# Exploiting Purity for Atomicity

Cormac Flanagan, *Member*, *IEEE*, Stephen N. Freund, and Shaz Qadeer

**Abstract**—Multithreaded programs often exhibit erroneous behavior because of unintended interactions between concurrent threads. This paper focuses on the noninterference property of *atomicity*. A procedure is atomic if, for every execution, there is an equivalent *serial* execution in which the actions of the atomic procedure are not interleaved with actions of other threads. This key property makes atomic procedures amenable to sequential reasoning techniques, which significantly facilitates subsequent validation activities such as code inspection and testing. Several existing tools verify atomicity by using commutativity of actions to show that every execution *reduces* to a corresponding serial execution. However, experiments with these tools have highlighted a number of interesting procedures that, while intuitively atomic, are not reducible. In this paper, we exploit the notion of *pure* code blocks to verify the atomicity of such irreducible procedures. If a pure block terminates normally, then its evaluation does not change the program state and, hence, these evaluation steps can be removed from the program trace before reduction. We develop a static typed-based analysis for atomicity based on this insight, and we illustrate this analysis on a number of interesting examples that could not be verified using earlier tools based purely on reduction.

**Index Terms**—Atomicity, purity, reduction, concurrent programs.

## 1 Introduction

Multiple threads of control are widely used in software development because they help reduce latency and provide better utilization of multiprocessor machines. However, reasoning about the correctness of multithreaded code is complicated by the nondeterministic interleaving of threads and the potential for unexpected interference between concurrent threads. Since exploring all possible interleavings of the various threads is clearly impractical, techniques for specifying and controlling the interference between concurrent threads are crucial for the development of reliable multithreaded software.

Previous work has addressed this problem by devising type systems [1], [2] and other static [3] and dynamic [4] checking tools for detecting *race conditions*. A race condition occurs when two threads simultaneously access the same data variable, and at least one of the accesses is a write.

Unfortunately, the absence of such race conditions is not sufficient to ensure the absence of errors due to unexpected thread interactions. To illustrate this point, consider the procedure `bad_increment` (Fig. 1) in which the data variable `x` is protected by the lock `m` and `t` is a thread-local variable.

This code does not have any race conditions, a property that can be easily verified with existing tools. However, executing this code fragment may not have the intended effect of atomically incrementing `x` by 1. For example, if $n$ such code blocks execute concurrently, the variable `x` may be incremented by any number between 1 and $n$.

A stronger noninterference property is required to ensure proper behavior, namely, *atomicity*. A procedure (or code block) is atomic if, for every (arbitrarily interleaved) program execution, there is an equivalent execution with the same overall behavior where the atomic procedure is executed serially, that is, the procedure's execution is not interleaved with actions of other threads.

The notion of atomicity provides several benefits for multithreaded software development:

- The noninterference guarantee provided by atomicity reduces the challenging problem of reasoning about an atomic procedure's behavior in a *multithreaded* context to the simpler problem of reasoning about the procedure's *sequential* behavior. The latter problem is significantly more amenable to standard techniques such as manual code inspection, dynamic testing, and static analysis.
- Atomicity is a natural methodology for multithreaded programming, and experimental results indicate that many existing procedures and library interfaces already follow this methodology [5].
- Many synchronization errors can be detected as violations of atomicity.

The notions of atomicity and race-freedom are closely related, and both are often achieved in practice using synchronization mechanisms such as mutual-exclusion locks, reader-writer locks, or semaphores. However, as the example above illustrates, race-freedom is not, by itself, sufficient to ensure atomicity.

### 1.1 Verifying Atomicity via Reduction

Recently, a number of analyses have been developed for verifying atomicity, using techniques such as theorem proving [6], static type systems [7], [8], dynamic analysis [5], [9], and model checking [10]. All of these approaches

- *C. Flanagan is with the Computer Science Department, University of California, Santa Cruz, 1156 High Street, Santa Cruz, CA 95064. E-mail: cormac@cs.ucsc.edu.*
- *S.N. Freund is with the Computer Science Department, Williams College, Williamstown, MA 01267. E-mail: freund@cs.williams.edu.*
- *S. Qadeer is with Microsoft Research, One Microsoft Way, Redmond, WA 98052. E-mail: qadeer@microsoft.com.*

*Manuscript received 27 Oct. 2004; revised 14 Feb. 2005; accepted 29 Mar. 2005; published online DD Mmmm, YYYY.*
*Recommended for acceptance by G. Rothermel.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log NumberTESI-0252-1004.*

```
void bad_increment() {
    acquire(m);
    t = x;
    release(m);
    acquire(m);
    x = t+1;
    release(m);
}
```

Fig. 1. Procedure `bad_increment`.

use *reduction* [11], [12], which is based on the notion of right and left movers.

An action $a$ is a *right mover* if, whenever $a$ is followed by any action $b$ of a different thread, the actions $a$ and $b$ can be swapped without changing the resulting state. Similarly, an action $a$ is a *left mover* if, whenever $a$ follows an action $b$ of a different thread, the actions $a$ and $b$ can be swapped, again without changing the resulting state.

Suppose a code block contains zero or more right movers followed by a single atomic action (that need not commute with steps of other threads) followed by zero or more left movers. Then, an execution where this code block has been fully executed can be *reduced* (by commuting noninterfering actions) to yield an equivalent *serial* execution, where the actions of the code block are performed contiguously.

To illustrate this notion of reduction, consider the revised implementation of `increment` in Fig. 2. In this procedure, the operation `acquire(m)` is a right mover because no other thread can manipulate `m` after it has been acquired, and the operation `release(m)` is a left mover for similar reasons. Moreover, if all threads access `x` only while holding the lock `m`, then reads from and writes to `x` are both right-movers and left-movers since no other thread can concurrently access the variable `x`. In contrast, if there may be concurrent reads or writes when accessing `x`, then that operation cannot commute since program behavior may change when accesses to the same variable are swapped in the execution trace.

As illustrated in Fig. 3, we can reduce any execution of `increment` interleaved with arbitrary steps ("`Z`") from other threads into an equivalent serial execution and, hence, `increment` is atomic.

In contrast, if we perform reduction on `bad_increment`, we are left with an execution trace like that shown in Fig. 4, in which a step `Z` from a different thread appears in the middle of the steps of the procedure call. That step cannot commute to the left because the preceding `release` operation is not a right-mover, and it cannot commute to the right because the succeeding `acquire` operations is not a left-mover. Therefore, `bad_increment` is not atomic.

```
atomic void increment() {
    acquire(m);
    t = x;
    x = t + 1;
    release(m);
}
```

Fig. 2. Procedure `increment`.

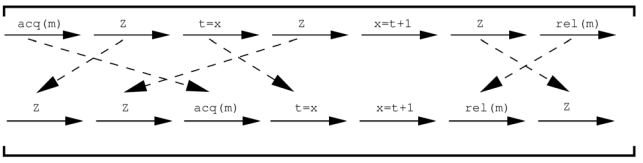



Fig. 3. Reduction of an execution of `increment`.

## 1.2 Purity

Reduction suffices to verify the atomicity of many procedures that use straightforward synchronization patterns, but it is often inadequate for procedures that use more subtle synchronization. A concrete example of this limitation is the procedure `busy_acquire` shown in Fig. 5, which uses a combination of busy-waiting and a compare-and-set (`CAS`) operation to acquire a mutually exclusive lock `m` (represented as a Boolean).

The operation `CAS(m,0,1)` has no effect and returns `false` if `m` $\neq$ `0`. However, if `m` $=$ `0`, then the operation `CAS(m,0,1)` sets `m` to `1` and returns `true`. This `CAS` operation does not commute with operations of concurrent threads since it inspects and potentially updates the shared variable `m`. Hence, any execution of `busy_acquire` where the loop iterates multiple times cannot be *reduced* to a serial execution, and previous tools based purely on reduction cannot verify the atomicity of `busy_acquire`. In particular, our previous type and effect system for atomicity [8] cannot verify the atomicity of irreducible procedures like `busy_acquire`.

The model checking approach described in [13] can verify the atomicity of `busy_acquire`, but is limited by the state-explosion problem. Similarly, the Calvin-R checker [6] can also verify such atomicity properties, but it focuses on checking more complete functional specifications of concurrent programs via theorem proving. As a result, it has a higher annotation overhead and analysis complexity than the technique in this paper.

In this paper, we present a lightweight and scalable static analysis for verifying the atomicity of irreducible procedures such as `busy_acquire`. We formalize our analysis as an effect system (essentially, a collection of syntax-directed rules). This effect system is analogous to traditional type systems, except that it reasons about effects (which describe computations) as opposed to types (which describe values).

A key novelty of our analysis is the exploitation of *purity* when reasoning about atomicity. Essentially, a code block is pure if, whenever it evaluates without interruption by other threads and terminates normally, it does not change the program state. This restriction does not apply when the block terminates *abruptly*, for example, via a `break` or `return` statement. The body of the while loop in `busy_acquire` is pure since, if it updates `m`, it immediately terminates abruptly via the `break` statement. Otherwise, control is returned to the loop head without changing the program store. We introduce the `pure-while` statement to

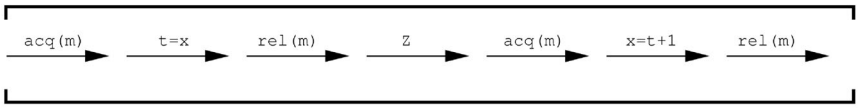



Fig. 4. Reduced execution of `bad_increment`.

indicate a pure loop and rewrite the `busy_acquire` procedure, as in Fig. 6.

The intuition behind the reasoning of our analysis for `busy_acquire` is shown in Fig. 7. The figure shows an execution of `busy_acquire` consisting of three normally terminating loop iterations in which the `CAS` fails, followed by an abruptly terminating iteration in which the `CAS` operation succeeds.

Since the normally terminating iterations do not change the program state, our verification technique essentially removes them from the execution trace to yield a trace containing a single loop iteration in which the `CAS` operation succeeds. Every execution of `busy_acquire` is serializable in this manner, and our analysis can conclude that `busy_acquire`, although irreducible, is atomic.

### 1.3 Abstraction via Purity

A more interesting example of our analysis technique is the procedure `alloc` of Fig. 8, which searches for a free disk block. The flag `free[i]` indicates whether the `i`th disk block is currently unused, and this flag is protected by the mutually exclusive lock `l[i]`. When `alloc` identifies a free block, it allocates the block by setting the appropriate bit to false and returns the index of that block. The `alloc` procedure returns $-1$ if it fails to find a free block.

This procedure is not actually serializable since there exist some nonserial executions of this procedure that are not equivalent to any serial executions. In particular, a concurrent thread could ensure that there is always at least one free block at any point in time, yet the sequential search performed by `alloc` could still fail to find a free block. Thus, the concrete implementation of `alloc` is not atomic, and this lack of atomicity significantly complicates reasoning about the behavior or correctness of `alloc`. However, `alloc` *is* atomic in an abstract sense because any execution performs the atomic action of either allocating a block or returning $-1$ if no free block was found.

