A Type and Effect System for Atomicity

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

Shaz Qadeer
Microsoft Research
One Microsoft Way
Redmond, WA 98052

ABSTRACT

Ensuring the correctness of multithreaded programs is difficult, due to the potential for unexpected and nondeterministic interactions between threads. Previous work 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, verifying the absence of such simultaneous-access 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 *atomicity*. Atomic methods can be assumed to execute serially, without interleaved steps of other threads. Thus, atomic methods are amenable to sequential reasoning techniques, which significantly simplifies both formal and informal reasoning about program correctness.

This paper presents a type system for specifying and verifying the atomicity of methods in multithreaded Java programs. The atomic type system is a synthesis of Lipton's theory of reduction and type systems for race detection.

We have implemented this atomic type system for Java and used it to check a variety of standard Java library classes. The type checker uncovered subtle atomicity violations in classes such as `java.lang.String` and `java.lang.StringBuffer` that cause crashes under certain thread interleavings.

Categories and Subject Descriptors

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

General terms

Reliability, Security, Languages, Verification

Keywords

Multithreading, race conditions, static checking, atomicity

PLDI'03, June 9–11, 2003, San Diego, California, 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 [18, 17] and other static [19] and dynamic [35] 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, verifying the absence of such simultaneous-access race conditions is insufficient to ensure the absence of errors due to unexpected thread interactions. To illustrate this idea, consider the following method, in which the shared variable `x` is protected by the lock `l`:

```
int x; // shared var guarded by lock l
void m() {
    int t;
    synchronized (l) { t = x; }
    t++;
    synchronized (l) { x = t; }
}
```

This method does not suffer from race conditions, a property that can be easily verified with existing tools such as `rccjava` [18]. However, the method may still not have the expected effect of simply incrementing `x` that it would in a sequential setting. In particular, if n calls to the method are made concurrently, the overall effect may be to increment `x` by any number between 1 and n.

We propose that a stronger non-interference property is required, namely *atomicity*. If a method is atomic, then any interaction between that method and steps of other threads is guaranteed to be benign, in the sense that these interactions do not change the program's overall behavior. Thus, having verified the atomicity of a method, we can subsequently specify and verify that method using standard sequential reasoning techniques, even though the scheduler is free to interleave threads at instruction-level granularity.

We believe that a fundamental correctness property common to many interfaces in multithreaded programs is that the methods of these interfaces are intended to be atomic. Programmers have precise expectations regarding the methods that should be atomic, as illustrated by the documentation for the class `java.lang.StringBuffer` in JDK1.4:

> "String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order

that is consistent with the order of the method calls made by each of the individual threads involved."

A methodology that supports concise specification and efficient checking of such expectations of atomicity would be invaluable to both the implementor and the client of such an interface. Unfortunately, existing checking methodologies are unable to either formally specify or verify such expectations. Although the notions of atomicity and race-freedom are closely related, race-freedom is not sufficient to prove atomicity, as shown in the example above; it is also not necessary, as we show in Section 1.1.

In this paper, we present a type system for specifying and checking atomicity properties of methods in multithreaded programs. Methods can be annotated with the keyword `atomic`. The type system checks that for any (arbitrarily-interleaved) execution, there is a corresponding serial execution with equivalent behavior in which the instructions of the atomic method are not interleaved with instructions from other threads.

To evaluate the utility of this atomic type system, we implemented an atomic type checker for the full Java programming language [23] and tested it on a variety of widely-used Java library classes. We discovered a number of previously unknown defects, including subtle atomicity violations in `java.lang.String` and `java.lang.StringBuffer` that cause crashes under certain interleavings. These errors are not due to race conditions, and would be missed by a race condition checker.

1.1 The need for atomicity

As an illustration of the problems that arise in multithreaded programming, consider the program shown below. This program allocates a new bank account, and makes two deposits into the account in parallel. This program is written in the language ConcurrentJava, which is essentially a multithreaded subset of Java extended with `let` and `fork` constructs.

```
class Account {
  int balance = 0;

  int deposit1(int x) {
    this.balance = this.balance + x;
  }
}

let Account a = new Account in {
  fork {a.deposit1(10)};
  fork {a.deposit1(10)}
}
```

The program may exhibit unexpected behavior. In particular, if the two calls to `deposit1` are interleaved, the final value of `balance` may reflect only one of the two deposits made to the account, which is clearly not the intended behavior of the program. That is, the program contains a race condition: two threads attempt to manipulate the field `deposit1` simultaneously, with incorrect results.

We can fix this error by protecting the field `balance` by the implicit lock of the account object and only accessing or updating `balance` when that lock is held:

```
  int deposit2(int x) {
    synchronized (this) {
      this.balance = this.balance + x;
    }
  }
```

The race condition checker `rccjava` [18] can detect the race condition in the original bank account implementation and can verify that the modified implementation is race-free.

In general, however, the absence of race conditions does not imply the absence of errors due to thread interactions. To illustrate this point, we extend the account implementation with two additional methods—`readBalance1` to return the current account balance and `withdraw1` to take money out of the account.

```
int readBalance1() {          void withdraw1(int amt) {
  int t;                        int b = readBalance1();
  synchronize (this) {          synchronize (this) {
    t = balance;                  balance = b - amt;
  };                            }
  return t;                   }
}
```

Even though there are no races in either method, the method `withdraw1` 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 { withdraw1(10); };          // Thread 1
  fork { deposit2(10); };           // Thread 2
```

We would expect an account balance of 10 after the program terminates, but certain executions violate this expectation. Suppose the scheduler first performs the call to `readBalance1` in `Thread 1` which returns 10. The scheduler then switches to `Thread 2` and completes the execution of `deposit2` ending with `balance` = 20. Finally, the scheduler switches back to `Thread 1` and completes the execution setting `balance` to 0. Thus, even though there are no races in this program, unexpected interaction between the threads can lead to incorrect behavior.

The account interface provided by the methods `deposit2`, `readBalance1`, and `withdraw1` is intended to be atomic, a fundamental 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 such atomicity properties, thus catching errors such as the one in `withdraw1` above.

Having caught the error with our type system, we fix the problem in `withdraw1` as shown below. At the same time, since `readBalance1` performs a single read of a single-word field, its `synchronized` statement is redundant, and we remove it.

