Science of Computer Programming 112 (2015) 227–260

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Cooperative types for controlling thread interference in Java

Jaeheon Yi [a,1], Tim Disney [a], Stephen N. Freund [b,*], Cormac Flanagan [a]

[a] *UC Santa Cruz, Santa Cruz, CA, USA*
[b] *Williams College, Williamstown, MA, USA*

ARTICLE INFO

Article history:
Received 20 June 2014
Received in revised form 3 August 2015
Accepted 4 August 2015
Available online 15 August 2015

Keywords:
Cooperability
Concurrency
Type systems

ABSTRACT

Multithreaded programs are notoriously prone to unintended interference between concurrent threads. To address this problem, we argue that yield annotations in the source code should document all thread interference, and we present a type system for verifying the absence of undocumented interference in Java programs. Under this type system, well-typed programs behave as if context switches occur only at yield annotations. Thus, well-typed programs can be understood using intuitive sequential reasoning, except where yield annotations remind the programmer to account for thread interference.

Experimental results show that yield annotations describe thread interference more precisely than prior techniques based on method-level atomicity specifications. In particular, yield annotations reduce the number of interference points one must reason about by an order of magnitude. The type system is also more precise than prior methods targeting race freedom, and yield annotations highlight all known concurrency defects in our benchmarks.

The type system reasons about program behavior modularly via method-level specifications. To alleviate the programmer burden of writing these specifications, we extend our system to support method specification inference, which makes our technique more practical for large code bases.

© 2015 Elsevier B.V. All rights reserved.

1. Controlling interference

Developing reliable multithreaded software is challenging due to the potential for nondeterministic interference between concurrent threads. Programmers use synchronization idioms such as locks to control thread interference but unfortunately do not typically document where interference may still occur in the source code. Consequently, every programming task (such as a feature extension, code review, etc.) may require the programmer to manually reconstruct the actual interference points by analyzing the synchronization idioms in the source code. This approach is problematic since manual reconstruction of interference points is tedious and error-prone. Moreover, interference points are fairly sparse in practice, and consequently programmers often simply assume that code is free of interference — a shortcut that may result in scheduler-dependent defects, which are notoriously difficult to detect or eliminate.

We propose to use yield annotations to document the actual interference points in the source code, thereby avoiding the need to manually infer interference points during program maintenance. Moreover, yield annotations enable us to decompose the hard problem of multithreaded program correctness into two simpler correctness requirements:

* Corresponding author at: 47 Lab Campus Drive, Williams College, Williamstown, MA 01267, USA. Tel.: +413 597 4260.
E-mail address: freund@cs.williams.edu (S.N. Freund).
[1] Current address: Google, Mountain View, CA, USA.

http://dx.doi.org/10.1016/j.scico.2015.08.004
0167-6423/© 2015 Elsevier B.V. All rights reserved.

```
1  class TSP {
2    final Object lock;
3    volatile int shortestPathLength;
4
5    compound void searchFrom(Path path) {
6      if (path.length >= this..shortestPathLength)
7        return;
8
9      if (path.isComplete()) {
10       ..synchronized (this.lock) {
11         if (path.length < this.shortestPathLength)
12           this.shortestPathLength = path.length;
13       }
14     } else {
15       for (Path c: path.children())
16         this..searchFrom#(c);
17     }
18   }
19 }
```

Fig. 1. Traveling Salesperson Algorithm.

Cooperative-preemptive equivalence: Does the program exhibit the same behavior under a *cooperative scheduler* (that performs context switches only at yield annotations) as it would under a traditional *preemptive scheduler* (that performs context switches at arbitrary program points)?

Cooperative correctness: Is the program correct when run under a cooperative scheduler?

In this paper, we present a static type and effect system for Java that verifies the correctness property of cooperative-preemptive equivalence, which indicates that all thread interference is documented with yield annotations. The type system checks that the instructions of each thread consist of a sequence of *transactions* separated by yield annotations, where each transaction is serializable according to Lipton's theory of reduction [33]. Consequently, any preemptive execution of a well-typed program is guaranteed to behave *as if* executing under a cooperative scheduler, where context switches happen only at explicit yield annotations.

Cooperative scheduling provides an appealing concurrency semantics with the following properties:

1. *Thread interference is always highlighted with yields.* Yield annotations remind the programmer to allow for the effects of interleaved concurrent threads. In addition, unintended interference does not go unnoticed because our type system will reject programs with missing yields.
2. *Sequential reasoning is correct by default for code fragments with no yield annotations.* A programmer may safely ignore concurrency when reasoning about yield-free fragments, which can drastically simplify correctness arguments. Moreover, any automatic program analysis for sequential code may be applied to those regions, including, for example, the many verification techniques based on axiomatic semantics, such as ESC/Java [20], SLAM [5], Blast [6], separation logic [40,46], and many others. In contrast, previous systems based on identifying atomic methods provide little assistance in reasoning about non-atomic code fragments since they do not highlight exactly where in those fragments interference may occur [18,25,54].

A previous user study showed that these properties provide a distinct benefit to the programmer [49]. Specifically, the presence of yield annotations produces a statistically significant improvement in the ability of programmers to identify concurrency defects during code inspection.

We have developed a prototype implementation, called the Java Cooperability Checker (JCC), of our type system for verifying cooperative-preemptive equivalence. Experimental results on a range of Java benchmarks show that JCC requires yield annotations on only about 16 interference points (IPs) per KLOC. In comparison, previous non-interference specifications generate significantly more interference points: 134 IP/KLOC using atomic methods specifications; 50 IP/KLOC when reasoning about data races; and 25 IP/KLOC when reasoning about both data races and atomic methods.

1.1. Example

To illustrate the benefits of explicit yield annotations, consider the traveling salesperson algorithm shown in Fig. 1. The `TSP` class contains a method `searchFrom` that recursively searches through all extensions of a particular `path`, aborting the search whenever `path.length` becomes greater than `shortestPathLength` (the length of the shortest complete path found so far). As we describe below, the "`..`" notation indicates yield points, and the `compound` and `#` annotations mark the declaration and call of a method containing yield points, respectively.

To exploit multicore processors, multiple threads may concurrently invoke `searchFrom` on the same `TSP` object. Thus the variable `shortestPathLength` is protected by `lock` for all writes, but reads are permitted without holding the lock to maximize performance. Consequently, the accesses of `shortestPathLength` on lines 6 and 12 may experience a data race. (A data race occurs when there are two or more accesses to a memory location, at least one of which is a write, that are not ordered by synchronization.) In contrast, the read on line 11 is race-free, since `lock` is held at that point. The `shortestPathLength` field is declared volatile to avoid non-sequentially consistent racy reads under the relaxed semantics of the Java Memory Model. Correctness of this algorithm relies on `shortestPathLength` only decreasing in value.

The code uses the notation "`..`" as a lightweight syntax for yield annotations. The "`..`" on line 6 indicates that interference may occur before the read of `shortestPathLength` on line 6 (because of concurrent writes). Similarly, thread interference may occur before the lock acquire at line 10. Effects of other threads may become visible at these points. For example, prior to acquiring the lock on line 10, another thread could acquire that lock and modify the `shortestPathLength` variable. Documenting that possibility indicates to the programmer that the value previously read from `shortestPathLength` may be stale by the time the lock is acquired. In contrast, the potentially racy write to the `shortestPathLength` on line 12 does not require a preceding yield, since our analysis can verify that observable interference is not possible at that point.

To facilitate modular reasoning, the method `searchFrom` is declared `compound`, meaning that it contains internal yield points. The method call on line 16 is highlighted with a postfix "#" to indicate that the callee is `compound`, and so invariants over shared state that hold before the call may not hold after the call returns. That call is also a yielding call (as indicated by the notation "`..`") because interference from other threads may become visible in between consecutive calls to that method at run time.

This example demonstrates the two key benefits of cooperative reasoning mentioned above:

1. *Thread interference is always highlighted with yields.* This program contains yield annotations exactly where interference is possible. If interference may occur at an unintended and thus unannotated point due to an oversight in design or a programming mistake, our type system will reject that program. Thus, concurrency errors manifest early in the development process as easily recognizable places requiring "unexpected" yield annotations.
2. *Sequential reasoning is correct by default for code fragments with no yield annotations.* Once the type system has verified the TSP class is well-typed, the programmer (or any automatic analysis tool) can use sequential reasoning to verify additional correctness properties of yield-free code fragments. In particular, the absence of any yields in the following test-and-set fragment from `searchFrom` enables a straight forward proof that `shortestPathLength` is decreasing:

```
11    if (path.length < this.shortestPathLength)
12      this.shortestPathLength = path.length;
```

This work uses the yield annotation syntax "`..`" in preference to the "`yield`" statements of prior work [3,4,8,28,58,59] to more precisely characterize *why* interference may occur. For example, the annotation at line 6 of Fig. 1 precisely documents that interference occurs on the read of `shortestPathLength`. In comparison, in the following code the `yield` statement suggests that one of the subsequent reads of `path.length` or `shortestPathLength` is interfering or racy, but it is not immediately obvious which one.

```
compound void searchFrom(Path path) {
  yield;
  if (path.length >= shortestPathLength)
  ...
}
```

Comparison to other approaches Several prior techniques have been proposed for controlling thread interference. Fig. 2 summarizes the most closely related proposals, divided along two orthogonal dimensions: whether transactions are analyzed or enforced, and whether one specifies transactions (atomic blocks) or transactional boundaries (yields):

Atomicity: A method or code block is *atomic* if it does not contain any thread interference points [27,35,34,13]. Previous studies have shown that as many as 90% of methods are typically atomic [15].
Unfortunately, the notion of atomicity is rather awkward for specifying interference in non-atomic methods like `searchFrom`. Some particular code blocks in `searchFrom`, such as the synchronized block, can be marked atomic, as shown in Fig. 3. The result is rather verbose and inadequate, however, since atomicity focuses on delimiting blocks where interference does not occur but still suggests that interference can occur everywhere outside those blocks, such as on the access to `path` on line 18.
Moreover, the use of atomic blocks requires a *bimodal* reasoning style that combines sequential reasoning inside atomic blocks with preemptive reasoning (pervasive interference) outside atomic blocks. The programmer is responsible for choosing the appropriate reasoning mode for each piece of code according to whether it is inside or

	Transaction	Yield
Compile-time analysis	Atomicity	**Cooperability**
Run-time enforcement	Transactional memory	Automatic mutual exclusion

Fig. 2. Alternatives to preemptive semantics: one may analyze or enforce either transactions or yields (transactional boundaries).

```
1  Object lock;
2  volatile int shortestPathLength;
3
4  compound void searchFrom(Path path) {
5    atomic {
6      if (path.length >= shortestPathLength)
7        return;
8    }
9
10   if (path.isComplete()) {
11     atomic {
12       synchronized (lock) {
13         if (path.length < shortestPathLength)
14           shortestPathLength = path.length;
15       }
16     }
17   } else {
18     for (Path c: path.children())
19       searchFrom#(c);
20   }
21 }
```

Fig. 3. Traveling Salesperson Algorithm with atomic blocks.

outside an atomic block, with obvious room for error. In contrast, cooperative reasoning is uniformly applicable to all code under our proposed methodology.

Transactional memory: Transactional memory offers an *enforcement* model of execution for transactions, where the run-time system guarantees that atomic blocks satisfy serializability via software or hardware mechanisms [12,24,31,45,47,51,60]. However, finding a robust semantics for transactional memory has proven elusive, while performance issues hold back widespread adoption.

Automatic mutual exclusion: Automatic mutual exclusion (AME) inverts transactional memory by executing all code in a single transaction, unless otherwise specified [28]. We are inspired by AME's feature of safety by default but focus on static analysis rather than run-time enforcement, since enforcement mechanisms may not be appropriate for legacy programs. Current AME implementations leverage transactional memory implementations, with the problems listed above. Observationally Cooperative Multithreading [52] is similar in spirit to AME but supports both lock-based and transactional implementations.

Section 9 contains a more detailed comparison with related work.

1.2. Contributions

This paper presents a type system and implementation for checking yield annotations for the Java programming language. Our experimental results validate that cooperability is an effective way to capture and reason about thread interference, and we show that the use of yield annotations dramatically reduces the number of interference points that must be considered by the programmer. We augment our type system with a method specification inference algorithm to synthesize the method specifications used by our checker. While the programmer must still annotate yield points, method specification inference drastically reduces the annotation overhead of our approach.

Expression Form	Non-yielding	Yielding
Field Read	$e\,.\,f$	$e\,.\,.\,f$
Field Write	$e\,.\,f = e$	$e\,.\,.\,f = e$
Atomic Method Call	$e\,.\,m\,(\bar{e})$	$e\,.\,.m\,(\bar{e})$
Non-Atomic Method Call	$e\,.\,m\#\,(\bar{e})$	$e\,.\,.m\#\,(\bar{e})$
Lock Synchronization	`synchronized(l) { ... }`	`..synchronized(l) { ... }`

Fig. 4. Non-yield and Yielding Expression Forms.

In summary, this work makes the following contributions:

- We demonstrate how documenting interference to ensure cooperability supports reasoning about thread interactions.
- We present a type system for verifying lightweight non-interference specifications in multithreaded software. We show that this type system satisfies the standard preservation property and ensures that well-typed programs are cooperative-preemptive equivalent.
- We present an inference algorithm to infer the method specifications used by the type system.
- We describe an implementation for Java.
- For our benchmarks, the type system identified just 16 interference points (IP) per KLOC, a significant improvement over prior non-interference specifications based on atomic methods (134 IP/KLOC), data race freedom (50 IP/KLOC), or a combination of atomic methods and data race freedom (26 IP/KLOC). In addition, all known concurrency bugs in these programs were highlighted by yield annotations.
- Although our technique does require programmer-supplied annotations, we have not found adding them to be a prohibitive burden. The type system only required roughly 35 annotations/KLOC to check these programs, and our method specification inference algorithm reduced that requirement to 21 annotations/KLOC.

Section 2 illustrates how cooperability facilitates programming. Sections 3 and 4 present our type system. Sections 5 and 6 presents JCC, our implementation for Java, and reports on our experience using it to check programs. Sections 7 and 8 then extend our formal system and JCC to infer method specifications and show how inference reduces the annotation burden for programmers. Sections 9 and 10 discuss related work and conclude.

This work extends earlier work [58,59,57] and includes a more thorough discussion of cooperability and its benefits: a formal development of the run-time semantics and correctness proof showing that well-typed programs exhibit cooperative-preemptive equivalence, as well as a new method specification inference algorithm to drastically reduce the annotation-overhead of using our checker.

2. Documenting interference

In a cooperative program, each thread should execute a sequence of serializable transactions separated by yields. A programmer must therefore identify yield points using the annotations described below, which we summarize in Fig. 4.

2.1. Yielding field accesses

The following syntax denotes a yielding read or write of a racy field f, where the yield (and hence potential interference) occurs just before the access to f:

```
e..f        // yield before racy read
e..f = e'   // yield before racy write
```

The yielding read $e\,.\,.\,f$ evaluates the subexpression e to an object reference and then reads the f field of the referenced object, and the yield annotation indicates that interference is possible after evaluating e but before reading the field. Typically, f will be a volatile field to avoid the additional memory model complexities of data races on non-volatile fields.

Annotations of this form are used when a program manipulates a shared memory location that may have data races. For example, if a program uses a global counter stored in the volatile field `p.x`, then incrementing and printing that counter would need to be written as

```
p..x = p.x + 1;
print(p..x);
```

since the write is not in the same serializable transaction as the first read. Similarly, the second read must yield because the value written may be stale due to interference after the write.

2.2. Yielding lock acquires

A programmer may also document that interference may occur prior to acquiring a lock, as in

```
..synchronized (e) { ... }
```

```
1  class Bit {
2
3    boolean b;
4
5    this?mover:atomic boolean get() {
6      synchronized(this) {
7        return this.b;
8      }
9    }
10
11   this?mover:atomic void set(boolean t) {
12     synchronized(this) {
13       this.b = t;
14     }
15   }
16
17   atomic boolean testAndSet() {
18     synchronized(this) {
19       if (!this.get()) {
20         this.set(true);
21         return true;
22       } else {
23         return false;
24       }
25     }
26   }
27
28   compound void busyWait() {
29     while (!this..testAndSet()) { }
30   }
31 }
```

Fig. 5. A YieldJava class implementing a shared boolean flag.

which is a synchronized block that may encounter interference after the evaluation of the lock expression e, but before the lock acquire. Note that a lock release cannot interfere with any concurrently executing action by another thread and thus never needs to start a new transaction.

2.3. Yielding method calls

Next, consider an atomic method `m()` whose body performs a complete transaction. In order to sequentially compose two calls to such methods, a yield point must occur in between them. For this purpose, we also allow yield annotations on a method call, where the yield occurs after e and e' have been evaluated and right before the call is performed:

```
e..m(e')        // yield before method call
```

To illustrate yielding calls, consider the `Bit` class in Fig. 5, which implements a thread-safe shared flag. While elementary in design, it illustrates more complex shared structures. The `get()` and `set()` methods each perform a single transaction when called without holding `Bit`'s lock. (These methods, and the others, include cooperability effect annotations, which we discuss below.)

A client using a `Bit` object `flag` would thus need to insert a yield point between consecutive calls to `get` and `set`, as shown on the left below. If the client synchronizes on `flag`'s lock around these operations, no yield is required, as shown on the right below. Indeed, this is how `testAndSet()` is implemented in order to ensure that it is an atomic operation with no internal yields.

```
if (!flag.get()) {
  flag..set(true);
}
```

```
synchronized(flag) {
  if (!flag.get()) {
    flag.set(true);
  }
}
```

A second example of invoking a yielding method is the `busyWait()` method in Fig. 5, which repeatedly invokes the atomic `testAndSet()` method and documents that interference may occur between consecutive calls to that method.

2.4. Non-atomic method calls

A method like busyWait() is non-atomic, or compound, because its body consists of multiple transactions separated by yield points. In that case, we require calls to busyWait() to use the postfix annotation "#":

```
flag.busyWait#()      // yields inside busyWait
```

This call-site annotation reminds the programmer that state properties holding before the call may not hold after the call, due to yields nested within the callee. For example, in the following, the assertion that a race-free global variable x contains the value written immediately before the call to busyWait may fail due to interference during that call:

```
x = 10;
flag.busyWait#();
assert x == 10;
```

2.5. Method cooperability specifications

To enable modular reasoning about cooperability, we use specifications on methods to describe their cooperability behavior. These could be as simple as atomic, which indicates a method is a single transaction with no internal yield points, or compound, which indicates a method consists of a series of yield-separated transactions. A method could also be annotated as a mover, which indicates that the method does not interfere with concurrent operations of other threads.[2]

Specifications for methods can also describe behavior conditional on which locks are held at call sites. For example, the specification this?mover:atomic for get() indicates that the method executes a single transaction that either is a mover (if invoked when the receiver's lock is held) or is atomic (if invoked when the receiver's lock is not held). Conditional forms are essential for expressiveness when reasoning about code with reentrant locks. The set() method has a similar specification.

