Types for Atomicity: Static Checking and Inference for Java

CORMAC FLANAGAN
University of California at Santa Cruz
STEPHEN N. FREUND and MARINA LIFSHIN
Williams College
and
SHAZ QADEER
Microsoft Research

20

Atomicity is a fundamental correctness property in multithreaded programs. A method is atomic if, for every execution, there is an equivalent serial execution in which the actions of the method are not interleaved with actions of other threads. Atomic methods are amenable to sequential reasoning, which significantly facilitates subsequent analysis and verification.

This article presents a type system for specifying and verifying the atomicity of methods in multithreaded Java programs using a synthesis of Lipton's theory of reduction and type systems for race detection. The type system supports guarded, write-guarded, and unguarded fields, as well as thread-local data, parameterized classes and methods, and protected locks. We also present an algorithm for verifying atomicity via type inference.

We have applied our type checker and type inference tools to a number of commonly used Java library classes and programs. These tools were able to verify the vast majority of methods in these benchmarks as atomic, indicating that atomicity is a widespread methodology for multithreaded programming. In addition, reported atomicity violations revealed some subtle errors in the synchronization disciplines of these programs.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics, and syntax*

C. Flanagan's work was supported by a Fellowship from the Alfred P. Sloan Foundation, by the National Science Foundation under Grant 0341179, and by faculty research funds granted by the University of California, Santa Cruz. S. Freund's and M. Lifshin's work was supported by the National Science Foundation under Grants 0306486, 0341387, and 0644130, and by research funds granted by Williams College.

Author's addresses: C. Flanagan: Department of Computer Science, University of California at Santa Cruz, Santa Cruz, CA 95064. S. Freund, M. Lifshin: Department of Computer Science, Williams College, Williamstown, MA 012667. S. Qadeer: One Microsoft Way, Redmond, WA 98052.

General Terms: Languages, Verification, Reliability

Additional Key Words and Phrases: Atomicity, concurrent programs, type systems, type inference

ACM Reference Format:

Flanagan, C., Freund, S. N., Lifshin, M., and Qadeer, S. 2008. Types for atomicity: Static checking and inference for Java. ACM Trans. Program. Lang. Syst. 30, 4, Article 20 (July 2008), 53 pages. DOI = 10.1145/1377492.1377495 http://doi.acm. org/10.1145/1377492.1377495

1. INTRODUCTION

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

Much previous work on controlling thread interference has focused on *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 [Savage et al. 1997]. In practice, race conditions are commonly avoided by protecting each data structure with a lock [Birrell 1989]. This lock-based synchronization discipline is supported by a variety of type systems [Flanagan and Freund 2000; Flanagan and Abadi 1999b, 1999a; Abadi et al. 2006; Boyapati and Rinard 2001; Boyapati et al. 2002; Grossman 2003] and other static [Sterling 1993; Flanagan et al. 2002; Chamillard et al. 1996; Corbett 1996] and dynamic [Savage et al. 1997; Choi et al. 2002; von Praun and Gross 2003; O'Callahan and Choi 2003; Pozniansky and Schuster 2003] analyses.

Unfortunately, the absence of race conditions is not sufficient to ensure the absence of errors due to unexpected interference between threads. As a concrete illustration of this limitation, consider the following bank account class:

```
class Account {
  int balance = 0;
  synchronized int read() { return balance; }
  synchronized void set(int b) { balance = b; }
  void deposit(int amt) {
    int b = read();
    set(b + amt);
  }
}
```

This class does not suffer from race conditions, a property that can be verified with existing tools such as `rccjava` [Abadi et al. 2006]. However, the method `deposit` may still behave incorrectly due to interactions between concurrent threads. In particular, if n calls to `deposit(1)` are interleaved, then the overall effect may be to increase `balance` by any number between 1 and n.

Recent results have shown that subtle defects of a similar nature are common, even in well-tested libraries [Flanagan and Qadeer 2003b; Flanagan et al.

2005]. Artho et al. [2003] report finding similar errors in NASA's Remote Agent spacecraft controller, and Burrows and Leino [2002] and von Praun and Gross [2003] have detected comparable defects in Java applications.

This paper focuses on the stronger noninterference property of *atomicity*. A method is atomic if any interaction between that method and other threads is guaranteed to be benign, in the sense that these interactions do not change the program's overall behavior. That is, 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. 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.

Atomicity corresponds to a natural programming methodology, essentially dating back to Hoare's monitors[1] [Hoare 1974]. The `Account` methods above are all intended to be atomic, as are many existing classes and library interfaces. For example, the documentation for the class `java.lang.StringBuffer` in JDK 1.4.0 [JavaSoft 2005] states:

> 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.

Atomicity provides a strong, indeed maximal, guarantee of noninterference between threads. This guarantee reduces the challenging problem of reasoning about an atomic method's behavior in a *multithreaded* context to the simpler problem of reasoning about the method's *sequential* behavior. The latter problem is significantly more amenable to standard techniques such as manual code inspection, dynamic testing, and static analysis.

We present a type system for specifying and checking atomicity properties of methods in multithreaded programs. Any method can be annotated with the keyword `atomic`. The type system checks that for any (arbitrarily interleaved) execution, there is a corresponding execution with equivalent behavior in which the instructions of each atomic method are executed serially.[2] The type system is a synthesis of our earlier type systems for race freedom [Flanagan and Abadi 1999a, 1999b; Flanagan and Freund 2004b; Abadi et al. 2006] and Lipton's theory of reduction [Lipton 1975]. It supports features such as parameterized classes and methods, thread-local data, and conditional atomicities. The type system relies on type annotations specifying

[1]Monitors are less general in that they rely on syntactic scope restrictions and do not support dynamically allocated shared data.

[2]For simplicity, we assume a sequentially-consistent memory model throughout this paper. Although we believe that our techniques extend to non-sequentially-consistent models, doing so would add further complexity to our formal development. We discuss the issue of memory models further in Section 8.3.

(1) the protecting lock of every field in the program and
(2) the atomicity of every method.

Given such a specification, our analysis checks that the implementation of the program conforms to it.

Expressing our atomicity analysis as a type system in this way offers several key benefits. Type checking is modular and more scalable to large programs than model-checking or whole-program analyses. Atomicity specifications also serve as useful and verifiable documentation of a program's synchronization requirements. Moreover, the type system can be extended to uniformly handle additional locking idioms, such as locks protecting other locks, as we illustrate below.

We also present a type inference algorithm for inferring these annotations. The *Rcc/Sat* subroutine, described in an earlier paper [Flanagan and Freund 2004b], infers the first class of annotations, those describing the protecting lock of each field. Here, we focus on the second class and present an algorithm to infer the most precise atomicity for each method. Our type inference algorithm is essentially a constraint-based analysis, but it is quite subtle, since the type system supports *conditional atomicities* that contain lock expressions, and thus we have a form of *dependent effects*. For soundness, the values of expressions embedded inside these conditional atomicities must not change during execution. Our constraint language includes special constructs to describe the well-formedness requirements on dependent atomicities, and the solver refers to judgments in the type system to enforce these requirements. Despite this complex interaction between the type system and constraint solver, the constraints can be solved with an iterative fixed-point algorithm.

We have implemented our type checking and type inference algorithms for the full Java programming language [Gosling et al. 1996], and we have evaluated this checker, named *Bohr*, on a variety of benchmarks totaling over 60,000 lines of code. The type inference algorithm is fast and works well in practice. Although the type system is necessarily incomplete, it has proved sufficiently expressive to accommodate the majority of synchronization patterns present in our benchmark programs and to verify the atomicity of most nonerroneous methods. These experimental results validate the hypothesis that atomicity is a widely used programming discipline in multithreaded programs, and they show that reported atomicity violations often reveal subtle errors in a program's synchronization discipline, including, for example, errors in the standard library classes `java.util.Vector` and `java.lang.StringBuffer`.

The following section presents an informal introduction to type-based atomicity checking. Sections 3 and 4 then formalize that approach for a small, multithreaded subset of Java. Sections 5 illustrates some atomicity violations caught using this type system. We then turn our attention to type inference in Section 6 and describe our inference tool for Java in Section 7. The results of applying this tool to various benchmarks is summarized in Section 8. Section 9 describes related work, and we conclude with Section 10. The online appendix available in the ACM Digital Library contains the full details of our formal development, including correctness proofs.

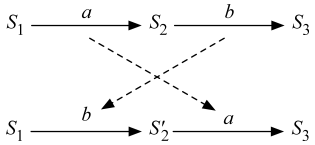
This article is based on work presented in preliminary form at conferences and workshops [Flanagan and Qadeer 2003c, 2003b; Flanagan and Freund 2004b; Flanagan et al. 2005]. Some of that work focused on imperative calculi (rather than Java) and consequently used variations of the atomicity analysis presented here. This article presents the contributions of the earlier papers in a unified framework that permits us to, for example, show that the type system is sound and that type inference is sound and complete with respect to the type system. We also discuss additional language features, simplify various aspects of our previous work, and present the technical development in more detail.

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 not 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 [1975], to prove the correctness of atomicity annotations.

An action a is a *right mover* if for any execution where the action a performed by one thread is immediately followed by an action b of a different thread, the actions a and b can be swapped without changing the resulting state S_3, as shown in the following. 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 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.

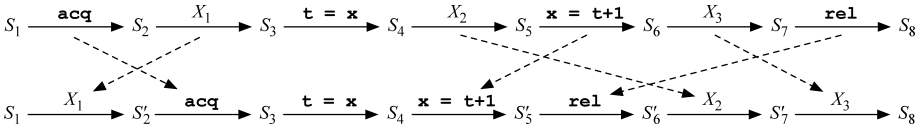
Similarly, consider an action a of one thread that is immediately followed by a lock release operation by a second thread. During a, the second thread holds the lock, and a can neither acquire nor release the lock. Hence the lock release operation can be moved to the left of a without changing the resulting 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 enforces this access restriction, no two threads may access the field at the same time, and therefore every access to the field is both a right mover and a left mover.

To illustrate how the theory of movers enables us to verify atomicity, consider the following Java method:

```
synchronized void inc() {
  int t = x;
  x = t + 1;
}
```

This method (1) acquires the lock on `this` (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 X_1, X_2, and X_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 commuting diagram. Thus the method is atomic.[3]



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 here the method is executed serially without any interleaved actions by other threads. Therefore, an `atomic` annotation on such a method is valid.

3. ATOMICJAVA

We base our formal development on the language AtomicJava, a multithreaded subset of Java with a type system for atomicity. This type system extends our previous atomicity type system [Flanagan and Qadeer 2003c, 2003b] to include support for thread-local objects and parameterized classes and methods. We previously explored these features in a type system for race freedom [Flanagan and Freund 2000, 2004b]. For clarity, AtomicJava also simplifies some aspects of our earlier formal development by, for example, not supporting inheritance. (Section 7 describes how our implementation handles inheritance and other aspects of the full Java programming language.)

An AtomicJava program is a sequence of class declarations together with an initial expression. (See Figure 1.) Each class declaration associates a class name with a body that consists of a sequence of field and method declarations. The self-reference variable "`this`" is implicitly bound within the class body, so that it can be referred to both within method bodies and within method and field types.

[3]In general, if a program is race-free and only acquires one lock at a time, all `synchronized` methods would be atomic. However, these requirements are rarely satisfied, and a key benefit of our analysis is that it can handle programs that do not satisfy them.

$$
\begin{array}{rcll}
P & ::= & \mathit{defn}^*\ e & \text{(program)} \\
\mathit{defn} & ::= & \texttt{class}\ cn\langle\texttt{ghost}\ x^*\rangle\ \mathit{body} & \text{(class declaration)} \\
\mathit{body} & ::= & \texttt{\{}\ \mathit{field}^*\ \mathit{meth}^*\ \texttt{\}} & \text{(class body)} \\
\mathit{field} & ::= & t\ fd\ g & \text{(field declaration)} \\
g & ::= & \texttt{final} & \text{(field guards)} \\
 & | & \texttt{guarded\_by}\ l & \\
 & | & \texttt{write\_guarded\_by}\ l & \\
 & | & \texttt{no\_guard} & \\
\mathit{meth} & ::= & a\ t\ md\langle\texttt{ghost}\ x^*\rangle\texttt{(}\mathit{arg}^*\texttt{)\ \{}\ e\ \texttt{\}} & \text{(method declaration)} \\
\mathit{arg} & ::= & t\ x & \text{(argument declaration)} \\
c & ::= & cn\langle l^*\rangle & \\
t & ::= & c \mid \texttt{int} \mid \texttt{long} & \text{(type)} \\
l & ::= & e & \text{(lock expression)} \\
 & & & \\
e & ::= & v & \text{(value)} \\
 & | & \texttt{new}_y\ c\texttt{(}e^*\texttt{)} & \text{(object allocation)} \\
 & | & e.fd & \text{(field access)} \\
 & | & e.fd\ \texttt{=}\ e & \text{(field update)} \\
 & | & e.md\langle l^*\rangle\texttt{(}e^*\texttt{)} & \text{(method call)} \\
 & | & \texttt{let}\ \mathit{arg}\ \texttt{=}\ e\ \texttt{in}\ e & \text{(variable binding)} \\
 & | & \texttt{while}\ e\ e & \text{(iteration)} \\
 & | & \texttt{if}\ e\ e\ e & \text{(conditional)} \\
 & | & \texttt{sync}\ l\ e & \text{(synchronization)} \\
 & | & e\texttt{.fork} & \text{(fork)} \\
 & | & \texttt{assert-atomic}\ e & \text{(assert atomic)} \\
 & & & \\
v & ::= & x & \text{(variable)} \\
 & | & \texttt{null} & \text{(null)} \\
 & | & n & \text{(number)}
\end{array}
$$

$$
\begin{array}{rclrcl}
x, y & \in & \mathit{Var} & cn & \in & \mathit{ClassName} \\
fd & \in & \mathit{FieldName} & md & \in & \mathit{MethodName} \\
a & \in & \mathit{Atomicity} & n & \in & \mathit{Numbers}
\end{array}
$$

Fig. 1. AtomicJava Syntax.

Each field declaration includes a *guard* g that specifies the synchronization discipline for that field. The possible guards are:

—`final`: the field cannot be written after initialization;

—`guarded_by` l: the lock denoted by the *lock expression* l must be held on all accesses (reads or writes) of that field;

—`write_guarded_by` l: the lock denoted by the lock expression l must be held on all writes of that field, but not for reads; and

—`no_guard`: the field can be read or written at any time.

A lock expression is an expression that denotes some lock in the program. To ensure soundness, lock expressions are well-formed only if they denote a fixed lock throughout program execution, and so, for example, they cannot access mutable fields.

The guard `no_guard` describes fields on which there are intentional race conditions. If all such fields are marked with the Java keyword `volatile`, then we

believe that our type system applies to the Java memory model [Manson et al. 2005], even though it is not sequentially-consistent.

The language provides *parameterized classes* to allow the fields of a class to be protected by some lock external to the class. A parameterized class declaration

$$\texttt{class}\ cn\langle\texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots\}$$

introduces a binding for the *ghost* variables $x_1 \ldots x_n$, which can be referred to from type annotations within the class body. The type $cn\langle l_1 \ldots l_n\rangle$ refers to an *instantiated version* of cn, where each x_i in the body is replaced by the lock expression l_i.

The AtomicJava language also supports *parameterized methods*. For example, the declaration

$$a\ t\ m\langle\texttt{ghost}\ x\rangle(cn\langle x\rangle\ y)\ \{\ \ldots\ \}$$

defines a method m of return type t that is parameterized by a ghost lock x and takes an argument of type $cn\langle x\rangle$. A corresponding invocation $e.m\langle z\rangle(e')$ must supply a ghost argument z and an actual parameter e' of type $cn\langle z\rangle$.

Each method declaration includes a specification a of the method's atomicity. The language of atomicities includes the keyword `atomic`, as well as more precise characterizations of method behavior, as described in the following section. Here, we just note that the atomicity a may refer to program variables in scope, including `this`, the ghost parameters of the containing class, and the ghost and normal parameters of the method itself.

The object allocation expression $\texttt{new}_y\ c(e^*)$ includes a sequence e^* of expressions used to initialize the object fields. For technical reasons, the `new` keyword is subscripted by y, which is a ghost variable bound to the object being created while evaluating the field initialization expressions. This enables the types of the initialization expressions to refer to the new object. We omit this binding from examples when it is not needed.

Other expressions in the language include field read and update, method calls, variable binding and reference, conditionals, loops, and synchronized blocks. We include basic types for both single-word `int`s and double-word `long`s. Only reads and writes of the former are atomic. Reads and writes of object references are also atomic.

