From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics

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 non trivial sublanguage of Java is presented. This language includes dynamic creation of objects, blocks, and synchronization of threads. First we introduce a simple operational description of the sequential part of the language, where the memory is treated as an algebra with suitably axiomatized operations. Then, the interaction between threads via a shared memory is described in terms of structures, called "event spaces," whose well-formedness conditions formalize directly the rules given in the Java language specification. Event spaces are included in the operational judgements to develop the semantics of the full multi-threaded sublanguage, which is shown to extend the one for sequential Java conservatively. The result allows sequential programs to be reasoned about in a simplified computational framework without loss of generality.

1 Introduction

Java is an object-oriented programming language which offers a simple and tightly integrated support for concurrent programming. A concurrent program consists of multiple tasks that are or behave as if they were executed all at the same time. In Java tasks are implemented using *threads* (short for "threads of execution"), which are sequences of instructions that run independently within the encompassing program. Informal descriptions of this model can be found in several books (see e.g. [1], [4]). A precise description is given in the Java language specification [3].

This paper presents a formal semantics of a non-trivial sublanguage of Java which includes dynamic creation of objects, blocks, and synchronization of threads. The semantics is given in the style of Plotkin's structural operational semantics (SOS) [7]. This technique has been used e.g. for the semantics of SML [6] and earlier for ADA [5].

The thread model, and in particular the interaction between threads via shared memory, is described here in terms of structures called *event spaces.* These correspond roughly to *configurations* in Winskel's *event structures* [8], which are used for denotational semantics of concurrent languages. By using similar structures in operational semantics, a technique which is new, to our knowledge, we obtain an abstract "declarative" description of the Java thread model which is an exact formal counterpart of the informal language description [3] and which leaves maximal freedom for different implementations.

We present the semantics in two steps: First we introduce a simple operational description of the sequential part of the language, where the memory is treated as an algebra with suitably axiomatized operations. Then the thread model is developed and shown to be a conservative extension of sequential Java. For reasons of space we consider in this paper only the following subset of the Java language: access to local variables and instance variables, assignment, class instance creation, blocks, local variable declaration, threads and synchronization. We cut out, among other things, class declaration, method call and exceptions. However, what we include (whose BNF is given in Appendix A) is enough to describe the thread model in full generality.

Closely related work is the formal semantics of a sublanguage of Java in [2]. This paper focuses on the Java type system and develops an operational semantics for a *sequential* sublanguage of Java only. Therefore our semantics of threads is complementary.

The paper is organized as follows: In Section 2 the semantics of single-threaded (sequential) Java programs is given. Section 3 introduces the notion of event space and sets the rules for a correct interaction between main memory and threads. Section 4 describes the refinement of single-threaded Java to multi-threaded Java. The paper concludes with some remarks and future developments.

2 Sequential Java

The operational semantics of sequential Java is quite conventional. We give an overview by means of an example.

```
class Point {
  int x, y;

  Point() { }
}

class Sample {
  public static void main(String[] argv) {
    Point p = new Point();

    p.x = 1; p.y = 2;
    p.x = p.y;
  }
}
```

The sample program consists of two *class declarations* of `Point` and `Sample`. The *class* `Point` has two *attributes* `x` and `y`, the coordinates of a point, and provides only the standard *constructor* `Point()` that is called upon creation of a new *instance* (*object*) of class `Point`. The second class `Sample` has a single *method* `main(String[] argv)` (and a hidden standard constructor).

Programs start and end with the execution of the `main()` method, which must be defined in some (and only one) class. In our example this method creates a new instance of class `Point` by executing the expression `new Point()`; a *reference* to this object is assigned to the *local variable* `p`. Our program proceeds by assigning 1 and 2 to the coordinates of the object referenced by `p`, and then by setting the value of the `x` coordinate to the value of `y`.

Since the scope of local variables is determined by the block structure of the program, we keep them in a *stack* which grows and shrinks upon entering and exiting blocks. On the other hand, objects are permanent entities which survive the blocks in which they are created; therefore the collection of their *instance variables* (containing the values of their attributes) is kept in a separate structure: the *store*. Intuitively, stores can be thought of as mapping left-values (addresses of instance variables) to right-values (the primitive data of Java). Later on we shall see how different threads interact through the store. A formal description of stacks, stores and the configurations of the operational semantics is given below.

Stores. We use a semantic domain *Store* for abstract stores, a domain *Obj* for abstract objects and two domains *LVal* and *RVal* for left and right-values. Since references to objects can be assigned to variables in Java, we stipulate that $Obj \subseteq RVal$. Stores are axiomatized below by means of the following semantic functions:

$$\begin{aligned} new_C &: Store \rightarrow Obj \times Store \\ upd &: LVal \times RVal \times Store \rightarrow Store \\ lval &: Obj \times Identifier \times Store \rightharpoonup LVal \\ rval &: LVal \times Store \rightharpoonup RVal \\ this &: Store \rightarrow Obj \end{aligned}$$

where functions in the family *new* are indexed by class types $C \in ClassType$. As we do not deal with class declarations here, we assume that a function new_C "knows" how to initialize the instance variables of a newly created object of class C. In particular, we assume that initial values are returned by a family of partial functions

$$init_C : Identifier \rightharpoonup RVal$$

whose domain is the set of attributes of C.

In our example the evaluation of `new Point()` produces the object $o_{\mathtt{p}}$ of $new_{\mathtt{Point}}(\mu) = (o_{\mathtt{p}}, \mu')$ where μ is the current store. A new store μ' is also produced with two new left-values $l_{\mathtt{p.x}}$ and $l_{\mathtt{p.y}}$ suitably initialized.

The function *upd* updates a store. The function *lval* finds in the store the location pointed by expressions like `p.x`, where the evaluation of `p` yields an object in in *Obj*. In particular, as shown by the axioms below, $lval(o, i, \mu)$ is defined for those $i \in Identifier$ that are attributes of the class of o. The function *rval* gets the right-value associated in a store with a given left-value, and *this* gets the object whose code is being currently executed.

