Exploiting Purity for Atomicity

Cormac Flanagan
Department of Computer Science
University of California, Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Shaz Qadeer
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

The notion that certain procedures are *atomic* is a fundamental correctness property of many multithreaded software systems. A procedure is atomic if for every execution there is an equivalent serial execution in which the actions performed by any thread while executing the atomic procedure are not interleaved with actions of other threads. 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 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. The techniques developed in this paper may also be applicable in other approaches for verifying atomicity, such as model checking and dynamic analysis.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification–*reliability*; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

General Terms: Languages, Verification, Reliability.

Keywords: 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 executions 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.

A canonical and widely-applicable non-interference guarantee is *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 multiple benefits.

- The non-interference 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 [12].
- Many synchronization errors can be detected as violations of atomicity.

Recently, a number of analyses have been developed for verifying atomicity, using techniques such as theorem proving [18], static type and effect systems [16, 17], dynamic analysis [12, 38], and model checking [22]. All of these approaches use *reduction* [26, 32], which is based on commuting operations in an execution performed by different threads when they do not interfere with each other to obtain an equivalent *serial execution* (where the operations of each atomic procedure are performed contiguously). An expression is *reducible* if it consists of zero or more *right movers* (steps that right-commute with steps of other threads), followed by at most one atomic step (that need not commute with steps of other threads), followed by zero or more *left movers* (steps that left-commute with steps of other threads).

To illustrate this notion of reduction, consider the procedure `setxy` shown below. In this procedure, the operation `acquire(m)` is a right mover, and the operation `release(m)` is a left mover. Moreover, if all threads access `x` and `y` only while holding the lock `m`, then the writes to `x` and `y` are both right-movers and left-movers since no

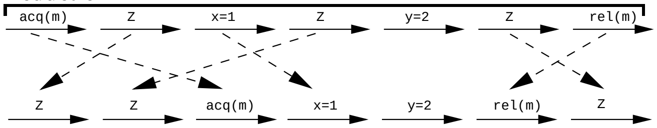
ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

other thread can concurrently access these variables. Thus, as illustrated below, we can reduce any execution of `setxy` interleaved with arbitrary steps (“Z”) from other threads into an equivalent serial execution.

Procedure `setxy`

```
atomic void setxy() {
    acquire(m); x=1; y=2; release(m);
}
```

Reduction



1.1 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 below, which uses a combination of busy-waiting and a compare-and-swap (`CAS`) operation to acquire a mutually-exclusive lock `m` (represented as a boolean).

Procedure `busy_acquire`

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

The operation `CAS(m,0,1)` has no effect and returns `false` if $\texttt{m} \neq \texttt{0}$. However, if $\texttt{m} = \texttt{0}$, 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 [17] cannot verify the atomicity of irreducible procedures like `busy_acquire`. The model checking approach described in [10] can verify the atomicity of `busy_acquire` but is limited by the state-explosion problem.

In this paper, we present a lightweight and scalable static analysis for verifying the atomicity of irreducible procedures such as `busy_acquire`. We present 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).

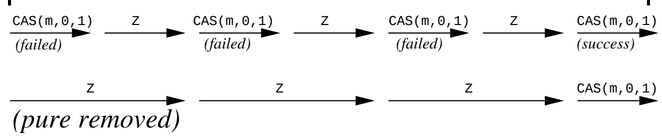
A key novelty of our analysis is the exploitation of *purity* when reasoning about atomicity. Essentially, a code block is pure if, whenever its evaluation terminates normally, it does not change the program state. This restriction does not apply when the block terminates *exceptionally*, for example, 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 exceptionally via the `break` statement. Otherwise, control is returned to the loop head without changing the program store. We introduce the `pure-while` statement to indicate a pure loop and rewrite the `busy_acquire` procedure as follows:

Procedure `busy_acquire` **with a** `pure` **loop**

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

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

Execution trace for `busy_acquire`



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

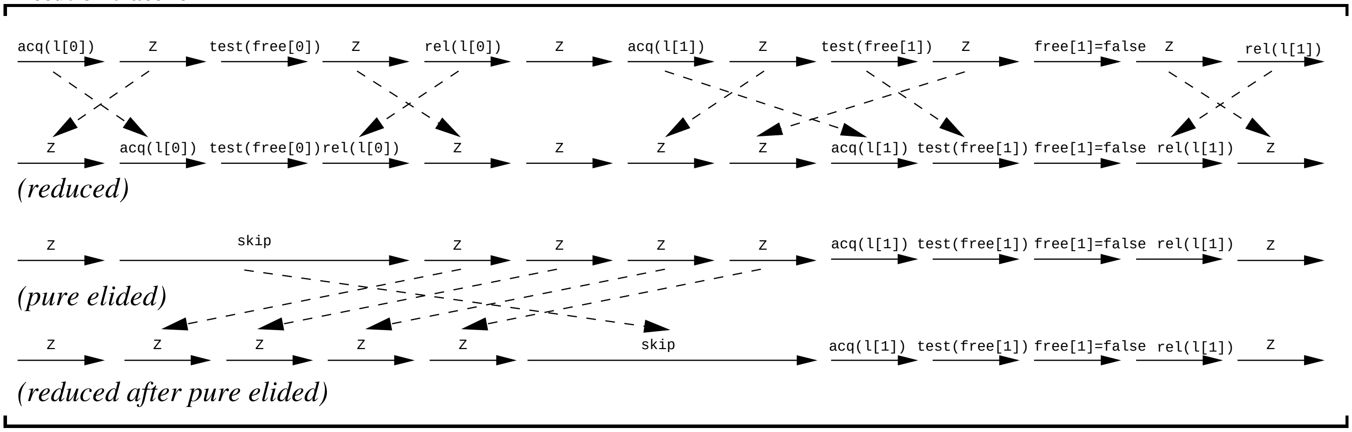
1.2 Abstraction via purity

A more interesting example of our analysis technique is the following procedure `alloc`, 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.

Procedure `alloc`

```
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;
}
```

Execution trace for `alloc`



This procedure is not actually serializable, since there exist some (non-serial) 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 non-atomic 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. In this abstract semantics, the execution of the `pure` block in `alloc` is *optional*, and may or may not be executed on any loop iteration. Thus, the abstract semantics introduces additional nondeterminism and admits additional execution sequences for `alloc`. 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 execution of `alloc` is correct, the contribution of our analysis is to verify that every possible *interleaved* execution of `alloc` yields the same behavior as some serial execution, thus allowing us to conclude that all interleaved executions of `alloc` are also correct. Thus, our analysis enables reasoning about behavior and correctness of abstractly atomic (though not atomic) procedures such as `alloc` using sequential reasoning. In contrast, previous techniques only achieved this goal (of enabling sequential reasoning) 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 above. The first trace contains an execution of `alloc` that succeeds on the second loop iteration, interleaved with arbitrary actions "`Z`" of concurrent threads. We only show steps of `alloc` that modify 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 effect-free, we replace it with `skip` in the execution sequence, at which point applying reduction a second time yields an equivalent serial execution.

The Calvin-R checker [18] can verify similar atomicity properties. However, that tool focuses on checking more complete functional specifications of concurrent programs and has a higher annotation overhead and analysis complexity than the technique in this paper.

1.3 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 access to an unstable variable writes a nondeterministic value and every read access reads a nondeterministic value. Under such an abstract semantics, read and write accesses to 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 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 sketches an extension of our technique based on a more flexible notion of purity. We discuss related work in Section 6 and conclude with a discussion of future directions in Section 7.

2 The language CAP

We formalize our ideas in terms of CAP, a small, imperative, multi-threaded 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. 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 $f(\overline{x})\ e$ introduces a function named f. The formal parameters $\overline{x}$ are bound within the body e, and they may be α-renamed in the usual fashion. For generality, we leave the set of primitives unspecified, but they might include, for example, synchronization primitives such as lock creation, acquire, and release operations for 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 *exceptionally* 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 evaluation of an expression `pure` e is optional, and may be skipped. The keyword `pure` states that, when the expression e is evaluated and terminates normally, that evaluation does not change the program state. (Only exceptionally 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. 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 "_".

Syntax

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

We introduce syntactic sugar for some common constructs.

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

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 behaviors 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 ϵ otherwise. Thus, all accesses to unstable variables or correctly synchronized stable variables will have conflict tag ϵ. Existing analysis techniques [4, 11, 20, 36, 14] can be used to infer these conflict tags.
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, Nielson, and Hankin [30] 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 non-numeric 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 [17, 16] 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 atomic.

Each expression in our language can terminate either normally (by evaluating to a value) or exceptionally (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 ($\top$).

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

$$\begin{array}{ccc}
 & \top & \\
 & | & \\
 & A & \\
 / & & \backslash \\
L & & R \\
\backslash & & / \\
 & B & \\
 & | & \\
 & \bot &
\end{array}$$

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.