While conditional effects could be assigned to testAndSet() and busyWait(), we have instead annotated them with the more coarse, but still sound, specifications of atomic and compound to indicate that they consist of a single transaction and a sequence of transactions, respectively. In addition to illustrating the ability to coarsen effects, these coarser specifications also better capture the intended use case in which testAndSet() and busyWait() are called from client code without the receiver lock held.

3. Effects for cooperability

As we outline in the following sections, we reason about cooperability via a type and effect system characterizing the behavior of each program subexpression. We use two kinds of effects to capture an expression's behavior: *mover effects* and *atomicity effects*.

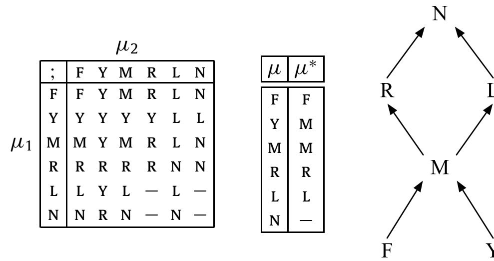
3.1. Mover effects

A *mover effect* μ characterizes the behavior of a program expression in terms of how operations of that expression commute with operations of other threads:

$$\mu \ ::= \ \text{F} \mid \text{Y} \mid \text{M} \mid \text{R} \mid \text{L} \mid \text{N}$$

- F: The effect F (for functional) describes expressions whose result does not depend on any mutable state. Hence, re-evaluating a functional expression is guaranteed to produce the same result. Examples of functional expressions include numeric constants, accesses to immutable local variables, accesses to a final field of an immutable object reference, and so on. (Our type system requires all lock names to be functional to ensure that it does not confuse distinct locks.)
- Y: The effect Y describes yield operations, denoted as "..". These expressions mark transactional boundaries where the current transaction ends and a new one starts.
- R: The effect R describes *right-mover* expressions such as lock acquires. In more detail, suppose a trace contains an acquire operation *a* that is immediately followed by an operation *b* of a different thread. Then the two operations *a* and *b* *commute* and can be swapped without changing the overall behavior or final state of the trace, as illustrated by the commuting diagram below.

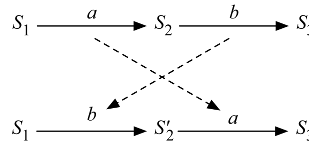
[2] The term mover is comes from Lipton's theory of reduction, as described in the next section.



;	F	Y	M	R	L	N
F	F	Y	M	R	L	N
Y	Y	Y	Y	Y	L	L
M	M	Y	M	R	L	N
R	R	R	R	R	N	N
L	L	Y	L	—	L	—
N	N	R	N	—	N	—

μ	μ^*
F	F
Y	M
M	M
R	R
L	L
N	—

Fig. 6. Iterative closure (μ^*) and sequential composition ($\mu_1; \mu_2$) operations for mover effects, and the mover effect lattice.



This commuting relationship holds because once operation a acquires the lock, operation b can neither acquire nor release that lock.

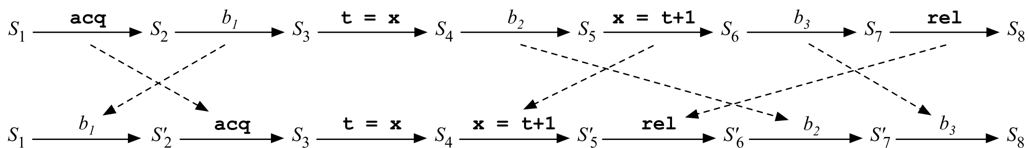
- L: The effect L describes left-mover expressions such as lock releases that commute to the left across a preceding operation of a different thread (such as operation b in the above diagram). Thus, if b in the above diagram is a lock release operation, then operation a can neither acquire nor release the lock. Hence the lock release operation can be moved to the left of a without changing the resulting state.
- M: The effect M describes both-mover expressions such as race-free variable accesses that commute both left and right with operations by other threads. If operation a above were a race-free access, operation b could not be a conflicting access to that same variable, and so a can commute to the right. Similarly, if b were a race-free access, it could commute to the left since a could not be accessing the same variable.
- N: The effect N describes non-mover code that may perform a racy access or contain right-movers followed by left-movers (as in a synchronized block).

Next, suppose the sequence of instructions α performed by a thread consists of: (1) zero or more right-movers, followed by (2) at most one non-mover, followed by (3) zero or more left-movers. Then, according to Lipton's theory of reduction [33], instructions of other threads that are interleaved into α can be commuted out so that α executes serially, without interleaved operations of other threads. We say that any sequence matching this reducible pattern of instructions is a *reducible transaction*, or simply a *transaction*.

To illustrate this notion, consider the following method:

```
synchronized void inc() {
  int t = x;
  x = t + 1;
}
```

This method acquires the lock on `this` (the operation `acq` in the first execution trace below), reads a variable `x` protected by that lock into a local variable `t` (`t=x`), updates that variable (`x=t+1`), and then releases the lock (`rel`). The diagram below shows the actions of this method interleaved with arbitrary actions b_1, b_2, and b_3 of other threads. Because the acquire operation is a right mover and the write and release operations are left movers, there exists an equivalent serial execution where the operations of the method are not interleaved with operations of other threads:



The type system presented in the next section ensures that the instructions performed by each thread consist of reducible transactions separated by yield annotations. The type system works by composing the effects for individual operations according to the sequential composition ($\mu_1; \mu_2$) and iterative closure (μ^*) operations in Fig. 6.

Given two operations e_1 and e_2 with mover effects μ_1 and μ_2, the effect of executing $e_1; e_2$ is $\mu_1; \mu_2$. For example, executing a right-mover and then a left-mover results in R; L = N. Sequential composition is partial and may fail if the composed effect is not reducible (indicated with a "—"). For example, the sequential composition (L; R) is undefined because

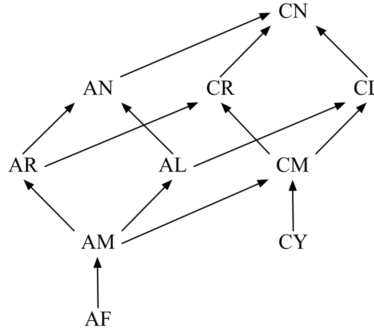


Fig. 7. Combined effect lattice.

it does not match the reducible pattern from above and therefore does not represent the effect of any valid sequence of transactions.

Note that effect F is idempotent for sequential composition because functional expressions have no visible side effects and cannot impact the commuting behavior of larger program fragments containing them.

Composition with Y is defined to capture how yielding may be used. For example, "Y; N" is L, to ensure that "N; Y; N" is permissible.

Mover effects are ordered by the relation $\sqsubseteq$ captured by the lattice in Fig. 6. Given an expression e with mover effect μ, the effect of the repeatedly executing e zero or more times is $\mu^* = \text{F} \sqcup \mu \sqcup \mu;\mu \sqcup \mu;\mu;\mu \sqcup \ldots$ where $\sqcup$ is the join operation for that lattice and F is the effect of the empty sequence.

3.2. Atomicity effects

Each program expression also has an *atomicity effect*

$$\tau \quad ::= \quad \text{A} \mid \text{C}$$

that is either A (for atomic) if the expression never yields or C (for compound) if the expression may yield. Ordering ($\sqsubseteq$), iterative closure (τ^*) and sequential composition ($\tau_1; \tau_2$) for atomicity effects are defined by:

$$\text{A} \sqsubset \text{C} \qquad \tau^* \stackrel{\text{def}}{=} \tau \qquad \tau_1; \tau_2 \stackrel{\text{def}}{=} \tau_1 \sqcup \tau_2$$

3.3. Combined effects

A *combined effect* κ is a pair $\tau\mu$ of an atomicity effect τ and a mover effect μ. Note that not all combined effects are meaningful. In particular, AY and CF are contradictory: an atomic piece of code may not contain a yield, and code with yields cannot be considered functional. We define the ordering relation and the join, iterative closure, and sequential composition operations on combined effects in a point-wise manner, and the diagram in Fig. 7 summarizes the resulting lattice of combined effects.

$$\begin{aligned}
\kappa \;&::=\; \tau\,\mu \\
\tau_1\,\mu_1 \sqsubseteq \tau_2\,\mu_2 \;&\text{ iff }\; \tau_1 \sqsubseteq \tau_2 \text{ and } \mu_1 \sqsubseteq \mu_2 \\
\tau_1\,\mu_1 \sqcup \tau_2\,\mu_2 \;&\stackrel{\text{def}}{=}\; \tau_3\,\mu_3 \text{ where } \tau_3 = \tau_1 \sqcup \tau_2, \mu_3 = \mu_1 \sqcup \mu_2 \\
\tau_1\,\mu_1\,;\,\tau_2\,\mu_2 \;&\stackrel{\text{def}}{=}\; \tau_3\,\mu_3 \text{ where } \tau_3 = \tau_1; \tau_2, \mu_3 = \mu_1; \mu_2 \\
(\tau\,\mu)^* \;&\stackrel{\text{def}}{=}\; \tau^*\mu^*
\end{aligned}$$

We note that when `atomic` is used in a source-level method specification, as in Fig. 5, it refers to the combined effect AN. Thus composing two `atomic` method calls does not result in a reducible transaction because AN; AN is not defined. The `mover` source-level annotation refers to AM, and `compound` refers to CN. While these are the most common explicitly mentioned in method specification effects, our Java implementation supports source-level syntax for all combined effects, as shown in Section 5.

3.4. Conditional effects

As outlined above, the effect of acquiring a lock `m` is AR, since an acquire is a right-mover containing no yields. If the lock `m` is already held by the current thread, however, then the re-entrant lock acquire is actually a no-op and is more precisely characterized as an atomic both-mover AM.

$$
\begin{array}{rcl}
P \in \mathit{Program} & = & \overline{\mathit{defn}} \\
\mathit{defn} \in \mathit{Definition} & ::= & \texttt{class}\ c\ \texttt{\{}\ \overline{\mathit{field}}\ \overline{\mathit{meth}}\ \texttt{\}} \\
\mathit{field} \in \mathit{Field} & ::= & c\ f \\
\mathit{meth} \in \mathit{Method} & ::= & a\ c\ m\ \texttt{(}\overline{c\ x}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} \\
f \in \mathit{FieldName} & = & \mathit{Normal} \cup \mathit{Final} \cup \mathit{Volatile} \\
e, \ell \in \mathit{Expr} & ::= & x \mid \texttt{null} \\
& & \mid e_\gamma f \mid e_\gamma f = e \\
& & \mid e_\gamma m\,\texttt{(}\overline{e}\texttt{)} \mid e_\gamma m\texttt{\#}\,\texttt{(}\overline{e}\texttt{)} \\
& & \mid \texttt{new}\ c\,\texttt{(}\overline{e}\texttt{)} \mid e_\gamma\,\texttt{sync}\ e \\
& & \mid \texttt{fork}\ e \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \\
& & \mid \texttt{if}\ e\ e\ e \mid \texttt{while}\ e\ e \\
\\
\gamma \in \mathit{OptYield} & ::= & \texttt{.} \mid \texttt{..} \\
c, d \in \mathit{ClassName} & & \\
x, y \in \mathit{Var} & & \\
m \in \mathit{MethodName} & & \\
a \in \mathit{Effect} & &
\end{array}
$$

Fig. 8. YIELDJAVA syntax.

We introduce *conditional effects* (or simply, *effects*) to capture situations like this where the effect of an operation depends on which locks are held by the current thread. We use ℓ to range over expressions that are functional (F) and which always denote the same lock. An *effect* a is then either a combined effect κ or a conditional effect $\ell\,?\,a_1 : a_2$, which is equivalent to a_1 if the lock ℓ is held, and is equivalent to a_2 otherwise. We extend the iterative closure, sequential composition, and join operations to conditional effects as follows:

$$
\begin{array}{rcl}
a & ::= & \kappa \mid \ell\,?\,a_1 : a_2 \\
(\ell\,?\,a_1 : a_2) \sqcup a & = & \ell\,?\,(a_1 \sqcup a) : (a_2 \sqcup a) \\
a \sqcup (\ell\,?\,a_1 : a_2) & = & \ell\,?\,(a \sqcup a_1) : (a \sqcup a_2) \\
(\ell\,?\,a_1 : a_2)^* & = & \ell\,?\,a_1^* : a_2^* \\
(\ell\,?\,a_1 : a_2); a & = & \ell\,?\,(a_1; a) : (a_2; a) \\
a; (\ell\,?\,a_1 : a_2) & = & \ell\,?\,(a; a_1) : (a; a_2)
\end{array}
$$

We extend the effect ordering to conditional effects in a manner similar to earlier type systems for atomicity [18]. To decide $a_1 \sqsubseteq a_2$, we use an auxiliary relation $\sqsubseteq_n^h$, where h is a set of locks known to be *h*eld by the current thread, and n is a set of locks known *n*ot to be held by the current thread. We define $a_1 \sqsubseteq a_2$ to be $a_1 \sqsubseteq_\emptyset^\emptyset a_2$ and check $a_1 \sqsubseteq_n^h a_2$ recursively as follows:

$$
\frac{\kappa_1 \sqsubseteq \kappa_2}{\kappa_1 \sqsubseteq_n^h \kappa_2}
\qquad
\frac{\begin{array}{c}\ell \notin n \Rightarrow a_1 \sqsubseteq_n^{h\cup\{l\}} a \\ \ell \notin h \Rightarrow a_2 \sqsubseteq_{n\cup\{l\}}^h a\end{array}}{\ell\,?\,a_1 : a_2 \sqsubseteq_n^h a}
\qquad
\frac{\begin{array}{c}\ell \notin n \Rightarrow \kappa \sqsubseteq_n^{h\cup\{l\}} a_1 \\ \ell \notin h \Rightarrow \kappa \sqsubseteq_{n\cup\{l\}}^h a_2\end{array}}{\kappa \sqsubseteq_n^h \ell\,?\,a_1 : a_2}
$$

These rules allow us to order conditional effects in ways illustrated by the following examples:

$$
\begin{array}{rcl}
\ell\,?\,\mathrm{AM} : \mathrm{CN} & \sqsubseteq & \mathrm{CN} \\
\mathrm{AM} & \sqsubseteq & \ell\,?\,\mathrm{AM} : (\ell\,?\,\mathrm{AF} : \mathrm{AM}) \\
\ell\,?\,(\ell\,?\,\mathrm{AM} : \mathrm{CM}) : \mathrm{AM} & \sqsubseteq & \mathrm{AM}
\end{array}
$$

4. The language YIELDJAVA

Now that our language of conditional effects has been introduced, we incorporate those effects into a type system for the idealized language YIELDJAVA, a multithreaded subset of Java with yield annotations. In this section, we present the YIELDJAVA syntax, a type system to verify cooperability, and the theorems showing that our type system accepts only cooperable programs.

4.1. Syntax

The YIELDJAVA syntax is summarized in Fig. 8. A program P is a sequence of class definitions. Each definition associates a name c with a collection of field and method declarations. A field declaration includes a type and a name. Field names are syntactically divided into three categories:

- *Normal* fields are mutable and are assumed to be race-free.
- *Final* fields are immutable and thus race-free.

- *Volatile* fields are mutable and may have races.

We assume that data race freedom for *Normal* fields is verified separately by, for example, a race-free type system [9,23,1]. Normal fields include thread-local fields as well as thread-shared fields that are correctly synchronized, for example, via locks. (Our implementation does support racy accesses to non-volatile fields, as described in Section 5.) In contrast to accesses to normal fields, accesses to volatile fields do not commute with steps from other threads. As in Java, we assume accesses to *Volatile* fields are synchronizing operations and exhibit sequentially consistent behavior.

Each method declaration $a\ c\ m\,(\overline{c\ x})\ \{\ e\ \}$ defines a method m with return type c that takes parameters $\overline{x}$ of type $\overline{c}$. The method declaration also includes an effect a, as described in Section 3. Method parameters and local variables, drawn from the set *Vars*, are immutable.

Several expression forms support optional yield annotations, as discussed previously in Section 2. For example, a field read expression $e_\gamma f$ comes in two forms, $e\,.\,f$ and $e\,.\,.\,f$, depending on the optional yield annotation γ for documenting thread interference. Field writes $e_\gamma f = e$ and method calls $e_\gamma m\,(\overline{e})$ also include an optional yield, and calls to `compound` or non-atomic methods must be marked as $e_\gamma m\#\,(\overline{e})$. YieldJava supports a synchronized block $e_{1\gamma}\,\texttt{sync}\;e_2$ that is analogous to Java's synchronized statement

```
synchronized (e1) { e2 }
```

That operation also supports an optional yield. (We use this alternative syntax in the formal development for symmetry with the other optionally yielding operations.)

Finally, an object allocation `new` $c\,(\overline{e})$ creates a new object of type c and initializes its fields to the values of the expression sequence $\overline{e}$. We assume that all programs include an empty class `Unit`. The language includes the special constant `null`, which has any class type, including `Unit`.

Our code examples include minor extensions to YieldJava, such as `int` and `boolean` types with supporting constants and operations, an explicit `return` keyword, braces to delineate blocks of code, sequential composition via ';', and so on. These extensions pose no technical challenges, but including them in the formal development would introduce additional complexity orthogonal to cooperability.

4.2. Type system

The YieldJava type system ensures that each thread consists of reducible transactions separated by yields, and consequently that thread interference is observable only at yield annotations. The core of the type system is a set of rules for reasoning about the effect of an expression, as captured by the judgment:

$$P; E \vdash e : c \cdot a$$

Here, e is an expression of type c, and a is an effect describing the behavior of e. The program P is included to provide access to class declarations, and the environment E maps free variables in e to their types:

$$E ::= \epsilon \mid E, c\ x$$

Figs. 9 and 10 present the complete set of rules for expressions, as well as auxiliary rules to reason about methods, classes, effects, and so on. We describe the most important rules below:

[EXP VAR] All variables are immutable in YieldJava and thus have cooperability effect AF, since a variable access is atomic and evaluates to a constant value.

