JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 4, July–August 2009

An Operational Semantics including "Volatile" for Safe Concurrency

John Boyland, University of Wisconsin–Milwaukee, USA

In this paper, we define a novel "write-key" operational semantics for a kernel language with fork-join parallelism, synchronization and "volatile" fields. We prove that programs that never suffer write-key errors are exactly those that are "data race free" and also those that are "correctly synchronized" in the Java memory model. This 3-way equivalence is proved using Twelf.

1 INTRODUCTION

This work is motivated by the desire to define a type system for a Java-like language to prevent data races. Data races are intrinsically a multi-threaded issue. However a scalable type system or program analysis analyzes each method body separately, using invariants and annotations to ensure that interactions follow desired patterns. It is well known that deadlock can be prevented by requiring that mutexes be acquired in strictly increasing order. Here we show how we can characterize programs without data races in a similar way, that is without explicitly needing to refer to multiple threads.

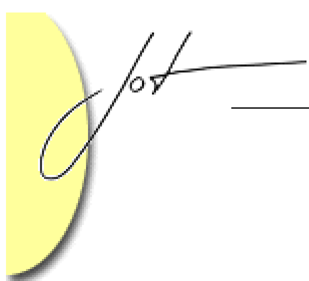
There are different understandings of what a data race is. At an intuitive level, a data race occurs when an execution of a multi-threaded program leads to the point where two conflicting accesses in two different threads occur "at the same time." Two accesses are *conflicting* if they are to the same object's field and one of them is a write. Somewhat more precisely, the current Java memory model (JMM) [?] defines a "happens before" partial order; a program is *correctly synchronized* if in all sequentially consistent executions, two conflicting accesses are always ordered by "happens before." Reading and writing of "volatile" fields affect the "happens before" order and thus whether a program is correctly synchronized. Reasoning about data races (simultaneous conflicting accesses) or about "happens before" explicitly involve reasoning about multiple threads at once.

This paper makes the following contributions:

- It defines a simple imperative language with Java-style (re-entrant) monitors,

Work supported in part by the National Science Foundation (CCF-0702635). The opinions expressed here are not necessarily those of the National Science Foundation or the US Government.

Cite this article2 as follows: John Boyland: *An Operational Semantics including "Volatile" for Safe Concurrency*, in Journal of Object Technology, vol. 8, no. 4, July–August 2009, pages



volatile fields and fork-join parallelism. A novel aspect of the operational semantics is that the running program uses “write keys” to simulate the “happens before” relation.

- The paper defines that a program is data-race free if no execution has a “write-key error” in which a thread attempts to access a (non-volatile) field for which it does not possess the write key.
- It is proved that this characterization is equivalent *both* to the intuitive concept of lack of “simultaneous” conflicting accesses, *and* to the JMM-inspired definition of “happens before”-ordered accesses. The proof is mechanically checked in Twelf’s metalogic [**?**].

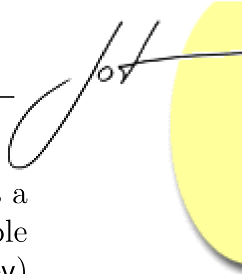
A corollary to the last contribution is that the two earlier conceptions of data-race freedom are equivalent. A similar result was proved by Boehm and Adve [**?**] for a generic system without dynamic allocation of objects or threads. We extend the proof to handle dynamic allocation, and also have a mechanically-checked proof.

The advantage of the “write-key error” conception for data races is that write-key errors are detected (and can be prevented) thread locally. In other words, if a type system can ensure for each thread that it always possesses the write keys for the (non-volatile) fields that it accesses and that it has exclusive access to fields it writes, then the entire program is thread safe. Moreover, since a write-key error causes a thread to get stuck in our semantics, then if a type system enjoys “progress” and “preservation” over the operational semantics, then *per force* the type system will also prevent race conditions.

2 BACKGROUND ON THE CURRENT JAVA MEMORY MODEL

This section briefly describes multi-threading primitives in Java and the “happens before” relation of the current Java Memory Model.

In Java, a new thread can be started which executes a given `run` method; we call this a fork action. At the other end, one may wait for a thread t to complete execution by executing t`.join()`; this is a join action. Thread mutual exclusion is effected by “synchronizing” on an object o. At the source-code level, this is expressed as `synchronized (`o`) {` *body* `}`. The runtime system ensures that two separate threads that both synchronize on the same object (playing the role of a *mutex*) mutually exclude each other’s “body” instructions. When a synchronized block is executed, it first attempts to acquire the mutex, blocking if some other thread is currently executing a synchronized block on the same object. Once acquisition is successful, the body is executed after which the mutex is released. Synchronized statements in Java are “re-entrant” in that if a synchronization block is nested dynamically within another synchronization block on the same object, the inner synchronization succeeds immediately.



Fields in Java may be declared as "volatile." This designation may be seen as a declaration that these fields may be read and written "simultaneously" by multiple threads. More importantly, accesses to volatile fields (denoted readv and writev) constrain the memory model.

A memory model is a contract between the programmer on the one hand and the compiler and the runtime system on the other hand. The most informative model for programmers is a "sequentially consistent" model that indicates that execution will always be consistent with a system in which the thread interleaving of instructions ensures that each instruction fully executes before the next starts. Sequential consistency however is very limiting for a compiler. Consider the following situation where two threads are executing in parallel, `x` and `y` represent shared mutable locations and `r`n represent unshared locations:

Initially `x = y = 0`	
`x = 1`	`r2 = y`
`y = 2`	`r1 = x`

In a sequentially consistent execution, no interleaving of the threads could result in `r1 = 0, r2 = 2`. If we restricted compilers to preserve sequential consistency, then a compiler would not be permitted to reorder the two write actions in the left thread because this could lead to a final result where `r1 = 0, r2 = 2`. Such a restriction thus prevents almost all reorderings, even here, where the assignments `x = 1` and `y = 2` have no data dependencies.

For such reasons, most memory models only guarantee sequential consistency for fields declared "volatile." For other fields, threads must use mutual exclusion techniques. The (intuitive) guarantee for normal (non-volatile) fields is that if a program is *data-race free*, that is, if no sequentially consistent execution ever exhibits a "race condition" for a normal field, then that program will enjoy sequentially consistent semantics. A *race condition* is when one thread is ready to write an object's field when another thread is ready to read or write the same object's field. If a program *could* exhibit a race condition under a sequentially consistent semantics, then most memory models usually do not guarantee a sequentially consistent semantics. In the small example given above, there is a race condition. Thus a compiler is justified if it wishes to reorder the statements, even though this reordering violates sequential consistency.

