arrows: $\xrightarrow{r}_{\blacktriangleright}$ denotes a one-step reduction with rule r; if $e = (r_1, \dots, r_n)$ is a list of rules then $\xrightarrow{e}_{\blacktriangleright}$ denotes $\xrightarrow{r_1}_{\blacktriangleright} \dots \xrightarrow{r_n}_{\blacktriangleright}$; if the list is irrelevant we write $\to_{\blacktriangleright}^{*}$. 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 $\to_{\triangleright}$ and $\to_{\blacktriangleright}$ can not be bisimilar by definition since $\to_{\triangleright}$ permits *Store*-actions where $\to_{\blacktriangleright}$ does not. But $\to_{\triangleright}$ cannot even be bisimilar to the reflexive closure of $\to_{\blacktriangleright}$, since simulating a $(pStore, \theta, l)$ and the following *Writes* 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 $Conf_{\blacktriangleright} \times Conf_{\triangleright}$, we define an equivalence on prescient configurations $\sim \subseteq Conf_{\triangleright} \times Conf_{\triangleright}$ as follows:

$$(T,\eta,\sigma,\mu) \sim (T',\eta',\sigma',\mu') \iff T = T' \wedge \sigma = \sigma' \wedge (T,\eta,\sigma,\mu) \downarrow (T',\eta',\sigma',\mu')$$

$$(T,\eta,\sigma,\mu) \downarrow (T',\eta',\sigma',\mu') \iff \forall \boldsymbol{a} \,.\, \eta \oplus \boldsymbol{a}\!\downarrow \Leftrightarrow \eta' \oplus \boldsymbol{a}\!\downarrow \wedge$$

$$\forall e. \, (T,\eta,\sigma,\mu) \xrightarrow{e}_{\triangleright}{}^{c} (T_1,\eta_1,\sigma_1,\mu_1) \wedge (T',\eta',\sigma',\mu') \xrightarrow{e}_{\triangleright}{}^{c} (T_2,\eta_2,\sigma_2,\mu_2) \Rightarrow \mu_1 = \mu_2$$

where $\boldsymbol{a}$ is any sequence of actions, e is a sequence of rules and $(T, \sigma, \eta, \mu) \longrightarrow^{c} (T', \sigma', \eta', \mu')$ if $(T, \sigma, \eta, \mu) \longrightarrow^{*} (T', \sigma', \eta', \mu')$ such that η' is complete.

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 $\Gamma_1 \xrightarrow{r} \Gamma_1'$ iff $\Gamma_2 \xrightarrow{r} \Gamma_2'$ 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:

$$(\theta,\ \{\ \texttt{synchronized}(o)\ \{\ l = v;\}_{\rho_\emptyset}\ \}_{\rho_\emptyset})\ \mid\ (\theta',\ \{\ l = v';\}_{\rho_\emptyset})$$

Its execution may give rise to a sequence of computation steps which contains the following complete subsequence of actions:

$$(Lock, \theta, o), (Assign, \theta, l, v), (Store, \theta, l, v), (pStore, \theta', l, v'),$$
$$(Write, \theta', l), (Write, \theta, l), (Unlock, \theta, o), (pAssign, \theta', l, v')$$

In a simulation the $(Store, \theta', l, v')$ 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 $(Write, \theta, l)$ is bound to occur before the *Unlock* and therefore also $(Store, \theta, l, v)$. Finally, after the *Assign* we must recover the pending prescient $(Store, \theta', l, v')$ and its corresponding $(Write, \theta', l)$. According to this simulation l has value v' in the global memory, but the reduction via $\longrightarrow$ yields v for l. Thus, both end-configurations are not equivalent, a contradiction.

Therefore, we have to restrict ourselves to "properly synchronized" programs. A multi-threaded program T is called *properly synchronized* if for any configuration $(T', \eta', \sigma', \mu')$ such that $(T, \eta, \sigma_\emptyset, \emptyset) \longrightarrow^{*} (T', \eta', \sigma', \mu')$ and $(Write, \theta_1, l, v_1) \leq (Write, \theta_2, l, v_2)$ in η' there is a $(Lock, \theta_3, o)$ in η' such that $(Write, \theta_1, l, v_1) \leq (Lock, \theta_3, o) \leq (Write, \theta_2, l, v_2)$. To be "properly synchronized" is a semantical (and rather intricate) property which for a program is hard to tell in advance. A sufficient condition for "properly synchronizedness" is the syntactic criterion that in a program shared variables may only be written in synchronized blocks. It is clear, that in any execution sequence two *Write* actions must then be separated by the corresponding *Lock*.