Effect System

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

[EXP CONST]
$$\frac{}{\Gamma \vdash c : B \uparrow \bot}$$

[EXP FUN]
$$\frac{\Gamma(f) = \langle a, b \rangle \quad \Gamma \vdash e : a \uparrow b}{\Gamma \vdash f(\overline{x})\, e : B \uparrow \bot}$$

[EXP PRIM]
$$\frac{\Gamma \vdash \overline{e} : a \uparrow b}{\Gamma \vdash p(\overline{e}) : (a; \Gamma(p)) \uparrow b}$$

[EXP READ]
$$\frac{}{\Gamma \vdash x_\epsilon : B \uparrow \bot}$$

[EXP READ RACE]
$$\frac{}{\Gamma \vdash x_\bullet : A \uparrow \bot}$$

[EXP ASSIGN]
$$\frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash x_\epsilon := e : (a; B) \uparrow b}$$

[EXP ASSIGN RACE]
$$\frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash x_\bullet := e : (a; A) \uparrow b}$$

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

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

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

[EXP FORK]
$$\frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash \texttt{fork}\ e : A \uparrow \bot}$$

[EXP INVOKE]
$$\frac{\begin{array}{c}\Gamma \vdash e : a \uparrow b \quad \Gamma \vdash \bar{e} : a' \uparrow b' \\ a'' = \sqcup\{a \mid f \in F \wedge \Gamma(f) = \langle a, b \rangle\} \\ b'' = \sqcup\{b \mid f \in F \wedge \Gamma(f) = \langle a, b \rangle\}\end{array}}{\Gamma \vdash e^F(\overline{e}) : (a; a'; a'') \uparrow (b \sqcup (a; b') \sqcup (a; a'; b''))}$$

[EXP BLOCK]
$$\frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash \texttt{block}\ e : (a \sqcup b) \uparrow \bot}$$

[EXP BREAK]
$$\frac{}{\Gamma \vdash \texttt{break} : \bot \uparrow B}$$

[EXP ATOMIC]
$$\frac{\Gamma \vdash e : a \uparrow b \quad a, b \sqsubseteq A}{\Gamma \vdash \texttt{atomic}\ e : a \uparrow b}$$

[EXP PURE]
$$\frac{\Gamma \vdash e : a \uparrow b \quad a \sqsubseteq A}{\Gamma \vdash \texttt{pure}\ e : B \uparrow b}$$

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

[EXP EMPTY SEQ]
$$\frac{}{\Gamma \vdash \epsilon : B \uparrow \bot}$$

[EXP SEQ]
$$\frac{\Gamma \vdash \bar{e} : a \uparrow b \quad \Gamma \vdash e : a' \uparrow b'}{\Gamma \vdash (\bar{e}.e) : (a; a') \uparrow (b \sqcup (a; b'))}$$

;	$\bot$	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

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

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

$$\begin{array}{rl}\Gamma & : \quad (\mathit{FnName} \to \mathit{Atomicity} \times \mathit{Atomicity}) \\ & \cup \quad (\mathit{Prim} \to \mathit{Atomicity})\end{array}$$

The atomicity of some common primitives are:

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

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 exceptional termination. The rules defining these judgments are mostly straightforward. For example, the "evaluation" of a constant terminates normally, does not interfere with other threads, and cannot terminate exceptionally.

[EXP CONST]
$$\frac{}{\Gamma \vdash c : B \uparrow \bot}$$

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 exceptional atomicity of a let expression reflects the places where the let expression could `break`.

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

Another example of how the exceptional atomicity reflects all the ways in which an expression may `break` is the [EXP IF] rule.

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

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

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

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.

$$\text{[EXP READ]} \quad \frac{}{\Gamma \vdash x_\epsilon : B \uparrow \bot} \qquad \text{[EXP READ RACE]} \quad \frac{}{\Gamma \vdash x_\bullet : A \uparrow \bot}$$

A `block` e expression never terminates exceptionally. 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 exceptionally and is a both mover.

$$\text{[EXP BLOCK]} \quad \frac{\Gamma \vdash e : a \uparrow b}{\Gamma \vdash \texttt{block}\ e : (a \sqcup b) \uparrow \bot} \qquad \text{[EXP BREAK]} \quad \frac{}{\Gamma \vdash \texttt{break} : \bot \uparrow B}$$

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 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 pure block to a both-mover B.

$$\text{[EXP PURE]} \quad \frac{\Gamma \vdash e : a \uparrow b \qquad a \sqsubseteq A}{\Gamma \vdash \texttt{pure}\ e : B \uparrow b}$$

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

$$\text{[EXP ATOMIC]} \quad \frac{\Gamma \vdash e : a \uparrow b \qquad a, b \sqsubseteq A}{\Gamma \vdash \texttt{atomic}\ e : a \uparrow b}$$

Our effect system is sound in the sense that any execution trace of a well-typed program is equivalent to a serial execution of that program. In this serial 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 execution. We refer the interested reader to an extended version of this paper for a full proof of this result [13].