As in Java, each object has an associated mutual exclusion lock that is initially unlocked. The expression $\texttt{sync}\ l\ e$ is evaluated in a manner similar to Java's `synchronized` statement: the subexpression l is evaluated first, and should yield an object whose lock is then acquired; the subexpression e is then evaluated; and finally the lock is released. The result of e is returned as the result of the synchronized expression. 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 expression e.`fork` starts a new thread (and always evaluates to 0). The expression e should evaluate to an object that includes a method `run` taking a single ghost parameter. The fork operation spawns a new thread that, conceptually, creates and acquires a new *thread-local lock* `tll` for instantiating the ghost parameter to the method `run`. This lock is always held by the new thread and may therefore be used by `run` to guard thread-local data, and it may be

passed as a ghost parameter to other methods that access thread-local data. Thus, AtomicJava leverages parameterized methods to reason about thread-local data. This approach replaces the escape analysis embedded in our earlier type system [Flanagan and Freund 2000].

The expression `assert-atomic` e specifies that e should be serializable with respect to the rest of the system. The AtomicJava type system ensures that this requirement is satisfied.

We present example programs in an extended language with additional integer and boolean constants and operations. The sequential composition $e_1; e_2$ abbreviates `let` x `=` e_1 `in` e_2, where x does not occur free in e_2, and 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.

4. TYPES FOR ATOMICITY

4.1 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 [Talpin and Jouvelot 1992], or effect [Lucassen and Gifford 1988], of that expression. The set of atomicities includes the following *basic atomicities*:

—`const`: The atomicity `const` describes any expression whose 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`: The atomicity `mover` describes any expression that 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.[4]

—`atomic`: The atomicity `atomic` describes any expression that is a single atomic action, or that can be considered to execute without interleaved actions of other threads.

—`cmpd`: The atomicity `cmpd` describes a compound expression for which none of the preceding atomicities apply.

—`error`: The atomicity `error` describes any expression violating the locking discipline specified by the type annotations.

If the basic atomicity b reflects the behavior of an expression e, then the *iterative closure* b^* reflects the behavior of executing e an arbitrary number of

[4]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.

times. Similarly, if basic atomicities b_1 and b_2 reflect the behavior of e_1 and e_2 respectively, then the *sequential composition* $b_1; b_2$ reflects the behavior of $e_1; e_2$. These iterative closure and sequential composition operations are defined as follows:

b	b^*
const	const
mover	mover
atomic	cmpd
cmpd	cmpd
error	error

;	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 b_1 and b_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 $b_1 \sqcup b_2$.

4.2 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 it has atomicity `error` otherwise. We assign such an access the *conditional atomicity*:

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

A conditional atomicity $l\,?\,a_1 : a_2$ is equivalent to atomicity a_1 if the lock l is currently held, and it is equivalent to atomicity a_2 if the lock is not held. Conditional atomicities provide a more precise characterization of the behavior of synchronized statements and methods. The set of atomicities thus includes both the basic atomicities described above and conditional atomicities:

$$\begin{aligned} b &::= \texttt{const} \mid \texttt{mover} \mid \texttt{atomic} \mid \texttt{cmpd} \mid \texttt{error} \\ a &::= b \mid l\,?\,a_1 : a_2 \end{aligned}$$

Each atomicity a is equivalent to a function $(\!|a|\!)$ from the set of currently held locks ls to a basic atomicity:

$$\begin{aligned} (\!|b|\!)(ls) &= b \\ (\!|l\,?\,a_1 : a_2|\!)(ls) &= \begin{cases} (\!|a_1|\!)(ls) & \text{if } l \in ls \\ (\!|a_2|\!)(ls) & \text{if } l \notin ls \end{cases} \end{aligned}$$

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:

$$\begin{aligned}
(l\,?\,a_1 : a_2)^* &= l\,?\,a_1^* : a_2^* \\
(l\,?\,a_1 : a_2); a &= l\,?\,(a_1; a) : (a_2; a) \\
a; (l\,?\,a_1 : a_2) &= l\,?\,(a; a_1) : (a; a_2) \\
(l\,?\,a_1 : a_2) \sqcup a &= l\,?\,(a_1 \sqcup a) : (a_2 \sqcup a) \\
a \sqcup (l\,?\,a_1 : a_2) &= l\,?\,(a \sqcup a_1) : (a \sqcup a_2)
\end{aligned}$$

These operations on conditional atomicities are the point-wise extensions of the corresponding operations on basic atomicities:

THEOREM 1. *For all atomicities a_1 and a_2 and all locksets ls:*

$$\begin{aligned}
(\!|a_1^*|\!)(ls) &= ((\!|a_1|\!)(l\,s))^* \\
(\!|a_1; a_2|\!)(ls) &= (\!|a_1|\!)(ls); (\!|a_2|\!)(ls) \\
(\!|a_1 \sqcup a_2|\!)(ls) &= (\!|a_1|\!)(ls) \sqcup (\!|a_2|\!)(ls)
\end{aligned}$$

We also extend the subatomicity ordering to conditional atomicities. To decide $a_1 \sqsubseteq a_2$, we use an auxiliary relation $\sqsubseteq_n^h$, where h is a set of locks known to be *h*eld by the current thread, and n is a set of locks known *n*ot to be held by the current thread. Intuitively, the condition $a_1 \sqsubseteq_n^h a_2$ holds if and only if $(\!|a_1|\!)(ls) \sqsubseteq (\!|a_2|\!)(ls)$ holds for every lockset ls that contains h and is disjoint from n. We define $a_1 \sqsubseteq a_2$ to be $a_1 \sqsubseteq_\emptyset^\emptyset a_2$ and check $a_1 \sqsubseteq_n^h a_2$ recursively as follows:

$$\frac{b_1 \sqsubseteq b_2}{b_1 \sqsubseteq_n^h b_2} \qquad \frac{l \notin n \Rightarrow a_1 \sqsubseteq_n^{h \cup \{l\}} a \quad l \notin h \Rightarrow a_2 \sqsubseteq_{n \cup \{l\}}^h a}{l\,?\,a_1 : a_2 \sqsubseteq_n^h a} \qquad \frac{l \notin n \Rightarrow b \sqsubseteq_n^{h \cup \{l\}} a_1 \quad l \notin h \Rightarrow b \sqsubseteq_{n \cup \{l\}}^h a_2}{b \sqsubseteq_n^h l\,?\,a_1 : a_2}$$

The subatomicity ordering on conditional atomicities is the point-wise extension of the ordering on basic atomicities:

THEOREM 2. *For all atomicities a_1 and a_2:*

$$a_1 \sqsubseteq a_2 \;\Leftrightarrow\; \forall ls.\ (\!|a_1|\!)(ls) \sqsubseteq (\!|a_2|\!)(ls)$$

Atomicities a_1 and a_2 are *equivalent*, written $a_1 \equiv a_2$, if $a_1 \sqsubseteq a_2$ and $a_2 \sqsubseteq a_1$. If $a_1 \equiv a_2$, then $\forall ls.\ (\!|a_1|\!)(ls) = (\!|a_2|\!)(ls)$. 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 a number of useful ordering and equivalence properties for atomicities.

THEOREM 3. *For all atomicities a, a_1, a_2, and a_3:*

(1) *Iterative closure is monotonic and idempotent.*

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

(2) *Sequential composition is monotonic and associative and has* `const` *as a left and right identity.*

$$\begin{aligned} a_1 &\sqsubseteq a_1;a_2 \\ a_2 &\sqsubseteq a_1;a_2 \\ (a_1;a_2);a_3 &\equiv a_1;(a_2;a_3) \\ \texttt{const};a &\equiv a \\ a;\texttt{const} &\equiv a \end{aligned}$$

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

$$\begin{aligned} a_1;(a_2 \sqcup a_3) &\equiv a_1;a_2 \sqcup a_1;a_3 \\ (a_1 \sqcup a_2);a_3 &\equiv a_1;a_3 \sqcup a_2;a_3 \\ (a_1 \sqcup a_2)^* &\equiv a_1^* \sqcup a_2^* \end{aligned}$$

4.3 List Example

To illustrate how atomicities capture the behavior of code fragments, consider the class `List` of Figure 2, which implements a linked list of `ListElems`. The extra type annotations are underlined.

The class `ListElem` is parameterized by a lock `x`, which protects the `num` and `next` fields, as indicated by the `guarded_by x` annotations. The method `ListElem.get` has conditional atomicity (`x ? mover : error`), which states that if the lock `x` is not held, then a call to `get` has atomicity `error`, because the call violates the program's synchronization discipline. If the lock `x` is held, then the method `get` is a `mover`, and its execution commutes with actions of concurrent threads.

The class `List` contains an `elems` field whose type is `ListElem⟨this⟩`, indicating that the implicit lock of the `List` object protects its `ListElems`. The method `List.get` consists of (1) a right mover (the lock acquire), (2) a both mover (the read of `this.elems`), (3) a second both mover (the call to `ListElem.get` with the lock held), and (4) a left mover (the lock release). Hence `List.get` is at most `atomic`. However, if the lock `this` is already held, the re-entrant locking operations are both movers, and so `List.get` is then a `mover`. Our type checker verifies that `List.get` satisfies the precise atomicity (`this ? mover : atomic`), and similarly for `List.add`.

The method `List.addPair` contains two nested calls to `this.add` and thus has atomicity:

$$(\texttt{this ? mover : atomic});(\texttt{this ? mover : atomic}) = \texttt{this ? mover : cmpd}$$

This inferred atomicity (`this ? mover : cmpd`) is inconsistent with the annotation declaring `List.addPair` as `atomic`, and the type checker reports an atomicity violation on this method. Note that, if the lock `this` is not held, the atomicity `cmpd` means that interleaved actions of concurrent threads may affect the behavior

```
class ListElem⟨ghost x⟩ {
  int num guarded_by x;
  ListElem⟨x⟩ next guarded_by x;

  (x?mover:error) int get() { return this.num; }
}

class List {
  ListElem⟨this⟩ elems guarded_by this;

  (this?mover:atomic) void add(int v) {
    sync(this) {
      this.elems = new ListElem⟨this⟩(v,this.elems);
    }
  }

  atomic void addPair(int i, int j) { // wrong, should be (this?mover:cmpd)
    this.add(i);
    this.add(j);
  }

  (this?mover:atomic) int get() {
    sync(this) {
      return this.elems.get();
    }
  }
}
```

Fig. 2. Class List with locking and atomicity annotations.

and correctness of addPair (even though there are no race-conditions). In particular, if a concurrent thread also adds entries to the list, then addPair will not achieve its intended behavior of adding its arguments to the list consecutively.

4.4 Type Rules

The AtomicJava type system ensures that all locking and atomicity specifications in a program are correct and that all expressions of the form `assert-atomic` e are serializable.

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

$$P; E \vdash e : t \cdot a$$

Here, t is the type inferred for the expression e, and a is the atomicity generated for e. The program P is included to provide access to class declarations, and E is an environment providing types for the free regular and ghost variables of the expression e:

$$E ::= \epsilon \mid E, t\ x \mid E, \texttt{ghost}\ x$$

The complete set of type rules for expressions appears in Figures 3 and 4. Type rules for various supporting judgments are shown in Figure 5. These judgments check the well-formedness of various entities, including: environments (via the

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

[EXP NULL]
$$\frac{P; E \vdash c}{P; E \vdash \texttt{null} : c \cdot \texttt{const}}$$

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

[EXP SYNC]
$$\frac{P; E \vdash_{\text{lock}} l \qquad P; E \vdash e : t \cdot a}{P; E \vdash \texttt{sync}\ l\ e : t \cdot S(l, a)}$$

[EXP INT]
$$\frac{P \vdash E}{P; E \vdash n : \texttt{int} \cdot \texttt{const}}$$

[EXP LONG]
$$\frac{P \vdash E}{P; E \vdash n : \texttt{long} \cdot \texttt{const}}$$

[EXP ASSERT]
$$\frac{P; E \vdash e : t \cdot a}{P; E \vdash \texttt{assert-atomic}\ e : t \cdot R(a)}$$

[EXP REF]
$$\frac{\begin{array}{c} P; E \vdash e : cn\langle l_{1..n}\rangle \cdot a' \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots t\ fd\ g \ldots\ \} \in P \\ \theta = [\texttt{this} := e, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash \theta(t) \\ (g \equiv \texttt{final}) \Rightarrow (a = \texttt{const}) \\ (g \equiv \texttt{guarded\_by}\ l) \Rightarrow (a = \theta(l)\,?\,\texttt{mover} : \texttt{error}) \\ (g \equiv \texttt{write\_guarded\_by}\ l) \Rightarrow (a = \theta(l)\,?\,\texttt{mover} : B(t)) \\ (g \equiv \texttt{no\_guard}) \Rightarrow (a = B(t)) \\ P; E \vdash a \uparrow a'' \end{array}}{P; E \vdash e.fd : \theta(t) \cdot (a'; a'')}$$

[EXP IF]
$$\frac{\begin{array}{c} P; E \vdash e_1 : \texttt{int} \cdot a_1 \\ P; E \vdash e_i : t \cdot a_i \qquad \forall i \in 2..3 \\ a = a_1; (a_2 \sqcup a_3) \end{array}}{P; E \vdash \texttt{if}\ e_1\ e_2\ e_3 : t \cdot a}$$

[EXP ASSIGN]
$$\frac{\begin{array}{c} P; E \vdash e_1 : cn\langle l_{1..n}\rangle \cdot a_1 \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots t\ fd\ g \ldots\ \} \in P \\ \theta = [\texttt{this} := e_1, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash e_2 : \theta(t) \cdot a_2 \\ (g \equiv \texttt{final}) \Rightarrow (a = \texttt{error}) \\ (g \equiv \texttt{guarded\_by}\ l) \Rightarrow (a = \theta(l)\,?\,\texttt{mover} : \texttt{error}) \\ (g \equiv \texttt{write\_guarded\_by}\ l) \Rightarrow (a = \theta(l)\,?\,B(t) : \texttt{error}) \\ (g \equiv \texttt{no\_guard}) \Rightarrow (a = B(t)) \\ P; E \vdash a \uparrow a' \end{array}}{P; E \vdash (e_1.fd = e_2) : \theta(t) \cdot (a_1; a_2; a')}$$

[EXP WHILE]
$$\frac{\begin{array}{c} P; E \vdash e_1 : \texttt{int} \cdot a_1 \\ P; E \vdash e_2 : t \cdot a_2 \\ a = a_1; ((a_2; a_1)^*) \end{array}}{P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{int} \cdot a}$$

Fig. 3. AtomicJava type rules (I).

judgment $P \vdash E$), types ($P; E \vdash t$), atomicities ($P; E \vdash a$), field declarations ($P; E \vdash \mathit{field}$), method declarations ($P; E \vdash \mathit{meth}$), class declarations ($P \vdash \mathit{defn}$), and programs ($P \vdash \mathit{wf}$). The judgment $P; E \vdash_{\text{lock}} e$ checks that e is a well-formed lock expression with respect to the given program and environment.

We briefly describe some of the more important rules defining these various judgments.

[EXP VAR] A variable access has `const` atomicity, since all variables are immutable in ATOMICJAVA. This rule retrieves the variable's type from the environment, after checking that that environment is well-formed.

[EXP IF] The atomicity of a conditional expression is the atomicity of the *test* subexpression, sequentially composed with the join of the atomicities of the *then* and *else* branches.

[EXP LET] This rule for $\texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$ infers atomicity expressions a_1 and a_2 for e_1 and e_2, respectively. Since the atomicity expression a_2 may refer to

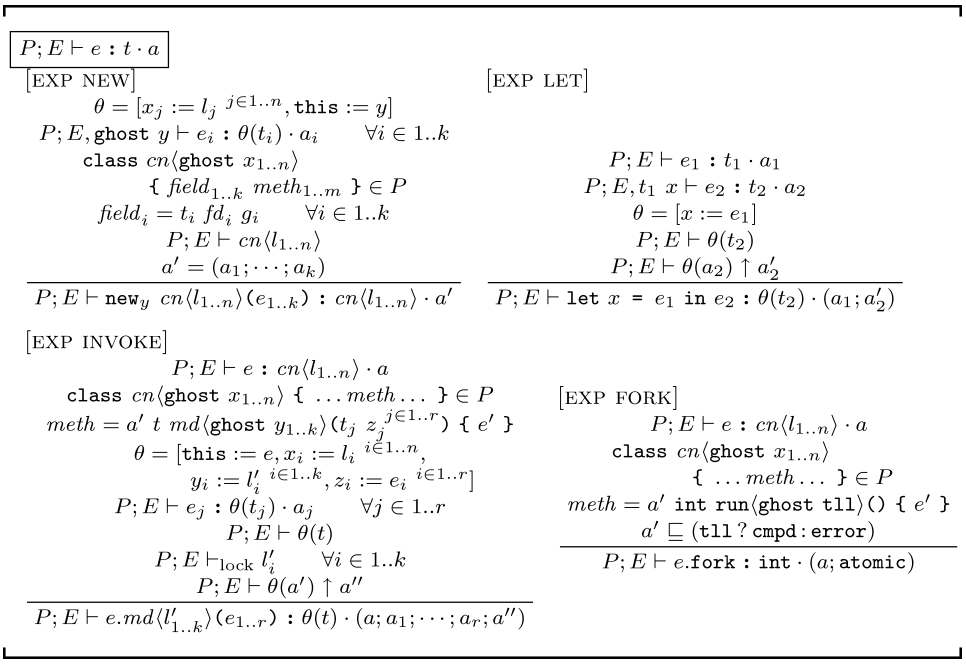


Fig. 4. AtomicJava type rules (II).

the let-bound variable x, we apply the substitution $\theta = [x := e_1]$ to yield a corresponding atomicity $\theta(a_2)$ that does not mention x.