In the following, object are ranged over by the metavariable o, left values by l, right values by v and stores by μ. All these can be variously decorated. We write $\mu[l \mapsto v]$ and $\mu(l)$ for $upd(l, v, \mu)$ and $rval(l, \mu)$ respectively.

Abstract stores are axiomatized by using a unary predicate $\downarrow$ (written in postfix notation) and a binary predicate $\preceq$ (in infix notation). The meaning of $e\downarrow$ is that e is defined, i.e. it denotes a value, while $e_1 \preceq e_2$ means 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, and by $e_1 = e_2$ we mean that both $e_1 \preceq e_2$ and $e_1\downarrow$ hold. We write $\uparrow$ the negation of $\downarrow$. The axioms for abstract stores are listed in Table 1.

$$\begin{array}{rl}
\mu(l) \preceq \mu'(l) & (new_C(\mu) = (o, \mu')) \\
lval(o, i, \mu)\uparrow & (new_C(\mu) = (o, \mu')) \\
init_C(i) \preceq \mu'(lval(o, i, \mu')) & (new_C(\mu) = (o, \mu')) \\
\mu[l \mapsto v](l) = v & \\
\mu[l' \mapsto v](l) \simeq \mu(l) & (l \neq l') \\
\mu[l \mapsto v'][l \mapsto v] = \mu[l \mapsto v] & \\
\mu[l' \mapsto v'][l \mapsto v] = \mu[l \mapsto v][l' \mapsto v'] & (l \neq l') \\
\mu[l \mapsto \mu(l)] \preceq \mu &
\end{array}$$

Table 1. Axiomatization of abstract stores

Environment stacks. Environments are pairs (I, ρ) where $I \subseteq$ *Identifier* is a *source* of identifiers and ρ is a partial function from I to right values:

$$Env = \textstyle\sum_{I \subseteq Identifier} I \rightharpoonup RVal\,.$$

Intuitively, I contains the local variables of a block. By abuse of notation, we write ρ for an environment (I, ρ) and indicate with $src(\rho)$ its source I. As usual, $\rho[i \mapsto v](j) = v$ if $i = j$ and $\rho[i \mapsto v](j) \simeq \rho(j)$ otherwise.

Let *S-Stack* be the domain of stacks of environments, ranged over by the metavariable σ. The empty environment and the empty stack are written respectively $\rho_\emptyset$ and $\sigma_\emptyset$. The operation *push* : *Env* × *S-Stack* → *S-Stack* is the usual one on stacks. All other stack operations we use are recursively defined in Table 2.

In our example, the bindings of the block in `main()` yield the stack $push(\rho_\emptyset[\mathrm{p} \mapsto o_{\mathrm{p}}], \sigma_\emptyset)$.

Terms. The operational semantics of single-threaded Java works on a set *S-Term* of *single-threaded abstract terms*. 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

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

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

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

Table 2. Operations on stacks

confusion arises. Abstract blocks are terms of the form $\{\, t \,\}_\rho$ where the environment ρ contains the local variables of the block. In our example, we have

$$\texttt{\{ Point p = new Point(); p.x = 1; p.y = 2; p.x = p.y \}}^\circ =$$
$$\{\ \texttt{Point p} = \texttt{new Point( ); p.x} = 1;\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto\texttt{null}]}\ .$$

Configurations and Rules. We call *configurations* elements of the domain *S-Term* × *S-Stack* × *Store*. We let γ range over this domain. The operational semantics is the binary relation $\rightarrow$ on configurations inductively defined by the rules that follow. Related pairs of configurations are written $\gamma_1 \rightarrow \gamma_2$ and are called *operational judgements.*

In the rule schemes (Tables 3–5), the metavariables (variously decorated) range as follows: $i \in$ *Identifier*, $k \in$ *Identifier* ∪ *LVal*, $e \in$ *Expression*, $\tau \in$ *Type*, $d \in$ *VariableDeclarator*, $D \in$ *VariableDeclarator*$^+$, $s \in$ *BlockStatement*, $S \in$ *BlockStatement** and $q \in$ *Block* (see Appendix A).

We understand statements, among which local variable declarations, as computations over the one-element domain $\{*\}$. Then, consistently with the type of the relation $\rightarrow$, we write $s, \sigma_1, \mu_1 \rightarrow \sigma_2, \mu_2$ for $s, \sigma_1, \mu_1 \rightarrow *, \sigma_2, \mu_2$.

Stacks and stores are omitted when not relevant; that is, we may write:

$$\frac{t_1 \rightarrow t_2}{t_3 \rightarrow t_4} \quad \text{for} \quad \frac{t_1, \sigma_1, \mu_1 \rightarrow t_2, \sigma_2, \mu_2}{t_3, \sigma_1, \mu_1 \rightarrow t_4, \sigma_2, \mu_2}\ .$$

The full set of rules can be found in Table 3 for expressions, Table 4 for local variable declarations, and Table 5 for statements.

For our example, a detailed run of (part of) the block in `main()` can be found in Figure 1. The annotations to the arrows indicate the rules applied. The object $o_{\texttt{p}}$, the locations $l_{\texttt{p.x}}$ and $l_{\texttt{p.y}}$, and the stores μ and μ' are defined as before.

3 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

[assign1]	$\frac{e_1 \rightarrow e_2}{e_1 = e \rightarrow e_2 = e}$	[assign2]	$\frac{e_1 \rightarrow e_2}{k = e_1 \rightarrow k = e_2}$
[assign3]	$l = v, \mu \rightarrow v, \mu[l \mapsto v]$	[assign4]	$i = v, \sigma \rightarrow v, \sigma[i \mapsto v]$
[unop1]	$\frac{e_1 \rightarrow e_2}{\mathtt{op}(e_1) \rightarrow \mathtt{op}(e_2)}$	[unop2]	$\mathtt{op}(v), \mu \rightarrow op(v), \mu$
[access1]	$\frac{e_1 \rightarrow e_2}{e_1 \,.\, i \rightarrow e_2 \,.\, i}$	[access2]	$o.i\,, \mu \rightarrow lval(o, i, \mu), \mu$
[this]	$\mathtt{this}\,, \mu \rightarrow this(\mu), \mu$	[pth]	$(e), \mu \rightarrow e, \mu$
[new]	$\mathtt{new}\ C\ ()\,, \mu \rightarrow new_C(\mu)$	[val]	$l\,, \mu \rightarrow \mu(l), \mu$
[var]	$i\,, \sigma \rightarrow \sigma(i), \sigma$		

