Types for Atomicity

Cormac Flanagan Shaz Qadeer

HP Systems Research Center,
1501 Page Mill Road, Palo Alto, CA 94304

ABSTRACT

Ensuring the correctness of multithreaded programs is difficult, due to the potential for unexpected and nondeterministic interactions between threads. Previous work has addressed this problem by devising tools for detecting *race conditions*, a situation where two threads simultaneously access the same data variable, and at least one of the accesses is a write. However, the absence of race conditions is neither necessary nor sufficient to ensure the absence of errors due to unexpected thread interactions.

We propose that a stronger non-interference property is required, namely the *atomicity* of code blocks, and we present a type system for specifying and verifying such atomicity properties. The type system allows statement blocks and functions to be annotated with the keyword `atomic`. If the program type checks, then the type system guarantees that for any arbitrarily-interleaved program execution, there is a corresponding execution with equivalent behavior in which the instructions of each atomic block executed by a thread are not interleaved with instructions from other threads. This property allows programmers to reason about the behavior of well-typed programs at a higher level of granularity, where each atomic block is executed "in one step", thus significantly simplifying both formal and informal reasoning.

Our type system is sufficient to verify a number of interesting examples. For example, it can prove that many methods of `java.util.Vector` are atomic, even though some methods have benign race conditions, and would be rejected by earlier type systems for race detection.

Categories and Subject Descriptors

D.1.3 Concurrent Programming, parallel programming; D.2.4 Software/Program Verification

General terms

Reliability, Security, Languages, Verification

TLDI'03, January 18, 2003, New Orleans, Louisiana, USA.

1. INTRODUCTION

Ensuring the correctness of multithreaded programs is difficult, due to the potential for unexpected and nondeterministic interactions between threads. Previous work has addressed this problem by devising type systems [9, 10] and other static [11] and dynamic [20] 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 following code, in which the data variable x is protected by the lock l, and t is a thread-local variable:

$$\begin{array}{l} \mathit{acquire}(l);\ t := x;\ \mathit{release}(l);\\ t := t + 1;\\ \mathit{acquire}(l);\ x := t;\ \mathit{release}(l); \end{array}$$

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

We propose that a stronger non-interference property is required, namely the *atomicity* of code blocks. If a code block is atomic, we can safely reason about the program's behavior at a higher level of granularity, where the atomic block is executed "in one step", even though the scheduler is free to interleave threads at instruction-level granularity.

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 illustrated in the example above, race-freedom is not sufficient to prove atomicity; nor is it necessary, as we will show later in Section 5. Thus, even though programmers typically have precise expectations regarding atomic code fragments, existing race-detection techniques cannot verify these expectations.

In this paper, we present a type system for verifying atomicity of code blocks in multithreaded programs. Statement blocks and functions in the target program can be annotated with the keyword `atomic`. Our type system guarantees that each atomic block in a well-typed program is *reducible*, which means that for any (arbitrarily-interleaved) execution, there is a corresponding execution with equivalent behavior in which the execution of each atomic block by a thread is not interleaved with instructions from other threads.

1.1 Motivating example

To illustrate the importance of atomicity, consider a bank with a single account whose current balance is stored in the variable `balance`, which is initially 0. The function `deposit` deposits money into the account.

```
int balance = 0;

void deposit(int amt) {
    balance = balance + amt;
}
```

Suppose that two ATMs simultaneously deposit money into the account, a situation that is modeled by the following multithreaded program.

```
fork { deposit(10); }
fork { deposit(15); }
```

There is clearly a race on the variable `balance`. If the two calls to `deposit` are interleaved, the final value of `balance` may reflect only one of the two deposits to the account.

We can fix this error by introducing a mutual exclusion lock `m` and re-coding the function `deposit`.

```
Mutex m;

void deposit(int amt) {
    acquire(m);
    balance = balance + amt;
    release(m);
}
```

This new implementation of `deposit` is race-free, and behaves correctly even when called from multiple threads. However, in general, the absence of races is not sufficient to ensure the absence of errors due to thread interactions. We illustrate this point by adding two functions—`read_balance` to return the current account balance and `withdraw` to take money out of the account.

```
int read_balance() {
    int t;
    acquire(m);
    t = balance;
    release(m);
    return t;
}

int withdraw(int amt) {
    int t = read_balance();
    acquire(m);
    if (t <= amt) {
        balance = 0;
    } else {
        balance = balance - amt;
        t = amt;
    }
    release(m);
    return t;
}
```

Even though there are no races in the functions shown above, the function `withdraw` is not atomic and may not behave correctly. For example, consider two concurrent transactions on the account —a withdrawal and a deposit— issued at a time when the account balance is 10.

```
fork { withdraw(10); };            // Thread 1
fork { deposit(10); };             // Thread 2
```

We would expect the account balance to remain 10 after the program terminates. However, there is an execution that violates this expectation. Suppose the scheduler first performs the call to `read_balance` in `Thread 1` which returns 10. The scheduler then switches to `Thread 2` and completes the execution of this thread ending with `balance` = 20. Finally, the scheduler switches back to `Thread 1` and completes the execution setting `balance` to 0. Even though a race-analysis tool will report that there are no races in the program shown above, unexpected interaction between the threads can lead to incorrect behavior.

The bank account interface provided by the three functions `deposit`, `read_balance`, and `withdraw` is intended to be atomic, a property common to many interfaces in multithreaded programs. A programmer using an atomic interface should not have to worry about unexpected interactions between concurrent invocations of the methods of the interface. Our type system provides the means to specify and verify atomicity properties, thus catching errors such as the one in `withdraw` above.

1.2 Types for atomicity

As we have seen, although the notions of atomicity and race-freedom are closely related, and both are commonly achieved using locks, race-freedom is neither necessary nor sufficient for ensuring atomicity. We now present a brief overview of our type system for checking atomicity.

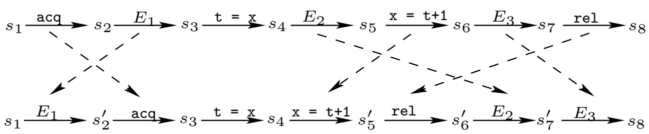
We allow any function and any code block to be annotated with keyword `atomic`, meaning that it is intended to be reducible. A code block is reducible if for any execution in which this code block has been fully executed, there is another execution with the same final state in which all actions of this code block happen consecutively without any interleaved actions by other threads.

Our type system uses Lipton's theory of right and left movers [18], to prove the correctness of `atomic` annotations. An action a is a *right mover* if whenever a is followed by an 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.

The type system classifies operations as left or right movers as follows. The lock acquire operation is a right mover, since it right-commutes with any immediately following operation performed by a second thread, including reads and writes of variables, and acquires and releases of other locks. Note that the second thread cannot either acquire or release the lock that was acquired by the first thread. For similar reasons, the lock release operation is a left mover. To determine whether a variable access (read or write) is a right or left mover, our type system relies on a race-detection system to determine if other threads may access that variable at the same time. If there is no race condition on the variable, then such simultaneous accesses can not occur, and hence the variable access is both a right and a left mover.

Suppose a code block contains a sequence of 0 or more right movers followed by a single atomic action followed by a sequence of 0 or more left movers. Then an execution where this code block has been fully executed can be *reduced* to another execution with the same resulting state where the code block is executed atomically without any interleaved actions by other threads.

To illustrate these ideas, consider a method that acquires a lock (a right mover action), reads a variable `x` protected by that lock (both a left and right mover action), updates that variable (also a both mover), and then releases the lock (a left mover action). Suppose the actions of a thread that calls this method are interleaved with actions E_1, E_2, E_3 of other threads. Then the following diagram implies that there exists an equivalent execution where the operations of the method are not interleaved with operations of other threads; thus the method is atomic.



