Effects for Cooperable and Serializable Threads

Jaeheon Yi Cormac Flanagan

Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA 95064
jaeheon@cs.ucsc.edu, cormac@cs.ucsc.edu

Abstract

Reasoning about the correctness of multithreaded programs is complicated by the potential for unexpected interference between threads. Previous work on controlling thread interference focused on verifying race freedom and/or atomicity. Unfortunately, race freedom is insufficient to prevent unintended thread interference. The notion of atomic blocks provides more semantic guarantees, but offers limited benefits for non-atomic code and it requires bimodal sequential/multithreaded reasoning (depending on whether code is inside or outside an atomic block).

This paper proposes an alternative strategy that uses yield annotations to control thread interference, and we present an effect system for verifying the correctness of these yield annotations. The effect system guarantees that for any preemptively-scheduled execution of a well-formed program, there is a corresponding cooperative execution with equivalent behavior in which context switches happen only at yield annotations. This effect system enables cooperative reasoning: the programmer can adopt the simplifying assumption of cooperative scheduling, even though the program still executes with preemptive scheduling and/or true concurrency on multicore processors. Unlike bimodal sequential/multithreaded reasoning, cooperative reasoning can be applied to all program code.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—parallel programming; D.2.4 [*Software Engineering*]: Software/Program Verification—reliability; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—specification techniques; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—program analysis

General Terms Reliability, Security, Languages, Verification

Keywords Effect system, yield, atomicity, race conditions

TLDI'10, January 23, 2010, Madrid, Spain.

1. Controlling Thread Interference

Multiple threads of control are widely used in software development because they help reduce latency and increase throughput on multicore and multiprocessor machines. Reasoning about the behavior and correctness of multithreaded code is difficult, however, due to the need to consider all possible interleavings of the various threads. Thus, methods for specifying and controlling the interference between threads are crucial to the cost-effective development of reliable multithreaded software.

Race Freedom Early attempts at controlling thread interference focused on *race conditions*, where two threads access a shared variable at the same time, and at least one access is a write. Race freedom ensures that thread interference occurs only at certain actions (those that manipulate volatile variables, locks, semaphores, etc). By itself, race freedom does not entirely prevent errors due to unexpected interactions between threads. We illustrate this limitation via the following code, in which the shared variable `x` is protected by the lock `m`, and `t` is a thread-local variable:

```
acquire(m); t = x; release(m);
t = t + 1;
acquire(m); x = t; release(m);
```

Although this code is race-free, it may not have the expected effect of incrementing `m` by 1, due to interference from other threads. Similar but less obvious errors arise in practice, often via multiple calls to synchronized methods [19].

Atomicity Motivated by this limitation of race-freedom, much recent work [5, 6, 18–23, 27, 40, 49–52] has used *atomic blocks* to specify where thread interference may occur. Programs can use traditional synchronization idioms (locks, etc.) to ensure that atomic blocks cannot be influenced by interleaved actions of concurrent threads. The atomic block annotations in a program are correct if any execution trace is *serializable*, that is, equivalent to a *serial* trace in which atomic blocks execute contiguously, without interleaved actions of other threads. Atomic blocks are amenable to sequential reasoning, which significantly simplifies subsequent correctness arguments.

Despite the benefits of atomicity, it suffers from two significant limitations. First, atomicity forces a form of *bimodal sequential/-multithreaded reasoning*. To illustrate this point, consider the simple code fragment[1]:

```
x++
```

If sequential reasoning is valid, then this code is a simple increment operation. If multithreaded reasoning is required, then this code becomes a *potentially non-atomic read-modify-write sequence* (with no syntactic hints to reflect this sea change from traditional, sequential semantics).

[1] We assume that `x` is `volatile`, to avoid memory model complexities.

In a world of atomic blocks, deciding which semantics is applicable critically depends on whether `x++` occurs inside or outside an atomic block. Thus, atomic blocks require the programmer to choose the appropriate mode of reasoning (sequential vs. multithreaded) for each program statement, with obvious room for confusion and error.

A second limitation of atomicity is that it provides little help in reasoning about non-atomic methods, for which the programmer is still exposed to the full difficulties of multithreaded reasoning based on instruction-level interleaving. This problem can be partially addressed by declaring parts of a non-atomic method to be atomic, but this strategy is limited by the static scope of explicit atomic blocks, as we illustrate below.

Yield Specifications This paper proposes an alternative strategy for controlling thread interference that addresses these limitations. Instead of delimiting code blocks where interference is guaranteed not to occur, we invert the problem and use `yield` annotations to specify program points where interference may occur. Our approach is inspired by prior ideas on cooperative scheduling [4, 7, 9] and automatic mutual exclusion [29], with the difference that we use `yield` entirely as a specification construct. That is, we assume programs use traditional synchronization constructs (such as locks) and we verify that the resulting thread interference satisfies these yield specifications.

The yield annotations in a program are correct if any arbitrarily-interleaved execution trace is *cooperable*, that is, equivalent to a *cooperative* trace in which context switches happen only at yield annotations. Thus, although the program can execute with preemptive interleaving (and true concurrency on multicore processors), yield annotations allow us to adopt the simplifying assumption that threads execute cooperatively and that thread interference happens only at yield annotations.

Benefits of Yield Specifications We believe that yield is a promising non-interference specification because it enables universal *cooperative reasoning*, instead of bimodal sequential/multithreaded reasoning. Sequential reasoning never considers thread interference, but is inapplicable to non-atomic methods. Multithreaded reasoning is applicable to such methods, but must allow for thread interference between every pair of instructions. Cooperative reasoning simplifies the analysis of non-atomic methods, since thread interference happens only at yield annotations. Moreover, cooperative reasoning reduces to sequential reasoning for atomic methods, and so conveniently applies to both atomic and non-atomic methods.

Under cooperative reasoning, the problematic operation `x++` mentioned above always represents an atomic increment operation. If `x` is concurrently manipulated by other threads, then the code must explicitly reflect its non-atomic read-modify-write nature:

```
{ int t = x; yield; x = t+1; }
```

We conjecture that including explicit yield specifications in a program's source code could provide benefits during code evolution, since these specifications provide a valuable reminder of exactly where thread interference may occur, and where the "natural" expectation of sequential semantics is violated. In addition, they allow tools to detect when additional (and possibly unintentional) thread interference is introduced.

Yield Effects In this paper, we address the problem of statically verifying if yield annotations are correct, that is, if every program trace is equivalent to a cooperative trace. We present our static analysis as an effect system [33, 34], where the effect of a program expression summarizes how the evaluation of that expression may interfere with concurrent actions of other threads.

In order to facilitate comparisons with earlier work on atomicity, our presentation follows that of [20], with the key addition that we now verify cooperability in addition to serializability. This presentation formalizes our results in terms of an idealized language, but prior work produced expressive type and effect systems for verifying atomic specifications in Java programs [5, 19–21, 23, 40, 49], and we believe that such techniques could be adapted, *mutatis mutandis*, to also verify yield specifications.

A Motivating Example We illustrate the utility of yield annotations via the code fragment shown in Figure 1 (a). At a high level, this code performs the update `x = f(x)` on the integer shared variable `x`. However, to avoid holding the lock `m` protecting `x` during the computation of `f(x)`, the code first reads `x` into a local variable `y`, computes `f(y)`, and then acquires the lock to perform the update of `x`. If `x` has changed since it was first read, then the update must be retried, looping until `done` is true. For performance reasons, the lock `m` is held for all writes to `x`, but not for all reads.

The `yield` annotation explicates that interleaved steps of other threads may cause thread interference, but this interference appears as if it happens only at the start of the loop, at the `yield` annotation. Thus, cooperative reasoning only needs to consider one point of thread interference, in contrast to multithreaded reasoning, which needs to consider thread interference at all points in the code.

In comparison, atomic blocks provide a less satisfactory specification construct, due to the awkward interaction between the static scope of atomic blocks and other program constructs such as `while` loops. The method `update_x` is not atomic, and so atomic blocks need to be inserted selectively into the method body instead.