In the sequel Δ (possibly with annotations) stands for configurations in $Conf_{\blacktriangleright}$ and Γ for new configurations in $Conf_{\triangleright}$. 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 $\rhd\!\!\longrightarrow : (Conf_{\blacktriangleright} \times E^*) \times (Conf_{\blacktriangleright} \times E^*)$ where $E = \{(pStore), (Write), (Read)\}$ by the rules of Table 6. Note that we do not need to treat $(Load)$ events (cf. rule (17.8)). The corresponding $\rhd\!\!\longrightarrow$-configurations (Δ, e) consist of an old configuration $\Delta \in Conf_{\blacktriangleright}$ 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 $split_{\theta,l}(e)$ is needed. Given a list of events e it yields a pair of lists (e_l, e') where both are sublists of e; e_l is obtained from e by extracting all $(pStore, \theta, l)$, $(Write, \theta, l)$ and $(Read, \theta', l)$ events simultaneously changing a $(pStore, \theta, l)$ into $(Store, \theta, l)$, and e' is e_l's complement w.r.t. e.

$$(\Delta, e) \rhd\!\!\xrightarrow{(pStore,\theta,l,v)} (\Delta, e \circ (pStore, \theta, l, v)) \tag{red$_\mathrm{s}$}$$

$$(\Delta, e) \rhd\!\!\xrightarrow{(Write,\theta,l)} (\Delta, e \circ (Write, \theta, l)) \quad \text{if} \quad (pStore, \theta, l, v) \in e \tag{red$_\mathrm{w}$}$$

$$(\Delta, e) \rhd\!\!\xrightarrow{(Read,\theta',l,v)} (\Delta, e \circ (Read, \theta', l, v)) \quad \text{if} \quad (Write, \theta, l) \in e \tag{red$_\mathrm{r}$}$$

$$(\Delta, e) \rhd\!\!\xrightarrow{(pAssign,\theta,l,v)} (\Delta', e') \quad \text{if} \quad \begin{array}{r} split_{\theta,l}(e) = (e_l, e') \wedge \\ \Delta \xrightarrow{(Assign,\theta,l,v)} \Delta_1 \xrightarrow{e_l} \Delta' \end{array} \tag{red$_\mathrm{a}$}$$

$$(\Delta, e) \rhd\!\!\xrightarrow{r} (\Delta', e) \text{ for any other case } r \text{ if } \Delta \xrightarrow{r} \Delta' \tag{red$_\mathrm{d}$}$$

Table 9. Rules for the simulating reduction relation

To relate configurations of $-\!\!\rhd$ and $\rhd\!\!\rightarrow$ reductions the simulation relation $\approx \subseteq Conf_{\rhd} \times (Conf_{\blacktriangleright} \times E^*)$ is defined as follows:

$$\Gamma \approx (\Delta, e) \quad \text{if, and only if,} \quad \Delta \overset{e}{-\!\!\rhd} \Gamma_\Delta \wedge \Gamma_\Delta \sim \Gamma$$

i.e. Γ is equivalent to (Δ, e) if Γ is equivalent to the completion of Δ, usually called Γ_Δ, by executing the pending events in e. Note that $-\!\!\rhd$ 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 4. *If $\Gamma \approx (\Delta, e)$ and $\Gamma \overset{r}{-\!\!\rhd} \Gamma'$, 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*

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

commutes, hence in particular $\Gamma \approx (\Delta, e) \rhd\!\!\xrightarrow{r} (\Delta', e) \approx \Gamma'$ holds.

Proof. (sketched) By definition of $\rhd\!\!\rightarrow$ we have $\Delta \xrightarrow{r} \Delta'$ as we consider case (red$_\mathrm{d}$). Next, we have to check that r does not depend on e, such that commutation is possible. Proof is by inspecting the relevant laws for event spaces: rules

(17.3.2), (17.3.4), (17.6.2) refer to *Load* events which are not possible as long as e contains a corresponding *pStore*, (17.3.7) is not relevant as matching *Writes* are treated in (red_w). Thus, we are left with (17.6.1). Cases, however, where *Store* and *Write* in e allow r to be an *Unlock* are excluded by the rules for labellings.

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. But this could only happen if $r = (\mathit{Write}, \theta, l)$ and another $(\mathit{Write}, \theta', l) \in e$ which is impossible since only properly synchronized programs are considered.