For our bank account example, there are no race conditions, and hence our type system infers that the implementations of `deposit` and `read_balance` are atomic. The body of `withdraw` consists of a call to the atomic function `read_balance`, followed by an atomic block of code. However, the sequential composition of two atomic code blocks is not necessarily atomic. Therefore, the type system is unable to prove that the implementation of `withdraw` is atomic and reports an error.

2. THE LANGUAGE CAT

We formalize the ideas of this paper in terms of Cat, a small, imperative, multithreaded language with higher-order functions and dynamic thread creation. Cat is loosely modeled after Cyclone [16], a type-safe variant of the C programming language, and is essentially a restricted subset of C, extended with facilities for reasoning about atomicity.

2.1 Syntax

Cat expressions include values, variable reference and assignment, primitive and function applications, conditionals, and while loops. In addition, the expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ allocates a fresh variable x in the heap, and initializes x with the result of evaluating the expression e_1. The variable x is bound within e_2, and may be α-renamed in the usual fashion. We use $e_1; e_2$ to abbreviate $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ whenever x does not occur free in e_2. The language supports multithreaded programming via the construct $\texttt{fork}\ e$, which spawns a new thread for the evaluation of e. Expressions can be annotated with the keyword `atomic`; the type system verifies that such expressions are reducible. The `in-atomic` construct is used to model program evaluation and should not appear in source programs.

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

Syntax

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

Synchronization locations are references to *synchronization values*. Our experience in analyzing systems software indicates that a variety of synchronization mechanisms are used in practice. For generality, we leave the set of synchronization values unspecified, but they might include, for example, mutual exclusion locks, reader-writer locks, semaphores, etc. These synchronization values are manipulated by primitive applications $p(\overline{e})$, which might include operations to allocate, acquire, and release mutual exclusion locks. In addition, the set of primitives also include arithmetic operations and the `assert` primitive.

To simplify our presentation, our type system does not reason about race-conditions, since this topic has been covered by a variety of previous tools [3, 10, 14]. Instead, we simply assume that a race detection system has already annotated each variable access (read or write) with an *conflict tag*, which is $\bullet$ if that access may be involved in a race condition, and is ϵ otherwise.

In addition, we assume that a program flow analysis has been used to annotate each function call $e^F(\overline{e})$ with a *call tag* F denoting the set of functions that may be invoked by that call. Finally, given that our goal is static detection of software defects, it would also be appropriate to run a conventional type checker to catch, for example, applications of arithmetic operations to non-numeric arguments. Factoring out all these separate issues allows us to develop a more elegant type system that focuses on the task at hand, namely, checking the correctness of atomicity annotations.

2.2 Semantics

A program state is a 3-tuple consisting of a heap H, which is a partial map from variables to values, a synchronization heap M, which is a partial map from synchronization locations to synchronization values, and a sequence T of threads. Each thread is either an expression or **wrong**. A thread becomes **wrong** if a primitive operation is applied to incorrect arguments. An evaluation context is used to identify the next part of an expression to be executed. An evaluation context E is an expression with a "hole" [ ] in place of the next sub-expression to be evaluated, and $E[e]$ denotes the operation of filling the hole in E with the expression e.

The transition relation $\rightarrow_i$ performs a single step of thread i. The notation $H[x := v]$ denotes a new heap that is identical to H except that it maps x to v. The `in-atomic` construct records that execution is proceeding inside an atomic block, and should only appear in an evaluation context positions.

The meaning of a primitive operation p is defined using the partial function $\mathcal{I}_p$, which takes a sequence of argument

Semantics

State space

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

Evaluation contexts

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

Transition relations

$$\rightarrow_i, \rightarrow, \twoheadrightarrow \;\subseteq\; \mathit{ProgState} \times \mathit{ProgState}$$

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

$$\begin{array}{rcll}
H, M, T.E[\texttt{while } e_1\ e_2].T' & \rightarrow_i & H, M, T.E[\texttt{if } e_1\ \{e_2;\texttt{while } e_1\ e_2\}\ 0].T' & \\
H, M, T.E[\texttt{if } v\ e_1\ e_2].T' & \rightarrow_i & H, M, T.E[e_1].T' & \text{if } v \neq 0 \\
H, M, T.E[\texttt{if } 0\ e_1\ e_2].T' & \rightarrow_i & H, M, T.E[e_2].T' & \\
H, M, T.E[\texttt{let } x = v \texttt{ in } e].T' & \rightarrow_i & H[x := v], M, T.E[e].T' & \text{if } x \notin \mathit{dom}(H) \\
H, M, T.E[x_r].T' & \rightarrow_i & H, M, T.E[H(x)].T' & \\
H, M, T.E[x_r := v].T' & \rightarrow_i & H[x := v], M, T.E[v].T' & \\
H, M, T.E[p(\overline{v})].T' & \rightarrow_i & H, M', T.E[v'].T' & \text{if } \mathcal{I}_p(\overline{v}, M) = \langle v', M' \rangle \\
H, M, T.E[p(\overline{v})].T' & \rightarrow_i & H, M, T.\mathbf{wrong}.T' & \text{if } \mathcal{I}_p(\overline{v}, M) = \mathbf{wrong} \\
H, M, T.E[(f(\overline{x})\ e)^F(\overline{v})].T' & \rightarrow_i & H[\overline{x} := \overline{v}], M, T.E[e].T' & \text{if } \overline{x} \cap \mathit{dom}(H) = \emptyset \\
H, M, T.E[\texttt{atomic } e].T' & \rightarrow_i & H, M, T.E[\texttt{in-atomic } e].T' & \\
H, M, T.E[\texttt{in-atomic } v].T' & \rightarrow_i & H, M, T.E[v].T' & \\
H, M, T.E[\texttt{fork } e].T' & \rightarrow_i & H, M, T.E[0].T'.e & \\
 & & & \\
H, M, T & \rightarrow & P' & \text{if } H, M, T \rightarrow_i P' \\
 & & & \\
H, M, T & \twoheadrightarrow & P' & \text{if } H, M, T \rightarrow_i P' \text{ and} \\
 & & & T_j \text{ does not contain } \texttt{in-atomic} \text{ for } j \neq i
\end{array}$$

values, a synchronization heap, and an integer identifying the current thread, and returns a pair of a return value and a (possibly modified) synchronization heap. In addition, if the primitive is applied to incorrect arguments, $\mathcal{I}_p$ may instead return **wrong**:

$$\begin{aligned}\mathcal{I}_p :& (\mathit{Const} \cup \mathit{SyncLoc})^* \times \mathit{SyncHeap} \times \mathit{Int} \rightharpoonup \\ & ((\mathit{Const} \cup \mathit{SyncLoc}) \times \mathit{SyncHeap}) \cup \{\mathbf{wrong}\}\end{aligned}$$

For example, the semantics of `assert`, addition and operations to allocate, acquire, and release locks, might be defined as follows (where ϵ is the empty sequence and $m.n$ is a sequence of length two):

$$\begin{aligned}
&\mathcal{I}_{\texttt{assert}}(v, M, \mathit{tid}) = \\
&\qquad \begin{cases} \langle 0, M\rangle & \text{if } v \neq 0 \\ \mathbf{wrong} & \text{if } v = 0 \end{cases} \\
&\mathcal{I}_{+}(m.n, M, \mathit{tid}) = \\
&\qquad \langle m+n, M\rangle \\
&\mathcal{I}_{\texttt{new_lock}}(\epsilon, M, \mathit{tid}) = \\
&\qquad \langle m, M[m := \langle \texttt{lock}, 0\rangle]\rangle \qquad \text{if } m \notin \mathit{dom}(M) \\
&\mathcal{I}_{\texttt{acquire}}(m, M[m := \langle \texttt{lock}, 0\rangle], \mathit{tid}) = \\
&\qquad \langle m, M[m := \langle \texttt{lock}, \mathit{tid}\rangle]\rangle \\
&\mathcal{I}_{\texttt{release}}(m, M, \mathit{tid}) = \\
&\qquad \begin{cases} \langle m, M[m := \langle \texttt{lock}, 0\rangle]\rangle & \text{if } M(m) = \langle \texttt{lock}, \mathit{tid}\rangle \\ \mathbf{wrong} & \text{otherwise} \end{cases}
\end{aligned}$$

An `assert` 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 only proceed 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.