In a first attempt, the loop body is declared as `atomic` , as shown in Figure 1(b), but this annotation suggests thread interference is possible outside that atomic block, for example, during the evaluation of the loop condition.

Figure 1(c) attempts to remedy this problem by adding two additional atomic blocks, but the resulting code remains unsatisfactory: the additional atomic blocks clutter the code, and yet still permit thread interference at points where no interference is intended, such as on entry to the `while` loop. In summary, atomic blocks are inadequate for specifying thread interference in non-atomic methods such as `update_x`.

Atomic and Yield An atomic method is simply one that can never execute a yield annotation. Our effect system exploits this connection to also verify atomicity specifications, in a manner similar to [1]. Indeed, we believe that `atomic` and `yield` are compatible and complementary specification idioms.

- Yield is primarily a "code-level" specification that explicates where thread interference may occur.
- Atomic is most useful as an "interface-level" specification that clarifies that a method's implementation does not include yield annotations, and so is not vulnerable to thread interference. In contrast, a non-atomic method may permit interfering actions of other threads.

We illustrate the complementary nature of these specification idioms further by the example in Section 5.2.

Outline The presentation of our results proceeds as follows. Section 2 introduces an effect language for reasoning about cooperable and serializable traces. Section 3 presents an idealized, multithreaded language, and Section 4 formalizes our effect system for this language. Section 5 illustrates the benefits of our effect system via two code development examples. Section 6 verifies the correctness properties of preservation, cooperability, and serializability. Section 7 and 8 discuss related and future work, and Section 9 concludes.

Figure 1: A Comparison of Yield vs. Atomic Specifications: Code performs update `x = f(x)` via transactional-retry

```
void update_x() {

  boolean done = false;
  int y = x;

  while ( !done ) {
    yield;
    int fy = f(y);
    acquire(m);
    if (x == y) {
      x = fy;
      done = true;
    } else {
      y = x;
    }
    release(m);

  }
}
```

(a) Using yield annotations

```
void update_x() {

  boolean done = false;
  int y = x;

  while ( !done ) {
    atomic {
      int fy = f(y);
      acquire(m);
      if (x == y) {
        x = fy;
        done = true;
      } else {
        y = x;
      }
      release(m);
    }
  }
}
```

(b) Using one atomic block annotation

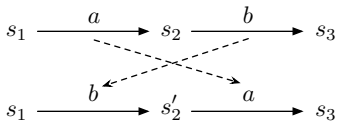
```
void update_x() {

  boolean done;
  int y;
  atomic {
    done = false;
    y = x;
  }

  while ( atomic { !done } ) {
    atomic {
      int fy = f(y);
      acquire(m);
      if (x == y) {
        x = fy;
        done = true;
      } else {
        y = x;
      }
      release(m);
    }
  }
}
```

(c) Using three atomic block annotations

2. Effects for Cooperability and Serializability

2.1 The Theory of Reduction

We use the theory of right and left movers, first proposed by Lipton [32], to verify yield annotations. An action a is a *right mover* if for any execution where the action a performed by one thread is immediately followed by an action b of a different thread, the actions a and b can be swapped without changing the resulting state S_3, as shown below.



Each lock acquire is a right mover, since if it is followed by an action b of a second thread, then b can neither acquire nor release the lock, and hence the acquire action can be moved to the right of b without changing the resulting state.

Similarly, an action b is a *left mover* if whenever b immediately follows an action a of a different thread, the actions a and b can be swapped, again without changing the resulting state. Every lock release is a left mover, since the preceding action b of a different thread can never influence the state of the lock, and so the two actions commute.

Finally, consider an access (read or write) to a shared variable. If that access is *racy* (*i.e.*, involved in a simultaneous access race condition), then it is neither a left nor a right mover. Conversely, if the access is not racy, then it is both a left and a right mover.

We use the term *epoch* to refer to the sequence of actions executed by a thread between two successive yield annotations. Consider an epoch that contains a sequence of right movers followed by a single atomic action followed by a sequence of left movers. Then an execution where this epoch has been fully executed can be *reduced* [32], by commuting out interleaved actions of other threads, to another execution with the same resulting state where the epoch is executed contiguously.

2.2 Cooperability Effects

To capture Lipton's theory of left and right movers, together with yield annotations, we introduce the following language of *cooperative effects*:

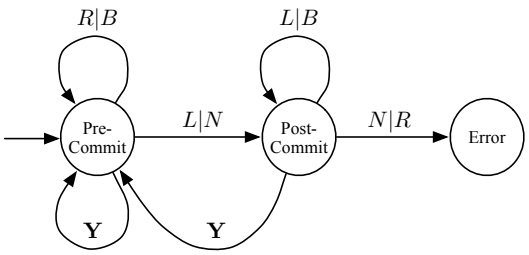
$$c ::= \text{R} \mid \text{L} \mid \text{B} \mid \text{N} \mid \text{Y}$$

with the following meaning:

- The effect R describes right mover actions such as lock acquires.
- The effect L describes left mover actions such as lock releases.
- The effect B describes *both mover* actions that are both left and right movers, such as race-free accesses.
- N describes *non mover* actions that are neither left nor right movers, such as racy accesses.
- Finally, the effect Y describes yield annotations.

Two traces are *equivalent* if one can be obtained from the other by repeatedly swapping adjacent actions according to their right and left mover properties. An execution trace is *cooperative* if context switches only happen at yield annotations, and a trace is *cooperable* if it is equivalent to a cooperative trace.

Our strategy for proving that a trace is cooperable is to demonstrate that every thread consists of reducible epochs separated by yield annotations, as illustrated by the following DFA. If we ignore the two bottom yield transitions, this DFA simply describes reducible epochs. The yield annotations then have the effect of resetting this "reducibility" DFA at the start of each new epoch.

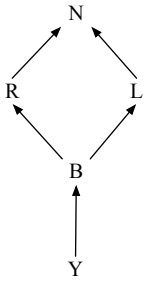


Our effect system is essentially a compositional analysis for verifying that the actions of each thread satisfy this DFA. The effect of a term summarizes the sequence of actions that may be performed by that term, and how that sequence of actions affects the acceptance of the entire thread by this DFA.

From this DFA, we derive the cooperative effect ordering:

$$\text{Y} \sqsubseteq \text{B} \sqsubseteq c \sqsubseteq \text{N} \qquad \text{for } c \in \{\text{L}, \text{R}\}$$

as illustrated below. For example, $\text{Y} \sqsubseteq \text{B}$, since for any effect sequence α and β, if $\alpha.\text{B}.\beta$ is accepted by this DFA then $\alpha.\text{Y}.\beta$ is also accepted. Let $\sqcup$ denote the join operator based on this ordering.



Terms in source programs can be composed via sequential composition and iteration. We develop analogous sequential composition ($c_1; c_2$) and iterative composition (c^*) operations on effects, as defined in the following tables.

$c_1; c_2$	Y	B	R	L	N
Y	Y	Y	Y	L	L
B	Y	B	R	L	N
R	R	R	R	N	N
L	Y	L	-	L	-
N	R	N	-	N	-

c	c^*
Y	B
B	B
R	R
L	L
N	-

Note that these operations are partial, and are undefined at entries marked "-". For example, two racy read actions (with effect N) cannot be sequentially composed, since $\alpha.\text{N}.\text{N}.\beta$ is never accepted by the DFA, and so the sequential composition N;N is undefined. If the racy reads are separated by a yield annotation, then the effect composition is N;Y;N, which is defined and evaluates to N.

2.3 Serializability Effects

The *serializability effect* of a program term simply summarizes whether that term may execute a yield operation:

$$s \ ::= \ \text{A} \mid \text{C}$$

Here, A denotes atomic (yield free) terms, and C denotes compound terms that may execute a yield on some code paths.[2] These effects are ordered by $\text{A} \sqsubseteq \text{C}$, with associated join operation $\sqcup$. For consistency, we also define sequential and iterative composition operators:

$$\begin{array}{rcl} s_1; s_2 & \stackrel{\text{def}}{=} & s_1 \sqcup s_2 \\ s^* & \stackrel{\text{def}}{=} & s \end{array}$$

[2] These effects correspond to the `NoYields` and `Yields` effects in [1].