[EXP NEW] The object allocation rule first retrieves the definition of the class c from P and ensures the arguments $e_{1..n}$ match the field types from c. The effect of the whole expression is the composition of effects of evaluating $e_{1..n}$ composed with the effect AM, reflecting that `new` is not functional (since re-evaluating an object allocation would not return the same object).

[EXP REF] This rule handles a read $e.f$ of a *Normal* or *Final* field. If f is *Normal* and thus race-free, the field access has effect AM as it commutes with steps by other threads. If f is *Final*, the access has the more precise effect AF.

[EXP REF VOLATILE] A racy read $e_\gamma f$ of a volatile field may be annotated with a yield point (if γ = "..") or not (if γ = "."). We use the auxiliary function $\Gamma(\gamma)$ to map γ to the corresponding effect:

$$\begin{aligned} \Gamma &: \textit{OptYield} \rightarrow \textit{Effect} \\ \Gamma(.) &= \text{AF} \\ \Gamma(..) &= \text{CY} \end{aligned}$$

Thus, if the expression e has effect a, then the non-yielding racy access $e\,.\,f$ has effect (a; AF; AN), the sequential composition of (1) the effect a of computing the object reference, (2) the effect AF to indicate that no yield is present at the access site, and (3) the effect AN for accessing the potentially racy field f. In contrast, the yielding racy access $e\,.\,.\,f$ has effect (a; CY; AN), where the middle effect now reflects that a yield is present.

$\boxed{P; E \vdash e : c \cdot a}$

[EXP VAR]
$$\frac{P \vdash E \quad E = E_1, c\,x, E_2}{P; E \vdash x : c \cdot \text{AF}}$$

[EXP NULL]
$$\frac{P \vdash E \quad \texttt{class}\; c\; \{\ \dots\ \} \in P}{P; E \vdash \texttt{null} : c \cdot \text{AF}}$$

[EXP NEW]
$$\frac{\texttt{class}\; c\; \{\ d_i\; x_i\; ^{i\in 1..n}\ \dots\ \} \in P \quad P; E \vdash e_i : d_i \cdot a_i \quad \forall i \in 1..n}{P; E \vdash \texttt{new}\; c\,(e_{1..n}) : c \cdot (a_1; \cdots; a_n; \text{AM})}$$

[EXP REF]
$$\frac{P; E \vdash e : c \cdot a \quad \texttt{class}\; c\; \{\dots d\; f \dots\} \in P \quad (f \in \mathit{Normal} \wedge a' = \text{AM}) \vee (f \in \mathit{Final} \wedge a' = \text{AF})}{P; E \vdash e.f : d \cdot (a; a')}$$

[EXP ASSIGN]
$$\frac{P; E \vdash e : c \cdot a \quad P; E \vdash e' : d \cdot a' \quad \texttt{class}\; c\; \{\dots d\; f \dots\} \in P \quad f \in \mathit{Normal}}{P; E \vdash (e.f = e') : d \cdot (a; a'; \text{AM})}$$

[EXP REF VOLATILE]
$$\frac{P; E \vdash e : c \cdot a \quad \texttt{class}\; c\; \{\dots d\; f \dots\} \in P \quad f \in \mathit{Volatile}}{P; E \vdash e_\gamma f : d \cdot (a; \Gamma(\gamma); \text{AN})}$$

[EXP ASSIGN VOLATILE]
$$\frac{P; E \vdash e : c \cdot a \quad P; E \vdash e' : d \cdot a' \quad \texttt{class}\; c\; \{\ \dots\ d\; f\ \dots\ \} \in P \quad f \in \mathit{Volatile}}{P; E \vdash (e_\gamma f = e') : d \cdot (a; a'; \Gamma(\gamma); \text{AN})}$$

[EXP CALL ATOMIC]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \\ \texttt{class}\; c\; \{\dots \mathit{meth} \dots\} \in P \\ \mathit{meth} = a'\; c'\; m\,(d_i\; x_i\; ^{i\in 1..n})\ \{\ e'\ \} \\ P; E \vdash a'[\texttt{this} := e, x_i := e_i\; ^{i\in 1..n}] \uparrow a'' \\ P; E \vdash e_i : d_i \cdot a_i \quad \forall i \in 1..n \quad a'' \sqsubseteq \text{AN} \end{array}}{P; E \vdash e_\gamma m\,(e_{1..n}) : c' \cdot (a; a_1; \cdots; a_n; \Gamma(\gamma); a'')}$$

[EXP CALL COMPOUND]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \\ \texttt{class}\; c\; \{\dots \mathit{meth} \dots\} \in P \\ \mathit{meth} = a'\; c'\; m\,(d_i\; x_i\; ^{i\in 1..n})\ \{\ e'\ \} \\ P; E \vdash a'[\texttt{this} := e, x_i := e_i\; ^{i\in 1..n}] \uparrow a'' \\ P; E \vdash e_i : d_i \cdot a_i \quad \forall i \in 1..n \end{array}}{P; E \vdash e_\gamma m\#\,(e_{1..n}) : c' \cdot (a; a_1; \cdots; a_n; \Gamma(\gamma); a'')}$$

[EXP SYNC]
$$\frac{P; E \vdash_{\text{lock}} \ell \quad P; E \vdash e : c \cdot a}{P; E \vdash \ell_\gamma\, \texttt{sync}\; e : c \cdot S(\ell, \gamma, a)}$$

[EXP FORK]
$$\frac{P; E \vdash e : c \cdot a}{P; E \vdash \texttt{fork}\; e : \texttt{Unit} \cdot \text{AL}}$$

[EXP WHILE]
$$\frac{P; E \vdash e_i : c_i \cdot a_i \quad \forall i \in 1..2 \quad a' = a_1; (a_2; a_1)^*}{P; E \vdash \texttt{while}\; e_1\; e_2 : \texttt{Unit} \cdot a'}$$

[EXP IF]
$$\frac{P; E \vdash e_1 : d \cdot a_1 \quad P; E \vdash e_i : c \cdot a_i \quad \forall i \in 2..3 \quad a' = a_1; (a_2 \sqcup a_3)}{P; E \vdash \texttt{if}\; e_1\; e_2\; e_3 : c \cdot a'}$$

[EXP LET]
$$\frac{P; E \vdash e_1 : c_1 \cdot a_1 \quad P; E, c_1\; x \vdash e_2 : c_2 \cdot a_2 \quad P; E \vdash a_2[x := e_1] \uparrow a_2'}{P; E \vdash \texttt{let}\; x\ =\ e_1\; \texttt{in}\; e_2 : c_2 \cdot (a_1; a_2')}$$

Fig. 9. YieldJava type rules (I).

[EXP SYNC] The rule for the synchronized statement $\ell_\gamma\,\texttt{sync}\; e$ first checks that ℓ is a valid lock expression ($P; E \vdash_{\text{lock}} \ell$), meaning that ℓ must have effect AF to guarantee that it always denotes the same lock at run time (when evaluated in the same environment).

The rule then computes the effect $S(\ell, \gamma, a)$, where a is the effect of e, and γ specifies whether there is a yield point. The function S is defined as:

a	$S(\ell, \gamma, a)$
κ	$\ell\,?\,\kappa : (\Gamma(\gamma); \text{AR}; \kappa; \text{AL})$
$\ell\,?\,a_1 : a_2$	$S(\ell, \gamma, a_1)$
$\ell'\,?\,a_1 : a_2$	$\ell'\,?\,S(\ell, \gamma, a_1) : S(\ell, \gamma, a_2)$ if $\ell \neq \ell'$

If the synchronized body e has a basic effect κ and the lock ℓ is already held, then the synchronized statement also has effect κ, since the acquire and release operations are no-ops. Note that in this case the yield operation is ignored, since it is unnecessary.

If e has effect κ and the lock is not already held, then the synchronized statement has effect $(\Gamma(\gamma); \text{AR}; \kappa; \text{AL})$, since the execution consists of a possible yield point, followed by a right-mover (the acquire), followed by κ (the body), followed by a left-mover (the release).

If e has conditional effect $\ell\,?\,a_1 : a_2$, where ℓ is the lock being acquired by this synchronized statement, then we ignore a_2 and recursively apply S to a_1, since ℓ is held within e.

Finally, if e has an effect that is conditional on some other lock ℓ', then we recursively apply S to both branches.

$\boxed{P \vdash E}$

[ENV ϵ]
$$\frac{}{P \vdash \epsilon}$$

[ENV X]
$$\frac{P \vdash E \quad x \notin dom(E) \qquad \texttt{class}\ c\ \{\ \ldots\} \in P}{P \vdash (E, c\ x)}$$

$\boxed{P; E \vdash a}$

[COOP BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa}$$

[COOP COND]
$$\frac{P; E \vdash_{\text{lock}} \ell \qquad P; E \vdash a_i \quad \forall i \in 1..2}{P; E \vdash \ell\,?\,a_1 : a_2}$$

$\boxed{P; E \vdash_{\text{lock}} e}$

[LOCK EXP]
$$\frac{P; E \vdash e : c \cdot \text{AF} \qquad |e| \le MaxLockSize}{P; E \vdash_{\text{lock}} e}$$

$\boxed{P; E \vdash a \uparrow a'}$

[LIFT BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa \uparrow \kappa}$$

[LIFT LOCK]
$$\frac{P; E \vdash a_i \uparrow a'_i \quad \forall i \in 1..2 \qquad P; E \vdash_{\text{lock}} \ell \implies a'' = \ell\,?\,a'_1 : a'_2 \qquad P; E \nvdash_{\text{lock}} \ell \implies a'' = a'_1 \sqcup a'_2}{P; E \vdash (\ell\,?\,a_1 : a_2) \uparrow a''}$$

$\boxed{P; E \vdash meth}$

[METHOD]
$$\frac{P; E, \overline{d\ x} \vdash e : c \cdot a' \qquad P; E, \overline{d\ x} \vdash a \qquad a' \sqsubseteq a}{P; E \vdash a\ c\ m\,(\overline{d\ x})\ \{\ e\ \}}$$

$\boxed{P \vdash defn}$

[CLASS]
$$\frac{field_i = d_i\ f_i \quad \forall i \in 1..m \qquad \texttt{class}\ d_i\ \{\ \ldots\} \in P \quad \forall i \in 1..m \qquad P; c\ \texttt{this} \vdash meth_i \quad \forall i \in 1..n}{P \vdash \texttt{class}\ c\ \{\ field_{1..m}\ meth_{1..n}\ \}}$$

$\boxed{P \vdash \text{OK}}$

[PROGRAM]
$$\frac{P = defn_{1..n} \qquad P \vdash defn_i \quad \forall i \in 1..n \qquad ClassesOnce(P) \qquad FieldsOnce(P) \qquad MethodsOnce(P)}{P \vdash \text{OK}}$$

Fig. 10. YIELDJAVA type rules (II).

[EXP LET] This rule for `let` x `=` e_1 `in` e_2 infers effects a_1 and a_2 for e_1 and e_2, respectively. Care must be taken when constructing the effect for the `let` expression because a_2 may refer to the let-bound variable x. For example, the body of the following `let` expression produces an effect that is conditional on whether the lock `x` is held.

let x = e_1 in x.sync ...

Thus, we apply the substitution $[x := e_1]$ to yield a corresponding effect $a_2[x := e_1]$ that does not mention x. However, e_1 may not have effect AF, in which case $a_2[x := e_1]$ may not be valid (because it could contain e_1 a non-constant lock expression). As in previous work [18], we use the judgment

$$P; E \vdash a_2[x := e_1] \uparrow a'_2$$

to *lift* the effect $a_2[x := e_1]$ to a well-formed effect a'_2 that is greater than or equal to $a_2[x := e_1]$. For example, if $a_2 = (x\,?\,\text{AM} : \text{AN})$ and $e_1 = y.f$, where f is a *Normal* field, then $a_2[x := e_1] = (y.f\,?\,\text{AM} : \text{AN})$, which is not a valid effect because $y.f$ is not a valid lock. Thus the above judgment lifts it to the valid effect $a'_2 = \text{AN}$. The lifting judgment is defined in Fig. 10 via the following rules:

[LIFT BASE] Basic effects are always well-formed and remain unchanged when lifted.
[LIFT LOCK] If a conditional effect refers to a well-formed lock, this rule preserves the conditional test and recursively lifts the two component effects. If a conditional effect refers to an ill-formed lock expression, the rule eliminates the conditional test and instead joins together the two recursively lifted component effects.

[EXP CALL ATOMIC] This rule handles calls to atomic methods. The callee's declared effect a' may refer to `this` or parameters $x_{1..n}$. Therefore, we substitute
- the actual receiver e for `this` and
- the actual arguments $e_{1..n}$ for the parameters $x_{1..n}$

to produce the effect $a'[\texttt{this} := e, x_i := e_i\ ^{i \in 1..n}]$, and ensure that the resulting effect is valid by lifting it to an effect a'' that is well-formed in the current environment. Effect a'' must be an atomic effect, that is, less than or equal to AN.

The rule for a compound method invocation $e_\gamma m\#(e_{1..n})$ is similar, but removes the requirement that the computed effect a'' is atomic.

[EXP FORK] A fork expression `fork` e creates a new thread to evaluate e. Since a fork operation cannot commute past the operations of its child thread, fork operations are left movers.

[LOCK EXP] The judgment $P; E \vdash_{\text{lock}} e$ checks that e is a well-formed lock expression in environment E. The lock expression e must denote a fixed lock throughout the execution of the program to ensure soundness. Thus, we require that e has effect AF.

In addition, each lock expression e has a size $|e|$, which is the number of field accesses it contains. To ensure termination of the specification inference algorithm presented in the second half of this article, we require that the size of each lock expression be bounded by the constant *MaxLockSize*. The size restriction poses no limitation in practice since lock relationships are never complex enough to be problematic. Choosing a *MaxLockSize* of 4 proved sufficient in our experiments, with larger values yielding no real improvement in precision.

[METHOD], [CLASS], and [PROG] These rules verify the basic well-formedness requirements of methods, classes, and programs. The [PROG] rule uses several additional predicates to ensure no class is declared twice (*ClassesOnce*(P)), no field name is declared twice in a class (*FieldsOnce*(P)), and no method name is declared twice in a class (*MethodsOnce*(P)). These are defined more precisely in [22]. While these properties could be verified by a separate type system, we include them here so that all necessary conditions for type safety are verified by a single set of rules.

4.3. Correctness

Appendix A presents a formal operational semantics for YIELDJAVA. A run-time state σ extends the program's source code with dynamically allocated objects and dynamically created threads. We present two variants of our operational semantics: a preemptive semantics $(\to)$ that context switches at arbitrary program points, and a cooperative semantics $(\to_c)$ that context switches only at yield annotations. As expected, the cooperative semantics is more restrictive than the preemptive, and thus $(\to_c) \subset (\to)$. Both satisfy the standard preservation property that evaluation preserves well-typing (under an appropriate extension of the type system to run-time states, denoted $\vdash \sigma$).

Theorem 4.1 *(Preservation). If* $\vdash \sigma$ *and* $\sigma \to \sigma'$ *then* $\vdash \sigma'$.

Proof is by case analysis on the operational semantics rules and is presented in [56].

The more interesting correctness theorem is that a well-typed program exhibits equivalent behavior under both semantics. We say a state σ is *yielding* if no thread in σ is in the middle of a transaction. If a well-typed yielding state σ can reach a yielding state σ' under the preemptive semantics, it can also reach that state under the cooperative semantics. Note that in any terminating execution, the initial and final states of that execution are yielding, and the following theorem guarantees equivalence under both semantics.

Theorem 4.2 *(Cooperative-preemptive equivalence). Suppose* $\vdash \sigma$ *where* σ *is yielding. If* $\sigma \to^* \sigma'$ *where* σ' *is also yielding, then* $\sigma \to_c^* \sigma'$.

We outline the proof of this theorem in Appendix B, where we also extend it to cover the technically more subtle case of when the program goes wrong.

With Theorem 4.2, we can reason about the correctness of well-typed programs under the simpler cooperative semantics $(\to_c)$, since this correctness result also applies to executions under the preemptive semantics $(\to)$. Note that the converse to this theorem also holds, since $(\to_c) \subset (\to)$.

5. Implementation

We have developed an implementation called JCC that extends the YIELDJAVA type system to support the Java language.[3] JCC uses the standard Java modifiers `final` and `volatile` to classify fields as either *Final* or *Volatile*; all other fields are considered *Normal*. We introduce one new modifier, `racy`, to capture intentionally racy *Normal* fields. JCC assumes that correct field annotations are provided for the input program. Such annotations could be verified or generated using RCC/JAVA [1] or any other analysis technique. For our experiments, we leveraged that tool, as well as the FASTTRACK [17] data race detector, to identify racy fields.

JCC supports annotations on methods to describe their effects, as in YIELDJAVA. The following three effect keywords are sufficient to annotate most methods:

`atomic`*:* an atomic non-mover method with effect AN.
`mover`*:* an atomic both-mover method with effect AM.
`compound`*:* a compound non-mover method with effect CN.

JCC supports annotations for all elements of the conditional effect lattice, such as `(atomic left-mover)` and `(this ? mover : compound)`, as well. Method effect annotations appear alongside standard method modifiers, as in the following illustrative examples:

[3] Our implementation does not currently support generic classes due to limitations in the front-end checker upon which JCC is built, but supporting generic types is not fundamentally problematic.

Table 1
Effects for other access forms.

Access type	Syntax	Effect
racy read	$e_1 \mathrel{..} f\#$	a_1; CL
racy write	$e_1 \mathrel{..} f\# = e_2$	a_1; a_2; CL
race-free array read	$e_1[e_2]$	a_1; a_2; AM
race-free array write	$e_1[e_2] = e_3$	a_1; a_2; a_3; AM
racy array read	$e_1[e_2]\#$	a_1; a_2; CL
racy array write	$e_1[e_2]\# = e_3$	a_1; a_2; a_3; CL
write-guarded read with lock held	$e_1 . f$	a_1; AM
write-guarded read without lock held	$e_1 \mathrel{..} f$	a_1; AN
write-guarded write with lock held	$e_1 . f = e_2$	a_1; a_2; AN

```
atomic void m1() { ... }

atomic left-mover void m2() { ... }

lock?mover:atomic void m3() { ... }

lock?(atomic left-mover):(compound non-mover) void m4() { ... }
```

As in YieldJava, field accesses and method invocations may be written using "`..`" in place of "`.`" to indicate interference points. Yielding synchronized statements use the syntax "`..synchronized(e) { ... }`".