```
int readBalance2() {       void withdraw2(int amt) {
    return balance;          synchronize (this) {
}                              balance = balance - amt;
                             }
                           }
```

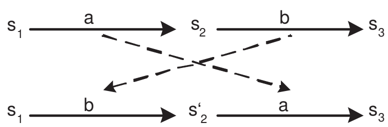
A race condition checker will report a race in `readBalance2`. However, this warning is a false alarm as the race condition is benign. Despite the presence of benign race conditions, our type system can still verify that the methods `deposit2`, `readBalance2`, and `withdraw2` are atomic.

1.2 An overview of 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 an overview of our type system for checking atomicity. We allow any method to be annotated with keyword `atomic`, and use the theory of right and left movers, first proposed by Lipton [28], to prove the correctness of `atomic` annotations.

An action a is a *right mover* if for any execution where the action a performed by one thread is immediately followed by an action b of a different thread, the actions a and b can be swapped without changing the resulting state, as shown below. Similarly, an action b is a *left mover* if whenever b immediately follows an action a of a different thread, the actions a and b can be swapped, again without changing the resulting state.

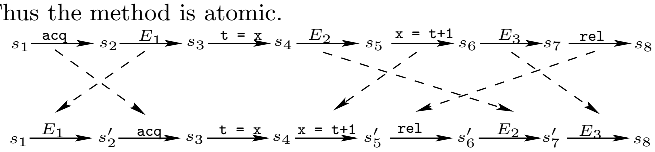


The type system classifies actions as left or right movers as follows. Consider an execution in which an acquire operation a on some lock is immediately followed by an action b of a second thread. Since the lock is already held by the first thread, the action b neither acquires nor releases the lock, and hence the acquire operation can be moved to the right of b without changing the resulting state. Thus the type system classifies each lock acquire operation as a right mover.

Similarly, consider an action a of one thread that is immediately followed by a lock release operation b by a second thread. During a, the second thread holds the lock, and a can neither acquire nor release the lock. Hence the lock release operation can be moved to the left of a without changing the resulting state, and thus the type system classifies lock release operations as left movers.

Finally, consider an access (read or write) to a shared variable declared with the guard annotation `guarded_by` l. This annotation states that the lock denoted by expression l must be held when the variable is accessed. Since our type system enfores this access restriction, no two threads may access the field at the same time, and therefore every access to this field is both a right mover and a left mover.

To illustrate how the theory of movers enables us to verify atomicity, consider a method that (1) acquires a lock (the operation `acq` in the first execution trace in the diagram below), (2) reads a variable `x` protected by that lock into a local variable `t` (`t=x`), (3) updates that variable (`x=t+1`), and then (4) releases the lock (`rel`). Suppose that the actions of this method are interleaved with arbitrary actions E_1, E_2, E_3 of other threads. Because the acquire operation is a right mover and the write and release operations are left movers, there exists an equivalent serial execution where the operations of the method are not interleaved with operations of other threads, as illustrated by the following diagram. Thus the method is atomic.



More generally, suppose a method contains a sequence of right movers followed by a single atomic action followed by a sequence of left movers. Then an execution where this method has been fully executed can be *reduced* to another execution with the same resulting state where the method is executed serially without any interleaved actions by other threads. Therefore, an `atomic` annotation on such a method is valid.

The remainder of the paper describes our type system in more detail. The following section presents a multithreaded subset of Java, and Section 3 formalizes the atomic type system for this Java subset. Section 4 describes the implementation of the atomic type system for the Java programming language [23]; the application of this type checker to a number of widely-used classes; and reports on atomicity violations caught using this checker. Section 5 describes related work, and we conclude with Section 6. Appendix A contains the full set of type rules for our system.

2. CONCURRENT JAVA

This section presents ConcurrentJava [18], a multithreaded subset of Java [23] that we use to formalize our type system. ConcurrentJava supports multithreaded programs by including the operation `fork` e which spawns a new thread for the evaluation of e. This evaluation is performed only for its effect; the result of e is never used. Locks are provided for thread synchronization. As in Java, each object has an associated lock that has two states, locked and unlocked, and is initially unlocked. The expression `synchronized` e_1 e_2 is evaluated in a manner similar to Java's `synchronized` statement: the subexpression e_1 is evaluated first, and should yield an object, whose lock is then acquired; the subexpression e_2 is then evaluated; and finally the lock is released. The result of e_2 is returned as the result of the synchronized expression. While evaluating e_2, the current thread is said to hold the lock. Any other thread that attempts to acquire the lock blocks until the lock is released. A forked thread does not inherit locks held by its parent thread.

The syntax of the `synchronized` and `fork` expressions and the rest of ConcurrentJava is shown in Figure 1. A program is a sequence of class definitions together with an initial expression. Each class definition associates a class name with a class body consisting of a super class, a sequence of field declarations, and a sequence of method declarations. A field declaration includes an initialization expression and an optional `final` modifier; if this modifier is present, then the field cannot be updated after initialization. We use "$[X]_{\text{opt}}$" in grammars to denote either "X" or the empty string. A method declaration consists of the method name, its return type, number and types of its arguments, and an expression for the method body. Types include class types, integers, and long integers. Class types include class names introduced by the program, as well as the predefined class `Object`, which serves as the root of the class hierarchy. Expressions include the typical operations for object allocation, field access and update, method invocation, variable binding and reference, conditionals, and loops, as well as the concurrency primitives. Variables are bound by `let`-expressions, formal parameter lists, and the special variable `this` is implicitly bound by a class declaration and is in scope within the body of that class.

$$
\begin{array}{rcll}
P & ::= & \mathit{defn}^*\ e & \text{(program)} \\
\mathit{defn} & ::= & \texttt{class}\ \mathit{cn}\ \mathit{body} & \text{(class decl)} \\
\mathit{body} & ::= & \texttt{extends}\ c & \\
 & & \quad \{\ \mathit{field}^*\ \mathit{meth}^* \} & \text{(class body)} \\
\mathit{field} & ::= & [\texttt{final}]_{\text{opt}}\ t\ \mathit{fd}\ \texttt{=}\ e & \text{(field decl)} \\
\mathit{meth} & ::= & t\ \mathit{mn}(\mathit{arg}^*)\ \{\ e\ \} & \text{(method decl)} \\
\mathit{arg} & ::= & t\ x & \text{(variable decl)} \\
s, t & ::= & c \mid \texttt{int} \mid \texttt{long} & \text{(type)} \\
c & ::= & \mathit{cn} \mid \texttt{Object} & \text{(class type)} \\
e & ::= & \texttt{new}\ c & \text{(allocate)} \\
 & \mid & x & \text{(variable)} \\
 & \mid & e.\mathit{fd} & \text{(field access)} \\
 & \mid & e.\mathit{fd}\ \texttt{=}\ e & \text{(field update)} \\
 & \mid & e.\mathit{mn}(e^*) & \text{(method call)} \\
 & \mid & \texttt{let}\ \mathit{arg}\ \texttt{=}\ e\ \texttt{in}\ e & \text{(variable binding)} \\
 & \mid & \texttt{while}\ e\ e & \text{(iteration)} \\
 & \mid & \texttt{if}\ e\ e\ e & \text{(conditional)} \\
 & \mid & \texttt{synchronized}\ e\ e & \text{(synchronization)} \\
 & \mid & \texttt{fork}\ e & \text{(fork)} \\
\mathit{cn} & \in & \text{class names} & \\
\mathit{fd} & \in & \text{field names} & \\
\mathit{mn} & \in & \text{method names} & \\
x, y & \in & \text{variable names} &
\end{array}
$$

Figure 1: ConcurrentJava.

We present example programs in an extended language with integer and boolean constants and operations, and the constant `null`. The sequential composition $e_1; e_2$ abbreviates `let` x `=` e_1 `in` e_2, where x does not occur free in e_2; the expression $e[x := e']$ denotes the capture-free substitution of e' for x in e. We sometimes enclose expressions in parentheses or braces for clarity and use `return` e to emphasize that the result of e is the return value of the current method.

3. TYPES FOR ATOMICITY

3.1 Basic Atomicities

Like conventional type systems, our type system assigns to each expression a type characterizing the value of that expression. In addition, our type system also assigns to each expression an *atomicity* characterizing the behavior [39] or effect of that expression. The set of atomicities includes the following *basic atomicities*:

- `const`: An expression is assigned the atomicity `const` if its evaluation does not depend on or change any mutable state. Hence the repeated evaluation of a `const` expression with a given environment always yields the same result.
- `mover`: An expression is assigned the atomicity `mover` if it both left and right commutes with operations of other threads. For example, an access to a field f declared as `guarded_by` l is a `mover` if the access is performed with the lock l held. Clearly, this access cannot happen concurrently with another access to f by a different thread if that thread also accesses f with the lock l held. Therefore, this access both left and right commutes with any concurrent operation by another thread. [1]
- `atomic`: An expression is assigned the atomicity `atomic` if it is a single atomic action or if it can be considered to execute without interleaved actions of other threads.
- `cmpd`: An expression is assigned the atomicity `cmpd` if none of the preceeding atomicities apply.
- `error`: An expression is assigned the atomicity `error` if it violates the locking discipline specified by the type annotations.

If the basic atomicity α reflects the behavior of an expression e, then the *iterative closure* α^* reflects the behavior of executing e an arbitrary number of times, and is defined as follows:

$$
\begin{array}{rcl}
\texttt{const}^* & = & \texttt{const} \\
\texttt{mover}^* & = & \texttt{mover} \\
\texttt{atomic}^* & = & \texttt{cmpd} \\
\texttt{cmpd}^* & = & \texttt{cmpd} \\
\texttt{error}^* & = & \texttt{error}
\end{array}
$$

Similarly, if basic atomicities α_1 and α_2 reflect the behavior of e_1 and e_2 respectively, then the *sequential composition* $\alpha_1; \alpha_2$ reflects the behavior of $e_1; e_2$, and is defined by the following table.

;	const	mover	atomic	cmpd	error
const	const	mover	atomic	cmpd	error
mover	mover	mover	atomic	cmpd	error
atomic	atomic	atomic	cmpd	cmpd	error
cmpd	cmpd	cmpd	cmpd	cmpd	error
error	error	error	error	error	error

Basic atomicities are ordered by the subatomicity relation:

$$\texttt{const} \sqsubset \texttt{mover} \sqsubset \texttt{atomic} \sqsubset \texttt{cmpd} \sqsubset \texttt{error}$$

Let $\sqcup$ denote the join operator based on this subatomicity ordering. If basic atomicities α_1 and α_2 reflect the behavior of e_1 and e_2 respectively, then the nondeterministic choice between executing either e_1 or e_2 has atomicity $\alpha_1 \sqcup \alpha_2$.

3.2 Conditional Atomicities

In some cases, the atomicity of an expression depends on the locks held by the thread evaluating that expression. For example, an access to a field declared as `guarded_by` l has atomicity `mover` if the lock l is held by the current thread, and has atomicity `error` otherwise. We assign such an access the *conditional atomicity*:

$$l\,?\,\texttt{mover} : \texttt{error}$$

A conditional atomicity $l\,?\,a : b$ is equivalent to atomicity a if the lock l is currently held, and is equivalent to atomicity b if the lock is not held. Conditional atomicities provide a more precise characterization of the behavior of synchronized statements and methods. We use $l\,?\,a$ to abbreviate $l\,?\,a : \texttt{error}$. The set of atomicities thus includes both the basic atomicities described above and conditional atomicities:

$$
\begin{array}{rcl}
a, b & ::= & \alpha \mid l\,?\,a : b \\
\alpha, \beta & ::= & \texttt{const} \mid \texttt{mover} \mid \texttt{atomic} \mid \texttt{cmpd} \mid \texttt{error} \\
l & ::= & e
\end{array}
$$

[1] Since Java does not provide separate lock acquire and release operations, we do not need separate left movers and right movers, since each expression is either a mover in both directions or not at all.

Each atomicity a is equivalent to a function $[\![a]\!]$ from the set of locks currently held to a basic atomicity:

$$\begin{array}{rcl} [\![\alpha]\!](ls) & = & \alpha \\ [\![l \,?\, a_1 : a_2]\!](ls) & = & \begin{cases} [\![a_1]\!](ls) & \text{if } l \in ls \\ [\![a_2]\!](ls) & \text{if } l \notin ls \end{cases} \end{array}$$

For example, the conditional atomicity a:

$$l_1 \,?\, \texttt{mover} : (l_2 \,?\, \texttt{atomic} : \texttt{error})$$

is equivalent to the function:

$$[\![a]\!](ls) = \begin{cases} \texttt{mover} & \text{if } l_1 \in ls \\ \texttt{atomic} & \text{if } l_1 \notin ls,\ l_2 \in ls \\ \texttt{error} & \text{if } l_1 \notin ls,\ l_2 \notin ls \end{cases}$$

We extend the calculation of iterative closure, sequential composition, and join operations to conditional atomicities as follows:

$$\begin{array}{rcl} (l \,?\, a : b)^* & = & l \,?\, a^* : b^* \\ (l \,?\, a_1 : a_2); b & = & l \,?\, (a_1; b) : (a_2; b) \\ \alpha; (l \,?\, b_1 : b_2) & = & l \,?\, (\alpha; b_1) : (\alpha; b_2) \\ (l \,?\, a_1 : a_2) \sqcup b & = & l \,?\, (a_1 \sqcup b) : (a_2 \sqcup b) \\ \alpha \sqcup (l \,?\, a_1 : a_2) & = & l \,?\, (\alpha \sqcup a_1) : (\alpha \sqcup a_2) \end{array}$$

We also extend the calculation of subatomicity ordering to conditional atomicities. To decide $a \sqsubseteq b$, we use an auxiliary relation $\sqsubseteq^h_n$, where h is a set of locks that is known to be *h*eld by the current thread, and n is a set of locks that is known to be *n*ot held by the current thread. Intuitively, the condition $a \sqsubseteq^h_n b$ holds if and only if $[\![a]\!](ls) \sqsubseteq [\![b]\!](ls)$ holds for every lockset ls that contains h and is disjoint from n. We define $a \sqsubseteq b$ to be $a \sqsubseteq^\emptyset_\emptyset b$ and check $a \sqsubseteq^h_n b$ recursively as follows:

$$\frac{\alpha \sqsubseteq \beta}{\alpha \sqsubseteq^h_n \beta}$$

$$\frac{(l \notin n \Rightarrow a_1 \sqsubseteq^{h \cup \{l\}}_n b) \quad (l \notin h \Rightarrow a_2 \sqsubseteq^h_{n \cup \{l\}} b)}{l \,?\, a_1 : a_2 \sqsubseteq^h_n b}$$

$$\frac{(l \notin n \Rightarrow \alpha \sqsubseteq^{h \cup \{l\}}_n b_1) \quad (l \notin h \Rightarrow \alpha \sqsubseteq^h_{n \cup \{l\}} b_2)}{\alpha \sqsubseteq^h_n l \,?\, b_1 : b_2}$$

The following theorem claims that the iterative closure, sequential composition, and join operations on conditional atomicities are the pointwise extensions of the corresponding operations on basic atomicities. Similarly, the subatomicity ordering on conditional atomicities is the pointwise extension of the subatomicity ordering on basic atomicities.

THEOREM 1. *For all atomicities a and b, the following statements are true.*

1. *For all locksets ls,*

$$\begin{array}{rcl} [\![a^*]\!](ls) & = & ([\![a]\!](ls))^* \\ [\![a; b]\!](ls) & = & [\![a]\!](ls); [\![b]\!](ls) \\ [\![a \sqcup b]\!](ls) & = & [\![a]\!](ls) \sqcup [\![b]\!](ls) \end{array}$$

2. $a \sqsubseteq b \Leftrightarrow \forall ls.\ [\![a]\!](ls) \sqsubseteq [\![b]\!](ls)$

The relation $\sqsubseteq$ is an equivalence relation with minimum element `const` and maximum element `error`. Atomicities a and b are *equivalent*, written $a \equiv b$, if $a \sqsubseteq b$ and $b \sqsubseteq a$. If $a \equiv b$, then $\forall ls.\ [\![a]\!](ls) = [\![b]\!](ls)$. The equivalence relation $\equiv$ identifies atomicities that are syntactically different but semantically equal. For example, $(l \,?\, \texttt{mover} : \texttt{mover}) \equiv \texttt{mover}$. The following theorem states interesting properties of atomicities.

THEOREM 2. *For all atomicities a, b, and c, the following statements are true.*

1. *Iterative closure is monotonic and idempotent.*

$$\begin{array}{rcl} a & \sqsubseteq & a^* \\ (a^*)^* & \equiv & a^* \end{array}$$

2. *Sequential composition is monotonic and associative and `const` is a left and right identity of this operation.*

$$\begin{array}{rcl} a & \sqsubseteq & a; b \\ (a; b); c & \equiv & a; (b; c) \\ \texttt{const}; a & \equiv & a \\ a; \texttt{const} & \equiv & a \end{array}$$

3. *Sequential composition and iterative closure distribute over the join operation.*

$$\begin{array}{rcl} a; (b \sqcup c) & \equiv & a; b \sqcup a; c \\ (a \sqcup b); c & \equiv & a; c \sqcup b; c \\ (a \sqcup b)^* & \equiv & a^* \sqcup b^* \end{array}$$

3.3 The Type System

The atomicity of a field access depends on the synchronization discipline used for that field. Our type system relies on the programmer to explicate this synchronization discipline as a type annotations. The annotation `guarded_by` l expresses the common synchronization discipline that the lock expression l must be held whenever the field is read or written. The annotation `write_guarded_by` l states that the lock expression l must be held for writes, but not necessarily for reads. If neither annotation is present, the field can be read or written at any time.

The soundness of the type system requires that each lock expression l denotes a fixed lock throughout the execution of the program. We satisfy this requirement by ensuring that each lock expression has atomicity `const`; such expressions include references to immutable variables[2], accesses to final fields of `const` expressions, and calls to `const` methods with `const` arguments.