2.4 Combined Effects

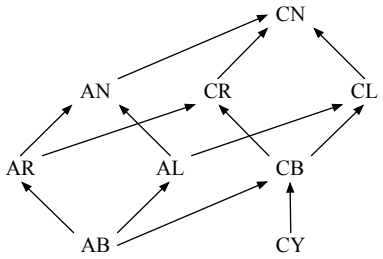
We now combine these serializability and cooperative effects into a *combined effect* κ, that allows us to simultaneously verify cooperability and serializability properties:

$$\kappa \ ::= \ \langle s, c \rangle$$

We introduce the following pointwise operations on combined effects:

$$\begin{array}{rcll} \langle s_1, c_1 \rangle \sqsubseteq \langle s_2, c_2 \rangle & \text{iff} & s_1 \sqsubseteq s_2 \text{ and } c_1 \sqsubseteq c_2 \\ \langle s_1, c_1 \rangle \sqcup \langle s_2, c_2 \rangle & \stackrel{\text{def}}{=} & \langle s_1 \sqcup s_2, c_1 \sqcup c_2 \rangle \\ \langle s_1, c_1 \rangle; \langle s_2, c_2 \rangle & \stackrel{\text{def}}{=} & \langle s_1; s_2, c_1; c_2 \rangle \\ \langle s, c \rangle^* & \stackrel{\text{def}}{=} & \langle s^*, c^* \rangle \end{array}$$

For brevity, we often abbreviate a combined effect such as $\langle \text{C}, \text{N} \rangle$ to simply CN. Note that the two components in the combined effect $\langle \text{A}, \text{Y} \rangle$ contradict each other regarding the presence of yields. Unsurprisingly, this combined effect is never used in our system, resulting in combined effects forming the following nine-element join-semilattice.



Unlike prior effect systems for atomicity [20], the introduction of yields results in a join-semilattice rather than a complete lattice, since the minimal elements AB and CY are incomparable. Furthermore, in the above lattice, the rightmost five elements $\langle \text{C}, \cdot \rangle$ can all be seen as refinements of the single top element "$\top$" from [20].

Note that sequential composition is monotonic and associative,

$$\begin{array}{rcl} (\kappa_1 \sqsubseteq \kappa_1') \wedge (\kappa_2 \sqsubseteq \kappa_2') & \Rightarrow & (\kappa_1; \kappa_2) \sqsubseteq (\kappa_1'; \kappa_2') \\ \kappa_1; (\kappa_2; \kappa_3) & = & (\kappa_1; \kappa_2); \kappa_3 \end{array}$$

AB is the left and right identity for sequential composition,

$$\begin{array}{rcl} \text{AB}; \kappa & = & \kappa \\ \kappa; \text{AB} & = & \kappa \end{array}$$

iterative composition is monotonic and idempotent,

$$\begin{array}{rcl} \kappa \sqsubseteq \kappa' & \Rightarrow & \kappa^* \sqsubseteq \kappa'^* \\ (\kappa^*)^* & = & \kappa^* \end{array}$$

and sequential composition distributes over joins.

$$\begin{array}{rcl} \kappa_1; (\kappa_2 \sqcup \kappa_3) & = & (\kappa_1; \kappa_2) \sqcup (\kappa_1; \kappa_3) \\ (\kappa_1 \sqcup \kappa_2); \kappa_3 & = & (\kappa_1; \kappa_3) \sqcup (\kappa_2; \kappa_3) \end{array}$$

3. The Language CAY

We formalize our effect system and its correctness properties in the context of the idealized language CAY, a restricted subset of C extended with atomic and yield annotations. CAY is a small, imperative, multithreaded language with higher-order functions and dynamic thread creation. CAY follows the syntax and semantic presentation of the CAT language [20], extending it with `yield` annotations and with a notion of cooperative execution.

3.1 Syntax

The syntax and semantics of CAY are summarized in Figure 2. The language supports multithreaded programming via the construct `fork` t, which spawns a new thread for the evaluation of t, and by providing both `atomic` blocks and `yield` annotations.

CAY includes variable reference and assignment, primitive and function applications, conditionals, and while loops. We use the notation $\overline{t}$ to denote a sequence of terms. The term `let` $x = t_1$ `in` t_2 allocates a fresh variable x in the heap, and initializes x with the result of evaluating t_1. The variable x is bound within t_2, and may be α-renamed in the usual fashion. We use $t_1; t_2$ to abbreviate `let` $x = t_1$ `in` t_2 when x is not free in t_2.

Values in CAY include constants, function definitions, and synchronization locations. The set of constants is left unspecified but should include integers. A function definition $f(\overline{x})\ t$ introduces a function named f, whose formal parameters $\overline{x}$ are bound within the body t, and may be α-renamed in the usual fashion.

Synchronization locations are references to *synchronization values*. For generality, we leave the set of synchronization values unspecified, but they might include, for example, mutual exclusion locks, reader-writer locks, semaphores, etc. These synchronization values are manipulated by primitive applications $p(\overline{t})$, which might include operations to allocate, acquire, and release mutual exclusion locks. In addition, the set of primitives also include arithmetic operations and the `assert` primitive.

For clarity, our effect system treats race detection and flow analysis as orthogonal to our central concern. We assume that a separate race detector (*e.g.*, [10, 17, 25]) has annotated each variable access with an *conflict tag*. This tag is $\bullet$ if that access may be involved in a race condition, and is ϵ otherwise. In addition, we assume a flow analysis has annotated each function call $t^F(\overline{t})$ with a *call tag* F denoting the set of functions that may be invoked by that call. Function names are used only to encode the results of this flow analysis; they are not considered variable in the program. This modularization allows our effect system to focus on the task at hand, namely verifying the correctness of `yield` and `atomic` annotations.

3.2 Semantics

A program state is a 3-tuple containing a heap H (a partial map from variables to values), a synchronization heap M (a partial map from synchronization locations to synchronization values), and a sequence T of threads. The $\multimap$ arrow denotes a partial map. Each thread is either a term or **wrong**. A thread becomes **wrong** if a primitive operation is applied to incorrect arguments. An evaluation context E is a term with a "hole" $[\ ]$ in place of the next sub-term to be evaluated, and $E[t]$ denotes the operation of filling the hole in E with the term t.

The transition relation $\rightarrow_i$ performs a single step of thread i. The notation $H[x := v]$ denotes a new heap that is identical to H except that it maps x to v. A `yield` operation is a nop in the semantics, reflecting its nature as a purely specification-level construct. The `in-atomic` construct records that execution is proceeding inside an atomic block, and should only appear in an evaluation context position. A `fork` operation adds a new thread to the program, prefixed with `yield`, which signifies that context switches may occur before the forked thread starts execution.