In the current Java memory model (JMM) [?], the guarantee is expressed in a different way. First, a "synchronizes with" relation is defined:

1. A release action synchronizes with an acquire action on the same object;
2. A writev action synchronizes with a readv action on the same object's field;
3. A fork action synchronizes with the first action in the spawned thread;
4. The last action in a thread synchronizes with a join action on that thread.

Additionally the default initialization of a field (with `null` for reference types) synchronizes with all actions in all threads.

Then the "happens before" partial order is defined as the transitive closure of (1) the intra-thread execution order and (2) the "synchronizes with" relation over actions already ordered by the execution.

Specifically of interest to the present paper, the JMM defines what it means to be "correctly synchronized":

> A program is *correctly synchronized* if and only if in all sequentially consistent executions, all conflicting accesses to non-volatile variables are ordered by "happens-before" edges.

The JMM (and variations [?]) guarantees that a correctly synchronized program will observe sequentially consistent semantics. The definition of "correctly synchronized" seems rather different from the definition of "data race free," but as proved in Sect. **??**, the definitions are equivalent for our small concurrent language.

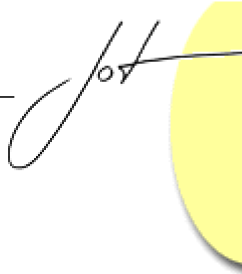
3 EXAMPLE

Figure 1 declares a node class. The surface syntax resembles Java, but method bodies contain expressions, not statements. For instance, `getNext()` returns the next field. The `count` method shows another difference: since the language omits dynamic dispatch for simplicity, one can call methods on null references. In the body, one may test for null. In this way, we can model so-called "static" methods. The `copy` method performs a deep copy; `nap` does a destructive append; `add1` extends the list by one node. The `Node` class has mutable state and thus cannot be safely used in a concurrent program without additional restrictions.

We now define several different classes wrapping a node list with the same interface: an `inc` method that adds to the list and a `get` method that counts the size of the current list. The first implementation, `Race` (Fig. 2), does nothing to protect the list. The main program forks off a thread that calls `inc` and `get` and then proceeds to do the same calls in its own thread. Lacking synchronization, the call `t.inc()` in one thread conflicts with `t.inc()` or `t.get()` in the other.

The traditional technique ("standard practice") for protecting mutable state is to designate a *protecting* object for each piece of mutable state (one object may protect many others) and ensure that all accesses to the state occur dynamically only within a synchronization on the protecting object. For example, see class `Traditional` in Fig. 3; the bodies of the methods `get()` and `inc()` include synchronizations around the access of the mutable state.

If `get()` calls are frequent and updates very infrequent, one can do better with a less-known pattern using volatile variables. Figure 4 shows how a volatile field can



```
class Node {
  Node next;
  Node(Node n) { next = n; }

  Node getNext() { next; }

  int count() {
    if this == null then 0
    else 1 + next.count();
  }

  Node copy() {
    if this == null then null
    else new Node(next.copy());
  }

  Node nap(Node n) {
    if this == null then n
    else (next = next.nap(n);
          this);
  }

  void add1() {
    this.nap(new Node(null));
  }
}
```

Figure 1: A simple node class.

```
class Race {
  Node nodes;

  Race() { }

  int get() {
    nodes.count();
  }

  void inc() {
    nodes = nodes.add1();
  }
}


class Main {

  void main() {
    let t = new Race() in
      ( fork { t.inc(); t.get(); }
        t.inc(); t.get() );
  }

}
```

Figure 2: A class with an unprotected field; and a test harness.

```
class Traditional {
  Node nodes;

  int get() {
    synch (this) do
      nodes.count();
  }

  void inc() {
    synch (this) do
      nodes = nodes.add1();
  }
}
```

Figure 3: Traditional approach.

```
class UsingVolatile {
  volatile Node nodes;

  int get() {
    nodes.count();
  }

  void inc() {
    synch (this) do
      nodes = nodes.copy().add1();
  }
}
```

Figure 4: Using volatility.

e ::=	*expression term:*
o	*literal reference*
x	*program variable*
`new` $(\overline{f})$	*allocation*
e`.`f	*field read*
e`.`f`:=`e	*field write*
`let` x`=`e `in` e	*local*
`if` c `then` e `else` e	*conditional*
`while` c `do` e	*loop*
m`(`$\overline{e}$`)`	*procedure call*
`fork` e	*fork a thread*
`join` e	*get thread result*
`synch` e `do` e	*synchronization*
`hold` o `do` e	*... in execution*
e_1`;`e_2 $\triangleq$	`let` `_`=e_1 `in` e_2

c ::=	*conditional term:*
`true`	*true*
`not` c	*negation*
c `and` c	*conjunction*
e `==` e	*equality*
`false` $\triangleq$	`not true`
c `or` c' $\triangleq$	`not(not` c `and not` c'`)`
t ::= $e \mid c$	*term*
d ::= m`(`$\overline{x}$`)` `=` e	*procedure definition*
g ::= d`;...;`d	*program*

Figure 5: Syntax.

substitute for synchronization. The reading method can simply access the nodes directly using a volatile field read, and then traverse the list without synchronization. The incrementing method copies the structure before modifying it, to avoid interfering with `get` calls. Furthermore, `inc` is synchronized to ensure that two increments are not carried out in parallel (to preserve “atomicity” [?], an important concept beyond the scope of this paper). We permit interleaving of `get()` and `inc()` calls since the `inc()` method never updates state the `get()` method can see, except for the volatile field.

4 OPERATIONAL SEMANTICS

This section defines the syntax and dynamic semantics of the paper’s kernel concurrent language. The set of all fields is F. A subset $F_V \subseteq F$ are “volatile” and one field in this subset ($\mathrm{Lock} \in F_V$) holds the state of the mutex associated with each object.

