Type Inference For Atomicity

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

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

Abstract

Atomicity is a fundamental correctness property in multithreaded programs. This paper presents an algorithm for verifying atomicity via type inference. The underlying type system supports guarded, write-guarded, and unguarded fields, as well as thread-local data, parameterized classes and methods, and protected locks. We describe an implementation of this algorithm for Java and discuss its performance and usability on benchmarks totaling sixty thousand lines of code.

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

General Terms: Languages, Verification, Reliability.

Keywords: Atomicity, type inference, reduction, concurrency.

This is an extended and revised version of our paper appearing in the Proceedings of TLDI '05.

1 Introduction

Reasoning about the correctness of multithreaded code is extremely difficult, due to the need to consider all possible interleavings of the various threads. In particular, errors often occur in multithreaded programs because certain interleavings cause unexpected interactions between concurrent threads. Thus, methods for specifying and controlling the interference between threads are crucial for the development of reliable multithreaded software.

This paper focuses on the non-interference property of *atomicity*. A method is atomic if, for every arbitrarily interleaved program execution, there is an equivalent execution with the same overall behavior where the atomic method is executed serially, that is, the method's execution is not interleaved with actions of other threads. The key benefit of atomicity is that atomic methods are amenable to sequential reasoning, which significantly facilitates standard validation techniques such as manual code inspection, dynamic testing, and static analysis. In addition, atomicity violations often reveal subtle errors in a program's synchronization discipline.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
TLDI'05, January 10, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-999-3/05/0001 ...$5.00

Over the last few years, several techniques for verifying atomicity have been developed, including via type systems [15, 16, 13], dynamic analysis [11, 38], theorem proving [18], and model checking [22]. The type-based approach seems particularly promising, since it avoids both the test coverage limitations of dynamic analyses and the scalability limitations of some other analyses. However, it does require the programmer to provide type annotations specifying, for example, the atomicity of every method and the protecting lock of every field in the program, which significantly increases the cost of using the system on large programs.

In this paper we present a type inference algorithm for atomicity. Our inference algorithm proceeds in two phases. The first phase reasons about potential race conditions, and infers the synchronization discipline used by the program, including which locks (if any) protect each field. This task is accomplished using the *Rcc/Sat* subroutine, which is described in an earlier paper [12]. (Essentially, this part of the type inference problem is NP-complete, and *Rcc/Sat* works via reduction to propositional satisfiability.)

The primary contribution of this paper is the second phase of our type inference algorithm. This phase infers the most precise atomicity (or effect [28]) for each method, using a constraint-based analysis. This phase 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 dependent atomicities must not change during execution. Our constraint language includes special constructs to describe 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.

A second limitation of our original type system for atomicity [15] is its limited expressiveness, particularly for large programs that reuse classes in contexts with different synchronization disciplines. In this paper, we overcome this limitation by basing our development on a more expressive type system that supports thread-local objects and parameterized classes and methods, adapting techniques from our earlier RFJ2 type system for race freedom [12, 10]. In addition, our type system also supports the notion of *protected locks*, which our experiments with the Atomizer dynamic analysis tool [11] revealed are often necessary for verifying atomicity in large programs.

We have implemented our type inference algorithm for the Java programming language and evaluated its performance on a variety of benchmarks totaling over 60,000 lines of code. This algorithm works well; in particular, due to the increased expressiveness of our type system, our algorithm can verify the atomicity of most methods in these benchmarks. These experimental results validate the hypothesis that atomicity is a widely-used programming discipline in multithreaded programs[1]. In addition, these results indicate that this type system for atomicity is sufficiently expressive to accommodate many of the common synchronization patterns of larger Java programs, and that our type inference algorithm is fast and works well in practice. These experiments also revealed a number of defects in these benchmarks, including, for example, three errors in the class `java.util.Vector`.

The presentation of our results proceeds as follows. The following section presents an idealized multithreaded subset of Java. Section 3 describes the atomicities inferred by our type system and Section 4 illustrates these atomicities on some examples. Section 5 describes our constraint-based type inference algorithm. Section 6 describes the implementation of our type inference algorithm, including extensions for expressiveness and to support the Java language. Section 7 evaluates our prototype implementation on our benchmark suite. Section 8 discusses related work, and we conclude with Section 9.

2 The Source Language AJ2

Figure 1: AJ2 Syntax

P	::=	$defn^*\ e$	(program)
$defn$	::=	$\texttt{class}\ cn\langle \texttt{ghost}\ x^* \rangle\ body$	(class declaration)
$body$	::=	$\{\ field^*\ meth^*\ \}$	(class body)
$field$	::=	$t\ fn\ g$	(field declaration)
g	::=	$\texttt{final}$	(field guards)
	|	$\texttt{guarded\_by}\ l$	
	|	$\texttt{write\_guarded\_by}\ l$	
	|	$\texttt{no\_guard}$	
$meth$	::=	$s\ t\ mn\langle \texttt{ghost}\ x^* \rangle \texttt{(}arg^*\texttt{)}\ \{\ e\ \}$	(method declaration)
arg	::=	$t\ x$	(argument declaration)
t, c	::=	$cn\langle l^* \rangle$	(class type)
l	::=	$e \mid \texttt{none}$	(lock expression)
e	::=	$x \mid n \mid \texttt{null} \mid \texttt{new}\ c\texttt{(}e^*\texttt{)} \mid e.fd \mid e.fd\ \texttt{=}\ e$	
	|	$e.mn\langle l^* \rangle \texttt{(}e^*\texttt{)} \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{while}\ e\ e$	
	|	$\texttt{if}\ e\ e\ e \mid \texttt{synchronized}\ e\ e \mid e.\texttt{fork}$	

$x, y \in Var$ $\quad$ $cn \in ClassName$
$fn \in FieldName$ $\quad$ $mn \in MethodName$

We base our formal development on the language AJ2, a multithreaded subset of Java with a type system for atomicity. This type system extends our previous atomicity type system [15] with thread-local objects and parameterized classes and methods [10, 11]. For clarity, AJ2 also simplifies some aspects of our earlier formal development by, for example, not supporting inheritance. (Section 6 describes how our implementation handles inheritance and other aspects of the full Java programming language.)

An AJ2 program (see Figure 1) is a sequence of class declarations together with an initial expression. 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.

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;
- `no_guard`: the field can be read or written at any time;
- `guarded_by` l: the lock denoted by the *lock expression* l must be held on all accesses (reads or writes) of that field; and
- `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.

A lock expression is either an expression that denotes some lock in the program, or the special lock `none`, which is described in Section 3.2.

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

$$\texttt{class}\ cn\langle \texttt{ghost}\ x_1 \ldots x_n \rangle\ \{\ \ldots\ \}$$

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 AJ2 language also supports *parameterized methods*. For example, the declaration

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

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

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

Expressions in the language include object allocation $\texttt{new}\ c\texttt{(}e^*\texttt{)}$, which initializes the new object's fields with its argument values; field read and update; method calls; variable binding and reference; conditionals; loops; and synchronized blocks.

The expression e`.fork` starts a new thread (and always evaluates to `null`). The expression e should evaluate to an object that includes a method `run` that takes 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 `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, AJ2 leverages parameterized methods to reason about thread-local data. This approach replaces the escape analysis embedded in our earlier type system [10].

[1] Previous experiments with the Atomizer [11] provided preliminary evidence supporting this hypothesis, but it was limited by the test coverage concerns inherent in any dynamic analysis.