However, e_1 may not have atomicity `const`, in which case $\theta(a_2)$ may not be a valid atomicity (because it could contain e_1 as part of a nonconstant lock expression). Therefore, we use the judgment $P; E \vdash \theta(a_2) \uparrow a_2'$ to *lift* the atomicity $\theta(a_2)$ to some well-formed atomicity a_2' that is greater than or equal to $\theta(a_2)$. This lifting judgment is defined by the following rules from Figure 5:

[LIFT BASE] Basic atomicities are always well-formed and remain unchanged when lifted.

[LIFT LOCK WELL-FORMED] If a conditional atomicity refers to a well-formed lock, this rule recursively lifts the two component atomicities.

[LIFT LOCK ILL-FORMED] If a conditional atomicity refers to an ill-formed lock, this rule removes the dependency on this lock by joining together the two recursively lifted component atomicities.

[EXP REF] The rule for a field read $e.fd$ first checks that e is of some type $cn\langle l_{1..n} \rangle$, and that cn is a class parameterized by ghost variables $x_{1..n}$ that declares a field fd of some type t. The type t may refer to the variables `this` and $x_{1..n}$, which are not in scope at the field access, and so the substitution θ replaces them with the corresponding expressions e and $l_{1..n}$. The type rule also ensures that $\theta(t)$ is a well-formed type, and then performs a case analysis on the field's guard:

—If the field is `final`, then the read operation has atomicity `const`, since there can be no concurrent writes.

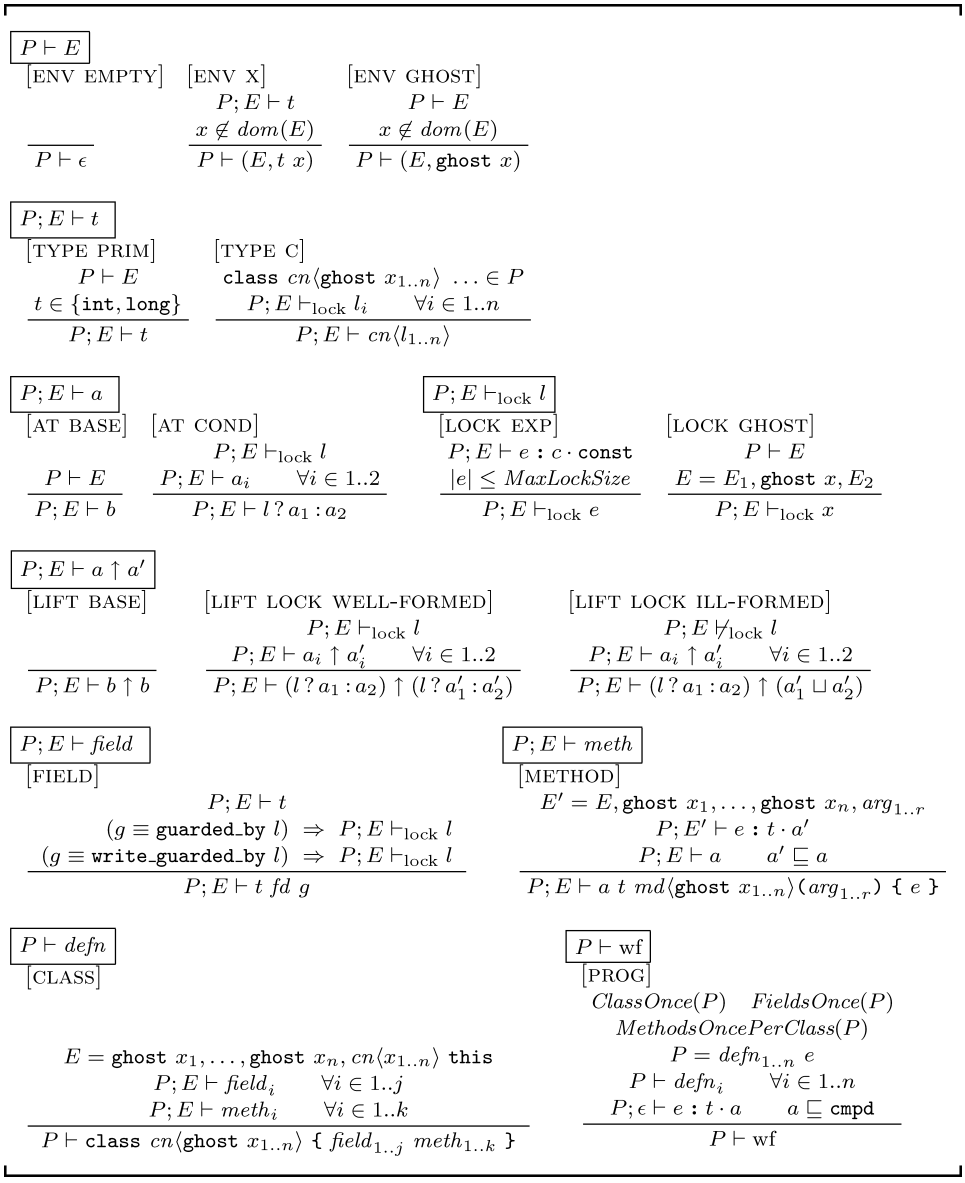


Fig. 5. AtomicJava type rules (III).

—If the field is `no_guard`, then the read operation has atomicity $B(t)$, where the function $B(t)$ yields the atomicity of a single unprotected read or write to a field of type t:

$$\begin{aligned} B(\texttt{int}) &= \texttt{atomic} \\ B(\texttt{long}) &= \texttt{cmpd} \\ B(c) &= \texttt{atomic} \end{aligned}$$

As in Java, integers and object references may be accessed atomically, but accesses to double-word `long`s have atomicity `cmpd`.

—If the field is `guarded_by` l, then the lock $\theta(l)$ must be held and the read operation has atomicity `mover`.
—If the field is `write_guarded_by` l, then the read operation has atomicity `mover` if the lock $\theta(l)$ is held, and it has atomicity $B(t)$ if the lock is not held.

As in rule [EXP LET], we ensure that the resulting atomicity is well-formed in E using the atomicity lifting judgment.

[EXP ASSIGN] The rule for field update is similar to the rule for field reads, with two notable differences. A write to a final field is assigned the atomicity `error`, indicating an error. A write to a write-guarded field when the appropriate lock is not held is also an error. Otherwise, such a write is assigned the atomicity $B(t)$ since it could occur concurrently with reads from other threads.

[EXP SYNC] The rule for the synchronized statement `sync` l e checks that l has atomicity `const`, and so always denotes the same lock. The rule then yields the atomicity expression $S(l, a)$, where a is the atomicity of e.

The function S, which we will define in the following, 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 both mover (the body), followed by a left mover (the release). If the body has conditional atomicity $l\,?\,a_1 : a_2$, then, since l is held within the synchronized body, we ignore a_2 and recursively apply S to a_1. If the body has some other conditional atomicity, then we recursively apply S to both branches.

$$\begin{aligned}
S(l, \texttt{const}) &= l\,?\,\texttt{const} : \texttt{atomic} \\
S(l, \texttt{mover}) &= l\,?\,\texttt{mover} : \texttt{atomic} \\
S(l, \texttt{atomic}) &= \texttt{atomic} \\
S(l, \texttt{cmpd}) &= \texttt{cmpd} \\
S(l, \texttt{error}) &= \texttt{error} \\
S(l, (l\,?\,a_1 : a_2)) &= S(l, a_1) \\
S(l, (l'\,?\,a_1 : a_2)) &= l'\,?\,S(l, a_1) : S(l, a_2) \ \text{ if } l \neq l'
\end{aligned}$$

[EXP NEW] The rule [EXP NEW] for an object creation expression $\texttt{new}_y\ cn\langle l_{1..n}\rangle(e_{1..k})$ first retrieves the corresponding class declaration

$$\texttt{class}\ cn\langle\texttt{ghost}\ x_{1..n}\rangle\ \{\ field_{1..k}\ meth_{1..m}\ \}$$

from P. The substitution $\theta = [x_j := l_j\ ^{j \in 1..n}, \texttt{this} := y]$ replaces the ghost parameters $x_{1..n}$ with the actual arguments $l_{1..n}$, and replaces occurrences of the self reference `this` with y. In effect, this rule uses the name y as a placeholder for the object that is about to be constructed. The rule checks that each expression e_i has the appropriate type $\theta(t_i)$, where t_i is the type of $field_i$, since in AtomicJava the arguments e_i are used to directly initialize these fields.

[EXP INVOKE] The rule [EXP INVOKE] for a method invocation expression $e.md\langle l'_{1..k}\rangle(e_{1..d})$ is similar to field access, but it is slightly more complex

due to the presence of both regular and ghost method parameters. The rule checks that e has some type $cn\langle l_{1..n}\rangle$ and that cn includes a matching method declaration

$$a'\ t\ md\langle \texttt{ghost}\ y_{1..k}\rangle(t_j\ z_j^{\ j\in 1..r})\ \{\ e'\ \}$$

The rule then constructs the substitution

$$\theta = [\texttt{this} := e, x_i := l_i^{\ i\in 1..n}, y_i := l'^{\ i\in 1..k}_i, z_i := e_i^{\ i\in 1..r}]$$

which substitutes
—the receiver's name e for `this`;
—the lock expressions $l_{1..n}$ for the class' ghost parameters $x_{1..n}$;
—the lock expression arguments $l'_{1..k}$ for the method's ghost parameters $y_{1..k}$; and
—the method arguments $e_{1..r}$ for the method's formal parameters $z_{1..r}$.
Each method argument e_j must have type $\theta(t_j)$, and the return type $\theta(t)$ must be well-formed. The atomicity of the call is the atomicity of each argument sequentially composed with the atomicity a' of the invoked method.

[EXP FORK] The expression e.`fork` creates a new thread. As such, e must be an object supporting an appropriate `run` method that expects a single ghost parameter `tll` for the thread-local lock. The newly spawned thread must not violate the program's locking discipline. Since the new thread implicitly acquires the thread-local lock `tll` before executing `run`, requiring the `run` to have atomicity at most `tll ? cmpd : error` is sufficient to ensure this.
The fork operation is itself an atomic operation, making the atomicity of e.`fork` be a;`atomic`, where a is the atomicity of e.

[EXP ASSERT] The rule for `assert-atomic` e ensures that the atomicity a of the expression e is at most `atomic`. However, we cannot simply check that $a \sqsubseteq$ `atomic`, since we must also consider what locks will be held when e is evaluated.
Instead, we enforce the requirement that the atomicity of e is never `cmpd`, by replacing all occurrences of `cmpd` in a with `error` using the function $R(a)$:

$$\begin{aligned} R(\texttt{cmpd}) &= \texttt{error} \\ R(b) &= b \qquad\qquad \text{if } b \neq \texttt{cmpd} \\ R(l'\,?\,a_1 : a_2) &= l\,?\,R(a_1) : R(a_2) \end{aligned}$$

To motivate how this rule enforces atomicity, suppose $a = l\,?\,\texttt{mover} : \texttt{cmpd}$. Then $R(a)$ is $l\,?\,\texttt{mover} : \texttt{error}$, which requires that the lock l is held whenever e is evaluated, or else e would exhibit non-atomic behavior.

[LOCK GHOST] and [LOCK EXP] The judgment $P; E \vdash_{\text{lock}} l$ checks that l is a well-formed lock expression in environment E. The lock expression l can be either a ghost variable or a program expression e. In the latter case, e must denote a fixed lock throughout the execution of the program to ensure soundness. Thus, we require that e has atomicity `const`.
In addition, each lock expression e has a size $|e|$, which is the number of field accesses it contains. To ensure termination of the type inference algorithm presented in the second half of this article, we require that the size of each

lock expression is bounded by the constant *MaxLockSize*. The size restriction poses no limitation in practice, because lock relationships are never complex enough to be problematic, even with a relatively small *MaxLockSize*.[5]

[PROG] This rule defines the top-level judgment $P \vdash wf$ stating that P is a well-formed program, based on the following additional predicates. (See Flatt et al. [1998] for their precise definition.)
—*ClassOnce*(P): no class is declared twice in P.
—*FieldsOnce*(P): no field name is declared twice in a class.
—*MethodsOncePerClass*(P): no method name is declared twice in a class.
This rule guarantees that the locking discipline for the program is followed by asserting that the atomicity of the main thread is at most `cmpd`.

We prove the type system is sound in the online appendix available on the ACM Digital Library and show that all code blocks appearing inside `assert-atomic` blocks are serializable.

5. EXAMPLES

This section describes our initial experience with the AtomicJava type system. Our prototype checker, *Bohr*, extends the type system outlined so far to handle the additional features of the full Java programming language, including inheritance, interfaces, subtyping constructors, static fields and methods, inner classes, and so on, as discussed in Section 7. 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. If a method body's atomicity does not match the declared atomicity of the method, an appropriate error message is produced. *Bohr* allows class declarations to be annotated as `atomic` to indicate that all methods in the class should be atomic.

In practice, programs use a variety of synchronization mechanisms, not all of which can be captured by our type rules. *Bohr* 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, such as when a particular race condition is considered benign.
—the `holds` annotation permits the checker to assume that a particular lock is held from the current program point to the end of the current statement block.
—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. This assumption eliminates a large number of spurious warnings and is valid for the benchmark programs examined, but may be violated by other classes, including some in the standard Java libraries [Stoller 2006]. We believe this command line flag could be replaced with a sound escape

[5]Setting *MaxLockSize* to be the size of the program is sufficient for any fully-annotated program, but setting it to be 4 or 6 has been sufficient for all the programs examined to date.

Table I. Manually Annotated Classes

Class	LOC	Annotations per KLOC			
		Total	Guard	Atomicity	Escapes
java.lang.String	2,307	3.0	1.3	1.3	0.4
java.util.StringBuffer	1,276	7.8	2.3	5.5	0
java.util.Vector	1,021	16.6	7.8	4.9	3.9
java.util.zip.Inflater	319	18.8	15.7	3.1	0
java.util.zip.Deflater	384	23.4	20.8	2.6	0
java.io.PrintWriter	738	48.8	4.1	39.3	5.4
java.net.URL	1,189	29.4	13.5	10.9	5.0
java.lang.String (1.4.0)	2,351	3.0	1.3	1.3	0.4
java.net.URL (1.4.0)	1,231	27.6	13.0	9.7	4.9

analysis [Choi et al. 1999; Salcianu and Rinard 2001] without significant reduction in the expressiveness of the system.

We used *Bohr* to check a number of standard Java library classes that are intended to be atomic. These classes are listed in Table I and include the JDK 1.4.2 versions of `StringBuffer`, `String`, `PrintWriter`, `Vector`, `URL`, `Inflator`, and `Deflator`. We also include the 1.4.0 versions of `String` and `URL`, which contained atomicity errors that were previously identified by our type system [Flanagan and Qadeer 2003b] but fixed prior to the 1.4.2 release. We used the command line flag `-constructor_holds_lock`. The atomicity checker checked each class in under a second and succeeded in detecting a number of subtle atomicity violations, including errors that would not be caught by a race condition checker:

java.util.StringBuffer. This class provides an excellent example of the benefits of our type system, since its documentation clearly states that all `StringBuffer` methods should be 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 method `append(StringBuffer sb)` then failed to type check:

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

  /*# atomic */ public synchronized int length() {
    return count;
  }
  /*# atomic */ public synchronized void getChars(...) { ... }

  // does not type check:
  /*# atomic */
  public synchronized StringBuffer append(StringBuffer sb) {
    if (sb == null) { sb = NULL; }
    int len = sb.length();            // len may become 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;
    }
  }
```

An examination of the method reveals that it violates its atomicity specification. In particular, after `append(StringBuffer sb)` calls the synchronized method `sb.length()`, a second thread could remove characters from `sb`. In this situation, `len` is now *stale* [Burrows and Leino 2002] and no longer reflects the current length of `sb`, and so `sb.getChars(...)` is called with invalid arguments and throws a `StringIndexOutOfBoundsException`. The following test harness triggers this crash.

```
public class StringBufferTest 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 StringBufferTest()).start();
    while(true) (new StringBuffer()).append(sb);
  }
}
```

Our type system identifies this error because the method calls to `sb.length()` and `sb.getChars(...)` are both specified to have atomicity `atomic`, making the overall atomicity be `cmpd`.

java.lang.String. This class (from JDK 1.4.0) contains a method `contentEquals` that 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
    ...
  }
```

This defect was fixed in the 1.4.2 release by having `contentEquals(StringBuffer sb)` acquire the lock on `sb` for the duration of the method call.