The meaning of a primitive operation p is defined using the partial function $\mathcal{I}_p$ that takes (1) a sequence of argument values, (2) a synchronization heap, and (3) an integer identifying the current thread, and returns a pair of a return value and a (possibly modified) synchronization heap. If the primitive is applied to incorrect arguments, $\mathcal{I}_p$ may instead return **wrong**:

$$\mathcal{I}_p : (Const \cup SyncLoc)^* \times SyncHeap \times Int \multimap ((Const \cup SyncLoc) \times SyncHeap) \cup \{\mathbf{wrong}\}$$

We illustrate how $\mathcal{I}_p$ models the semantics of `assert`, addition, and operations to allocate, acquire, and release locks as follows (where ϵ is the empty sequence and $m.n$ is a sequence of length two):

$$\mathcal{I}_{\texttt{assert}}(v, M, tid) = \begin{cases} \langle 0, M\rangle & \text{if } v \neq 0 \\ \mathbf{wrong} & \text{if } v = 0 \end{cases}$$

$$\mathcal{I}_{+}(m.n, M, tid) = \langle m + n, M\rangle$$

$$\mathcal{I}_{\texttt{newLock}}(\epsilon, M, tid) = \langle m, M[m := \langle \texttt{lock}, 0\rangle]\rangle \quad \text{if } m \notin dom(M)$$

$$\mathcal{I}_{\texttt{acquire}}(m, M[m := \langle \texttt{lock}, 0\rangle], tid) = \langle m, M[m := \langle \texttt{lock}, tid\rangle]\rangle$$

$$\mathcal{I}_{\texttt{release}}(m, M, tid) = \begin{cases} \langle m, M[m := \langle \texttt{lock}, 0\rangle]\rangle & \text{if } M(m) = \langle \texttt{lock}, tid\rangle \\ \mathbf{wrong} & \text{otherwise} \end{cases}$$

An `assert` goes wrong if its argument is 0, and otherwise terminates normally. Addition does not modify the synchronization heap. The `newLock` operation returns a synchronization location m that refers to a newly-allocated lock (perhaps chosen nondeterministically) containing 0, indicating that is not held by any thread. The `acquire` operation acquires a lock, provided the lock is not held by any thread; and otherwise blocks, in which case execution must proceed on other threads. The `release` operation never blocks; if the current thread holds the lock then the lock is released, and otherwise `release` goes wrong.

We define three transition relations over program states, reflecting different scheduling policies:

- The preemptive transition relation $\rightarrow_p$ performs a single step of an arbitrarily chosen thread.
- The serial transition relation $\rightarrow_s$ is similar to $\rightarrow_p$, with the additional restriction that a thread cannot perform a step if another thread is inside an `in-atomic` block. Thus $\rightarrow_s$ does not interleave the execution of an `atomic` block with instructions from other threads.
- The cooperative transition relation $\rightarrow_c$ is also similar to $\rightarrow_p$, with the restriction that context switches only happen at particular yield points. More specifically, a thread i is *yielding* if either:
  1. it has not started execution ($|T| < i$);
  2. if the system is single-threaded ($|T| = 1$);
  3. if the next operation is `yield` ($T_i = E[\texttt{yield}]$); or
  4. if the thread has gone wrong ($T_i = \mathbf{wrong}$).

  A thread can only perform a cooperative step if all other threads are yielding.

Note that the evaluation rule for the expression `fork` t creates a new thread (`yield`; t) that is initially yielding, in order to permit cooperative execution to continue on the original forking thread.

We use $\rightarrow^+$ and $\rightarrow^*$ to denote the transitive and reflexive-transitive closure of a transition relation $\rightarrow$, respectively.

Figure 2: CAY **Syntax and Semantics**

Syntax

$$
\begin{array}{rcrcl}
t & \in & \mathit{Term} & ::= & v \mid x_r \mid x_r := t \\
 & & & \mid & p(\overline{t}\,) \mid e^F(\overline{t}\,) \\
 & & & \mid & \texttt{if}\ t\ t\ t \mid \texttt{while}\ t\ t \mid \texttt{let}\ x = t\ \texttt{in}\ t \mid \texttt{fork}\ t \\
 & & & \mid & \texttt{atomic}\ t \mid \texttt{in-atomic}\ t \\
 & & & \mid & \texttt{yield} \\
v & \in & \mathit{Value} & ::= & c \mid m \mid f(\overline{x}\,)\ t \\
r & \in & \mathit{Tag} & ::= & \bullet \mid \epsilon \\
p & \in & \mathit{Prim} & ::= & \texttt{acquire} \mid \texttt{release} \mid \cdots \\
x & \in & \mathit{Var} & & \\
m & \in & \mathit{SyncLoc} & & \\
f & \in & \mathit{FnName} & & \\
F & \in & 2^{\mathit{FnName}} & & \\
c & \in & \mathit{Const} & &
\end{array}
$$

Evaluation contexts

$$
\begin{array}{rcrcl}
E & \in & \mathit{EvalCtxt} & ::= & [\,] \mid x_r := E \\
 & & & \mid & p(\overline{v}, E, \overline{t}\,) \mid E^F(\overline{t}\,) \mid v^F(\overline{v}, E, \overline{t}\,) \\
 & & & \mid & \texttt{if}\ E\ t\ t \mid \texttt{let}\ x = E\ \texttt{in}\ t \\
 & & & \mid & \texttt{in-atomic}\ E
\end{array}
$$

State space

$$
\begin{array}{rcrcl}
P & \in & \mathit{ProgState} & = & \mathit{Heap} \times \mathit{SyncHeap} \times \mathit{ThreadSeq} \\
H & \in & \mathit{Heap} & = & \mathit{Var} \rightharpoonup \mathit{Value} \\
M & \in & \mathit{SyncHeap} & = & \mathit{SyncLoc} \rightharpoonup \mathit{SyncValue} \\
T & \in & \mathit{ThreadSeq} & = & (\mathit{Expr} \cup \{\mathbf{wrong}\})^*
\end{array}
$$

Transition relations

$$\rightarrow_i, \rightarrow_p, \rightarrow_s, \rightarrow_c \quad \subseteq \quad \mathit{ProgState} \times \mathit{ProgState}$$

Thread i transition relation $\rightarrow_i$ (where $i = |T| + 1$)

$$
\begin{array}{lcll}
H,\, M,\, T.E[x_r].T' & \rightarrow_i & H,\, M,\ T.E[H(x)].T' & \\
H,\, M,\, T.E[x_r := v].T' & \rightarrow_i & H[x := v],\, M,\ T.E[v].T' & \\
H,\, M,\, T.E[p(\overline{v}\,)].T' & \rightarrow_i & H,\, M',\, T.E[v'].T' & \text{if } \mathcal{I}_p(\overline{v}\,, M, i) = \langle v', M' \rangle \\
H,\, M,\, T.E[p(\overline{v}\,)].T' & \rightarrow_i & H,\, M,\ T.\mathbf{wrong}.T' & \text{if } \mathcal{I}_p(\overline{v}\,, M, i) = \mathbf{wrong} \\
H,\, M,\, T.E[(f(\overline{x}\,)\ t)^F(\overline{v}\,)].T' & \rightarrow_i & H[\overline{x} := \overline{v}],\, M,\ T.E[t].T' & \text{if } \overline{x} \cap \mathit{dom}(H) = \emptyset \\
H,\, M,\, T.E[\texttt{if}\ v\ t_1\ t_2].T' & \rightarrow_i & H,\, M,\ T.E[t_1].T' & \text{if } v \neq 0 \\
H,\, M,\, T.E[\texttt{if}\ 0\ t_1\ t_2].T' & \rightarrow_i & H,\, M,\ T.E[t_2].T' & \\
H,\, M,\, T.E[\texttt{while}\ t_1\ t_2].T' & \rightarrow_i & H,\, M,\ T.E[\texttt{if}\ t_1\ (t_2; \texttt{while}\ t_1\ t_2)\ 0].T' & \\
H,\, M,\, T.E[\texttt{let}\ x = v\ \texttt{in}\ t].T' & \rightarrow_i & H[x := v],\, M,\ T.E[t].T' & \text{if } x \notin \mathit{dom}(H) \\
H,\, M,\, T.E[\texttt{fork}\ t].T' & \rightarrow_i & H,\, M,\ T.E[0].T'.(\texttt{yield}; t) & \\
H,\, M,\, T.E[\texttt{atomic}\ t].T' & \rightarrow_i & H,\, M,\ T.E[\texttt{in-atomic}\ t].T' & \\
H,\, M,\, T.E[\texttt{in-atomic}\ v].T' & \rightarrow_i & H,\, M,\ T.E[v].T' & \\
H,\, M,\, T.E[\texttt{yield}].T' & \rightarrow_i & H,\, M,\ T.E[0].T' &
\end{array}
$$