3 Atomicities

We next briefly review Lipton's theory of left and right-movers [26], upon which our type system is based. An action b is a *right-mover* if, for any execution where the action b performed by one thread is immediately followed by an action c of a concurrent thread, the actions b and c can be swapped without changing the resulting state. Conversely, an action c is a *left-mover* if whenever c immediately follows an action b of a different thread, the actions b and c can be swapped, again without changing the resulting state. We classify operations performed by a thread as (left or right) movers as follows:

Operation	Mover Status
lock acquire	right-mover
lock release	left-mover
access to data protected by a lock	both-mover
access to unprotected data	non-mover

Suppose an execution path through a method contains a sequence of right-movers, followed by at most one non-mover action and then a sequence of left-movers. Then this path can be *reduced* to an equivalent serial execution, with the same resulting state, where the path is executed without any interleaved actions by other threads.

Our type system verifies that every possible path through each atomic method is reducible. It works by assigning to each subexpression an *atomicity* [15] characterizing the behavior of that expression. Atomicities include the basic atomicities of our earlier work, an extended notion of conditional atomicities (to support protected locks), and atomicity variables (introduced for type inference). Figure 2 summarizes these forms, which are explained below.

Figure 2: Atomicity Syntax

b	$::=$	`const` | `mover`	(basic atomicities)
	|	`atomic` | `cmpd` | `error`	
a	$::=$	$b \mid p\,?\,a_1 : a_2$	(atomicities)
p	$::=$	$l \mid$ `isNone`(l)	(lock predicates)
s	$::=$	$a \mid \alpha$	(open atomicities)
α	$\in$	*AtomVar*	(atomicity variables)

3.1 Basic Atomicities

An expression may be assigned one of the *basic atomicities* `const`, `mover`, `atomic`, `cmpd`, or `error` according to the following conditions:

- `const`: The expression does not depend on or change any mutable state. In particular, the repeated evaluation of a `const` expression with a given environment yields the same result. Such expressions include references to immutable variables[2], accesses to final fields of `const` expressions, and calls to `const` methods with `const` arguments.
- `mover`: The evaluation of the expression both left and right commutes with operations of other threads. For example, a field access is a `mover` if no thread can concurrently access the same field.
- `atomic`: The expression can be considered to execute without interleaved actions of other threads.
- `cmpd`: The expression is not atomic.
- `error`: The expression violates the program's locking discipline by, for example, accessing a variable without first acquiring the appropriate lock.

[2] All variables are immutable in AJ2, but only final variables are in Java.

Suppose that the basic atomicities b_1 and b_2 reflect the behavior of e_1 and e_2 respectively. Then the *iterative closure* $b_1{}^*$ reflects the behavior of executing e_1 an arbitrary number of times and is defined as follows:

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

Similarly, the *sequential composition* $b_1;b_2$ reflects the behavior of executing $e_1;e_2$, and it is defined by the table.

$\cdot;\cdot$	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. The atomicity $b_1 \sqcup b_2$ reflects the non-deterministic choice between executing either e_1 or e_2.

3.2 Conditional and Open Atomicities

In some cases, the atomicity of an expression may depend on a certain *lock predicate* p. For this purpose, we introduce *conditional atomicities* of the form

$$p\ ?\ a_1\ :\ a_2$$

The lock predicate p may be simply an expression l that denotes a lock in the program. In this case, this conditional atomicity is equivalent to atomicity a_1 if the lock l is currently held, and is equivalent to a_2 if the lock is not held. For soundness, the lock expression l must always denote the same lock, and so we require that l has atomicity `const`. For example, the expression `this` is `const`, as is `this.f`, provided `f` is a final field.

Our experiments with the Atomizer dynamic analysis tool [11] revealed that, due to the layers of abstraction used in large programs, locks used in one data structure are often subsumed by additional locks in a containing data structure. To precisely reason about such redundant or protected locks, our system allows the type of a lock l to specify that l is *protected* by another lock. In particular, if l has type $cn\langle m, \ldots\rangle$, then m is the protecting lock for l. In this case, m must be held whenever l is acquired, and acquires and releases of l can be more precisely characterized as both-movers, since there are no conflicts on accesses to l.

We express the situation where l is not a protected lock by saying that its protecting lock m is `none`. We use the lock predicate `isNone`(m) to check if l has a protected lock. For example, if the atomicity of e is `mover`, then the atomicity of `synchronized` $l\ e$ is:

$$\texttt{isNone}(m)\,?\,\texttt{atomic}:(m\,?\,\texttt{mover}:\texttt{error})$$

That is, if l is an unprotected lock, then $m =$ `none` and the synchronized block is `atomic`. Alternatively, if l is protected then the protecting lock m must be held and the synchronized block has atomicity `mover`. If the protecting lock is not held, then the synchronized block has atomicity `error`, since the synchronization discipline is being violated.

Technically, a field could be declared as `guarded_by none`, but since the lock `none` is not an expression, this lock could never be acquired and such a field could never be accessed.

An *atomicity* a is either a basic atomicity b or the conditional atomicity $p\,?\,a_1 : a_2$. We extend the iterative closure, sequential composition, and join operations to conditional atomicities as follows:

$$\begin{array}{rcl}
(p\,?\,a_1 : a_2)^* & = & p\,?\,a_1{}^* : a_2{}^* \\
(p\,?\,a_1 : a_2); a_3 & = & p\,?\,a_1; a_3 : a_2; a_3 \\
b; (p\,?\,a_1 : a_2) & = & p\,?\,(b; a_1) : (b; a_2) \\
(p\,?\,a_1 : a_2) \sqcup a_3 & = & p\,?\,(a_1 \sqcup a_3) : (a_2 \sqcup a_3) \\
b \sqcup (p\,?\,a_1 : a_2) & = & p\,?\,(b \sqcup a_1) : (b \sqcup a_2)
\end{array}$$

We also extend the subatomicity ordering to conditional atomicities, using the auxiliary relation $\sqsubseteq^t_f$, where t is a set of lock predicates known to be true and f is a set of lock predicates known to be false. We define $a_1 \sqsubseteq a_2$ to be $a_1 \sqsubseteq^{\{\texttt{isNone}(\texttt{none})\}}_{\emptyset} a_2$ and check $a_1 \sqsubseteq^t_f a_2$ recursively as follows:

$$\frac{b_1 \sqsubseteq b_2}{b_1 \sqsubseteq^t_f b_2} \qquad \frac{\begin{array}{c}(p \notin f \Rightarrow a_1 \sqsubseteq^{t \cup \{p\}}_f a_3) \\ (p \notin t \Rightarrow a_2 \sqsubseteq^t_{f \cup \{p\}} a_3)\end{array}}{(p\,?\,a_1 : a_2) \sqsubseteq^t_f a_3} \qquad \frac{\begin{array}{c}(p \notin f \Rightarrow b \sqsubseteq^{t \cup \{p\}}_f a_1) \\ (p \notin t \Rightarrow b \sqsubseteq^t_{f \cup \{p\}} a_2)\end{array}}{b \sqsubseteq^t_f (p\,?\,a_1 : a_2)}$$

To support type inference, we introduce type variables α, which may be mentioned in source programs. During type inference, each atomicity variable is resolved to some atomicity. An *open atomicity* s is either an atomicity or an atomicity variable.

4 Examples

4.1 List Example

To illustrate the properties our type system can verify, consider the class `List` of Figure 3, which implements a linked list of `ListElem`s. This example uses integers and sequential composition, which we treat in the expected fashion. This example includes annotations (`guarded_by` clauses and class parameters) inferred by *Rcc/Sat*. For example, the class `ListElem` is parameterized by a lock, called `x`, that protects the `num` and `next` fields, as indicated by the `guarded_by x` annotations.