Syntax

Figure 5 gives the syntax. For simplicity, we omit primitive types and arithmetic operators. Expressions include literal object references (natural numbers) and uses of local variables. We omit classes: instead a new object is allocated with a given set of fields. Fields of objects can be read or written. The “`let`,” “`if`” and “`while`” constructs are conventional. Procedure calls are included, but not dynamic dispatch

$$
\begin{array}{lcl}
\texttt{null} & \triangleq & 0 \\
f & \triangleq & \texttt{this.}f \\
f\texttt{=}e & \triangleq & \texttt{this.}f\texttt{:=}e \\
e_0\texttt{.}m\texttt{(}\overline{e}\texttt{)} & \triangleq & m\texttt{(}e_0\texttt{,}\overline{e}\texttt{)} \\
\texttt{new}\ C\texttt{(}\overline{e}\texttt{)} & \triangleq & C\texttt{(new (Lock,fields(}C\texttt{)),}\overline{e}\texttt{)} \\
C\texttt{(}\overline{\tau\ x}\texttt{) \{}e\texttt{; \}} & \triangleq & C\texttt{(this,}\overline{x}\texttt{) = (}e\texttt{;this)} \\
C\texttt{() \{\}} & \triangleq & C\texttt{(this) = this} \\
\tau\ m\texttt{(}\overline{\tau\ x}\texttt{) \{}e\texttt{; \}} & \triangleq & m\texttt{(this,}\overline{x}\texttt{) = }e \\
\hline
\texttt{class}\ C\ \texttt{\{}\overline{\tau\ f}\ \overline{d}\ \texttt{\}} & \triangleq & \overline{\overline{d}}
\end{array}
$$

Figure 6: Translating surface syntax to underlying language

because the details would obscure the emphasis of this work.

The concurrency-related terms are fork-join terms (`fork` creates a new thread and starts it; and `join` waits for it to terminate) and synchronization (`synch` and `hold`). A `hold` expression is used to indicate that this thread is currently executing a `synch` statement. Programs do not include `hold` expressions; they arise during evaluation.

The examples in the previous section use a surface syntax with classes, methods and types. Figure 6 shows one way in which these features can be stripped out: we first see that the null pointer is represented by zero; read and writes of "this" fields in method bodies are made explicit; method calls are converted into procedure calls; "new" calls are converted into calls to a procedure (named by the class) on an object allocated with a lock field and the other fields of the class; the constructor declaration itself is converted into a procedure with additional formal parameter `this`; method declarations are similarly converted into procedure declarations. The last line of Fig. 6 simply says we retain only the procedure declarations converted from the constructors and methods in the program. We assume no duplicate name clashes, that all and only fields marked `volatile` are in the set F_V and that fields(C) (used to translate surface `new` expressions) includes all fields declared in class C before the translation.

Semantics

This section defines the small-step operational semantics. Novel here is the use of "write keys." Write keys allow us to separate the notion of "happens before" from considering the execution of multiple threads and instead look at a single thread at a time. A (possibly new) *write key* (a natural number) is generated whenever a normal field is written. This field is added to the *knowledge* of the thread that performed the write. Knowledge is monotonically non-decreasing. Write keys are

passed from one thread to another during synchronization actions *indirectly* through memory. In this way, if a write in one thread happens before the read in the other thread, the read is guaranteed to have the necessary key.

The main evaluation relation $(\mu; \theta; \kappa) \xrightarrow[g]{} (\mu'; \theta'; \kappa')$ relates triples:

μ maps a location (pair of object reference and field name (o, f)) to a pair (W, o') of a set of write keys and a value. For a normal field, $W = \{w\}$, where w is the key from the most recent write; for a volatile field, W is the set of keys from all threads having written it.

θ maps a thread identifier (object reference) to the expression that the thread is currently executing.

κ maps a thread identifier to its set of known write keys.

The following notations are used for map update, where $\bigcirc$ may be any operation:

$$f[x \mapsto v](x') = \begin{cases} v & \text{if } x = x' \\ f(x) & \text{otherwise} \end{cases} \qquad f[x \overset{\bigcirc}{\mapsto} v] = f[x \mapsto f(x) \bigcirc v]$$

Evaluation proceeds using EVAL (see Fig. **??**): a thread is chosen non-deterministically and evaluates one step. Here $T[.]$ defines the evaluation context. Here $(\mu; \theta; \kappa; t) \xrightarrow[g]{p} (\mu'; \theta'; \kappa'; t')$ states that thread p in program g makes progress by converting t into t' while side-effecting μ, θ and κ.

For explanatory reasons, the evaluation rules are presented in two groups. The first group of evaluation rules (Fig. **??**) are those that have no side-effects. A procedure call uses rule E-CALL once all the parameters are evaluated: we find a procedure in the program with the correct number of arguments and replace the call with the procedure body, substituting the parameters. A `let`-bound variable is substituted in the body once its value is ready. An `if` with a constant boolean is evaluated by choosing the appropriate branch. A `while` loop is converted immediately into an `if`. Conditions use short-circuit evaluation (E-ANDFALSE). Object equality uses equality on the (natural number) literals.

Figure **??** includes the remaining evaluation rules. As mentioned previously, normal fields are associated with a write key that indicates what knowledge is needed to read the field. When a `new` expression is encountered, all of the fields are initialized with null using a write key (0) that all threads know. (This follows the JMM—default initialization synchronizes with the first action in every thread.) Every object is allocated with a mutex (special field Lock).

Field reads and writes of non-volatile fields (E-READ, E-WRITE) require that the thread has knowledge of the write that produced the value: $w \in \kappa(p)$. For a write, an arbitrary write key w' is used to label the new write. In general, this may be one that no thread is yet aware of. Using such a key would cause the `Race`

$$\mathrm{T}[\bullet] ::= \bullet \mid \mathrm{T}[\bullet]\texttt{.}f \mid \mathrm{T}[\bullet]\texttt{.}f \texttt{ := } e \mid o\texttt{.}f \texttt{ := } \mathrm{T}[\bullet] \mid m(\overline{o}\texttt{,}\mathrm{T}[\bullet]\texttt{,}\overline{e})$$
$$\mid \texttt{let } x\texttt{=}\mathrm{T}[\bullet] \texttt{ in } e \mid \texttt{if } \mathrm{T}[\bullet] \texttt{ then } e \texttt{ else } e \mid \texttt{synch } \mathrm{T}[\bullet] \texttt{ do } e$$
$$\mid \texttt{hold } o \texttt{ do } \mathrm{T}[\bullet] \mid \mathrm{T}[\bullet] \texttt{ and } c \mid \texttt{not } \mathrm{T}[\bullet] \mid \mathrm{T}[\bullet] \texttt{ == } e \mid o \texttt{ == } \mathrm{T}[\bullet]$$

EVAL
$$\frac{\theta p = \mathrm{T}[t] \qquad (\mu;\theta;\kappa;t) \xrightarrow[g]{p} (\mu';\theta';\kappa';t')}{(\mu;\theta;\kappa) \xrightarrow[g]{} (\mu';\theta'[p \mapsto \mathrm{T}[t']];\kappa')}$$

E-CALL
$$\frac{g = \texttt{...; } m(\overline{x}) \texttt{ = } e\texttt{; ...} \qquad |\overline{x}| = |\overline{o}|}{(\mu;\theta;\kappa;m(\overline{o})) \xrightarrow[g]{p} (\mu;\theta;\kappa;[\overline{x \mapsto o}]e)}$$