Preemptive transition relation $\rightarrow_p$

$$H, M, T \quad \rightarrow_p \quad P' \qquad \text{if } H, M, T \rightarrow_i P'$$

Serial transition relation $\rightarrow_s$

$$H, M, T \quad \rightarrow_s \quad P' \qquad \text{if } H, M, T \rightarrow_i P' \text{ and } T_j \text{ does not contain } \texttt{in-atomic} \text{ for } j \neq i$$

Cooperative transition relation $\rightarrow_c$

$$H, M, T \quad \rightarrow_c \quad P' \qquad \text{if } H, M, T \rightarrow_i P' \text{ and } T_j \text{ is yielding for } j \neq i$$

4. An Effect System for Cooperability and Serializability

We now develop an effect system [33, 34] to verify cooperability and serializability of CAY programs. This effect system uses an environment Γ to map function names to corresponding effect specifications, which are then checked via assume-guarantee reasoning. The environment also maps primitives to corresponding effect specifications. For the primitives mentioned in the previous section, appropriate effect specifications are:

$$\begin{array}{rcl} \Gamma(\texttt{assert}) & = & \text{AB} \\ \Gamma(\texttt{+}) & = & \text{AB} \\ \Gamma(\texttt{newLock}) & = & \text{AB} \\ \Gamma(\texttt{acquire}) & = & \text{AR} \\ \Gamma(\texttt{release}) & = & \text{AL} \end{array}$$

The core of our system is a collection of rules for reasoning about the effect judgment:

$$\Gamma \vdash t : \kappa$$

which states that term t has effect κ in environment Γ. The rules defining this judgement are shown in Figure 3, and are mostly straightforward, since much of the novelty of our system is encapsulated in the effect language and its composition operators.

The rule [CONST] says that the effect of a constant is AB, since the "evaluation" of a constant is atomic (A), as it performs no yield operations, and is a both mover (B), as its evaluation does not interfere with other threads. A similar rule applies for synchronization locations.

The effect of a variable read x_r depends on the conflict tag r. If $r = \epsilon$, then this read commutes (in both directions) with steps of other threads, and so has effect AB, via the rule [READ]. If $r = \bullet$, then this read has effect AN, indicating that it is an atomic action that may not commute with steps of other threads: see [READ RACE]. The rules for variable writes are similar.

The effect of $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ is the composition $\kappa_1;\kappa_2$ of the effects of t_1 and t_2. The rule [WHILE] for $\texttt{while}\ t_1\ t_2$ determines the effects κ_1 and κ_2 of t_1 and t_2, and summarizes the effect of the while loop as $\kappa_1;(\kappa_2;\kappa_1)^*$, reflecting its iterative nature.

The rule [FUN] for a function definition $f(\overline{x})\ t$ checks that the effect of the function body t must be at most $\Gamma(f)$. The rule [INVOKE] for a function application $t^F(\overline{t})$ computes the join $\sqcup_{f\in F}\Gamma(f)$ of all possible callees (via the call tag F) when computing the overall effect of the function call. Thus, function specifications are checked via assume-guarantee reasoning.

Interestingly, the effect of a `fork` operation is AL, since this operation could be immediately followed (but not preceded) by a conflicting operation of the newly created thread in a preemptive execution. The effect of `yield` is CY, reflecting that it is a yield operation and hence compound (that is, not atomic). The effect of the body of an atomic block must be at most AN.

The effect of a primitive application $p(\overline{t})$ is the sequential composition of the effects of the argument terms $\overline{t}$, followed by the effects $\Gamma(p)$ of p.

Finally, the judgement $\Gamma \vdash P$ checks if a program state P is well-formed, by checking that all threads and all heap values are well-formed, and that `in-atomic` only occurs in evaluation context positions.

Figure 3: Effect System for Cooperability and Serializability

$$\boxed{\Gamma \vdash t : \kappa}$$

$$\frac{}{\Gamma \vdash c : \text{AB}} \quad [\text{CONST}]$$

$$\frac{}{\Gamma \vdash m : \text{AB}} \quad [\text{SYNCLOC}]$$

$$\frac{\Gamma \vdash t : \kappa \qquad \kappa \sqsubseteq \Gamma(f)}{\Gamma \vdash f(\overline{x})\ t : \text{AB}} \quad [\text{FUN}]$$

$$\frac{\Gamma \vdash t_i : \kappa_i}{\Gamma \vdash p(\overline{t}) : (\kappa_1;\ldots;\kappa_n;\Gamma(p))} \quad [\text{PRIM}]$$

$$\frac{}{\Gamma \vdash x_\epsilon : \text{AB}} \quad [\text{READ}]$$

$$\frac{}{\Gamma \vdash x_\bullet : \text{AN}} \quad [\text{READ RACE}]$$

$$\frac{\Gamma \vdash t : \kappa}{\Gamma \vdash x_\epsilon := t : (\kappa;\text{AB})} \quad [\text{ASSIGN}]$$

$$\frac{\Gamma \vdash t : \kappa}{\Gamma \vdash x_\bullet := t : (\kappa;\text{AN})} \quad [\text{ASSIGN RACE}]$$

$$\frac{\Gamma \vdash t_1 : \kappa_1 \quad \Gamma \vdash t_2 : \kappa_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : (\kappa_1;\kappa_2)} \quad [\text{LET}]$$

$$\frac{\Gamma \vdash t_i : \kappa_i}{\Gamma \vdash \texttt{if}\ t_1\ t_2\ t_3 : (\kappa_1;(\kappa_2 \sqcup \kappa_3))} \quad [\text{IF}]$$

$$\frac{\Gamma \vdash t_1 : \kappa_1 \quad \Gamma \vdash t_2 : \kappa_2}{\Gamma \vdash \texttt{while}\ t_1\ t_2 : (\kappa_1;(\kappa_2;\kappa_1)^*)} \quad [\text{WHILE}]$$

$$\frac{\Gamma \vdash t : \kappa \quad \Gamma \vdash t_i : \kappa_i}{\Gamma \vdash t^F(\overline{t}) : (\kappa;\kappa_1;\ldots;\kappa_n;(\sqcup_{f\in F}\Gamma(f)))} \quad [\text{INVOKE}]$$

$$\frac{\Gamma \vdash t : \kappa}{\Gamma \vdash \texttt{fork}\ t : \text{AL}} \quad [\text{FORK}]$$

$$\frac{}{\Gamma \vdash \texttt{yield} : \text{CY}} \quad [\text{YIELD}]$$

$$\frac{\Gamma \vdash t : \kappa \quad \kappa \sqsubseteq \text{AN}}{\Gamma \vdash \texttt{atomic}\ t : \kappa} \quad [\text{ATOMIC}]$$

$$\frac{\Gamma \vdash t : \kappa \quad \kappa \sqsubseteq \text{AN}}{\Gamma \vdash \texttt{in-atomic}\ t : \kappa} \quad [\text{INATOMIC}]$$

$$\frac{}{\Gamma \vdash \textbf{wrong} : \text{AB}} \quad [\text{WRONG}]$$

$$\boxed{\Gamma \vdash P}$$

$$\frac{\begin{array}{c}\forall x \in dom(H).\ \Gamma \vdash H(x) : \text{AB} \\ \Gamma \vdash T_i : \kappa_i \\ \forall t.\ (T_i \text{ contains } \texttt{in-atomic}\ t \Rightarrow \\ \qquad \exists E.\ T_i \equiv E[\texttt{in-atomic}\ t])\end{array}}{\Gamma \vdash H, M, T_1 \ldots T_n} \quad [\text{PROG}]$$

5. Two Applications

5.1 Yield as a Code-Level Specification

We now consider how our effect system might help during the development of the code from Figure 1. The programmer's goal is to develop a simple function `update_x` that updates the shared

variable x according to the function f, as illustrated by this initial implementation:

```
void update_x() {
  x• = f(x•);
}
```

This initial implementation is rejected by our effect system because the two races on x (annotated by a race detector) are not separated by a yield. Protecting the shared variable x by the lock m fixes the problem:

```
void update_x() {
  acquire(m);
  x_ϵ = f(x_ϵ); // no races
  release(m);
}
```