The atomicities inferred by our inference algorithm are underlined. The atomicity (`x?mover:error`) inferred for `ListElem.get` is a conditional atomicity. It states that, if the lock `x` is not held, then a call to `get` has atomicity `error`, meaning that this call violates the program's synchronization discipline. In other words, the lock `x` should be held before any call to `get`. In the case where `x` is held, `get` is a `mover`, meaning that the execution of `get` commutes with actions of concurrent threads.

The execution of `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`), and (4) a left-mover (the lock release). Hence `List.get` is at most `atomic`. For the case where the lock `this` is already held, the re-entrant locking operations are both-movers, and so `List.get` is then

Figure 3: Class `List` with Locking and Atomicity Annotations

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

α1 :   (x?mover:error)
       int get() { return this.num; }
     }

     class List {
       ListElem⟨this⟩ elems guarded_by this;

α2 :   (this?mover:atomic)
       void add(int v) {
         synchronized (this) {
           this.elems = new ListElem⟨this⟩(v,this.elems);
         }
       }

α3 :   (this?mover:cmpd)
       void addTwo(int i,int j){this.add(i); this.add(j);}

α4 :   (this?mover:atomic)
       int get() {
         synchronized (this) { return this.elems.get(); }
       }
     }
```

a `mover`. Our type inference system infers the precise atomicity (`this?mover:atomic`) for `List.get` (and similarly for `List.add`).

The atomicity inferred for `List.addTwo` is (`this?mover:cmpd`), indicating an atomicity violation. If the lock `this` is not held, the atomicity `cmpd` means that interleaved actions of concurrent threads may affect the behavior and correctness of `addTwo`, even though there are no race-conditions. In particular, if a concurrent thread also adds entries to the list, then `addTwo` will not achieve its intended behavior of adding its arguments to the list consecutively.

4.2 Protected Locks

The need for protecting locks is illustrated by the class `Set` in Figure 4, which is implemented using a synchronized version of the class `List`. The method `Set.add` calls `List.contains` and then `List.add`. In the absence of protecting locks, these two `List` method invocations would each be assigned the atomicity `list?mover:atomic`, and then `Set.add` would have the atomicity `cmpd` when it is called without holding `list`. This imprecise atomicity incorrectly suggests that interleaved actions of concurrent threads may affect the behavior of `Set.add`.

To infer a more precise atomicity for `Set.add`, our type inference system treats the first parameter `x` to the class `List` as a protecting lock for the `List` lock, and infers the atomicity

`isNone(x)?(this?mover:atomic):(x?mover:error)`

for `List.add` and `List.contains`. If `x` is the dummy lock `none`, then these two methods still have atomicity `this?mover:atomic`. If `x` is a real protecting lock for the `List` lock, then `x` must be held on each call to the two `List` methods, which are then `mover`s.

The field `Set.list` is inferred to be of type`List⟨this⟩`, indicat-

Figure 4: Annotations for `List` and `Set`

```
class Set {
  List⟨this⟩ list;
  ...
  (this?mover:atomic) void add(int v) {
    synchronized(this) {
      if (!list.contains(v)) list.add(v);
    }
  }
}

class List⟨ghost x⟩ {
  isNone(x)?(this?mover:atomic):(x?mover:error)
  void add(int v) { synchronized(this) {...} }