The fine-grain transition relation $\rightarrow$ performs a single step of an arbitrarily chosen thread. We use $\rightarrow^+$ and $\rightarrow^*$ to denote the transitive and reflexive-transitive closure of $\rightarrow$, respectively. The coarse-grain transition relation $\twoheadrightarrow$ 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 $\twoheadrightarrow$ does not interleave the execution of an `atomic` block with instructions from other threads.

We use the operational semantics to formalize the meaning of the conflict tags and call tags. An expression e is *about to read* x if $e \equiv E[x_r]$. Similarly, e is *about to write* x if $e \equiv E[x_r := v]$. The conflict tags in a program P are *correct* if whenever $P \rightarrow^* H, M, T$:

1. if $T_i \equiv E[x_\epsilon]$, then no other thread in T is about to write x, and

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

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

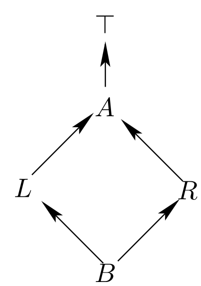
3. CAT TYPE SYSTEM

Traditional type systems reason about the set of values produced by particular expressions, and focus on ensuring that primitives are only applied to appropriate values. Race-free type systems [3, 10, 14] check an additional property, namely that the appropriate protecting lock is held whenever a shared variable is accessed. Our type system focuses on an additional important correctness property: ensuring that all blocks and functions marked `atomic` are in fact reducible.

In general, an expression is reducible if it consists of 0 or more 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 0 or more steps that left-commute with steps of other threads. The type system assigns to each expression an *atomicity*, which 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); can be viewed as a single atomic action (A); or whether none of these properties hold ($\top$).

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

Atomicities are ordered by $B \sqsubset a \sqsubset A \sqsubset \top$ for $a \in \{L, R\}$, as illustrated below. Let $\sqcup$ denote the join operator based on this ordering. If atomicities a_1 and a_2 reflect the behavior of expressions e_1 and e_2 respectively, then the *sequential composition* $a_1; a_2$ reflects the behavior of $e_1; e_2$, and is defined by the following table.



;	B	L	R	A	$\top$
B	B	L	R	A	$\top$
R	R	A	R	A	$\top$
L	L	L	$\top$	$\top$	$\top$
A	A	A	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

Similarly, if atomicity a reflects the behavior of e, then the *iterative closure* a^* reflects the behavior of executing e multiple times, and is defined by:

$$\begin{aligned} B^* &= B \\ R^* &= R \\ L^* &= L \\ A^* &= \top \\ \top^* &= \top \end{aligned}$$

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) both sequential composition and iterative closure distribute over joins.

A type environment Γ maps function names and primitives to corresponding atomicities. Our type system checks that the atomicity of the implementation of a function f is $\Gamma(f)$. However, the type system assumes that the atomicities of primitives are correct. 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 $\mathcal{I}_p(\overline{v}, M, i) = \langle v', M'\rangle$ and $\mathcal{I}_q(\overline{u}, M', j) = \langle u', M''\rangle$, then there is M''' such that $\mathcal{I}_q(\overline{u}, M, j) = \langle u', M'''\rangle$ and $\mathcal{I}_p(\overline{v}, M''', i) = \langle v', M''\rangle$.

Type System

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

[EXP CONST] $\dfrac{}{\Gamma \vdash c : B}$ [EXP SYNCLOC] $\dfrac{}{\Gamma \vdash m : B}$ [EXP FUN] $\dfrac{\Gamma \vdash e : \Gamma(f)}{\Gamma \vdash f(\overline{x})\ e : B}$ [EXP PRIM] $\dfrac{\Gamma \vdash e_i : a_i}{\Gamma \vdash p(\overline{e}) : (a_1; \ldots; a_n; \Gamma(p))}$ [EXP READ] $\dfrac{}{\Gamma \vdash x_\epsilon : B}$ [EXP READ RACE] $\dfrac{}{\Gamma \vdash x_\bullet : A}$

[EXP ASSIGN] $\dfrac{\Gamma \vdash e : a}{\Gamma \vdash x_\epsilon := e : (a; B)}$ [EXP ASSIGN RACE] $\dfrac{\Gamma \vdash e : a}{\Gamma \vdash x_\bullet := e : (a; A)}$ [EXP LET] $\dfrac{\Gamma \vdash e_1 : a_1 \quad \Gamma \vdash e_2 : a_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : (a_1; a_2)}$ [EXP IF] $\dfrac{\Gamma \vdash e : a \quad \Gamma \vdash e_i : b_i}{\Gamma \vdash \texttt{if}\ e\ e_1\ e_2 : (a; (b_1 \sqcup b_2))}$

[EXP WHILE] $\dfrac{\Gamma \vdash e_1 : a_1 \quad \Gamma \vdash e_2 : a_2}{\Gamma \vdash \texttt{while}\ e_1\ e_2 : (a_1; (a_2; a_1)^*)}$ [EXP INVOKE] $\dfrac{\Gamma \vdash e : a \quad \Gamma \vdash e_i : a_i}{\Gamma \vdash e^F(\overline{e}) : (a; a_1; \ldots; a_n; (\sqcup_{f \in F} \Gamma(f)))}$ [EXP FORK] $\dfrac{\Gamma \vdash e : a}{\Gamma \vdash \texttt{fork}\ e : A}$

[EXP ATOMIC] $\dfrac{\Gamma \vdash e : a \quad a \sqsubseteq A}{\Gamma \vdash \texttt{atomic}\ e : a}$ [EXP INATOMIC] $\dfrac{\Gamma \vdash e : a \quad a \sqsubseteq A}{\Gamma \vdash \texttt{in-atomic}\ e : a}$ [WRONG] $\dfrac{}{\Gamma \vdash \textbf{wrong} : B}$

$\boxed{\Gamma \vdash P}$

[PROG]
$$\dfrac{\begin{array}{c}\forall x \in \mathit{dom}(H).\ \Gamma \vdash H(x) : B \quad \Gamma \vdash T_i : a_i \\ \forall e.(T_i \text{ contains } \texttt{in-atomic}\ e \Rightarrow \exists E.\ T_i = E[\texttt{in-atomic}\ e])\end{array}}{\Gamma \vdash H, M, T_1 \ldots T_n}$$

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

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

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

For example, these two properties are satisfied by the following atomicities:

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

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

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

$$\Gamma \vdash e : a\,,$$

which states that expression e has atomicity a. The type rules defining these judgments are mostly straightforward. The atomicity of a constant is B, since the "evaluation" of a constant does not interfere with other threads.

[EXP CONST]
$$\dfrac{}{\Gamma \vdash c : B}$$

The rule [EXP LET] states that the atomicity of a let expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is the composition $a_1; a_2$ of the atomicities of e_1 and e_2.