Given a program with cooperability annotations, JCC reports a warning whenever interference may occur at a program point not corresponding to a yield, and whenever a method's specification is not satisfied by its implementation.

5.1. Type system extensions

JCC extends YieldJava to support subtyping, subclassing, and method overriding. It allows the effect of an overriding method to change covariantly, and so requires that $b \sqsubseteq a$ in the following:

```
class C           {  a t m() { ... }   }
class D extends C {  b t m() { ... }   }
```

In addition, although data races should in general be avoided, large programs often have some intentional races, which JCC supports via a `racy` annotation on *Normal* fields. A read from a racy field must be written as $e \mathrel{..} f\#$. Here, the "`..`" syntax as usual indicates a yield point, and the trailing `#` identifies the racy nature of the read (and that the programmer needs to consider the consequences of Java's relaxed memory model [36]). The overall effect of $e \mathrel{..} f\#$ is the composition of a yield and a non-mover memory access: (CY; AN) = CL. Writes are similar. We summarize these extra rules, and those that follow, in Table 1.

The YieldJava checker handles array accesses similar to *Normal* fields. Note that in Java, array elements are never `final` or `volatile`. Racy array accesses must be annotated with "#", as in $a[i]\#$, and are assumed to be yield points.

YieldJava also supports write-guarded fields (such as `shortestPathLength` from Fig. 1) for which a protecting lock is held for all writes but not necessarily for reads. In this case, a read while holding the protecting lock is a both-mover, since there can be no concurrent writes. However, a write with the lock held is a non-mover, since there may be concurrent reads that do not hold the lock.

5.2. Default field and method annotations

To reduce the burden of annotating code with cooperability effects, JCC uses carefully chosen defaults when annotations are absent. In particular, unannotated fields are race-free, and unannotated methods are atomic both-movers. Default annotations help most for classes that were not designed to be shared (or are at least race free) and for code that contains no synchronization. They help less for code where synchronization is used.

Table 2
Interference points and unintended yields.

Program	Size (lines)	Annotation time (min)	Annotation count	Interference points					Unintended yields
				NoSpec	Race	Atomic	AtomicRace	Yields	
java.util.zip.Inflater	317	9	4	44	12	0	0	0	0
java.util.zip.Deflater	381	7	8	57	13	0	0	0	0
java.lang.StringBuffer	1276	20	9	235	39	10	3	1	1
java.lang.String	2307	15	5	258	34	6	2	1	0
java.io.PrintWriter	534	40	108	120	82	114	81	51	9
java.util.Vector	1019	25	43	214	41	37	17	7	1
java.util.zip.ZipFile	490	30	60	140	86	87	55	33	0
sparse	868	15	15	328	32	50	16	7	0
tsp	706	10	45	475	53	405	48	24	0
elevator	1447	30	62	492	76	223	42	22	0
raytracer-fixed	1915	10	46	629	124	104	38	26	2
sor-fixed	958	10	30	249	38	126	22	15	0
moldyn-fixed	1352	10	43	991	52	657	34	28	0
Total	13,570	231	478	4232	682	1819	358	215	13
per KLOC		17	35	312	50	134	26	16	1

6. Experimental evaluation

We applied JCC to benchmark programs including a number of standard library classes from Java 1.4[4]; `sparse`, `raytracer`, `sor`, and `moldyn` from the Java Grande suite [29]; `tsp`, a solver for the traveling salesperson problem [53]; and `elevator`, a real-time discrete event simulator [53]. These programs use a variety of synchronization idioms, and previous work has revealed a number of interesting concurrency bugs in these programs. Thus, these benchmarks show the ability of our annotations to capture thread interference under various conditions and to highlight unintended, problematic interference. Three of these programs (`raytracer`, `sor`, and `moldyn`) use broken barrier implementations [17]. We discuss those problems below and use versions with corrected barrier code (named `raytracer-fixed`, `sor-fixed`, and `moldyn-fixed`) in our experiments. We performed all experiments on an Apple MacBook Pro computer with a 2.6 GHz Intel Core i7 processor and 16 GB of memory. We used the Java 1.7.0 HotSpot 64-bit Server VM. The checker analyzed each benchmark in under 1 second.

Table 2 shows the size of each benchmark program, the number of annotations required to enable successful type checking, and the wall clock time required for one programmer to identify and add all annotations necessary for JCC to accept the program without warnings. The annotation count includes all `racy` field annotations, method specifications, and occurrences of "`..`" and "`#`". The annotation burden was relatively low, but could still be expensive for large code bases. Each program was manually annotated and checked in about 10 to 30 minutes, and roughly one annotation per 30 lines of code was required. We did have some previous experience using these programs, which facilitated the annotation process, but since we intend JCC to be used during development, we believe our experience reflects the cost incurred by the intended use of our technique.

Overall, we found that inserting yield annotations requires a solid understanding of the synchronization protocols used by the target program and of where interference may occur, but was otherwise relatively straight-forward. Inserting method specifications was often as time consuming as inserting yield annotations, in part because potentially many method specifications would change whenever a yield annotation was added or removed. We discuss mitigating that overhead in the next section when we discuss method specification inference.

We show how to mitigate much of this overhead in Section 7, where we discuss type inference for method specifications.

6.1. Examples of defects

The primary purpose of yield annotations is to facilitate formal and informal reasoning about higher-level correctness properties. At the same time, yield annotations also facilitate recognizing concurrency errors. To explore this aspect, the final column in Table 2 shows the number of *unintended yields* in each program, where manual code inspection shows that thread interference is unintentional and erroneous. These unintended yields highlight all known concurrency bugs, including atomicity violations and data races [16,18] detectable with prior tools. (Those errors were known to us before annotating the programs.)

[4] We used Java 1.4 classes because JCC was built as an extension to a Java front end [20] that does not support generic classes. Handling generic classes and features from later versions of Java pose no substantial theoretical challenges but would add complexity to our formal system and implementation.

```
1  class RayTracerRunner implements Runnable {
2    int id;
3
4    compound public void run() {
5      // init
6      br.DoBarrier#(id);
7      // render
8      ..synchronized (scene) {
9        for(int i=0;i<JGFRayTracerBench.nthreads;i++)
10         if (id == i)
11           JGFRayTracerBench..checksum1 =
12               JGFRayTracerBench..checksum1
13               + checksum;
14     }
15     br.DoBarrier#(id);
16     // cleanup
17   }
18 }
```

Fig. 11. RayTracer.

6.2. StringBuffer and Vector

The yield annotation shown on line 12 in Fig. 13 highlights that other threads may concurrently modify `sb`, potentially causing `append()` to crash, in violation of its specified thread-safe behavior under JDK 1.4. A constructor in the `Vector` class suffers from a similar defect. Similar pitfalls occur in `Vector`'s inherited methods `removeAll(c)` and `retainAll(c)` [18]. In this experiment, we did not verify the correctness of inherited code, but JCC readily catches those errors when that code is checked.

6.3. RayTracer

The raytracer benchmark uses a barrier `br` to coordinate several rendering threads, as shown in Fig. 11. After rendering, each thread acquires the lock `scene` before adding its local `checksum` to the global shared variable `checksum1`. However, each thread creates its own `scene` object, and thus acquiring `scene` fails to ensure mutual exclusion over the updates to `checksum1`. This is made clear by the explicit yields on the reads and writes of `checksum1` on lines 11 and 12.

6.4. SOR and Moldyn

In `sor` (see Fig. 12), the computation threads synchronize on a barrier implemented as a shared two-dimensional array `sync`. Unfortunately, the barrier is broken, since the `volatile` keyword applies only to the array reference, not the array elements. Thus, the barrier synchronization code at the bottom of the main processing loop may not properly coordinate the threads, leading to races on the data array `G`. This problem is obvious when using JCC because racy annotations must be added in dozens of places, essentially to all accesses of `sync` and `G`. When the barrier is fixed, we obtain much cleaner code: the yield count decreases from 40 to 12. In particular, the accesses to `G` between barrier calls are free of yields, signifying that between barriers, sequential reasoning is applicable.

The `moldyn` benchmark uses a barrier object with a similar error in the use of `volatile`s and arrays. This bug leads to potential races on all data accesses intended to be synchronized by the barrier, and a large number (58) of yield annotations were necessary to document all such cases.

6.5. Non-interference specifications

Yield annotations provide a precise specification of exactly where thread interference may occur. In this section, we compare yield annotations with two established techniques for reasoning about thread interference based on identifying data races and atomic methods. Specifications for data races come in the form of `race` annotations on all accesses that may be involved in a data race. Specifications for atomic methods come in the form of either `atomic` or `non-atomic` annotations on methods.

While these specifications for data races and atomic methods do not directly identify all IPs, IPs can be inferred by performing additional analysis based on these specifications. This additional analysis is analogous to the reasoning performed by our type system but is less precise, as it is based on coarser specifications than the conditional effects used by our system.

To experimentally compare cooperability with these prior approaches we count the IPs in each benchmark under the following five approaches for specifying non-interference.

```
1  class SORRunner implements Runnable {
2    double G[][];
3    volatile long sync[][];
4
5    compound public void run() {
6      ...
7      for (int p = 0; p < 2*num_iterations; p++) {
8        for (int i=ilow+(p%2); i < iupper; i=i+2) {
9          ...
10         for (int j=1; j < Nm1; j=j+2){
11           G[i][j]# = omega_over_four *
12               ( G[i-1][j]# + G[i+1][j]# +
13                 G[i][j-1]# + G[i][j+1]# ) +
14              one_minus_omega * G[i][j]#;
15           ...
16         }
17       }
18
19       sync[id][0]# = sync[id][0]# + 1;
20       if (id > 0)
21         while (sync[id-1][0]# < sync[id][0]#) ;
22       if (id < JGFSORBench.nthreads-1)
23         while (sync[id+1][0]# < sync[id][0]#) ;
24     }
25   }
26 }
```

Fig. 12. Original SOR Algorithm.

NoSpec: If we have no information about thread interference, then there is a potential IP immediately before any field access or lock acquire, since those operations may conflict with operations of concurrent threads. Additionally, method calls are considered IPs, since interference could occur inside the callee. We exclude operations that never cause interference, such as accesses to local variables, lock releases, method calls, etc. from this count.

Race: If we have information about race-free fields, then we only count IPs on each racy field access, lock acquires, and method calls.

Atomic: If we know that certain methods are atomic and hence free of interference, then we consider IPs to occur at all field accesses, lock acquires, and method calls occurring inside a non-atomic method.

AtomicRace: If we have information about both racy fields and atomic methods, then we only count IPs at racy field accesses, lock acquires, and method calls inside non-atomic methods.

Yield: We use the yield annotation "`..`" and the compound method call annotation "`#`" to identify exactly those program points at which interference may occur.

Table 2 shows the number of interference points (IPs) in each benchmark under each non-interference specification. Benchmarks in which all methods are `atomic`, such as `Inflater`, have zero interference points under both the *Atomic* and *Yield* specifications. The results confirm that both *Race* (50 IP/KLOC) and *Atomic* (135 IP/KLOC) are useful non-interference specifications that reduce the number of interference points in comparison to the base-case *NoSpec* (312 IP/KLOC). *AtomicRace* (26 IP/KLOC) combines complementary benefits from both *Atomic* and *Race*, and is better than either of these approaches in isolation. Finally, *Yield* (16 IP/KLOC) provides a significant improvement over these prior approaches. Put differently, these results indicate that for these benchmarks, the prior approaches for specifying non-interference significantly over-approximate the set of IPs.

We sketch two situations illustrating why yield annotations based on the reasoning performed by our type system is more precise than *AtomicRace*. First, for the TSP algorithm in Fig. 1 under the *AtomicRace* specifications, both `path.isComplete()` and `path.children()` would be labeled as `atomic` and `searchFrom` would be labeled as `non-atomic`. Thus, *AtomicRace* requires an interference point before the calls to `path.isComplete()` and `path.children()` from within `searchFrom`. In contrast, our type system identifies that these two methods are more precisely characterized as both movers that do not interfere with other threads, and so no yield point is necessary at these calls.

This example illustrates that atomic method specifications are not sufficient to precisely identify IPs. Additional information about which atomic methods are both-movers would help, but that extension would then force the programmer to perform additional reasoning to properly identify IPs. In contrast, *Yield* supports directly annotating those IPs in a syntactically explicit manner that requires no extra reasoning by the programmer

```
1  public final class StringBuffer ··· {
2    (this ? mover : atomic) int length() { ... }
3    (this ? mover : atomic) void getChars(...) { ... }
4
5    compound synchronized StringBuffer append(StringBuffer sb) {
6      ...
7      int len = sb.length();
8      int newcount = count + len;
9      if (newcount > value.length) {
10         expandCapacity(newcount);
11     }
12     sb..getChars(0, len, value, count);
13     count = newcount;
14     return this;
15   }
16 }
```

Fig. 13. StringBuffer.

Table 3
Programmer-inserted annotations in benchmarks when method specifications are all added manually, added in conjunction with defaults, and automatically inferred.

Program	Size (lines)	Programmer-Inserted Annotations (count)								
		Yield Point	Non-Atomic Call	Racy Field	Method Specs.			Total		
					Manual	Defaults	Inference	Manual	Defaults	Inference
java.util.zip.Inflater	317	0	0	0	25	4	0	25	4	0
java.util.zip.Deflater	381	0	0	1	26	7	0	27	8	1
java.lang.StringBuffer	1276	1	0	0	45	8	0	46	9	1
java.lang.String	2307	1	0	1	59	3	0	61	5	2
java.io.PrintWriter	534	26	42	3	37	37	0	108	108	71
java.util.Vector	1019	4	3	0	75	36	0	82	43	7
java.util.zip.ZipFile	490	25	5	7	33	18	0	70	55	37
sparse	868	6	3	0	46	6	0	55	15	9
tsp	706	19	12	3	17	11	0	51	45	34
elevator	1447	21	11	1	64	24	0	97	57	33
raytracer-fixed	1915	25	5	3	88	12	0	121	45	33
sor-fixed	958	12	6	0	48	12	0	66	30	18
moldyn-fixed	1352	27	12	0	54	10	0	93	49	39
Total	13,570	167	99	19	617	188	0	902	473	285
per KLOC		12	7	1	45	14	0	66	35	21

As a second example, consider the method `StringBuffer.append()` in Fig. 13. This non-atomic method in turn calls two methods `sb.length()` and `sb.getChars()`, both of which are atomic (since the lock `sb` is not held at these call sites). Under *AtomicRace*, an interference point must be assumed before each of these calls to an atomic method from a non-atomic context. In contrast, our type system can verify that the lock acquire of `this` at the start of `append()` can move right to just before the `sb.length()` call, so no yield is required at the call to `sb.length()`. Thus, *AtomicRace* requires two interference points in this method, whereas our type system requires just one.

These two examples illustrate why the notions of data races and atomic methods are not by themselves sufficient to identify interference points in a precise manner. Extending *AtomicRace* to include more precise method specifications for identifying mover methods and conditionally atomic methods would help, but then both the method specifications and the programmer's manual analysis to reason about IPs begins to converge on the reasoning performed by our type system. Our system both automates the process of reasoning about IPs and ensure that they are syntactically explicit in the source program.

7. Method specification inference

While JCC demonstrates the feasibility of programming with cooperability, the need to fully annotate the source code does impose a burden on the programmer, particularly for large or frequently changing code bases. Table 3 shows, for each benchmark, how many annotations must be added for JCC to verify type safety. That table includes the number of annotations for identifying "Yield Points", "Non-Atomic Method Calls", "Racy Field Declarations", and "Method Specifications". The number of programmer-inserted method specifications is reported for three scenarios:

- *Manual*: No default specifications or inference techniques are used.

- *Defaults*: The default method specification of `atomic both-mover` from Section 5 is used when applicable.
- *Inference*: Type inference is used to infer all specifications.

We also report the sum of all programmer-inserted specifications for these three scenarios in the "Total" columns. That is, the "Manual Total" column sums the "Yield Point", "Non-Atomic Call", "Racy Field", and "Manual Method Specs" required when default values and inference are not used. In that case, JCC requires about 66 annotation per KLOC for our benchmarks. When default method specifications are used, the number of annotations required drops to 35 per KLOC, of which about 40% are method specifications, as reported in the "Default Total" column. (That column corresponds to the number of annotations in Table 2).

As mentioned earlier, using the default annotations for methods works well for synchronization-free and race-free methods, but not as well on methods that perform synchronization or contain yield points. For example, the defaults are not applicable to the `searchFrom` method (Fig. 1), the methods of the `Bit` class (Fig. 5), or the code fragments presented in the previous section (Figs. 11–13). Specifications for those remaining methods are often the most subtle to get right and are prone to change if code is modified (by, for example, inserting/removing new yield points or method calls). Moreover, determining what they should be accounted for a non-trivial fraction (about 30–60%) of the time spent annotating the programs in our study, even when the default annotations were used for common cases. Additionally, method specifications often had to be revised as additional yield points were added to satisfy the type checker.

To mitigate this annotation overhead for methods, we introduce an algorithm for inferring *all* method specifications not provided by the programmer, reducing the number of required annotations in the benchmarks from 35 per KLOC to 21 per KLOC. The inferred annotations can be inserted into the source code to enable subsequent modular checking of cooperability and provide documentation to the user, and they can be regenerated as needed if later changes to the code significantly changes the behavior of many methods. Our inference algorithm enables us to eliminate most of the time spent annotating methods in our benchmarks, and it also enables specifications to be recomputed automatically if code changes introduce or remove yield points.

Our algorithm infers the most precise cooperability specification for each method via a constraint-based effect inference technique. That is, syntax-directed typing rules generate a set of constraints containing fresh "effect variables" for the missing annotations. Those constraints are then solved via a least fixed-point computation over the effect lattice. As in our previous work on type inference for atomicity specifications [18], the presence of conditional effects containing lock expressions that must be immutable leads to significant complexity. To adhere to these immutability restrictions, constraints generated by our analysis include constructs to describe well-formedness requirements on conditional effects, and the constraint solver leverages the type system to enforce them.

When using method specification inference, the programmer is only responsible for inserting the annotations capturing racy fields, yield points, and non-atomic calls. Fields with potential races could be identified via existing sound, static analyses (such as [37]), but inferring yield points and non-atomic call sites remains for future work. Inferring yield points would complement our effect inference algorithm presented here, but it is a subtle inference problem since there is some choice in where yields are placed in the code (*e.g.*, within the methods of a class versus at call sites to those methods). Thus, inferring yield points has an optimization aspect that will be better understood once we have more experience with JCC.