E-LET
$$(\mu;\theta;\kappa;\texttt{let } x \texttt{ = } o_1 \texttt{ in } e_2) \xrightarrow[g]{p} (\mu;\theta;\kappa;[x \mapsto o_1]e_2)$$

E-IF
$$(\mu;\theta;\kappa;\texttt{if } c \texttt{ then } e_{\texttt{true}} \texttt{ else } e_{\texttt{false}}) \xrightarrow[g]{p} (\mu;\theta;\kappa;e_c)$$

E-WHILE
$$(\mu;\theta;\kappa;\texttt{while } c \texttt{ do } e) \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{if } c \texttt{ then } e\texttt{; while } c \texttt{ do } e \texttt{ else } 0)$$

E-NOTNOTTRUE
$$(\mu;\theta;\kappa;\texttt{not false}) \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{true})$$

E-ANDTRUE
$$(\mu;\theta;\kappa;\texttt{true and } c) \xrightarrow[g]{p} (\mu;\theta;\kappa;c)$$

E-ANDFALSE
$$(\mu;\theta;\kappa;\texttt{false and } c) \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{false})$$

E-EQUALTRUE
$$\frac{o = o'}{(\mu;\theta;\kappa;o\texttt{==}o') \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{true})}$$

E-EQUALFALSE
$$\frac{o \neq o'}{(\mu;\theta;\kappa;o\texttt{==}o') \xrightarrow[g]{p} (\mu;\theta;\kappa;\texttt{false})}$$

Figure 7: Non-concurrency-related evaluation rules.

program in earlier Fig. 2 to get stuck when the second increment executes. The new key is then placed in this thread's knowledge.

One way in which knowledge of writes can be transmitted is through volatile fields (E-READV, E-WRITEV). Writing a volatile field adds the thread's knowledge $\kappa(p)$ to the memory with the written value. When the volatile field is read, the reading thread picks up this knowledge. This follows the JMM rule that says that writing a volatile field synchronizes with all following reads.

For E-FORK, the new thread gets the knowledge of the "forker." This corresponds to the JMM rule that a fork synchronizes with the first action in the new thread. An object is allocated to represent the thread. In E-JOIN, this thread can only progress if the other thread has finished execution (down to a value). It gets a copy of all the thread's knowledge. This follows from the JMM's rule that the final

E-NEW

$$\frac{f_0 = \text{Lock} \qquad (o, \text{Lock}) \notin \text{Dom}(\mu) \qquad f_i \text{ distinct}}{(\mu; \theta; \kappa; \texttt{new } (f_1, \ldots, f_n)) \xrightarrow[g]{p} (\mu[(o, f_i) \mapsto (\{0\}, 0) \mid 0 \leq i \leq n]; \theta; \kappa; o)}$$

E-READ

$$\frac{\mu(o, f) = (\{w\}, o') \quad w \in \kappa(p) \qquad f \notin F_V}{(\mu; \theta; \kappa; o\,.\,f) \xrightarrow[g]{p} (\mu; \theta; \kappa; o')}$$

E-WRITE

$$\frac{\mu(o, f) = (\{w\}, _) \quad w \in \kappa(p) \qquad f \notin F_V \qquad w' \text{ arbitrary} \quad \mu' = \mu[(o, f) \mapsto (\{w'\}, o')] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o\,.\,f \texttt{:=} o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

E-READV

$$\frac{\mu(o, f) = (W, o') \qquad \text{Lock} \neq f \in F_V \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} W]}{(\mu; \theta; \kappa; o\,.\,f) \xrightarrow[g]{p} (\mu; \theta; \kappa'; o')}$$

E-WRITEV

$$\frac{\mu(o, f) = (W, _) \qquad \text{Lock} \neq f \in F_V \quad \mu' = \mu[(o, f) \mapsto (W \cup \kappa(p), o')]}{(\mu; \theta; \kappa; o\,.\,f \texttt{:=} o') \xrightarrow[g]{p} (\mu'; \theta; \kappa; o')}$$

E-FORK

$$\frac{(p', \text{Lock}) \notin \text{Dom}(\mu)}{(\mu; \theta; \kappa; \texttt{fork } e) \xrightarrow[g]{p} (\mu[(p', \text{Lock}) \mapsto (\{0\}, 0)]; \theta[p' \mapsto e]; \kappa[p' \mapsto \kappa(p)]; p')}$$

E-JOIN

$$\frac{\theta(p') = o}{(\mu; \theta; \kappa; \texttt{join } p') \xrightarrow[g]{p} (\mu; \theta; \kappa[p \overset{\cup}{\mapsto} \kappa(p')]; o)}$$

E-RE-ENTER

$$\frac{\mu(o, \text{Lock}) = (\emptyset, p)}{(\mu; \theta; \kappa; \texttt{synch } o \texttt{ do } e) \xrightarrow[g]{p} (\mu; \theta; \kappa; e)}$$

E-ACQUIRE

$$\frac{\mu(o, \text{Lock}) = (W, 0) \qquad W \neq \emptyset \quad \mu' = \mu[(o, \text{Lock}) \mapsto (\emptyset, p)] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} W]}{(\mu; \theta; \kappa; \texttt{synch } o \texttt{ do } e) \xrightarrow[g]{p} (\mu'; \theta; \kappa'; \texttt{hold } o \texttt{ do } e)}$$

E-RELEASE

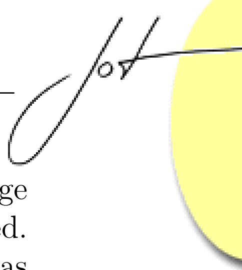
$$\frac{\mu(o, \text{Lock}) = (\emptyset, p) \quad \mu' = \mu[(o, \text{Lock}) \mapsto (\kappa(p), 0)]}{(\mu; \theta; \kappa; \texttt{hold } o \texttt{ do } o') \xrightarrow[g]{p} (\mu'; \theta; \kappa; o')}$$

Figure 8: Remaining evaluation rules.

action in a thread synchronizes with a thread that "joins" it.

During synchronization, the lock's value is replaced with the number of the acquiring thread, and the knowledge is replaced by the empty set. In E-RE-ENTER, if we synchronize on a lock that this thread already has acquired, the body is simply executed without any effect on the lock. This last rule corresponds to Java's re-entrant monitors; here, we avoid the need to count multiple entrances because the evaluation rule drops the release action as well as the acquire action.