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

The rule [EXP WHILE] for $\texttt{while}\ e_1\ e_2$ determines the atomicities a_1 and a_2 of e_1 and e_2, and states that the atomicity of the while loop is $a_1; (a_2; a_1)^*$, reflecting the iterative nature of the while loop.

[EXP WHILE]
$$\dfrac{\Gamma \vdash e_1 : a_1 \quad \Gamma \vdash e_2 : a_2}{\Gamma \vdash \texttt{while}\ e_1\ e_2 : (a_1; (a_2; a_1)^*)}$$

The atomicity of a primitive application $p(\overline{e})$ is the sequential composition of the atomicities of the argument expressions $\overline{e}$, followed by the atomicity $\Gamma(p)$ of p.

[EXP PRIM]
$$\dfrac{\Gamma \vdash e_i : a_i}{\Gamma \vdash p(\overline{e}) : (a_1; \ldots; a_n; \Gamma(p))}$$

The atomicity of a variable read x_r depends on the conflict tag r. If $r = \epsilon$, then this read commutes (in both directions) with steps of other threads, and so has atomicity B.

[EXP READ]
$$\dfrac{}{\Gamma \vdash x_\epsilon : B}$$

If $r = \bullet$, then this read has atomicity A, indicating that it is an atomic action that may not commute with steps of other threads.

[EXP READ RACE]
$$\dfrac{}{\Gamma \vdash x_\bullet : A}$$

The type rules for a variable write are similar. Finally, the atomicity of the body of an atomic construct is required to be at most A.

4. CORRECTNESS OF TYPE SYSTEM

Our type system guarantees that in any well-typed program, each atomic block is reducible. Our proof of this result depends on the following theorem, which is inspired by the reduction theorem of Cohen and Lamport [6].

We start by introducing some additional notation. For any state predicate $X \subseteq \mathit{ProgState}$ and transition relation $Y \subseteq \mathit{ProgState} \times \mathit{ProgState}$, by X/Y we mean the transition relation obtained by restricting Y to pairs whose first component is in X. Similarly, by $Y \backslash X$ we mean the restriction of Y to pairs whose second component is in X.

The composition $Y \circ Z$ of two transition relations Y and Z is the set of all transitions (p, r) such that there is a state q and transitions $(p, q) \in Y$ and $(q, r) \in Z$. A transition relation Y *right-commutes* with a transition relation Z if $Y \circ Z \subseteq Z \circ Y$, and Y *left-commutes* with Z if $Z \circ Y \subseteq Y \circ Z$.

THEOREM 1 (REDUCTION). *For all i, let $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ be sets of states, and $\rightharpoonup_i$ be a transition relation. Suppose for all i,*

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

and for all $j \neq i$,

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

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

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

PROOF. See appendix. $\square$

Let Γ be a fixed type environment. We define the atomicity $\alpha(e)$ of an expression e to be a if $\Gamma \vdash e : a$. An examination of the type rules shows that α is a well-defined partial function. Recall that our type system checks that an atomic block consists of a sequence of left movers followed by an atomic action followed by a sequence of right movers. For each thread i, we begin by classifying each well-typed state according to whether thread i is (1) not currently executing an atomic block, (2) executing the "right-mover" part of some atomic block (before the atomic action), or (3) executing the "left-mover" part of some atomic block (after the atomic action).

$$
\begin{aligned}
WT &= \{\langle H, M, T\rangle \mid \Gamma \vdash \langle H, M, T\rangle\} \\
N_i &= WT \cap \{\langle H, M, T\rangle \mid |T| < i \ \vee\ T_i \not\equiv E[\texttt{in-atomic}\ e]\} \\
W_i &= WT \cap \{\langle H, M, T\rangle \mid T_i \equiv \textbf{wrong}\} \\
R_i &= WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-atomic}\ e] \ \wedge\ \alpha(e) \not\sqsubseteq L\} \\
L_i &= WT \cap \{\langle H, M, T\rangle \mid T_i \equiv E[\texttt{in-atomic}\ e] \ \wedge\ \alpha(e) \sqsubseteq L\}
\end{aligned}
$$

By the following subject reduction theorem, every state reachable from an initial, well-typed state is well-typed.

THEOREM 2 (SUBJECT REDUCTION). *Let P be a program with correct call tags and let Γ a type environment. If $\Gamma \vdash P$, and $P \to P'$, then $\Gamma \vdash P'$.*

PROOF. By case analysis on the transition rule for $P \to P'$. $\square$

Using Theorem 1 and Theorem 2, we now have the necessary machinery to prove two fundamental correctness properties of our type system. First, we show that if program execution starts from a initial, well-typed state P and reaches a subsequent state Q, where no thread in P or Q is inside an `in-atomic` block, then Q can also be reached from P according to the coarser transition relation $\twoheadrightarrow$ where each atomic block is executed atomically, and is not interleaved with steps from other threads.

The second property of our type system allows us to check program assertions at the coarse level and conclude that the assertions will hold at the concrete level as well. We call a program state *bad* if a thread (having misapplied a primitive) is **wrong** in that state. We show that if program execution starts from a well-typed state P and reaches a bad state, then there is an execution from P according to the coarser transition relation $\twoheadrightarrow$ that ends in a bad state. The proof of this property requires that once an atomic block has *committed*, *i.e.*, reached a stage where all subsequent operations are left movers, then it must be able to terminate. A program P is *nonblocking* if whenever $P \to^* Q$ and $L_i(Q)$, then $Q \to_i^* Q'$ such that $\neg L_i(Q')$.

THEOREM 3 (CORRECTNESS). *Let P be a program with correct conflict and call tags and let Γ be a valid type environment such that $\Gamma \vdash P$. Suppose $\forall i.\ N_i(P)$ and $P \to^* Q$. Then the following statements are true.*

1. *If $\forall i.\ N_i(Q)$, then $P \twoheadrightarrow^* Q$.*
2. *If P is nonblocking and $\exists i.\ W_i(Q)$, then $P \twoheadrightarrow^* Q'$ and $\exists i.\ W_i(Q')$.*

PROOF SKETCH. Using Theorem 2, we show in the appendix that for all thread indices i,

1. R_i and L_i are disjoint,
2. $(L_i / \to_i \backslash R_i)$ is false,

and for all thread indices $j \neq i$,

3. $\to_i$ and $\to_j$ are disjoint,
4. $(\to_i \backslash R_i)$ right-commutes with $\to_j$,
5. $(L_i / \to_i)$ left-commutes with $\to_j$,
6. if $P \to_i Q$, then $R_j(P) \Leftrightarrow R_j(Q)$, $L_j(P) \Leftrightarrow L_j(Q)$, and $W_j(P) \Leftrightarrow W_j(Q)$.

Suppose P and Q are states such that $\forall i.\ N_i(P)$ and $P \to^* Q$. We use Theorem 1 by substituting the set W_i for $\mathcal{W}_i$, the set R_i for $\mathcal{R}_i$, the set L_i for $\mathcal{L}_i$, the relation $\to_i$ for $\rightharpoonup_i$, the relation $\twoheadrightarrow$ for $\rightharpoonup\!\!\rightharpoonup$, the state P for p, and the state Q for q.