7.1. Language extensions for specification inference

To support type inference for method effects, we extend cooperability effects to include effect variables φ. An *open effect* s is either an (explicit) conditional effect a or an effect variable φ.

μ	$::= \text{F} \mid \text{Y} \mid \text{M} \mid \text{R} \mid \text{L} \mid \text{N}$	(mover effect)
τ	$::= \text{A} \mid \text{C}$	(atomicity effects)
κ	$::= \tau\,\mu$	(combined effect)
a	$::= \kappa \mid l\,?\,a_1 : a_2$	(conditional effects)
φ	$\in$ *EffectVar*	(effect variables)
s	$::= a \mid \varphi$	(open effects)

We permit methods to be annotated by effect variables as well as explicit cooperability effects:

$$meth ::= s\; t\; m\,(\,arg^*\,)\;\{\; e\;\}$$

A program is *explicitly-typed* if it does not contain effect variables. The *specification inference* problem is, given a program P with effect variables, to replace each effect variable with a conditional effect so that the resulting explicitly typed program is well-typed. The following subsections describe the constraint language, the type inference rules that generate constraints, and our constraint solving algorithm.

$$
\begin{array}{rcl|rcl}
[\![a]\!] & = & a & A(a) & = & a \\
[\![r_1 \mathbf{;} r_2]\!] & = & [\![r_1]\!];[\![r_2]\!] & A(r_1\mathbf{;}r_2) & = & A(r_1)\mathbf{;}A(r_2) \\
[\![r_1 \boldsymbol{\sqcup} r_2]\!] & = & [\![r_1]\!] \sqcup [\![r_2]\!] & A(r_1\boldsymbol{\sqcup}r_2) & = & A(r_1)\boldsymbol{\sqcup}A(r_2) \\
[\![r^{\boldsymbol{*}}]\!] & = & [\![r]\!]^* & A(r^{\boldsymbol{*}}) & = & A(r)^{\boldsymbol{*}} \\
[\![l\,\mathbf{?}\,r_1 \mathbf{:} r_2]\!] & = & l\,?\,[\![r_1]\!] : [\![r_2]\!] & A(r\boldsymbol{\cdot}\theta) & = & A(r)\boldsymbol{\cdot}\theta \\
[\![r\boldsymbol{\cdot}\theta]\!] & = & \theta([\![r]\!]) & A(l\,\mathbf{?}\,r_1\mathbf{:}r_2) & = & l\,\mathbf{?}\,A(r_1)\mathbf{:}A(r_2) \\
[\![\mathbf{S}(l,\gamma,r)]\!] & = & S(l,\gamma,[\![r]\!]) & A(\mathbf{S}(l,\gamma,r)) & = & \mathbf{S}(l,\gamma,A(r)) \\
[\![\mathbf{Lift}(P,E,r)]\!] & = & a \quad \text{such that } P;E \vdash [\![r]\!] \uparrow a & A(\mathbf{Lift}(P,E,r)) & = & \mathbf{Lift}(A(P),E,A(r))
\end{array}
$$

Fig. 14. Definitions of the meaning function $[\![\cdot]\!]$ and assignment mappings for cooperability expressions.

7.2. Cooperability constraints

A *constraint* C relates a *cooperability expression* r and an open effect s:

$$C \ ::= \ r \boldsymbol{\sqsubseteq} s$$

Cooperability expressions include open effects as well as constructs for representing various operations on effects, such as sequential composition, join, iteration, and substitution.

$$r \ ::= \ s \mid r\mathbf{;}r \mid r\boldsymbol{\sqcup}r \mid r^{\boldsymbol{*}} \mid l\,\mathbf{?}\,r\,\mathbf{:}\,r \mid r\boldsymbol{\cdot}\theta \mid \mathbf{S}(l,\gamma,r) \mid \mathbf{Lift}(P,E,r)$$

We use bold symbols such as "$\boldsymbol{\sqsubseteq}$" and "$\mathbf{;}$" to distinguish the syntactic constructs relating effect expressions from the corresponding semantic operations "$\sqsubseteq$" and "$;$" on effects. The expression forms for sequential composition ($r\mathbf{;}r$), join ($r\boldsymbol{\sqcup}r$), iterative closure ($r^{\boldsymbol{*}}$), conditional effects ($l\,\mathbf{?}\,r\,\mathbf{:}\,r$), and synchronization ($\mathbf{S}(l,\gamma,r)$) are analogous to the underlying operations on effects. The delayed substitution expression form $r\boldsymbol{\cdot}\theta$, where $\theta = [x_1 = e_1, \ldots, x_2 = e_n]$ is a map from program variables to expressions, and the lifting expression form $\mathbf{Lift}(P,E,r)$ are described where used below.

An cooperability expression is *closed* if it does not contain effect variables. The meaning function $[\![\cdot]\!]$ shown in Fig. 14 maps closed cooperability expressions to effects:

$$[\![\cdot]\!] : \mathit{ClosedCoopExpr} \rightarrow \mathit{Effect}$$

7.3. Assignments

An *assignment* A maps effect variables to effects:

$$A : \mathit{EffectVar} \rightarrow \mathit{Effect}$$

The ordering relation for assignments is the point-wise extension of the sub-cooperability relation:

$$
\begin{array}{rcl}
A_1 \sqsubseteq A_2 & \text{iff} & \forall\varphi.\ A_1(\varphi) \sqsubseteq A_2(\varphi) \\
\bot & \stackrel{\text{def}}{=} & \lambda\varphi.\ \text{AF}
\end{array}
$$

We extend assignments in a compatible manner to arbitrary cooperability expressions, as shown in Fig. 14. We also extend assignments to programs, so that the program $A(P)$ is identical to P, except that each effect variable φ is replaced with its meaning $A(\varphi)$.

An assignment A *satisfies* a constraint $C = r \boldsymbol{\sqsubseteq} s$ (written $A \models C$) if, after applying the assignment, the meaning of the left-hand side of the constraint is a sub-cooperability-effect of the right-hand side:

$$A \models r \boldsymbol{\sqsubseteq} s \quad \text{iff} \quad [\![A(r)]\!] \sqsubseteq A(s)$$

If $A \models C$ for all $C \in \bar{C}$ then A is a *solution* for $\bar{C}$, written $A \models \bar{C}$. A set of constraints $\bar{C}$ is *valid*, written $\models \bar{C}$, if every assignment is a solution for $\bar{C}$. In particular, if A is a solution for $\bar{C}$, then $A(\bar{C})$ is valid, and vice-versa.

7.4. Type inference rules

The YieldJava type inference judgments and rules are shown in Figs. 15–16. Mostly, these judgments extend the type checking judgments of the earlier type system to also generate atomicity constraints. For example, the main type inference judgment:

$$P; E \vdash e \mathbf{:} t \cdot r \cdot \bar{C}$$

now yields a set of constraints $\bar{C}$ generated from type checking e with respect to program P and environment E. We highlight the most interesting extensions:

$$\boxed{P; E \vdash e : c \cdot r \cdot \bar{C}}$$

[INF VAR]
$$\frac{P \vdash E \quad E = E_1, c\,x, E_2}{P; E \vdash x : c \cdot \text{AF} \cdot \emptyset}$$

[INF NULL]
$$\frac{P \vdash E \quad \texttt{class}\; c\; \{\; \ldots\; \} \in P}{P; E \vdash \texttt{null} : c \cdot \text{AF} \cdot \emptyset}$$

[INF NEW]
$$\frac{\texttt{class}\; c\; \{\; d_i\; x_i\;^{i\in 1..n}\; \ldots\; \} \in P \quad P; E \vdash e_i : d_i \cdot r_i \cdot \bar{C}_i \quad \forall i \in 1..n}{P; E \vdash \texttt{new}\; c\,(e_{1..n}) : c \cdot (r_1; \cdots; r_n; \text{AM}) \cdot \bar{C}_{1..n}}$$

[INF REF]
$$\frac{P; E \vdash e : c \cdot r \cdot \bar{C} \quad \texttt{class}\; c\; \{\ldots d\; f \ldots\} \in P \quad (f \in \mathit{Normal} \wedge r' = \text{AM}) \vee (f \in \mathit{Final} \wedge r' = \text{AF})}{P; E \vdash e.f : d \cdot (r;r') \cdot \bar{C}}$$

[INF ASSIGN]
$$\frac{P; E \vdash e : c \cdot r \cdot \bar{C} \quad P; E \vdash e' : d \cdot r' \cdot \bar{C}' \quad \texttt{class}\; c\; \{\ldots d\; f \ldots\} \in P \quad f \in \mathit{Normal}}{P; E \vdash (e.f = e') : d \cdot (r;r';\text{AM}) \cdot (\bar{C} \cup \bar{C}')}$$

[INF REF VOLATILE]
$$\frac{P; E \vdash e : c \cdot r \cdot \bar{C} \quad \texttt{class}\; c\; \{\ldots d\; f \ldots\} \in P \quad f \in \mathit{Volatile}}{P; E \vdash e_\gamma f : d \cdot (r;\Gamma(\gamma);\text{AN}) \cdot \bar{C}}$$

[INF ASSIGN VOLATILE]
$$\frac{P; E \vdash e : c \cdot r \cdot \bar{C} \quad P; E \vdash e' : d \cdot r' \cdot \bar{C}' \quad \texttt{class}\; c\; \{\; \ldots\; d\; f\; \ldots\; \} \in P \quad f \in \mathit{Volatile}}{P; E \vdash (e_\gamma f = e') : d \cdot (r;r';\Gamma(\gamma);\, \text{AN}) \cdot (\bar{C} \cup \bar{C}')}$$

[INF CALL ATOMIC]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot r \cdot \bar{C} \\ \texttt{class}\; c\; \{\ldots \mathit{meth} \ldots\} \in P \\ \mathit{meth} = r'\; c'\; m\,(d_i\; x_i\;^{i\in 1..n})\; \{\; e'\; \} \\ \theta = [\texttt{this} := e, x_i := e_i\;^{i\in 1..n}] \\ r'' = r;r_1;\cdots;r_n;\Gamma(\gamma);\mathbf{Lift}(P, E, r'\bullet\theta) \\ P; E \vdash e_i : d_i \cdot r_i \cdot \bar{C}_i \quad \forall i \in 1..n \\ \bar{C}' = \bar{C} \cup \bar{C}_{1..n} \cup \{\mathbf{Lift}(P, E, r'\bullet\theta) \sqsubseteq \text{AN}\} \end{array}}{P; E \vdash e_\gamma m\,(e_{1..n}) : c' \cdot r'' \cdot \bar{C}'}$$

[INF CALL COMPOUND]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot r \cdot \bar{C} \\ \texttt{class}\; c\; \{\ldots \mathit{meth} \ldots\} \in P \\ \mathit{meth} = r'\; c'\; m\,(d_i\; x_i\;^{i\in 1..n})\; \{\; e'\; \} \\ \theta = [\texttt{this} := e, x_i := e_i\;^{i\in 1..n}] \\ r'' = r;r_1;\cdots;r_n;\Gamma(\gamma);\mathbf{Lift}(P, E, r'\bullet\theta) \\ P; E \vdash e_i : d_i \cdot r_i \cdot \bar{C}_i \quad \forall i \in 1..n \end{array}}{P; E \vdash e_\gamma m\#\,(e_{1..n}) : c' \cdot r'' \cdot (\bar{C} \cup \bar{C}_{1..n})}$$

[INF SYNC]
$$\frac{P; E \vdash_{\text{lock}} \ell \cdot \bar{C} \quad P; E \vdash e : c \cdot r \cdot \bar{C}'}{P; E \vdash \ell_\gamma\, \texttt{sync}\; e : c \cdot \mathbf{S}(\ell, \gamma, r) \cdot (\bar{C} \cup \bar{C}')}$$

[INF FORK]
$$\frac{P; E \vdash e : c \cdot r \cdot \bar{C}}{P; E \vdash \texttt{fork}\; e : \texttt{Unit} \cdot \text{AL} \cdot \bar{C}}$$

[INF WHILE]
$$\frac{P; E \vdash e_1 : c_1 \cdot r_1 \cdot \bar{C}_1 \quad P; E \vdash e_2 : c_2 \cdot r_2 \cdot \bar{C}_2 \quad r' = r_1;(r_2;r_1)^{*}}{P; E \vdash \texttt{while}\; e_1\; e_2 : \texttt{Unit} \cdot r' \cdot \bar{C}_{1..2}}$$

[INF IF]
$$\frac{P; E \vdash e_1 : d \cdot r_1 \cdot \bar{C}_1 \quad P; E \vdash e_i : c \cdot r_i \cdot \bar{C}_i \quad \forall i \in 2..3 \quad r' = r_1;(r_2 \sqcup r_3)}{P; E \vdash \texttt{if}\; e_1\; e_2\; e_3 : c \cdot r' \cdot \bar{C}_{1..3}}$$

[INF LET]
$$\frac{P; E \vdash e_1 : c_1 \cdot r_1 \cdot \bar{C}_1 \quad P; E, c_1\, x \vdash e_2 : c_2 \cdot r_2 \cdot \bar{C}_2 \quad r_2' = \mathbf{Lift}(P, E, r_2\bullet[x := e_1])}{P; E \vdash \texttt{let}\; x\; =\; e_1\; \texttt{in}\; e_2 : c_2 \cdot (r_1;r_2') \cdot \bar{C}_{1..2}}$$

Fig. 15. AtomicJava type inference rules (I).

[INF SYNC] The cooperability effect for a synchronized expression "$l_\gamma\,\texttt{sync}\; e$" is $\mathbf{S}(l, \gamma, r)$, where d is the cooperability expression for e. The semantics of $\mathbf{S}(l, \gamma, r)$ is defined in terms of the function S, that is, $[\![\mathbf{S}(l, \gamma, r)]\!] = S(l, \gamma, [\![r]\!])$.

[INF LET] This rule for $\texttt{let}\; x\; =\; e_1\; \texttt{in}\; e_2$ infers cooperability expressions r_1 and r_2 for e_1 and e_2, respectively. Since r_2 may mention x, we introduce the substitution $\theta = [x := e_1]$ as in the earlier type checking rule [EXP LET], but here we need to use the cooperability expression form $r_2\bullet\theta$ to delay applying this substitution until after effect variables in r_2 are resolved. (Similar delayed substitutions occur in [INF CALL ATOMIC] and [INF CALL COMPOUND].)
Furthermore, e_1 may not be AF (in general, we cannot determine which expressions are constant until after type inference), in which case $r_2\bullet\theta$ may not be a valid cooperability effect. Therefore, we use the cooperability expression $\mathbf{Lift}(P, E, r_2\bullet\theta)$ to yield a cooperability effect for e_2 that is well-formed in environment E.

[INF LOCK EXP] The judgment $P; E \vdash_{\text{lock}} e \cdot \bar{C}$ ensures that e is a valid lock expression. This rule checks that e denotes a fixed lock throughout the execution of the program by generating the constraint that e has effect AF.
As in the type system, the requirement that $|e| \le \mathit{MaxLockSize}$ ensures that there is only a finite number of valid lock expressions at any program point, which in turn bounds the size of conditional effects and guarantees termination of our type inference algorithm.

The method specification inference system defines the top-level judgment

$$P \vdash \bar{C}$$

where $\bar{C}$ is the generated set of constraints for the program P.

$\boxed{P; E \vdash_{\text{lock}} e \cdot \bar{C}}$

[INF LOCK EXP]
$$\frac{P; E \vdash e : c \cdot r \cdot \bar{C} \quad |e| \leq \mathit{MaxLockSize}}{P; E \vdash_{\text{lock}} e \cdot \bar{C} \cup \{r \sqsubseteq \text{AF}\}}$$

$\boxed{P; E \vdash s \cdot \bar{C}}$

[INF COOP BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa \cdot \emptyset}$$

[INF COOP COND]
$$\frac{P; E \vdash_{\text{lock}} \ell \cdot \bar{C} \quad P; E \vdash r_i \cdot \bar{C}_i \quad \forall i \in 1..2}{P; E \vdash \ell\, ?\, r_1 : r_2 \cdot (\bar{C} \cup \bar{C}_{1..2})}$$

[INF COOP VAR]
$$\frac{}{P; E \vdash \varphi \cdot \emptyset}$$

$\boxed{P; E \vdash \mathit{meth} \cdot \bar{C}}$

[INF METHOD]
$$\frac{E' = E, \overline{d\ x} \quad P; E' \vdash e : c \cdot r \cdot \bar{C} \quad P; E' \vdash s \cdot \bar{C}'}{P; E \vdash s\ c\ m\,(\overline{d\ x})\ \texttt{\{}\ \ e\ \texttt{\}} \cdot (\bar{C} \cup \bar{C}' \cup \{\mathbf{Lift}(P, E', r) \sqsubseteq s\})}$$

$\boxed{P \vdash \mathit{defn} \cdot \bar{C}}$

[INF CLASS]
$$\frac{\mathit{field}_i = d_i\ f_i \quad \forall i \in 1..m \quad \texttt{class}\ d_i\ \texttt{\{}\ \ldots\ \texttt{\}} \in P \quad \forall i \in 1..m \quad P; c\ \texttt{this} \vdash \mathit{meth}_i \cdot \bar{C}_i \quad \forall i \in 1..n}{P \vdash \texttt{class}\ c\ \texttt{\{}\ \ \mathit{field}_{1..m}\ \mathit{meth}_{1..n}\ \texttt{\}} \cdot \bar{C}_{1..n}}$$

$\boxed{P \vdash \bar{C}}$

[INF PROGRAM]
$$\frac{P = \mathit{defn}_{1..n} \quad P \vdash \mathit{defn}_i \cdot \bar{C}_i \quad \forall i \in 1..n \quad \mathit{ClassesOnce}(P) \quad \mathit{FieldsOnce}(P) \quad \mathit{MethodsOnce}(P)}{P \vdash \bar{C}_{1..n}}$$

Fig. 16. AtomicJava type inference rules (II).

We demonstrate the type inference process with a version of the earlier-described `Bit` class that has not been explicitly-annotated with method specifications. A version of that class definition containing effect variables for missing annotations and a simple client is shown in Fig. 17. For this program, the type system generates the constraints in Fig. 18, which also shows the typing rule and source location causing each constraint to be generated. That figure omits several trivial constraints that do not involve effect variables, *e.g.* AF $\sqsubseteq$ AF. Also, the constants `true` and `false` have effect AF, as does the boolean operation `!`e.