If the lock is not held by any thread (E-ACQUIRE), the lock field is assigned the



number of this thread, and we get the keys from the lock. The lock's knowledge (which must not be empty) is cleared to indicate that the lock has been acquired. The synchronization block is then converted into a hold block. When the body has finished evaluation (E-RELEASE), the lock is given the knowledge of the current thread. This knowledge is thus made available for the next thread which acquires the lock. These rules again follow from the JMM.

The semantics defined here is sequentially consistent, but if a thread lacks the necessary write key, it gets stuck. Thus if the program has race conditions, it *may* get stuck (but does not necessarily get stuck, for instance if an old key is chosen by E-WRITE). A type system for this language that enjoys progress and preservation for *all* executions will prevent this. We have designed a type system [?] based on fractional permissions [?, ?] that we believe will achieve this goal, but space precludes including it here.

5 ANALYSIS

In this section, we go back to the example programs and show how they fare using the operational semantics defined in the previous section. Execution starts by calling the `main` procedure in thread 0, which starts with no knowledge except write key 0. More formally, we start execution in a special state $I = (\mu_0; \theta_0; \kappa_0)$:

$$\begin{aligned}
\mu_0 &= [(0, \text{Lock}) \mapsto (\{0\}, 0)] \\
\theta_0 &= [0 \mapsto \texttt{main()}] \\
\kappa_0 &= [0 \mapsto \{0\}]
\end{aligned}$$

At the beginning, there is only one allocated object, that for the "null" reference. It has no (normal) fields, and its lock is currently unacquired. For mathematical convenience, the "null object" is used for the initial thread as well. Of course in a realistic system, there would be no such object, and the initial thread would be non-null. We use an object for null because our formalization of new allocates any free location, and we want to make sure it never allocates null.

Figure **??** shows a possible evaluation of the code in Figs. 1 and 2 in which execution gets stuck. In this example, the main thread proceeds along calling `inc` until we reach θ_{16} just after field `1.nodes` is assigned using a write key 88, not known to thread 2. Thread 2 can proceed one step as seen in θ_{17}, but cannot progress further because the only write key it knows is the initial key, 0, that it received when it was forked by thread 1 (as seen in θ_5). Eventually thread 0 will complete and execution will be stuck. (The astute reader may notice that our low-level formalism doesn't actually support integer addition. It can only be hoped that this deficiency does not detract too much from the example.)

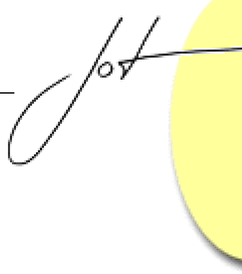
The evaluation is not guaranteed to get stuck. For instance if when defining μ_5, we had reused write key 0, there would be no problem. But, because of the *possibility*

$$
\begin{array}{lll}
(\mu_0, \theta_0, \kappa_0) & & \\
\underset{g}{\rightarrow} (\mu_0, \theta_1, \kappa_0) & \text{where} & \theta_1 = \theta_0[0 \mapsto \texttt{let t = Race(new(nodes)) in ...}] \\
\underset{g}{\rightarrow} (\mu_1; \theta_2; \kappa_0) & \text{where} & \mu_1 = \mu_0[(1, \text{Lock}) \mapsto (\{0\}, 0), (1, \texttt{nodes}) \mapsto (\{0\}, 0)], \\
 & & \theta_2 = \theta_1[0 \mapsto \texttt{let t = Race(1) in ...}] \\
\underset{g}{\rightarrow} (\mu_1, \theta_3, \kappa_0) & \text{where} & \theta_3 = \theta_2[0 \mapsto \texttt{let t = 1 in ...}] \\
\underset{g}{\rightarrow} (\mu_1, \theta_4, \kappa_0) & \text{where} & \theta_4 = \theta_3[0 \mapsto \texttt{fork\{inc(1);get(1)\};inc(1);get(1)}] \\
\underset{g}{\rightarrow} (\mu_2; \theta_5; \kappa_1) & \text{where} & \mu_2 = \mu_1[[(2, \text{Lock}) \mapsto (\{0\}, 0)]], \\
 & & \theta_5 = \theta_4[0 \mapsto \texttt{2;inc(1);get(1)}][2 \mapsto \texttt{inc(1);get(1)}] \\
 & & \kappa_1 = \kappa_0[2 \mapsto \{0\}] \\
\underset{g}{\rightarrow} (\mu_2, \theta_6, \kappa_1) & \text{where} & \theta_6 = \theta_5[0 \mapsto \texttt{inc(1);get(1)}] \\
\underset{g}{\rightarrow} (\mu_2, \theta_7, \kappa_1) & \text{where} & \theta_7 = \theta_6[0 \mapsto \texttt{1.nodes:=add1(1.nodes);get(1)}] \\
\underset{g}{\rightarrow} (\mu_2, \theta_8, \kappa_1) & \text{where} & \theta_8 = \theta_7[0 \mapsto \texttt{1.nodes:=add1(0);get(1)}] \\
\underset{g}{\rightarrow} (\mu_2, \theta_9, \kappa_1) & \text{where} & \theta_9 = \theta_8[0 \mapsto \texttt{1.nodes:=nap(0,Node(new ...));get(1)}] \\
\underset{g}{\rightarrow} (\mu_3; \theta_{10}; \kappa_1) & \text{where} & \mu_3 = \mu_2[(3, \text{Lock}) \mapsto (\{0\}, 0), (3, \texttt{next}) \mapsto (\{0\}, 0)], \\
 & & \theta_{10} = \theta_9[0 \mapsto \texttt{1.nodes:=nap(0,Node(3,0));get(1)}] \\
\underset{g}{\rightarrow} (\mu_3, \theta_{11}, \kappa_1) & \text{where} & \theta_{11} = \theta_{10}[0 \mapsto \texttt{1.nodes:=nap(0,3.next:=0;3);get(1)}] \\
\underset{g}{\rightarrow} (\mu_4; \theta_{12}; \kappa_2) & \text{where} & \mu_4 = \mu_3[(3, \texttt{next}) \mapsto (\{42\}, 0)], \\
 & & \theta_{12} = \theta_{11}[0 \mapsto \texttt{1.nodes:=nap(0,0;3);get(1)}] \\
 & & \kappa_2 = \kappa_1[0 \mapsto \{0, 42\}] \\
\underset{g}{\rightarrow} (\mu_4, \theta_{13}, \kappa_2) & \text{where} & \theta_{13} = \theta_{12}[0 \mapsto \texttt{1.nodes:=nap(0,3);get(1)}] \\
\underset{g}{\rightarrow} (\mu_4, \theta_{14}, \kappa_2) & \text{where} & \theta_{14} = \theta_{13}[0 \mapsto \texttt{1.nodes:=if0==0then3else...;get(1)}] \\
\underset{g}{\rightarrow} (\mu_4, \theta_{15}, \kappa_2) & \text{where} & \theta_{15} = \theta_{14}[0 \mapsto \texttt{1.nodes:=3;get(1)}] \\
\underset{g}{\rightarrow} (\mu_5; \theta_{16}; \kappa_3) & \text{where} & \mu_5 = \mu_4[(1, \texttt{nodes}) \mapsto (\{88\}, 3)], \\
 & & \theta_{16} = \theta_{15}[0 \mapsto \texttt{3;get(1)}] \\
 & & \kappa_3 = \kappa_2[0 \mapsto \{0, 42, 88\}] \\
\underset{g}{\rightarrow} (\mu_5, \theta_{17}, \kappa_3) & \text{where} & \theta_{17} = \theta_{16}[2 \mapsto \texttt{1.nodes:=add1(1.nodes);get(1)}] \\
 & \vdots &
\end{array}
$$