In order to facilitate sequential reasoning for nonatomic procedures such as `alloc`, this paper introduces the notion of *abstract atomicity* and shows that the procedure `alloc` is atomic under a more permissive or *abstract* semantics. Recall that a `pure` block is not allowed to *modify* the program state during normal execution. Under the abstract semantics, normal execution of a `pure` block is not allowed to *read* the program state either. Specifically, any read of a shared variable within a normally terminating pure block returns an arbitrary value rather than the variable's true value. (This notion of abstraction generalizes our eariler approach [43].) Thus, the abstract semantics introduces additional nondeterminism and admits additional execution sequences that could occur if the code is interleaved with threads that make arbitrary updates to shared variables.

Consider an uninterrupted (serial) execution of the `pure` block in the `i`th iteration of the loop in `alloc`. Under the concrete semantics, such an execution may terminate normally only from a state in which `free[i]` is `false`. But, under the abstract semantics, such an execution is allowed to terminate normally from any program state since the read of `free[i]` returns an arbitrary value. Despite this additional nondeterminism, every serial execution of `alloc` under the abstract semantics satisfies the correctness specification of returning either $-1$ or the index of a free block. This correctness property can be verified using sequential reasoning techniques.

Given that every *serial* abstract execution of `alloc` is correct, the contribution of our analysis is to verify that every possible *interleaved* abstract execution of `alloc` yields the same behavior as some serial abstract execution, thus allowing us to conclude that all interleaved executions of `alloc` are also correct. Thus, our analysis enables sequential reasoning about behavior and correctness of abstractly atomic (though not atomic) procedures such as `alloc`. In contrast, previous techniques achieved this goal (of enabling sequential reasoning) only for procedures that are both atomic and reducible.

The central intuition behind the reasoning performed by our analysis to verify abstract atomicity is illustrated graphically in the execution traces for `alloc` shown in Fig. 9. The first trace contains an execution of `alloc` that succeeds on the second loop iteration, interleaved with arbitrary actions "`Z`" of concurrent threads. We show only steps of `alloc` that access shared variables. By reduction, we can prove that each individual loop iteration is reducible. Since the first execution of the `pure` block is normally terminating and, hence, does not read or modify any shared variable, we replace it with `skip` in the execution sequence. The `skip` command does not modify or read the program state. At this point, we apply reduction a second time to create an equivalent serial execution.

```
atomic void busy_acquire() {
    while (true) {
        if (CAS(m,0,1)) break;
    }
}
```

Fig. 5. Procedure `busy_acquire`.

```
atomic void busy_acquire() {
    pure-while (true) {
        if (CAS(m,0,1)) break;
    }
}
```

Fig. 6. Procedure `busy_acquire` with a `pure` loop.

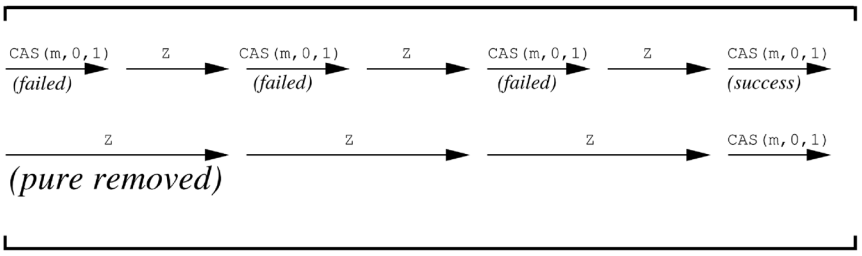



Fig. 7. Execution trace for `busy_acquire`.

Note that our analysis facilitates the verification of correctness properties that hold in both the standard and abstract semantics, but is not applicable to correctness properties that do not hold in the abstract semantics.

### 1.4 Abstraction via Instability

Our analysis also supports *unstable* variables, such as performance counters, which do not affect program correctness. These variables are typically not protected by locks and have race conditions on them. Consequently, accesses to these variables do not commute. Our analysis verifies atomicity with respect to an abstract semantics in which every write to an unstable variable writes a nondeterministic value and every read returns a nondeterministic value. Under this abstract semantics, reads and writes of unstable variables both right and left commute. For example, a program may use an unstable `packetCount` variable to record the number of packets received for tracking performance. Operations on that variable do not affect the abstract atomicity of the code in which they appear. We present a complete example in Section 4.4.

**Outline**. The presentation of our results proceeds as follows: The following section introduces an idealized language that we use for studying atomicity. Section 3 presents the effect system for atomicity, and Section 4 illustrates this analysis on a number of example programs. Section 5 discusses how to handle thread-local variables.

```
atomic int alloc() {
    int i = 0;
    int r = -1;
    while (i < max) {
        pure {
            acquire(l[i]);
            if (free[i]) {
                free[i] = false;
                release(l[i]);
                r = i;
                break;
            }
            release(l[i]);
        }
        i++;
    }
    return r;
}
```

Fig. 8. Procedure `alloc`.

We discuss related work in Section 6 and conclude with a discussion of future directions in Section 7. The Appendices contain a formal definition of an effect system for checking purity, the operational semantics of our idealized language, and a sketch of the correctness proof for our analysis.

## 2 The Language CAP

We formalize our ideas in terms of CAP, a small, imperative, multithreaded language with higher-order functions and dynamic thread creation. In essence, CAP is a restricted subset of C, extended with facilities for reasoning about atomicity and purity.

CAP expressions include values, variable reference and assignment, primitive and function applications, conditionals, and `let`-expressions: See Fig. 10. The `fork` $e$ expression creates a new thread for the evaluation of $e$. Values are constants and function definitions. Constants must include integer constants, but are otherwise unspecified. The definition $f(\bar{x})\ e$ introduces a function named $f$. The formal parameters $\bar{x}$ are bound within the body $e$, and they may be $\alpha$-renamed in the usual fashion. For generality, we leave the set of primitives unspecified, but they might include, for example, synchronization primitives that create, acquire, and release mutual exclusion locks. We assume the set of primitives also include arithmetic operations and `assert`.

In addition to terminating normally and yielding a resulting value, the evaluation of a CAP expression can also terminate *abruptly* via the `break` construct, which transfers control from the current expression to the end of the closest dynamically enclosing `block` construct. The construct `loop` $e$ repeatedly evaluates $e$ until $e$ `break`'s to an enclosing `block`.

To facilitate our atomicity analysis, expressions can be annotated with the keyword `pure`. The keyword `pure` states that, when the expression $e$ is evaluated and terminates normally, that evaluation does not change the program state. (Only abruptly terminating evaluations of a pure expression are allowed to change the program state.) If a `pure` expression temporarily changes the program state, for example, by acquiring a lock, then it must restore the state by releasing the lock before terminating normally. Under the abstract semantics, pure expressions are not allowed to read shared variables either. If a pure block attempts to read a shared variable, an arbitrary value is returned. To simplify our correctness proof, we also require a pure block to evaluate to a constant (rather than a function).

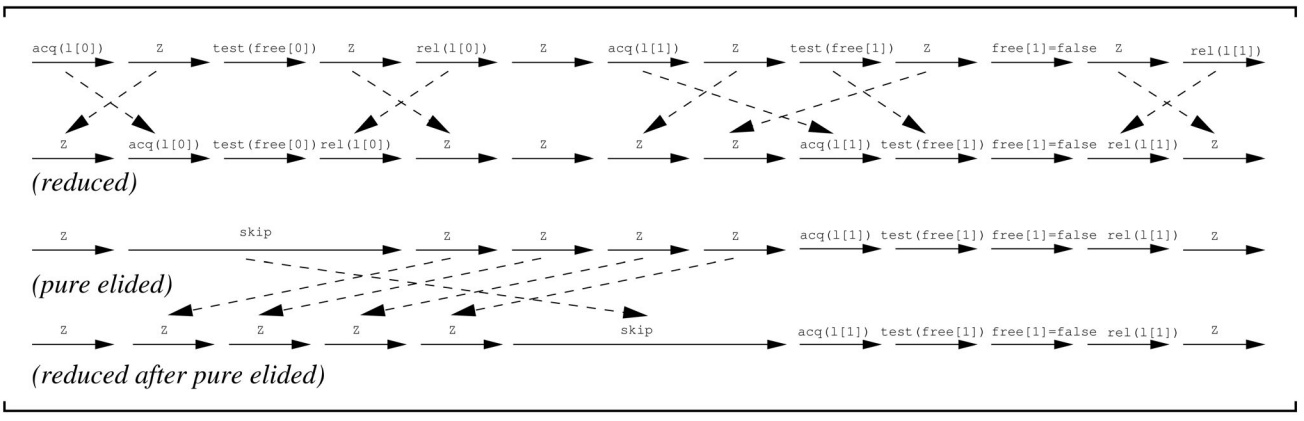



Fig. 9. Execution trace for `alloc`.

The language CAP supports unstable variables, so the set of variable names is divided into stable and unstable variables. By convention, unstable variable names begin with "_". The formal semantics of CAP is defined in Appendix B.

We introduce syntactic shorthands for some common constructs.

$$\begin{array}{rcl} e_1; e_2 & \equiv & \texttt{let}\ x = e_1 \in e_2 \quad \text{for } x \text{ not free in } e_2 \\ \texttt{while}\ e_1\ e_2 & \equiv & \texttt{block loop}\ \{\texttt{if}\ e_1\ e_2\ \texttt{break}\} \\ \texttt{pure-while}\ e_1\ e_2 & \equiv & \texttt{block loop pure}\ \{\texttt{if}\ e_1\ e_2\ \texttt{break}\} \end{array}$$

Note that if $e_1$ and $e_2$ are pure, then `while` $e_1\ e_2$ and `pure-while` $e_1\ e_2$ are semantically equivalent (that is, replacing `pure-while` $e_1\ e_2$ in a program with `while` $e_1\ e_2$ does not change the observable behavior of the program).

To simplify our presentation, the CAP effect system does not reason about race conditions, control flow, or purity since these topics can be addressed by other analyses. Instead, we assume the program has already been annotated and checked by alternative analyses as follows:

1. Each variable access (read or write) has a *conflict tag*, which is $\bullet$ if that access may be involved in a race condition on a stable variable and is $\epsilon$ otherwise. Thus, all accesses to unstable variables or correctly synchronized stable variables will have conflict tag $\epsilon$. Existing analysis techniques [2], [3], [14], [15], [16] can be used to infer these conflict tags.

$$\begin{array}{rcrcl} e & \in & \mathit{Expr} & ::= & v \mid x_r \mid x_r := e \mid p(\overline{e}) \mid e^F(\overline{e}) \\ & & & \mid & \texttt{if}\ e\ e\ e \mid \texttt{loop}\ e \mid \texttt{fork}\ e \\ & & & \mid & \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{block}\ e \mid \texttt{break} \\ & & & \mid & \texttt{atomic}\ e \mid \texttt{pure}\ e \\ v & \in & \mathit{Value} & ::= & c \mid f(\overline{x})\ e \\ r & \in & \mathit{Tag} & ::= & \bullet \mid \epsilon \\ x & \in & \mathit{Var} & = & \mathit{StableVar} \uplus \mathit{UnstableVar} \\ f & \in & \mathit{FnName} & & \\ F & \in & 2^{\mathit{FnName}} & & \\ p & \in & \mathit{Prim} & & \\ c & \in & \mathit{Const} & & \end{array}$$

Fig. 10. Syntax.

2. Each function call $e^F(\overline{e})$ has a *call tag* $F$ denoting the set of functions that may be invoked by that call. These call tags can be computed by a standard flow analysis.
3. Each `pure` $e$ expression is side-effect-free when $e$ evaluates normally. We present an effect system to check purity in the Appendix. Nielson et al. [17] provide a general overview of other effect-based techniques for tracking side effects, and these may be extended for our purposes as well.