As a technical requirement, we introduce the notion of a well-formed assignment. An assignment A is *well-formed* for $\bar{C}$ if, for all constraints $(\mathbf{Lift}(P, E, r) \sqsubseteq \varphi)$ in $\bar{C}$, $A(P); E \vdash A(\varphi)$. In other words, $A(\varphi)$ cannot refer to lock expressions that are not well-formed in the environment E.

With this definition, we can now state that if A is a well-formed solution to the constraints for a program P, then the explicitly-typed program $A(P)$ is well-typed.

Theorem 7.1 *(Type inference yields well-typed programs). If $P \vdash \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$ then $A(P) \vdash wf$.*

The proof is by induction over the derivation of $P \vdash \bar{C}$. The well-formedness requirement on A ensures that the cooperability effects for effect variables are well-formed in the scopes in which they appear.

7.5. Solving constraints

To solve a generated constraint system $\bar{C}$, we start with the minimal assignment $\perp$ and iteratively increase this assignment via the relation $A \rightarrow_{\bar{C}} A'$:

$$\frac{(r \sqsubseteq \varphi) \in \bar{C}}{A \rightarrow_{\bar{C}} A[\varphi := A(\varphi) \sqcup [\![A(r)]\!]]}$$

If this rule is applied to some $r \sqsubseteq \varphi \in \bar{C}$ not satisfied by the assignment A, *i.e.* $[\![A(r)]\!] \not\sqsubseteq A(\varphi)$, it produces a larger assignment A' that does satisfy that constraint. If applied to some constraint that is already satisfied by A, the assignment is not changed.

We repeatedly apply this rule until a fixed point is reached. Specifically, we compute $\perp \rightarrow^*_{\bar{C}} A$ by iteratively applying the single-step rule until an A is reached where there does not exist an A' such that $A \rightarrow_{\bar{C}} A'$ and $A \neq A'$. We show below that the fixed point algorithm always terminates and that, when a fixed point A is reached, it exhibits two key properties:

1. If $A \models \bar{C}$, then we have found a solution for $\bar{C}$ and the program $A(P)$ is well-typed.
2. If $A \not\models \bar{C}$, then there exists a constraint $(r \sqsubseteq a) \in \bar{C}$ in $\bar{C}$ where $[\![A(r)]\!] \not\sqsubseteq a$. In this case, A is not a solution for $\bar{C}$ and the program $A(P)$ is not well-typed. Further, the program is not well-typed under any assignment.

Thus, the algorithm finds a solution for $\bar{C}$ if and only if a solution exists.

```
1  class Bit {
2
3    boolean b;
4
5    φ1 boolean get() {
6      synchronized(this) {
7        return this.b;
8      }
9    }
10
11   φ2 void set(boolean t) {
12     synchronized(this) {
13       this.b = t;
14     }
15   }
16
17   φ3 boolean testAndSet() {
18     synchronized(this) {
19       if (!this.get()) {
20         this.set(true);
21         return true;
22       } else {
23         return false;
24       }
25     }
26   }
27
28   φ4 void busyWait() {
29     while (!this..testAndSet()) { }
30   }
31 }
32
33 class Client {
34   final Bit flag;
35
36   φ5 void f() {
37     flag.set(false);
38     flag.busyWait#();
39     flag..set(false);
40   }
41 }
```

Fig. 17. A version of `Bit` and a client, both of which include effect variables.

For the constraints in Fig. 18, our algorithm yields the minimal solution below. We also express those effects in the more readable surface-level syntax from Section 5.

`Bit.get:`	$\varphi_1 = (\texttt{this}\,?\,\text{AM}:\text{AN})$	`= this ? mover : atomic`
`Bit.set:`	$\varphi_2 = (\texttt{this}\,?\,\text{AM}:\text{AN})$	`= this ? mover : atomic`
`Bit.testAndSet:`	$\varphi_3 = (\texttt{this}\,?\,\text{AM}:\text{AN})$	`= this ? mover : atomic`
`Bit.busyWait:`	$\varphi_4 = (\texttt{this}\,?\,\text{CY}:\text{CL})$	`= this ? (compound yield) : (compound left)`
`Client.f:`	$\varphi_5 = (\texttt{this}\,?\,\text{CY}:\text{CN})$	`= this ? (compound yield) : compound`

Note that the algorithm yields more precise effects for `testAndSet` and `busyWait` than those discussed for the original version of `Bit` in Fig. 5, because it assumes that the lock on a `Bit` object may be held on calls to those objects. As mentioned earlier, the programmer may choose to either retain these precise effects or coarsen them to more intuitive or suitable effects, such as `atomic` (AN) and `compound` (CN), respectively, in these cases.

7.6. Correctness of the algorithm

We now show that the inference algorithm always terminates and is correct:

$$\begin{array}{rll}
\mathbf{Lift}(P, E_1, \mathbf{S}(\texttt{this}, \text{“}\texttt{.}\text{”}, (\textsc{af};\textsc{am}))) \sqsubseteq \varphi_1 & & [\textsc{inf method}], \text{line } 5 \\
\mathbf{Lift}(P, E_2, \mathbf{S}(\texttt{this}, \text{“}\texttt{.}\text{”}, (\textsc{af};\textsc{af};\textsc{am}))) \sqsubseteq \varphi_2 & & [\textsc{inf method}], \text{line } 11 \\
\mathbf{Lift}(P, E_3, \varphi_1 \cdot [\texttt{this} := \texttt{this}]) \sqsubseteq \textsc{an} & & [\textsc{inf call atomic}], \text{line } 19 \\
\mathbf{Lift}(P, E_3, \varphi_2 \cdot [\texttt{this} := \texttt{this}]) \sqsubseteq \textsc{an} & & [\textsc{inf call atomic}], \text{line } 20 \\
\mathbf{Lift}(P, E_3, \mathbf{S}(\texttt{this}, \text{“}\texttt{.}\text{”}, (\textsc{af};(\textsc{af};\textsc{af};\mathbf{Lift}(P, E_3, \varphi_1 \cdot [\texttt{this} := \texttt{this}]))); & & \\
((\textsc{af};\textsc{af};\textsc{af};\mathbf{Lift}(P, E_3, \varphi_2 \cdot [\texttt{this} := \texttt{this}]);\textsc{af}) \sqcup \textsc{af}))) \sqsubseteq \varphi_3 & & [\textsc{inf method}], \text{line } 17 \\
\mathbf{Lift}(P, E_4, \varphi_3 \cdot [\texttt{this} := \texttt{this}]) \sqsubseteq \textsc{an} & & [\textsc{inf call atomic}], \text{line } 29 \\
\mathbf{Lift}(P, E_4, (\textsc{af};(\textsc{af};\textsc{cy};\mathbf{Lift}(P, E_4, \varphi_3 \cdot [\texttt{this} := \texttt{this}]))); & & \\
(\textsc{af};(\textsc{af};(\textsc{af};\textsc{cy};\mathbf{Lift}(P, E_4, \varphi_3 \cdot [\texttt{this} := \texttt{this}]))))^{*}) \sqsubseteq \varphi_4 & & [\textsc{inf method}], \text{line } 28 \\
\mathbf{Lift}(P, E_5, \varphi_2 \cdot [\texttt{this} := \texttt{this.flag}]) \sqsubseteq \textsc{an} & & [\textsc{inf call atomic}], \text{lines } 37 \& 39 \\
\mathbf{Lift}(P, E_5, ((\textsc{af};\textsc{af});\textsc{af};\textsc{af};\mathbf{Lift}(P, E_5, \varphi_2 \cdot [\texttt{this} := \texttt{this.flag}])); & & \\
((\textsc{af};\textsc{af});\textsc{af};\mathbf{Lift}(P, E_5, \varphi_2 \cdot [\texttt{this} := \texttt{this.flag}])); & & \\
((\textsc{af};\textsc{af});\textsc{af};\textsc{cy};\mathbf{Lift}(P, E_5, \varphi_2 \cdot [\texttt{this} := \texttt{this.flag}]))) \sqsubseteq \varphi_5 & & [\textsc{inf method}], \text{line } 36
\end{array}$$

where

$E_1 = \texttt{Bit this}$
$E_2 = \texttt{Bit this, boolean t}$
$E_3 = \texttt{Bit this}$
$E_4 = \texttt{Bit this}$
$E_5 = \texttt{Client this}$

Fig. 18. Constraints generated for the `Bit` program in Fig. 17, and the typing rule that generated each one.

Theorem 7.2 *(Method specification inference). Given program P and constraints $\bar{C}$ such that $P \vdash \bar{C}$:*

1. *The transitive closure of $\rightarrow_{\bar{C}}$ applied to $\bot$ always reaches a fixed point.*
2. *If $\exists A'.\ A'(P) \vdash wf$ and $\bot \rightarrow^{*}_{\bar{C}} A$ where A is a fixed point, then $A \models \bar{C}$.*
3. *If $\bot \rightarrow^{*}_{\bar{C}} A$ and A is a fixed point where $A \models \bar{C}$, then $A(P) \vdash wf$.*

We begin by showing that our algorithm always terminates. This property, and those that follow, only hold for constraints generated by the type inference rules, which generate constraint sets exhibiting several key features that we highlight below. A crucial aspect of showing termination is ensuring that the constraint solver does not produce arbitrarily large lock expressions, which would in turn lead to infinite ascending chains in the assignment lattice. We bound the size of lock expressions appearing in A to exclude this possibility. A lock expression l is *bounded* if $|l| < MaxLockSize$. We extend that notion of bounded size to lock expressions appearing in effects and assignments as well. A cooperability expression or constraint is bounded if it is only conditional on bounded lock expressions, and if every delayed substitution occurs inside the construct $\mathbf{Lift}(P, E, \cdot)$.

Lemma 7.3. *If r is a closed, bounded cooperability expression then $[\![r]\!]$ is also bounded.*

Proof. The only difficulty is that $[\![r]\!]$ may apply delayed substitutions in r, resulting in non-bounded lock expressions. However, all delayed substitutions are within an enclosing $\mathbf{Lift}(P, E, \cdot)$ construct that will filter out these non-bounded lock expressions. $\square$

Lemma 7.4. *If A and r are bounded then $A(r)$ is bounded.*

Lemma 7.5 *(Termination). For any bounded constraint system $\bar{C}$, the transitive closure of $\rightarrow_{\bar{C}}$ applied to $\bot$ always reaches a fixed point.*

Proof. Since $\bar{C}$ is bounded, every generated assignment is also bounded by the previous Lemmas. Any program, and thus any generated $\bar{C}$, contains only a finite number of distinct variables and field names. All bounded lock expressions in

the generated bounded assignments are derived from these names. Thus there are a finite number of possible bounded assignments. Also note that if $A \rightarrow^*_{\bar{C}} A'$ then $A \sqsubseteq A'$. Since the relation is non-decreasing and there are a finite number of bounded assignments, we must reach a fixed point. □

Part 1 of Theorem 7.2 follows directly from Lemma 7.5. Since the type inference rules only generate bounded constraint systems, the algorithm will terminate for any set of constraints generated while checking a program.

We now turn our attention to showing that the constraint solver finds a valid solution if and only if the program is well-typed. We first state several key properties about the relation $A \rightarrow_{\bar{C}} A'$.

Lemma 7.6 *(Step). Suppose $A \rightarrow_{\bar{C}} A'$. Then $A \sqsubseteq A'$. If in addition there exists A'' such that $A \sqsubseteq A''$ and $A'' \models \bar{C}$, then we also have that $A' \sqsubseteq A''$.*

Lemma 7.7 *(Contradiction). If A is a fixed point for $\rightarrow_{\bar{C}}$ and $A \not\models \bar{C}$, then there is no A' such that $A \sqsubseteq A'$ and $A' \models \bar{C}$.*

Also, the algorithm only computes well-formed assignments. (Recall that well-formed assignments do not cause variables to be referenced out of their scopes.)

Lemma 7.8 *(Well-formed). If A is well-formed for $\bar{C}$ and $A \rightarrow_{\bar{C}} A'$ then A' is well-formed for $\bar{C}$.*

Proof. The type inference rules produce constraints $\bar{C}$ exhibiting the following two properties:

1. Each $C \in \bar{C}$ has one of two forms: $(r \sqsubseteq a)$ or $(\textbf{Lift}(P, E, r) \sqsubseteq \varphi)$.
2. There is at most one lower bound $(\textbf{Lift}(P, E, r) \sqsubseteq \varphi)$ in $\bar{C}$ for each φ.

If $A \rightarrow_{\bar{C}} A'$, then $A' = A[\varphi := A(\varphi) \sqcup [\![A(\textbf{Lift}(P, E, r))]\!]]$, where $(\textbf{Lift}(P, E, r) \sqsubseteq \varphi) \in \bar{C}$ is the unique lower bound on φ. Since A is well-formed, $A(P); E \vdash A(\varphi)$. Also, $[\![A(\textbf{Lift}(P, E, r))]\!] = a$, where $A(P); E \vdash [\![A(r)]\!] \uparrow a$. This implies that $A(P); E \vdash a$. The effect $A'(\varphi) = A(\varphi) \sqcup a$ will also be well-formed in E, since it can only refer to lock expressions already present in $A(\varphi)$ and a. Thus, A' is well-formed for $\bar{C}$. □

The previous lemmas are sufficient to show that the algorithm only produces correct results, and Lemma 7.6 also guarantees that the algorithm computes the most precise satisfying assignment.

The second part of Theorem 7.2 follows from Lemmas 7.6 and 7.7. To show the third part, suppose $\bot \rightarrow^*_{\bar{C}} A$ and A is a fixed point where $A \models \bar{C}$. Since $\bot$ is well-formed for $\bar{C}$ and $\bot \rightarrow_{\bar{C}} \ldots \rightarrow_{\bar{C}} A$, Lemma 7.8 states that A is well-formed for $\bar{C}$. Theorem 7.1 then concludes that $A(P) \vdash wf$.

8. Implementation and evaluation of method specification inference

We have extended the JCC checker to perform method specification inference using the algorithm described in the previous section. JCC thus takes Java source code with yield and field annotations, but it does not require all methods to be annotated with their cooperability specifications. If all methods are annotated, JCC simply verifies that the code is cooperable and that the annotations are correct.

If some or all method annotations are missing, JCC inserts fresh effect variables in their place and proceeds to check the program via our constraint-based inference algorithm. If a solution for the effect variables is found, JCC outputs a fully-annotated version of the source. Otherwise, JCC reports a warning for each cooperability error identified.

Our implementation of the specification inference algorithm follows the formal development closely. We do, however, include on-the-fly simplification of computed effects and an additional constraint form to enable support for the inference of specifications in the case of overriding method declarations. These two extensions were previously developed as part of an earlier type system for atomicity [18] and are straight-forward to adapt to this setting. We configured *MaxLockSize* to bound lock expressions to contain no more than four field accesses. Larger values of *MaxLockSize* increased run times with no precision improvements, and smaller bounds degraded precision. Overall, the performance of the specification inference algorithm is quite fast.

Applying our inference algorithm to the previously presented benchmarks enabled us to verify cooperability without adding any method annotations. That is, our inference algorithm inferred an appropriate effect for every method in the target program. As described earlier in Table 3, our inference algorithm reduces the number of necessary manually-inserted annotations to 21 per KLOC in our benchmarks, significantly improving the usability of JCC.

In addition to removing the need for method specifications, a another benefit of the inference algorithm is that it facilitates the process of inserting yield annotations, since errors during the inference process typically reflect missing yields in the target program.

Table 4
Method specification inference performance.

Program	Size (lines)	$\lvert\bar{C}\rvert$	Time (sec)
java.util.zip.Inflater	317	40	0.07
java.util.zip.Deflater	381	42	0.06
java.lang.StringBuffer	1276	87	0.10
java.lang.String	2308	103	0.12
java.io.PrintWriter	534	97	0.19
java.util.Vector	1019	144	0.19
java.util.zip.ZipFile	490	66	0.18
sparse	868	90	0.14
tsp	706	52	0.14
elevator	1447	150	0.17
raytracer-fixed	1915	237	0.18
sor-fixed	958	96	0.14
moldyn-fixed	1352	116	0.90
Total	13,571	1320	2.58
per KLOC		97	0.19

The third column in Table 4 shows the number of constraints generated for each target program, and the fourth shows the JCC running time. The run times were quite good, with JCC configured to use method specification inference still taking less than a second on each program.

Since it is guaranteed to find the most precise method effects possible, JCC in some cases annotated methods with effects more precise than a programmer may assign, as highlighted in Section 7.5. One possible way to mitigate this in the future if it proves problematic would be to have the algorithm attempt to infer effects from the most widely-used subset of the effect lattice first, only reverting to using the full effect lattice if that fails.

9. Related work

Cooperative multithreading is a thread execution model in which context switches between threads may occur only at `yield` statements [3,4,8]. That is, cooperative multithreading permits concurrency but disallows parallel execution of threads. In contrast, cooperability guarantees behavior equivalent to cooperative multithreading, but actually allows execution in a preemptive manner, enabling full use of modern multicore hardware.

Automatic mutual exclusion (AME) is an execution model ensuring mutual exclusion by default [28]; `yield` statements demarcate where thread interference is permitted. A key difference is that AME enforces serializability at run time via transactional memory techniques; in contrast, JCC guarantees serializability statically. *Observationally Cooperative Multithreading* similarly uses run-time systems to safely run threads with explicit yields in parallel [52].

The Panini project [32,44] explores an alternative approach to minimizing thread interference points in the design of a new programming language. This language supports both objects and capsules, where each object is local to a capsule, and at most one thread is active in each capsule at a time. Thus, thread interference can only happen when one capsule calls a procedure of a different capsule.

In prior work, we explored a simpler type system for cooperability [58], and dynamic analyses for checking cooperability and inferring yield annotations for legacy programs [59]. Others have explored *task types*, a data-centric approach to obtaining *pervasive atomicity* [30], a notion closely related to cooperability. Atomic sets are a useful, but complementary, technique for specifying groups of fields that must always be updated atomically. In contrast to cooperability, that approach enforces atomicity requirements by automatically inserting synchronization operations.

There is extensive literature on how to find and fix data races efficiently. Dynamic detectors may track the happens-before relation [17], implement the lockset algorithm [50], or combine both [39]. Static race detectors may make use of a type system [9,1], implement a static lockset algorithm [38,42], or use model checking [43]. Data races often reflect problems in synchronization, and expose a weak memory model to programmers, compromising software reliability, and race freedom remedies this issue by guaranteeing behavior equivalent to executing with sequentially consistent memory [2].