1. Since $\forall i.\ N_i(Q)$, we get from the first part of Theorem 1 that $P \twoheadrightarrow^* Q$.

2. We show that for any state S, if $\exists j.\ W_j(S)$ then $S \rightarrow^* S'$ such that $\exists j.\ W_j(S')$ and $\forall i.\ \neg L_i(S')$. For any state S, let $l(S) = \{i \mid L_i(S)\}$. We do the proof by induction on $|l(S)|$. If $|l(S)| = 0$, the hypothesis is trivially true. If $|l(S)| > 0$, pick $i \in l(S)$. Since P is nonblocking, we have $S \rightarrow_i^* S''$ such that $\neg L_i(S'')$. We also have that $j \in l(S)$ iff $j \in l(S'')$ for all $j \neq i$. Therefore $l(S'') = l(S) \setminus \{i\}$. Suppose $W_j(S)$ for some thread j. Since $L_i(S)$, we have $i \neq j$ and therefore $W_j(S'')$ as well. Thus, we also get that $\exists j.\ W_j(S'')$. By the induction hypothesis, there is S' such that $S'' \rightarrow^* S'$, $\exists j.\ W_j(S')$, and $\forall i.\ \neg L_i(S')$. Therefore, we have $S \rightarrow^* S'$.

   Since $\exists j. W_j(Q)$, we use the result proved in the previous paragraph to conclude that $Q \rightarrow^* Q''$, $\exists j.\ W_j(Q'')$, and $\forall i.\ \neg L_i(Q'')$. We now apply the second part of Theorem 1 to get a state Q' such that $\exists j.\ W_j(Q')$ and $P \twoheadrightarrow^* Q'$. $\square$

5. AN APPLICATION

To illustrate the benefits of our type system, we now consider its application to `java.util.Vector` (from JDK 1.1), a widely-used Java library class. Throughout this section we use Java syntax; extending our type system from CAT to Java is the topic of a related paper [12].

The methods in `Vector` are all intended to be atomic. The atomicity of a method is typically ensured by acquiring the lock of the object on method entry and releasing it on exit. Java provides a convenient way to perform these lock operations by adding the `synchronized` modifier to the method definition. In older versions of Java, the `synchronized` modifier was retained in the interface documentation produced by `javadoc`. However, `synchronized` is merely an implementation detail; in fact, it is not even necessary for achieving atomicity, as we will show later in this section. Our type system provides the keyword `atomic` which allows the atomicity of a method to be documented independent of the particular technique used to achieve atomicity.

In addition to better documenting the interface, our type system catches defects where a method is intended to be atomic, but may not be. For example, the following method from `Vector` includes an unsynchronized access to the field `elementCount`, which may later result in an exception inside the two-argument version of `lastIndexOf`:

```
public final int lastIndexOf(Object elem) {
    return lastIndexOf(elem_ε, elementCount_•-1);
}
```

Our earlier race condition checker [10] detected this bug, but would pass the following fix, since it is technically free of race conditions, even though the same error may still occur:

```
public final int lastIndexOf(Object elem) {
    int c;
    synchronized (this) { c_ε = elementCount_ε; }
    return lastIndexOf(elem_ε, c_ε-1);
}
```

In contrast, the type system of this paper would reject this fix, since the method is not atomic, thus encouraging the programmer to produce the correct (and atomic) implementation:

```
synchronized public final int lastIndexOf(Object elem) {
    return lastIndexOf(elem_ε, elementCount_ε-1);
}
```

Finally, we consider the implementation of the `size()` method:

```
public final int size() {
    return elementCount_•;
}
```

This method is unsynchronized, and may read `elementCount` when other threads are about to write it, as reflected in the conflict tag $\bullet$. Yet, since the method contains a single step (the read of `elementCount`), it is still atomic. The unsynchronized read of `elementCount` does complicate the atomicity argument for other methods, which we illustrate by considering an extension of `Vector` with the following method:

```
public final void synchronized removeLastElement() {
    elementCount_• = elementCount_ε-1;
}
```

Since `removeLastElement()` is synchronized, the read of the field `elementCount` has conflict tag ϵ, but the write to `elementCount` has a conflict, due to the unsynchronized (and hence possibly concurrent) read in `size()`. Yet, since the body of `removeLastElement()` contains a single atomic action (the write of `elementCount`) preceded by several operations that commute with steps of other threads, the method is atomic, a property that can be verified by our type system.

Thus, our type system has sufficient power to specify and verify fundamental properties concerning the behavior and correctness of `Vector`. In particular, our type language allows us to formally document a crucial property of the public interface to `Vector`, which is that many methods can be considered to execute atomically. In addition, our type system is capable of verifying the atomicity of these methods, despite subtle optimizations such as those in the `size()` method. Thus, the client programmer can rely on these atomicity properties, and safely reason about the behavior of a multithreaded program based on the assumption that each atomic method achieves its (informal) sequential specification in one step.

Our type system also revealed that certain `Vector` methods are not atomic. The unsynchronized method `capacity()` returns the length of the underlying array used to represent a `Vector`. Because of the two accessor methods `capacity()` and `size()`, a thread may observe two distinct changes in the state of a `Vector` whenever a concurrent thread calls `addElement(x)`. First, the result of `capacity()` may double, if the underlying array is full and needs to be replaced with a larger array. Second, the result of `size()` increases by one, reflecting the additional element in the `Vector`. Thus, because the synchronization discipline of `Vector` is optimized for performance, it does not provide the atomicity guarantee for `addElement` that would be desirable. We note that `addElement` is atomic provided the client code does not call `capacity()`; extending our current type system to verify such *conditional* atomicity properties is a topic for future work.

6. RELATED WORK

Lipton [18] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [8], Back [2], and Lamport and Schneider [17] extended this work to allow proofs of general safety properties. Cohen and Lamport [6] extended reduction to allow proofs of liveness properties. Misra [19] has proposed a reduction theorem for programs built with monitors [15] communicating via procedure calls. Bruening [4] and Stoller [22] have used reduction to improve the efficiency of model checking. Recently, we used reduction for checking concise functional specifications of interfaces in multithreaded programs [13].

A number of tools have been developed for detecting race conditions, both statically and dynamically. The Race Condition Checker [10] uses a type system to catch race conditions in Java programs. This approach has been extended [3] and adapted to other languages [14]. Other static race detection tools include Warlock [21], for ANSI C programs, and ESC/Java [11], which catches a variety of software defects in addition to race conditions. Vault [7] is a system designed to check resource management protocols, and lock-based synchronization can be considered to be such a protocol. Aiken and Gay [1] also investigate static race detection, in the context of SPMD programs. Eraser [20] detects race conditions and deadlocks dynamically, rather than statically. The Eraser algorithm has been extended to object-oriented languages [23] and for improved precision and performance [5].

Thus, reduction have already been studied in depth, as have type systems for preventing race conditions. The goal of this paper is to combine these existing techniques in a type system that provides an effective means of checking atomicity properties of multithreaded programs.

7. CONCLUSION

Reasoning about the correctness of multithreaded programs, either formally or informally, is difficult, due to the potential for subtle interactions between threads. However, the knowledge that in certain atomic code fragments such interactions do not occur significantly simplifies such correctness arguments. Programmers often intend that certain functions should be atomic, and document this intention by stating that these functions are "synchronized" or "thread-safe". However, programmers currently have little support for formally documenting or verifying these atomicity properties.

This paper shows that such atomicity properties can be specified and verified in a natural manner using an extended type system. Our type system guarantees that, in any well-typed program, each atomic block can be safely considered to execute in one step. This guarantee enables the programmer to safely reason about the program's behavior in a significantly simpler manner, at a higher level of granularity.

For sequential languages, standard type systems provide a means for expressing and checking fundamental correctness properties. We hope that type systems such as ours will play a similar role for reasoning about atomicity, a fundamental correctness property of multithreaded programs.