We also assume programs being checked have passed a conventional type checker to catch basic type errors, such as performing an arithmetic operation on nonnumeric arguments. Factoring these other issues enables us to focus on the key aspects of this work without the added complexity of these other analyses. The core focus of our analysis is on verifying that every expression or procedure that is annotated as `atomic` is, in fact, serializable.

## 3 Effect System

We formalize our static analysis for abstract atomicity as an effect system. Previous type and effect systems [8], [7] could only verify the atomicity of procedures that are reducible. By introducing optionally executed pure blocks and unstable variables, our effect system can also verify many interesting irreducible procedures, such as those in Sections 1 and 4, are still abstractly atomic.

Each expression in our language can terminate either normally (by evaluating to a value) or abruptly (via `break`). For each termination mode, our effect system assigns to each expression an *atomicity* from the following set:

$$a, b, c \in \mathit{Atomicity} = \{R, L, B, \bot, A, \top\}.$$

This atomicity identifies whether the evaluation of the expression

- right-commutes with operations of other threads (R),
- left-commutes with operations of other threads (L),
- both right and left-commutes (B),
- cannot terminate in that mode ($\bot$),
- can be viewed as a single atomic action (A), or
- exhibits none of these properties ($\top$).

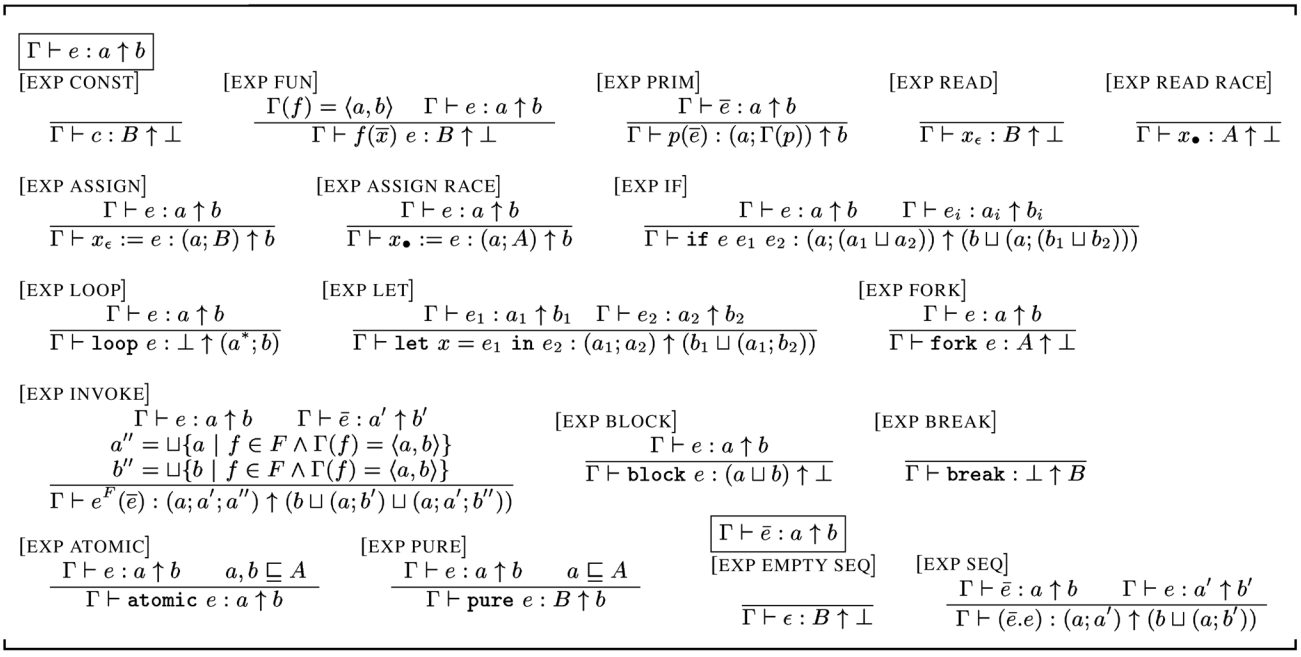
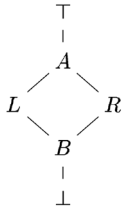



Fig. 11. Effect system.

Atomicities are partially ordered by the relation $\sqsubseteq$, as follows:




Let $\sqcup$ denote the join operator based on this ordering. If atomicities $a_1$ and $a_2$ reflect the normal-termination behavior of expressions $e_1$ and $e_2$, respectively, then the *sequential composition* $a_1; a_2$ reflects the normal-termination behavior of $e_1; e_2$ and is defined by the following table. For example, the Sequential composition of a right mover operation and an atomic operation is atomic.

| ; | $\bot$ | $B$ | $L$ | $R$ | $A$ | $\top$ |
|---|---|---|---|---|---|---|
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |
| $B$ | $\bot$ | $B$ | $L$ | $R$ | $A$ | $\top$ |
| $L$ | $\bot$ | $L$ | $L$ | $\top$ | $\top$ | $\top$ |
| $R$ | $\bot$ | $R$ | $A$ | $R$ | $A$ | $\top$ |
| $A$ | $\bot$ | $A$ | $A$ | $\top$ | $\top$ | $\top$ |
| $\top$ | $\bot$ | $\top$ | $\top$ | $\top$ | $\top$ | $\top$ |

Similarly, if atomicity $a$ reflects the normal-termination behavior of $e$, then the *iterative closure* $a^*$ reflects the normal-termination behavior of executing $e$ zero or more times and is defined by

$$\begin{aligned} \bot^* &= B \\ A^* &= \top \\ a^* &= a \text{ for } a \in \{B, L, R, \top\}. \end{aligned}$$

Note that

1. sequential composition is associative and $B$ is the left and right identity of this operation,
2. iterative closure is idempotent, and
3. sequential composition distributes over joins.

An effect environment $\Gamma$ maps each function name to a pair of atomicities $\langle a, b\rangle$ that describe the function's behavior under normal and abrupt termination. In addition, $\Gamma$ also maps each primitive operation to a corresponding atomicity (note that primitives never terminate abruptly):

$$\begin{aligned} \Gamma \quad : \quad & (FnName \rightarrow Atomicity \times Atomicity) \\ \cup \quad & (Prim \rightarrow Atomicity). \end{aligned}$$

The atomicity of some common primitives are shown below. The operations `assert`, `+`, and `new_lock` do not interfere with steps of other threads and, so, have atomicity $B$. The operation $\texttt{CAS}(\texttt{m}, v_1, v_2)$ is atomic (non-mover) because if it updates `m`, that update could be observed by a preceding or succeeding operation of another thread.

$$\begin{aligned} \Gamma(\texttt{assert}) &= B \\ \Gamma(\texttt{CAS}) &= A \\ \Gamma(+) &= B \\ \Gamma(\texttt{new\_lock}) &= B \\ \Gamma(\texttt{acquire}) &= R \\ \Gamma(\texttt{release}) &= L. \end{aligned}$$

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

$$\Gamma \vdash e : a \uparrow b$$

This judgment states that the expression $e$ has atomicity $a$ under normal termination and atomicity $b$ under abrupt termination. The rules defining these judgments shown in Fig. 11 are mostly straightforward. For example, the "evaluation" of a constant terminates normally, does not interfere with other threads, and cannot terminate abruptly.

[EXP CONST]

$$\frac{}{\Gamma \vdash c : B \uparrow \bot}$$

The atomicity of a variable read $x_r$ depends on the conflict tag $r$. If $r = \epsilon$, then this read commutes with steps of other threads and, so, has normal atomicity $B$. If $r = \bullet$, then this read has normal atomicity $A$, indicating that it is an atomic action that may not commute with steps of other threads. The rules for variable writes are similar.

[EXP READ] [EXP READ RACE]

$$\frac{}{\Gamma \vdash x_\epsilon : B \uparrow \bot} \qquad \frac{}{\Gamma \vdash x_\bullet : A \uparrow \bot}$$

The rule [EXP LET] states that the normal atomicity of a let expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is the sequential composition $a_1; a_2$ of the normal atomicities of $e_1$ and $e_2$. The abrupt atomicity of a let expression reflects the places where the let expression could `break`.

[EXP LET]

$$\frac{\Gamma \vdash e_1 : a_1 \uparrow b_1 \qquad \Gamma \vdash e_2 : a_2 \uparrow b_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (a_1; a_2) \uparrow (b_1 \sqcup (a_1; b_2))}$$

Similarly, in the [EXPIF] rule, the abrupt atomicity reflects all the ways in which an expression may `break`.

[EXP IF]

$$\frac{\Gamma \vdash e : a \uparrow b \qquad \Gamma \vdash e_i : a_i \uparrow b_i}{\Gamma \vdash \texttt{if}\ e\ e_1\ e_2 : (a; (a_1 \sqcup a_2)) \uparrow (b \sqcup (a; (b_1 \sqcup b_2)))}$$

The rule [EXP LOOP] states that the normal atomicity of the loop is $\bot$ since it never terminates normally. The abrupt atomicity for a loop reflects the fact that the loop body could terminate normally many times before terminating abruptly.

[EXP LOOP]

$$\frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash \texttt{loop}\ e : \bot \uparrow (a^*; b)}$$

A `block` $e$ expression never terminates abruptly. Either the body $e$ terminates normally, or it executes a `break` expression that terminates $e$ early. In the latter case, we still consider `block` $e$ to exit normally. A `break` expression only terminates abruptly and is a both mover.

[EXP BLOCK] [EXP BREAK]

$$\frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash \texttt{block}\ e : (a \sqcup b) \uparrow \bot} \qquad \frac{}{\Gamma \vdash \texttt{break} : \bot \uparrow B}$$

A key innovation of our effect system is our treatment of pure blocks. The rule [EXP PURE] for `pure` $e$ states that the normal atomicity of the body of a pure block must be at most $A$. This requirement ensures that any temporary side effects during the evaluation of $e$ are not visible to other threads. Since, under normal termination, the pure block has no observable effect, our effect system "optimizes" the normal atomicity of a pure block to a both-mover $B$.

```
atomic void init() {
    block {
        pure { if (x• != null) break; }
        acquire(l);
        if (xε == null) x• = new();
        release(l);
    }
}
```

Fig. 12. Double-checked locking.

[EXP PURE]

$$\frac{\Gamma \vdash e : a \uparrow b \qquad a \sqsubseteq A}{\Gamma \vdash \texttt{pure}\ e : B \uparrow b}$$

Finally, the normal and abrupt atomicities of the body of an atomic construct are required to be at most $A$.

[EXP ATOMIC]

$$\frac{\Gamma \vdash e : a \uparrow b \qquad a, b \sqsubseteq A}{\Gamma \vdash \texttt{atomic}\ e : a \uparrow b}$$

Our effect system is sound in the sense that any abstract execution trace of a well-typed program is equivalent to a serial abstract execution of that program. In this serial abstract execution, the steps of each `atomic` block are executed sequentially, without steps interleaved from other threads. To verify this serializability property, we first reduce each normally terminating `pure` block into a sequence of contiguous steps and we replace that sequence with a single `skip` step. We then reduce `atomic` blocks in the modified execution to obtain an equivalent serial abstract execution. This correctness property is formalized and proven in Appendix C.