Atomicity is an analysis approach that checks if atomic blocks are serializable. Both static [18,25,54] and dynamic tools [15,55,19,14] have been developed to check atomicity, as well as transactional memory techniques that enforce serializability at run time [26,12,51,24]. While the notion of atomicity is very beneficial when reasoning about atomic methods, it is less helpful in documenting thread interference in non-atomic methods [58]. Moreover, atomicity introduces a form of bimodal reasoning, combining sequential reasoning inside atomic blocks with traditional multithreaded reasoning outside atomic blocks. In contrast, cooperative concurrency provides a uniform semantics for reasoning about the correctness of all code in a multithreaded system.

Deterministic parallelism generalizes atomicity to guarantee that the result of executing multiple threads is invariant across thread schedules. There are various approaches to ensure deterministic parallelism, including static analyses [7], dynamic analyses [48,10], and run-time enforcement [41,11]. One interesting area for future work is to extend our system to reason about determinism for cooperative programs.

10. Summary

Reasoning about multithreaded software correctness is notoriously difficult under the preemptive semantics provided by multicore architectures. This paper proposes an approach where the programmer writes software with traditional synchronization idioms, and also explicitly documents intended sources of thread interference with yield annotations. Any annotated program verified by our type system behaves *as if* it is executing under a cooperative semantics where context switches between threads happen only at specified yield points.

This cooperative semantics provides a nicer foundation for reasoning about program behavior and correctness. In particular, intuitive sequential reasoning is now valid, except at yield annotations, and prior user studies have shown that yield annotations makes it significantly easier for programmers to identify defects during code reviews [49].

An important problem for future work is developing an inference system for identifying where to insert yield annotations into existing, unannotated program. That would complement our work on method specification inference and dramatically reduce the overhead of using our technique. Other interesting avenues for future work are to incorporate cooperative reasoning into a proof system, such as rely-guarantee reasoning, or to extend cooperability to reason about determinism properties.

Acknowledgements

This work was supported by the National Science Foundation under grants 1116883, 1116825, 1337278, 1421051, 1421016, and 1439042.

Appendix A. Formal semantics of YieldJava

Syntax extensions for run-time terms To characterize the run-time behavior of YieldJava programs, we extend the syntax of expressions to include object addresses ρ and the `in-sync` construct, as shown in Fig. A.1. We assume that addresses are divided into distinct sets, one for each type of object. The membership relation $\rho \in Addr_c$ indicates that the object at address ρ is of type c. The expression `in-sync` ρ e describes an expression e that is executing while holding the lock ρ.

Run-time semantics for YieldJava A run-time state $\sigma = \langle P, H, M\rangle$ is a combination of a program P, an object map H, and a thread map M. An *object map* H is a partial map from addresses to objects. An object is a sequence of field values for a given class, along with a lock o indicating which thread is holding the lock for that object, if any. (As in Java, every object has a lock implicitly associated with it at run time.) A *thread map* M is a partial map from threads to thread state. A thread state T is either:

- an expression e,
- **ready** e, which describes a newly created thread ready to execute an expression e, or
- **wrong**, which occurs when a thread tries to dereference `null`.

Fig. A.2 presents the formal transition rules for the evaluation of YieldJava programs. Each rule describes a transition step $\rightarrow_t$ that could be taken by the thread t. We use an evaluation context $\mathcal{E}$, an expression with a “hole” [ ], to indicate the next subexpression to evaluate. No transition rule applies for a thread t if that thread's expression is either a value or **wrong**.

For simplicity, we adopt the following notation to access constituent parts of a state. If $\sigma = \langle P, H, M\rangle$, then:

- $\sigma[c]$ refers to the class definition ‘`class c { ... }`’ for the class named c in P,
- $\sigma[\rho]$ refers to the object stored at address ρ in H, that is, $H[\rho]$, and
- $\sigma[t]$ refers to the thread state for thread t in M, that is, $M[t]$.

Updates to the state are similarly written as follows:

- The state $\sigma[\rho \mapsto \{\overline{f = v}\}^o]$ is the same as σ except ρ is mapped to the given object state. Similarly, $\sigma[\rho.f \mapsto v]$ preserves all components of σ, except for the single updated field.
- The state $\sigma[t \mapsto T]$ is the same as σ, except thread t maps to T.

We define two semantics using the single-step transition rules. In the preemptive semantics $\rightarrow$, the steps of the various threads are interleaved non-deterministically, and a thread can perform a step at any time (provided that thread is not blocked). This nondeterminism means that reasoning about the behavior of programs under the preemptive semantics is very difficult.

In the cooperative semantics $\rightarrow_c$, context switches between threads can happen only at yield operations, which are indicated by dots (“..”), or on thread termination. In more detail, a thread t is *yielding in* σ (or simply *yielding* if the context is clear) if for $\sigma(t) = T$,

$$
\begin{array}{rcll}
e & \in & \textit{Expr} & ::= \cdots \mid \texttt{in-sync}\ \rho\ e \\
v & \in & \textit{Value} & ::= \rho \mid \texttt{null} \\
\rho & \in & \textit{Addr} & = \bigcup_{c \,\in\, \textit{ClassName}} \textit{Addr}_c \\
\sigma & \in & \textit{State} & = \textit{Program} \times (\textit{Addr} \rightharpoonup \textit{Object}) \times (\textit{Thread} \rightharpoonup \textit{ThreadState}) \\
\textit{obj} & \in & \textit{Object} & ::= \{\, \overline{f = v}\, \}^{o} \\
o & \in & \textit{Lock} & ::= \bot \mid \textit{Thread} \\
T & \in & \textit{ThreadState} & ::= e \mid \textbf{ready}\ e \mid \textbf{wrong} \\
t & \in & \textit{Thread} & = \{1, 2, 3, \ldots\}
\end{array}
$$

Fig. A.1. YIELDJAVA run-time constructs and state.

Evaluation contexts

$$
\begin{array}{rl}
\mathcal{E} ::= & [\,] \mid \texttt{new}\, c\, (\overline{v}, \mathcal{E}, \overline{e}) \mid \mathcal{E}_\gamma f \mid \mathcal{E}_\gamma f\ \texttt{=}\ e \mid v_\gamma f\ \texttt{=}\ \mathcal{E} \\
\mid & \mathcal{E}_\gamma m\, (\overline{e}) \mid \mathcal{E}_\gamma m\#\, (\overline{e}) \mid v_\gamma m\, (\overline{v}, \mathcal{E}, \overline{e}) \mid v_\gamma m\#\, (\overline{v}, \mathcal{E}, \overline{e}) \\
\mid & \texttt{let}\ x\ \texttt{=}\ \mathcal{E}\ \texttt{in}\ e \mid \texttt{if}\ \mathcal{E}\ e\ e \mid \mathcal{E}_\gamma \texttt{sync}\ e \mid \texttt{in-sync}\ \rho\ \mathcal{E}
\end{array}
$$

Transition rules

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma f]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[v]] \ \text{ if } \sigma(\rho) = \{\cdots, f = v, \cdots\}^{o} \qquad \text{[RED READ]}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma f\ \texttt{=}\ v]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[v], \rho.f \mapsto v] \qquad \text{[RED WRITE]}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma m\, (v_{1..n})\,]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[e[\texttt{this} := \rho, x_i := v_i{}^{i \in 1..n}]]] \qquad \text{[RED INVOKE]}$$
$$\text{if } \rho \in \textit{Addr}_c \text{ and } \sigma[c] = \texttt{class}\ c\ \texttt{\{}\ \cdots\ a\, d'\, m\, (\overline{d\, x})\ \texttt{\{}\ e\ \texttt{\}}\ \cdots\ \texttt{\}}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma m\#\, (v_{1..n})\,]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[e[\texttt{this} := \rho, x_i := v_i{}^{i \in 1..n}]]] \qquad \text{[RED INVOKE COMPOUND]}$$
$$\text{if } \rho \in \textit{Addr}_c \text{ and } \sigma[c] = \texttt{class}\ c\ \texttt{\{}\ \cdots\ a\, d'\, m\, (\overline{d\, x})\ \texttt{\{}\ e\ \texttt{\}}\ \cdots\ \texttt{\}}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{new}\, c(v_{1..n})]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[\rho], \rho \mapsto \{\, f_i = v_i{}^{\,i \in 1..n}\, \}^{\bot}] \qquad \text{[RED NEW]}$$
$$\text{if } \rho \notin \textit{dom}(\sigma) \text{ and } \rho \in \textit{Addr}_c \text{ and } \sigma[c] = \texttt{class}\ c\ \texttt{\{}\ \overline{t\ f}\ \cdots\ \texttt{\}}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,], \rho \mapsto \{\, \overline{f = v}\, \}^{\bot}] \ \to_t\ \sigma[t \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\ e], \rho \mapsto \{\, \overline{f = v}\, \}^{t}] \qquad \text{[RED SYNC]}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,], \rho \mapsto \{\, \overline{f = v}\, \}^{t}] \ \to_t\ \sigma[t \mapsto \mathcal{E}[e], \rho \mapsto \{\, \overline{f = v}\, \}^{t}] \qquad \text{[RED SYNC REENTRANT]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\ v], \rho \mapsto \{\, \overline{f = v}\, \}^{t}] \ \to_t\ \sigma[t \mapsto \mathcal{E}[v], \rho \mapsto \{\, \overline{f = v}\, \}^{\bot}] \qquad \text{[RED IN-SYNC]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{fork}\, e]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[\texttt{null}], t' \mapsto \textbf{ready}\ e] \ \text{ if } t' \notin \textit{Dom}(\sigma) \qquad \text{[RED FORK]}$$

$$\sigma[t \mapsto \textbf{ready}\ e] \ \to_t\ \sigma[t \mapsto [e]] \qquad \text{[RED READY]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[e[x := v]]] \qquad \text{[RED LET]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{if}\ v\ e_2\ e_3]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[e_2]] \ \text{ if } v \neq \texttt{null} \qquad \text{[RED IF-NONNULL]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{if}\ \texttt{null}\ e_2\ e_3]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[e_3]] \qquad \text{[RED IF-NULL]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{while}\, e_1\, e_2]] \ \to_t\ \sigma[t \mapsto \mathcal{E}[\texttt{if}\ e_1\ (e_2\texttt{;}\ \texttt{while}\ e_1\ e_2)\ \texttt{null}]] \qquad \text{[RED WHILE]}$$

$$\sigma[t \mapsto \mathcal{E}[e]] \ \to_t\ \sigma[t \mapsto \textbf{wrong}] \qquad \text{[RED WRONG]}$$
$$\text{if } e \in \{\texttt{null}_\gamma f,\ \texttt{null}_\gamma f\ \texttt{=}\ v,\ \texttt{null}_\gamma m\, (\overline{v}),\ \texttt{null}_\gamma m\#\, (\overline{v}),\ \texttt{null}_\gamma \texttt{sync}\ e'\}$$

Transition relations

(*preemptive semantics*) $\sigma \ \to\ \sigma'$ if $\sigma \to_t \sigma'$

(*cooperative semantics*) $\sigma \ \to_c\ \sigma'$ if $\sigma \to_t \sigma'$ and $\sigma(t)$ is cooperatively enabled

Fig. A.2. YIELDJAVA semantics.

$\boxed{P; E \vdash e : c \cdot a}$

[EXP ADDR]
$$\frac{P \vdash E \quad E = E_1, c\ \rho, E_2}{P; E \vdash \rho : c \cdot \text{AF}}$$

[EXP INSYNC]
$$\frac{P; E \vdash_{\text{lock}} \rho \quad P; E \vdash e : c \cdot a}{P; E \vdash \texttt{in-sync}\ \rho\ e : c \cdot SI(\rho, a)}$$

$\boxed{P \vdash E}$

[ENV ADDR]
$$\frac{P \vdash E \quad \rho \notin dom(E) \quad P \vdash c \quad \rho \in Addr_c}{P \vdash (E, c\ \rho)}$$

$\boxed{P; E \vdash T}$

[THREAD OK]
$$\frac{P; E \vdash e : c \cdot a}{P; E \vdash e}$$

[THREAD READY]
$$\frac{P; E \vdash e : c \cdot a}{P; E \vdash \textbf{ready}\ e}$$

[THREAD WRONG]
$$\frac{}{P; E \vdash \textbf{wrong}}$$

$\boxed{P; E \vdash obj : c}$

[OBJECT]
$$\frac{\texttt{class}\ c\ \{\ d_i\ f_i\ ^{i \in 1..n}\ \ldots\ \} \in P \quad P; E \vdash v_i : d_i \cdot a_i \quad \forall i \in 1..n}{P; E \vdash \{\ f_i = v_i\ ^{i \in 1..n}\ \}^o : c}$$

$\boxed{\vdash \sigma}$

[STATE]
$$\frac{\begin{array}{ll} P \vdash \text{OK} & \\ E = c_1\ \rho_1, \ldots, c_m\ \rho_m & \\ P \vdash E & \\ P; E \vdash obj_j : c_j & \forall j \in 1..m \\ P; E \vdash T_k & \forall k \in 1..n \end{array}}{\vdash \langle P, [\ \rho_j \mapsto obj_j\ ^{j \in 1..m}\ ], [\ t_k \mapsto T_k\ ^{k \in 1..n}\ ] \rangle}$$

Fig. A.3. YieldJava type rules for run-time terms.

1. $T = \textbf{ready}\ e$
2. $T \in Value$,
3. $T = \textbf{wrong}$,
4. $T = \mathcal{E}[\rho\,.\,.\,f\,]$,
5. $T = \mathcal{E}[\rho\,.\,.\,f = v]$,
6. $T = \mathcal{E}[\rho\,.\,.\,\texttt{sync}\ e\,]$,
7. $T = \mathcal{E}[\rho\,.\,.\,m\,(\,\overline{v}\,)\,]$, or
8. $T = \mathcal{E}[\rho\,.\,.\,m\#\,(\,\overline{v}\,)\,]$.

The thread t is *cooperatively enabled* in a state σ if for all t' in $Dom(\sigma) \setminus \{t\}$, we have t' is yielding in σ. Thus, t can only take a cooperative step if all other threads are yielding. This means that if t takes a cooperative step, then every other thread is either at a yield operation, is not enabled (either terminated at a value or **wrong**), or has not started yet. A state is *yielding* if all threads are yielding in that state.

Type system extensions for run-time terms The additional typing rules shown in Fig. A.3 extend the YieldJava type system to include run-time states. Environments are also extended to include run-time object addresses:

$$E ::= \epsilon \mid E, c\ x \mid E, c\ \rho$$

The rule [State] ensures all three components of a state are well-formed. This rule constructs an environment E containing addresses for all objects in the object map and verifies that each object is well-typed via [Object].

We similarly verify that each thread state is well-formed via the rules for judgment form $P; E \vdash T$. If a thread state is an expression e or the construct **ready** e, then rules [Thread Ok] and [Thread Ready] simply check that e is a well-formed expression. If a thread state is **wrong**, rule [Thread Wrong] concludes that the state is well-formed.

The most important new rule for expressions is the rule [exp insync] for `in-sync` $\rho\ e$. That rule captures the cooperability effect of evaluating e with the lock ρ held followed by releasing ρ. (That is, it captures the effect of running the remainder of a critical section on ρ up to and including the release at the end of the section.) Given the effect a of e, rule [exp insync] uses the following function to determine the overall effect of the term:

a	$SI(\rho, a)$
κ	κ; AL
$\rho\,?\,a_1 : a_2$	$SI(\rho, a_1)$
$\rho'\,?\,a_1 : a_2$	$\rho'\,?\,SI(\rho, a_1) : SI(\rho, a_2)$ if $\rho \neq \rho'$

Briefly, if a is a combined effect κ, then the overall effect is κ sequentially followed by AL, the effect for a lock release operation. If a is an effect conditional on holding lock ρ, then we simplify a by considering only the conditional branch of a reflecting its behavior when ρ is held. However, if a is a conditional effect on a different lock ρ', then we recursively simplify both conditional branches of a.

Appendix B. Cooperative-preemptive equivalence

In this appendix, we prove that for a well-typed program, the preemptive and cooperative semantics coincide: if a program, starting from a yielding state, can reach a yielding state σ under $\rightarrow$, then it can reach σ under $\rightarrow_c$ as well.

The proof depends on the following reduction theorem [18]. This theorem is stated in terms of an arbitrary transition system, and requires some additional notation: For any state predicate $S \subseteq \mathit{State}$ and transition relation $T \subseteq \mathit{State} \times \mathit{State}$, the *left restriction* of T to S, or S/T, is the transition relation T with each pair's first component contained in S. Similarly, the *right restriction* of T to S, or $T \backslash S$, is the transition relation T with each pair's second component contained in S. The *composition* of two transition relations T and U, or $T \circ U$, is the set of all transitions (p, r) such that there exists a state q with transitions $(p, q) \in T$ and $(q, r) \in U$. For transition relations T and U, we say that T *right-commutes* with U, and U *left-commutes* with T, if we have $T \circ U \subseteq U \circ T$.

Each thread executes in a series of transactions, each of which consists of a sequence of right-movers, followed by at most one non-mover, followed by a sequence of left-movers. A thread may also go wrong at any point. We use the following three predicates to describe the current state of each thread i:

- $\mathcal{R}_i$ is the set of states where the thread i is in the right-mover part of some transaction;
- $\mathcal{L}_i$ is the set of states where the thread i is in the left-mover part of some transaction; and
- $\mathcal{W}_i$ is the set of states where the thread i has gone wrong.

Those sets are used in the reduction theorem below that relates the following three transition relations:

- $\hookrightarrow_i$ is the transition relation that describes the behavior of the thread i.
- $\hookrightarrow$ is the transition relation where, at each state, one may choose a thread i and use the transition $\hookrightarrow_i$.
- $\hookrightarrow_c$ is the transition relation that describes the serial behavior of a program, in which at most one thread may be in a transaction at any time. (All others would thus be completed or at yield points.)

The following reduction theorem then shows that if the transition relation $\hookrightarrow$ and the sets $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ satisfy certain constraints, then each transaction is serializable, that is, that the standard semantics ($\hookrightarrow$) and the serialized semantics ($\hookrightarrow_c$) are essentially equivalent. The necessary conditions are:

1. A thread i can be in at most one of $\mathcal{R}_i$, $\mathcal{L}_i$, or $\mathcal{W}_i$.
2. A step by thread i cannot cause a transition from $\mathcal{L}_i$ to $\mathcal{R}_i$. Thus a thread never changes from being in the left-mover part of an transaction to being in the right-mover part.
3. No steps are possible for thread i once it enters $\mathcal{W}_i$.
4. Steps by distinct threads i and j are disjoint, meaning that steps by different threads cannot have the same overall effect on program state.
5. A step taken by thread i while in $\mathcal{R}_i$ right-commutes with steps taken by any other thread j.
6. A step taken by thread i while in $\mathcal{L}_i$ left-commutes with steps taken by any other thread j.
7. A step taken by thread i cannot change whether any another thread j is in $\mathcal{R}_j$, $\mathcal{L}_j$, or $\mathcal{W}_j$. That is, a thread cannot affect the which partition any other thread currently belongs to.