The method `String.hashCode()` in both tested JDK releases is also not atomic because it caches the hashcode for the `String` object in an unprotected field. However, this can only result in redundant hash recomputations and not erroneous behavior, and we suppressed the warning with a `no_warn` annotation. Analysis techniques that abstract away benign atomicity violations [Flanagan et al. 2005] can eliminate some spurious warnings like this one.

java.util.Vector. The class `java.util.Vector` illustrates the need for conditional atomicities. The synchronized method `removeElement` calls the methods `indexOf` followed by `removeElementAt`. To verify that `removeElement` is atomic, we need to know that `removeElementAt` behaves as a `mover` when called with the vector's lock held. Thus, we declare `removeElementAt` as having the conditional atomicity `this ? mover : atomic` (and similarly for `indexOf`). These conditional atomicities enable us to conclude that `removeElement` has the conditional atomicity `this ? mover : atomic`, which guarantees that its behavior will always be atomic, regardless of which locks are held when it is invoked.

```
public class Vector ... {

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

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

}
```

Given this specification, the type system assigns the atomicity `this ? mover : atomic` to both the call to `indexOf` and the call to `removeElementAt` in the body of `removeElement`, making the atomicity of the entire method body be `this ? mover : cmpd`. Since `removeElement` is synchronized, the overall atomicity for the method is $S(\texttt{this}, \texttt{this ? mover : cmpd}) = \texttt{this ? mover : atomic}$.

Bohr also found three errors in the class `Vector`, one of which is shown below. This error was independently detected by Wang and Stoller [2006].

```
interface Collection {
  /*# this ? mover : atomic */ int size();
  /*# this ? mover : atomic */ Object[] toArray(Object a[]);
}
```

```
class Vector ... {
  Object elementData[] /*# guarded_by this */;
  int elementCount     /*# guarded_by this */;

  // does not type check:
  /*# atomic */ Vector(Collection c) {
    elementCount = c.size();
    elementData = new Object[Math.min((elementCount*110L)/100,
                                      Integer.MAX_VALUE)];
    c.toArray(elementData);
  }
  ...
}
```

For simplicity, we annotated `Collection` under the assumption that `Collection` objects are internally synchronized, as is the case for `Hashtable` and `Vector`.[6] The `Vector` constructor should set `elementCount` to the size of its argument `c` and copy the contents of `c` into the newly-created array `elementData`. However, since the lock `c` is not held between the calls to `c.size()` and `c.toArray(...)`, another thread could concurrently modify `c`, resulting either in an improperly initialized `Vector` or an `ArrayIndexOutOfBounds` exception.

The type system identifies this error because it assigns the atomicity `(c?mover:atomic)` to both the call to `c.size()` and the call to `c.toArray(...)` in the body of `removeElement`, making the atomicity of the entire method body be `c?mover:cmpd`. The specification for the constructor is violated because `(c?mover:cmpd)` $\not\sqsubseteq$ `atomic`. The methods `Vector.removeAll(c)` and `Vector.retainAll(c)` exhibit similar defects.

java.util.PrintWriter. Type checking `java.io.PrintWriter` raised interesting issues concerning *rep-exposure*, which occurs when a component of an abstraction leaks outside of that abstraction's implementation [Detlefs et al. 1998].

The superclass of `PrintWriter` is the abstract class `Writer`, which includes methods for writing a single character or a `String` to a stream. These two methods are made atomic by synchronizing on the lock stored in the instance variable `lock`. The default value for the `lock` is the self reference.

```
public abstract class Writer {
  // The object used to synchronize operations on this stream.
  protected Object lock;

  Writer() { this.lock = this; }

  /*# lock ? mover : atomic */
```

[6]The Java library does contain subtypes of `Collection` that require external synchronization, such as `LinkedList` and `HashMap`. Permitting both internally and externally synchronized subtypes of `Collection` requires several extensions to the type system, as described in Sections 7.2 and 7.4. The simpler annotations above are sufficient to illustrate the potential error.

```
  public void write(int ch) { ... }

  /*# lock ? mover : atomic */
  public void write(String str) { ... }

  ...
}
```

Each `PrintWriter` object contains a reference `out` to an underlying `Writer` object and provides methods for printing a variety of data types to that `Writer`. For example, the method `print(int x)` prints an integer; `println()` prints a new line character; and `println(int x)` prints an integer followed by a new line character. The code below shows a simplified implementation of these features.

```
// Does not type check!
public class PrintWriter extends Writer {

  protected Writer out;

  public PrintWriter(Writer out) { super(); this.out = out; }

  public void print(int x) {
    synchronized (lock) {
      out.write(Integer.toString(x));
    }
  }

  public void println() {
    synchronized (lock) {
      out.write(lineSeparator);
    }
  }

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

The three methods `print(int x)`, `println()`, and `println(int x)` all synchronize on the lock `lock` inherited from the superclass `Writer`, and so one might expect these methods to be atomic. However, the methods do *not* synchronize on the lock `out.lock` of the underlying `Writer`. Hence, some other thread could concurrently write characters to the underlying `Writer` without acquiring the protecting lock used by the `PrintWriter`. For example, if the `PrintWriter p` uses `w` as the underlying `Writer`, two threads could concurrently call `p.println(3)`

and `w.write("hello")`, causing the output "hello" to be incorrectly printed between "3" and the new line.

To deal with this problem, we declared `println` and 9 similar methods in `PrintWriter` as `cmpd` and the remaining 17 public methods as `atomic`. We then succeeded in type checking `PrintWriter` without warnings.

An alternative would be to annotate the `print(int x)` and `println()` methods with the atomicity

```
out.lock ? (lock ? mover : atomic) : atomic
```

and the `println(int x)` method with the atomicity

```
out.lock ? (lock ? mover : atomic) : cmpd
```

While somewhat complex, this specification does clearly reflect the undesirable requirement that clients of a `PrintWriter` must perform locking on the underlying writer to ensure atomicity if the writer is shared among threads. However, it does come at the expense of exposing the underlying representation of the `PrintWriter` to the client. One could also ensure that uses of a `PrintWriter` are atomic by employing an ownership type system [Boyapati et al. 2002; Boyapati and Rinard 2001] or escape analysis [Choi et al. 1999; Salcianu and Rinard 2001] to reason about rep-exposure and whether the underlying `out` writer is shared among threads.

java.net.URL. The synchronization discipline used by `java.net.URL` is fairly involved, and the atomicity checker reported a number of race conditions on both versions 1.4.0 and 1.4.2. We have not yet determined if these warnings reflect real errors in the program or benign race conditions. Instead, we added `no_warn` annotations to instruct the checker to ignore the problematic field accesses. We did find one particularly suspicious code fragment with our tool in the 1.4.0 version. In particular, 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);
}
```

The type system identifies this error because there is no protecting lock for `specifyHandlerPerm`, and each access to that variable is therefore assigned the atomicity `atomic`. Any method that accesses the variable multiple times will be given the atomicity `cmpd`.

6. TYPE INFERENCE FOR ATOMICITY

Our type checker provides fairly promising results, but it does require the programmer to fully annotate the code. Table I shows the number of annotations

per thousand lines of code for the classes discussed in the previous section. The table also breaks the annotations down by type: guard annotations (`guarded_by` and `write_guarded_by`), atomicity annotations, and escapes (`holds`, `no_warn`). On average, roughly 20 annotations per thousand lines were required. The time necessary to understand the code and add these annotations significantly increases the cost of using this type system on large programs.

To address this shortcoming, we now develop a type inference algorithm for atomicity. Our inference algorithm proceeds in two phases. The first phase infers which locks (if any) protect each field. This task is accomplished using the *Rcc/Sat* subroutine, which is described in an earlier paper [Flanagan and Freund 2004b]. Essentially, this part of the type inference problem is NP-complete, and *Rcc/Sat* works via reduction to propositional satisfiability.

We focus here on the second type inference phase, which infers the most precise atomicity for each method, using a constraint-based analysis. This phase is quite subtle, since the type system supports conditional atomicities that contain lock expressions whose values must not change during execution. Our constraint language includes special constructs to describe such well-formedness requirements on conditional atomicities, and the constraint solver refers to judgments in the type system to enforce these requirements. Once this interaction between the type system and constraint solver is properly structured, the constraints can be solved with an iterative fixed-point algorithm.

6.1 Language Extensions for Type Inference

To support type inference, we extend atomicities to include atomicity variables α. An *open atomicity* s is either an (explicit) atomicity a or an atomicity variable α.

$$
\begin{array}{lll}
b & ::= & \texttt{const} \mid \texttt{mover} \mid \texttt{atomic} \mid \texttt{cmpd} \mid \texttt{error} \quad \text{(basic atomicities)} \\
a & ::= & b \mid l\,?\,a_1 : a_2 \quad \text{(atomicities)} \\
s & ::= & a \mid \alpha \quad \text{(open atomicities)} \\
\alpha & \in & \mathit{AtomVar} \quad \text{(atomicity variables)}
\end{array}
$$

We permit methods to be annotated by atomicity variables as well as explicit atomicities:

$$\mathit{meth} \ ::= \ s\ t\ \mathit{md}\langle \texttt{ghost}\ x_{1..n}\rangle(\mathit{arg}^*)\ \{\ e\ \}$$

An AtomicJava program is *explicitly-typed* if it does not contain atomicity variables. The *type inference* problem is, given a program P with atomicity variables, to replace each atomicity variable with an atomicity so that the resulting explicitly-typed program is well-typed.

We illustrate the type inference process using the unannotated version of the `List` class from Section 4.3 shown in Figure 6(a). Figure 6(b) shows the program after inferring the `guarded_by` clauses and class parameters with *Rcc/Sat* and after inserting the atomicity variables $\alpha_1, \ldots, \alpha_4$.

Our type inference rules perform a syntax-directed traversal of the program with atomicity variables to generate a collection of constraints over those variables. A subsequent constraint-solving phase then finds the most precise (minimal) solution to these constraints or determines that no solution exists, in which

(a) Unannotated Program

```
class ListElem {
  int num;
  ListElem next;

  int get() { return this.num; }
}

class List {
  ListElem elems;

  void add(int v) {
    sync(this) {
      this.elems =
        new ListElem(v,this.elems);
    }
  }

  void addPair(int i, int j) {
    this.add(i);
    this.add(j);
  }

  int get() {
    sync(this) {
      return this.elems.get();
    }
  }
}
```

(b) Program with Inferred Annotations

```
class ListElem⟨ghost x⟩ {
  int num guarded_by x;
  ListElem⟨x⟩ next guarded_by x;

  α1 int get() { return this.num; }
}

class List {
  ListElem⟨this⟩ elems guarded_by this;

  α2 void add(int v) {
    sync(this) {
      this.elems =
        new ListElem⟨this⟩(v,this.elems);
    }
  }

  α3 void addPair(int i, int j) {
    this.add(i);
    this.add(j);
  }

  α4 int get() {
    sync(this) {
      return this.elems.get();
    }
  }
}
```

Fig. 6. The example class `List` and inferred locking and atomicity annotations.

case type inference fails. The following subsections describe the constraint language, the type inference rules that generate constraints, and our constraint solving algorithm.

6.2 Atomicity Constraints

A *constraint* C is a subatomicity constraint between an *atomicity expression* d and an open atomicity s:

$$C ::= d \sqsubseteq s$$

Atomicity expressions include open atomicities as well as constructs for representing various operations on atomicities, such as sequential composition, join, iteration, and substitution.

$$\begin{aligned} d ::= {} & s \mid d \mathbin{;} d \mid d \sqcup d \mid d^* \mid l \mathbin{?} d : d && \text{(atomicity expressions)} \\ & \mid d \cdot \theta \mid \mathcal{S}(l, d) \mid \mathcal{R}(d) \mid \mathit{lift}(P, E, d) \end{aligned}$$

We use bold symbols such as "$\sqsubseteq$" and "**;**" to distinguish the syntactic constructs relating atomicity expressions from the corresponding semantic operations "$\sqsubseteq$"

and ";" on atomicities. The expression forms for sequential composition ($d;d$), join ($d \sqcup d$), iterative closure (d^*), conditional atomicities ($l\,?\,d : d$), synchronization ($\mathcal{S}(l, d)$), and assertion ($\mathcal{R}(d)$) are analogous to the underlying operations on atomicities. We discuss the atomicity expression forms for delayed substitution ($d \cdot \theta$) and lifting ($\mathit{lift}(P, E, d)$) where they are used in the following.

An atomicity expression is *closed* if it does not contain atomicity variables. The meaning function $[\![\cdot]\!]$ maps closed atomicity expressions to atomicities:

$$[\![\cdot]\!] : \mathit{ClosedAtomExpr} \to \mathit{Atomicity}$$

$$\begin{array}{rclcrcl} [\![a]\!] & = & a & & [\![d^*]\!] & = & [\![d]\!] \\ [\![d_1;d_2]\!] & = & [\![d_1]\!];[\![d_2]\!] & & [\![l\,?\,d_1 : d_2]\!] & = & l\,?\,[\![d_1]\!] : [\![d_2]\!] \\ [\![d_1 \sqcup d_2]\!] & = & [\![d_1]\!] \sqcup [\![d_2]\!] & & [\![d \cdot \theta]\!] & = & \theta([\![d]\!]) \end{array}$$

$$\begin{array}{rcl} [\![\mathcal{S}(l, d)]\!] & = & S(l, [\![d]\!]) \\ [\![\mathcal{R}(d)]\!] & = & R([\![d]\!]) \\ [\![\mathit{lift}(P, E, d)]\!] & = & a \quad \text{such that } P; E \vdash [\![d]\!] \uparrow a \end{array}$$

6.3 Assignments

An *assignment* A maps atomicity variables to atomicities:

$$A : \mathit{AtomVar} \to \mathit{Atomicity}$$

The ordering relation for assignments is the point-wise extension of the subatomicity relation:

$$\begin{array}{c} A_1 \sqsubseteq A_2 \text{ iff } \forall \alpha.\ A_1(\alpha) \sqsubseteq A_2(\alpha) \\ \bot \stackrel{\text{def}}{=} \lambda \alpha.\ \texttt{const} \end{array}$$

We extend assignments in a compatible manner to arbitrary atomicity expressions:

$$\begin{array}{rclcrcl} A(a) & = & a & & A(d^*) & = & A(d)^* \\ A(d_1;d_2) & = & A(d_1);A(d_2) & & A(d \cdot \theta) & = & A(d) \cdot \theta \\ A(d_1 \sqcup d_2) & = & A(d_1) \sqcup A(d_2) & & A(l\,?\,d_1 : d_2) & = & l\,?\,A(d_1) : A(d_2) \end{array}$$

$$\begin{array}{rcl} A(\mathcal{S}(l, d)) & = & \mathcal{S}(l, A(d)) \\ A(\mathcal{R}(d)) & = & \mathcal{R}(A(d)) \\ A(\mathit{lift}(P, E, d)) & = & \mathit{lift}(A(P), E, A(d)) \end{array}$$

We also extend assignments to programs, so that the program $A(P)$ is identical to P, except that each atomicity variable α is replaced with its meaning $A(\alpha)$.

An assignment A *satisfies* a constraint $C = d \sqsubseteq s$ (written $A \models C$) if, after applying the assignment, the meaning of the left-hand side of the constraint is a subatomicity of the right-hand side:

$$A \models d \sqsubseteq s \quad \text{iff} \quad [\![A(d)]\!] \sqsubseteq A(s)$$

If $A \models C$ for all $C \in \bar{C}$ then A is a *solution* for $\bar{C}$, written $A \models \bar{C}$. A set of constraints $\bar{C}$ is *valid*, written $\models \bar{C}$, if every assignment is a solution for $\bar{C}$. In particular, if A is a solution for $\bar{C}$, then $A(\bar{C})$ is valid, and vice-versa.