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. We start by considering the double-checked initialization pattern, commonly used to ensure that a shared variable is initialized exactly once [35].[1]

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

4.1 Double-checked initialization

To avoid excessive synchronization overhead, the variable `x` below 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.

Double-Checked Locking

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

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 each possible execution of `init` has an equivalent serial execution, and hence `init` is atomic.

4.2 Caching

In the next example, the function `compute` constructs the value for a given key but is an expensive operation, so we wish to cache previously-computed results. We assume the cache operations `cachePut` and `cacheGet` are atomic (for example, because they acquire the lock protecting the cache); `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 if when concurrently invoked by multiple threads.

Caching

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

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

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

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 and to produce an equivalent, reducible trace. Thus, all executions through the function `lookup` have an equivalent serial execution, and so the function `lookup` is atomic.

4.3 Wait and notify

The `wait` and `notify` routines facilitate notification between concurrent threads. The routine `wait(l)` should only be called if the lock `l` is held; this routine then releases l, blocks until a concurrent thread calls `notify(l)`, and then returns after re-acquiring 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 following code fragment illustrates the use of `wait` to iterate until the variable `x` (protected by lock `l`) is `false`, and we assume that *body* is atomic.

Wait example

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

For this example, even though `wait(l)` is not atomic, our type system can verify that the entire code fragment, although irreducible, is still atomic. Before applying our type system, we first need to refactor this code using the following equivalence rules for program expressions:

Equivalence rules

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

Applying these rules to the above code fragment in the appropriate manner yields the following refactored code 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.)

Refactored wait example

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

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

4.4 Packet counter

The following example 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 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.)

Packet counter

```
int _packetCount;
Queue packets;

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

atomic void receive(Packet p) {
    _packetCountϵ++;
    enqueue(packetsϵ, pϵ);
}
```

5 Abstraction via weak purity

The technique of optionally-executed pure blocks is sufficient to handle many examples, such as those described in Section 4. In this section, we sketch a more general notion of purity that yields a more expressive effect system for atomicity.

Consider the following function which models optimistic concurrency control based on transaction retry. We have a shared data variable `z` that we wish to update according to `z = f(z)`. However, the function `f` is a long-running operation, so we do not wish to hold `z`'s protecting lock `m` when computing `f`. Instead, we record a local copy `x` of `z`, compute `f(x)`, and then update `z` if the value of `z` has not changed. If `z` has changed, then we retry the transaction. This technique ensures that the update of `z` to `f(z)` is serialized with respect to other updates to `z`, without requiring the lock guarding `z` to be held while computing `f(z)`.

Transaction retry

```
atomic void apply_f() {
    int x, fx;
    weak-pure {
        acquire(m);
        xε = zε;
        release(m);
    }
    weak-pure-while (true) {
        fxε = f(xε);
        acquire(m);
        if (xε == zε) {
            zε = fxε;
            release(m);
            break;
        }
        xε = zε;
        release(m);
    }
}
```

The code block before the loop is not pure because it modifies the local variable `x`. The body of the while loop is also not pure because it modifies the local variables `x` and `fx`. To deal with this prototypical example, we introduce a weaker notion of purity that allows us to prove the atomicity of `apply_f`. We first classify variables as either *thread-local* or *shared*, depending on whether the variable may be accessed by one or multiple threads, respectively. A code block can be annotated as `weak-pure` if it is atomic and does not modify any shared variables under normal termination. Its evaluation may modify thread-local variables, making `weak-pure` less restrictive than `pure` (which may not modify either thread-local or shared variables under normal termination).

The construct `weak-pure-while` e_1 e_2 is desugared in a fashion similar to `pure-while`:

$$\texttt{weak-pure-while}\ e_1\ e_2 \equiv \\ \texttt{block loop weak-pure}\ \{\ \texttt{if}\ e_1\ e_2\ \texttt{break}\ \}$$

and is semantically equivalent to `while` e_1 e_2 (provided $\{\texttt{if}\ e_1\ e_2\ \texttt{break}\}$ is weakly-pure).