## 4 Applications

In this section, we present several examples to illustrate the expressiveness of our effect system for atomicity. In these examples, we sometimes enclose expressions in parentheses or braces and use additional constructs such as `return` for clarity.

### 4.1 Double-Checked Initialization

We start by considering the double-checked initialization pattern, commonly used to ensure that a shared variable is initialized exactly once [18].[1] To avoid excessive synchronization overhead, the variable `x` in Fig. 12 is initially tested without holding its protecting lock `l`. If the first test fails, the lock is acquired and, if `x` is still `null`, then it is initialized. Note that the read $\texttt{x}_\epsilon$ is not a conflicting access since it commutes with concurrent reads, but the write $\texttt{x}_\bullet$ may conflict with reads of other threads. Since the procedure consists of an atomic operation (the first read of `x`) followed by a right-mover operation (`acquire(l)`), the procedure is not reducible and cannot be verified as atomic using previous analyses.

1. Our analysis assumes a sequentially consistent memory model. Double-checked initialization may not work correctly under other memory models.

```
atomic void cachePut(String k, Object v) {...}
atomic pure Object cacheGet(String k) {...}

// expensive operation
both-mover Object compute(String k) {...}

atomic Object lookup(String k) {
    pure {
        Object r = cacheGet(k_ε);
        if (r_ε != null) return r_ε;
    }
    Object r = compute(k_ε);
    cachePut(k_ε,r_ε);
    return r_ε;
}
```

Fig. 13. Caching.

Our approach exploits the fact that the first test of `x` is both pure and optional; omitting this test does not affect the correctness of the program, only its performance and, thus, we can enclose this test in a `pure` construct. If the first test succeeds, the procedure returns via a reducible trace. If the first test fails, then that test has no effect on the program store and we replace it by skip in the trace (just as for `alloc`), yielding a reducible trace through the function `init`. By this reasoning, our effect system verifies that each possible abstract execution of `init` has an equivalent serial abstract execution and, hence, `init` is abstractly atomic.

There are other ways to code the double-checked locking idiom for which our analysis is not immediately applicable. One area for future work is to explore techniques to generalize our approach to handle additional programming patterns, possibly through refactoring rules, as illustrated in Section 4.3.

### 4.2 Caching

In the next example, the function `compute` (Fig. 13) constructs the value for a given key, but is an expensive operation, so we wish to cache previously computed results. We assume that the cache operations `cachePut` and `cacheGet` are atomic (for example, because they acquire the lock protecting the cache), that `cacheGet` is a pure (side-effect free) function, and that `compute` is a both-mover. We would like to verify that `lookup` is atomic, to ensure that it still behaves correctly even when concurrently invoked by multiple threads.

The function `lookup` is irreducible since it contains sequentially composed atomic operations, `cacheGet` and `cachePut`. Note that the alternative implementation of holding the cache lock throughout `lookup` would introduce undesirable contention since `compute` is a long-running operation. However, the cache lookup is clearly an optimization and can be omitted without affecting program correctness. We exploit this fact by enclosing the cache lookup in a `pure` construct. If the cache lookup is successful, the function `lookup` immediately returns via a reducible trace. If the cache lookup fails, it has no effect on the program store. Our analysis leverages this information (documented by the `pure` keyword) to essentially "remove" the cache lookup from the trace by replacing it with skip to produce an equivalent, reducible trace. Thus, all abstract executions through the function `lookup` have an equivalent abstract serial execution and, so, the function `lookup` is abstractly atomic.

```
acquire(l);
while x {
    wait(l);
}
body;
release(l);
```

Fig. 14. Wait example.

### 4.3 Wait and Notify

The `wait` and `notify` routines facilitate notification between concurrent threads. The routine `wait(l)` should be called only if the lock `l` is held; this routine then releases `l`, blocks until a concurrent thread calls `notify(l)`, and then returns after reacquiring `l`. Typically, the routine `wait(l)` is called inside a loop that iterates until a desired condition holds, and concurrent threads call `notify(l)` whenever a state change may affect the desired condition. We model `wait(l)` and `notify(l)` as `{release(l);acquire(l)}` and `skip`, respectively. This model captures the essence that other threads may acquire `l` during the execution of `wait(l)`. In other words, `wait` is not atomic.

The code fragment in Fig. 14 illustrates the use of `wait` to iterate until the variable `x` (protected by lock `l`) is `false`, and we assume that *body* is atomic. For this example, even though `wait(l)` is not atomic, our type system can verify that the entire code fragment, although irreducible, is still abstractly atomic. Before applying our type system, we first need to refactor this code using the equivalence rules for program expressions illustrated in Fig. 15.

Applying these rules to the above code fragment in the appropriate manner yields the refactored code of Fig. 16 that has equivalent behavior, but where the body of the loop is now pure. (Note that not all uses of `wait` can be refactored in this manner.)

The purity of the refactored loop allows our effect system to verify that each loop iteration except the last has no side-effect and can be elided from the execution sequence. The resulting abstract execution sequence acquires the lock, checks that `x` is `false`, executes *body*, and releases the lock. This sequence is both atomic and reducible. Since every

$$
\begin{array}{rcll}
e;\texttt{block}\ e' & = & \texttt{block}\ e;e' & \text{if } e \text{ cannot } \texttt{break} \\
e;\texttt{loop}\ \{e';e\} & = & \texttt{loop}\ \{e;e'\} & \\
\texttt{break} & = & \texttt{break};e & \\
\texttt{if}\ e_1\ \{e_2;e\}\ \{e_3;e\} & = & \{\texttt{if}\ e_1\ e_2\ e_3\};e &
\end{array}
$$

Fig. 15. Equivalence rules.

```
block loop pure {
    acquire(l);
    if x release(l) break;
}
body;
release(l);
```

Fig. 16. Refactored wait example.

possible execution of the original code fragment is equivalent to such an atomic execution, the original code fragment is abstractly atomic.

### 4.4 Packet Counter

The example in Fig. 17 counts the number of packets received in a program with the `_packetCount` variable, which is used only for monitoring or performance purposes.

To avoid synchronization overhead, the program accesses `_packetCount` without synchronization with the expectation that the resulting race conditions will not cause the resulting count to be substantially incorrect. By marking `_packetCount` as unstable, we can still consider procedures like `receive` to be abstractly atomic despite the presence of race conditions. (We do need to check the sequential correctness of `receive` under the abstract semantics, where `_packetCount` may change nondeterministically.)

### 4.5 Transaction Retry

The function in Fig. 18 models optimistic concurrency control based on transaction retry. It updates a shared data variable `z` according to `z = f(z)`. However, the function `f` is a long-running operation, so the transaction code does not hold `z`'s protecting lock when computing `f`. Instead, it acquires the lock, records the value of `z`, releases the lock, applies `f` to the recorded value of `z`, then reacquires the lock and updates `z`, provided `z` has not changed. If `z` has changed, then the entire transaction is retried.

We express this computation using two pure blocks in order to verify its abstract atomicity in our type system. The first pure block reads the value of `z` while the lock is held and returns this value, which is then stored in a temporary variable `x`. Under the abstract semantics, this first pure block may return an arbitrary value, but this arbitrary value may only cause inefficient but not incorrect behavior. After the computation `fx = f(x)` finishes, a second pure block

```
int _packetCount;
Queue packets;

atomic void enqueue(Queue q, Packet p) { ... }

atomic void receive(Packet p) {
    _packetCount_ε++;
    enqueue(packets_ε, p_ε);
}
```

Fig. 17. Packet counter.

```
atomic void apply_f() {
    int x, fx;
    while (true) {
        x_ε = pure {
            acquire(m);
            let t = z_•;
            release(m);
            t_ε;
        }
        fx = f(x);
        pure {
            acquire(m);
            if (x_ε == z_ε) {
                z_ε = fx_ε;
                release(m);
                break;
            }
            release(m);
        }
    }
}
```

Fig. 18. Transaction retry.

either updates `z` to complete the transaction or terminates normally without any side-effect if `z` has changed, in which case the transaction is retried.

Straightforward sequential reasoning enables us to ensure that this code is correct under the abstract sequential semantics. Our type system then verifies that every arbitrary interleaved execution of this code is also therefore correct.

## 5 Modifying Local Variables in Pure Blocks

A number of examples can be handled by pure blocks that do not modify the program state, such as those described in Section 4. However, it is sometimes convenient to permit some assignments within pure blocks. In this section, we relax our restriction that pure blocks be entirely side-effect free. Specifically, we introduce the notion of *thread-local* variables, and we permit a pure block to read from and assign to thread-local variables. Many standard escape analyses can be used to identify thread-local variables [2], [15], [19], [20].

For example, the pure block in the left column of Fig. 19 reads a shared variable `z` and writes its value to a thread-local variable `x`. The reason this relaxed requirement is still correct is that this pure block can be rewritten so that,

```
pure {
    acquire(m);
    x = z;
    release(m);
}
```

```
x = pure {
    acquire(m);
    let t = z;
    release(m);
    t;
}
```

Fig. 19. Pure blocks.

instead of assigning to x, it returns the value that should be assigned to x, but that assignment is actually done outside the pure block, as shown in the right column in Fig. 19. The typing rule for pure blocks remains unchanged.

This technique generalizes to multiple thread-local variables by allowing the pure block to return a tuple of the values to be assigned to the various thread-local variables. This more expressive notion of pure blocks allows us to express programs such as the transaction retry example more naturally.

## 6 Related Work

Lipton [11] first proposed reduction as a way to reason about deadlocks in concurrent programs without considering all possible interleavings. Reduction has subsequently been extended to support proofs of general safety and liveness properties [21], [22], [23], [24], [25]. Bruening [26] and Stoller [27] have used reduction to improve the efficiency of model checking. Flanagan and Qadeer have pursued a similar approach [28], and Qadeer et al. [29] have used reduction to infer procedure summaries in concurrent programs.

We previously applied reduction to verify atomicity in a static type and effect system for Java programs [8], [7]. This paper improves on that approach by enabling us to reason about the atomicity of code that is not immediately reducible.

The Calvin-R checker for multithreaded code relates procedure implementations to their functional specifications with an abstraction relation based on both reduction and simulation [6]. While capable of checking the atomicity of the examples in this paper, the overhead of that approach, in terms of annotation size and analysis complexity, is much greater. In contrast, the approach presented in this paper is more scalable, intuitive, and easier to use for checking atomicity properties.

Wang and Stoller [9] have developed a dynamic algorithm that can verify the atomicity of some irreducible code sequences. Their approach constructs the feasible interleavings of steps from two blocks of code and then determines whether all such interleavings are serializable. Unlike our approach, that algorithm does not require abstraction or auxiliary analysis to recognize pure blocks, and it is, in some sense, a complementary approach to ours.

The Atomizer is another dynamic analysis tool for detecting atomicity violations [5]. Our experience with the Atomizer, which uses reduction, suggests that the techniques developed in this paper could eliminate a nontrivial number of spurious warnings in reduction-based atomicity checkers.

The use of model checking for verifying atomicity is being explored by Hatcliff et al. [10], and they present two approaches, based on Lipton's theory of reduction and partial-order reductions [30]. In comparison with our effect system, model checking requires many fewer programmer-inserted annotations and can accommodate complex synchronization disciplines more easily, but is less scalable. Their experimental results suggest that verifying atomicity via model-checking is feasible for unit-testing. Their approach currently verifies atomicity only of reducible procedures, but we believe that integrating our notions of abstraction and atomicity into their system could yield many of the benefits of both approaches.