Acknowledgments

We thank Martin Abadi and the anonymous reviewers for valuable feedback that helped us to improve the paper.

8. REFERENCES

[1] A. Aiken and D. Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.

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

[3] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69, 2001.

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

[5] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[6] E. Cohen and L. Lamport. Reduction in TLA. In *International Conference on Concurrency Theory*, pages 317–331, 1998.

[7] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[8] T. Doeppner, Jr. Parallel program correctness through refinement. In *Proceedings of the 4th Symposium on Principles of Programming Languages*, pages 155–169, 1977.

[9] C. Flanagan and M. Abadi. Types for safe locking. In *Proceedings of European Symposium on Programming*, pages 91–108, 1999.

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

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

[12] C. Flanagan and S. Qadeer. Types for atomic interfaces. Submitted for publication, 2002.

[13] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. Submitted for publication, 2002.

[14] D. Grossman. Type-safe multithreading in Cyclone. In *Proceedings of the Workshop on Types in Language Design and Implementation*, 2003.

[15] C. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[16] T. Jim, G. Morrisett, D. Grossman, M. Hicks,

J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Technical Conference Proceedings*, pages 275–288, 2002.

[17] L. Lamport and F. Schneider. Pretending atomicity. Research Report 44, DEC Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, USA, 1989.

[18] R. Lipton. Reduction: A method of proving properties of parallel programs. In *Communications of the ACM*, volume 18:12, pages 717–721, 1975.

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

[20] S. Savage, M. Burrows, C. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

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

[22] S. Stoller. Model-checking multi-threaded distributed Java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, pages 224–244. Springer-Verlag, 2000.

[23] C. von Praun and T. Gross. Object-race detection. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 78–82, 2001.

APPENDIX

A. PROOFS

Theorem 1 *For all i, let $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ be sets of states, and $\rightharpoonup_i$ be a transition relation. Suppose for all i,*

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

and for all $j \neq i$,

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

Let $\mathcal{N}_i = \neg(\mathcal{R}_i \vee \mathcal{L}_i)$, $\mathcal{N} = \forall i.\ \mathcal{N}_i$, $\mathcal{W} = \exists i.\ \mathcal{W}_i$, ${\rightharpoonup} = \exists i.\ {\rightharpoonup_i}$, and ${\rightharpoonup\mkern-14mu\rightharpoonup} = \exists i.\ (\forall j \neq i.\ \mathcal{N}_j) / {\rightharpoonup_i}$. Suppose $p \in \mathcal{N}$ and $p \rightharpoonup^* q$. Then the following statements are true.

1. *If $q \in \mathcal{N}$, then $p \rightharpoonup\mkern-14mu\rightharpoonup^* q$.*
2. *If $q \in \mathcal{W}$ and $\forall i.\ q \notin \mathcal{L}_i$, then $p \rightharpoonup\mkern-14mu\rightharpoonup^* q'$ and $q' \in \mathcal{W}$.*

Proof. We will prove the theorem by induction. Our induction hypothesis is:

> If $p \in \mathcal{N}$ and $p \rightharpoonup^* q$, then there exist $k, l \geq 0$ and a transition sequence
>
> $$\begin{aligned} p &= p_1 \rightharpoonup^+_{t(1)} p_2 \rightharpoonup^+_{t(2)} p_3 \cdots p_k \rightharpoonup^+_{t(k)} p_{k+1} \\ &= q_1 \rightharpoonup^+_{u(1)} q_2 \rightharpoonup^+_{u(2)} q_3 \cdots q_l \rightharpoonup^+_{u(l)} q_{l+1} = q \end{aligned}$$
>
> with the following properties:
>
> - for all $1 \leq j \leq k$, if $p_j = p_{j,1} \rightharpoonup_{t(j)} \cdots \rightharpoonup_{t(j)} p_{j,x} = p_{j+1}$, then
>   1. $p_{j,1} \in \mathcal{N}_{t(j)}$,
>   2. $p_{j,2}, \ldots, p_{j,x-1} \in \mathcal{R}_{t(j)} \vee \mathcal{L}_{t(j)}$, and
>   3. $p_{j,x} \in \mathcal{L}_{t(j)} \vee \mathcal{N}_{t(j)}$.
> - for all $1 \leq j \leq l$, if $q_j = q_{j,1} \rightharpoonup_{u(j)} \cdots \rightharpoonup_{u(j)} q_{j,x} = q_{j+1}$, then
>   1. $q_{j,1} \in \mathcal{N}_{u(j)}$, and
>   2. $q_{j,2}, \ldots, q_{j,x} \in \mathcal{R}_{u(j)}$.

We first show that this induction hypothesis implies the theorem. Suppose $p \in \mathcal{N}$ and $p \rightharpoonup^* q$.

1. Suppose $q \in \mathcal{N}$. Then $l = 0$ and $p_{k+1} = q_1 = q$. We show by induction that $p_j \in \mathcal{N}$ for all $j \leq k+1$. We prove that if $j \leq k$ and $p_{j+1} \in \mathcal{N}$, then $p_j \in \mathcal{N}$. Suppose $p_{j+1} \in \mathcal{N}$. Since $p_j \rightharpoonup^+_{t(j)} p_{j+1}$, we have $p_j \in \mathcal{N}_i$ for all $i \neq t(j)$. We also know that $p_j \in \mathcal{N}_{t(j)}$. Therefore $p_j \in \mathcal{N}$. It is now easy to show that every transition in $p = p_1 \rightharpoonup^+_{t(1)} p_2 \rightharpoonup^+_{t(2)} p_3 \cdots p_k \rightharpoonup^+_{t(k)} p_{k+1} = q$ satisfies $\rightharpoonup\mkern-14mu\rightharpoonup$. Therefore $p \rightharpoonup\mkern-14mu\rightharpoonup^* q$.

2. Suppose $q \in \mathcal{W}$ and $\forall i.\ q \notin \mathcal{L}_i$. We show by induction $q_j \in \mathcal{W}$ and $\forall i.\ q_j \notin \mathcal{L}_i$ for all $1 \leq j \leq l+1$. For the base case, we have $q_{l+1} = q$. For the inductive case, suppose $q_{j+1} \in \mathcal{W}$ and $\forall i.\ q_{j+1} \notin \mathcal{L}_i$. Then $q_j \notin \mathcal{L}_i$ for all $i \neq u(j)$ and $q_j \in \mathcal{N}_{u(j)}$. Therefore $\forall i.\ q_j \notin \mathcal{L}_i$. Moreover, since $q_{j+1} \in \mathcal{R}_{u(j)}$, $q_{j+1} \in \mathcal{W}_i$ for some $i \neq uj$. Therefore $q_j \in \mathcal{W}_i \subseteq \mathcal{W}$. Therefore, we get $p_{k+1} = q_1 \in \mathcal{W}$ and $\forall i.\ p_{k+1} \notin \mathcal{L}_i$.

   We again use induction to show that $\forall i.\ p_j \notin \mathcal{L}_i$ for all $1 \leq j \leq k+1$. The induction proceeds exactly as the induction argument in the last paragraph. Finally, we use induction to show that $p_j \in \mathcal{N}$ for all $1 \leq j \leq k+1$. For the base case, we have $p_1 = p \in \mathcal{N}$. Suppose $p_j \in \mathcal{N}$. Then $p_{j+1} \in \mathcal{N}_i$ for all $i \neq t(j)$. From the induction hypothesis, we know that $p_{j+1} \in \mathcal{N}_{t(j)} \vee \mathcal{L}_{t(j)}$. We have also shown that $\forall i.\ p_{j+1} \notin \mathcal{L}_i$. Therefore $p_{j+1} \in \mathcal{N}_{t(j)}$ and we conclude that $p_{j+1} \in \mathcal{N}$. Therefore, we get $p_{k+1} \in \mathcal{N}$. Let $q' = p_{k+1}$. It is now easy to show that every transition in $p = p_1 \rightharpoonup^+_{t(1)} p_2 \rightharpoonup^+_{t(2)} p_3 \cdots p_k \rightharpoonup^+_{t(k)} p_{k+1} = q'$ satisfies $\rightharpoonup\mkern-14mu\rightharpoonup$. Therefore $p \rightharpoonup\mkern-14mu\rightharpoonup^* q'$ and we have already shown that $q' = p_{k+1} \in \mathcal{W}$.