  isNone(x)?(this?mover:atomic):(x?mover:error)
  boolean contains(int v) { synchronized(this) {...} }
}
```

ing that the `List` lock is protected by the enclosing `Set` lock.[3] Our analysis then infers the atomicity `this?mover:atomic` for `Set.add`, which guarantees that actions of concurrent threads do not interfere with the behavior of `Set.add`.

5 Type Checking and Type Inference

An AJ2 program is *explicitly-typed* if it contains no 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 follow a *constraint-based* [2] approach to type inference. The type rules perform a syntax-directed traversal of the program to generate a collection of constraints over atomicity variables. A subsequent constraint-solving phase then finds the most precise (minimal) solution to these constraints, or determines that no solution exists, in which case type inference fails. The following subsections describe the constraint language, the type rules that generate these constraints, and our constraint solving algorithm.

5.1 Atomicity Constraints

A *constraint* C is a subatomicity constraint between an *atomicity expression* and an open atomicity. Atomicity expressions include open atomicities as well as syntactic constructs for representing various operations on atomicities, such as sequential composition, join, iteration, and substitution. We use bold symbols such as "**;**" to distinguish a syntactic atomicity construct from the corresponding semantic operation ";" on atomicities.

Figure 5: Atomicity Expressions

C	$::=$	$d \boldsymbol{\sqsubseteq} s$	(atomicity constraints)
d	$::=$	$s \mid d \boldsymbol{;} d \mid d \boldsymbol{\sqcup} d \mid d^{\boldsymbol{*}} \mid p\,\boldsymbol{?}\,d \boldsymbol{:} d$	(atomicity expressions)
	$\mid$	$d \boldsymbol{\cdot} \theta \mid \mathcal{S}(l, l', d) \mid wfa(P, E, d)$	
θ	$::=$	$[\vec{x} := \vec{l}]$	(substitutions)

An atomicity expression is *closed* if it contains no atomicity variables. The meaning function $[\![\cdot]\!]$ maps closed atomicity expressions to atomicities by performing the semantic operation indicated by each syntactic construct. The first five cases are straightforward, and we discuss the delayed substitution atomicity form $d \cdot \theta$, the synchronization form $\mathcal{S}(l, l', d)$, and the well-formed atomicity form $wfa(P, E, d)$ below, where they are used:

$$
\begin{array}{rcl}
[\![\cdot]\!] : \mathit{ClosedAtomExpr} & \rightarrow & \mathit{Atomicity} \\
[\![a]\!] & = & a \\
[\![d_1 \boldsymbol{;} d_2]\!] & = & [\![d_1]\!] ; [\![d_2]\!] \\
[\![d_1 \boldsymbol{\sqcup} d_2]\!] & = & [\![d_1]\!] \sqcup [\![d_2]\!] \\
[\![d^{\boldsymbol{*}}]\!] & = & [\![d]\!]^{*} \\
[\![p\,\boldsymbol{?}\,d_1 \boldsymbol{:} d_2]\!] & = & p\,?\,[\![d_1]\!] : [\![d_2]\!] \\
[\![d \boldsymbol{\cdot} \theta]\!] & = & \theta([\![d]\!]) \\
[\![\mathcal{S}(l, l', d)]\!] & = & S(l, l', [\![d]\!]) \\
[\![wfa(P, E, d)]\!] & = & \mathit{WFA}(P, E, [\![d]\!])
\end{array}
$$

An *assignment* $A : \mathit{AtomVar} \rightarrow \mathit{Atomicity}$ maps atomicity variables to atomicities. We order assignments according to the point-wise extension of the subatomicity relation:

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

An assignment A *satisfies* a constraint C (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:

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

If $A \models C$ for all C in the set of constraints $\bar{C}$, 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.

5.2 AJ2 Type System

The core of the AJ2 type system is a set of rules for reasoning about the judgment:

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

Here, the program P is included to provide access to class declarations, and E is an environment providing types for the free program and ghost variables of the expression e:

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

The type t is the type inferred for e; d is the atomicity inferred for e; and $\bar{C}$ is the set of constraints generated from this expression. The complete set of type judgments and rules is contained in Figure 6. We briefly describe some of the more important rules.

[LOCK EXP] The judgment $P; E \vdash_{\text{lock}} l : \bar{C}$ checks that l is a well-formed lock expression in environment E. The lock expression l can be either a ghost variable, the dummy lock `none`, 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, we require the size $|e|$ of the lock expression to be bounded by the constant *MaxLockSize*. This requirement 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 ensures termination of our type inference algorithm.

[EXP VAR] A variable access has `const` atomicity, since all variables are immutable in AJ2.

[EXP IF] The atomicity of a conditional expression is the atomicity of the *test* subexpression, sequentially composed with the join

[3] The parameter `this` is inferred by the *Rcc/Sat* phase.

Figure 6: AJ2 Type Rules

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

[EXP NULL]

$$\frac{P;E \vdash c : \bar{C}}{P;E \vdash \texttt{null} : c \cdot \texttt{const} \cdot \bar{C}}$$

[EXP VAR]

$$\frac{\begin{array}{c} P \vdash E : \bar{C} \\ E = E_1, t\, x, E_2 \end{array}}{P;E \vdash x : t \cdot \texttt{const} \cdot \bar{C}}$$

[EXP SYNC]

$$\frac{\begin{array}{c} P;E \vdash l : cn\langle l_{1..n}\rangle \cdot \texttt{const} \cdot \bar{C} \\ P;E \vdash e : t \cdot d \cdot \bar{C}' \\ n = 0 \quad \Rightarrow \quad (d' = \mathcal{S}(l, \texttt{none}, d)) \\ n > 0 \quad \Rightarrow \quad (d' = \mathcal{S}(l, l_1, d)) \end{array}}{P;E \vdash \texttt{synchronized}\ l\ e : t \cdot d' \cdot (\bar{C} \cup \bar{C}')}$$

[EXP WHILE]

$$\frac{\begin{array}{c} P;E \vdash e_i : t_i \cdot d_i \cdot \bar{C}_i \qquad \text{for } i = 1,2 \\ d \ = \ d_1;((d_2;d_1)^*) \end{array}}{P;E \vdash \texttt{while}\ e_1\ e_2 : t_2 \cdot d \cdot (\bar{C}_1 \cup \bar{C}_2)}$$

[EXP REF]

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

[EXP ASSIGN]

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

[EXP NEW]

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

[EXP LET]

$$\frac{\begin{array}{c} P;E \vdash e_1 : t_1 \cdot d_1 \cdot \bar{C}_1 \\ P;E, t_1\ x \vdash e_2 : t_2 \cdot d_2 \cdot \bar{C}_2 \\ \theta = [x := e_1] \\ P;E \vdash \theta(t_2) : \bar{C}_3 \\ \bar{C} = (\bar{C}_1 \cup \bar{C}_2 \cup \bar{C}_3) \\ d = (d_1;wfa(P,E,d_2 \cdot \theta)) \end{array}}{P;E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \theta(t_2) \cdot d \cdot \bar{C}}$$

[EXP INVOKE]

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

[EXP IF]

$$\frac{\begin{array}{c} P;E \vdash e_1 : t_1 \cdot d_1 \cdot \bar{C}_1 \\ P;E \vdash e_i : t \cdot d_i \cdot \bar{C}_i \qquad \text{for } i = 2..3 \\ d \ = \ d_1;(d_2 \sqcup d_3) \\ \bar{C}' = (\bar{C}_1 \cup \bar{C}_2 \cup \bar{C}_3) \end{array}}{P;E \vdash \texttt{if}\ e_1\ e_2\ e_3 : t \cdot d \cdot \bar{C}'}$$

[EXP FORK]

$$\frac{\begin{array}{c} P;E \vdash e : cn\langle l_{1..n}\rangle \cdot d \cdot \bar{C} \\ \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ldots meth \ldots\} \in P \\ meth = a\ t'\ \texttt{run}\langle \texttt{ghost tll}\rangle\texttt{()}\ \{\ e'\ \} \\ a = (\texttt{tll ? cmpd : error}) \\ P;E \vdash t : \bar{C}' \end{array}}{P;E \vdash e.\texttt{fork} : t \cdot (d;\texttt{atomic}) \cdot (\bar{C} \cup \bar{C}')}$$

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

[CLASS]

$$\frac{\begin{array}{c} garg_i = \texttt{ghost}\ x_i \\ E = garg_{1..n}, cn\langle x_{1..n}\rangle\ \texttt{this} \\ P;E \vdash field_i : \bar{C}_i \qquad \forall i \in 1..j \\ P;E \vdash meth_i : \bar{C}'_i \qquad \forall i \in 1..k \\ \bar{C} = (\bar{C}_{1..j} \cup \bar{C}'_{1..k}) \end{array}}{P \vdash \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \{\ field_{1..j}\ meth_{1..k}\} : \bar{C}}$$

$\boxed{P;E \vdash_{\text{lock}} e : \bar{C}}$

[LOCK EXP]

$$\frac{\begin{array}{c} P;E \vdash e : t \cdot \texttt{const} \cdot \bar{C} \\ |e| \leq MaxLockSize \end{array}}{P;E \vdash_{\text{lock}} e : \bar{C}}$$

[LOCK GHOST]