Table 3. Expressions

[decl1]	$\frac{e_1 \rightarrow e_2}{i = e_1 \rightarrow i = e_2}$	[decl2]	$i = v, \sigma \rightarrow \sigma[i = v]$
[declseq1]	$\frac{d_1 \rightarrow d_2}{d_1\, D \rightarrow d_2\, D}$	[declseq2]	$\frac{d, \sigma_1 \rightarrow \sigma_2}{d\, D, \sigma_1 \rightarrow D, \sigma_2}$
[locvardecl1]	$\frac{D_1 \rightarrow D_2}{\tau\ D_1 \rightarrow \tau\ D_2}$	[locvardecl2]	$\frac{D, \sigma_1 \rightarrow \sigma_2}{\tau\ D\,, \sigma_1 \rightarrow \sigma_2}$

Table 4. Local variable declarations

[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 formalized in this section as "well-formedness" conditions for structures called *event spaces.* In the next section event spaces are included in the configurations of multi-threaded Java to constrain the applicability of certain operational rules.

[statseq1]	$\frac{s_1 \rightarrow s_2}{s_1\, S \rightarrow s_2\, S}$	[statseq2]	$\frac{s, \mu_1 \rightarrow \mu_2}{s\, S, \mu_1 \rightarrow S, \mu_2}$
[expstat1]	$\frac{e_1 \rightarrow e_2}{e_1\,; \rightarrow e_2\,;}$	[expstat2]	$\frac{e, \mu_1 \rightarrow v, \mu_2}{e\,;\ , \mu_1 \rightarrow \mu_2}$
[skip]	$;\ , \sigma \rightarrow \sigma$	[block1]	$\{\ \}_\rho\,, \sigma \rightarrow \sigma$
[block2]	$\frac{S_1, push(\rho_1, \sigma_1) \rightarrow S_2, push(\rho_2, \sigma_2)}{\{S_1\}_{\rho_1}, \sigma_1 \rightarrow \{S_2\}_{\rho_2}, \sigma_2}$		

Table 5. Statements

$$\{\ \texttt{Point p} = \texttt{new Point}();\ \texttt{p.x} = 1;\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto\texttt{null}]}, \sigma_\emptyset, \mu$$

[block2, statseq1, expstat1, $\downarrow$ locvardecl1, decl1, new]

$$\{\ \texttt{Point p} = o_{\texttt{p}};\ \texttt{p.x} = 1;\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto\texttt{null}]}, \sigma_\emptyset, \mu'$$

[block2, statseq2, expstat2, $\downarrow$ locvardecl2, decl2]

$$\{\ \texttt{p.x} = 1;\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto o_{\texttt{p}}]}, \sigma_\emptyset, \mu'$$

[block2, statseq1, expstat1, $\downarrow$ access1, var]

$$\{\ o_{\texttt{p}}.\texttt{x} = 1;\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto o_{\texttt{p}}]}, \sigma_\emptyset, \mu'$$

[block2, statseq1, expstat1, $\downarrow$ assign1, access2]

$$\{\ l_{\texttt{p.x}} = 1;\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto o_{\texttt{p}}]}, \sigma_\emptyset, \mu'$$

[block2, statseq2, expstat2, $\downarrow$ assign3]

$$\{\ \texttt{p.y} = 2;\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset[\texttt{p}\mapsto o_{\texttt{p}}]}, \sigma_\emptyset, \mu'[l_{\texttt{p.x}} \mapsto 1]$$

$$\ldots \downarrow \ldots$$

Figure 1. Sample run of `Sample.main()`

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 on 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.

For instance, a rule about the interaction of locks and variables [3, 17.6, p. 407] states for a thread θ, a variable V and a lock L:

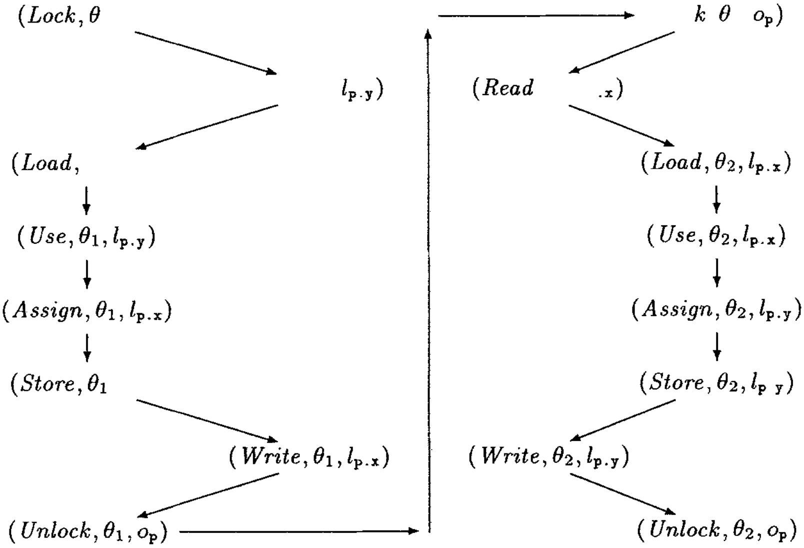
> "Between an *assign* action by [θ] on V and subsequent *unlock* action by [θ] on L, a *store* action by [θ] on V must intervene; moreover, the *write* action corresponding to that *store* must precede the *unlock* action, as seen by the main memory. (Less formally: if a thread is to perform an *unlock* action on *any* lock, it must first copy *all* assigned values in its working memory back out to main memory.)"