The revised code is now accepted by our effect system. Unfortunately, it may not provide adequate performance due to lock contention, particularly if the computation f(x) takes a long time. A naive attempt to improve performance is to avoid holding the lock m while calling f:

```
void update_x() {
  int fx = f(x•);
  acquire(m);
  x• = fx;
  release(m);
}
```

This code is rejected, since it requires a yield to explicate where thread interference may occur:

```
void update_x() {
  int fx = f(x•);
  yield;
  acquire(m);
  x• = fx;
  release(m);
}
```

This yield annotation highlights the problem that another thread could change the value of x between the read and write. Fixing this problem requires a re-design using a loop that iterates until x has not changed during the computation of f, where the yield annotation accentuates the programmer's expectation that other threads might change x during this computation:

```
void update_x() {
  boolean done = false;
  int y = x•;
  while (!done) {
    yield;
    int fy = f(y);
    if (x• == y) {
       x• = fy;
       done = true;
    } else {
       y = x•;
    }
  }
}
```

This code is rejected by our effect system, since there are additional points of thread interference not documented via yield annotations, such as between the comparison operation (x == y) and the subsequent update of x. To avoid this potential thread interference, the programmer adds additional synchronization:

```
void update_x() {
  boolean done = false;
  int y = x•;
  while (!done) {
    yield;
    int fy = f(y);
    acquire(m);
    if (x_ϵ == y) {
       x• = fy;
       done = true;
    } else {
       y = x_ϵ;
    }
    release(m);
  }
}
```

In this final version of update_x, the lock m is held for all writes to x, but not for all reads. Hence, as illustrated by the conflict tags, the write to x may be in a race with a concurrent read. Similarly, the first read of x without the lock held may race with a concurrent write. However, subsequent reads of x while holding the lock are race-free.

Based on these conflict tags, this final version of update_x is accepted by the effect system, and so can be assumed to execute cooperatively. Informal cooperative reasoning then argues that thread interference at the yield annotation is benign, since it simply causes the loop to retry.

In summary, this short development example illustrates the kinds of benefits we hope our effect system might provide by highlighting exactly where thread interference (and hence heisenbugs) may occur. In particular, this example shows how our effect system is helpful for non-atomic methods (such as update_x) that are not well-served by existing tools that focus exclusively on atomicity.

5.2 Atomic as an Interface-Level Specification

To illustrate the compatible and complementary nature of atomic and yield specifications, we consider the implementation of a linked list of Nodes. We assume a primitive operation newNode of type Int × Node → Node that allocates and initializes a new Node. The effect specification for newNode is AB.

The following function add adds a new element onto the linked list list, which is protected by the lock m. Since each access to list is race-free, the effect of add is $(\text{AR}; \text{AL}) = \text{AN}$, an atomic non-mover; we annotate each function definition with its effect.

```
Node list;
Lock m;

AN void add(int v) {
  acquire(m);
  list_ϵ = newNode(v, list_ϵ)
  release(m);
}
```

We next consider two functions, add2 and add2s, each of which adds two elements to list, but with different synchronization policies. In particular, add2s guarantees that the two new elements are adjacent in the resulting list. The function add2 does not provide this guarantee, essentially because the yield between the two calls to add allows other threads to interpose other elements into the list.

```
CN void add2(int v, int w) {
  add(v);
  yield;
  add(w);
}
```

```
AN void add2s(int v, int w) {
  acquire(m);
  list_ε = newNode(v, list_ε);
  list_ε = newNode(w, list_ε);
  release(m);
}
```

This key difference between `add2` and `add2s` is captured by their effects, CN and AN, respectively, and in particular by the serializability component of these effects. The serializability effect C (compound) for `add2` means that multiple epochs may execute inside this function, and implies thread interference may complicate correctness reasoning in the calling context. In contrast, the serializability effect A (atomic) for `add2s` implies that `add2s` is free of interference from other threads. The absence of such thread interference points may greatly simplify correctness arguments by enabling serial reasoning at the call site.

6. Verifying Cooperability and Serializability

As a first step in verifying the correctness of our effect system, we formalize our assumptions about the conflict tags, the call tags, and the type environment.

A term t is *about to read* x if $t \equiv E[x_r]$. Similarly, t is *about to write* x if $t \equiv E[x_r := v]$. The conflict tags in a program P are *correct* if whenever $P \rightarrow_p^* H, M, T$:

1. If $T_i \equiv E[x_\epsilon]$, then no other thread in T is about to write x.
2. If $T_i \equiv E[x_\epsilon := v]$, then no other thread in T is about to read or write x.

Similarly, the call tags in P are *correct* if whenever $P \rightarrow_p^* H, M, T$ and $T_i \equiv E[(f(\overline{x})\ t)^F(\overline{v})]$, then $f \in F$.

A primitive p *right commutes* with a primitive q if for for all i and j such that $i \neq j$:

1. if $\mathcal{I}_p(\overline{v}, M_1, i) = \langle v', M_2\rangle$ and $\mathcal{I}_q(\overline{u}, M_2, j) = \langle u', M_3\rangle$ then there exists M_4 such that $\mathcal{I}_q(\overline{u}, M_1, j) = \langle u', M_4\rangle$ and $\mathcal{I}_p(\overline{v}, M_4, i) = \langle v', M_3\rangle$;
2. if $\mathcal{I}_p(\overline{v}, M_1, i) = \mathbf{wrong}$ and $\mathcal{I}_q(\overline{u}, M, j) = \langle u', M_2\rangle$ then $\mathcal{I}_p(\overline{v}, M_2, i) = \mathbf{wrong}$;
3. if $\mathcal{I}_p(\overline{v}, M_1, i) = \langle v', M_2\rangle$ and $\mathcal{I}_q(\overline{u}, M_2, j) = \mathbf{wrong}$ then $\mathcal{I}_q(\overline{u}, M_1, j) = \mathbf{wrong}$.

A type environment is *valid* if the effects of primitives are correct, in that the following two properties hold:

1. If $\Gamma(p) \sqsubseteq$ R then p right commutes with any primitive q.
2. If $\Gamma(p) \sqsubseteq$ L then any primitive q right commutes with p.

By the following preservation theorem, every state reachable from an initial, well-formed state via preemptive execution is well-formed.

THEOREM 1 (Preservation). *Let P be a program with correct call tags and let Γ a type environment. If $\Gamma \vdash P$ and $P \rightarrow_p P'$ then $\Gamma \vdash P'$.*

Our effect system guarantees that for any well-formed program, any preemptive execution is equivalent to some cooperative execution. Our proof of this correctness guarantee relies on the following reduction theorem [20], which is phrased in terms of a generic transition system.

We first present some notation for stating the reduction theorem. For any state predicate[3] $X \subseteq \mathit{ProgState}$ and transition relation $Z \subseteq \mathit{ProgState} \times \mathit{ProgState}$, by X/Z we mean the transition relation obtained by restricting Z to pairs whose first component is in X. Similarly, by $Z\backslash X$ we mean the restriction of Z to pairs whose second component is in X. We use $Y \circ Z$ to denote the composition of two transition relations Y and Z.

$$\begin{array}{rcl} X/Z & \stackrel{\text{def}}{=} & \{(a,b) \in Z \mid a \in X\} \\ Z\backslash X & \stackrel{\text{def}}{=} & \{(a,b) \in Z \mid b \in X\} \\ Y \circ Z & \stackrel{\text{def}}{=} & \{(a,c) \mid \exists b.\ (a,b) \in Y \text{ and } (b,c) \in Z\} \end{array}$$

For two transition relations $Y, Z \subseteq \mathit{ProgState} \times \mathit{ProgState}$, if $Z \circ Y \subseteq Y \circ Z$ then we say that Y *right-commutes* with Z, and also that Z *left-commutes* with Y.