$\boxed{P; E \vdash e : t \cdot d \cdot \bar{C}}$

[INF EXP NULL]
$$\frac{}{P; E \vdash \texttt{null} : c \cdot \texttt{const} \cdot \bar{C}}$$

[INF EXP VAR]
$$\frac{P \vdash E \cdot \bar{C} \qquad E = E_1, t\ x, E_2}{P; E \vdash x : t \cdot \texttt{const} \cdot \bar{C}}$$

[INF EXP SYNC]
$$\frac{P; E \vdash_{\text{lock}} l \cdot \bar{C}_1 \qquad P; E \vdash e : t \cdot d \cdot \bar{C}_2 \qquad \bar{C} = (\bar{C}_1 \cup \bar{C}_2)}{P; E \vdash \texttt{sync}\ l\ e : t \cdot \mathcal{S}(l, d) \cdot C}$$

[INF EXP INT]
$$\frac{P \vdash E \cdot \bar{C}}{P; E \vdash n : \texttt{int} \cdot \texttt{const} \cdot \bar{C}}$$

[INF EXP LONG]
$$\frac{P \vdash E \cdot \bar{C}}{P; E \vdash n : \texttt{long} \cdot \texttt{const} \cdot \bar{C}}$$

[INF EXP ASSERT]
$$\frac{P; E \vdash e : t \cdot d \cdot \bar{C}}{P; E \vdash \texttt{assert-atomic}\ e : t \cdot \mathcal{R}(d) \cdot \bar{C}}$$

[INF EXP REF]
$$\frac{\begin{array}{c} P; E \vdash e : cn\langle l_{1..n}\rangle \cdot d' \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots t\ fd\ g \ldots\ \} \in P \\ \theta = [\texttt{this} := e, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash \theta(t) \cdot \bar{C}' \\ (g \equiv \texttt{final}) \Rightarrow (d = \texttt{const}) \\ (g \equiv \texttt{guarded\_by}\ l) \Rightarrow (d = \theta(l)\,\texttt{?}\,\texttt{mover}\,\texttt{:}\,\texttt{error}) \\ (g \equiv \texttt{write\_guarded\_by}\ l) \Rightarrow (d = \theta(l)\,\texttt{?}\,\texttt{mover}\,\texttt{:}\,B(t)) \\ (g \equiv \texttt{no\_guard}) \Rightarrow (d = B(t)) \end{array}}{P; E \vdash e.fd : \theta(t) \cdot (d'\texttt{;}\mathit{lift}(P, E, d)) \cdot (\bar{C} \cup \bar{C}')}$$

[INF EXP IF]
$$\frac{P; E \vdash e_1 : \texttt{int} \cdot d_1 \cdot \bar{C}_1 \qquad P; E \vdash e_i : t \cdot d_i \cdot \bar{C}_i \quad \forall i \in 2..3 \qquad d = d_1\texttt{;}(d_2 \sqcup d_3)}{P; E \vdash \texttt{if}\ e_1\ e_2\ e_3 : t \cdot d \cdot \bar{C}_{1..3}}$$

[INF EXP ASSIGN]
$$\frac{\begin{array}{c} P; E \vdash e_1 : cn\langle l_{1..n}\rangle \cdot d_1 \cdot \bar{C}_1 \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\{\ \ldots t\ fd\ g \ldots\ \} \in P \\ \theta = [\texttt{this} := e_1, x_j := l_j\ ^{j \in 1..n}] \\ P; E \vdash e_2 : \theta(t) \cdot d_2 \cdot \bar{C}_2 \\ (g \equiv \texttt{final}) \Rightarrow (d = \texttt{error}) \\ (g \equiv \texttt{guarded\_by}\ l) \Rightarrow (d = \theta(l)\,\texttt{?}\,\texttt{mover}\,\texttt{:}\,\texttt{error}) \\ (g \equiv \texttt{write\_guarded\_by}\ l) \Rightarrow (d = \theta(l)\,\texttt{?}\,B(t)\,\texttt{:}\,\texttt{error}) \\ (g \equiv \texttt{no\_guard}) \Rightarrow (d = B(t)) \end{array}}{P; E \vdash (e_1.fd = e_2) : \theta(t) \cdot (d_1\texttt{;}d_2\texttt{;}\mathit{lift}(P, E, d)) \cdot (\bar{C}_1 \cup \bar{C}_2)}$$

[INF EXP WHILE]
$$\frac{P; E \vdash e_1 : \texttt{int} \cdot d_1 \cdot \bar{C}_1 \qquad P; E \vdash e_2 : t \cdot d_2 \cdot \bar{C}_2 \qquad d = d_1\texttt{;}((d_2\texttt{;}d_1)^*)}{P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{int} \cdot d \cdot \bar{C}_{1..2}}$$

Fig. 7. AtomicJava type inference rules (I).

6.4 Type Inference Rules

The ATOMICJAVA type inference judgments and rules are shown in Figures 7–9. Mostly, these judgments extend the type checking judgments of Figures 3–5 to also generate atomicity constraints. For example, the main type inference judgment:

$$P; E \vdash e : t \cdot d \cdot \bar{C}$$

now yields a set of constraints $\bar{C}$ generated from type checking e with respect to program P and environment E. We highlight the most interesting extensions:

[INF EXP SYNC] The atomicity of a synchronized expression `sync` l e is $\mathcal{S}(l, d)$, where d is the atomicity of e. Recall that the semantics of $\mathcal{S}(l, d)$ is defined in terms of the function S, that is, $[\![\mathcal{S}(l, d)]\!] = S(l, [\![d]\!])$.

[INF EXP LET] This rule for `let` x `=` e_1 `in` e_2 infers atomicity expressions d_1 and d_2 for e_1 and e_2, respectively. Since d_2 may mention x, we introduce the substitution $\theta = [x := e_1]$ as in the earlier type checking rule [EXP LET], but

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

[INF EXP NEW]

$$\frac{\begin{array}{c}\theta = [x_j := l_j\ ^{j\in 1..n}, \texttt{this} := y] \\ P; E, \texttt{ghost}\ y \vdash e_i : \theta(t_i) \cdot d_i \cdot \bar{C}_i \qquad \forall i \in 1..k \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle \\ \{\ field_{1..k}\ meth_{1..m}\ \} \in P \\ field_i = t_i\ fd_i\ g_i \qquad \forall i \in 1..k \\ P; E \vdash cn\langle l_{1..n}\rangle \cdot \bar{C} \\ \bar{C}' = \bar{C}_{1..k} \cup \bar{C} \\ d' = (d_1; \cdots; d_k)\end{array}}{P; E \vdash \texttt{new}_y\ cn\langle l_{1..n}\rangle(e_{1..k}) : cn\langle l_{1..n}\rangle \cdot d' \cdot \bar{C}'}$$

[INF EXP LET]

$$\frac{\begin{array}{c}P; E \vdash e_1 : t_1 \cdot d_1 \cdot \bar{C}_1 \\ P; E, t_1\ x \vdash e_2 : t_2 \cdot d_2 \cdot \bar{C}_2 \\ \theta = [x := e_1] \\ P; E \vdash \theta(t_2) \cdot \bar{C} \\ d = d_1; lift(P, E, d_2 \cdot \theta) \\ \bar{C}' = \bar{C}_{1..2} \cup \bar{C}\end{array}}{P; E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \theta(t_2) \cdot d \cdot \bar{C}'}$$

[INF EXP INVOKE]

$$\frac{\begin{array}{c}P; E \vdash e : cn\langle l_{1..n}\rangle \cdot d \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ \ldots meth \ldots\ \} \in P \\ meth = s\ t\ md\langle \texttt{ghost}\ y_{1..k}\rangle(t_j\ z_j^{\,j\in 1..r})\ \{\ e'\ \} \\ \theta = [\texttt{this} := e, x_i := l_i\ ^{i\in 1..n}, \\ y_i := l'_i\ ^{i\in 1..k}, z_i := e_i\ ^{i\in 1..r}] \\ P; E \vdash e_j : \theta(t_j) \cdot d_j \cdot \bar{C}_j \qquad \forall j \in 1..r \\ P; E \vdash \theta(t) \cdot \bar{C}' \\ P; E \vdash_{\text{lock}} l'_i \cdot \bar{C}'_i \qquad \forall i \in 1..k \\ d' = d; d_1; \cdots; d_r; lift(P, E, s \cdot \theta) \\ \bar{C}'' = (\bar{C} \cup \bar{C}_{1..r} \cup \bar{C}' \cup \bar{C}'_{1..k})\end{array}}{P; E \vdash e.md\langle l'_{1..k}\rangle(e_{1..r}) : \theta(t) \cdot d' \cdot \bar{C}''}$$

[INF EXP FORK]

$$\frac{\begin{array}{c}P; E \vdash e : cn\langle l_{1..n}\rangle \cdot d \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle \\ \{\ \ldots meth \ldots\ \} \in P \\ meth = s\ \texttt{int run}\langle \texttt{ghost tll}\rangle()\ \{\ e'\ \} \\ \bar{C}'' = \bar{C} \cup \{\ s \sqsubseteq (\texttt{tll}\ ?\ \texttt{cmpd} : \texttt{error})\}\end{array}}{P; E \vdash e.\texttt{fork} : \texttt{int} \cdot (d; \texttt{atomic}) \cdot \bar{C}''}$$

Fig. 8. AtomicJava type inference rules (II).

here we need to use the atomicity expression form $d_2 \cdot \theta$ to delay applying this substitution until after atomicity variables are resolved. (A similar delayed substitution occurs in [INF EXP INVOKE].)

Furthermore, e_1 may not be `const` (in general, we cannot determine which expressions are `const` until after type inference), in which case $d_2 \cdot \theta$ may not be a valid atomicity. Therefore, we use the atomicity expression $lift(P, E, d_2 \cdot \theta)$ to yield an atomicity for e_2 that is well-formed in environment E.

[INF LOCK EXP] The judgment $P; E \vdash_{\text{lock}} e \cdot \bar{C}$ ensures that e is a valid lock expression. This rule checks that e denotes a fixed lock throughout the execution of the program by generating the constraint that e has atomicity `const`.

As in the type system, the requirement that $|e| \leq MaxLockSize$ ensures that there is only a finite number of valid lock expressions at any program point, which in turn bounds the size of conditional atomicities and guarantees termination of our type inference algorithm.

The type inference system defines the top-level judgment

$$P \vdash \bar{C}$$

where $\bar{C}$ is the generated set of constraints for the program P.

For the example program `List` of Figure 6, our system generates the four atomicity constraints in Figure 10. (We omit several trivial constraints that do not involve atomicity variables, such as `const` $\sqsubseteq$ `const`.)

$\boxed{P \vdash E \cdot \bar{C}}$

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

[INF ENV X]
$$\frac{P; E \vdash t \cdot \bar{C} \quad x \notin dom(E)}{P \vdash (E, t\ x) \cdot \bar{C}}$$

[INF ENV GHOST]
$$\frac{P \vdash E \cdot \bar{C} \quad x \notin dom(E)}{P \vdash (E, \texttt{ghost}\ x) \cdot \bar{C}}$$

$\boxed{P; E \vdash t \cdot \bar{C}}$

[INF TYPE PRIM]
$$\frac{P \vdash E \cdot \bar{C} \quad t \in \{\texttt{int}, \texttt{long}\}}{P; E \vdash t \cdot \bar{C}}$$

[INF TYPE C]
$$\frac{\texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \ldots \in P \quad P; E \vdash_{\text{lock}} l_i \cdot \bar{C}_i \quad \forall i \in 1..n}{P; E \vdash cn\langle l_{1..n}\rangle \cdot \bar{C}_{1..n}}$$

$\boxed{P; E \vdash s \cdot \bar{C}}$

[INF AT BASE]
$$\frac{P \vdash E \cdot \bar{C}}{P; E \vdash b \cdot \bar{C}}$$

[INF AT COND]
$$\frac{P; E \vdash_{\text{lock}} l \cdot \bar{C} \quad P; E \vdash a_i \cdot \bar{C}_i \quad \forall i \in 1..2}{P; E \vdash l\,?\,a_1 : a_2 \cdot (\bar{C} \cup \bar{C}_{1..2})}$$

[INF AT VAR]
$$\frac{P \vdash E \cdot \bar{C}}{P; E \vdash \alpha \cdot \bar{C}}$$

$\boxed{P; E \vdash_{\text{lock}} l \cdot \bar{C}}$

[INF LOCK EXP]
$$\frac{P; E \vdash e : c \cdot d \cdot \bar{C} \quad |e| \leq MaxLockSize}{P; E \vdash_{\text{lock}} e \cdot (\bar{C} \cup \{d \sqsubseteq \texttt{const}\})}$$

[INF LOCK GHOST]
$$\frac{P \vdash E \cdot \bar{C} \quad E = E_1, \texttt{ghost}\ x, E_2}{P; E \vdash_{\text{lock}} x \cdot \bar{C}}$$

$\boxed{P; E \vdash field \cdot \bar{C}}$

[INF FIELD]
$$\frac{\begin{array}{c} P; E \vdash t \cdot \bar{C} \\ (g \equiv \texttt{final}) \Rightarrow \bar{C}' = \emptyset \\ (g \equiv \texttt{guarded\_by}\ l) \Rightarrow P; E \vdash_{\text{lock}} l \cdot \bar{C}' \\ (g \equiv \texttt{write\_guarded\_by}\ l) \Rightarrow P; E \vdash_{\text{lock}} l \cdot \bar{C}' \\ (g \equiv \texttt{no\_guard}) \Rightarrow \bar{C}' = \emptyset \end{array}}{P; E \vdash t\ fd\ g \cdot (\bar{C} \cup \bar{C}')}$$

$\boxed{P; E \vdash meth \cdot \bar{C}}$

[INF METHOD]
$$\frac{\begin{array}{c} E' = E, \texttt{ghost}\ x_1, \ldots, \texttt{ghost}\ x_n, arg_{1..r} \\ P; E' \vdash e : t \cdot d \cdot \bar{C} \\ P; E' \vdash s \cdot \bar{C}' \\ \bar{C}'' = (\bar{C} \cup \bar{C}' \cup \{lift(P, E', d) \sqsubseteq s\}) \end{array}}{P; E \vdash s\ t\ md\langle \texttt{ghost}\ x_{1..n}\rangle(arg_{1..r})\ \{\ e\ \} \cdot \bar{C}''}$$

$\boxed{P \vdash defn \cdot \bar{C}}$

[INF CLASS]
$$\frac{\begin{array}{c} E = \texttt{ghost}\ x_1, \ldots, \texttt{ghost}\ x_n, cn\langle x_{1..n}\rangle\ \texttt{this} \\ P; E \vdash field_i \cdot \bar{C}_i \quad \forall i \in 1..j \\ P; E \vdash meth_i \cdot \bar{C}'_i \quad \forall i \in 1..k \\ \bar{C} = (\bar{C}_{1..j} \cup \bar{C}'_{1..k}) \end{array}}{P \vdash \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ field_{1..j}\ meth_{1..k}\ \} \cdot \bar{C}}$$

$\boxed{P \vdash \bar{C}}$

[INF PROG]
$$\frac{\begin{array}{c} ClassOnce(P) \\ FieldsOnce(P) \\ ObjectDefined(P) \\ MethodsOncePerClass(P) \\ P = defn_{1..n}\ e \\ P \vdash defn_i \cdot \bar{C}_i \quad \forall i \in 1..n \\ P; \epsilon \vdash e : t \cdot d \cdot \bar{C} \end{array}}{P \vdash \bar{C}_{1..n} \cup \bar{C} \cup \{d \sqsubseteq \texttt{cmpd}\}}$$

Fig. 9. AtomicJava type inference rules (III).

As a technical requirement, we introduce the notion of a well-formed assignment. An assignment A is *well-formed* for $\bar{C}$ if, for all constraints $(lift(P, E, d) \sqsubseteq \alpha)$ in $\bar{C}$, $A(P); E \vdash A(\alpha)$. In other words, $A(\alpha)$ cannot refer to lock expressions that are not well-formed in the environment E.

$$lift(P, E_1, \texttt{const;}lift(P, E_1, \texttt{x ? mover : error})) \sqsubseteq \alpha_1$$

$$lift(P, E_2, \mathcal{S}(\texttt{this}, (\texttt{const;const;const;}lift(P, E_2, \texttt{this ? mover : error}); lift(P, E_2, \texttt{this ? mover : error})))) \sqsubseteq \alpha_2$$

$$lift(P, E_3, (\texttt{const;const;}lift(P, E_3, \alpha_2 \cdot [\texttt{this} := \texttt{this}])); (\texttt{const;const;}lift(P, E_3, \alpha_2 \cdot [\texttt{this} := \texttt{this}]))) \sqsubseteq \alpha_3$$

$$lift(P, E_4, \mathcal{S}(\texttt{this}, (\texttt{const;}lift(P, E_4, \texttt{this ? mover : error}); lift(P, E_4, \alpha_1 \cdot [\texttt{this} := \texttt{this.elems}, \texttt{x} := \texttt{this}])))) \sqsubseteq \alpha_4$$

where

$E_1 = \texttt{ghost x, ListElem}\langle\texttt{x}\rangle\ \texttt{this}$ $\quad$ $E_3 = \texttt{List this, int i, int j}$

$E_2 = \texttt{List this, int v}$ $\quad$ $E_4 = \texttt{List this}$

Fig. 10. Constraints for the `List` program.

With this definition, we can now state that if A is a well-formed solution to the constraints for a program P, then the explicitly-typed program $A(P)$ is well-typed.

THEOREM 4 (TYPE INFERENCE YIELDS WELL-TYPED PROGRAMS). *If $P \vdash \bar{C}$ and $A \models \bar{C}$ and A is well-formed for $\bar{C}$ then $A(P) \vdash wf$.*

The proof proceeds by induction over the derivation of $P \vdash \bar{C}$ and makes use of the fact that every constraint generated by that derivation is satisfiable. The well-formedness requirement on A ensures that the atomicities substituted for atomicity variables are well-formed in the scopes in which they appear. See Appendix F in the online appendix for details.