We briefly recapitulate those rules by means of the example "Possible Swap" of [3, 17.10] where two threads θ_1 and θ_2 running in parallel want to manipulate the coordinates of the same point object $o_{\texttt{p}}$, referenced in both threads by the local variable p. The thread θ_1 wants to do `p.x = p.y`, while θ_2 wants to do `p.y = p.x`. These manipulations are to run under mutual exclusion; in Java this is obtained by a *synchronization* on a shared object:

$$(\theta_1,\ \texttt{synchronized(p)\{p.x=p.y;\}})\,|\,(\theta_2,\ \texttt{synchronized(p)\{p.y=p.x;\}})$$

In order to enter the critical (`synchronized`) region both threads must perform a *Lock* action on $o_{\texttt{p}}$; by the rules of the Java language specification a *Lock* action of one thread on an object prevents any other thread to perform a *Lock* action on the same object. We assume that θ_1 is first. Now, θ_1 may proceed to obtain the y coordinate of the object referenced by p. By the rules for locks the value of `p.y`, which resides in the main memory, must be loaded first in θ_1's working memory. The language specification requires that such a *Load* action be preceded by a corresponding *Read* action of the main memory. Once this protocol is completed the requested value can be used by θ_1 with a *Use* action and assigned to the working copy of `p.x` with an *Assign* action.

Putting the new value of `p.x` back in the main memory reverses the chain of responsibilities: the working memory of θ_1 issues this value to the main memory by a *Store* action; then the main memory writes the value to the master copy of `p.x` by a *Write* action. This chain is again enforced by the *Unlock* action that ends the critical region. Now, θ_2 may proceed to achieve the lock on o_p, and so forth; the complete series of actions is depicted in Figure 2. Note that the omission of the synchronization would considerably liberalize the order of execution of the actions. Especially, a non-synchronized thread is not forced to write the contents of its working memory back to the main memory.



$(Lock, \theta$
$k\ \theta\ \ o_p)$
$l_{p.y})$
$(Read$
$.x)$
$(Load,$
$(Load, \theta_2, l_{p.x})$
$(Use, \theta_1, l_{p.y})$
$(Use, \theta_2, l_{p.x})$
$(Assign, \theta_1, l_{p.x})$
$(Assign, \theta_2, l_{p.y})$
$(Store, \theta_1$
$(Store, \theta_2, l_{p\ y})$
$(Write, \theta_1, l_{p.x})$
$(Write, \theta_2, l_{p.y})$
$(Unlock, \theta_1, o_p)$
$(Unlock, \theta_2, o_p)$

Figure 2. Event space of example "Possible Swap"

We proceed to formalize this behaviour. Let the metavariable A stand for a generic action name. Moreover, let B range over the set of thread actions and C over the set of memory actions, that is:

$$B \in \{\mathit{Use}, \mathit{Assign}, \mathit{Load}, \mathit{Store}, \mathit{Lock}, \mathit{Unlock}\}\ ,$$
$$C \in \{\mathit{Read}, \mathit{Write}, \mathit{Lock}, \mathit{Unlock}\}\ .$$

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 $o \in Obj$. A triple (A, θ, x) is read "θ performs an A action on x," for x a location or an object, while (A, θ, l, v) is read "(A, θ, l) with value v."

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 lower case letters stand for either. Sometimes we omit components of an action or event: we may write $(Read, l)$ 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 (1–15) below. These conditions, which formalize directly the rules of [3, Chapter 17], are expressed by clauses of the form:

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

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 $\exists \boldsymbol{b}_i \in \eta \,.\, \Psi_i$). 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 $\forall \boldsymbol{a} \in \eta \,.\, (true \Rightarrow \ldots)$ is written $\boldsymbol{a} \Rightarrow (\ldots)$. We include some short, informal explanation of the rules and refer to [3] for more detail.

The actions performed by any one thread are totally ordered, and so are the actions performed by the main memory for any one variable [3, 17.2, 17.5].

$$(B, \theta), (B', \theta) \Rightarrow (B, \theta) \leq (B', \theta) \text{ or } (B', \theta) \leq (B, \theta) \tag{1}$$

$$(C, x), (C', x) \Rightarrow (C, x) \leq (C', x) \text{ or } (C', x) \leq (C, x) \tag{2}$$

Hence, the occurrences of any action (A, θ, x) are totally ordered in an event space. 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. When two indices m and n are applied in a rule to instances of the same action, it is meant that $m \neq n$.

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 [3, 17.3]:

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

A thread is not permitted to write data from its working memory back to main memory for no reason [3, 17.3]:

$$\begin{aligned}(Store, \theta, l)_m \leq (Store, \theta, l)_n \Rightarrow \\ (Store, \theta, l)_m \leq (Assign, \theta, l) \leq (Store, \theta, l)_n\end{aligned} \tag{4}$$

Threads start with an empty working memory and new variables are created only in main memory and not initially in any thread's working memory [3, 17.3]:

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

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

A *Store* action transmits the contents of the thread's working copy of a variable to main memory [3, 17.1]:

$$\begin{aligned}&(Assign,\theta,l,v)_n \leq (Store,\theta,l,v') \Rightarrow \\ &\quad v = v' \text{ or } (Assign,\theta,l,v)_n \leq (Assign,\theta,l)_m \leq (Store,\theta,l,v')\end{aligned} \tag{7}$$

Each *Load* or *Write* action is uniquely paired respectively with a matching *Read* or *Store* action that precedes it [3, 17.2, 17.3]:

$$(Load,\theta,l)_n \Rightarrow (Read,\theta,l)_n \leq (Load,\theta,l)_n \tag{8}$$

$$(Write,\theta,l)_n \Rightarrow (Store,\theta,l)_n \leq (Write,\theta,l)_n \tag{9}$$

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 [3, 17.3]:

$$(Store,\theta,l)_m \leq (Load,\theta,l)_n \Rightarrow (Write,\theta,l)_m \leq (Read,\theta,l)_n \tag{10}$$

A thread is not permitted to unlock a lock it does not own. 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 [3, 17.5]:

$$(Unlock,\theta,o)_n \Rightarrow (Lock,\theta,o)_n \leq (Unlock,\theta,o)_n \tag{11}$$

$$(Lock,\theta,o)_n \leq (Lock,\theta',o) \text{ and } \theta \neq \theta' \Rightarrow (Unlock,\theta,o)_n \leq (Lock,\theta',o) \tag{12}$$