Figure 9: Sample Execution of Race Program (Figs. 1 and 2).

of an execution getting stuck, a type system for which progress and preservation were proved could not accept this program.

With traditional synchronization, the second thread would be unable to execute its procedure body while the left `inc()` is executing. Then, when the body is finally entered, the second thread gets knowledge of the write key used to update the field. (This explanation assumes that the original thread acquired the lock first. Should the second thread acquire the lock first, an analogous explanation can be given.)



$$
\begin{array}{lll}
(\mu_2; \theta_6; \kappa_1) & & \\
\underset{g}{\rightarrow} (\mu_2, \theta_7, \kappa_1) & \text{where} & \theta_7 = \theta_6[0 \mapsto \texttt{synch 1 do 1.nodes:=...;get(1)}] \\
\underset{g}{\rightarrow} (\mu_3; \theta_8; \kappa_2) & \text{where} & \mu_3 = \mu_2[(1, \text{Lock}) \mapsto (\{\}, 0)], \\
 & & \theta_8 = \theta_7[0 \mapsto \texttt{hold 1 do 1.nodes:=...;get(1)}] \\
 & & \kappa_2 = \kappa_1[0 \mapsto \{0\}] \\
 & \vdots & \\
\underset{g}{\rightarrow} (\mu_3, \theta_{11}, \kappa_2) & \text{where} & \theta_{11} = \theta_{10}[0 \mapsto \texttt{hold 1 do 1.nodes:=add1(0);get(1)}] \\
\underset{g}{\rightarrow} (\mu_3, \theta_{12}, \kappa_2) & \text{where} & \theta_{12} = \theta_{11}[0 \mapsto \texttt{hold 1 do 1.nodes:=nap(...);get(1)}] \\
 & \vdots & \\
\underset{g}{\rightarrow} (\mu_4, \theta_{18}, \kappa_3) & \text{where} & \theta_{18} = \theta_{17}[0 \mapsto \texttt{hold 1 do 1.nodes:=3; get(1)}] \\
\underset{g}{\rightarrow} (\mu_5; \theta_{19}; \kappa_3) & \text{where} & \mu_5 = \mu_4[(1, \texttt{nodes}) \mapsto (\{0, 42\}, 3)], \\
 & & \theta_{19} = \theta_{18}[0 \mapsto \texttt{hold 1 do 3; get(1)}] \\
\underset{g}{\rightarrow} (\mu_6; \theta_{20}; \kappa_3) & \text{where} & \mu_6 = \mu_5[(1, \text{Lock}) \mapsto (\{0, 42\}, 0)], \\
 & & \theta_{20} = \theta_{19}[0 \mapsto \texttt{3; get(1)}] \\
\underset{g}{\rightarrow} (\mu_6, \theta_{21}, \kappa_3) & \text{where} & \theta_{21} = \theta_{20}[0 \mapsto \texttt{get(1)}] \\
\underset{g}{\rightarrow} (\mu_6, \theta_{22}, \kappa_3) & \text{where} & \theta_{22} = \theta_{21}[2 \mapsto \texttt{synch 1 do 1.nodes:=...;get(1)}] \\
 & \vdots & \\
\underset{g}{\rightarrow} (\mu_9; \theta_{34}; \kappa_4) & \text{where} & \mu_9 = \mu_8[(1, \texttt{nodes}) \mapsto (\{0, 42, 25\}, 4)], \\
 & & \theta_{34} = \theta_{33}[2 \mapsto \texttt{hold 1 do 4; get(1)}] \\
\underset{g}{\rightarrow} (\mu_{10}; \theta_{35}; \kappa_4) & \text{where} & \mu_{10} = \mu_9[(1, \text{Lock}) \mapsto (\{0, 42, 25\}, 0)], \\
 & & \theta_{35} = \theta_{34}[2 \mapsto \texttt{4; get(1)}] \\
\underset{g}{\rightarrow} (\mu_{10}, \theta_{36}, \kappa_4) & \text{where} & \theta_{36} = \theta_{35}[2 \mapsto \texttt{get(1)}] \\
\underset{g}{\rightarrow} (\mu_{10}, \theta_{37}, \kappa_4) & \text{where} & \theta_{37} = \theta_{36}[0 \mapsto \texttt{count(1.nodes)}] \\
\underset{g}{\rightarrow} (\mu_{10}, \theta_{38}, \kappa_4) & \text{where} & \theta_{38} = \theta_{37}[0 \mapsto \texttt{count(4)}] \\
 & & \kappa_4 = \kappa_4[0 \mapsto \{0, 42, 25\}, 0]
\end{array}
$$

Figure 10: Sample execution using `volatile` (Figs. 1 and 4)

Figure **??** shows a sample execution of the code with class `UsingVolatile` replacing `Race`. The mutual exclusion prevents the two `inc` calls from running in parallel. The execution in the figure assumes that the original thread first calls `inc` and then the second thread. Then it assumes that the first thread calls `get`. (If this call had happened before thread 2 wrote `1.nodes`, it would get the old value of the list.) When it reads the volatile field `1.nodes` at step θ_{38}, rule E-ReadV gives to thread 0 the knowledge (25) of the node added by thread 2 and so execution proceeds. Indeed in all possible interleavings, a thread will always have the knowledge it needs to read the fields it accesses. In other words, the code is thread-safe.

6 EQUIVALENCE

Programs that execute in our operational semantics without ever blocking because of missing write keys are "correctly synchronized" according to (our variant) of the Java Memory Model *and* to the traditional definition of "race free." In other words, we show a three-way equivalence.