THEOREM 2 (Reduction Theorem). *For all i, let $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ be sets of states, and $\rightharpoonup_i$ be a transition relation. Suppose for all i,*

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

and for all $j \neq i$,

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

Let $\mathcal{N}_i = \neg(\mathcal{R}_i \vee \mathcal{L}_i)$, $\mathcal{N} = \forall i.\ \mathcal{N}_i$, $\mathcal{W} = \exists i.\ \mathcal{W}_i$, $\rightharpoonup = \exists i.\ \rightharpoonup_i$, and $\twoheadrightarrow = \exists i.\ (\forall j \neq i.\ \mathcal{N}_j)/\rightharpoonup_i$. Suppose $p \in \mathcal{N}$ and $p \rightharpoonup^* q$. Then the following statements are true.

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

PROOF: See [20].

6.1 Verifying Cooperability

We now consider how to instantiate the reduction theorem for our setting. We begin by introducing the set WF of well-formed states and the set N_i of states where thread i is yielding (not in an epoch). The set N describes *yielding states* in which all threads are yielding:

$$\begin{array}{rcl} WF & = & \{\langle H, M, T\rangle \mid \Gamma \vdash \langle H, M, T\rangle\} \\ N_i & = & WF \cap \{\langle H, M, T\rangle \mid T_i \text{ is yielding}\} \\ N & = & WF \cap \{\langle H, M, T\rangle \mid \forall i.\ T_i \text{ is yielding}\} \end{array}$$

Each epoch must consist of a collection of right movers, followed by at most one non mover (which we call the *commit point* of the epoch), followed by a collection of left movers. To apply the reduction theorem, we must determine if each thread T_i is currently in the right-mover or left-mover part of an epoch. If $\Gamma \vdash T_i : \langle s, c\rangle$ where $c \sqsubseteq$ L, then thread i has passed its commit point, and subsequent operations of thread i (up to the next yield) must commute to the left to join with previous operations by that thread. Conversely, if $c \not\sqsubseteq$ L, then this epoch has not committed, subsequent operations of thread i may commute to the right to join with future operations by that thread.

[3] For notational convenience, this section exploits the isomorphism between sets and predicates.

Based on this intuition, we characterize whether each thread i is in the left-mover L_i or right-mover R_i part of an epoch as follows:

$$L_i = WF \cap \{\langle H, M, T\rangle \mid \Gamma \vdash T_i : \langle s, c\rangle \text{ and } c \sqsubseteq \text{L}\} \setminus N_i$$
$$R_i = WF \cap \{\langle H, M, T\rangle \mid \Gamma \vdash T_i : \langle s, c\rangle \text{ and } c \not\sqsubseteq \text{L}\} \setminus N_i$$

Using Theorem 1 and Theorem 2, we now have the necessary machinery to prove two fundamental correctness properties of our effect system. First, we consider any preemptive execution from a well-formed state P to a subsequent state Q. If P and Q are both yielding then Q can also be reached from P via a cooperative execution.

The second property of our effect system allows us to check program assertions on cooperative executions and conclude that the assertions will hold on preemptive executions as well. The set W_i describes states where thread i has gone wrong by misapplying a primitive, and a program state is *bad* if any thread has gone wrong:

$$W_i = WF \cap \{\langle H, M, T\rangle \mid T_i \equiv \mathbf{wrong}\}$$

We show that if a preemptive execution from a well-formed state P reaches a bad state Q, then there is a cooperative execution from P that also ends in a bad state. The proof of this property requires that once an epoch has reached its commit point, then it must be able to finish. A program P is *nonblocking* if whenever $P \rightarrow_p^* Q$ and $L_i(Q)$, then there exists Q' such that $Q \rightarrow_i^* Q'$ and $\neg L_i(Q')$.

THEOREM 3 (Cooperability). *Let P be a program with correct conflict and call tags and let Γ be a valid type environment such that $\Gamma \vdash P$. Suppose $N(P)$ and $P \rightarrow_p^* Q$. Then:*

1. *If $N(Q)$ then $P \rightarrow_c^* Q$.*
2. *If P is nonblocking and Q is bad then there exists Q' such that $P \rightarrow_c^* Q'$ and Q' is bad.*

PROOF. Suppose P and Q are states such that $N(P)$ and $P \rightarrow_p^* Q$. We apply the Reduction Theorem (Theorem 2) by substituting 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 $\rightarrow_i$ for $\rightharpoonup_i$, the relation $\rightarrow_c$ for $\twoheadrightarrow$, the state P for p, and the state Q for q. This substitution satisfies the six preconditions of the Theorem 2. From this instantiation of the Reduction Theorem, we prove Theorem 3 as follows.

1. Since $N(Q)$ by assumption, we get from the first part of the Reduction Theorem that $P \rightarrow_c^* Q$.
2. We show that for any program state S, if S is bad then there exists state S' such that $S \rightarrow_p^* S'$ and S' is bad and $\forall i.\ \neg L_i(S')$.

   For any state S, let $\ell(S) = \{i \mid L_i(S)\}$. The proof is by induction on $|\ell(S)|$. If $|\ell(S)| = 0$, the hypothesis is trivially true. Now assume that $|\ell(S)| > 0$, and pick some thread $i \in \ell(S)$.

   Since P is nonblocking, we have a state S'' such that $S \rightarrow_i^* S''$ and $\neg L_i(S'')$. We also have that $j \in \ell(S)$ iff $j \in \ell(S'')$ and also $W_j(S)$ iff $W_j(S'')$ for all $j \neq i$.Therefore $\ell(S'') = \ell(S) \setminus \{i\}$ and also $\exists j.\ W_j(S'')$. By the induction hypothesis, there is a state S' such that $S'' \rightarrow_p^* S'$, $\exists j.\ W_j(S')$, and $\forall i.\ \neg L_i(S')$. Therefore, we have $S \rightarrow_p^* S'$: a bad state moving to another bad state.

   Since Q is bad by assumption, we use the previous paragraph's result to conclude that there exists a state Q'' such that $Q \rightarrow_p^* Q''$ and Q'' is bad and $\forall i.\ \neg L_i(Q'')$. We may now apply the second part of the Reduction Theorem: there exists a state Q' such that Q' is bad and $P \rightarrow_c^* Q'$.

6.2 Verifying Serializability

A program may have both yield annotations and atomic block annotations. We show that every cooperative execution is also a serial execution: that is, atomic block annotations are satisfied by a cooperative execution.

THEOREM 4 (Cooperative Traces are Serial). *If $\Gamma \vdash P$ and $P \rightarrow_c Q$ then $P \rightarrow_s Q$.*

PROOF. The proof proceeds by contradiction. Let $P = \langle H, M, T\rangle$ and suppose that $P \rightarrow_c Q$ because $P \rightarrow_i Q$, and also that $P \not\rightarrow_s Q$ because T_j contains `in-atomic` $f$or some $j \neq i$.

From the rule [PROG], we have that $T_j \equiv E[\texttt{in-atomic}\ t]$ for some E, t. From $P \rightarrow_c Q$, we have that T_j is yielding. Thus $T_j \equiv E[\texttt{in-atomic}\ E'[\texttt{yield}]]$.

From the rule [YIELD], we have $\Gamma \vdash \texttt{yield} : \text{CY}$. Because T_j is well formed, we have $\Gamma \vdash E'[\texttt{yield}] : \kappa$, where $\text{CY} \sqsubseteq \kappa$.

From the antecedent in the rule [INATOMIC], T_j is ill-formed, contradicting our assumption that $\Gamma \vdash P$.

7. Related Work

Lipton [32] proposed reduction as a way to reason about concurrent programs without considering all interleavings. He focused primarily on reasoning about deadlocks. Doeppner [14], Back [8], and Lamport and Schneider [31] extended this work to allow proofs of general safety properties.