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 [3, 17.6] (this rule formalizes the quotation above):

$$\begin{aligned}&(Assign,\theta,l) \leq (Unlock,\theta) \Rightarrow \\ &\quad (Assign,\theta,l) \leq (Store,\theta,l)_n \leq (Write,\theta,l)_n \leq (Unlock,\theta)\end{aligned} \tag{13}$$

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 [3, 17.6]:

$$\begin{aligned}&(Lock,\theta) \leq (Use,\theta,l) \Rightarrow \\ &\quad (Lock,\theta) \leq (Assign,\theta,l) \leq (Use,\theta,l) \text{ or} \\ &\quad (Lock,\theta) \leq (Read,\theta,l)_n \leq (Load,\theta,l)_n \leq (Use,\theta,l)\end{aligned} \tag{14}$$

$$(Lock,\theta) \leq (Store,\theta,l) \Rightarrow (Lock,\theta) \leq (Assign,\theta,l) \leq (Store,\theta,l) \tag{15}$$

Discussion. Each of the above rules corresponds to one rule in [3]. Conversely, any rule in [3] which we have not included above can be derived in our axiomatization. In particular,

$$(\mathit{Load},\theta,l) \leq (\mathit{Store},\theta,l) \Rightarrow (\mathit{Load},\theta,l) \leq (\mathit{Assign},\theta,l) \leq (\mathit{Store},\theta,l) \qquad (*)$$

of [3, 17.3] holds in any event space. 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 $(\mathit{Store},\theta,l) \leq (\mathit{Load},\theta,l) \leq (\mathit{Store},\theta,l)$ with no *Assign* in between, which contradicts (4). Similarly, the following rule of [3, 17.3] derives from (8) and (9):

$$\begin{aligned}&\forall(\mathit{Load},\theta,l)_n,\ (\mathit{Store},\theta,l)_m,\ (\mathit{Write},\theta,l)_m \in \eta\,.\\ &\qquad(\mathit{Load},\theta,l)_n \leq (\mathit{Store},\theta,l)_m \Rightarrow (\mathit{Read},\theta,l)_n \leq (\mathit{Write},\theta,l)_m\end{aligned}$$

The clauses (6) and (15) simplify the corresponding rules of [3, 17.3, 17.6] which include a condition $(\mathit{Load},\theta,l) \leq (\mathit{Store},\theta,l)$ to the right of the implication. This would be redundant because of $(*)$.

Note that the language specification requires any *Read* action to be completed by a corresponding *Load* and similarly for *Store* and *Write*. We do not translate such rules into well-formedness conditions for event spaces because the latter must capture incomplete program executions.

Usage in operational semantics. Event spaces serve two purposes: On the one hand they provide all the information to reconstruct the current working memories of all threads (which in fact do not appear in the configurations). On the other hand event spaces record the "historical" information on the computation which constrain the execution of certain actions according to the language specification, and hence the applicability of certain operational rules.

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 (Read,θ,l) in η, and if A is *Write* then $c \leq a$ for all instances c of $(\mathit{Store},\theta,l)$ in η. The term $\eta \oplus a$ denotes the space thus obtained, provided it obeys the above rules, and it is otherwise undefined. For example, by (5), the term $\eta \oplus (\mathit{Use},\theta,l)$ is defined only if a suitable $(\mathit{Assign},\theta,l)$ or (Load,θ,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 $\eta \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$.

4 Multi-Threaded Java

Stores assume in multi-threaded Java a more active role than they have in sequential Java because of the way the main memory interacts with the working memories: a "silent" computational step changing the store may occur without the direct intervention of a thread's execution engine. Changes to the store are subject to the previous occurrence of certain events which affect the state of computation. Event spaces are included in the configurations to record such historical information.

We first state the necessary extensions for the notions of terms, stacks, and configurations from the single-threaded to the multi-threaded case. Then we give the operational rules for multi-threaded Java and illustrate their use with the "Possible Swap" example. Finally we show that the multi-threaded semantics conservatively extends the semantics of Section 2.

Multi-threaded terms, stacks, and configurations. A multi-threaded Java configuration may include multiple S-terms, one for each running thread. An abstract term T of multi-threaded Java is a set of pairs (θ, t), where $\theta \in$ *Thread_id*, $t \in$ *S-Term* and no distinct elements of T bear the same thread identifier. The set of abstract terms of multi-threaded Java is called *M-Term*. M-terms $\{(\theta_1, t_1), (\theta_2, t_2), \ldots\}$ are written as lists $(\theta_1, t_1) \mid (\theta_2, t_2) \mid \ldots$ and pairs (θ, t) are written t when θ is irrelevant.

Each thread of execution of a Java program has its own stack. We call *M-Stack* the domain of multi-threaded stacks, ranged over by σ. More precisely, *M-Stack* = *Thread_id* $\rightharpoonup$ *S-Stack*. Given $\sigma \in$ *M-Stack*, the multi-threaded stacks $push(\theta, \rho, \sigma)$, $\sigma[\theta, i \mapsto v]$ and $\sigma[\theta, i = v]$ map θ' to $\sigma(\theta')$ when $\theta \neq \theta'$, and otherwise map θ respectively to $push(\rho, \sigma(\theta))$, $\sigma(\theta)[i \mapsto v]$ and $\sigma(\theta)[i = v]$.

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.

Multi-threaded rules. The operational rules make use of the following notation. We write $store_\eta(\theta, l)$ for the oldest unwritten value of l stored by θ in η. More formally: let an event $(Store, \theta, l)_n$ in η be called *unwritten* if $(Write, \theta, l)_n$ is undefined in η; then, $store_\eta(\theta, l) = v$ if there exists an unwritten $(Store, \theta, l, v)_n$ such that for any unwritten $(Store, \theta, l)_m$ we have $n \leq m$; if no such *Store* event exists, $store_\eta(\theta, l)$ is undefined. Similarly, we write $rval_\eta(\theta, l)$ for the latest value of l assigned or loaded (obtained by the corresponding *Read*) by θ in η.