Formalism

In order to prove equivalence, we need to formally define the aspects we are showing equivalent. To start with, we restrict programs so that they do not include arbitrary object reference constants:

Definition 6.1 A program g is *valid* if every declaration $m(\bar{x}) = e$ in g has no instance of a literal object reference except the null reference 0.

If a program had access to an unallocated reference, we would lose the ability to reorder an allocation in one thread with an access in another thread that "accidentally" uses the newly allocated reference. The inability to reorder would falsify Theorem **??**.

We formalize what it means for there to be a *write key error* in a program:

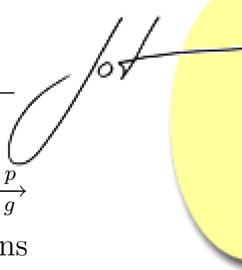
Definition 6.2 A program $g = \overline{d}$ has a write key error if for some execution $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ a read or write access on a non-volatile field $o.f$ is ready to execute in thread p ($\theta(p) = \mathrm{T}[o\,.\,f\,{:=}o']$ or $\theta(p) = \mathrm{T}[o\,.\,f]$, where $f \notin F_V$), and the thread does not have the required write key: ($\mu(o, f) = (\{w\}, _)$ and $w \notin \kappa(p)$).

Next, we formalize what it means to have a "race condition": a write access to a field happens at the "same time" as a read or write access to the same field, and that field is not volatile. We start off by defining what "conflicting accesses" are:

Definition 6.3 Two terms t_1 and t_2 are *conflicting accesses* of a non-volatile field $o.f$ ($f \notin F_v$) if one of them is a write to this field ($t_i = o\,.\,f\,{:=}o'$) and the other is a write ($t_{3-i} = o\,.\,f\,{:=}o''$) or a read ($t_{3-i} = o\,.\,f$) of the same field.

Definition 6.4 A program g exhibits a *race condition* if there is some execution $I \xrightarrow[g]{*} (\mu, \theta, \kappa)$ such that for two threads $p_1 \neq p_2$ we have $\theta(p_i) = T[t_i]$ and t_1, t_2 are conflicting accesses.

Before we can define what it means to be "correctly synchronized," we must define an "action" and the "happens-before" relation for actions:



Definition 6.5 An *action* λ is an instance of the evaluation relation $(\mu;\theta;\kappa;t) \xrightarrow[g]{p} (\mu';\theta';\kappa';t')$. An evaluation sequence $I \xrightarrow[g]{*} (\mu,\theta,\kappa)$ induces a sequence of the actions above the line for each instance of EVAL: $\lambda_1, \lambda_2, \ldots, \lambda_n$. This is called an *execution.*

Definition 6.6 Given an execution $\lambda_1, \ldots, \lambda_n$, we define a *happens-before* (written $i \sqsubset j$) relation on the subset of natural numbers $\{1, \ldots, n\}$. It is the smallest transitive relation that includes the following pairs:

1. $i \sqsubset j$ if $i < j$ and λ_i is an instance of E-RELEASE and λ_j is an instance of E-ACQUIRE on the same object.
2. $i \sqsubset j$ if $i < j$ and λ_i is an instance of E-WRITEV and λ_j is an instance of E-READV on the same field of the same object.
3. $i \sqsubset j$ if $i < j$ and $\lambda_i = _ \xrightarrow[g]{p} _$ and $\lambda_j = _ \xrightarrow[g]{p} _$.
4. $i \sqsubset j$ if $i < j$ and $\lambda_i = (\mu;\theta;\kappa;\texttt{fork}\ t) \xrightarrow[g]{p} (\mu';\theta';\kappa';q)$ and $\lambda_j = _ \xrightarrow[g]{q} _$.
5. $i \sqsubset j$ if $i < j$ and $\lambda_i = _ \xrightarrow[g]{q} _$ and $\lambda_j = (\mu;\theta;\kappa;\texttt{join}\ q) \xrightarrow[g]{p} (\mu';\theta';\kappa';t)$.

It can be easily shown that $\sqsubset$ is a partial order compatible with $<$.

The initial acquisition of a lock does not induce any extra "happens-before" edges. This definition of "happens-before" closely follows the JMM, but is simplified in several ways. The greatest difference is that we assume a sequentially consistent semantics, whereas a "valid" execution in the JMM can "commit" writes before they happen in program order. The JMM also has a particular semantics of "final" fields that we do not model.

Our final definition for *correctly synchronized* also follows the style of the JMM:

Definition 6.7 A program g is correctly synchronized if for any execution of g: $\lambda_1, \ldots, \lambda_n$ and any i for which λ_i is an instance of E-WRITE and any j for which λ_j is an instance of E-READ or E-WRITE for the same field, then either $i \sqsubset j$ or $j \sqsubset i$.

It might seem that because our operational semantics detects race conditions, the conflicting access would never execute and thus could not demonstrate an incorrect synchronization, but because write keys are arbitrary, the write could use 0 and thus enable execution. The semantics does not ensure that *all* executions of a program with race conditions will get stuck, just that there will be *some* execution that does.

We now show the three-way equivalence between the three conceptions of race-freedom:

Theorem 6.8 *The following statements about a valid program g are equivalent:*

1. *g exhibits a race condition;*
2. *g has a write key error;*
3. *g is incorrectly synchronized.*

PROOF (Sketch)

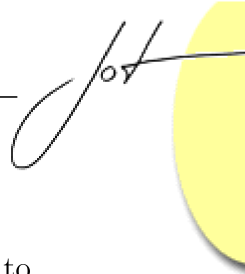
(1) ⇒ (2): Suppose we have a program with a race condition. Starting with the execution state that exhibits the race condition, we choose to evaluate the write first. If this write cannot execute because of a missing write key, we are done. Otherwise we choose a new write key not known by the other thread, and we now have a write key error.

(2) ⇒ (3): We prove the contrapositive: if the program is correctly synchronized, there will be no write-key error. This is because if there is a happens-before connection between two actions, the thread knowledge of the second will include that produced by the first, and thus the second access will succeed. The connection between write key knowledge and happens-before follows from the fact that the knowledge never decreases (the first case for happens-before) and the other cases for happens-before involve the reader/acquirer getting all the write keys left by the writer.

(3) ⇒ (1): Suppose we have an incorrectly synchronized program, in which the code of the first action λ_i is a write executed in thread p and the second action λ_j is an access executed in thread q. (The case that λ_i is a read and λ_j is a write is analogous.)