$$\frac{\begin{array}{c} P \vdash E : \bar{C} \\ E = E_1, \texttt{ghost}\ x, E_2 \end{array}}{P;E \vdash_{\text{lock}} x : \bar{C}}$$

[LOCK NONE]

$$\frac{P \vdash E : \bar{C}}{P;E \vdash_{\text{lock}} \texttt{none} : \bar{C}}$$

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

[TYPE C]

$$\frac{\begin{array}{c} \texttt{class}\ cn\langle \texttt{ghost}\ x_{1..n}\rangle\ \ldots \in P \\ P;E \vdash_{\text{lock}} l_i : \bar{C}_i \qquad \forall i \in 1..n \end{array}}{P;E \vdash cn\langle l_{1..n}\rangle : \bar{C}_{1..n}}$$

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

[ENV EMPTY]

$$\frac{}{P \vdash \varepsilon : \emptyset}$$

[ENV X]

$$\frac{P;E \vdash t : \bar{C} \quad x \notin Dom(E)}{P \vdash (E, t\ x) : \bar{C}}$$

[ENV GHOST]

$$\frac{P \vdash E : \bar{C} \quad x \notin Dom(E)}{P \vdash (E, \texttt{ghost}\ x) : \bar{C}}$$

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

[FIELD]

$$\frac{\begin{array}{c} P;E \vdash t : \bar{C}_1 \\ (g \equiv \texttt{guarded\_by}\ l) \ \Rightarrow\ P;E \vdash_{\text{lock}} l : \bar{C}_2 \\ (g \equiv \texttt{write\_guarded\_by}\ l) \ \Rightarrow\ P;E \vdash_{\text{lock}} l : \bar{C}_2 \\ (g \equiv \texttt{final}) \ \Rightarrow\ \bar{C}_2 = \emptyset \\ (g \equiv \texttt{no\_guard}) \ \Rightarrow\ \bar{C}_2 = \emptyset \end{array}}{P;E \vdash t\ fn\ g : (\bar{C}_1 \cup \bar{C}_2)}$$

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

[METHOD]

$$\frac{\begin{array}{c} meth = s\ t\ mn\langle \texttt{ghost}\ x_{1..n}\rangle(arg_{1..d})\ \{\ e\ \} \\ garg_i = \texttt{ghost}\ x_i \qquad \forall i \in 1..n \\ E' = E, garg_{1..n}, arg_{1..d} \\ P;E' \vdash e : t \cdot d \cdot \bar{C} \end{array}}{P;E \vdash meth : (\bar{C} \cup \{d \sqsubseteq s\})}$$

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

[PROG]

$$\frac{\begin{array}{c} ClassOnce(P) \quad FieldsOnce(P) \\ MethodsOnce(P) \\ P = defn_{1..n}\ e \\ P \vdash defn_i : \bar{C}_i \qquad \forall i \in 1..n \\ P;\varepsilon \vdash e : t \cdot d \cdot \bar{C} \end{array}}{P \vdash \bar{C}_{1..n} \cup \bar{C} \cup \{d \sqsubseteq \texttt{cmpd}\}}$$

of the atomicities of the *then* and *else* branches. The generated constraint set is the union of the constraints of the subexpressions.

[EXP SYNC] This rule for a synchronized expression `synchronized` l e checks that l has type $cn\langle l_{1..n}\rangle$ and atomicity `const`. The atomicity of the synchronized expression is $\mathcal{S}(l,l_1,d)$, where d is the atomicity of e and l_1 is the protecting lock (if any) for l. The meaning of this atomicity construct is given by:

$$[\![\mathcal{S}(l,l_1,d)]\!] = S(l,l_1,[\![d]\!])$$

The function $S(l,m,a)$, defined below, determines the atomicity of a synchronized statement `synchronized` l e where m is the protecting lock for l and a is the atomicity of the body e:

$$\begin{array}{lcl}
S(l,m,\texttt{const}) & = & \texttt{isNone}(m)\ ?\ (l\ ?\ \texttt{const} : \texttt{atomic}) \\
 & & \quad : (m\,?\,\texttt{const}:\texttt{error}) \\
S(l,m,\texttt{mover}) & = & \texttt{isNone}(m)\ ?\ (l\ ?\ \texttt{mover} : \texttt{atomic}) \\
 & & \quad : (m\,?\,\texttt{mover}:\texttt{error}) \\
S(l,m,\texttt{atomic}) & = & \texttt{isNone}(m)\ ?\ \texttt{atomic} \\
 & & \quad : (m\,?\,\texttt{atomic}:\texttt{error}) \\
S(l,m,\texttt{cmpd}) & = & \texttt{isNone}(m)\,?\,\texttt{cmpd}:(m\,?\,\texttt{cmpd}:\texttt{error}) \\
S(l,m,\texttt{error}) & = & \texttt{error} \\
S(l,m,(l\,?\,b_1:b_2)) & = & S(l,m,b_1) \\
S(l,m,(p\,?\,b_1:b_2)) & = & p\,?\,S(l,m,b_1):S(l,m,b_2)\ \ \text{if } l \not\equiv p
\end{array}$$

We describe several cases: (1) If the body is a mover and l is an unprotected lock ($m = \texttt{none}$) that is already held, then the synchronized statement is also a mover, since the acquire and release operations are no-ops. (2) If the body is a mover and l is not held but is protected by m, then m must be held and the synchronized statement is also a mover; if m is not held then the synchronized statement has atomicity `error`. (3) If the body is a mover and l is unprotected and not 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). (4) If the body has conditional atomicity $l\,?\,b_1:b_2$, then we ignore b_2 and recursively apply S to b_1, since we know that l is held within the synchronized body. (5) If the body has some other conditional atomicity, then we recursively apply S to both branches.

[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 the atomicity expression d_2 may refer to the let-bound variable x, we apply the substitution $\theta = [x := e_1]$ to yield a corresponding atomicity that does not mention x. Several complications arise here.

First, since d_2 may include an atomicity variable α, we cannot apply the substitution θ immediately because α may later resolve to x. Instead, we use the *delayed substitution form* $d_2 \cdot \theta$ to delay this substitution until after atomicity variables are resolved.

Second, 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 well-formed atomicity form $wfa(P,E,d_2 \cdot \theta)$ to yield a valid atomicity for e_2 that is well-formed in environment E. The atomicity expression $wfa(P,E,d)$ denotes the smallest atomicity that is not less than d and that is a well-formed atomicity in the environment E. Its meaning is defined in terms of the following function $WFA(P,E,a)$.

$$[\![wfa(P,E,d)]\!] = WFA(P,E,[\![d]\!])$$

The function $WFA(P,E,a)$ returns the smallest atomicity a' such that $a \sqsubseteq a'$ and the lock expressions in a' are well-typed in E.

$$\begin{array}{l}
WFA(P,E,b) = b \\
WFA(P,E,p\,?\,a_1:a_2) = \begin{cases} p\,?\,a'_1:a'_2 & \text{if } P;E \vdash_{\text{lock}} l : \bar{C} \text{ and } \models \bar{C} \\ a'_1 \sqcup a'_2 & \text{otherwise} \end{cases} \\
\qquad \text{where} \quad a'_1 = WFA(P,E,a_1) \\
\qquad \text{and} \quad a'_2 = WFA(P,E,a_2) \\
\qquad \text{and} \quad p \equiv l \text{ or } p \equiv \texttt{isNone}(l)
\end{array}$$

As described above, the judgment $P;E \vdash_{\text{lock}} l : \bar{C}$ checks that l is a well-formed lock expression in environment E.

[EXP REF] The rule for a field access *e.fn* first checks that e is of some type $cn\langle l_{1..n}\rangle$, and that cn is a class parameterized by n ghost variables, say $x_{1..n}$, that declares a field *fn* of some type t. The type t may refer to the variables `this` and $x_{1..n}$ in scope at the field declaration. Since these variables are not in scope at the field access, the type rule introduces a substitution θ that replaces them with the corresponding expressions e and $l_{1..n}$, and ensures that $\theta(t)$ is a well-formed type.

The [EXP REF] rule 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. If the field has `no_guard`, then the read operation is `atomic`, since it may not commute with concurrent writes. If the field is `guarded_by` l, then the lock $\theta(l)$ must be held and the read operation is a `mover`. If the field is `write_guarded_by` l, then if the lock $\theta(l)$ is held, there can be no concurrent writes, and so the read operation is again a `mover`. If the lock is not held, the operation is `atomic`.

[PROG] This rule defines the top-level judgment $P \vdash \bar{C}$, where $\bar{C}$ is the generated set of constraints for the program P. This rule uses a number of predicates defined informally as follows. (See [17] for their precise definition.)

- *ClassOnce*(P): no class is declared twice in P.
- *FieldsOnce*(P): no field name is declared twice in a class.
- *MethodsOnce*(P): no method name is declared twice in a class.

If the top-level judgment $P \vdash \bar{C}$ holds and a solution A for the constraints $\bar{C}$ exists, the following theorem states that the explicitly-typed program $A(P)$ is well-typed.

THEOREM 1 (CORRECT ASSIGNMENT). *If $P \vdash \bar{C}$ and $A \models \bar{C}$ then $A(P) \vdash A(\bar{C})$ and $\models A(\bar{C})$.*

Proof $A(P) \vdash A(\bar{C})$ follows by induction over the derivation of $P \vdash \bar{C}$. That $\models A(\bar{C})$ holds follows from $A \models \bar{C}$, given the definition of the relation $\cdot \models \cdot$. □