Each method declaration includes a specification of the method's atomicity. The type system checks that the body of the method has this atomicity, and uses this atomicity at call sites of the method. We extend the syntax of field and method declarations to include these type annotations, and refer to the extended language as ATOMICJAVA.

$$\begin{array}{rcll} \textit{field} & ::= & [\texttt{final}]_{\text{opt}}\ t\ \textit{fd}\ [g]_{\text{opt}}\ \texttt{=}\ e & \text{(fields)} \\ \textit{meth} & ::= & a\ t\ \textit{mn}\texttt{(}\textit{arg}^*\texttt{)}\ \texttt{\{}\ e\ \texttt{\}} & \text{(methods)} \\ g & ::= & \texttt{guarded\_by}\ l \mid \texttt{write\_guarded\_by}\ l & \text{(guards)} \\ l & ::= & e & \text{(lock expression)} \end{array}$$

A method declaration may also contain the type annotation `requires` $l_1, \ldots, l_n$ stating that the locks $l_1, \ldots, l_n$ should be held at any call site of the method. A method declaration with a `requires` clauses, such as:

$$a\ t\ \textit{mn}\texttt{(}\textit{arg}^*\texttt{)}\ \texttt{requires}\ l_1, \ldots, l_n\ \texttt{\{}\ e\ \texttt{\}}$$

[2] All variables are immutable in ATOMICJAVA, but only final variables are in Java.

is an abbreviation for the declaration:

$$(l_1 \,?\, l_2 \,?\, \ldots \,?\, l_n \,?\, a)\ t\ mn(arg^*)\ \{\ e\ \}$$

where the conditional atomicity $(l_1 \,?\, l_2 \,?\, \ldots \,?\, l_n \,?\, a)$ is equivalent to a if the locks $l_1, \ldots, l_n$ are all held, and equivalent to `error` otherwise.

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

$$P; E \vdash e : t \ \&\ a\,.$$

Here, P (the program being checked) is included in the judgment to provide information about class definitions in the program; E is an environment providing types for the free variables of e; t is the type of e, and a is the atomicity of e.

The rule [EXP WHILE] for `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.

$$\textsc{[exp while]}\quad \frac{P;E \vdash e_1 : \texttt{int}\ \&\ a_1 \qquad P;E \vdash e_2 : t\ \&\ a_2}{P;E \vdash \texttt{while}\ e_1\ e_2 : \texttt{int}\ \&\ (a_1;(a_2;a_1)^*)}$$

The atomicity of a field access $e.fd$ depends on the synchronization discipline, if any, used for that variable. If fd is a final field, then the rule [EXP REF FINAL] checks that e is a well-typed expression of some class type c and that c declares or inherits a final field fd of type t. It states that $e.fd$ has type t and atomicity $a;$`const`, where a is the atomicity of e.

$$\textsc{[exp ref final]}\quad \frac{\begin{array}{c}P;E \vdash e : c\ \&\ a\\ P;E \vdash (\texttt{final}\ t\ fd\ \texttt{=}\ e') \in c\end{array}}{P;E \vdash e.fd : t\ \&\ (a;\texttt{const})}$$

The rule [EXP REF RACE] deals with the case where fd is not final and not protected.

$$\textsc{[exp ref race]}\quad \frac{\begin{array}{c}P;E \vdash e : c\ \&\ a\\ P;E \vdash (t\ fd\ \ \texttt{=}\ e') \in c\end{array}}{P;E \vdash e.fd : t\ \&\ (a;A(t))}$$

The atomicity $A(t)$ of the field reference depends on the type t. If t is a class type or `int`, then the field reference is `atomic`. If t is `long`, then the field may be read with two 32-bit loads, and hence is `cmpd`.

$$A(t) = \begin{cases} \texttt{atomic} & \text{if } t \neq \texttt{long} \\ \texttt{cmpd} & \text{if } t = \texttt{long} \end{cases}$$

The rule [EXP REF GUARD] applies when fd is guarded by a lock l. In this case, the field reference has atomicity `mover` provided the lock is held, and has atomicity `error` otherwise. The substitution $l[\texttt{this} := e]$ accounts for the aliasing between `this` and e. That is, occurrences of `this` in the lock expression l refer to the object being dereferenced, which is the same object as that denoted by e.

$$\textsc{[exp ref guard]}\quad \frac{\begin{array}{c}P;E \vdash e : c\ \&\ a\\ P;E \vdash (t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e') \in c\\ b \equiv (l[\texttt{this} := e]\,?\,\texttt{mover}) \qquad P;E \vdash b\end{array}}{P;E \vdash e.fd : t\ \&\ (a;b)}$$

The rule [EXP REF WRITE GUARD] applies when fd is write guarded by a lock, *i.e.*, the lock must be held for writes but not for reads. If the lock is held, then the reference is a mover, since it commutes with reads by other threads, and no other thread can write to the field. If the lock is not held, the field reference has atomicity $A(t)$, *i.e.*, the read is atomic if and only if fd is not of type `long`.

$$\textsc{[exp ref write guard]}\quad \frac{\begin{array}{c}P;E \vdash e : c\ \&\ a\\ P;E \vdash (t\ fd\ \texttt{write\_guarded\_by}\ l\ \texttt{=}\ e') \in c\\ b \equiv (l[\texttt{this} := e]\,?\,\texttt{mover} : A(t)) \qquad P;E \vdash b\end{array}}{P;E \vdash e.fd : t\ \&\ (a;b)}$$

The rules for field updates $e.fd = e'$ are similar to those for field accesses. A final field cannot be updated. A `guarded_by` field can only be updated if the appropriate lock is held, and the update is a mover. A `write_guarded_by` field can only be updated if the appropriate lock is held, and the update has atomicity $A(t)$, where t is the type of the field. An unprotected field can always be updated, and the update has atomicity $A(t)$.

The atomicity of a method call reflects the atomicity of the callee. The substitution $b[\texttt{this} := e_0]$ in the method call rule accounts for the aliasing between `this` and e_0. That is, occurrences of `this` in the method's atomicity b refer to the object being invoked, which is the same object as that denoted by e_0.

$$\textsc{[exp call]}\quad \frac{\begin{array}{c}P;E \vdash e_i : t_i\ \&\ a_i \qquad t_0 = c \qquad P;E \vdash b[\texttt{this} := e_0]\\ P;E \vdash (b\ s\ mn(t_1\ y_1, \ldots, t_n\ y_n)\ \{\ e\ \}) \in c\end{array}}{P;E \vdash e_0.mn(e_1, \ldots, e_n) : s\ \&\ (a_0;a_1;\ldots;a_n;b[\texttt{this} := e_0])}$$

The rule [EXP SYNC] for `synchronized` $l\ e$ checks that l is a `const` expression of some class type c and infers the atomicity a of the synchronized body e.

$$\textsc{[exp sync]}\quad \frac{P;E \vdash l : c\ \&\ \texttt{const} \qquad P;E \vdash e : t\ \&\ a}{P;E \vdash \texttt{synchronized}\ l\ e : t\ \&\ S(l,a)}$$

The function S defined below determines the atomicity of the synchronized statement. For example, if the body is a mover and the lock is already held, then the synchronized statement is also a mover, since the acquire and release operations are no-ops. If the body is a mover and the lock is not already held, then the synchronized statement is atomic, since the execution consists of a right mover (the acquire), followed by a left and right mover (the body), followed by a left mover (the release). If the body has conditional atomicity $l \,?\, b_1 : b_2$, then we ignore b_2 and recursively apply S to b_1, since we know that l is held within the synchronized body. If the body has some other conditional atomicity, then we recursively apply S to both branches.

$$\begin{array}{rcl}
S(l, \texttt{const}) & = & l \,?\, \texttt{const} : \texttt{atomic} \\
S(l, \texttt{mover}) & = & l \,?\, \texttt{mover} : \texttt{atomic} \\
S(l, \texttt{atomic}) & = & \texttt{atomic} \\
S(l, \texttt{cmpd}) & = & \texttt{cmpd} \\
S(l, \texttt{error}) & = & \texttt{error} \\
S(l, (l \,?\, b_1 : b_2)) & = & S(l, b_1) \\
S(l, (l' \,?\, b_1 : b_2)) & = & l' \,?\, S(l,b_1) : S(l,b_2) \quad \text{if } l \neq l'
\end{array}$$

The rule [EXP FORK] for `fork` e requires that the forked expression have atomicity `cmpd`. In particular, a coarser atomicity such as $l\,?\,\texttt{cmpd} : \texttt{error}$ is not allowed, because this atomicity is equivalent to `error` when the lock l is not held, and the newly forked thread does not initially hold any locks.

$$\frac{P; E \vdash e : t \;\&\; \texttt{cmpd}}{P; E \vdash \texttt{fork}\ e : \texttt{int} \;\&\; \texttt{atomic}} \quad \text{[EXP FORK]}$$

The type system also supports subtyping and subatomicities.

$$\frac{\begin{array}{c} P; E \vdash e : s \;\&\; a \\ P \vdash s <: t \quad P; E \vdash a \sqsubseteq b \end{array}}{P; E \vdash e : t \;\&\; b} \quad \text{[EXP SUB]}$$

The remaining type rules are mostly straightforward. The complete set of type judgments and rules is contained in Appendix A. If a program P is well-typed according to these rules, and an arbitrarily-interleaved execution of P reaches a state s in which no thread is executing an atomic method, then the state s is also reachable via a serial execution of P. An execution of P is serial if the execution of an atomic method is never interleaved with actions of other threads. This soundness property has been formally proved for an earlier version of our type system [20] for a sequentially consistent shared-memory model.

3.4 Atomic bank accounts

To illustrate the use of our type system, we now apply it to the bank account example of Section 1.1. We first add type annotations to the initial version of the bank account stating that the field `balance` is guarded by `this`, and that all bank account methods are atomic.