Theorem 5. *For properly synchronized programs the relation $\approx$ is a simulation relation of $\longrightarrow\!\!\triangleright$ and $\rhd\!\!\twoheadrightarrow$, i.e. if $\Gamma \xrightarrow{r}\!\!\triangleright\ \Gamma'$ during the execution of such a program and $\Gamma \approx (\Delta, e)$ then there is a (Δ', e') such that $(\Delta, e) \overset{r}{\rhd\!\!\twoheadrightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e')$.*

Proof. Assume $\Gamma \approx (\Delta, e)$, i.e. $\Delta \xrightarrow{e}\!\!\triangleright\ \Gamma_\Delta \sim \Gamma$. We do a case analysis for $\Gamma \xrightarrow{r}\!\!\triangleright\ \Gamma'$:
Case $\Gamma \xrightarrow{\mathit{Write}}\!\!\triangleright\ \Gamma'$: if $(\mathit{pStore}, \theta, l) \in e$ then it holds that $(\Delta, e) \overset{r}{\rhd\!\!\twoheadrightarrow} (\Delta, e \circ r)$ by (red_w). Moreover, by Lemma 3, $\Gamma' \approx (\Delta, e \circ r)$.

If $(\mathit{pStore}, \theta, l) \notin e$ then by Lemma 4, $(\Delta, e) \overset{r}{\rhd\!\!\twoheadrightarrow} (\Delta', e')$ and $\Gamma' \approx (\Delta', e)$.
Case $\Gamma \xrightarrow{\mathit{pAssign}}\!\!\triangleright\ \Gamma'$. Let $\mathit{split}_{\theta,l}(e) = (e_l, e')$. Since an *Assign* is always possible, assume that $\Delta \xrightarrow{(\mathit{Assign}, \theta, l, v)} \Delta_1$. 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 $(\Delta, e) \overset{r}{\rhd\!\!\twoheadrightarrow} (\Delta', e')$. One can prove analogously to Lemma 4 that the left rectangle in

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

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

For *pStore* and *Read* one proceeds as for *Write*, all other cases follow from Lemma 4.

Our main result 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 6. *Given $\Gamma \in \mathit{Conf}_\rhd$ from a properly synchronized program and $\Delta \in \mathit{Conf}_\blacktriangleright$, if $\Gamma \sim \Delta$ and $\Gamma \longrightarrow\!\!\triangleright^* \Gamma'$ such that the event space $\eta_{\Gamma'}$ of Γ' is consistently* *complete, then for any complete labelling of $\eta_{\Gamma'}$ there is a reduction sequence $\Delta \longrightarrow^* \Delta'$ such that $\Gamma' \sim \Delta'$.*

Moreover, if two different complete 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 5, we get $(\Delta, \varepsilon) \rhd\!\!\longrightarrow^* (\Delta', e)$ and $\Gamma' \approx (\Delta', e)$. Now $e = \varepsilon$ follows from the fact that Γ' is consistently 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 $\Delta'_1 \sim \Gamma' \sim \Delta'_2$.

For our running example we can conclude that the corollary is applicable if all threads write `o` exclusively in `synchronized` blocks.

7 Conclusion

We have presented an event space semantics for multi-threaded Java with prescient stores. The informal statements in [3, §17.8] have been formalized and proven completely. In fact, the main motivation for this work was to understand what they meant. Correspondingly, we presented an operational semantics for prescient stores by just refining the axioms of the event space, leaving untouched the laws of the operational semantics. This demonstrates the flexibility of the event space approach.

Future work will include the extension of the treated language, e.g. `wait` and `notify`, exceptions, method calls, and the application of the semantics to correctness proofs of Java programs.

Acknowledgement: We used Paul Taylor's `diagram.sty`.

References

1. Ken Arnold and James Gosling. *The Java Programming Language*. Addison–Wesley, Reading, Mass., 1996.
2. Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In *Proc. 6th Int. Conf. Algebraic Methodology and Software Technology*, Lect. Notes Comp. Sci., Berlin, 1997. Springer. To appear.
3. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison–Wesley, Reading, Mass., 1996.
4. Doug Lea. *Concurrent Programming in Java*. Addison–Wesley, Reading, Mass., 1997.
5. Gordon D. Plotkin. Structural Operational Semantics (Lecture notes). Technical Report DAIMI FN–19, Aarhus University, 1981 (repr. 1991).
6. Glynn Winskel. An Introduction to Event Structures. In Jacobus W. de Bakker, editor, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes Comp. Sci.*, Berlin, 1988. Springer.