Provided these conditions are met, if a program in state p in which no threads are in transactions evaluates to a similar state q, then q can also be reached by an execution in which context switches only occur outside transactions (at yield points). Moreover, if a wrong state q is reachable from p, where no thread in q is in the left-mover part of a transaction, then a wrong state is also reachable from p under the cooperative semantics.

Theorem B.1 *(Reduction Theorem). For all threads i, let $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ be sets of states, and $\hookrightarrow_i$ be a transition relation. Suppose for all i that the following is true:*

1. *$\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ are pairwise disjoint,*
2. *$(\mathcal{L}_i / \hookrightarrow_i \backslash \mathcal{R}_i)$ is false,*
3. *$\mathcal{W}_i / \hookrightarrow_i$ is false,*

and for all $j \neq i$,

4. *$\hookrightarrow_i$ and $\hookrightarrow_j$ are disjoint,*
5. *$(\hookrightarrow_i \backslash \mathcal{R}_i)$ right-commutes with $\hookrightarrow_j$,*
6. *$(\mathcal{L}_i / \hookrightarrow_i)$ left-commutes with $\hookrightarrow_j$, and*
7. *if $p \hookrightarrow_i q$, then $\mathcal{R}_j(p) \Leftrightarrow \mathcal{R}_j(q)$, $\mathcal{L}_j(p) \Leftrightarrow \mathcal{L}_j(q)$, and $\mathcal{W}_j(p) \Leftrightarrow \mathcal{W}_j(q)$.*

We define $\mathcal{N}_i$, $\mathcal{N}$, $\mathcal{W}$, $\hookrightarrow$, and $\hookrightarrow_c$ as follows:

- $\mathcal{N}_i = \neg(\mathcal{R}_i \vee \mathcal{L}_i)$
- $\mathcal{N} = \forall i.\, \mathcal{N}_i$
- $\mathcal{W} = \exists i.\, \mathcal{W}_i$
- $\hookrightarrow = \exists i.\, \hookrightarrow_i$
- $\hookrightarrow_c = \exists i.\, ((\forall j \neq i.\, \mathcal{N}_j)/\hookrightarrow_i)$

Now suppose $p \in \mathcal{N}$ and $p \hookrightarrow^* q$. Then the following statements are true.

1. *If $q \in \mathcal{N}$, then $p \hookrightarrow_c^* q$.*
2. *If $q \in \mathcal{W}$ and $\forall i.\, q \notin \mathcal{L}_i$, then $\exists q'.\, p \hookrightarrow_c^* q'$ and $q' \in \mathcal{W}$.*

Proof. Refer to [21]. □

We now turn our attention to the specific case of YieldJava. Consider a fixed program P. Any well-typed state σ with an object map $[\rho_j \mapsto obj_j{}^{j \in 1..m}]$ has a corresponding object environment $E_\sigma = c_1\ \rho_1, \ldots, c_n\ \rho_n$ where $\rho_j \in Addr_{c_j}$.

Given a state σ, if $\sigma(i) = e$, then we define the effect $\alpha(\sigma, e)$ to be a, where $P; E_\sigma \vdash e : c \cdot a$. An examination of the typing rules demonstrates that α is a well-defined partial function.

Let WT be the set of well-typed states for P:

$$WT = \{\sigma \mid \vdash \sigma\}$$

We define the following sets of states:

$$\begin{aligned} N_i &= WT \cap \{\sigma \mid i \notin Dom(\sigma) \vee i \text{ is yielding in } \sigma\} \\ W_i &= WT \cap \{\sigma \mid \sigma(i) \equiv \textbf{wrong}\} \\ R_i &= WT \cap \{\sigma \mid \alpha(\sigma, \sigma(i)) \not\sqsubseteq \text{CL}\} \setminus N_i \\ L_i &= WT \cap \{\sigma \mid \alpha(\sigma, \sigma(i)) \sqsubseteq \text{CL}\} \setminus N_i \end{aligned}$$

We also define two auxiliary sets of states N and W:

- $N = \forall i.\, N_i$. That is, N is the set of well-typed states where all threads yielding.
- $W = \exists i.\, W_i$. That is, W is the set of well-typed states for which some thread is **wrong**.

A state σ is *non-blocking* if whenever a transaction reachable from σ starts executing its left-mover part, then that transaction can terminate, *i.e.* $\sigma \to^* \sigma'$ where $L_i(\sigma')$, then $\sigma' \to_i^* \sigma''$ such that $\neg L_i(\sigma'')$.

Those definitions now enable us to show that Theorem B.1 applies to YieldJava:

Theorem B.2 *(Cooperability). Let σ be a state such that $\vdash \sigma$. Suppose $\sigma \in N$, and there exists state σ' such that $\sigma \to^* \sigma'$. Then the following statements are true:*

1. *If $\sigma' \in N$, then $\sigma \to_c^* \sigma'$.*
2. *If σ is non-blocking and $\sigma' \in W$, then there exists a state σ'' such that $\sigma \to_c^* \sigma''$ and $\sigma'' \in W$.*

Proof. The first part of this theorem follows from the Reduction Theorem. That is, we show that for all threads i, the seven preconditions for instantiating the Reduction Theorem hold:

1. R_i, L_i, and W_i are pairwise disjoint,
2. $L_i / \to_i \setminus R_i$ is false,
3. $W_i / \to_i$ is false,
4. $\to_i$ and $\to_j$ are disjoint for all $j \neq i$,
5. $(\to_i \setminus R_i)$ right-commutes with $\to_j$ for all $j \neq i$,
6. $(L_i / \to_i)$ left-commutes with $\to_j$ for all $j \neq i$,
7. if $p \to_i q$, then $R_j(p) \Leftrightarrow R_j(q)$, $L_j(p) \Leftrightarrow L_j(q)$, and $W_j(p) \Leftrightarrow W_j(q)$, for all $j \neq i$.

Detailed proofs for these seven requirements appear in [56]. Once they are shown to hold, we apply the Reduction Theorem by substituting the following:

- the set W_i for $\mathcal{W}_i$,
- the set R_i for $\mathcal{R}_i$,
- the set L_i for $\mathcal{L}_i$,
- the relation $\to_i$ for $\hookrightarrow_i$,
- the relation $\to_c$ for $\hookrightarrow_c$,

- the state σ for p,
- the state σ' for q.

We prove the second part of this theorem as follows. For any non-blocking state π, let $L(\pi) = \{i \mid L_i(\pi)\}$. We show by induction on $|L(\pi)|$ that for any non-blocking state π with $W(\pi)$, there exists a π' such that $\pi \to^* \pi'$, $W(\pi')$, and $L(\pi') = \emptyset$. The base case trivially holds when $L(\pi) = \emptyset$. For the inductive case, pick $i \in L(\pi)$. Since π is non-blocking, $\pi \to_i^* \pi''$ where $W(\pi'')$ and $L(\pi'') = L(\pi) \setminus \{i\}$. By induction, $\pi'' \to^* \pi'$ and hence $\pi \to^* \pi'$ where $W(\pi')$ and $L(\pi') = \emptyset$.

From the above result we have that $\sigma \to^* \sigma'''$ where $W(\sigma''')$ and $L(\sigma''') = \emptyset$. We apply part 2 of the Reduction Theorem to then conclude that $\exists \sigma''. \sigma \to_c^* \sigma''$ and $W(\sigma'')$. □

Theorem 4.2 (Cooperative-preemptive equivalence), stated in Section 4.3, follows immediately from the first part of Theorem B.2 and applies to any well-typed terminating program.

For well-type programs that may not terminate, the second part of Theorem B.2 applies and shows that if the program may go wrong under the preemptive semantics, then it also may go wrong under the cooperative semantics. To prove this property, the theorem requires that the initial state σ of the program is non-blocking. This non-blocking property is necessary, as otherwise the wrong state may not be reached under the cooperative semantics since the transaction containing the error may not ever be run under a cooperative scheduler.

References

[1] M. Abadi, C. Flanagan, S.N. Freund, Types for safe locking: static race detection for Java, ACM Trans. Program. Lang. Syst. 28 (2) (2006) 207–255.
[2] S.V. Adve, K. Gharachorloo, Shared memory consistency models: a tutorial, Computer 29 (12) (1996) 66–76.
[3] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, J.R. Douceur, Cooperative task management without manual stack management, in: Annual Technical Conference, 2002, pp. 289–302.
[4] R.M. Amadio, S.D. Zilio, Resource control for synchronous cooperative threads, in: International Conference on Concurrency Theory, 2004, pp. 68–82.
[5] T. Ball, S.K. Rajamani, The SLAM project: debugging system software via static analysis, in: Symposium on Principles of Programming Languages, 2002, pp. 1–3.
[6] D. Beyer, T.A. Henzinger, R. Jhala, R. Majumdar, The software model checker Blast, Int. J. Softw. Tools Technol. Transf. 9 (5–6) (2007) 505–525.
[7] R.L. Bocchino Jr., V.S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, M. Vakilian, A type and effect system for deterministic parallel Java, in: Object-Oriented Programming, Systems, Languages, and Applications, 2009, pp. 97–116.
[8] G. Boudol, Fair cooperative multithreading, in: International Conference on Concurrency Theory, 2007, pp. 272–286.
[9] C. Boyapati, M. Rinard, A parameterized type system for race-free Java programs, in: Object-Oriented Programming, Systems, Languages, and Applications, 2001, pp. 56–69.
[10] J. Burnim, K. Sen, Asserting and checking determinism for multithreaded programs, in: International Symposium on Foundations of Software Engineering, 2009, pp. 3–12.
[11] J. Devietti, B. Lucia, L. Ceze, M. Oskin, DMP: deterministic shared memory multiprocessing, in: Conference on Architectural Support for Programming Languages and Operating Systems, 2009, pp. 85–96.
[12] D. Dice, Y. Lev, M. Moir, D. Nussbaum, Early experience with a commercial hardware transactional memory implementation, in: Conference on Architectural Support for Programming Languages and Operating Systems, 2009, pp. 157–168.
[13] J.L. Eppinger, L.B. Mummert, A.Z. Spector, Camelot and Avalon: A Distributed Transaction Facility, Morgan Kaufmann, 1991.
[14] A. Farzan, P. Madhusudan, Monitoring atomicity in concurrent programs, in: Computer Aided Verification, 2008, pp. 52–65.
[15] C. Flanagan, S.N. Freund, Atomizer: a dynamic atomicity checker for multithreaded programs, in: Symposium on Principles of Programming Languages, 2004, pp. 256–267.
[16] C. Flanagan, S.N. Freund, Adversarial memory for detecting destructive races, in: Conference on Programming Language Design and Implementation, 2010, pp. 244–254.
[17] C. Flanagan, S.N. Freund, FastTrack: efficient and precise dynamic race detection, Commun. ACM 53 (11) (2010) 93–101.
[18] C. Flanagan, S.N. Freund, M. Lifshin, S. Qadeer, Types for atomicity: static checking and inference for Java, ACM Trans. Program. Lang. Syst. 30 (4) (2008).
[19] C. Flanagan, S.N. Freund, J. Yi, Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs, in: Conference on Programming Language Design and Implementation, 2008, pp. 293–303.
[20] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, R. Stata, Extended static checking for Java, in: Conference on Programming Language Design and Implementation, 2002, pp. 234–245.
[21] C. Flanagan, S. Qadeer, Types for atomicity, in: Types in Language Design and Implementation, 2003, pp. 1–12.
[22] M. Flatt, S. Krishnamurthi, M. Felleisen, Classes and mixins, in: Symposium on Principles of Programming Languages, 1998, pp. 171–183.
[23] D. Grossman, Type-safe multithreading in Cyclone, in: Types in Language Design and Implementation, 2003, pp. 13–25.
[24] T. Harris, K. Fraser, Language support for lightweight transactions, in: Object-Oriented Programming, Systems, Languages, and Applications, 2003, pp. 388–402.
[25] J. Hatcliff, Robby, M.B. Dwyer, Verifying atomicity specifications for concurrent object-oriented software using model-checking, in: Conference on Verification, Model Checking, and Abstract Interpretation, 2004, pp. 175–190.
[26] M. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, in: International Symposium on Computer Architecture, 1993, pp. 289–300.
[27] C.A.R. Hoare, Towards a theory of parallel programming, in: Operating Systems Techniques, in: A.P.I.C. Stud. Data Process., vol. 9, 1972, pp. 61–71.
[28] M. Isard, A. Birrell, Automatic mutual exclusion, in: Workshop on Hot Topics in Operating Systems, 2007, pp. 1–6.
[29] Java Grande Forum, Java Grande benchmark suite, available at http://www.javagrande.org, 2008.
[30] A. Kulkarni, Y.D. Liu, S.F. Smith, Task types for pervasive atomicity, in: Object-Oriented Programming, Systems, Languages, and Applications, 2010, pp. 671–690.
[31] J.R. Larus, R. Rajwar, Transactional Memory, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2006.
[32] E. Lin, H. Rajan, Panini: a capsule-oriented programming language for implicitly concurrent program design, in: Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '13, Indianapolis, IN, USA, October 26–31, 2013, Companion volume, 2013, pp. 19–20.
[33] R.J. Lipton, Reduction: a method of proving properties of parallel programs, Commun. ACM 18 (12) (1975) 717–721.

[34] B. Liskov, D. Curtis, P. Johnson, R. Scheifler, Implementation of Argus, in: Proceedings of the Symposium on Operating Systems Principles, 1987, pp. 111–122.
[35] D.B. Lomet, Process structuring, synchronization, and recovery using atomic actions, in: Language Design for Reliable Software, 1977, pp. 128–137.
[36] J. Manson, W. Pugh, S.V. Adve, The Java memory model, in: Symposium on Principles of Programming Languages, 2005, pp. 378–391.
[37] M. Naik, A. Aiken, Conditional must not aliasing for static race detection, in: POPL, 2007, pp. 327–338.
[38] M. Naik, A. Aiken, J. Whaley, Effective static race detection for Java, in: Conference on Programming Language Design and Implementation, 2006, pp. 308–319.
[39] R. O'Callahan, J.-D. Choi, Hybrid dynamic data race detection, in: Symposium on Principles and Practice of Parallel Programming, 2003, pp. 167–178.
[40] P.W. O'Hearn, J.C. Reynolds, H. Yang, Local reasoning about programs that alter data structures, in: Computer Science Logic, CSL, 2001, pp. 1–19.
[41] M. Olszewski, J. Ansel, S. Amarasinghe, Kendo: efficient deterministic multithreading in software, in: Conference on Architectural Support for Programming Languages and Operating Systems, 2009, pp. 97–108.
[42] P. Pratikakis, J.S. Foster, M. Hicks, Context-sensitive correlation analysis for detecting races, in: Conference on Programming Language Design and Implementation, 2006, pp. 320–331.
[43] S. Qadeer, D. Wu, Kiss: keep it simple and sequential, in: Conference on Programming Language Design and Implementation, 2004, pp. 14–24.
[44] H. Rajan, Building scalable software systems in the multicore era, in: Proceedings of the Workshop on Future of Software Engineering Research, 2010, pp. 293–298.
[45] R. Rajwar, J.R. Goodman, Speculative lock elision: enabling highly concurrent multithreaded execution, in: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture, in: MICRO, vol. 34, IEEE Computer Society, Washington, DC, USA, 2001, pp. 294–305.
[46] J.C. Reynolds, Separation logic: a logic for shared mutable data structures, in: IEEE Symposium on Logic in Computer Science, LICS, 2002, pp. 55–74.
[47] A. Roy, S. Hand, T. Harris, A runtime system for software lock elision, in: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys '09, ACM, New York, NY, USA, 2009, pp. 261–274.
[48] C. Sadowski, S.N. Freund, C. Flanagan, SingleTrack: a dynamic determinism checker for multithreaded programs, in: European Symposium on Programming, 2009, pp. 394–409.
[49] C. Sadowski, J. Yi, Applying usability studies to correctness conditions: a case study of cooperability, in: Onward! Workshop on Evaluation and Usability of Programming Languages and Tools, 2010.
[50] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T.E. Anderson, Eraser: a dynamic data race detector for multi-threaded programs, ACM Trans. Comput. Syst. 15 (4) (1997) 391–411.
[51] N. Shavit, D. Touitou, Software transactional memory, in: ACM Symposium on Principles of Distributed Computing, 1995, pp. 204–213.
[52] C.A. Stone, M.E. O'Neill, The OCM Team, Observationally cooperative multithreading, in: OOPSLA Companion, 2011, pp. 205–206.
[53] C. von Praun, T. Gross, Static conflict analysis for multi-threaded object-oriented programs, in: Conference on Programming Language Design and Implementation, 2003, pp. 115–128.
[54] C. von Praun, T. Gross, Static detection of atomicity violations in object-oriented programs, in: Workshop on Formal Techniques for Java-Like Programs, 2003.
[55] L. Wang, S.D. Stoller, Runtime analysis of atomicity for multithreaded programs, IEEE Trans. Softw. Eng. 32 (2006) 93–110.
[56] J. Yi, T. Disney, S.N. Freund, C. Flanagan, Types for precise thread interference, Technical report UCSC-SOE-11-22, The University of California at Santa Cruz, 2011, at http://www.soe.ucsc.edu/research/technical-reports/ucsc-soe-11-22.
[57] J. Yi, T. Disney, S.N. Freund, C. Flanagan, Cooperative types for controlling thread interference in Java, in: Proceedings of the International Symposium on Software Testing and Analysis, 2012, pp. 232–242.
[58] J. Yi, C. Flanagan, Effects for cooperable and serializable threads, in: Types in Language Design and Implementation, 2010, pp. 3–14.
[59] J. Yi, C. Sadowski, C. Flanagan, Cooperative reasoning for preemptive execution, in: Symposium on Principles and Practice of Parallel Programming, 2011, pp. 147–156.
[60] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, S. Jagannathan, A uniform transactional execution environment for Java, in: European Conference on Object-Oriented Programming, 2008, pp. 129–154.