```
class Account {
  int balance guarded_by this = 0;
  atomic int deposit2(int x)    { ... }
  atomic int readBalance1()     { ... }
  atomic int withdraw1(int amt) { ... }
}
```

Our type system detects that the `withdraw1` method is not atomic, since it consists of two sequentially composed atomic expressions and therefore behaves erroneously under certain thread interleavings.

We replace `withdraw1` with the fixed method `withdraw2`, and also optimize `readBalance1` to `readBalance2`, resulting in an optimized synchronization discipline that we explicate using the `write_guarded_by` annotation:

```
class Account {
  int balance write_guarded_by this = 0;
  atomic int deposit2(int x)    { ... }
  atomic int readBalance2()     { ... }
  atomic int withdraw2(int amt) { ... }
}
```

The corrected and optimized implementation type checks, indicating that all these methods are atomic.

An alternative implementation of the bank account may rely on its clients to perform the necessary synchronization operations. The following method signatures explicate the requirement that the object's lock must be acquired before calling certain methods, but not others. The methods `deposit3` and `withdraw3` are versions of `deposit2` and `withdraw2` where the synchronized statement is hoisted out of the method bodies and left to the caller. Our type system can again verify that all these methods are atomic.

```
class Account {
  int balance write_guarded_by this = 0;
  atomic int deposit3(int x)    requires this { ... }
  atomic int readBalance2()                   { ... }
  atomic int withdraw3(int amt) requires this { ... }
}
```

4. EVALUATION

To evaluate the usefulness of our type system, we have implemented it for the full Java programming language [23] and applied it to a variety of standard Java library classes.

4.1 Implementation

Our implementation extends the type system outlined so far to handle the additional features of Java, including arrays, interfaces, constructors, static fields and methods, inner classes, and so on. The extra type and atomicity annotations required by the type checker are embedded in special Java comments that start with the character "`#`", thus preserving compatibility with existing Java compilers and other tools. The default atomicity for unannotated routines is `cmpd`, thus the atomic type checker passes all unannotated, well-typed Java programs. The checker allows class declarations to be annotated with an atomicity that is the default atomicity for each method in that declaration, which makes it is easy to specify that every method in a class is atomic.

The atomicity checker is built on top of the race condition checker `rccjava` [18], and re-uses `rccjava`'s machinery for reasoning about the set of locks held at each program point and the locks used to protect fields. The checker also infers the atomicity of each expression and statement in the program, and checks the atomicity of each method body. If a method body's atomicity does not match the declared atomicity of the method, an appropriate error message is produced. This error message describes the inferred atomicity of each operation in the method body, which is crucial for determining the cause of atomicity violations.

In practice, programs use a variety of synchronization mechanisms, not all of which can be captured by our type rules. Like `rccjava`, the atomicity checker is able to relax the formal type system in several ways when it proves too restrictive. The `no_warn` annotation turns off certain kinds of warnings on a particular line of code, and is commonly used if a particular race condition is considered benign. The `holds` annotation causes the checker to assume that a particular lock is held from the current program point to the end of that statement block.

The checker may be configured to make global assumptions about when locks are held. For instance, the command line flag "`-constructor_holds_lock`" causes the checker to assume that the lock `this` is held in constructors. This assumption is sound as long as references to `this` do not escape to other threads before the constructor returns. Violations of this assumption are unlikely, and using it eliminates a large number of spurious warnings. We believe this command line flag could be replaced with a sound escape analysis [10, 34] without significant reduction in the expressiveness of the system.

Extending the atomic type system to handle arrays introduces a number of technical challenges. Following `rccjava`, we use the type annotation `/*# elems_guarded_by` l `*/` to specify the lock guarding the elements in an array. We also introduce type annotations for arrays that are local to a particular thread, and for read-only arrays. In many cases, a newly-allocated array is local to its allocating thread during its initialization phase, and is later shared between threads, either protected by a lock, or in a read-only mode. Since the protection mechanism is part of the array's type, we use typecasts to accomodate such changes in the protection mechanism. These typecasts are currently unsound, as in C, rather than dynamically checked, as in Java. Many of these typecasts could be statically checked by extending our system with linear, unique, or ownership types [31, 6, 5].

4.2 Applications

To evaluate and gain experience with the atomicity checker, we applied it to check several standard Java classes from JDK1.4 that are intended to be atomic. These classes include `StringBuffer`, `String`, `PrintWriter`, `Vector`, `URL`, `Inflator`, and `Deflator`, and vary in size from 296 to 2399 lines of code.

Adding appropriate type annotations to these classes was mostly straightforward, once the synchronization discipline of each class was understood. Determining the synchronization discipline of each class was often an iterative process, where we used the checker to investigate the behavior of large code files, and to find violations of a hypothesised synchronization discipline. While verifying the atomicity of these classes, we had to add appropriate atomicity annotations to called methods in other classes. We also used the command line flag `-constructor_holds_lock`.

The atomicity checker succeeded in detecting a number of subtle atomicity violations, including errors that would not be caught by a race condition checker. A particularly clear example of the benefits of our type system is provided by the the class `java.util.StringBuffer` (version 1.70 from JDK 1.4). The documentation of this class states that all `StringBuffer` methods are atomic. The `StringBuffer` implementation uses lock-based synchronization to achieve this atomicity guarantee, and we formalized this synchronization discipline using `guarded_by` annotations. The following `StringBuffer` method `append` failed to type check, and an examination of the method reveals that it violates its atomicity specification:

```
public final class StringBuffer ... {
 ...
private int count /*# guarded_by this */;

/*# atomic */   // does not type check
public synchronized StringBuffer append(StringBuffer sb){
  if (sb == null) { sb = NULL; }
  int len = sb.length();             // len may be stale
  int newcount = count + len;
  if (newcount > value.length) expandCapacity(newcount);
  sb.getChars(0, len, value, count); // use of stale len
  count = newcount;
  return this;
}
/*# atomic */
public synchronized int length() { return count; }

/*# atomic */
public synchronized void getChars(...) { ... }
}
```

After `append` calls the synchronized method `sb.length()`, a second thread could remove characters from `sb`. In this situation, `len` is now *stale* [9] and no longer reflects the current length of `sb`, and so `getChars` is called with invalid arguments and throws a `StringIndexOutOfBoundsException`. The following test harness triggers this crash.

```
public class BreakStringBuffer extends Thread {
    static StringBuffer sb = new StringBuffer("abc");
    public void run() {
      while(true) { sb.delete(0,3); sb.append("abc"); }
    }
    public static void main(String[] argv) {
      (new BreakStringBuffer()).start();
      while(true) (new StringBuffer()).append(sb);
    }
}
```

We also type checked `java.lang.String`, and discovered that it contains a method `contentEquals`, which suffers from a similar defect: a property is checked in one synchronized block and assumed to still hold in a subsequent synchronized block, resulting in a potential `ArrayIndexOutOfBoundsException`.

```
  public boolean contentEquals(StringBuffer sb) {
    if (count != sb.length()) return false;
    // under a sequential execution count == sb.length()
    // but concurrent threads may change that property
    ...
    char v2[] = sb.getValue();
    // subsequent code wrongly assumes v2.length==count
    // and may throw an ArrayIndexOutOfBoundsException
    ...
  }
```

Type checking `java.io.PrintWriter` raised interesting issues concerning *rep-exposure* [15]. For example, the following `PrintWriter` method tries to ensure atomicity using synchronization:

```
    public void println(int x) {
        synchronized (lock) {
            print(x);
            println();
        }
    }
```

However, both `print` and `println` write to an underlying `Writer`, which was originally passed to the `PrintWriter` constructor. Hence, some other thread could concurrently write characters to the `Writer`, without acquiring the protecting lock used by `PrintWriter`. To deal with this problem, we declared `println` and 9 similar methods in `PrintWriter` as `cmpd`, and the remaining 17 public methods as `atomic`, and then succeeded in type checking `PrintWriter`. Our experience suggests that an ownership type system [5, 6] or escape analysis [10, 34] for reasoning about rep-exposure [15] would be helpful in verifying atomicity.

The class `java.util.Vector` illustrates the need for the extra precision in our type system afforded by conditional atomicities. The public method `removeElementAt` is `atomic` when called without the vector's lock being held, but also must be a `mover` when called from `removeElement` with the vector's lock held, in order to verify the atomicity of `removeElement`. Assigning `removeElementAt` the atomicity `this ? mover : atomic` allows this class to type check.

```
public class Vector ... {

  /*# conditional_atomicity this ? mover : atomic */
  public void synchronized  removeElementAt(int index)
  { ... }
```

Class	LOC	Annotations per KLOC					
		total	guard	requires	atomicity	arrays	escapes
`java.util.zip.Inflater`	296	20.3	16.9	0.0	3.4	0.0	0.0
`java.util.zip.Deflater`	364	24.7	19.2	0.0	5.5	0.0	0.0
`java.io.PrintWriter`	557	35.9	5.4	0.0	25.1	0.0	5.4
`java.util.Vector`	1029	13.6	2.9	1.0	3.9	2.9	2.9
`java.net.URL`	1269	33.1	10.2	0.8	9.5	0.0	12.6
`java.lang.StringBuffer`	1272	18.9	2.4	3.9	4.7	7.1	0.8
`java.lang.String`	2399	21.7	0.0	0.0	1.3	19.2	1.3
All benchmarks	7186	23.3	4.7	1.0	5.9	8.1	3.6

Table 1: Programs analyzed using the Atomicity Checker.

```
  /*# atomic */
  public synchronized boolean removeElement(Object obj){
      modCount++;
      int i = indexOf(obj);
      if (i >= 0) {
          removeElementAt(i);
          return true;
      }
      return false;
  }
  /*# atomic */
  public int indexOf(Object elem) { ... }
  ...
}
```

Apart from the need for conditional atomicities, type checking `java.util.Vector` was mostly straightforward. In particular, the race condition in `lastIndexOf` from JDK1.1 detected by `rccjava` has been fixed.

The synchronization discipline used by `java.net.URL` is fairly involved, and the atomicity checker reported a number of race conditions. For example, the following method can be simultaneously called from multiple threads, resulting in multiple initializations of the field `specifyHandlerPerm`:

```
private static NetPermission specifyHandlerPerm;
private void checkSpecifyHandler(SecurityManager sm) {
  if (specifyHandlerPerm == null)
      specifyHandlerPerm =
          new NetPermission("specifyStreamHandler");
  sm.checkPermission(specifyHandlerPerm);
}
```

We have not yet determined if these warnings reflect real errors in the program or benign race conditions.

We summarize our experience in checking these classes in Table 1. It shows the names and the sizes of the various classes that we checked; the number of annotations per thousand lines of code required for each class; and breaks down this number into guard annotations (`guarded_by` and `write_guarded_by`), `requires` annotations, atomicity annotations, array annotations (`elems_guarded_by`, etc), and escapes from the type system (`holds`, `no_warn`).

5. RELATED WORK

Lipton [28] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [38], Back [4], and Lamport and Schneider [27] extended this work to allow proofs of general safety properties. Cohen and Lamport [12] extended reduction to allow proofs of liveness properties. Misra [32] has proposed a reduction theorem for programs built with monitors [26] communicating via procedure calls. Bruening [8] and Stoller [37] have used reduction to improve the efficiency of model checking.

A number of tools have been developed for detecting race conditions, both statically and dynamically. The Race Condition Checker [18] uses a type system to catch race conditions in Java programs. This approach has been extended [7, 5] and adapted to other languages [24]. Other static race detection tools include Warlock [36], for ANSI C programs, and ESC/Java [19], which catches a variety of software defects in addition to race conditions. ESC/Java has been extended to catch "higher-level" race conditions, where a stale value from one synchronized block is used in a subsequent synchronized block [9]. Vault [13] 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 [35] detects race conditions and deadlocks dynamically, rather than statically. The Eraser algorithm has been extended to object-oriented languages [40] and has been improved for precision and performance [11]. A variety of other approaches have been developed for race and deadlock prevention; they are discussed in more detail in earlier papers [17, 18]. An alternative approach is to generate synchronization code automatically from high-level specifications [14].

Thus, reduction has been studied in depth, as have type systems for preventing race conditions. This paper combines these existing techniques in a type system that provides an effective means for checking atomicity.

Recently, Freund and Qadeer have combined both reduction and simulation in the Calvin checker to verify concise procedure specifications in multithreaded programs [22]. Our atomic type system is inspired by the Calvin checker, but represents a different point in the tradeoff between scalability and expressiveness. While Calvin's semantic analysis based on verification conditions and automatic theorem proving is more powerful, the syntactic type-based analysis of this paper provides several key benefits; it is simpler, more predictable, more scalable, and requires fewer annotations than the Calvin checker.

Atomicity is a semantic correctness condition for multithreaded software. In this respect, it is similar to strict serializability [33] for database transactions and linearizability [25] for concurrent objects. However, we are not aware of any automated techniques to verify these conditions. We hope that the lightweight analysis for atomicity presented in this paper can be leveraged to develop checking tools for other semantic correctness conditions as well.

While our type system can check the atomicity of code blocks, researchers have proposed using atomic blocks as a language primitive. Lomet [30] first proposed the use of atomic blocks for synchronization. The Argus [29] and Avalon [16] projects developed language support for implementing atomic objects. Persistent languages [2, 3] are attempting to augment atomicity with data persistence in order to introduce transactions into programming languages.

6. CONCLUSION

Reasoning about the behavior of multithreaded programs is difficult, due to the potential for subtle interactions between threads. However, programmers often expect that in certain "atomic" methods, such interactions do not occur, and document these beliefs by characterizing these methods as "synchronized" or "thread-safe". Knowing that certain methods are atomic significantly simplifies subsequent (formal or informal) reasoning about the correctness of those methods, since they can be checked using traditional sequential reasoning techniques. However, despite the crucial role of atomicity in reasoning about the behavior of multithreaded programs, programmers have had little support for formally documenting or verifying atomicity properties.

To remedy this situation, we propose an extension to the type language to allow methods to be annotated as `atomic`. In addition, we present a type system for checking these atomicity assertions. Although necessarily incomplete, this atomic type system can handle a number of widely-used synchronization disciplines. We have implemented the atomic type system, and our experience to date indicates that this technique is a promising approach for building more reliable multithreaded software. Our type checker uncovered atomicity violations in classes such as `java.lang.String` and `java.lang.StringBuffer` that cause crashes under certain thread interleavings.

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 crucial property of many methods in multithreaded programs.

Acknowledgments: We thank Martín Abadi, Chandrashekhar Boyapati, Dan Grossman, Stephen Freund, Shriram Krishnamurthi and Sanjit Seshia for comments on this paper.

7. REFERENCES

[1] A. Aiken and D. Gay. Barrier inference. In *POPL 98: Principles of Programming Languages*, pages 243–354. ACM Press, 1998.

[2] 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.

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

[4] 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.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA 02: Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM Press, 2002.

[6] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report 853, MIT Laboratory for Computer Science, June 2002.

[7] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA 01: Object-Oriented Programming, Systems, Languages, and Applications*, pages 56–69. ACM Press, 2001.

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

[9] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. Technical Note 2002-4, Compaq Systems Research Center, May 2002.

[10] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *OOPSLA 99: Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19. ACM Press, 1999.

[11] 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 *PLDI 02: Programming Language Design and Implementation*, pages 258–269. ACM Press, 2002.

[12] E. Cohen and L. Lamport. Reduction in TLA. In *CONCUR 98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 1998.

[13] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*, pages 59–69. ACM Press, 2001.

[14] X. Deng, M. Dwyer, J. Hatcliff, and M. Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *ICSE 02: International Conference on Software Engineering*, pages 442–452. ACM Press, 2002.

[15] D. L. Detlefs, K. R. M. Leino, and C. G. Nelson. Wrestling with rep exposure. Research Report 156, DEC Systems Research Center, July 1998.

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

[17] C. Flanagan and M. Abadi. Types for safe locking. In *ESOP 99: European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108, 1999.

[18] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI 00: Programming Language Design and Implementation*, pages 219–232. ACM Press, 2000.

[19] C. Flanagan, K. R. M. Leino, M. D. Lillibridge, C. G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 02: Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[20] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI 03: Types in Language Design and Implementation*, pages 1–12. ACM Press, 2003.

[21] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL 98: Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

[22] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. Technical Note 01-2002, Williams College, December 2002.

[23] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[24] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI 03: Types in Language Design and Implementation*, pages 13–25. ACM Press, 2003.

[25] 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.