6.5 Solving Constraint Systems

To solve a generated constraint system $\bar{C}$, we start with the minimal assignment $A = \bot$ and iteratively increase this assignment until we reach a solution or obtain a contradiction. The relation $A \rightarrow^{\bar{C}} A'$ performs one step of this iterative computation. It identifies some constraint $d \sqsubseteq \alpha$ which is not satisfied by the current assignment A and produces a larger assignment A' that does satisfy that constraint:

$$A \rightarrow^{\bar{C}} A' \text{ iff } \exists\, (d \sqsubseteq \alpha) \in \bar{C} \text{ and } [\![A(d)]\!] \not\sqsubseteq A(\alpha) \text{ and } A' = A[\alpha := A(\alpha) \sqcup [\![A(d)]\!]]$$

The relation $A \rightarrow^{\bar{C}} \textsc{Error}$ detects if some constraint in $\bar{C}$ cannot be satisfied by the current assignment A or any larger assignment:

$$A \rightarrow^{\bar{C}} \textsc{Error} \text{ iff } \exists\, (d \sqsubseteq a) \in \bar{C} \text{ and } [\![A(d)]\!] \not\sqsubseteq a$$

Our constraint solving algorithm $Solve(\bar{C})$, defined in Figure 11, is an iterative least fix-point computation based on these two relations. For the atomicity

```
Solve(C̄) {
    A := ⊥;
    while ∃A′ such that A →^C̄ A′ {
      A := A′;
    }
    if A →^C̄ ERROR then {
      return "no solution";
    } else {
      return A;
    }
}
```

Fig. 11. Constraint solving algorithm.

constraints of Figure 10, this constraint solving algorithm yields the minimal solution:

$$\begin{aligned}
\texttt{ListElem.get}: &\quad \alpha_1 = (\texttt{x?mover:error})\\
\texttt{List.add}: &\quad \alpha_2 = (\texttt{this?mover:atomic})\\
\texttt{List.addPair}: &\quad \alpha_3 = (\texttt{this?mover:cmpd})\\
\texttt{List.get}: &\quad \alpha_4 = (\texttt{this?mover:atomic})
\end{aligned}$$

This solution yields the atomicities matching those discussed for the original version of `List` in Figure 2.

6.6 Correctness of the Algorithm

The *Solve* algorithm is correct and terminates for all constraint sets $\bar{C}$ generated by the type inference rules. To show this, we first characterize the two relations $A \rightarrow^{\bar{C}} A'$ and $A \rightarrow^{\bar{C}}$ ERROR and prove that if neither of these relations is applicable to an assignment A, then that assignment is a solution to $\bar{C}$. We assume throughout this section that $\bar{C}$ was generated by the type inference rules, that is, that there exists P such that $P \vdash \bar{C}$.

LEMMA 5 (STEP). *Suppose $A \rightarrow^{\bar{C}} A'$. Then $A \sqsubset A'$. If in addition there exists A'' such that $A \sqsubset A''$ and $A'' \models \bar{C}$, then we also have that $A' \sqsubseteq A''$.*

LEMMA 6 (CONTRADICTION). *Suppose $A \rightarrow^{C}$ ERROR. Then there is no A'' such that $A \sqsubseteq A''$ and $A'' \models \bar{C}$.*

LEMMA 7 (SOLUTION). *Suppose $A \not\rightarrow^{\bar{C}}$ ERROR and for all A', $A \not\rightarrow^{\bar{C}} A'$. Then $A \models \bar{C}$.*

Also, the algorithm *Solve* computes only well-formed assignments.

LEMMA 8 (WELL-FORMED). *If A is well-formed for $\bar{C}$ and $A \rightarrow^{\bar{C}} A'$ then A' is well-formed for $\bar{C}$.*

PROOF. Inspection of the type inference rules indicates that $\bar{C}$ will always have the following two properties:

(1) Each $C \in \bar{C}$ has one of two forms: $(d \sqsubseteq \alpha)$ or $(\mathit{lift}(P, E, d) \sqsubseteq \alpha)$.
(2) There is at most one lower bound $(\mathit{lift}(P, E, d) \sqsubseteq \alpha)$ in $\bar{C}$ for each α.

If $A \rightarrow^{\bar{C}} A'$, then $A' = A[\alpha := A(\alpha) \sqcup [\![A(\mathit{lift}(P, E, d))]\!]]$, where $(\mathit{lift}(P, E, d) \sqsubseteq \alpha) \in \bar{C}$ is the unique lower bound on α. Since A is well-formed, $A(P); E \vdash A(\alpha)$. Also, $[\![A(\mathit{lift}(P, E, d))]\!] = a$, where $A(P); E \vdash [\![A(d)]\!] \uparrow a$. This implies that $A(P); E \vdash a$. The atomicity $A'(\alpha) = A(\alpha) \sqcup a$ will also be well-formed in E, since it can only refer to lock expressions already present in $A(\alpha)$ and a. Thus, A' is well-formed for $\bar{C}$. □

The previous four lemmas are sufficient to show in Theorem 12 that the algorithm only produces correct results. Moreover, Lemma 5 guarantees that the algorithm computes the most precise satisfying assignment.

Proving termination is more difficult, because delayed substitutions could lead to arbitrarily large lock expressions and infinite ascending chains of atomicities and assignments. We bound the size of lock expressions to exclude this possibility. A lock expression l is *bounded* if $|l| < \mathit{MaxLockSize}$. Similarly, an atomicity is bounded if it only contains bounded lock expressions. An assignment is bounded if it only yields bounded atomicities. An atomicity expression or constraint is bounded if it is only conditional on bounded lock expressions, and if every delayed substitution occurs inside the construct $\mathit{lift}(P, E, \cdot)$.

LEMMA 9. *If d is a closed, bounded atomicity expression, then $[\![d]\!]$ is also bounded.*

PROOF. The only difficulty is that $[\![d]\!]$ may apply delayed substitutions in d, which could result in non-bounded lock expressions, but the enclosing construct $\mathit{lift}(P, E, \cdot)$ will filter out these non-bounded lock expressions. □

LEMMA 10. *If A and d are bounded, then $A(d)$ is bounded.*

LEMMA 11 (TERMINATION). *The constraint solving algorithm terminates on any bounded, constraint system $\bar{C}$.*

PROOF. Since $\bar{C}$ is bounded, every generated assignment is also bounded. In addition, $\bar{C}$ contains only a finite number of distinct variable and field names. All bounded lock expressions appearing in the generated assignments are derived from these names. Only a finite number of such bounded lock expressions exist, so there is only a finite set of bounded assignments containing them. Since the generated assignments are increasing, the algorithm must terminate, as otherwise it would generate an infinite ascending chain of bounded assignments drawn from this set. □

Since the type inference rules only generate bounded constraint systems, the algorithm will terminate for any set of constraints generated while checking a program. Thus, we may state the following correctness theorem.

THEOREM 12 (TYPE INFERENCE CORRECTNESS). *Given program P and constraints $\bar{C}$ such that $P \vdash \bar{C}$:*

(1.) *$\mathit{Solve}(\bar{C})$ always terminates.*

(2.) *If $\exists A'.\ A'(P) \vdash \mathit{wf}$, then $\mathit{Solve}(\bar{C})$ returns an assignment A.*

(3.) *If $\mathit{Solve}(\bar{C})$ returns an assignment A, then $A(P) \vdash \mathit{wf}$.*

PROOF. The first two parts follow from Lemma 11 and Lemmas 5–6, respectively. For part (3), suppose $Solve(\bar{C})$ returns A. Lemma 7 indicates that $A \models \bar{C}$. Since $\bot$ is well-formed for $\bar{C}$ and $\bot \rightarrow^{\bar{C}} \ldots \rightarrow^{\bar{C}} A$, Lemma 8 states that A is well-formed for $\bar{C}$. Theorem 4 then concludes that $A(P) \vdash wf$. □

7. IMPLEMENTATION

We have extended the *Bohr* type checker to perform type inference. *Bohr* takes Java source code with optional `guarded_by` and atomicity annotations. If all annotations are provided, *Bohr* simply ensures the annotations are correct, as described in Section 5. If some or all annotations are missing, *Bohr* proceeds to infer them and to report violations of any annotations that were provided.

Inference runs in two phases. The first phase infers appropriate guards for each field using the *Rcc/Sat* subroutine. Since field guards may refer to ghost parameters, *Rcc/Sat* also infers appropriate formal and actual ghost parameters for class and method declarations and uses, respectively. For more details on *Rcc/Sat*, we refer the interested reader to our earlier paper [Flanagan and Freund 2004b].

The key novelty of our inference algorithm is the second phase of inference. This phase first adds a fresh atomicity variable α to each method without an explicit atomicity. It then checks the program according to the constraint-based type inference algorithm described in Section 6. If a solution to the generated constraints is found, *Bohr* outputs a fully annotated version of the source code. Otherwise, the checker prints warning messages for each atomicity violation identified.

A common and significant problem with many type-inference techniques is the inability to construct meaningful error messages when inference fails (see, for example, [Wand 1986; Yang et al. 2000; Haack and Wells 2003]). An interesting contribution of our approach is that if the source program is completely unannotated, then type inference never fails; instead it just assigns an atomicity such as `cmpd` to each nonatomic method in the program.

The remainder of this section describes a number of implementation details regarding both performance and extensions necessary to support expressive subtyping and additional common synchronization idioms of Java programs.

7.1 Avoiding Exponential Explosion

Our initial implementation of the type inference algorithm often produced atomicities with millions of terms. To illustrate why, note that the sequential composition of two conditional atomicities

$$(l\,?\,a_1 : a_2)\ ;\ (l\,?\,a_3 : a_4)$$

yields the atomicity

$$l\ ?\ (l\,?\,(a_1;a_3) : (a_1;a_4))\ :\ (l\,?\,(a_2;a_3) : (a_2;a_4))$$

containing many duplicate subterms. More generally, the sequential composition of n conditional atomicities yields an atomicity whose size is exponential in n.

$$\frac{}{b \rightarrow_n^h b}\ [\text{SIMP ID}] \qquad \frac{l \in h \quad a_1 \rightarrow_n^h a}{l\,?\,a_1 : a_2 \rightarrow_n^h a}\ [\text{SIMP HELD}] \qquad \frac{l \in n \quad a_2 \rightarrow_n^h a}{l\,?\,a_1 : a_2 \rightarrow_n^h a}\ [\text{SIMP UNHELD}]$$

$$\frac{a_1 \rightarrow_n^{h \cup \{l\}} a' \quad a_2 \rightarrow_{n \cup \{l\}}^h a'}{l\,?\,a_1 : a_2 \rightarrow_n^h a'}\ [\text{SIMP EQ}] \qquad \frac{a_1 \rightarrow_n^{h \cup \{l\}} a'_1 \quad a_2 \rightarrow_{n \cup \{l\}}^h a'_2}{l\,?\,a_1 : a_2 \rightarrow_n^h l\,?\,a'_1 : a'_2}\ [\text{SIMP REC}]$$

Fig. 12. Atomicity simplification rules.

These large atomicities typically contain redundant information and can be simplified. For example, the above result can be simplified to

$$l\,?(a_1;a_3):(a_2;a_4)$$

The simplification rules in Figure 12 define a relation $a \rightarrow_n^h a'$ that simplifies a to a' by removing redundant information, under the assumption that the locks in h are held, and the locks in n are not held. *Bohr* always applies the first of these rules that is applicable.

One strategy for applying these rules is, after computing $a = [\![d]\!]$, to immediately simplify a via $a \rightarrow_\emptyset^\emptyset a'$. However, the intermediate atomicity a may still be prohibitively large. Instead, we use an optimized routine that directly computes the simplified atomicity a' from d in a single pass, applying the simplification rules on-the-fly to avoid unnecessarily large intermediate atomicities. The running time of this optimized algorithm is linear in the size of the resulting atomicity a'. Although a' may still, in theory, be exponential in the size of the program, our algorithm works well in practice since a' is typically small.

An interesting area for future work is to explore the use of binary decision diagrams [Bryant 1986] to represent and manipulate conditional atomicities efficiently.

7.2 Subtyping and Covariant Atomicity Specifications

The most significant extension to our type system and type inference rules for supporting large programs is to support inheritance and subtyping. Consider a class `C` with a subclass `D`:

```
class C⟨ghost x⟩ {
  s1 t1 f() { ... }
}

class D⟨ghost y⟩ extends C⟨z⟩ {
  s2 t2 f() { ... }
}
```

We consider a type `D`$\langle l \rangle$ to be an immediate subtype of `C`$\langle m \rangle$ provided $m \equiv z[y := l]$. (The extension to multiple ghost arguments is straightforward, and it is

omitted for clarity.) The subtyping relation is the reflexive-transitive closure of this rule.

Note that the class `C` declares a method `f()` that is overridden in `D`. We require $t_2 = \theta(t_1)$, that is, that the return type of the overriding and overridden methods must match exactly, after applying the type parameter substitution $\theta = [x := z]$ induced by the inheritance hierarchy. A similar requirement applies for argument types.

The original prototype of *Bohr* required that $\theta(s_1) = s_2$, but this was not expressive enough in many cases. For increased expressiveness, we permit the atomicity of a method to change covariantly, intuitively requiring only that

$$s_2 \sqsubseteq s_1 \cdot \theta$$

However, if s_1 is an atomicity variable α and this constraint is not satisfied by the current assignment A, it is unclear how to increase $A(\alpha)$ to satisfy this constraint. We therefore convert the constraint into a more standard format before running the algorithm. First, we replace the substitution θ on the right-hand side with a corresponding *inverse substitution* on the left-hand side. For this purpose, we introduce the *inverse substitution function*:

$$\theta^{-1}(l) \stackrel{\text{def}}{=} \{l' \mid \theta(l') = l\}$$

This function yields a set of all lock names l' that yield l under substitution θ. We extend the inverse substitution function to atomicities:

$$\begin{aligned} \theta^{-1}(b) &= b \\ \theta^{-1}(l\,?\,a_1 : a_2) &= l_1\,?\,a'_1 : (l_2\,?\,a'_1 \cdots : (l_n\,?\,a'_1 : a'_2) \cdots) \\ &\qquad \text{where } \theta^{-1}(a_i) = a'_i \quad \text{for } i = 1, 2 \\ &\qquad \text{and } \quad \theta^{-1}(l) = \{l_1, \ldots, l_n\} \end{aligned}$$

Each $l_{1..n}$ maps to a'_1 to reflect that all of these locks become l after applying θ.

We then introduce a new atomicity expression construct $invsub(\theta, d)$ with the following meaning:

$$[\![invsub(\theta, d)]\!] = \theta^{-1}([\![d]\!])$$

This expression permits us to express the above requirement $s_2 \sqsubseteq s_1 \cdot \theta$. as the more easily manipulated constraint

$$lift(P, E, invsub(\theta, s_2)) \sqsubseteq s_1$$

where the environment E of the class `C` ensures that the resulting atomicity for s_1 is well-formed in `C`. Interfaces are handled in a similar fashion.

7.3 Protected Locks

The next two sections describe extensions to the annotation language that we have found valuable in practice for describing additional synchronization idioms.

First, lock acquire and release operations are normally right and left movers, respectively. However, large programs typically contain redundant lock operations that we can more precisely characterize as both movers. For example,

(a) Without Protecting Locks

```
class Vector {
  this ? mover : atomic
  synchronized void add(Object o) { ... }

  this ? mover : atomic
  synchronized boolean contains(Object o) { ... }
  ...
}

class Observable {
  Vector obs = new Vector();

  (this ? mover : atomic)    // Fails to typecheck
  synchronized void addObserver(Observer o) {
    // ...
    if (!obs.contains(o)) obs.addElement(o);
  }
  ...
}
```

(b) With Protecting Locks

```
class Vector⟨protecting x⟩ {
  (x = none) ? (this ? mover : atomic) : (x ? mover : error)
  synchronized void add(Object o) { ... }

  (x = none) ? (this ? mover : atomic) : (x ? mover : error)
  synchronized boolean contains(Object o) { ... }
  ...
}

class Observable {
  Vector⟨this⟩ obs = new Vector⟨this⟩();

  (this ? mover : atomic)
  synchronized void addObserver(Observer o) {
    // ...
    if (!obs.contains(o)) obs.addElement(o);
  }
  ...
}
```

Fig. 13. Annotations for `Vector` and `Observable`.

the AtomicJava type system already treats re-entrant lock operations as both movers.

One common cause of redundant lock operations is illustrated by the `Vector` and `Observable` classes in Figure 13(a). The method `Observable.addObserver` calls `Vector.contains` and then `Vector.add`. With the type system outlined so far, the most precise atomicity assignable to the two `Vector` methods is `this?mover:atomic`, which causes `Observable.addObserver` to be `cmpd`.

However, that method can shown to be atomic by introducing the notion of *protected locks*.