For the example program `List` of Figure 3, our system introduces the atomicity variables $\alpha_1, \ldots, \alpha_4$ for method atomicities, and then generates the atomicity constraints in Figure 7 for the program. The following section describes how to solve such a collection of constraints.

5.3 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, and it is applicable if there exists some constraint containing a variable on the right-hand side that is not satisfied by the current assignment. The relation

Figure 7: Constraints for the `List` Program

$$
\begin{array}{l}
\texttt{const;}\mathit{wfa}(P,E_1,\texttt{x?mover:error}) \hfill \sqsubseteq \alpha_1\\
\mathcal{S}(\texttt{this},\texttt{none},(\texttt{const;const;const;}\\
\qquad\qquad \mathit{wfa}(P,E_2,\texttt{this?mover:error});\\
\qquad\qquad \mathit{wfa}(P,E_2,\texttt{this?mover:error}))) \hfill \sqsubseteq \alpha_2\\
(\texttt{const;const;}\mathit{wfa}(P,E_3,\alpha_2\cdot[\texttt{this} := \texttt{this}]));\\
(\texttt{const;const;}\mathit{wfa}(P,E_3,\alpha_2\cdot[\texttt{this} := \texttt{this}])) \hfill \sqsubseteq \alpha_3\\
\mathcal{S}(\texttt{this},\texttt{none},(\texttt{const;}\mathit{wfa}(P,E_4,\texttt{this?mover:error});\\
\qquad\qquad \mathit{wfa}(P,E_4,\alpha_1\cdot[\texttt{this} := \texttt{this.elems},\texttt{x} := \texttt{this}]))) \sqsubseteq \alpha_4\\
\text{where}\\
\quad E_1 = \texttt{ghost x, ListElem}\langle\texttt{x}\rangle\ \texttt{this}\\
\quad E_2 = \texttt{List this, int v}\\
\quad E_3 = \texttt{List this, int i, int j}\\
\quad E_4 = \texttt{List this}
\end{array}
$$

produces a larger assignment A' in which that variable is increased so that the constraint is now satisfied:

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

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

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

Our constraint solving algorithm is an iterative least fix-point computation based on these two relations.

Figure 8: Constraint Solving Algorithm

$A := \bot$;
while $\exists A'$ such that $A \rightarrow^{\bar{C}} A'$ **do**
  $A := A'$;
if $A \rightarrow^{\bar{C}} \textsc{Err}$ **then return** "no solution";
else return A;

For the atomicity constraints of Figure 7, this constraint solving algorithm yields the minimal solution:

$$\alpha_1 = (\texttt{x?mover:error}) \qquad \alpha_2 = (\texttt{this?mover:atomic})$$
$$\alpha_3 = (\texttt{this?mover:cmpd}) \qquad \alpha_4 = (\texttt{this?mover:atomic})$$

5.4 Correctness of the Algorithm

We next show that the above algorithm is correct and always terminates. We first characterize the two relations $A \rightarrow^{\bar{C}} A'$ and $A \rightarrow^{\bar{C}} \textsc{Err}$ and show that if neither of these relations is applicable to an assignment A, then that assignment satisfies $\bar{C}$.

LEMMA 2 (STEP). *Suppose $A \rightarrow^{\bar{C}} A'$. Then $A \sqsubseteq A'$. If in addition there exists A'' such that $A \sqsubseteq A''$ and $A'' \models \bar{C}$, then we also have that $A' \sqsubseteq A''$.*

LEMMA 3 (CONTRADICTION). *Suppose $A \rightarrow^{\bar{C}} \textsc{Err}$. Then there is no A'' such that $A \sqsubseteq A''$ and $A'' \models \bar{C}$.*

LEMMA 4 (SOLUTION). *Suppose $A \not\rightarrow^{\bar{C}} \textsc{Err}$ and for all A', $A \not\rightarrow^{\bar{C}} A'$. Then $A \models \bar{C}$.*

It is straightforward to show, based on these lemmas, that the above algorithm only produces correct results. In particular, the resulting assignment, if one is found, is a solution to the constraints, and hence by Theorem 1, the explicitly-typed program $A(P)$ is well-typed. Similarly, if the constraints are unsatisfiable, then the original program is untypable.[4]

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 $|l| < MaxLockSize$. Similarly, an atomicity is *bounded* if it only contains bounded lock expressions, and an assignment is *bounded* if it only yields bounded atomicities.

LEMMA 5. *There are no infinite ascending chains of bounded atomicities or bounded assignments.*

An atomicity expression or constraint is *bounded* if it is only conditional on bounded lock expressions, and every delayed substitution occurs inside the construct $\mathit{wfa}(P,E,\cdot)$.

LEMMA 6. *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 $\mathit{wfa}(P,E,\cdot)$ will filter out these non-bounded lock expressions. □

LEMMA 7. *If A and d are bounded then $A(d)$ is bounded.*

THEOREM 8 (TERMINATION). *The constraint solving algorithm terminates on bounded constraint systems.*

Proof Since $\bar{C}$ is bounded, every generated assignment is also bounded. Since the generated assignments are increasing, the algorithm must terminate, as otherwise it would generate an infinite ascending chain of bounded assignments. □

6 Implementation

We have implemented our type inference algorithm, including extensions necessary to support the full Java programming language. This section describes some interesting details of our implementation, inheritance and other Java features, and how we handle a synchronization idiom found in a number of common library classes.

Our implementation, called *Bohr*, takes as input a Java source program. The source may optionally contain annotations in stylized comments starting with "`#`", as in "`/*# mover */`". *Bohr* runs in two phases. The first phase uses the *Rcc/Sat* tool to infer appropriate guards for each field, appropriate formal and actual ghost parameters for class and method declarations and uses, and protecting locks. For more details on *Rcc/Sat*, we refer the interested reader to our earlier paper [12].

The key novelty of our present work is the second phase of *Bohr*. This phase first adds an atomicity annotation α, where α is fresh, to each method without an explicit atomicity. It then checks the program according to our type inference algorithm. 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. The tool also simplifies any `isNone` constraints that are guaranteed to be false.

[4]This completeness argument covers the second phase of type inference. Our earlier paper [12] characterizes the completeness of *Rcc/Sat*.

6.1 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

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

yields the atomicity

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

containing many duplicate subterms. More generally, the sequential composition of n conditional atomicities yields an atomicity whose size is exponential in n. These large atomicities typically contain redundant information and can be simplified. For example, the above result can be simplified to:

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

The following rules define a relation $a \rightarrow^t_f a'$ that simplifies a to a' by removing redundant information, under the assumption that the lock predicates in t are true, and the lock predicates in f are false. We always apply the first applicable rule.

Figure 9: Atomicity Simplification Rules

$$\frac{}{b \rightarrow^t_f b} \qquad \frac{p \in t \qquad a_1 \rightarrow^t_f a}{p\,?\,a_1 : a_2 \rightarrow^t_f a} \qquad \frac{p \in f \qquad a_2 \rightarrow^t_f a}{p\,?\,a_1 : a_2 \rightarrow^t_f a}$$

$$\frac{a_1 \rightarrow^{t \cup \{p\}}_f a' \qquad a_2 \rightarrow^t_{f \cup \{p\}} a'}{p\,?\,a_1 : a_2 \rightarrow^t_f a'} \qquad \frac{a_1 \rightarrow^{t \cup \{p\}}_f a'_1 \qquad a_2 \rightarrow^t_{f \cup \{p\}} a'_2}{p\,?\,a_1 : a_2 \rightarrow^t_f p\,?\,a'_1 : a'_2}$$

One strategy for applying these rules is, after computing $a = [\![d]\!]$, to immediately simplify a via $a \rightarrow^{\{\mathtt{isNone(none)}\}}_{\emptyset} a'$. 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 wherever possible to avoid 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 [6] to efficiently represent and manipulate conditional atomicities.

6.2 Inheritance and Subtyping

The most significant extension to our type system for supporting the full Java language is dealing with 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() { ... } }
```