The abstract semantics executes `weak-pure` e as normal, except that if e accesses a shared variable x, then an arbitrary value is returned. This arbitrary value is consistent for all accesses to x during a single execution of e. Thus the abstract semantics for `weak-pure` introduces additional execution traces and we need to check (formally or informally) the sequential correctness of `apply_f` under this abstract semantics. The choice of abstract semantics ensures that `weak-pure` blocks neither read nor modify values of shared variables, and enables us to treat atomic `weak-pure` blocks as both-movers.

The typing rule for `weak-pure` is identical to the rule used to reason about pure blocks. The rule requires that the normal atomicity of the body of a weak-pure block must be at most A and "optimizes" the normal atomicity of the block to a both-mover B.

[EXP WEAKPURE]

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

6 Related work

Lipton [26] 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 [8, 3, 25, 6, 29]. Bruening [5] and Stoller [37] have used reduction to improve the efficiency of model checking. Flanagan and Qadeer have pursued a similar approach [15], and Qadeer *et al* [33] 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 [17, 16]. 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 [18]. 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 [38] 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 [12]. 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* [22], and they present two approaches, based on Lipton's theory of reduction and partial-order reductions [19], respectively. Model checking offers several advantages over our effect system. For example, it requires many fewer programmer-inserted annotations and can accommodate complex synchronization disciplines more easily. Their experimental results suggest that verifying atomicity via model-checking is feasible for unit-testing. Their approach currently only verifies the atomicity 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, Robby *et al* [34] 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, and possibly specify its behavior with standard pre- and post-conditions. However, the entire `alloc` function could not be shown to be atomic in an abstract sense, without performing an analysis like the one outlined in this paper.

A number of tools have been developed for detecting race conditions, both statically and dynamically. The Race Condition

Checker [11] uses a type system to catch race conditions in Java programs. This approach has been extended [4] and adapted to other languages [20]. Other static race detection tools include Warlock [36], for ANSI C programs, and ESC/Java [14], 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 [31], a correctness condition for database transactions, and linearizability [23], 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 [24] and Lomet [28] first proposed the use of atomic blocks for synchronization, and the Argus [27] and Avalon [9] projects developed language support for implementing atomic objects. Persistent languages [1, 2] augment atomicity with data persistence in order to introduce transactions into programming languages. Other recent approaches to supporting atomicity include lightweight transactions [21, 39] and automatic generation of synchronization code from high-level specifications [7].

7 Conclusion

Atomicity is an important correctness property for multithreaded software. Current reduction-based tools can verify atomicity requirements 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 many such problematic cases, by applying reduction to an abstraction of the program. The abstraction notions we have presented —based on optional execution, purity, and instability— are intuitive, and the correctness of abstractly atomic procedures under the serial abstract semantics can be verified using sequential reasoning techniques. Our static analysis then verifies that all interleaved executions of these abstractly atomic procedures are 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 [22]) could identify and exploit pure code blocks while performing reduction. Dynamic analyses for atomicity [12] could potentially benefit from these ideas as well.

Acknowledgments

We thank Leslie Lamport and Chandu Thekkath for useful discussions. Cormac Flanagan was partly supported by the National Science Foundation 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 National Science Foundation under Grant CCR-0341387 and by faculty research funds granted by Williams College.

8 References

[1] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, 1981.

[2] M. P. Atkinson and D. Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, 1985.

[3] 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. Springer-Verlag, 1989.

[4] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 56–69, 2001.

[5] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, Massachusetts Institute of Technology, 1999.

[6] E. Cohen and L. Lamport. Reduction in TLA. In *Proceedings of the International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[7] 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*, pages 442–452, 2002.

[8] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 155–169, 1977.

[9] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[10] C. Flanagan. Verifying commit-atomicity using model-checking. In *SPIN 2004: 11th International SPIN Workshop on Model Checking of Software*, pages 252–266, 2004.

[11] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[12] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 256–267, 2004.

[13] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. Technical Note 04-05, Williams College, 2004.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[15] C. Flanagan and S. Qadeer. Transactions for software model checking. In *Proceedings of the Workshop on Software Model Checking*, 2003.

[16] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

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

[18] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Journal of Object Technology*, 2004. (to appear). A preliminary version appeared in *Workshop on Formal Techniques for Java-like Programs*, 2003.