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

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

$\boxed{P \vdash t}$

[TYPE C] $$\frac{\texttt{class}\ c\ \ldots \in P}{P \vdash c}$$

[TYPE OBJECT] $$\frac{}{P \vdash \texttt{Object}}$$

[TYPE INT] $$\frac{}{P \vdash \texttt{int}}$$

[TYPE LONG] $$\frac{}{P \vdash \texttt{long}}$$

$\boxed{P \vdash s <: t}$

[SUBTYPE REFL] $$\frac{P \vdash t}{P \vdash t <: t}$$

[SUBTYPE CLASS] $$\frac{P \vdash c_1 <: c_2 \quad \texttt{class}\ c_2\ \texttt{extends}\ c_3 \ldots \in P}{P \vdash c_1 <: c_3}$$

$\boxed{P \vdash E}$

[ENV EMPTY] $$\frac{}{P \vdash \emptyset}$$

[ENV VAR] $$\frac{P \vdash E \quad P \vdash t \quad x \notin Dom(E)}{P \vdash E, t\ x}$$

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

[ATOM BASIC] $$\frac{P \vdash E}{P; E \vdash \alpha}$$

[ATOM COND] $$\frac{P; E \vdash l : c\ \&\ \texttt{const} \quad P; E \vdash a_i}{P; E \vdash l\,?\,a_1 : a_2}$$

$\boxed{P; E \vdash e : t\ \&\ a}$

[EXP SUB] $$\frac{\begin{array}{c} P; E \vdash e : s\ \&\ a \\ P \vdash s <: t \quad a \sqsubseteq b \end{array}}{P; E \vdash e : t\ \&\ b}$$

[EXP NEW] $$\frac{P \vdash E \quad P \vdash c}{P; E \vdash \texttt{new}\ c : c\ \&\ \texttt{mover}}$$

[EXP VAR] $$\frac{P \vdash E \quad E = E_1, t\ x, E_2}{P; E \vdash x : t\ \&\ \texttt{const}}$$

[EXP REF FINAL] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash (\texttt{final}\ t\ fd\ \texttt{=}\ e') \in c \end{array}}{P; E \vdash e.fd : t\ \&\ (a; \texttt{const})}$$

[EXP REF RACE] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash (t\ fd\ \ \texttt{=}\ e') \in c \end{array}}{P; E \vdash e.fd : t\ \&\ (a; A(t))}$$

[EXP REF GUARD] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e') \in c \\ b \equiv (l[\texttt{this} := e]\,?\,\texttt{mover}) \quad P; E \vdash b \end{array}}{P; E \vdash e.fd : t\ \&\ (a; b)}$$

[EXP REF WRITE GUARD] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash (t\ fd\ \texttt{write\_guarded\_by}\ l\ \texttt{=}\ e') \in c \\ b \equiv (l[\texttt{this} := e]\,?\,\texttt{mover} : A(t)) \quad P; E \vdash b \end{array}}{P; E \vdash e.fd : t\ \&\ (a; b)}$$

[EXP ASSIGN RACE] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash e' : t\ \&\ a' \\ P; E \vdash (t\ fd\ \texttt{=}\ e'') \in c \end{array}}{P; E \vdash e.fd\ \texttt{=}\ e' : t\ \&\ (a; a'; A(t))}$$

[EXP ASSIGN GUARD] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash e' : t\ \&\ a' \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e'') \in c \\ b \equiv (l[\texttt{this} := e]\,?\,\texttt{mover}) \quad P; E \vdash b \end{array}}{P; E \vdash e.fd\ \texttt{=}\ e' : t\ \&\ (a; a'; b)}$$

[EXP ASSIGN WRITE GUARD] $$\frac{\begin{array}{c} P; E \vdash e : c\ \&\ a \\ P; E \vdash e' : t\ \&\ a' \\ P; E \vdash (t\ fd\ \texttt{write\_guarded\_by}\ l\ \texttt{=}\ e'') \in c \\ b \equiv (l[\texttt{this} := e]\,?\,A(t)) \quad P; E \vdash b \end{array}}{P; E \vdash e.fd\ \texttt{=}\ e' : t\ \&\ (a; a'; b)}$$

[EXP CALL] $$\frac{\begin{array}{c} P; E \vdash e_i : t_i\ \&\ a_i \quad t_0 = c \quad P; E \vdash b[\texttt{this} := e_0] \\ P; E \vdash (b\ s\ mn(t_1\ y_1, \ldots, t_n\ y_n)\ \{\ e\ \}) \in c \end{array}}{P; E \vdash e_0.mn(e_1, \ldots, e_n) : s\ \&\ (a_0; a_1; \ldots; a_n; b[\texttt{this} := e_0])}$$

[EXP LET] $$\frac{\begin{array}{c} P; E \vdash e_1 : t_1\ \&\ a_1 \\ P; E, t\ x \vdash e_2 : t_2\ \&\ a_2 \quad P; E \vdash a_2[x := e_1] \end{array}}{P; E \vdash \texttt{let}\ t\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : t_2\ \&\ (a_1; a_2[x := e_1])}$$

[EXP IF] $$\frac{P; E \vdash e_1 : \texttt{int}\ \&\ a_1 \quad P; E \vdash e_2 : t\ \&\ a_2 \quad P; E \vdash e_3 : t\ \&\ a_3}{P; E \vdash \texttt{if}\ e_1\ e_2\ e_3 : t\ \&\ (a_1; (a_2 \sqcup a_3))}$$

[EXP WHILE] $$\frac{P; E \vdash e_1 : \texttt{int}\ \&\ a_1 \quad P; E \vdash e_2 : t\ \&\ a_2}{P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{int}\ \&\ (a_1; (a_2; a_1)^*)}$$

[EXP SYNC] $$\frac{P; E \vdash l : c\ \&\ \texttt{const} \quad P; E \vdash e : t\ \&\ a}{P; E \vdash \texttt{synchronized}\ l\ e : t\ \&\ S(l, a)}$$

[EXP FORK] $$\frac{P; E \vdash e : t\ \&\ \texttt{cmpd}}{P; E \vdash \texttt{fork}\ e : \texttt{int}\ \&\ \texttt{atomic}}$$

$\boxed{P; E \vdash \mathit{field}}$

[FIELD NO GUARD] $$\frac{\begin{array}{c} P \vdash E \\ P; \emptyset \vdash e : t\ \&\ \texttt{mover} \end{array}}{P; E \vdash [\texttt{final}]_{\mathrm{opt}}\ t\ fd\ \texttt{=}\ e}$$

[FIELD GUARD] $$\frac{\begin{array}{c} P; E \vdash l : c\ \&\ \texttt{const} \\ P; \emptyset \vdash e : t\ \&\ \texttt{mover} \end{array}}{P; E \vdash t\ fd\ [\texttt{write\_}]\texttt{guarded\_by}\ l\ \texttt{=}\ e}$$

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

[METHOD] $$\frac{\begin{array}{c} P; E \vdash a \quad P \vdash t \\ P; E, arg_{1\ldots n} \vdash e : t\ \&\ a \end{array}}{P; E \vdash a\ t\ mn(arg_{1\ldots n})\ \{\ e\ \}}$$

$\boxed{P \vdash \mathit{field} \in c}$

[FIELD MEMBER] $$\frac{\begin{array}{c} P \vdash c <: c' \\ \texttt{class}\ c'\ \ldots\ \{\ldots\ \mathit{field}\ \ldots\} \in P \end{array}}{P \vdash \mathit{field} \in c}$$

$\boxed{P \vdash \mathit{meth} \in c}$

[METHOD MEMBER] $$\frac{\begin{array}{c} P \vdash c <: c' \\ \texttt{class}\ c'\ \ldots\ \{\ldots\ \mathit{meth}\ \ldots\} \in P \end{array}}{P \vdash \mathit{meth} \in c}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS] $$\frac{\begin{array}{c} E = cn\ \texttt{this} \quad P \vdash c \\ P; E \vdash \mathit{field}_i \quad P; E \vdash \mathit{meth}_i \end{array}}{P \vdash \texttt{class}\ cn\ \texttt{extends}\ c\ \{\ \mathit{field}_{1\ldots j}\ \mathit{meth}_{1\ldots k}\}}$$

$\boxed{\vdash P}$

[PROG] $$\frac{\begin{array}{c} \mathit{ClassOnce}(P) \quad \mathit{WFClasses}(P) \quad \mathit{FieldsOnce}(P) \\ \mathit{MethodsOnce}(P) \quad \mathit{OverridesOK}(P) \\ P = \mathit{defn}_{1\ldots n}\ e \quad P \vdash \mathit{defn}_i \quad P; \emptyset \vdash e : t\ \&\ \texttt{cmpd} \end{array}}{\vdash P}$$

Figure 2: The typing rules.

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

[29] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *SOSP 87: Symposium on Operating Systems Principles*, pages 111–122, 1987.

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

[31] N. H. Minsky. Towards alias-free pointers. In *ECOOP 96: European Conference for Object-Oriented Programming*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer-Verlag, 1996.

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

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

[34] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[35] S. Savage, M. Burrows, C. G. Nelson, P. Sobalvarro, and T. A. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 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 *SPIN 00: Workshop on Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 224–244. Springer-Verlag, 2000.

[38] T. W. Doeppner, Jr. Parallel program correctness through refinement. In *POPL 77: Principles of Programming Languages*, pages 155–169. ACM Press, 1977.

[39] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *LICS 92: Logic in Computer Science*, pages 162–173. IEEE Computer Society Press, 1992.

[40] C. von Praun and T. Gross. Object-race detection. In *OOPSLA 01: Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82. ACM Press, 2001.

APPENDIX

A. THE TYPE SYSTEM

This appendix presents the type system described in Section 3. We define the following predicates informally, based on similar predicates in [21].

Predicate	Meaning
$\mathit{ClassOnce}(P)$	no class is declared twice in P
$\mathit{WFClasses}(P)$	there are no cycles in the class hierarchy
$\mathit{FieldsOnce}(P)$	no class contains two fields with the same name, either declared or inherited
$\mathit{MethodsOnce}(P)$	no class contains two declared methods with the same name
$\mathit{OverridesOK}(P)$	overriding methods have the same atomicity, return type, parameter types, and `requires` set as the overridden method

A typing environment is defined as

$$E \quad ::= \quad \emptyset \;\mid\; E, \mathit{arg}$$

We define the type system using the following judgments and the typing rules in Figure 2. We use the notation [`write_`]`guarded_by` to denote either `write_guarded_by` or `guarded_by`.

Judgment	Meaning
$P \vdash t$	t is a well-formed type
$P \vdash s <: t$	s is a subtype of t
$P \vdash E$	E is a well-formed typing environment
$P; E \vdash a$	a is a well-formed atomicity
$P; E \vdash e : t \;\&\; a$	expression e has type t and atomicity a
$P; E \vdash \mathit{field}$	field is a well-formed field
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P \vdash \mathit{field} \in c$	class c declares/inherits field
$P \vdash \mathit{meth} \in c$	class c declares/inherits meth
$P \vdash \mathit{defn}$	defn is a well-formed class definition
$\vdash P$	program P is well-formed