The operational semantics for multi-threaded Java is given in Table 6. There is a "primed" version [x'] for of each rule [x] of Section 2; [x'] is omitted if it reads as [x] by the notational conventions.

Properly speaking, those of Table 6 are rule *schemes* whose instances are obtained by replacing the metavariables with suitable semantic objects. This point is crucial for a correct understanding of the rules [assign3', val', lock, unlock, read, load, store, write]. Indeed, suitably instances of such schemes can be found only if the operation $\oplus$ is defined for the given arguments, that is if the action being performed complies with the requirements of the language specification. By [assign3'] and [val'] *Assign* and *Use* actions are only added to an event space, when dictated by execution of the current thread [3, 17.3]. The rules [read, load, store, write] are applied spontaneously. The [store] rule "guesses" the value of the last *Assign*: axiom (7) ensures that the guess is right.

For a concrete example, consider the "Possible Swap" program of Section 3. Assume that $\sigma(\theta_1, \mathrm{p}) = \sigma(\theta_2, \mathrm{p}) = o_{\mathrm{p}}$ for a stack σ and that $\mu(l_{\mathrm{p.x}}) = 1$ and $\mu(l_{\mathrm{p.y}}) = 2$ for a store μ. Then any run of this program starting from an empty

[assign3']	$(\theta, l = v), \eta \rightarrow (\theta, v), \eta \oplus (Assign, \theta, l, v)$
[assign4']	$(\theta, i = v), \sigma \rightarrow (\theta, v), \sigma[\theta, i \mapsto v]$
[val']	$(\theta, l), \eta \rightarrow (\theta, rval_\eta(\theta, l)), \eta \oplus (Use, \theta, l)$
[var']	$(\theta, i), \sigma \rightarrow (\theta, \sigma(\theta, i)), \sigma$
[block2']	$\dfrac{(\theta, S_1), push(\theta, \rho_1, \sigma_1) \rightarrow (\theta, S_2), push(\theta, \rho_2, \sigma_2)}{(\theta, \{S_1\}_{\rho_1}), \sigma_1 \rightarrow (\theta, \{S_2\}_{\rho_2}), \sigma_2}$
[synchro1]	$\dfrac{e_1 \rightarrow e_2}{\mathtt{synchronized}(e_1)\ q \rightarrow \mathtt{synchronized}(e_2)\ q}$
[synchro2]	$\dfrac{q_1 \rightarrow q_2}{\mathtt{synchronized}(o)\ q_1 \rightarrow \mathtt{synchronized}(o)\ q_2}$
[lock]	$\dfrac{(\theta, e), \eta_1 \rightarrow (\theta, o), \eta_2}{(\theta, \mathtt{synchronized}(e)\ q), \eta_1 \rightarrow (\theta, \mathtt{synchronized}(o)\ q), \eta_2 \oplus (Lock, \theta, o)}$
[unlock]	$(\theta, \mathtt{synchronized}(o)\ \{\ \}_\rho), \eta \rightarrow \eta \oplus (Unlock, \theta, o)$
[read]	$T, \eta, \mu \rightarrow T, \eta \oplus (Read, \theta, l, \mu(l)), \mu$
[load]	$T, \eta \rightarrow T, \eta \oplus (Load, \theta, l)$
[store]	$T, \eta \rightarrow T, \eta \oplus (Store, \theta, l, v)$
[write]	$T, \eta, \mu \rightarrow T, \eta \oplus (Write, \theta, l), \mu[l \mapsto store_\eta(\theta, l)]$
[par]	$\dfrac{t_1 \rightarrow t_2}{t_1 \mid T \rightarrow t_2 \mid T}$

Table 6. Multi-threaded Java

event space, stack σ, and store μ will eventually end up with $\mu(l_{\mathtt{p.x}}) = \mu(l_{\mathtt{p.y}}) = 1$ or $\mu(l_{\mathtt{p.x}}) = \mu(l_{\mathtt{p.y}}) = 2$. We detail a run where θ_1 is first in Figure 3. The event space of the end-configuration corresponds eventually to Figure 2, the final store is $\mu[l_{\mathtt{p.x}} \mapsto 2][l_{\mathtt{p.y}} \mapsto 2]$.

Conservativity. The operational semantics of multi-threaded Java extends conservatively the semantics given in Section 2 for the sequential part of the language. This is shown by Theorem 1 below, which exhibits a bisimulation between the two semantics where bisimilar configurations feature identical abstract terms. The significance of the theorem consists in showing that sequential programs can be reasoned about without loss of generality by using a simpler model of computation, free of working memories and forgetful of the past.

Below, a simple (unindexed) arrow $\rightarrow$ stands for the sequential Java semantics of Section 2 and $\rightarrow^=$ for its reflexive closure. We write $\rightarrow_\theta$ for the restriction of the multi-threaded Java semantics where the rules [read, load, store, write] involve actions of the thread θ only. The read/load extension $\rightsquigarrow_\theta$ of $\rightarrow_\theta$ to the left is inductively defined as follows: $\Gamma_1 \rightsquigarrow_\theta \Gamma_2$ when $\Gamma_1 \rightarrow_\theta \Gamma_2$ or $\Gamma_1 \rightarrow_\theta \Gamma_1'$