A lock l *protects* lock m if l is held whenever a `synchronized` m e is encountered. In this case, the atomicity of `synchronized` m e is simply the atomicity of e, since no other thread will be trying to synchronize on m. In Figure 13(a), the lock of the `Observable` object protects the lock of the underlying `Vector`.

To support protecting locks, the first ghost parameter to a class may be specified as the `protecting` lock for instances of that class, as in the declaration

$$\texttt{class A}\langle\texttt{protecting } x_1, \texttt{ghost } x_2, x_3\rangle$$

Given an object a of type $\texttt{A}\langle l_1, l_2, l_3\rangle$, the lock l_1 is the protecting lock of a and must be held when synchronizing on a. Since not all instances of the class `A` may have protecting locks, we also introduce the special lock "`none`" to indicate that an object has no protecting lock, as in $\texttt{A}\langle\texttt{none}, l_2, l_3\rangle$. In this case, a thread can synchronize on the object without restriction. We could permit more than one parameter to be designated as a protecting lock, but we have never needed that added expressiveness. We have extended *Rcc*/*Sat* to infer protecting locks.

In Figure 13(b), we may now assign to the `Vector` methods the more precise *predicated atomicity*

$$\texttt{(x=none)?(this?mover:atomic):(x?mover:error)}$$

This atomicity uses the *conditional predicate* `x = none` to test the equivalence of the lock substituted for the ghost parameter `x` against the lock name `none`. Thus, this method atomicity is conditional both (1) whether the `Vector` on which the method is invoked has a protecting lock, (2) whether the protecting lock is held, and (3) whether the `this` lock is held. After inferring this atomicity for the `Vector` methods, *Bohr* can verify that `addObserver` is `atomic`.

Another common case involving protecting locks occurs when a lock is only manipulated by a single thread, such as when a `Vector` is created and used only within one thread. The `Vector`'s lock can be considered protected by the thread-local lock for that thread (see Section 3).

7.4 Internal Synchronization

A common pattern in the Java collections library is the use of synchronized wrapper classes. This pattern is illustrated in Figure 14, in which the interface `Counter` declares a method `inc`, and different `Counter` implementations use different synchronization disciplines.

The `UnsyncCounter` implementation requires clients to acquire a protecting lock `lock2` before calling the method `UnsyncCounter.inc`, which has atomicity

$$\texttt{lock2?mover:error}$$

The parameter `lock2` may be instantiated with the thread-local lock to create counters for use in a single thread, or with a lock protecting accesses from different threads when a counter is shared.

```
interface Counter⟨ghost lock1⟩ {
  (lock1 = always_held) ? atomic : (lock1 ? mover : error) int inc();
}

class UnsyncCounter⟨ghost lock2⟩ implements Counter⟨lock2⟩ {
  int num guarded_by lock2;

  (lock2 ? mover : error) int inc() { return num++; }
}

class SyncCounter implements Counter⟨always_held⟩ {
  Counter⟨this⟩ c guarded_by this;

  (this ? mover : atomic) int inc() {
    synchronized(this) { return c.inc(); }
  }
}

let SyncCounter sc = new SyncCounter(new UnsyncCounter⟨sc⟩()) in {
  // share sc between threads
  sc.inc();
}
```

Fig. 14. Synchronized wrapper class.

The SynchCounter class is a wrapper class that internally synchronizes the inc operation avoid the need for external locking. No locks need to be held before calling the method SynchCounter.inc, which is atomic.

A major difficulty in checking this code is that the Counter interface must be a supertype of both the externally and internally synchronized subclasses. To simultaneously support both synchronization disciplines, we introduce a special lock "always_held". This lock is implicitly simultaneously held by all threads, but cannot be used to guard fields. We assign the method Counter.inc the predicated atomicity:

$$(\texttt{lock1} = \texttt{always\_held})\,?\,\texttt{atomic} : (\texttt{lock1}\,?\,\texttt{mover} : \texttt{error})$$

The predicate (lock1 = always_held) holds if Counter is parameterized by the special lock always_held; if so, then inc is internally synchronized and is atomic; if not, then inc has the standard conditional atomicity (lock1 ? mover : error).

The program in Figure 14 declares a SyncCounter sc that is a wrapper around an UnsyncCounter, where the UnsyncCounter is protected by the lock sc. Our implementation puts the declared variable sc in scope (as a ghost variable) in the initialization expression for sc, in order to support a natural initialization syntax for such synchronized wrappers.

8. TYPE INFERENCE EVALUATION

We have applied *Bohr* to a number of benchmarks, including both standard library classes and complete programs. Table II summarizes the results of these

Table II. Performance and Results of *Bohr* Applied to Benchmark Programs and Thread-Safe Classes

Name	Size (LOC)	Time(s)	$\|\tilde{C}\|$	Manual Annot./ KLOC	Exported Methods		Synchronized Blocks	
					Number	Non-Atomic	Number	Non-Atomic
`java.lang.String`	2,307	0.45	139	0	69	1	2	0
`java.util.StringBuffer`	1,276	0.53	91	1.57	45	1	33	1
`java.util.Vector`	3,546	0.87	197	9.02	48	3	36	2
`java.util.zip.Inflater`	319	0.14	44	0	17	0	12	0
`java.util.zip.Deflater`	384	0.15	48	2.60	19	0	12	0
`java.util.zip.ZipFile`	498	0.89	61	4.02	9	1	3	0
`java.util.Observable`	198	0.97	68	0	8	0	8	0
`java.util.SynchronizedList`	3,837	3.02	254	7.04	28	2	23	2
`java.net.URL`	1,201	0.74	64	9.16	27	3	5	1
`java.io.PrintWriter`	712	0.33	90	4.21	31	11	14	5
`concurrent.SynchronizedBoolean`	450	0.19	41	4.44	18	4	10	2
`concurrent.SynchronizedDouble`	444	0.18	41	4.50	18	4	11	2
`elevator` [von Praun and Gross 2003]	529	0.60	22	7.56	13	2	8	0
`tsp` [von Praun and Gross 2003]	723	1.43	21	6.91	19	9	6	0
`sor` [von Praun and Gross 2003]	687	0.83	24	1.46	18	0	4	0
`raytracer` [Java Grande Forum 2003]	1,982	1.73	132	2.52	117	8	15	1
`moldyn` [Java Grande Forum 2003]	1,408	4.89	78	2.13	68	7	14	0
`montecarlo` [Java Grande Forum 2003]	3,674	1.48	223	0.27	209	5	14	0
`mtrt` [SPEC 2000]	11,315	7.80	468	0.97	376	7	7	0
`jbb` [SPEC 2000]	30,532	11.2	1170	1.47	993	153	241	51

experiments. Columns 1 and 2 show the name of each benchmark and its size in terms of lines of code.

Column 3 shows the running time of our implementation (excluding the time required for the *Rcc/Sat* subroutine, whose performance is documented in an earlier paper [Flanagan and Freund 2004b]). These experiments were performed on a Linux computer with a 3.06 GHz Pentium 4 Xeon processor and 2GB of memory. We set *MaxLockSize* to permit no more than four field accesses in lock expressions. Larger values of *MaxLockSize* slowed down performance with no increase in precision, and smaller values degraded precision. Overall, the performance of the type inference algorithm is quite fast.

Column 4 shows the number of subatomicity constraints generated for each benchmark. Column 5 shows the number of type annotations we manually added to some benchmarks. These type annotations enable *Rcc/Sat* to more precisely infer locking information, to ignore infeasible races, and to infer annotations most suitable for *Bohr* when *Rcc/Sat* may choose among multiple correct but incomparable annotations.

Columns 6 through 9 evaluate the precision of our type inference algorithm. Since our inference algorithm could infer the trivial atomicity `cmpd` for each method in unannotated Java programs, we use the following two heuristics to specify which methods and code blocks should be atomic:

(1) The *exported methods* heuristic states that all public or package-level methods should be atomic, with the exceptions of `main` and `run`, which are starting points for new threads.
(2) The *synchronized blocks* heuristic states that all synchronized methods and synchronized blocks should be atomic. Experience indicates that nonatomic synchronized blocks are often a source of errors.

For each heuristic, columns 6 and 8 of Table II show the number of methods and code blocks which that heuristic states should be atomic, and columns 7 and 9 show the number of potential atomicity violations reported.

8.1 Standard Library Classes

Table II contains two groups of benchmarks. The first group contains a number of classes from the Sun JDK 1.4.2 library and Doug Lea's concurrency package [Lea 2004] that are intended to be atomic (i.e., all exported methods are atomic, regardless of the calling context). Since our implementation infers atomicities for all methods in the target class's supertypes, the "Size" column includes the size of the class and all of its supertypes.

Our type inference system was able to verify the atomicity of the vast majority of methods in these classes. For example, the exported methods heuristic suggests that 68 methods in the `java.lang.String` benchmark should be atomic; our system validated the atomicity of all but the previously described `hashCode` method. Our system verified the atomicity of all methods in `java.lang.StringBuffer`, except for `append`. It also identified the known problems in `PrintWriter`.

```
interface Collection⟨ghost x⟩ {
  (x = always_held) ? atomic : (x ? mover : error) int size();

  (x = always_held) ? atomic : (x ? mover : error) Object[] toArray(Object a[]);
}

class Vector ... {
  Object elementData[] guarded_by this;
  int elementCount guarded_by this;

  (y = always_held) ? cmpd : (y ? mover : error) Vector⟨ghost y⟩(Collection⟨y⟩ c) {
    elementCount = c.size();
    elementData = new Object[...];
    c.toArray(elementData);
  }
  ...
}
```

Fig. 15. Atomicities for `Vector` and `Collection`.

Three new errors were detected in `Vector`, for which *Bohr* inferred precise conditional atomicities, as described in Section 7.3 and 7.4, for methods in the supertypes `AbstractCollection`, `AbstractList`, `SynchronizedCollection`, `List`, and `Collection`. We show in Figure 15 these more precise atomicities for the code snippet previously described in Section 5. (*Bohr* also infers protecting lock ghost variables for `Collections` and `Vectors`, but we omit them from this example since they do not affect the example shown.) *Bohr* detected similar atomicity violations in the `removeAll` and `retainAll` methods of both `Vector` and `SynchronizedList`, which is a synchronized wrapper class for `List` objects.

The warnings for the remaining classes involve subtle synchronization patterns that are not verifiable with our current analysis, but which appear to be correct.

8.2 Complete Programs

The second benchmark group contains complete programs that were previously used to evaluate the Atomizer dynamic atomicity checker [Flanagan and Freund 2004a]. *Bohr* reported warnings on pieces of code that may not execute atomically, even with the guarantee that the locking discipline inferred by *Rcc/Sat* for a whole, unannotated program is never violated.

We originally expected that *Bohr* would issue significantly more warnings than the Atomizer, due to (1) the greater coverage of the static approach, and (2) the inherent approximations of any static analysis. However, the *Bohr* warnings are only slightly higher than the Atomizer in most cases, suggesting that *Bohr* may scale to checking large programs as easily as our dynamic checker, but with stronger safety guarantees. The *Bohr* warnings also differed to some degree from the Atomizer's, because the *Rcc/Sat* subroutine performs an escape analysis not present in the Atomizer.

The number of warnings for jbb reported by *Bohr* was significantly higher than the number reported by the Atomizer. We do not yet understand jbb's synchronization discipline sufficiently well to confidently classify these warnings as either real errors or false alarms. However, many of them appear to be spurious warnings triggered by unusual allocation and initialization patterns that cannot be handled precisely by *Rcc*/*Sat*.

Our experimental results corroborate our earlier findings with the Atomizer that atomicity is a widely-used programming discipline in multithreaded programs. In particular, the results for the *exported methods* heuristic suggest that the vast majority of exported methods in multithreaded applications are atomic.

The *synchronized blocks* heuristic revealed a previously-known defect in the computation of a checksum in raytracer [O'Callahan and Choi 2003; Flanagan and Freund 2004a].

8.3 Limitations

Bohr has been quite successful at inferring atomicity specifications and identifying atomicity violations, but there are some limitations to this approach.

Incorrect Specifications. Bohr relies on programmer-supplied specifications. If a programmer does not properly annotate which methods should be atomic, *Bohr* may not find some concurrency problems. For example, a programmer may have mistakenly assumed that only synchronized methods needed to be atomic when annotating the following version of Account:

```
class Account {
  int balance = 0;
  atomic synchronized int read() { return balance; }
  atomic synchronized void set(int b) { balance = b; }
  void deposit(int amt) {
    int b = read();
    set(b + amt);
  }
}
```

Clearly, deposit should be atomic as well, but *Bohr* would not report any warnings. However, even with this potential pitfall, the tool can be quite useful. In a number of cases we examined, such as the StringBuffer.append() method in Section 5, annotating only synchronized methods as atomic would have uncovered concurrency errors.

Moreover, there are a number of ways in which to make specifications less likely to include errors. For example, we could make all methods atomic by default, making false negatives due to missing annotations less likely. A *specification review* process could also help identify incorrect atomicity specifications.

Relaxed Memory Models and Race Conditions. A programming language's memory model [Gharachorloo 1995] can also impact the ability of *Bohr* to

reason about programs. Our analysis is sound for any program executed under a sequentially-consistent memory model. It is also sound for any program executed under a relaxed memory model, provided that the program does not have any race conditions on nonvolatile data.

However, the analysis is not necessarily sound for programs that contain race conditions when they are executed under a relaxed model. Such race conditions can result quite subtle behavior, which we currently do not model in our semantics and type system for the sake of simplicity. We believe AtomicJava can be extended to properly reason about data races in relaxed models, such as the Java Memory Model [Manson et al. 2005], but we leave this for future work.

Synchronization Idioms. Bohr currently reasons about mutex locks quite effectively, but large programs typically employ additional synchronization idioms that our analysis cannot currently handle. For example, condition variables (implemented with Java's `wait` and `notify` operations) can be used to ensure mutual exclusion and to implement synchronization barriers [Birrell 1989]. We currently verify the correctness of these idioms manually and then add annotations to suppress race condition warnings resulting from these features.

Bohr also has trouble with nonblocking data structures, such as those implemented in the Java 1.5 `java.util.concurrent` package [Goetz et al. 2006], as well as synchronization patterns that vary over time as the program executes. Since *Bohr* attempts to identify a single, fixed lock protecting each field, it will generate spurious warnings when the lock protecting a field varies over time. Including a more precise race condition analysis, such as a *happens-before* analysis [Lamport 1978], could alleviate some of these limitations.

Irreducible Code Sequences. Bohr may also report false alarms on code paths that are not reducible yet are still conceptually atomic. For example, performance counters typically are not protected by locks, under the assumption that the resulting race conditions will not significantly affect the final counter values. Any method that increments such a counter will be considered `compound`, despite the race conditions not impacting the overall correctness at an abstract level that ignores these performance counters.

A more interesting situation is the following method `alloc`, which is modeled after code that searches for a free disk block in a file system. The flag `free[i]` indicates whether the `i`-th disk block is currently unused, and this flag is protected by `lock[i]`. When `alloc` identifies a free block, it allocates the block by setting the appropriate bit to false and returns the index of that block. The method returns -1 if it fails to find a free block.

```
Object lock[];
boolean free[]; // free[i] guarded_by locks[i]

atomic int alloc() {
  for (int i = 0; i < max; i++) {
    synchronized(lock[i]);
```

```
            if (free[i]) {
              free[i] = false;
              return i;
            }
          }
        }
        return -1;
      }
```

The method is not `atomic`, since there exist some nonserial executions of this method that are not equivalent to any serial executions. In particular, concurrent calls to `alloc` and a method to free a block are not serializable, since the exact interleaving of steps by each method could impact which block the `alloc` method returns. However, `alloc` *is* atomic in an abstract sense because any execution performs the atomic action of either allocating a block or returning -1. Currently *Bohr* cannot reason about the fine-grained locking used in this example or about the abstract atomicity property of `alloc`, and we are forced to insert "`no_warn`" annotations to avoid reporting warnings.

We have explored an analysis to identify some abstractly atomic methods by exploiting properties of *pure* code blocks that have no side effects visible to other threads [Flanagan et al. 2005]. In essence the analysis is able to ignore any iteration of the loop inside `alloc` that does not change program state visible outside the current thread.

9. RELATED WORK

Lipton [1975] first proposed reduction as a way to reason about concurrent programs without considering all possible interleavings. He focused primarily on checking deadlock freedom. Doeppner [1977], Back [1989], and Lamport and Schneider [1989] extended this work to allow proofs of general safety properties. Cohen and Lamport [1998] extended reduction to allow proofs of liveness properties. Misra [2001] has proposed a reduction theorem for programs built with monitors [Hoare 1974] communicating via procedure calls. Bruening [1999] and Stoller [2000] have used reduction to improve the efficiency of model checking. Flanagan and Qadeer [2003a] have pursued a similar approach, and Qadeer et al. [2004] have used reduction to infer procedure summaries in concurrent programs.