[19] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the IEEE Conference on Computer Aided Verification*, pages 176–185, 1991.

[20] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*, pages 13–25, 2003.

[21] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on*

Object-Oriented Programming, Systems, Languages and Applications, pages 388–402, 2003.

[22] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 175–190, 2004.

[23] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[24] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71. Academic Press, 1972.

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

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

[27] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Symposium on Operating Systems Principles*, pages 111–122, 1987.

[28] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *Language Design for Reliable Software*, pages 128–137, 1977.

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

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

[31] C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, 1986.

[32] D. Peled. Combining partial order reductions with on-the-fly model checking. In D. Dill, editor, *CAV 94: Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 377–390. Springer-Verlag, 1994.

[33] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 245–255, 2004.

[34] Robby, E. Rodriguez, M. B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 404–420, 2004.

[35] D. C. Schmidt and T. H. Harrison. Double-checked locking – a optimization pattern for efficiently initializing and accessing thread-safe objects. In R. Martin, F. Buschmann, and D. Riehle, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.

[36] N. Sterling. WARLOCK — a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, Winter 1993.

[37] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *Workshop on Model Checking and Software Verification*, pages 224–244, 2000.

[38] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proceedings of the Workshop on Runtime Verification*, volume 89(2) of *Electronic Notes in Computer Science*. Elsevier, 2003.

[39] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, 2004.

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 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 Π 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 not appearing in X may be modified.

[PURE READ]

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

[PURE ASSIGN]

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

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

[PURE LET]

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

We introduce specific rules for the primitive operations that acquire and release mutual exclusion 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 top-level judgment

$$\Pi \vdash_p P$$

states that the annotation `pure` e is valid if

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

That is, a pure block may not change any stable variables or terminate with a different set of locks held than when evaluation started: see [PURE PROG]. This rule also requires that every function in Π is pure.

Purity Effect System

$$\boxed{\Pi, X \vdash_p e : L_1 \to L_2}$$

[PURE CONST]
$$\frac{}{\Pi, X \vdash_p v : L \to L}$$

[PURE WRONG]
$$\frac{}{\Pi, X \vdash_p \mathbf{wrong} : L \to L}$$

[PURE PRIM]
$$\frac{\Pi, X \vdash_p \overline{e} : L_1 \to L_2 \qquad p \text{ is effect-free}}{\Pi, X \vdash_p p(\overline{e}) : L_1 \to L_2}$$

[PURE READ]
$$\frac{}{\Pi, X \vdash_p x_r : L \to L}$$

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

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

[PURE LOOP]
$$\frac{\Pi, X \vdash_p e : L \to L}{\Pi, X \vdash_p \texttt{loop } e : L \to L}$$

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

[PURE IF]
$$\frac{\Pi, X \vdash_p e : L_1 \to L_2 \qquad \Pi, X \vdash_p e_i : L_2 \to L_3}{\Pi, X \vdash_p \texttt{if } e\ e_1\ e_2 : L_1 \to L_3}$$

[PURE BREAK]
$$\frac{}{\Pi, X \vdash_p \texttt{break} : L \to L'}$$

[PURE INVOKE]
$$\frac{\Pi, X \vdash_p e : L_1 \to L_2 \qquad \Pi, X \vdash_p \bar{e} : L_2 \to L_3 \qquad F \subseteq \Pi}{\Pi, X \vdash_p e^F(\overline{e}) : L_1 \to L_3}$$

$$\boxed{\Pi, X \vdash_p \bar{e} : L_1 \to L_2}$$

[PURE EMPTY SEQ]
$$\frac{}{\Pi, X \vdash_p \epsilon : L \to L}$$

[PURE SEQ]
$$\frac{\Pi, X \vdash_p \bar{e} : L_1 \to L_2 \quad \Pi, X \vdash_p e : L_2 \to L_3}{\Pi, X \vdash_p \bar{e}, e : L_2 \to L_3}$$

$$\boxed{\Pi \vdash_p P}$$

[PURE PROG]
$$\frac{\begin{array}{rcl} P \text{ contains } f(\overline{x})\ e \text{ and } f \in \Pi & \Rightarrow & \Pi, \mathit{UnstableVar} \vdash_p e : \emptyset \to \emptyset \\ P \text{ contains } \texttt{pure}\ e & \Rightarrow & \Pi, \mathit{UnstableVar} \vdash_p e : \emptyset \to \emptyset \end{array}}{\Pi \vdash_p P}$$