$$\begin{tikzcd}
((\theta_1,\ \texttt{synchronized(p)}\ \{\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \emptyset, \sigma, \mu) \arrow[d, "{[\text{lock,var'}]}"] \\
((\theta_1,\ \texttt{synchronized}(o_\texttt{p})\ \{\ \texttt{p.x} = \texttt{p.y};\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \{(\mathit{Lock}, \theta_1, o_\texttt{p})\}, \sigma, \mu) \arrow[d, "{\ldots]}", "{[\ldots,}"'] \\
((\theta_1,\ \texttt{synchronized}(o_\texttt{p})\ \{\ l_{\texttt{p.x}} = l_{\texttt{p.y}};\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \{(\mathit{Lock}, \theta_1, o_\texttt{p})\}, \sigma, \mu) \arrow[d, "{[\text{read}]}"] \\
((\theta_1,\ \texttt{sy}\ldots(o_\texttt{p})\ \{\ l_{\texttt{p.x}} = l_{\texttt{p.y}};\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \{\cdots \leq (\mathit{Read}, \theta_1, l_{\texttt{p.y}}, 2)\}, \sigma, \mu) \arrow[d, "{[\text{load}]}"] \\
((\theta_1,\ \texttt{sy}\ldots(o_\texttt{p})\ \{\ l_{\texttt{p.x}} = l_{\texttt{p.y}};\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \{\cdots \leq (\mathit{Load}, \theta_1, l_{\texttt{p.y}})\}, \sigma, \mu) \arrow[d, "{\text{assign2, val'}]}", "{[\text{synchro2, block2, statseq1, expstat1,}}"'] \\
((\theta_1,\ \texttt{sy}\ldots(o_\texttt{p})\ \{\ l_{\texttt{p.x}} = 2;\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \{\cdots \leq (\mathit{Use}, \theta_1, l_{\texttt{p.y}})\}, \sigma, \mu) \arrow[d, "{\text{assign3'}]}", "{[\text{synchro2, block2, statseq2, expstat2,}}"'] \\
((\theta_1,\ \texttt{sy}\ldots(o_\texttt{p})\ \{\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t),\ \{\cdots \leq (\mathit{Assign}, \theta_1, l_{\texttt{p.x}}, 2)\}, \sigma, \mu) \arrow[d, "{[\text{store}]}"] \\
((\theta_1,\ \texttt{sy}\ldots(o_\texttt{p})\ \{\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t), \{\cdots \leq (\mathit{Store}, \theta_1, l_{\texttt{p.x}}, 2)\}, \sigma, \mu) \arrow[d, "{[\text{write}]}"] \\
((\theta_1,\ \texttt{sy}\ldots(o_\texttt{p})\ \{\ \}_{\rho_\emptyset}) \mid (\theta_2,\ t), \{\cdots \leq (\mathit{Write}, \theta_1, l_{\texttt{p.x}})\}, \sigma, \mu[l_{\texttt{p.x}} \mapsto 2]) \arrow[d, "{[\text{unlock}]}"] \\
((\theta_1) \mid (\theta_2,\ t), \{\cdots \leq (\mathit{Unlock}, \theta_1, o_\texttt{p})\}, \sigma, \mu[l_{\texttt{p.x}} \mapsto 2])
\end{tikzcd}$$

Figure 3. Sample run of "Possible Swap"

by a *Read* or a *Load* action and $\Gamma_1' \rightsquigarrow_\theta \Gamma_2$. The store/write completion $\downarrow_\theta$ of $\rightarrow_\theta$ is defined as follows: $(\eta_1, \mu_1) \downarrow_\theta (\eta_2, \mu_2)$ when $(\eta_1, \mu_1) \rightarrow_\theta^* (\eta_2, \mu_2)$ by *Store* and *Write* actions only, and there is no Γ such that $(\eta_2, \mu_2) \rightarrow_\theta \Gamma$ by such an action. We write $(\eta_1, \mu_1) \downarrow_\theta \mu_2$ if there is an η_2 such that $(\eta_1, \mu_1) \downarrow_\theta (\eta_2, \mu_2)$. It is easy to verify that $\Gamma \downarrow_\theta \Gamma_1$ and $\Gamma \downarrow_\theta \Gamma_2$ imply $\Gamma_1 = \Gamma_2$. (Note that we use the same conventions as for the statement of the rules, i.e. we omit irrelevant configuration components.)

Let the binary relation $\sim_\theta$ between configurations of the sequential and multithreaded Java semantics be defined as follows:

$$(t, \sigma_1, \mu_1) \sim_\theta (T, \eta, \sigma_2, \mu_2) \quad \text{iff} \quad T = (\theta, t) \text{ and } \sigma_1 = \sigma_2(\theta) \text{ and } (\eta, \mu_2) \downarrow_\theta \mu_1\ .$$

Theorem 1. *For any configurations γ and Γ, if $\gamma \sim_\theta \Gamma$ then:*

(i) for all Γ', if $\Gamma \rightarrow_\theta \Gamma'$ then there exists γ' such that $\gamma \rightarrow^= \gamma'$ and $\gamma' \sim_\theta \Gamma'$;
(ii) for all γ', if $\gamma \rightarrow \gamma'$ then there exists Γ' such that $\Gamma \rightsquigarrow_\theta \Gamma'$ and $\gamma' \sim_\theta \Gamma'$.

Proof. By induction on the length of derivation of the operational judgements. The interesting cases involve the silent memory actions. We detail only some particularily involved cases for each direction.

(i) [val'] Let $l, \eta, \mu \rightarrow_\theta v, \eta \oplus (\mathit{Use}, \theta, l), \mu$, with $v = \mathit{rval}_\eta(\theta, l)$, and let $(l, \mu_1) \sim_\theta (l, \eta, \mu)$. It must be $(\eta, \mu) \downarrow_\theta \mu_1$. By rule (5), either an event $(\mathit{Assign}, \theta, l, v)$ is the most recent assignment to l by θ in η, or events $(\mathit{Read}, \theta, l, v) \leq (\mathit{Load}, \theta, l)$ occur in η and no assignment afterwards. In either cases $(\eta, \mu) \downarrow_\theta \mu_1$ imply $\mu_1(l) = v$. Hence, $l, \mu_1 \rightarrow v, \mu_1$ by [val]. Now, the store/write completion of η and $\eta \oplus (\mathit{Use})$ is the same and we thus have $(\eta \oplus (\mathit{Use}, \theta, l), \mu) \downarrow_\theta \mu_1$. Therefore, $(v, \mu_1) \sim_\theta (v, \eta \oplus (\mathit{Use}, \theta, l), \mu)$ as required.

[store, write] Let $(T, \mu_1) \rightarrow_\theta \Gamma$ by a *Store* or a *Write* action, and let $(t, \mu_2) \sim_\theta (T, \mu_1)$. It must be $(T, \mu_1) \downarrow_\theta \mu_2$. If $\Gamma \downarrow_\theta \mu_3$ then, composing transitions, we have $(T, \mu_1) \downarrow_\theta \mu_3$ and hence $\mu_2 = \mu_3$. Therefore $(t, \mu_2) \sim_\theta \Gamma$ as required after an identity $\rightarrow^=$ transition.