In related work, Rodriguez et al. [31] demonstrate how to refactor code in order to extract some reducible code blocks embedded inside irreducible functions. This technique could, for example, refactor `alloc` to utilize an auxiliary (and reducible) method that contains a variant of the code inside the body of the `for` loop. In this way, one could check the atomicity of the auxiliary method, but not of the entire `alloc` function.

A number of tools have been developed for detecting race conditions, both statically and dynamically. The Race Condition Checker [2] uses a type system to catch race conditions in Java programs. This approach has been extended [14] and adapted to other languages [15]. Other static race detection tools include Warlock [16], for ANSI C programs, and ESC/Java [3], which catches a variety of software defects in addition to race conditions.

Atomicity is a semantic correctness condition for multithreaded software. It is related to strict serializability [32], a correctness condition for database transactions, and linearizability [33], a correctness condition for concurrent objects. It is possible that techniques for verifying atomicity can be leveraged to develop lightweight checking tools for related correctness conditions.

Other languages have included a notion of atomicity as a primitive operation. Hoare [34] and Lomet [35] first proposed the use of atomic blocks for synchronization, and the Argus [36] and Avalon [37] projects developed language support for implementing atomic objects. Persistent languages [38], [39] augment atomicity with data persistence in order to introduce transactions into programming languages. Other recent approaches to supporting atomicity include lightweight transactions [40], [41] and automatic generation of synchronization code from high-level specifications [42].

## 7 Conclusion

Atomicity is an important correctness property for multithreaded software. Current reduction-based tools can verify atomicity in common cases, but they cannot handle situations in which code that is intuitively atomic is not immediately reducible. A number of frequently used programming idioms fall into this category.

This paper describes a static analysis technique capable of verifying the atomicity of some problematic cases by applying reduction to an abstraction of the program. The abstraction notions we have presented—based on purity and instability—are intuitive. The correctness of an abstractly atomic procedure under the serial abstract semantics can be verified using sequential reasoning. Our static analysis then verifies that every interleaved execution of this abstractly atomic procedure is also correct.

Although we present our analysis as an effect system, these concepts may be applicable in other domains. For example, software model checkers (such as the one described in [10]) could identify and exploit pure code blocks while performing reduction, and dynamic analyses for atomicity [5] could perhaps benefit from these ideas as well.

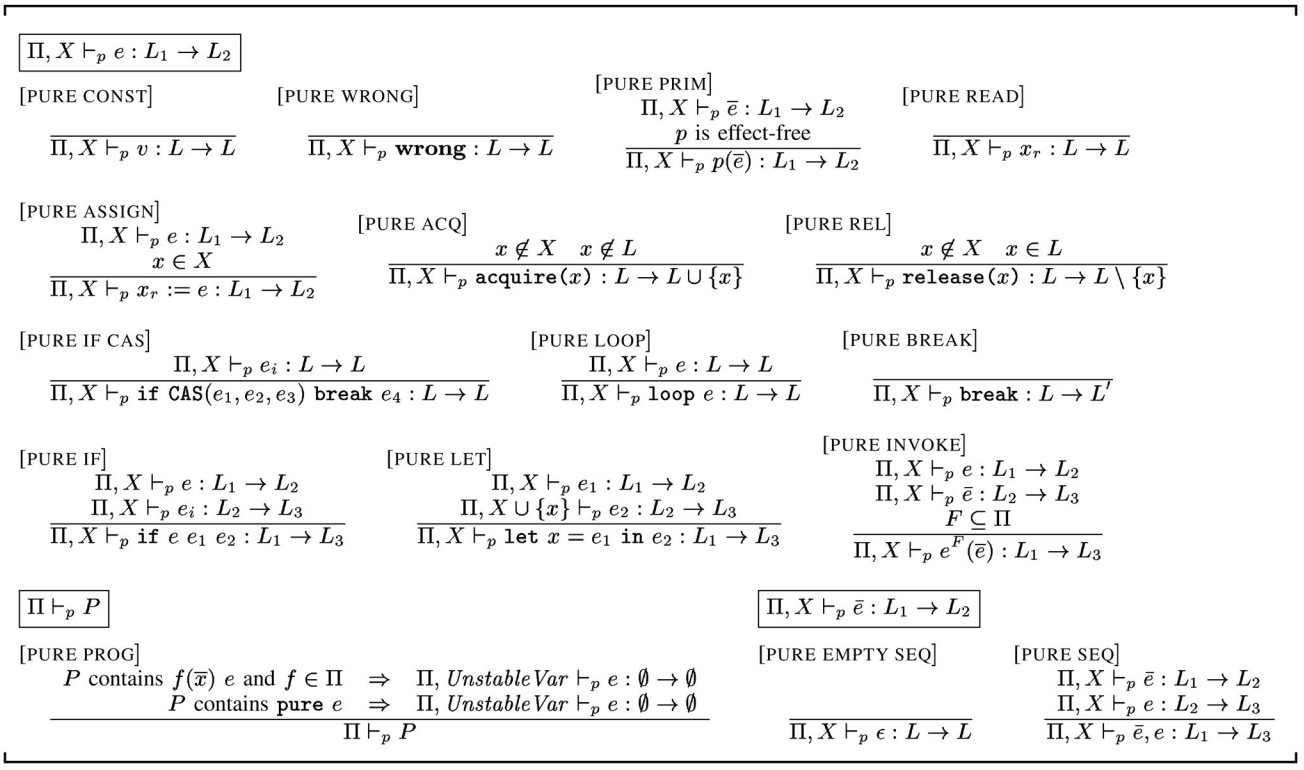



Fig. 20. Purity effect system.

## APPENDIX A
## EFFECT SYSTEM FOR PURITY

We present in this appendix an effect system to check that all normally terminating `pure` expressions in a program are side-effect-free. This effect system (Fig. 20) is relatively simple but sufficient to check all examples in this paper. The effect system essentially tracks all locks acquired by each pure expression to ensure that these locks are released before termination. More complex analyses could improve precision by, for example, tracking more precise control-flow and data-flow information.

The effect system reasons about the judgment

$$\Pi, X \vdash_p e : L \to L'$$

where $\Pi$ is the set of functions that are side-effect-free under normal-termination, and $X$ is the set of variables that may change during evaluation of $e$. The set $L$ is the set of locks held at the beginning of evaluation of $e$, and $L'$ is the set of locks held after $e$ terminates normally.

Most rules are straightforward. Any variable may be read, but only variables appearing in $X$ may be modified.

[PURE READ]

$$\frac{}{\Pi, X \vdash_p x_r : L \to L}$$

[PURE ASSIGN]

$$\frac{\Pi, X \vdash_p\ e : L_1 \to L_2 \qquad x \in X}{\Pi, X \vdash_p x_r\ :=\ e : L_1 \to L_2}$$

The rules typically construct the set of locks held after evaluation by "threading" the lockset through each sub-expression, as demonstrated by the rule for let expressions:

[PURE LET]

$$\frac{\Pi, X \vdash_p e_1 : L_1 \to L_2 \qquad \Pi, X \cup \{x\} \vdash_p e_2 : L_2 \to L_3}{\Pi, X \vdash_p \texttt{let } x = e_1 \texttt{ in } e_2 : L_1 \to L_3}$$

We introduce specific rules for the primitive operations that acquire and release locks as well as for the idiom of breaking when a `CAS` operation succeeds. Additional rules could model other synchronization primitives, as necessary.

[PURE ACQ]

$$\frac{x \notin X \quad x \notin L}{\Pi, X \vdash_p \texttt{acquire}(x) : L \to L \cup \{x\}}$$

[PURE REL]

$$\frac{x \notin X \quad x \in L}{\Pi, X \vdash_p \texttt{release}(x) : L \to L \setminus \{x\}}$$

[PURE IF CAS]

$$\frac{\Pi, X \vdash_p e_i : L \to L}{\Pi, X \vdash_p \texttt{if CAS}(e_1, e_2, e_3) \texttt{ break } e_4 : L \to L}$$

The rule [PURE PROG] for the top-level judgment

$$\Pi\ \vdash_p\ P$$

states that the annotation `pure` $e$ is valid if

$$\Pi, \mathit{UnstableVar} \vdash_p\ e : \emptyset \to \emptyset$$

$$
\begin{array}{rcrcl}
e & \in & \mathit{Expr} & ::= & \ldots \mid \texttt{in-atomic}\ e \\
 & & & & \mid \texttt{in-pure}\ e \mid \texttt{in-pureX}\ e \\
v & \in & \mathit{Value} & ::= & \ldots \mid m \\
m & \in & \mathit{SyncLoc} & &
\end{array}
$$

Fig. 21. Runtime syntax.

That is, a pure block may not change any stable variables or terminate with a different set of locks held than when evaluation started. This rule also requires that every function in $\Pi$ is pure.

## APPENDIX B
## FORMAL SEMANTICS OF CAP

To reason about the correctness of our analysis, we formalize the runtime behavior of CAP programs using an operational semantics (Fig. 22). For this purpose, we extend CAP with additional runtime expressions (Fig. 21), which record that execution is proceeding inside an atomic block or inside a pure block. In addition, since nonterminating pure blocks should not change the program state, we distinguish evaluation of a pure block that will terminate normally (`in-pure`) from one that will terminate abruptly (`in-pureX`). We also extend the set of values to include

**State space**

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

**Evaluation contexts**

$$
\begin{array}{rcl}
E & ::= & [\,] \mid x_r := E \mid p(\overline{v}, E, \overline{e}) \mid E^F(\overline{e}) \mid v^F(\overline{v}, E, \overline{e}) \\
 & & \mid \texttt{if}\ E\ e\ e \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{block}\ E \\
 & & \mid \texttt{in-atomic}\ E \mid \texttt{in-pure}\ E \mid \texttt{in-pureX}\ E
\end{array}
$$

**Transition relations** $\quad \rightarrow_i, \rightarrow, \mapsto, \leadsto \quad \subseteq \quad \mathit{State} \times \mathit{State}$