A type $\mathtt{D}\langle l \rangle$ is an immediate subtype of $\mathtt{C}\langle m \rangle$ if $m \equiv z[y := l]$. (The straightforward extension to multiple ghost arguments 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. For increased expressiveness, we permit the atomicity of a method to change covariantly, intuitively requiring only that: $s_2 \sqsubseteq \theta(s_1)$. Suppose that s_1 is an atomicity variable α and this constraint is not satisfied by the current assignment A. To determine how to increase $A(\alpha)$ to satisfy this constraint, 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* on atomicities:

$$\begin{array}{l}
\theta^{-1}(b) = b \\
\theta^{-1}(p\,?\,a_1 : a_2) = p_1\,?\,a'_1 : (p_2\,?\,a'_1 \cdots : (p_n\,?\,a'_1 : a'_2) \cdots) \\
\quad \text{where} \qquad\qquad a'_i \;=\; \theta^{-1}(a_i) \quad \text{for } i = 1,2 \\
\quad\quad \text{and} \quad \{p_1, \ldots, p_n\} \;=\; \{p' \mid \theta(p') = p\}
\end{array}$$

Each $p_1, \ldots, p_n$ maps to a'_1 to reflect that all of these predicates become p after applying θ.

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

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

and we express the above requirement $s_2 \sqsubseteq \theta(s_1)$ as the constraint:

$$\mathit{wfa}(P, E, \mathit{invsub}(\theta, s_2)) \sqsubseteq s_1$$

where the environment E of the class `C` ensures that the resulting atomicity for s_1 is well-formed in `C`.

In AJ2, a lock x of type $C\langle l_1, \ldots, l_n \rangle$ is considered to be protected by l_1. The situation in Java is somewhat different, since each class or interface is a subclass of `Object`. We declare `Object` to take a single ghost parameter, and thus every lock x must have the type $\mathtt{Object}\langle l \rangle$ for some l, and we use this lock l as the protecting lock for x. We allow an interface declaration to $\mathtt{extend\ Object}\langle l \rangle$, indicating that l is the protecting lock for objects of that interface. If an interface type $I\langle l_1, \ldots, l_n \rangle$ does not `extend` another interface or `Object`, it is assumed to be an immediate subtype of $\mathtt{Object}\langle l_1 \rangle$. For soundness, our type checker issues a type error if, via a combination of class and (possibly multiple) interface inheritance, a lock can be assigned two distinct types $\mathtt{Object}\langle l \rangle$ and $\mathtt{Object}\langle l' \rangle$.

6.3 Other Java Features

Inner classes, static members, thread-local data, escapes from the type system, and other Java-specific features are handled as they are in *Rcc/Sat* [12]. We summarize their treatment below.

Inner classes. Non-static inner classes may access the type parameters from the enclosing class and may declare their own parameters. Thus, the complete type for such a class is $\mathtt{Outer}\langle l_{1..n} \rangle\mathtt{.Inner}\langle m_{1..k} \rangle$.

Static fields, methods, and inner classes. Static members may not refer to the enclosing class' type parameters since static members are not associated with a specific instantiation of the class.

Thread objects. To allow `Thread` objects to store thread-local data in their fields, *Bohr* adds an implicit `final` field `tll` to each `Thread` class. This field is analogous to (and replaces) the `ghost` parameter on the `run` method in AJ2. It may guard other fields and is assumed to be held when `run` is invoked.

Escape mechanisms. We provide escapes from the AJ2 type system through a "`no_warn`" annotation that suppresses the generation of constraints for a line of code. Also, since ghost parameters are erased at run time, the ghost parameters in typecasts of the form $(\mathtt{C}\langle\mathtt{a}\rangle)\mathtt{x}$ are not checked dynamically.

6.4 Internal Synchronization

A common pattern in the Java collections library is the use of synchronized wrapper classes. This pattern is illustrated in Figure 10, where different `Counter` implementations use different synchronization disciplines for the method `inc`.

Figure 10: Synchronized Wrapper Class

```
interface Counter⟨ghost lock1⟩ extends Object⟨none⟩ {
  isAlways(lock1)?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⟩ {
  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();
}
```

The `UnsyncCounter` implementation requires clients to acquire a protecting lock `lock2` before calling the method `UnsyncCounter.inc`, which has atomicity `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. The `SyncCounter` class is a wrapper class that internally synchronizes the `inc` operation to avoid the need for external locking. No locks need to be held before calling the method `SyncCounter.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`". This lock is implicitly simultaneously held by all threads, but cannot guard fields. We assign the method `Counter.inc` the atomicity:

`isAlways(lock1)?atomic:(lock1?mover:error)`

The lock predicate `isAlways(lock1)` is true if `Counter` is parameterized by the special lock `always`; 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 10 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.

7 Evaluation

We have applied *Bohr* to a number of benchmarks, including both standard library classes and complete programs. Table 1 summarizes the results. 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 [12]). 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 numbers are quite promising.

Column 4 shows the number of subatomicity constraints generated for each benchmark. Column 5 shows the number of type annotations that 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 annotations. We added annotations only in situations where immediately identifiable local properties ensured correctness.

Columns 6 through 11 evaluate the precision of our type inference algorithm. Columns 6 and 9 show the number of methods and synchronized blocks in each benchmark, while columns 7 and 10 show the number (and percentage) that our type inference algorithm verified as being atomic. Columns 8 and 11 show the number (and percentage) that were not. We exclude the methods `main` and `run` because they are typically not atomic.

7.1 Standard Library Classes

The first group of benchmarks in Table 1 contains classes from the Sun Java 1.4.2 library and Doug Lea's concurrency package [25] that are intended to be atomic (i.e., all 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 supertypes.

Bohr was able to verify the atomicity of the vast majority of methods in these classes. For example, it verifies that 68 of the 69 methods in `java.lang.String` are atomic. (The method `String.hashCode` is not atomic, but this only results in redundant hash re-computations.) Our system verified that 47 out of 48 methods in `java.lang.StringBuffer` are atomic, and it detected one previously reported defect in `StringBuffer.append` [15].

We detected three errors in `Vector`. One of these errors is described in the excerpt in Figure 11 (this error was independently detected by Wang and Stoller [38]). The `Vector` constructor should copy the contents of its argument `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`. This would cause `elementCount` to be inconsistent with `elementData` and results in an improperly initialized `Vector`. *Bohr* detected similar defects in `Vector`'s and `SynchronizedList`'s `removeAll` and `retainAll`.

The warnings from `PrintWriter` are due to the potential for the writer's underlying stream to be shared among multiple threads, causing output from different threads to be improperly interleaved, as reported in [15]. The warnings for the remaining classes involve subtle synchronization patterns that are not verifiable with our current analysis, but which do not immediately appear to be incorrect.