A number of tools have been developed for detecting race conditions, both statically and dynamically. Our previous work on `rccjava` [Abadi et al. 2006] uses a type system to catch race conditions in Java programs. This approach has been extended [Boyapati and Rinard 2001; Boyapati et al. 2002] and adapted to other languages [Grossman 2003]. Other static race detection tools include Warlock [Sterling 1993] and Locksmith [Pratikakis et al. 2006] for ANSI C programs, Chord [Naik et al. 2006], and ESC/Java [Flanagan et al. 2002], 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 [Burrows and Leino 2002]. Vault [DeLine and Fähndrich 2001] is a system designed to

check resource management protocols, and lock-based synchronization can be considered to be such a protocol. Aiken and Gay [1998] also investigate static race detection, in the context of SPMD programs.

Eraser [Savage et al. 1997] detects race conditions and deadlocks dynamically, rather than statically. The Eraser algorithm has been extended to object-oriented languages [von Praun and Gross 2001] and has been improved for precision and performance [Choi et al. 2002]. Agarwal and Stoller [2004] present a dynamic type inference technique for the type system of Boyapati and Rinard [2001].

A variety of other approaches have been developed for race and deadlock prevention; they are discussed in more detail in earlier papers [Flanagan and Abadi 1999b; Flanagan and Freund 2000; Abadi et al. 2006].

Since an atomicity describes aspects of the behavior or effect [Lucassen and Gifford 1988] of an expression, we are essentially performing a form of effect reconstruction [Tofte and Talpin 1994; Talpin and Jouvelot 1992]. However, our atomicities are quite different from traditional effects; in particular, our atomicities may include program variables and expressions, and thus we have *dependent effects*. Similarly, our parameterized classes are actually dependent types. Cardelli [1988] was among the first to explore type checking for dependent types. Our dependent types and effects are comparatively limited in expressive power, but the resulting type checking and type inference problems are decidable.

In summary, 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.

Sasturkar et al. [2005] also present a type inference algorithm for atomicity. Their type system also extends Flanagan and Qadeer [2003b] with parameterized classes [Flanagan and Freund 2000]. Unlike our system, their system includes a notion of object ownership [Boyapati and Rinard 2001], but does not, for example, support protected locks. In contrast to our static type inference algorithm, they use a dynamic analysis to infer race condition information and ghost parameters.

Freund and Qadeer [2004] combined both reduction and simulation in the Calvin checker to verify functional procedure specifications in multithreaded programs. Our atomic type system is inspired by the Calvin checker, but represents a different point in the trade-off 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. We have explored adding abstraction based on purity to a type system for atomicity [Flanagan et al. 2005]. A pure block of code does not change the program state under normal termination and can be removed from the program trace before reduction. This notion may reduce spurious warnings in some cases.

The use of model checking for verifying atomicity had been explored by Hatcliff et al. [2004]. This model checking approach is more expressive than our type-based analysis, but is vulnerable to state-space explosion. Their results

suggest that verifying atomicity via model-checking is feasible for unit-testing. A more general (but more expensive) technique for verifying atomicity during model checking is *commit-atomicity* [Flanagan 2004].

Several tools have explored verifying atomicity dynamically [Flanagan and Freund 2004a; Wang and Stoller 2003], but these tools are sensitive to test case coverage, unlike our static analysis.

Atomicity is a semantic correctness condition for multithreaded software. In this respect, it is similar to strict serializability [Papadimitriou 1986] for database transactions and linearizability [Herlihy and Wing 1990] 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.

Other languages have included a notion of atomicity as a primitive operation. Hoare [1972] and Lomet [1977] first proposed the use of atomic blocks for synchronization, and the Argus [Liskov et al. 1987] and Avalon [Eppinger et al. 1991] projects developed language support for implementing atomic objects. Recent approaches to supporting atomicity also include lightweight transactions [Harris and Fraser 2003; Welc et al. 2004; Ringenburg and Grossman 2005] and automatic generation of synchronization code from high-level specifications [Deng et al. 2002] or atomicity specifications [McCloskey et al. 2006; Vaziri et al. 2006; Hicks et al. 2006].

10. CONCLUSIONS

Atomicity facilitates the validation of multithreaded programs by reducing the number of thread interleavings that need to be considered, since each atomic method can be considered to execute sequentially. However, verifying atomicity can be nontrivial. Previous approaches were limited by test case coverage [Flanagan and Freund 2004a; Wang and Stoller 2003] or to systems with small states spaces [Hatcliff et al. 2004].

The primary contribution of this article is a type-based approach for specifying and checking atomicity properties in concurrent programs. Our analysis is scalable to larger systems than previous static approaches and provides stronger guarantees by performing all checking statically. However, using the type system by itself does require substantial assistance from the programmer in order to annotate the source code.

Our second main contribution is a type inference algorithm that removes this burden from the programmer by automatically inferring the most precise atomicity for each unannotated method in a program. When used in conjunction with *Rcc/Sat* [Flanagan and Freund 2004b], this inference algorithm can identify atomicity violations in unannotated source code by inferring both the code's synchronization discipline and atomicity properties. Our tool, *Bohr*, thus provides a convenient and effective means to verify many atomicity properties in large programs. For example, it can verify that 85% of the exported methods in `jbb` (our largest benchmark) are atomic.

Additional extensions to the type system and inference algorithms may help reduce the number of false alarms and improve precision. For example, *Bohr* only reasons about mutual exclusion locks and not other synchronization techniques, such as `wait` and `notify` or nonblocking data structures. The former could be handled by replacing our lockset-based race-condition analysis [Savage et al. 1997; Flanagan and Freund 2000] with a happens-before analysis [Lamport 1978] to identify conflicting accesses. Recent studies suggest that, while more expensive, such an analysis may be tractable [von Praun and Gross 2003]. Nonblocking data structures, and other synchronization primitives such as *compare-and-swap* or *load-linked / store-conditional*, can be handled by applying reduction to an abstraction of the program [Flanagan et al. 2005; Sasturkar et al. 2005]. However, these techniques add significant complexity to the analysis and their effectiveness in practice remains to be seen.

Once atomicity errors are identified, the programmer must still fix them, which may not be straightforward in all cases. Another avenue for future work is to explore synchronization correction algorithms that not only identify atomicity errors statically, but also give hints to the programmer as how to correct them [Flanagan and Freund 2005].

Recent studies on lightweight transactions [Harris and Fraser 2003; Welc et al. 2004; Ringenburg and Grossman 2005] offers an alternative mechanism for ensuring atomicity. Current work on improving the performance of transactions may enable programmers to forego lock-based synchronization altogether in some cases. However, we believe that a synthesis of transactions and programmer-supplied synchronization code will be the most effective programming methodology in the future, and ensuring the correctness of code with explicit synchronization will continue to be important for ensuring the correctness of concurrent software.

ACKNOWLEDGMENTS

We thank Martín Abadi, Dan Grossman, Shriram Krishnamurthi, Sanjit Seshia, and Scott Stoller for comments on this work.

REFERENCES

Abadi, M., Flanagan, C., and Freund, S. N. 2006. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst. 28*, 2, 207–255.

Agarwal, R. and Stoller, S. D. 2004. Type inference for parameterized race-free Java. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*. 149–160.

Aiken, A. and Gay, D. 1998. Barrier inference. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 243–354.

Artho, C., Havelund, K., and Biere, A. 2003. High-level data races. In *Proceedings of the First International Workshop on Verification and Validation of Enterprise Information Systems*.

Back, R.-J. 1989. A method for refining atomicity in parallel algorithms. In *Proceedings of the Parallel Architectures and Languages Europe (PARLE'89)*. Lecture Notes in Computer Science, vol. 366. Springer-Verlag, 199–216.

Birrell, A. D. 1989. An introduction to programming with threads. Res. rep. 35, Digital Equipment Corporation Systems Research Center.

BOYAPATI, C., LEE, R., AND RINARD, M. 2002. A type system for preventing data races and deadlocks in Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 211–230.

BOYAPATI, C. AND RINARD, M. 2001. A parameterized type system for race-free Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 56–69.

BRUENING, D. 1999. Systematic testing of multithreaded Java programs. M.S. thesis, Massachusetts Institute of Technology.

BRYANT, R. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. C-35*, 8, 677–691.

BURROWS, M. AND LEINO, K. R. M. 2002. Finding stale-value errors in concurrent programs. Technical Note 2002-004, Compaq Systems Research Center.

CARDELLI, L. 1988. Typechecking dependent types and subtypes. Lecture Notes in Computer Science, Foundations of Logic and Functional Programming. 45–57.

CHAMILLARD, A. T., CLARKE, L. A., AND AVRUNIN, G. S. 1996. An empirical comparison of static concurrency analysis techniques. Tech. rep. 96-084, Department of Computer Science, University of Massachusetts at Amherst.

CHOI, J.-D., GUPTA, M., SERRANO, M. J., SREEDHAR, V. C., AND MIDKIFF, S. P. 1999. Escape analysis for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 1–19.

CHOI, J.-D., LEE, K., LOGINOV, A., O'CALLAHAN, R., SARKAR, V., AND SRIDHARA, M. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 258–269.

COHEN, E. AND LAMPORT, L. 1998. Reduction in TLA. In *Proceedings of the International Conference on Concurrency Theory*. Lecture Notes in Computer Science, vol. 1466. Springer-Verlag, 317–331.

CORBETT, J. C. 1996. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng. 22*, 3, 161–180.

DELINE, R. AND FÄHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 59–69.

DENG, X., DWYER, M., HATCLIFF, J., AND MIZUNO, M. 2002. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the International Conference on Software Engineering*. 442–452.

DETLEFS, D. L., LEINO, K. R. M., AND NELSON, C. G. 1998. Wrestling with rep exposure. Research rep. 156, DEC Systems Research Center.

DOEPPNER, JR., T. W. 1977. Parallel program correctness through refinement. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 155–169.

EPPINGER, J. L., MUMMERT, L. B., AND SPECTOR, A. Z. 1991. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann.

FLANAGAN, C. 2004. Verifying commit-atomicity using model-checking. In *Proceedings of the International SPIN Workshop on Model Checking of Software*.

FLANAGAN, C. AND ABADI, M. 1999a. Object types against races. In *Proceedings of the International Conference on Concurrency Theory*. Lecture Notes in Computer Science, vol. 1664. 288–303.

FLANAGAN, C. AND ABADI, M. 1999b. Types for safe locking. In *Proceedings of European Symposium on Programming*. Lecture Notes in Computer Science, vol. 1576. 91–108.

FLANAGAN, C. AND FREUND, S. N. 2000. Type-based race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 219–232.

FLANAGAN, C. AND FREUND, S. N. 2004a. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 256–267.

FLANAGAN, C. AND FREUND, S. N. 2004b. Type inference against races. In *Proceedings of the Static Analysis Symposium*. 116–132.

FLANAGAN, C. AND FREUND, S. N. 2005. Automatic synchronization correction. In *Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages*.

FLANAGAN, C., FREUND, S. N., AND LIFSHIN, M. 2005. Type inference for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*. 47–58.

FLANAGAN, C., FREUND, S. N., AND QADEER, S. 2005. Exploiting purity for atomicity. *IEEE Trans. Softw. Eng. 31*, 4, 275–291.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 234–245.

FLANAGAN, C. AND QADEER, S. 2003a. Transactions for software model checking. In *Proceedings of the Workshop on Software Model Checking*.

FLANAGAN, C. AND QADEER, S. 2003b. A type and effect system for atomicity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 338–349.

FLANAGAN, C. AND QADEER, S. 2003c. Types for atomicity. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation*. 1–12.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 171–183.

FREUND, S. N. AND QADEER, S. 2004. Checking concise specifications for multithreaded software. *J. Object Tech. 3*, 6, 81–101.

GHARACHORLOO, K. 1995. Memory consistency models for shared-memory multiprocessors. Ph.D. thesis, Stanford University.

GOETZ, B., PEIERLS, T., BLOCH, J., BOWBEER, J., HOLMES, D., AND LEA, D. 2006. *Java Concurrency in Practice*. Addison-Wesley.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley.

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

HAACK, C. AND WELLS, J. B. 2003. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the European Symposium on Programming*. 284–301.

HARRIS, T. AND FRASER, K. 2003. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 388–402.

HATCLIFF, J., ROBBY, AND DWYER, M. B. 2004. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*. 175–190.

HERLIHY, M. P. AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3, 463–492.

HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. 2006. Inferring locking for atomic sections. In *Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*.

HOARE, C. 1974. Monitors: an operating systems structuring concept. *Comm. ACM 17*, 10, 549–557.

HOARE, C. A. R. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*. A.P.I.C. Studies in Data Processing, vol. 9. 61–71.

JAVA GRANDE FORUM. 2003. Java Grande benchmark suite. `http://www.javagrande.org/`.

JAVASOFT. 2005. Java Developers Kit, version 1.4.0. `http://java.sun.com`.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM 21*, 7, 558–565.

LAMPORT, L. AND SCHNEIDER, F. B. 1989. Pretending atomicity. Research rep. 44, DEC Systems Research Center.

LEA, D. 2004. The `util.concurrent` package, release 1.3.4. `http://gee.cs.oswego.edu/dl/`.

LIPTON, R. J. 1975. Reduction: A method of proving properties of parallel programs. *Comm. ACM 18*, 12, 717–721.

LISKOV, B., CURTIS, D., JOHNSON, P., AND SCHEIFLER, R. 1987. Implementation of Argus. In *Proceedings of the Symposium on Operating Systems Principles*. 111–122.

LOMET, D. B. 1977. Process structuring, synchronization, and recovery using atomic actions. *Lang. Design Reliable Softw.*, 128–137.

LUCASSEN, J. M. AND GIFFORD, D. K. 1988. Polymorphic effect systems. In *Proceedings of the ACM Conference on Lisp and Functional Programming*. 47–57.

MANSON, J., PUGH, W., AND ADVE, S. V. 2005. The Java memory model. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 378–391.

MCCLOSKEY, B., ZHOU, F., GAY, D., AND BREWER, E. 2006. Autolocker: synchronization inference for atomic sections. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 346–358.

MISRA, J. 2001. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications*. Springer-Verlag.

NAIK, M., AIKEN, A., AND WHALEY, J. 2006. Effective static race detection for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 308–319.

O'CALLAHAN, R. AND CHOI, J.-D. 2003. Hybrid dynamic data race detection. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*. 167–178.

PAPADIMITRIOU, C. 1986. *The Theory of Database Concurrency Control*. Computer Science Press.

POZNIANSKY, E. AND SCHUSTER, A. 2003. Efficient on-the-fly data race detection in multihreaded C++ programs. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*. 179–190.

PRATIKAKIS, P., FOSTER, J. S., AND HICKS, M. 2006. Context-sensitive correlation analysis for detecting races. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 320–331.

QADEER, S., RAJAMANI, S. K., AND REHOF, J. 2004. Summarizing procedures in concurrent programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 245–255.

RINGENBURG, M. F. AND GROSSMAN, D. 2005. AtomCaml: first-class atomicity via rollback. In *Proceedings of the ACM International Conference on Functional Programming*. 92–104.

SALCIANU, A. AND RINARD, M. 2001. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. 12–23.

SASTURKAR, A., AGARWAL, R., WANG, L., AND STOLLER, S. D. 2005. Automated type-based analysis of data races and atomicity. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*. 83–94.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. E. 1997. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Trans. Comput. Syst. 15*, 4, 391–411.

SPEC. 2000. Standard Performance Evaluation Corporation JBB2000 Benchmark. Available from `http://www.spec.org/osg/jbb2000/`.

STERLING, N. 1993. Warlock: A static data race analysis tool. In *Proceedings of the USENIX Winter Technical Conference*. 97–106.

STOLLER, S. 2006. Personal communication.

STOLLER, S. D. 2000. Model-checking multi-threaded distributed Java programs. In *Proceedings of the Workshop on Model Checking and Software Verification*. Lecture Notes in Computer Science, vol. 1885. Springer-Verlag, 224–244.

TALPIN, J.-P. AND JOUVELOT, P. 1992. Polymorphic type, region and effect inference. *J. Funct. Program. 2*, 3, 245–271.

TOFTE, M. AND TALPIN, J.-P. 1994. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 188–201.

VAZIRI, M., TIP, F., AND DOLBY, J. 2006. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 334–345.

VON PRAUN, C. AND GROSS, T. 2001. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 70–82.

VON PRAUN, C. AND GROSS, T. 2003. Static conflict analysis for multi-threaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 115–128.

WAND, M. 1986. Finding the source of type errors. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 38–43.

WANG, L. AND STOLLER, S. D. 2003. Runtime analysis for atomicity. In *Proceedings of the Workshop on Runtime Verification*.

WANG, L. AND STOLLER, S. D. 2006. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng. 32*, 2, 93–110.

WELC, A., JAGANNATHAN, S., AND HOSKING, A. L. 2004. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*. 519–542.

YANG, J., MICHAELSON, G., TRINDER, P., AND WELLS, J. B. 2000. Improved type error reporting. In *Proceedings of the International Workshop on Implementation of Functional Languages*. 71–86.

Received July 2006; revised April 2007; accepted April 2007