Now we prove the induction hypothesis by induction over the length of the execution $p \rightharpoonup^* q$. The base case where $p = q$ is trivial as we can choose $k = l = 0$. Otherwise suppose $p \in \mathcal{N}$ and $p \rightharpoonup^* q \rightharpoonup_i q'$. Then the induction hypothesis holds for $p \rightharpoonup^* q$. We do a case analysis on q.

- $q \in \mathcal{N}_i$. We show the following two statements by mutual induction:
  - $q_j \in \mathcal{N}_i$ for all $1 \leq j \leq l+1$
  - $i \neq u(j)$ for all $1 \leq j \leq l$

  For the base case, we have $q_{l+1} = q \in \mathcal{N}_i$. There are two inductive cases. Suppose $q_{j+1} \in \mathcal{N}_i$. Since $q_{j+1} \in \mathcal{R}_{u(j)}$, we have $i \neq u(j)$. Therefore $q_j \in \mathcal{N}_i$. In particular, we get that $p_{k+1} = q_1 \in \mathcal{N}_i$.

  Since $u(1), \ldots, u(l)$ are all different from i, we commute all the actions performed by these threads to the right of the action by thread i to get the execution sequence

  $$\begin{aligned} &p = p_1 \rightharpoonup^+_{t(1)} p_2 \rightharpoonup^+_{t(2)} p_3 \cdots p_k \rightharpoonup^+_{t(k)} p_{k+1} \\ &\rightharpoonup_i q'_1 \rightharpoonup^+_{u(1)} q'_2 \rightharpoonup^+_{u(2)} q'_3 \cdots q'_l \rightharpoonup^+_{u(l)} q'_{l+1} = q' . \end{aligned}$$

- $q \in \mathcal{L}_i$. Again, we show by induction that $q_j \in \mathcal{N}_i$ for all $1 \le j \le l+1$ and $i \ne u(j)$ for all $1 \le j \le l$. The proof proceeds exactly as in the last case. We have that $p_{k+1} = q_1 \in \mathcal{N}_i$. There must be some j such that $t(j) = i$. Otherwise $p = p_1 \in \mathcal{N}_i$, which is a contradiction. Consider the greatest j such that $t(j) = l$. Then $t(j+1), \ldots, t(k)$ and $u(1), \ldots, u(l)$ are all different from i. We commute the action performed by thread i to the left of all actions performed by these threads to get the execution sequence

$$\begin{aligned} p = p_1 \rightharpoonup^+_{t(1)} p_2 \cdots p_j \rightharpoonup^+_{t(j)} \\ p'_{j+1} \rightharpoonup^+_{t(j+1)} p'_{j+2} \cdots p'_k \rightharpoonup^+_{t(k)} p'_{k+1} \\ = q'_1 \rightharpoonup^+_{u(1)} q'_2 \rightharpoonup^+_{u(2)} q'_3 \cdots q'_l \rightharpoonup^+_{u(l)} q'_{l+1} = q' \ . \end{aligned}$$

Since $q \in \mathcal{L}_i$, we have $q' \in \mathcal{L}_i \vee \mathcal{N}_i$. Therefore $q'_{j+1} \in \mathcal{L}_i \vee \mathcal{N}_i$.

- $q \in \mathcal{R}_i$. We first prove by contradiction that $u(1), \ldots, u(l)$ are all distinct from each other. Suppose $1 \le a, b \le l$ are such that $u(a) = u(b)$ and $u(j) \ne u(a)$ for all $a < j < b$. Then we know that $q_{a+1} \in \mathcal{R}_{u(a)}$. Therefore $q_b \in \mathcal{R}_{u(a)}$ which is a contradiction since $q_b \in \mathcal{N}_{u(b)}$.

  We now prove by contradiction that $i = u(j)$ for some j such that $1 \le j \le l$. If not, then $p_{k+1} = q_1 \in \mathcal{R}_i$. We now perform a case analysis.

  1. Suppose there is no j' such that $t(j') = i$. Then $p = p_1 \in \mathcal{R}_i$ which is a contradiction since $p \in \mathcal{N}$.
  2. Suppose j' is the greatest such that $t(j') = i$. Then $p_{j'+1} \in \mathcal{R}_i$, which is a contradiction since $p_{j'+1} \in \mathcal{L}_{t(j')} \vee \mathcal{N}_{t(j')}$.

  Therefore $u(1), \ldots, u(j), u(j+1), \ldots, u(l)$ are all different from i. We first commute the actions performed by $u(j+1), \ldots, u(l)$ to the right of the action performed by i to get the execution sequence

$$\begin{aligned} p = p_1 \rightharpoonup^+_{t(1)} p_2 \rightharpoonup^+_{t(2)} p_3 \cdots p_k \rightharpoonup^+_{t(k)} p_{k+1} \\ = q_1 \rightharpoonup^+_{u(1)} q_2 \cdots q_j \rightharpoonup^+_{u(j)} q'_{j+1} \cdots q'_l \rightharpoonup^+_{u(l)} q'_{l+1} = q' \ . \end{aligned}$$

  If $q'_{j+1} \in \mathcal{R}_{u(j)}$ then we are done. Otherwise, we commute actions performed by threads $u(1), \ldots, u(j-1)$ to the right of all actions performed by thread $u(j)$ to get the execution sequence

$$\begin{aligned} p = p_1 \rightharpoonup^+_{t(1)} p_2 \rightharpoonup^+_{t(2)} p_3 \cdots p_k \rightharpoonup^+_{t(k)} p_{k+1} \\ = q'_1 \rightharpoonup^+_{u'(1)} q'_2 \cdots q'_j \rightharpoonup^+_{u'(j)} q'_{j+1} \cdots q'_l \rightharpoonup^+_{u'(l)} q'_{l+1} = q' \end{aligned}$$

  where $u'(1) = u(j), u'(2) = u(1), \ldots, u'(j) = u(j-1), u'(j+1) = u(j+1), \ldots, u'(l) = u(l)$.

Since either $q \in \mathcal{N}_i$ or $q \in \mathcal{R}_i$ or $q \in \mathcal{L}_i$, our case analysis and the proof is complete. □

Theorem 3 *Let P be a program with correct conflict and call tags and let Γ be a valid type environment such that $\Gamma \vdash P$. Suppose $\forall i.\ N_i(P)$ and $P \to^* Q$. Then the following statements are true.*

1. *If $\forall i.\ N_i(Q)$, then $P \twoheadrightarrow^* Q$.*
2. *If P is nonblocking and $\exists i.\ W_i(Q)$, then $P \twoheadrightarrow^* Q'$ and $\exists i.\ W_i(Q')$.*

PROOF. We show that for all thread indices i,

1. R_i, L_i, and W_i are pairwise disjoint,
2. $(L_i / \!\to_i \backslash R_i)$ is false,

and for all thread indices $j \ne i$,

3. $\to_i$ and $\to_j$ are disjoint,
4. $(\to_i \backslash R_i)$ right-commutes with $\to_j$,
5. $(L_i / \!\to_i)$ left-commutes with $\to_j$,
6. if $P \to_i Q$, then $R_j(P) \Leftrightarrow R_j(Q)$, $L_j(P) \Leftrightarrow L_j(Q)$, and $W_j(P) \Leftrightarrow W_j(Q)$.