Name	Size (LOC)	Time (s)	$\lvert\bar{C}\rvert$	Manual *Rcc/Sat* Annot.	Methods (excluding `run` and `main`) Total #	Atomic #	Atomic (%)	Non-Atomic #	Non-Atomic (%)	Synchronized Blocks Total #	Atomic #	Atomic (%)	Non-Atomic #	Non-Atomic (%)
`java.lang.String`	2,307	0.45	139	0	69	68	(99%)	1	(1%)	2	2	(100%)	0	(0%)
`java.util.StringBuffer`	1,276	0.53	91	2	48	47	(98%)	1	(2%)	33	32	(97%)	1	(3%)
`java.util.Vector`	3,546	0.87	197	16	50	47	(94%)	3	(6%)	36	34	(94%)	2	(6%)
`java.util.zip.Inflater`	319	0.14	44	0	18	18	(100%)	0	(0%)	12	12	(100%)	0	(0%)
`java.util.zip.Deflater`	384	0.15	48	1	20	20	(100%)	0	(0%)	12	12	(100%)	0	(0%)
`java.util.zip.ZipFile`	498	0.89	61	2	14	13	(93%)	1	(7%)	3	3	(100%)	0	(0%)
`java.util.Observable`	198	0.97	68	0	10	10	(100%)	0	(0%)	8	8	(100%)	0	(0%)
`java.util.SynchronizedList`	3,837	3.02	254	20	28	26	(93%)	2	(7%)	23	21	(91%)	2	(9%)
`java.net.URL`	1,201	0.74	64	11	33	30	(91%)	3	(9%)	5	4	(80%)	1	(20%)
`java.io.PrintWriter`	712	0.33	90	3	34	23	(68%)	11	(32%)	14	9	(64%)	5	(36%)
`concurrent.SynchronizedBoolean`	450	0.19	41	2	18	14	(78%)	4	(22%)	10	8	(80%)	2	(20%)
`concurrent.SynchronizedDouble`	444	0.18	41	2	18	14	(78%)	4	(22%)	11	9	(82%)	2	(18%)
`elevator` [37]	529	0.6	22	4	20	15	(75%)	5	(25%)	8	8	(100%)	0	(0%)
`tsp` [37]	723	1.43	21	5	19	10	(53%)	9	(47%)	6	6	(100%)	0	(0%)
`sor` [37]	687	0.83	24	1	19	19	(100%)	0	(0%)	4	4	(100%)	0	(0%)
`raytracer` [24]	1,982	1.73	132	5	117	110	(94%)	7	(6%)	15	14	(93%)	1	(7%)
`moldyn` [24]	1,408	4.89	78	3	68	61	(90%)	7	(10%)	14	14	(100%)	0	(0%)
`montecarlo` [24]	3,674	1.48	223	1	213	200	(94%)	13	(6%)	14	14	(100%)	0	(0%)
`mtrt` [33]	11,315	7.8	468	11	447	429	(96%)	18	(4%)	7	7	(100%)	0	(0%)
`jbb` [33]	30,519	11.2	1170	45	1073	876	(82%)	197	(18%)	241	190	(79%)	51	(21%)

Table 1. Performance and results of *Bohr* applied to benchmark programs and library classes.

Figure 11: Atomicities for `Vector` and `Collection`

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

  (isAlways(x)?atomic:(x?mover:error)
  Object[] toArray(Object a[]);
}
class Vector {
  Object elementData[] guarded_by this;
  int elementCount guarded_by this;

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

7.2 Complete Programs

The second benchmark group contains complete programs that were used to evaluate the Atomizer [11]. We expected that *Bohr* would issue significantly more warnings than the Atomizer, due to (1) the greater coverage and soundness 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*'s precision on large programs may be comparable to 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.

For `jbb`, *Bohr* reported significantly more warnings than 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*. The *synchronized blocks* heuristic revealed a previously-known defect in the computation of a checksum in `raytracer` [29, 11]. Overall, our experimental results show that the vast majority of methods in multithreaded applications are atomic.

8 Related Work

Since an atomicity annotation describes aspects of the behavior or effect of an expression, we are essentially performing a form of effect reconstruction [36, 35]. 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 [7] was among the first to explore type checking for dependent types. Our lightweight dependent types and effects are comparatively limited in expressive power, but the resulting type checking and type inference problems are decidable.

A number of tools, including other type systems [4, 20, 1], have been developed for detecting race conditions, both statically and dynamically. We refer the interested reader to our previous work [12] for an in depth discussion of race detection tools.

In independent work, Sasturkar, Agarwal, and Stoller [32] also present a type inference algorithm for atomicity. Their type system also extends [15] with parameterized classes [10]. Unlike our system, their system includes a notion of object ownership [3], but does not, for example, support protected locks. In contrast to our work, they use a dynamic analysis to infer race condition information and ghost parameters.

Lipton [26] first proposed reduction as a way to reason about deadlocks without considering all possible interleavings. Partial-order reduction techniques are based on similar ideas [19]. Bruening [5] and Stoller [34] have used Lipton's theory of reduction to improve the efficiency of model checking. Flanagan and Qadeer have pursued a similar approach [14], and Qadeer *et al* [31] have used reduction to infer procedure summaries in concurrent programs. We have explored adding abstraction based on purity to a type system for atomicity [13]. Introducing this notion into our type inference algorithm may reduce spurious warnings in some cases.

The use of model checking for verifying atomicity is being explored by Hatcliff *et al* [22]. This model checking approach is more expressive than our type-based analysis, but it is vulnerable to state-space explosion. Their results suggest that verifying atomicity via model-checking is feasible for unit-testing. Several tools have explored verifying atomicity dynamically [11, 38], but these tools are sensitive to test case coverage.

Atomicity is a semantic correctness condition for multithreaded software. It is related to strict serializability [30], a correctness condition for database transactions, and linearizability [23], a correctness condition for concurrent objects. It is possible that techniques for verifying atomicity can be leveraged to develop checking tools for related correctness conditions. Other languages, such as Argus [27] and Avalon [9], have included language support for implementing atomic objects. Recent approaches to supporting atomicity include lightweight transactions [21, 39] and automatic generation of synchronization code from high-level specifications [8].

9 Conclusions

Atomicity significantly facilitates the validation of multithreaded programs, since each atomic method can be considered to execute sequentially. However, verifying atomicity properties in large programs is non-trivial. Previous approaches were limited by test case coverage [11], were limited to systems with small states spaces [22], or required substantial assistance from the programmer [15, 18]. We believe our type inference algorithm provides a convenient and effective means to verify many atomicity properties in large programs. For example, it can verify that over 80% of the methods in our largest benchmark are atomic.

Acknowledgments. This work was supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387. We thank Scott Stoller for comments on a draft of this paper.

10 References

[1] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proc. Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[2] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2):79–111, 1999.

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

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

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

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, Aug. 1986.

[7] L. Cardelli. Typechecking dependent types and subtypes. In *Lecture notes in computer science on Foundations of logic and functional programming*, pages 45–57, 1988.

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

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

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

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

[12] C. Flanagan and S. N. Freund. Type inference against races. In *Static Analysis Symposium*, pages 116–132, 2004.

[13] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. In *Proc. International Symposium on Software Testing and Analysis*, pages 221–231, 2004.

[14] C. Flanagan and S. Qadeer. Transactions for software model checking. In *Proc. Workshop on Software Model Checking*, 2003.

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

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

[17] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. ACM Symposium on the Principles of Programming Languages*, pages 171–183, 1998.

[18] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Journal of Object Technology*, volume 3(6), pages 81–101, 2004.

[19] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science 1032. Springer-Verlag, 1996.

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

[21] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 388–402, 2003.

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

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

[24] Java Grande Forum. Java Grande benchmark suite. Available from `http://www.javagrande.org/`, 2003.

[25] D. Lea. `util.concurrent` package, release 1.3.4, 2004. Available at `http://gee.cs.oswego.edu/dl/`.

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

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

[28] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 47–57, 1988.

[29] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

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

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

[32] A. Sasturkar, R. Agarwal, and S. D. Stoller. Extended parameterized Atomic Java, 2004. Submitted for publication.

[33] Standard Performance Evaluation Corporation. SPEC Benchmarks. Available from `http://www.spec.org/`, 2004.

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

[35] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[36] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proc. ACM Symposium on the Principles of Programming Languages*, pages 188–201, 1994.

[37] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.

[38] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. Technical Report DAR 04-14, Computer Science Department, SUNY Stony Brook, July 2004. A preliminary version appeared in *Proc. Workshop on Runtime Verification*, 2003.

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