(ii) [assign1] Let $(e_1 = e, \mu_1) \sim_\theta (e_1 = e, \eta, \mu_2)$. We have immediately $(e_1, \mu_1) \sim_\theta (e_1, \eta, \mu_2)$. Let $e_1 = e, \mu_1 \rightarrow e_2 = e, \mu_3$ by a derivation whose last step involves the rule [assign1]. It must be $e_1, \mu_1 \rightarrow e_2, \mu_3$ by a shorter derivation. Hence, by inductive hypothesis, $e_1, \eta, \mu_2 \rightsquigarrow_\theta e_2, \eta_1, \mu_4$ and $(e_2, \mu_3) \sim_\theta (e_2, \eta_1, \mu_4)$ where, by definition, the read/load extension can be split into $e_1, \eta, \mu_2 \rightarrow_\theta^* e_1, \eta_2, \mu_2$ by a possibly empty sequence of silent *Read* and *Load* actions, and $e_1, \eta_2, \mu_2 \rightarrow_\theta e_2, \eta_1, \mu_4$. It follows that $e_1 = e, \eta, \mu_2 \rightarrow_\theta^* e_1 = e, \eta_2, \mu_2$ by the same sequence of silent actions and $e_1 = e, \eta_2, \mu_2 \rightarrow_\theta e_2 = e, \eta_1, \mu_4$ by [assign1], that is $e_1 = e, \eta, \mu_2 \rightsquigarrow_\theta e_2 = e, \eta_1, \mu_4$. Moreover, since $(e_2, \mu_3) \sim_\theta (e_2, \eta_1, \mu_4)$, we have $(e_2 = e, \mu_3) \sim_\theta (e_2 = e, \eta_1, \mu_4)$ as required.

5 Conclusions and Future Work

In this paper we have presented a structural operational semantics of the concurrency model of Java and we have shown how it relates to sequential Java. Our semantics covers a substantial part of the dynamic behaviour of the language. Most notably method calls, exceptions, and type information (class, interface and method declarations) are missing. We plan to investigate those parts in a further study. Method calls can easily be included in our semantics by the usual SOS techniques. The inclusion of exceptions is slightly more complicated; it requires the definition of evaluation contexts in order to keep the number of rules small. Concerning type information, we expect that one can easily combine our semantics with the type system developed in [2].

We have also not covered the full concurrency model of Java. Especially wait sets and notification as described in [3, 17.14] have to be added. There are some more detailed rules for variables declared `volatile` and for the non-atomic treatement of `double` and `long` variables; these are easily incorporated.

Furthermore, we are studying the flexibility of our approach by means of an extension to the so-called "prescient" store actions [3, 17.8]. These actions "allow optimizing Java compilers to perform certain kinds of code rearrangements that preserve the semantics of properly synchronized programs [...]."

References

1. Ken Arnold and James Gosling. *The Java Programming Language.* Addison–Wesley, Reading, Mass., 1996.
2. Sophia Drossopoulou and Susan Eisenbach. Java is Type Safe — Probably. In Mehmet Aksit, editor, *Proc. 11th Europ. Conf. Object-Oriented Programming*, volume 1241 of *Lect. Notes Comp. Sci.*, pages 389–418, Berlin, 1997. Springer.
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. Wei Li. An Operational Semantics of Multitasking and Exception Handling in Ada. In *Proc. AdaTEC Conf. Ada*, pages 138–151, New York, 1982. ACM SIGAda.
6. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Mass., 1997.
7. Gordon D. Plotkin. Structural Operational Semantics (Lecture notes). Technical Report DAIMI FN–19, Aarhus University, 1981 (repr. 1991).
8. 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.

A Syntax

$$
\begin{array}{rcl}
\mathit{Block} & ::= & \texttt{\{}\ \mathit{BlockStatement}^{*}\ \texttt{\}} \\
\mathit{BlockStatement} & ::= & \mathit{LocalVariableDeclaration} \mid \mathit{Statement} \\
\mathit{LocalVariableDeclaration} & ::= & \mathit{Type}\ \mathit{VariableDeclarator}^{+} \\
\mathit{VariableDeclarator} & ::= & \mathit{Identifier}\ \texttt{=}\ \mathit{Expression} \\
\mathit{Statement} & ::= & \texttt{;} \mid \mathit{Block} \mid \mathit{ExpressionStatement}\,\texttt{;} \\
 & & \mid \texttt{synchronized(}\ \mathit{Expression}\ \texttt{)}\ \mathit{Block} \\
\mathit{ExpressionStatement} & ::= & \mathit{Assignment} \mid \texttt{new}\ \mathit{ClassType}\ \texttt{()} \\
\mathit{Assignment} & ::= & \mathit{LeftHandSide}\ \texttt{=}\ \mathit{AssignmentExpression} \\
\mathit{LeftHandSide} & ::= & \mathit{Name} \mid \mathit{FieldAccess} \\
\mathit{Name} & ::= & \mathit{Identifier} \mid \mathit{Name}\ .\ \mathit{Identifier} \\
\mathit{FieldAccess} & ::= & \mathit{Primary}\ .\ \mathit{Identifier} \\
\mathit{AssignmentExpression} & ::= & \mathit{Assignment} \mid \mathit{UnaryExpression} \\
\mathit{UnaryExpression} & ::= & \mathit{UnaryOperator}\ \mathit{UnaryExpression} \\
 & & \mid \mathit{Primary} \mid \mathit{Name} \\
\mathit{Primary} & ::= & \mathit{Literal} \mid \texttt{this} \mid \mathit{FieldAccess} \\
 & & \mid \texttt{(}\ \mathit{Expression}\ \texttt{)} \mid \texttt{new}\ \mathit{ClassType}\ \texttt{()} \\
\mathit{Expression} & ::= & \mathit{AssignmentExpression}
\end{array}
$$