If the actions are already consecutive, we have the required race condition in the state just before the first executed. Otherwise, we consider how evaluation actions can be reordered (between different threads, never within a thread) to get the accesses adjacent. We partition the intervening actions into those that happen before j and those which do not. The second must include action i, from the definition of incorrect synchronization. We find the last action λ_* in the second group. It cannot be "happens before" any in the first group, or a transitive happens-before relation would exist putting it *in* the first group. Now we reorder it step-by-step with all later actions until it is after λ_j. If λ_* was λ_i, then the last reordering would have resulted in the required race condition. Otherwise, now that it is after λ_j we have reduced the number of intervening instructions. This process must terminate at some point.

□



Twelf Realization

See http://www.cs.uwm.edu/~boyland/papers/simple-concur.html for how to download the Twelf proof.

7 EXTENSIONS

Extending the simple language here to full Java is almost entirely just a matter of complex but uninteresting details. Static fields and static synchronization can be modeled using instance fields and instance synchronization of singleton objects. Types, primitive values and dynamic dispatch have no effect on concurrency.

The `Thread` class includes a number of deprecated methods that permit one thread to suspend or terminate another. These we can omit from the formalism. Other methods such as `holdsLock` can be implemented without affecting the proof substantially, because they only apply to the current thread.

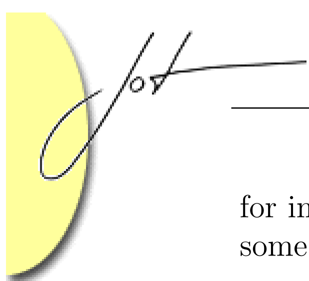
Java's `wait`/`notify` system would require substantive changes to the formalism. When a thread calls `wait`, it first releases the object's lock, then it waits to be "notified" and then it waits to re-acquire the lock. The lock release and acquisition lead to the corresponding standard happens-before relations. Another missing piece is thread interruption (and the corresponding interrupted exception). My guess is that the proof could be modified to handle `wait` and interruption.

On the other hand, Java 5 adds a new concurrency library, some of whose primitives (such as compare-and-swap and "fences") have unclear effects on the "happens before" relation. Timing issues would also be difficult to model and handle in proofs, since the proofs rely on the ability to swap unrelated thread computations.

8 RELATED WORK

A precursor to the JMM by Manson and Pugh [?] defined a semantics in which write keys were generated at writes and threads kept track of all writes known to them. Unlike the present work, it requires all write keys to be distinct, the knowledge includes the values written as well as the write key, and it does not require reads or writes to have knowledge of the "current" write. Indeed it does not even assume there is a single value for every field: a field read can pick up the value of any previous write not known to be overwritten.

The current Java memory model is much more complex than what is modeled here. In particular it gives semantics for programs that are *not* properly synchronized. Aspinall and Ševčík [?] formally prove the main guarantee—that correctly synchronized programs will have a sequentially consistent semantics (whereas the work described here *assumes* sequential consistency). The initialization of reference fields causes some concern which we avoid by using a universally known write key



for initialization. Saraswat provides a variant model that abstracts and generalizes some of the concepts found in the JMM and other memory models [?].

Boehm and Adve [?] describe the proposed C++ memory model. Here the semantics of programs that are not correctly synchronized is left undefined. This gives much more freedom to the compiler writer. They prove that the informal idea of race conditions coincides with the more formal definition of incorrect synchronization. Here we add to this result through a mechnically checked proof for a richer language (with conditionals and procedures). More importantly, we show that the correspondence between the two concepts holds even in the presence of dynamic allocation, as long as programs are not permitted to manufacture object identifiers (i.e. no casts from `int` to pointers.) This condition is not true in C++, and thus it seems that the correspondence Boehm and Adve prove would not hold for full C++.

Cenciarelli, Knapp and Sibilio [?] give a vastly different semantics of some aspects of the Java Memory Model based on configuration structures. As with the papers just reviewed, it handles programs with data races, not just properly synchronized ones, and does not assume sequential consistency.

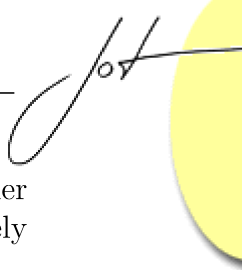
Type systems have been proposed that prevent race conditions and sometimes deadlocks in concurrent programming languages. Flanagan and Abadi [?, ?] define two separate type systems for avoiding races, both of which are accompanied by operational semantics. One is based on Gordon and Hankin's concurrent object calculus [?] in which mutable objects are represented in the syntax as concurrent processes. The other uses a conventional store. Neither semantics directly detects race conditions, nor includes "volatile." In either case, a race condition is defined as the (global) possibility that a write could occur at the same time as a read of the same field. (In one system [?], two "simultaneous" reads are also considered a race.) The type system maintains certain invariants that are shown to prevent data races.

Later work (such as Flanagan and Freund [?], Greenhouse [?] and Boyapati and Rinard [?, ?]) omit formal specification of operational semantics, implicitly following the same approach just outlined. Volatile fields, if they are handled at all, are simply regarded as loopholes in the type system.

Permandla and Boyapati [?] define a small-step semantics for a subset of Java virtual machine language (JVML) including synchronization (but not volatile fields) and show that well-typed programs are free of concurrency errors. The semantics however enforces an ownership model and uses method annotations that indicate required locking state. The operational semantics is not independent of the type system.

Guava [?] uses a type system to prevent races in a dialect of Java. Guava permits reader/reader parallelism, but omits volatiles. Guava is defined by (informally described) compilation to Java byte-code. Guava is intended as a practical programming language rather than as a minimal concurrent language.

Brookes [?] gives the semantics of a concurrent program by defining its set of "action traces." Roughly this means that all possible interleavings are considered. A



race condition in which a write to mutable state is directly interleaved with another access to the same state is "catastrophic," in that this particular trace immediately aborts. The semantics omits "volatile."

9 CONCLUSIONS

This paper defines an operational semantics of volatile fields that enables a type system to reason compositionally about them. It uses write keys to detect threading violations. It shows that write-key errors occur if and only if the program may exhibit a race condition, if and only if it is not correctly synchronized.

Acknowledgments

I thank Aaron Greenhouse and Jonathan Aldrich for providing helpful feedback on early drafts. I thank Yang Zhao, Bill Retert and Mohamed ElBendary for frequent conversations on the topic. I thank the many anonymous reviewers for their comments.

SDG

ABOUT THE AUTHORS

John Boyland is an Associate Professor in the Department of EE & Computer Science at the University of Wisconsin–Milwaukee. He can be reached at boyland@cs.uwm.edu. See also http://www.cs.uwm.edu/faculty/boyland.