A number of tools have been developed for detecting race conditions, both statically and dynamically. Our previous work on `rccjava` [3] uses a type system to catch race conditions in Java programs. This approach has been extended [5, 10, 11] and adapted to other languages [25]. Many other static race detectors have been developed, including Warlock [44], Locksmith [39], Chord [36], and [16, 36, 48]. Eraser [41] detects race conditions dynamically, and this approach has been refined to eliminate false positives, reduce overhead, and handle additional synchronization idioms, as in [37, 46], or combined with happens-before reasoning [38, 53]. Other approaches have combined dynamic analysis with a global static analysis to improve precision and performance [12, 15, 47].

A variety of tools have been developed to check for atomicity violations, both statically and dynamically. The Atomizer [18] uses Lipton's theory of reduction [32] to check whether steps of each transaction conform to a pattern guaranteed to be serializable. Wang and Stoller developed an alternative block-based algorithm [51] and also more precise commit-node algorithms [50]. Prior work has developed expressive type and effect systems for verifying atomic specifications in Java programs [5, 19–21, 23, 40, 49]; we believe these ideas could be adapted to also verify yield annotations.

Our yield effect system was inspired by work on automatic mutual exclusion, a recently proposed concurrency programming model based on having mutual exclusion by default [1, 2, 29]. A key difference is that automatic mutual exclusion *enforces* a yield annotation at run time, while our effect system uses static analysis to *check* that yield annotations provide equivalent cooperative execution.

Several recent studies have focused on lightweight transactions [26, 30, 42] and automatic generation of synchronization code from high-level specifications [13, 28, 35, 45]. Much of this work is orthogonal to ours, and while these approaches offer a promising alternative to explicit concurrency control, we believe that a combination of the two approaches will be the most effective programming model for the foreseeable future.

8. Discussion and Future Work

This paper proposes `yield` as a specification of where thread interference may occur, and presents some preliminary technical results. A number of open issues remain for future work.

One immediate question is how well this effect system would scale to larger programs, assuming we had appropriately precise flow and race condition information for those programs. A limitation of our current system is that it cannot capture situations where the effect of a program expression is context- or state-dependent, as happens with re-entrant lock acquires, which are essentially nops.

More generally, we would like a combined type-and-effect system that incorporates control-flow, race condition, and cooperability reasoning. Prior work has developed similar type-and-effect systems focused on verifying atomicity in class-based languages such as Java [5, 19–21, 23, 40, 49]. These system include a variety of features to handle large-scale programs, such as conditional effects (to handle re-entrant locking), dependent classes parameterized by locks, type and effect inference, etc. We expect that similar techniques could be developed for verifying yield annotations. Experience with such systems suggests that type checking (and possibly type inference) would likely scale to large programs, but may encounter some difficulty in verifying some less common synchronization idioms.

Prior work has explored dynamically verifying and/or inferring atomicity specifications. An interesting idea is to use similar techniques to dynamically infer and insert yield specifications into the source code of large multithreaded software systems. Such an inference system would encounter some heuristic design choices about where to place yield annotations. As one example, should a yield annotation at a call boundary be inserted into the caller or callee?

Nonetheless, these automatically inferred yield specifications could provide a valuable reminder to the programmer during ongoing code maintenance of exactly where thread interference may occur, and where the "natural" expectation of sequential semantics is violated. Keeping yield annotations in source code would also allow (static or dynamic) tools to highlight when additional and possibly unintentional thread interference is introduced during maintenance activities.

Yield annotations could also improve the efficiency of model checkers and other code verification tools, by enabling them to consider fewer interleavings. Although partial-order methods [24] already play a similar role, programmer-specified yield annotations might provide tighter bounds on thread interference.

9. Conclusion

Multithreading is increasingly used to improve performance and scalability of programs on today's multicore machines. However, reasoning about multithreaded programs is a difficult endeavor, due to the explosive number of thread interleavings that must be considered for program correctness.

Atomicity is a specification which identifies code blocks that are serializable, that is, equivalent to a serial execution. Although the benefits of atomicity have been explored in prior work, we argue that atomicity has shortcomings which limit its utility. Specifically, atomicity forces two distinct modes of reasoning (sequential/multithreaded) on the programmer, and still exposes the full difficulty of reasoning about concurrency outside of atomic blocks.

We believe that yield annotations offer a complementary approach that provides a single, universal mode of cooperative reasoning throughout the program. This paper introduces a static effect system which guarantees that any preemptive execution of a well-formed program is guaranteed to behave equivalently to some cooperative execution. We hope that our effect system helps to reduce the difficulty of reasoning about multithreaded programs.

Acknowledgements: We thank Stephen Freund for preliminary discussions, Jevgenia Smorgun for her contributions to a class project [43] that explored some of these ideas, and Martin Abadi, Thomas Austin, and Caitlin Sadowski for helpful comments on this paper. This work was supported in part by NSF Grants 0707885 and 0905650.

References

[1] M. Abadi. Automatic mutual exclusion and atomicity checks. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 510–526. Springer, 2008.

[2] M. Abadi and G. Plotkin. A model of cooperative threads. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 29–40, New York, NY, USA, 2009. ACM.

[3] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[4] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302. USENIX Association, 2002.

[5] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[6] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized runtime race detection and atomicity checking using partial discovered types. In *International Conference on Automated Software Engineering (ASE)*, pages 233–242, 2005.

[7] R. M. Amadio and S. D. Zilio. Resource control for synchronous cooperative threads. In *International Conference on Concurrency Theory (CONCUR)*, pages 68–82. Springer-Verlag, 2004.

[8] R.-J. Back. A method for refining atomicity in parallel algorithms. In *PARLE 89: Parallel Architectures and Languages Europe*, volume 366 of *Lecture Notes in Computer Science*, pages 199–216, 1989.

[9] G. Boudol. Fair cooperative multithreading. In *International Conference on Concurrency Theory (CONCUR)*, pages 272–286, 2007.

[10] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 56–69, 2001.

[11] C. Boyapati, R. Lee, and M. Rinard. A type system for preventing data races and deadlocks in Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, 2002.

[12] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridhara. Efficient and precise datarace detection for multithreaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, 2002.

[13] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *International Conference on Software Engineering (ICSE)*, pages 442–452, 2002.

[14] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 155–169, 1977.

[15] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–255, 2007.

[16] D. R. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[17] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

[18] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[19] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003.

[20] C. Flanagan and S. Qadeer. Types for atomicity. In *Workshop on Types in Language Design and Implementation*, pages 1–12, 2003.

[21] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *Workshop on Types in Language Design and Implementation(TLDI)*, pages 47–58, 2005.

[22] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, 2005.

[23] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(4), 2008.

[24] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[25] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation(TLDI)*, pages 13–25, 2003.

[26] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, 2003.

[27] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004.

[28] M. Hicks, J. S. Foster, and P. Pratikakis. Inferring locking for atomic sections. In *Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[29] M. Isard and A. Birrell. Automatic mutual exclusion. In *HOTOS'07: Proceedings of the 11th USENIX workshop on hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[30] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, 2005.

[31] L. Lamport and F. B. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 1989.

[32] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[33] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 47–57, 1988.

[34] D. Marino and T. D. Millstein. A generic type-and-effect system. In *Workshop on Types in Language Design and Implementation(TLDI)*, pages 39–50, 2009.

[35] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 346–358, 2006.

[36] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319, 2006.

[37] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[38] E. Pozniansky and A. Schuster. MultiRace: Efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[39] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 320–331, 2006.

[40] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 83–94, 2005.

[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[42] N. Shavit and D. Touitou. Software transactional memory. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[43] J. Smorgun and J. Yi. An effect system for checking consistency of synchronization and yields. Technical Report UCSC-SOE-09-33, Department of Computer Science, University of California at Santa Cruz, 2009.

[44] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[45] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 334–345, 2006.

[46] C. von Praun and T. Gross. Object race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 70–82, 2001.

[47] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 115–128, 2003.

[48] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 205–214, 2007.

[49] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 61–71, 2005.

[50] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146, 2006.

[51] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, 32: 93–110, Feb. 2006.

[52] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, 2005.

[53] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 221–234, 2005.