The proofs for the seven statements given above follow:

1. By the definition of R_i, L_i, and W_i.
2. Suppose $P_1 \to_i P_2$ where $P_1 \in L_i$ and $P_2 \in R_i$. The proof follows by a straightforward case analysis on the transition rule for $P_1 \to_i P_2$.
3. The transition relation $\to_i$ changes the expression representing thread i but leaves the expressions of all other threads unchanged. Therefore $\to_i$ and $\to_j$ are disjoint for all $i \ne j$.
4. Suppose $P_1 \to_i P_2 \to_j P_3$ where $i \ne j$ and $P_2 \in R_i$. We proceed by case analysis on the rule for $P_1 \to_i P_2$.
   (a) [Variable read] We have $P_1 = H, M, T$ such that $T_i \equiv E[x_r]$, and $P_2 = H, M, T[i \mapsto E[H(x)]]$. Since $P_2 \in R_i$, we have $E \equiv E'[\texttt{in-atomic}\ E'']$ and $\alpha(E''[H(x)]) \not\sqsubseteq L$. Therefore $P_1 \in R_i \vee L_i$ and from part (5) of this theorem we get $P_1 \in R_i$. From the subject reduction theorem, we know that $\alpha(E''[x_r]) \sqsubseteq A$ and $\alpha(E''[H(x)]) \sqsubseteq A$. Therefore, we get $\alpha(E''[x_r]) \in \{R, A\}$ and $\alpha(E''[H(x)]) \in \{R, A\}$. From Lemma 1, we have $\alpha(E''[x_r]) = \alpha(x_r); \alpha(E''[H(x)])$. The only way to satisfy this equation is to have $\alpha(x_r) = B$. Therefore $r = \epsilon$. Since the conflict tag is correct and the other thread j is not about to read or write x, so $\to_i$ right-commutes with $\to_j$.
   (b) [Variable write] Similar.
   (c) [Primitive application] In this case $P_1 = H, M, T$ where $T_i \equiv E[\texttt{in-atomic}\ E'[p(\overline{v})]]$. Again, by similar reasoning, $\alpha(p(\overline{v})) \sqsubseteq R$, and hence $\Gamma(p) \sqsubseteq R$. Hence, the application of p right-commutes with any primitive application of thread j (which are the only steps that modify the synchronization heap), so $\to_i$ right-commutes with $\to_j$.
   (d) [Fork] In this case $P_1 = H, M, T$ such that $T_i \equiv E[\texttt{in-atomic}\ E'[\texttt{fork}\ e]]$ and $P_2 = H, M, T[i \mapsto E[\texttt{in-atomic}\ E'[0]]$. From the subject reduction theorem, we know that $\alpha(E'[\texttt{fork}\ e]) \sqsubseteq A$. From Lemma 1, we know that
   $$\alpha(E'[\texttt{fork}\ e]) = \alpha(\texttt{fork}\ e); \alpha(E'[0]) = A; \alpha(E'[0]).$$
   Therefore, we must have $\alpha(E'[0]) \sqsubseteq L$. Therefore $P_2 \in L_i$ and we have a contradiction.
   (e) [Other rules] If $P_1 \to_i P_2$ by any of while, if, let, call, atomic, or in-atomic, then these rules do not modify the heap or the synchronization heap, and so $\to_i$ right-commutes with $\to_j$.
5. Suppose $P_1 \to_j P_2 \to_i P_3$ where $i \ne j$ and $P_2 \in L_i$. A straightforward case analysis on the rule for $P_2 \to_i P_3$ will yield the desired result.
   (a) [Variable read] In this case $P_2 = H, M, T$ such that $T_i \equiv E[x_r]$ and $P_3 = H, M, T[i \mapsto E[H(x)]$. Since $P_2 \in L_i$, we have $E \equiv E'[\texttt{in-atomic}\ E'']$,

and $\alpha(E''[x_r]) \sqsubseteq L$. We also have $P_3 \in L_i$ and $\alpha(E''[H(x)]) \sqsubseteq L$. By Lemma 1, we have

$$\alpha(E''[x_r]) = \alpha(x_r); \alpha(E''[H(x)]).$$

Therefore $\alpha(x_r) \sqsubseteq L$, and so $r = \epsilon$. Since the conflict tags are correct, the other thread j is not about to read or write x, so $\rightarrow_i$ left-commutes with $\rightarrow_j$.

(b) [Variable write] Similar.

(c) [Primitive application] In this case $P_2 = H, M, T$ where $T_i \equiv E[\texttt{in-atomic}\ E'[p(\overline{v})]]$. Again, by similar reasoning, $\alpha(p(\overline{v})) \sqsubseteq L$, and hence $\Gamma(p) \sqsubseteq L$. Hence, the application of p left-commutes with any primitive application of thread j (which are the only steps that modify the synchronization heap), so $\rightarrow_i$ left-commutes with $\rightarrow_j$.

(d) [Fork] In this case $P_2 = H, M, T$ such that $T_i \equiv E[\texttt{in-atomic}\ E'[\texttt{fork}\ e]]$ and $P_3 = H, M, T[i \mapsto E[\texttt{in-atomic}\ E'[0]]$. From the subject reduction theorem, we know that $\alpha(E'[\texttt{fork}\ e]) \sqsubseteq A$. From Lemma 1, we know that

$$\alpha(E'[\texttt{fork}\ e]) = \alpha(\texttt{fork}\ e); \alpha(E'[0]) = A; \alpha(E'[0]).$$

Therefore $\alpha(E'[\texttt{fork}\ e]) \not\sqsubseteq L$. Therefore $P_2 \notin L_i$ and we have a contradiction.

(e) [Other rules] If $P_2 \rightarrow_i P_3$ by any of while, if, let, call, atomic, or in-atomic, then these rules do not modify the heap or the synchronization heap, and so $\rightarrow_i$ left-commutes with $\rightarrow_j$.

6. Suppose $P \rightarrow_i Q$. If this step is not a fork step, then threads other than i do not change in going from P to Q. Therefore $R_j(P)$ iff $R_j(Q)$, $L_j(P)$ iff $L_j(Q)$, and $W_j(P)$ iff $W_j(Q)$.

   If this step forks a new thread j, then $P = H, M, T$ and $T_i = E[\texttt{fork}\ e]$ where e does not contain `in-atomic`, since it does not occur in an evaluation context position. Let j be the newly forked thread. Then $N_j(P)$ (by the definition of N_j) and $N_j(Q)$. Furthermore, since threads other than i or j do not change in going from P to Q, we have that $R_j(P)$ iff $R_j(Q)$, $L_j(P)$ iff $L_j(Q)$, and $W_j(P)$ iff $W_j(Q)$.

The remainder of the proof follows the proof sketch of this theorem in Section 4. □

LEMMA 1. *For all evaluation contexts E and expressions e, if $\alpha(E[e])$ is defined, then*

1. *$\alpha(e)$ is defined, and*
2. *for all values v such that $\alpha(v)$ is defined, the atomicity $\alpha(E[v])$ is defined and $\alpha(E[e]) = \alpha(e); \alpha(E[v])$.*

PROOF. By induction on the derivation $\Gamma \vdash E[e] : a$. □