**Transition rules** (where $i = |T| + 1$)

| | | | | |
|---|---|---|---|---|
| [RED READ] | $H, M, T.E[x_r].T'$ | $\rightarrow_i$ | $H, M, T.E[H(x)].T'$ | |
| [RED WRITE] | $H, M, T.E[x_r := v].T'$ | $\rightarrow_i$ | $H[x := v], M, T.E[v].T'$ | if $x \in \mathit{Dom}(H)$ |
| [RED PRIM] | $H, M, T.E[p(\overline{v})].T'$ | $\rightarrow_i$ | $H, M', T.E[v'].T'$ | if $\mathcal{I}_p(\overline{v}, M, i) = \langle v', M' \rangle$ |
| [RED WRONG] | $H, M, T.E[p(\overline{v})].T'$ | $\rightarrow_i$ | $H, M, T.\mathbf{wrong}.T'$ | if $\mathcal{I}_p(\overline{v}, M, i) = \mathbf{wrong}$ |
| [RED CALL] | $H, M, T.E[(f(\overline{x})\ e)^F(\overline{v})].T'$ | $\rightarrow_i$ | $H[\overline{x} := \overline{v}], T.E[e].T'$ | if $\overline{x} \cap \mathit{dom}(H) = \emptyset$ |
| [RED IF1] | $H, M, T.E[\texttt{if}\ v\ e_1\ e_2].T'$ | $\rightarrow_i$ | $H, M, T.E[e_1].T'$ | if $v \neq 0$ |
| [RED IF2] | $H, M, T.E[\texttt{if}\ 0\ e_1\ e_2].T'$ | $\rightarrow_i$ | $H, M, T.E[e_2].T'$ | |
| [RED LOOP] | $H, M, T.E[\texttt{loop}\ e].T'$ | $\rightarrow_i$ | $H, M, T.E[e; \texttt{loop}\ e].T'$ | |
| [RED LET] | $H, M, T.E[\texttt{let}\ x = v\ \texttt{in}\ e].T'$ | $\rightarrow_i$ | $H[x := v], M, T.E[e].T'$ | if $x \notin \mathit{dom}(H)$ |
| [RED FORK] | $H, M, T.E[\texttt{fork}\ e].T'$ | $\rightarrow_i$ | $H, M, T.E[0].T'.e$ | |
| [RED ATOMIC] | $H, M, T.E[\texttt{atomic}\ e].T'$ | $\rightarrow_i$ | $H, M, T.E[\texttt{in-atomic}\ e].T'$ | |
| [RED IN-ATOMIC] | $H, M, T.E[\texttt{in-atomic}\ v].T'$ | $\rightarrow_i$ | $H, M, T.E[v].T'$ | |
| [RED PURE] | $H, M, T.E[\texttt{pure}\ e].T'$ | $\rightarrow_i$ | $H, M, T.E[\texttt{in-pure}\ e].T'$ | |
| [RED PURE EX] | $H, M, T.E[\texttt{pure}\ e].T'$ | $\rightarrow_i$ | $H, M, T.E[\texttt{in-pureX}\ e].T'$ | |
| [RED IN-PURE] | $H, M, T.E[\texttt{in-pure}\ c].T'$ | $\rightarrow_i$ | $H, M, T.E[c].T$ | |
| [RED BREAK] | $H, M, T.E[\texttt{block}\ E'[\texttt{break}]].T'$ | $\rightarrow_i$ | $H, M, T.E[0].T'$ if $E'$ does not contain `block` [ ] or `in-pure` [ ] | |
| [RED BLOCK] | $H, M, T.E[\texttt{block}\ v].T'$ | $\rightarrow_i$ | $H, M, T.E[v].T'$ | |
| [ABS PURE SKIP] | $H, M, T.E[\texttt{pure}\ e].T'$ | $\rightarrow_i$ | $H, M, T.E[c].T'$ if $\exists H', M', H'', M''$ such that $H', M', T.E[\texttt{pure}\ e].T' \rightarrow_i^* H'', M'', T.E[c].T'$ | |
| [ABS UNSTABLE READ] | $H, M, T.E[x_r].T'$ | $\rightarrow_i$ | $H, M, T.E[v].T'$ if $x \in \mathit{Dom}(H) \cap \mathit{UnstableVar}$ | |
| [ABS UNSTABLE WRITE] | $H, M, T.E[x_r := v].T'$ | $\rightarrow_i$ | $H[x := v'], M, T.E[v].T'$ if $x \in \mathit{Dom}(H) \cap \mathit{UnstableVar}$ | |
| [ABS GARBAGE] | $H, M, T$ | $\rightarrow_i$ | $H', M', T$ if $H$ and $H'$ agree on $\mathit{Dom}(H) \cap \mathit{StableVar}$ and $M$ and $M'$ agree on $\mathit{Dom}$(M) | |
| (standard semantics) | $H, M, T$ | $\leadsto$ | $P'$ if $H, M, T \rightarrow_i P'$ by a rule other than [ABS PURE SKIP], [ABS GARBAGE], [ABS UNSTABLE WRITE], or [ABS UNSTABLE READ] | |
| (abstract semantics) | $H, M, T$ | $\rightarrow$ | $P'$ if $H, M, T \rightarrow_i P'$ | |
| (serial abstract semantics) | $H, M, T$ | $\mapsto$ | $P'$ if $H, M, T \rightarrow_i P'$ and $T_j$ does not contain `in-atomic` for $j \neq i$ | |

Fig. 22. Semantics.

synchronization locations $m$, which are manipulated by the synchronization primitives.

A program state $P$ is a three-tuple $H, M, T$. A heap $H$ is a partial map from variables to values. A synchronization heap $M$ maps synchronization locations to synchronization values, and $T$ is a sequence of threads. A thread is either an expression or **wrong**.

Evaluation contexts define the order of evaluation for CAP. An evaluation context $E$ is an expression with a "hole" at the location of the next subexpression to be evaluated, and $E[e]$ denotes the operation of filling the hole in $E$ with the expression $e$. The notation $H[x := v]$ denotes a new heap that is identical to $H$ except that it maps $x$ to $v$.

The transition relation $\rightarrow_i$ performs a single step of thread $i$. In the standard semantics, evaluation of `pure` $e$ has two possible outcomes. For the case where $e$ terminates normally, yielding some value $v$, we reduce `pure` $e$ to `in-pure` $e$ via [RED PURE], which in turn evaluates to `in-pure` $c$, which reduces to $c$ via [RED IN-PURE]. (Recall that pure expressions may return only constants.) Alternatively, if $e$ terminates abruptly (via `break`), we reduce `pure` $e$ to `in-pureX` $e$ via [RED PURE EX], which evaluates to `in-pureX` $E$[`break`] and the `break` propagates outside the `in-pureX` expression via [RED BREAK]. Thus, these rules characterize all possible executions of `pure` $e$, but also capture (via `in-pure`/`in-pureX`) the future termination mode of $e$, which is necessary for the correctness proof.

Note that this semantics requires a choice to be made about the behavior of $e$ when the `pure` $e$ block is entered. The wrong choice will simply cause evaluation to block when incorrect behavior is observed. For example, the [RED BREAK] rule is not applicable if a `break` causes an abrupt exit from an `in-pure` block. Similarly, `in-pureX` $c$ cannot be further reduced.

The transition relation $\rightsquigarrow$ performs a single step of an arbitrarily chosen thread in the standard semantics. However, we verify atomicity with respect to an *abstract semantics* $\rightarrow$ that admits additional execution sequences. First, in the abstract semantics, the evaluation of `pure` $e$ may be skipped entirely via [ABS PURE SKIP]. The value returned as a result of skipping `pure` $e$ must be consistent with the value returned by evaluating `pure` $e$ in some arbitrary global state, modeling that a read of a global variable by a normally terminating pure block returns an arbitrary value.

The abstract semantics also includes the rules [ABS UNSTABEL READ] and [ABS UNSTABLE WRITE] for reading/writing arbitrary values from/to unstable variables. Since `pure` expressions may create garbage by allocating new temporaries, we also introduce the "garbage creation" rule [ABS GARBAGE]. This rule enables us to show that a trace including the normal evaluation of a `pure` expression is equivalent to a trace with its evaluation replaced by a [ABS PURESKIP] step followed by a [ABS GARBAGE] step. While not strictly necessary, we permit that rule to change unstable variables to simplify some technical aspects of our correctness proof.

We use $\rightarrow^+$ and $\rightarrow^*$ to denote the transitive and reflexive-transitive closure of $\rightarrow$, respectively. The *serial abstract semantics* $\mapsto$ is similar to $\rightarrow$, with the additional restriction that a thread cannot perform a step if another thread is inside an `in-atomic` block. Thus, $\mapsto$ does not interleave the execution of an `atomic` block with instructions from other threads.

For simplicity, we do not model thread-local variables in this semantics, but that could be accomplished by following the approach of [15].

### B.1 Primitive Operations

The meaning of a primitive operation $p$ is given by the partial function $\mathcal{I}_p$, which takes a sequence of argument values, a synchronization heap, and an integer identifying the current thread, and returns a value and a (possibly modified) synchronization heap. If the primitive is applied to incorrect arguments, $\mathcal{I}_p$ returns **wrong**.

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

The `assert` primitive operation goes wrong if its argument is 0 and otherwise terminates normally without modifying the heap. Addition is a pure primitive operation and does not modify the synchronization heap. The `new_lock` operation returns a synchronization location $m$ that refers to a newly allocated lock containing 0, indicating that is not held by any thread. The `acquire` operation acquires a lock, provided the lock is not held by another thread. If the lock is held by some thread, then the `acquire` operation blocks, and execution can proceed only on the other threads. The `release` operation never blocks; if the current thread holds the lock, then the lock is released and, otherwise, the `release` operation goes wrong. We define the semantics of these primitive operations as follows (where $\epsilon$ is the empty sequence and $v.s$ prepends $v$ at the front of sequence $s$):

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

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

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

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

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

$$\mathcal{I}_{\text{CAS}}(m.c_1.c_2, M, tid) = \begin{cases} \langle 0, M\rangle & \text{if } M(m) \neq c_1 \\ \langle 1, M[m := c_2]\rangle & \text{if } M(m) = c_1. \end{cases}$$

We now formalize our assumption on primitive atomicities. A primitive $p$ *right-commutes* with a primitive $q$ if, for all $i$ and $j$ such that $i \neq j$, the following conditions hold:

1. If $\mathcal{I}_p(\bar{v}, M, i) = \langle v', M'\rangle$ and $\mathcal{I}_q(\bar{u}, M', j) = \langle u', M''\rangle$, then there is $M'''$ such that $\mathcal{I}_q(\bar{u}, M, j) = \langle u', M'''\rangle$ and $\mathcal{I}_p(\bar{v}, M''', i) = \langle v', M''\rangle$.
2. If $\mathcal{I}_p(\bar{v}, M, i) = \mathbf{wrong}$ and $\mathcal{I}_q(\bar{u}, M, j) = \langle u', M'\rangle$, then $\mathcal{I}_p(\bar{v}, M', i) = \mathbf{wrong}$.

3. If $\mathcal{I}_p(\bar{v}, M, i) = \langle v', M' \rangle$ and $\mathcal{I}_q(\bar{u}, M', j) = \textbf{wrong}$, then $\mathcal{I}_q(\bar{u}, M, j) = \textbf{wrong}$.

The primitive $p$ *left-commutes* with the primitive $q$ if $q$ right-commutes with $p$. An effect environment is *valid* if the atomicity $\Gamma(p)$ of each primitive $p$ is correct in that the following properties hold:

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

Note that, since primitive operations read or write only the synchronization heap and not the regular heap, they trivially commute with all nonprimitive operations which access only the regular heap.

### B.2 Correctness of Annotations

This section defines the meaning of the conflict annotations, call annotations, and purity in terms of the CAP semantics. An expression $e$ is *about to read* $x$ if $e \equiv E[x_r]$. Similarly, $e$ is *about to write* $x$ if $e \equiv E[x_r := v]$. An expression $e$ is *about to access* $x$ if $e$ is about to read or write $x$. The following conditions characterize correct annotations. As mentioned earlier, auxiliary type and effect analyses may check conformance to these requirements.

1. The conflict annotations in a program $P$ are *correct* if, whenever $P \rightarrow^* H, M, T$:
   a. If $T_i \equiv E[x_\epsilon]$ and $x \in StableVar$, then no other thread in $T$ is about to write $x$.
   b. If $T_i \equiv E[x_\epsilon := v]$ and $x \in StableVar$, then no other thread in $T$ is about to read or write $x$.
2. The call annotations in $P$ are *correct* if, whenever $P \rightarrow^* H, M, T$ and $T_i \equiv E[(f(\bar{x})\ e)^F(\bar{v})]$, then $f \in F$.
3. The purity annotations in $P$ are *correct* if, whenever $i = |T|$ and

$$\begin{array}{rcl} P & \rightarrow^* & H_1, M_1, T.E[\texttt{pure}\ e].T' \\ & \rightarrow_i^* & H_2, M_2, T.E[c].T'.T'' \end{array}$$

and, in each intermediate state, the $i$th thread has outermost evaluation context $E$, then $T'' = \epsilon$ and

$$H_1, M_1, T.E[c].T' \rightarrow_i^* H_2, M_2, T.E[c].T'.$$

## APPENDIX C
## CORRECTNESS PROOF FOR ATOMICITY

Our effect system (extended to runtime states as in Fig. 23) guarantees a correspondence between nonserial abstract executions and serial abstract executions of any well-formed program. In particular, if, under the nonserial abstract semantics ($\rightarrow$), a well-formed program $P$ can reach a state $Q$ where no thread is executing an atomic block, then $P$ can also reach $Q$ under the serial abstract semantics ($\mapsto$). Hence, the serial abstract semantics in which we do not have to consider interleaved actions of concurrent threads inside an atomic block suffices for reasoning about the reachability of such states $Q$.

$\boxed{\Gamma \vdash e : a \uparrow b}$

[EXP REF] $\dfrac{}{\Gamma \vdash m : B \uparrow \bot}$

[EXP INATOMIC] $\dfrac{\Gamma \vdash e : a \uparrow b \quad a, b \sqsubseteq A}{\Gamma \vdash \texttt{in-atomic}\ e : a \uparrow b}$

[EXP INPURE] $\dfrac{\Gamma \vdash e : a \uparrow b \quad a \sqsubseteq A}{\Gamma \vdash \texttt{in-pure}\ e : B \uparrow \bot}$

[EXP INPURE EX] $\dfrac{\Gamma \vdash e : a \uparrow b \quad a \sqsubseteq A}{\Gamma \vdash \texttt{in-pureX}\ e : \bot \uparrow b}$

[EXP WRONG] $\dfrac{}{\Gamma \vdash \textbf{wrong} : B \uparrow \bot}$

$\boxed{\Gamma \vdash P}$

[PROG]
$$\frac{\begin{array}{c} \forall x \in dom(H).\ \Gamma \vdash H(x) : B \uparrow \bot \\ \Gamma \vdash T_i : a_i \uparrow b_i \qquad \forall i \in 1..n \\ \forall e, i.(T_i \text{ contains } \texttt{in-atomic}\ e \Rightarrow \exists E.\ T_i \equiv E[\texttt{in-atomic}\ e]) \\ \forall e, i.(T_i \text{ contains } \texttt{in-pure}\ e \Rightarrow \exists E.\ T_i \equiv E[\texttt{in-pure}\ e]) \\ \forall e, i.(T_i \text{ contains } \texttt{in-pureX}\ e \Rightarrow \exists E.\ T_i \equiv E[\texttt{in-pureX}\ e]) \end{array}}{\Gamma \vdash H, M, T_1 \ldots T_n}$$

Fig. 23. Effect system for runtime states.

In the absence of `pure` blocks, we can prove this correspondence property via reduction [11] since each atomic block consists of a sequence of right-movers, followed by at most one atomic action, followed by a sequence of left-movers. However, `pure` blocks significantly complicate the correctness proof. This appendix contains an outline of the correctness proof; the complete details are available in a companion technical report [43].

We begin by showing via subject reduction that every state reachable from an initial, well-formed state is well-formed. For this purpose, we extend the effect system to handle runtime expressions and states, as follows. The rule [PROG] ensures that runtime expressions appear only in evaluation contexts.

**Theorem 1 (Subject Reduction).** *If the call annotations in $P$ are correct and $\Gamma \vdash P$ and $P \rightarrow P'$, then $\Gamma \vdash P'$.*

Let $\Gamma$ be a fixed effect environment and let $WT$ be the set of well-formed states under $\Gamma$.

$$WT = \{\langle H, M, T \rangle \mid \Gamma \vdash \langle H, M, T \rangle\}.$$

If $\Gamma \vdash e : a \uparrow b$, then we define the normal atomicity $\alpha_n(e)$, abrupt atomicity $\alpha_x(e)$, and (combined) atomicity $\alpha(e)$ of the expression $e$ as

$$\begin{array}{rcl} \alpha_n(e) & = & a \\ \alpha_x(e) & = & b \\ \alpha(e) & = & a \sqcup b. \end{array}$$

An examination of the effect rules shows that these are well-defined partial functions.

Each pure block that terminates normally evaluates via a sequence of `in-pure` states. Our correctness proof proceeds by induction on the nesting depth $n$ of `in-pure` blocks. First, we reduce each `in-pure` block at depth $n$ to a serial execution in which all of the actions of the `in-pure` block occur contiguously. Then, by the definition of purity,

we can replace that `in-pure` fragment of the execution sequence with a single [ABS PURE SKIP] step, followed by an [ABS GARBAGE] step to create any garbage generated by the removed `in-pure` fragment. This technique reduces the nesting depth of `in-pure` blocks by one. Proceeding in this manner eventually yields an execution sequence with no `in-pure` blocks at which point we can apply reduction to serialize each atomic block and produce an execution of the serial semantics ($\mapsto$).

To assist in reasoning about the nesting depth of `in-pure` blocks, we define $D^n$ to denote evaluation contexts with at least $n$ nested `in-pure` blocks so that $E = D^0 \supset D^1 \supset \cdots$.

$$\begin{array}{rcl} D^0 & = & E \\ D^{n+1} & = & \{E[\texttt{in-pure}\ E'] \mid E \in D^n\}. \end{array}$$

For all thread identifiers $i$ and integers $n$, we partition the set of well-formed states into four categories:

- $R_i^n$—states where thread $i$ is in the right-mover part of an `in-pure` block at depth $n$.
- $L_i^n$—states where thread $i$ is in the left-mover part of an `in-pure` block at depth $n$.
- $U_i^n$—states where thread $i$ is in an `in-pure` block at depth $n$ that has normal atomicity $\bot$ and, hence, diverges.
- $N_i^n$—states where thread $i$ does not contain `in-pure` block at depth $n$.

In addition, $N^n$ contains states where no thread is in an `in-pure` block at depth $n$ (and, so, $N^1$ describes states containing no `in-pure` blocks).

$$\begin{array}{rcl} R_i^n & = & WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-pure}\ e] \\ & & \qquad \wedge E \in D^{n-1} \wedge \alpha_n(e) \in \{R, A, \top\}\} \\ L_i^n & = & WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-pure}\ e] \\ & & \qquad \wedge E \in D^{n-1} \wedge \alpha_n(e) \in \{B, L\}\} \\ U_i^n & = & WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-pure}\ e] \\ & & \qquad \wedge E \in D^{n-1} \wedge \alpha_n(e) = \bot\} \\ N_i^n & = & WT \cap \{\langle H, M, T\rangle \mid \forall E \in D^{n-1}. T_i \not\equiv E[\texttt{in-pure}\ e]\} \\ N^n & = & \forall i. N_i^n. \end{array}$$

We next define two restrictions of the transition relation $\rightarrow_i$. The definition of these relations requires some additional notation. For any state predicate $X \subseteq State$ and transition relation $Y \subseteq State \times State$, by $Y/X$ we mean the transition relation obtained by restricting $Y$ to pairs whose first component is in $X$. Similarly, by $Y \setminus X$ we mean the restriction of $Y$ to pairs whose second component is in $X$. The first restricted relation $\overset{n}{\Longrightarrow}_i$ (for $n \geq 0$) is a restriction of $\rightarrow_i$ to states where thread $i$ has at most $n$ nested `in-pure` blocks (at nesting depth 1 through $n$ inclusive). The second relation $\overset{n}{\Longmapsto}_i$ (for $n \geq 1$) is a serialized version of $\overset{n}{\Longrightarrow}_i$ where each `in-pure` block at depth $n$ is executed serially with respect to other blocks at depth $n$ or greater.

$$\begin{array}{rcl} \overset{n}{\Longrightarrow}_i & = & N^{n+1}/ \rightarrow_i \\ \overset{n}{\Longmapsto}_i & = & (N^{n+1} \wedge \forall j \neq i. N_j^n)/ \rightarrow_i \\ \\ \overset{n}{\Longrightarrow} & = & \exists i. \overset{n}{\Longrightarrow}_i \\ \overset{n}{\Longmapsto} & = & \exists i. \overset{n}{\Longmapsto}_i \end{array}$$

The following properties are obvious consequences of these definitions.

$$\begin{array}{ccccc} \overset{n}{\Longmapsto}_i & \subset & \overset{n}{\Longrightarrow}_i & \subset & \rightarrow_i \\ \overset{n}{\Longmapsto} & \subset & \overset{n}{\Longrightarrow} & \subset & \rightarrow \end{array}$$

Using these relations, we show that any nonserial execution via $\overset{n}{\Longrightarrow}$ can be reduced to an equivalent serial execution via $\overset{n}{\Longmapsto}$.

**Lemma 1 (Pure Reduction).** *Let $P$ be a well-formed program with correct conflict and call annotations. Suppose $N^n(P)$ and $N^n(Q)$ and $P \overset{n}{\Longrightarrow}^* Q$, where $n \geq 1$. Then, $P \overset{n}{\Longmapsto}^* Q$.*

Given any execution via $\overset{n}{\Longmapsto}$ (in which `in-pure` blocks at depth $n$ are executed serially), we can elide each of these serially executed `in-pure` blocks at depth $n$, thus reducing the nesting depth of `in-pure` blocks and yielding an execution sequence under $\overset{n-1}{\Longrightarrow}$.

**Lemma 2 (Pure Eliding).** *If the pure annotations in $P$ are correct and $P_0 \overset{n}{\Longmapsto}^* P_k$ and $N^n(P_0)$ and $N^n(P_k)$, where $n \geq 1$, then $P_0 \overset{n-1}{\Longrightarrow}^* P_k$.*

Thus, given any execution sequence $P \overset{n}{\Longrightarrow}^* Q$, by repeatedly applying the Pure Reduction and Pure Eliding Lemmas, we eventually obtain an execution sequence $P \overset{0}{\Longrightarrow}^* Q$, where each intermediate state satisfies $N^0$ and thus does not contain `in-pure` blocks. We now leverage reduction one more time to reduce each atomic block in the execution sequence $P \overset{0}{\Longrightarrow}^* Q$. To assist in performing this reduction, we again partition the set of well-formed states into four categories:

- $R_i$—states where thread $i$ is executing the right-mover part of an `in-atomic` block.
- $L_i$ —states where thread $i$ is executing the left-mover part of an `in-atomic` block.
- $U_i$—states where thread $i$ is executing an `in-atomic` block that will never terminate.
- $N_i$—states where thread $i$ does not execute an `in-atomic` blocks.

We define $N$ to contain states where no thread is executing an `in-atomic` block.

$$\begin{array}{rcl} R_i & = & WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-atomic}\ e] \wedge \alpha(e) \not\sqsubseteq L\} \\ L_i & = & WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-atomic}\ e] \\ & & \qquad \wedge B \sqsubseteq \alpha(e) \sqsubseteq L\} \\ U_i & = & WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-atomic}\ e] \wedge \alpha(e) = \bot\} \\ N_i & = & WT \cap \{\langle H, M, T\rangle \mid |T| < i \vee T_i \not\equiv E[\texttt{in-atomic}\ e]\} \\ N & = & \forall i. N_i. \end{array}$$

The following theorem then shows that we can reduce an execution $P \overset{0}{\Longrightarrow}^* Q$ to a serial execution $P \mapsto^* Q$.

**Lemma 3 (Atomic Reduction).** *Let $P$ be a program with correct conflict and call annotations and let $\Gamma$ be a valid effect environment such that $\Gamma \vdash P$. Suppose $N(P)$ and $N(Q)$ and $P \overset{0}{\Longrightarrow}^* Q$. Then, $P \mapsto^* Q$.*

The following Correctness Theorem states that, given any nonserial execution $P \rightarrow^* Q$ with arbitrarily nested `in-pure` and in-atomic blocks, we can always obtain an equivalent serial execution $P \mapsto^* Q$. To avoid the consideration of partially executed atomic blocks, we require that no thread is executing an atomic block in either the initial state $P$ or the final state $Q$. We conjecture that our results can be extended to cover some partially executed blocks by the following existing techniques [8].

**Theorem 2 (Correctness).** *Let $P$ be a program with correct annotations and let $\Gamma$ be a valid effect environment such that $\Gamma \vdash P$. Suppose $N(P)$ and $N(Q)$ and $P \rightarrow^* Q$. Then, $P \mapsto^* Q$.*

**Proof.** This theorem follows from the following induction hypothesis, since there exists $n$ such that $P \overset{n}{\Longrightarrow}^* Q$.

**Induction Hypothesis**: Let $P$ be a program with correct annotations and let $\Gamma$ be a valid effect environment such that $\Gamma \vdash P$. Suppose $N(P)$ and $N(Q)$ and $P \overset{n}{\Longrightarrow}^* Q$. Then, $P \mapsto^* Q$.

The proof of this hypothesis is by induction on $n$. If $n = 0$, the conclusion follows by Lemma 3. Otherwise, if $P \overset{n}{\Longrightarrow}^* Q$, then

$$
\begin{aligned}
&P \overset{n}{\models\Longrightarrow}^* Q && \text{by Lemma 1}\\
&P \overset{n-1}{\Longrightarrow}^* Q && \text{by Lemma 2}\\
&P \rightarrow^* Q && \text{by the inductive hypothesis.}
\end{aligned}
$$

□

## ACKNOWLEDGMENTS

The authors would like to thank Leslie Lamport and Chandu Thekkath for useful discussions. Cormac Flanagan was partly supported by the US National Science Foundation (NSF) under Grant CCR-0341179 and by faculty research funds granted by the University of California at Santa Cruz. Stephen Freund was partly supported by the NSF under Grants CCR-0306486 and CCR-0341387 and by faculty research funds granted by Williams College.

## REFERENCES

[1] C. Flanagan and M. Abadi, "Types for Safe Locking" *Proc. European Symp. Programming,* pp. 91-108, 1999.

[2] C. Flanagan and S.N. Freund, "Type-Based Race Detection for Java" *Proc. ACM Conf. Programming Language Design and Implementation,* pp. 219-232, 2000.

[3] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata, "Extended Static Checking for Java," *Proc. ACM Conf. Programming Language Design and Implementation,* pp. 234-245, 2002.

[4] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson, "Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs," *ACM Trans. Computer Systems,* vol. 15, no. 4, pp. 391-411, 1997.

[5] C. Flanagan and S.N. Freund, "Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs," *Proc. ACM Symp. Principles of Programming Languages,* pp. 256-267, 2004.

[6] S.N. Freund and S. Qadeer, "Checking Concise Specifications for Multithreaded Software," *J. Object Technology,* vol. 3, no. 6, pp. 81-101, 2004.

[7] C. Flanagan and S. Qadeer, "A Type and Effect System for Atomicity," *Proc. ACM Conf. Programming Language Design and Implementation,* pp. 338-349, 2003.

[8] C. Flanagan and S. Qadeer, "Types for Atomicity," *Proc. ACM Workshop Types in Language Design and Implementation,* pp. 1-12, 2003.

[9] L. Wang and S.D. Stoller, "Run-Time Analysis for Atomicity," *Proc. Workshop Runtime Verification,* 2003.

[10] J. Hatcliff, Robby, and M.B. Dwyer, "Verifying Atomicity Specifications for Concurrent Object-Oriented Software Using Model-Checking," *Proc. Int'l Conf. Verification, Model Checking, and Abstract Interpretation,* pp. 175-190, 2004.

[11] R.J. Lipton, "Reduction: A Method of Proving Properties of Parallel Programs," *Comm. ACM,* vol. 18, no. 12, pp. 717-721, 1975.

[12] D. Peled, "Combining Partial Order Reductions with On-the-Fly Model Checking," *Proc. Conf. Computer Aided Verification,* pp. 377-390, 1994.

[13] C. Flanagan, "Verifying Commit-Atomicity Using Model-Checking," *Proc. 11th Int'l SPIN Workshop Model Checking of Software,* pp. 252-266, 2004.

[14] C. Boyapati and M. Rinard, "A Parameterized Type System for Race-Free Java Programs," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 56-69, 2001.

[15] D. Grossman, "Type-Safe Multithreading in Cyclone," *Proc. ACM Workshop Types in Language Design and Implementation,* pp. 13-25, 2003.

[16] N. Sterling, "WARLOCK—A Static Data Race Analysis Tool," *Proc. USENIX Technical Conf.,* pp. 97-106, 1993.

[17] F. Nielson, H.R. Nielson, and C. Hank, *Principles of Program Analysis.* Springer, 1999.

[18] D.C. Schmidt and T.H. Harrison, "Double-Checked Locking—A Optimization Pattern for Efficiently Initializing and Accessing Thread-Safe Objects," *Pattern Languages of Program Design 3,* R. Martin et al., eds., 1997.

[19] J.-D. Choi, M. Gupta, M.J. Serrano, V.C. Sreedhar, and S.P. Midkiff, "Escape Analysis for Java," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 1-19, 1999.

[20] B. Blanchet, "Escape Analysis for Object-Oriented Languages. Application to Java," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 20-34, 1999.

[21] T.W. Doeppner Jr., "Parallel Program Correctness through Refinement," *Proc. ACM Symp. Principles of Programming Languages,* pp. 155-169, 1977.

[22] R.-J. Back, "A Method for Refining Atomicity in Parallel Algorithms," *Proc. Conf. Parallel Architectures and Languages Europe,* pp. 199-216, 1989.

[23] L. Lamport and F.B. Schneider, "Pretending Atomicity," Research Report 44, DEC Systems Research Center, 1989.

[24] E. Cohen and L. Lamport, "Reduction in TLA," *Proc. Int'l Conf. Concurrency Theory,* pp. 317-331, 1998.

[25] J. Misra, *A Discipline of Multiprogramming: Programming Theory for Distributed Applications.* Springer- Verlag, 2001.

[26] D. Bruening, "Systematic Testing of Multithreaded Java Programs," master's thesis, Massachusetts Inst. of Technology, 1999.

[27] S.D. Stoller, "Model-Checking Multi-Threaded Distributed Java Programs," *Proc. Workshop Model Checking and Software Verification,* pp. 224-244, 2000.

[28] C. Flanagan and S. Qadeer, "Transactions for Software Model Checking," *Proc. Workshop Software Model Checking,* 2003.

[29] S. Qadeer, S.K. Rajamani, and J. Rehof, "Summarizing Procedures in Concurrent Programs," *Proc. ACM Symp. Principles of Programming Languages,* pp. 245-255, 2004.

[30] P. Godefroid, "Using Partial Orders to Improve Automatic Verification Methods," *Proc. IEEE Conf. Computer Aided Verification,* pp. 176-185, 1991.

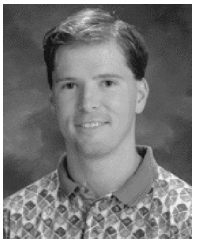
[31] R.E. Rodriguez, M.B. Dwyer, and J. Hatcliff, "Checking Strong Specifications Using an Extensible Software Model Checking Framework," *Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems,* pp. 404-420, 2004.

[32] C. Papadimitriou, *The Theory of Database Concurrency Control.* 1986.

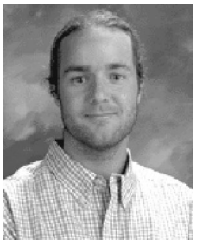
[33] M.P. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems,* vol. 12, no. 3, pp. 463-492, 1990.

[34] C.A.R. Hoare, "Towards a Theory of Parallel Programming," *Operating Systems Techniques,* pp. 61-71, 1972.

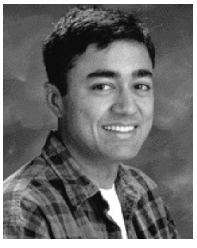
[35] D.B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions," *Language Design for Reliable Software,* pp. 128-137, 1977.
[36] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus," *Proc. Symp. Operating Systems Principles,* pp. 111-122, 1987.
[37] J.L. Eppinger, L.B. Mummert, and A.Z. Spector, *Camelot and Avalon: A Distributed Transaction Facility.* Morgan Kaufmann, 1991.
[38] M.P. Atkinson, K.J. Chisholm, and W.P. Cockshott, "PS-Algol: An Algol with a Persistent Heap," *ACM SIGPLAN Notices,* vol. 17, no. 7, pp. 24-31, 1981.
[39] M.P. Atkinson and D. Morrison, "Procedures as Persistent Data Objects," *ACM Trans. Programming Languages and Systems,* vol. 7, no. 4, pp. 539-559, 1985.
[40] T.L. Harris and K. Fraser, "Language Support for Lightweight Transactions," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 388-402, 2003.
[41] A. Welc, S. Jagannathan, and A.L. Hosking, "Transactional Monitors for Concurrent Objects," *Proc. 18th European Conf. Object-Oriented Programming,* pp. 519-542, 2004.
[42] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno, "Invariant-Based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs," *Proc. Int'l Conf. Software Eng.,* pp. 442-452, 2002.
[43] C. Flanagan, S.N. Freund, and S. Qadeer, "Exploiting Purity for Atomicity," Technical Note 04-05, Williams College, 2004.



**Cormac Flanagan** received the PhD degree from Rice University in 1997. He is a faculty member in the Computer Science Department at the University of California, Santa Cruz, and previously was a research scientist for Digital, Compaq, and Hewlet Packard. His research focuses on static and dynamic checking tools that improve program reliability and reduce development cost, with a particular focus on concurrent software systems. He is the recipient of a Sloan Foundation Fellowship. He is a member of the IEEE.



**Stephen N. Freund** received the PhD degree in computer science from Stanford University in 2000 and was previously a member of the research staff at the Compaq Systems Research Center. He is currently a faculty member in the Computer Science Department at Williams College.



**Shaz Qadeer** received the PhD degree from the Electrical Engeneering and Computer Science Department of the University of California at Berkeley in 1999. He is a member of the Software Productivity Tools group at Microsoft Research. Before joining Microsoft Research, he was a member of the research staff at the Compaq Systems Research Center from 1999 to 2002. His current research is focused on tools for detecting errors in concurrent software systems.