Automated Type-Based Analysis of Data Races and Atomicity *

Amit Sasturkar Rahul Agarwal Liqiang Wang Scott D. Stoller

ABSTRACT

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. This motivated the development of type systems that statically ensure the absence of some common kinds of concurrent programming errors including data races and atomicity violations. A method is atomic if every execution of the concurrent program is equivalent to an execution in which the atomic method is executed without being interleaved with other concurrently executed methods. Atomicity is a common correctness requirement in concurrent programs; atomicity violations may indicate incorrect synchronization. This paper presents Extended Parameterized Atomic Java (EPAJ), a type system for specifying and verifying atomicity in Java programs. EPAJ combines Flanagan and Qadeer's atomicity types [11] with a new and significantly more expressive type system for analyzing data races, called Extended Parameterized Race-Free Java (EPRFJ), allowing a more accurate analysis of atomicity. The paper also presents a type discovery algorithm to automatically obtain EPRFJ types, and a static interprocedural type inference algorithm that, given EPRFJ types, infers atomicity types. These algorithms can be incorporated into testing and debugging tools, benefiting users who know nothing about type systems. We report our experience with a prototype implementation.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software/Program Verification**]: Formal Methods, Reliability—*type systems*; D.2.5 [**Testing and Debugging**]: Debugging Aids; D.3.3 [**Language Constructs and Features**]: Concurrent Programming Structures

Keywords

Atomicity, Data Races, Type System, Type Inference

General Terms

Languages, Verification, Reliability

*Address: Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400. This work was supported in part by NSF under Grants CCR-9876058 and CCR-0205376 and by ONR under Grants N00014-02-1-0363 and N00014-04-1-0722. Email: {amits,ragarwal,liqiang,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{amits,ragarwal,liqiang,stoller}

PPoPP'05, June 15–17, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. A data race occurs when two threads simultaneously access a shared variable and at least one of the accesses is a write; data races often indicate the presence of synchronization errors. However, the absence of race conditions is neither sufficient nor necessary to show the absence of synchronization errors. Atomicity is a common higher-level correctness requirement that expresses non-interference between concurrently executed methods. A method is atomic if every execution of the concurrent program is equivalent to an execution in which the atomic method is executed without being interleaved with other concurrently executed methods. This is analogous to serializability (also called isolation) of transactions in databases.

Type systems are a promising approach for detecting data races and atomicity violations, and verifying their absence. Flanagan and Freund [7] and Boyapati and Rinard [4] developed type systems that ensure statically that a Java program is race-free, *i.e.*, contains no data races. The resulting programming languages (*i.e.*, Java with their extensions to the type system) are called *Race Free Java* (RFJ) and *Parameterized Race Free Java* (PRFJ), respectively. Flanagan and Freund applied their race-free type system [8] to about 49 KLOC, finding several bugs and showing that most of the code is race-free. Flanagan and Qadeer proposed a type system for specifying and verifying atomicity [11] by extending RFJ; the resulting extension of Java is called *Atomic Java.* Flanagan, Freund and Lifshin [9] applied their atomicity type system to around 49 KLOC of application code and 11 KLOC of Java library code, verifying that most of the methods are atomic and discovering several non-atomic methods.

This paper presents Extended Parameterized Atomic Java (EPAJ), a type system for atomicity that combines the atomicity types of Atomic Java with a new race-free type system, called Extended Parameterized Race-Free Java (EPRFJ).

EPRFJ extends PRFJ and is significantly more expressive than both PRFJ and RFJ. This allows a more accurate analysis of atomicity as well as race conditions.

Type inference [8, 9] and type discovery [3, 2] algorithms can greatly reduce or completely eliminate the need for users to manually annotate programs with extra type information. Type discovery is a combination of static and dynamic techniques; the main idea is to monitor executions of the program and generate candidate types based on the observed behavior and the results of static analysis.

This paper presents a two-step algorithm that infers the EPAJ types for a program. First, an extension of the type discovery algorithm in [3, 2] is used to obtain EPRFJ types. Second, a static interprocedural algorithm is used to infer the atomicity types. The latter algorithm does not assume that the input program is well-typed in EPRFJ, and it can determine the atomicity of methods in programs with races.

Our system can be useful even to users who have no knowledge of the type system, even for programs for which type discovery (or type inference) does not completely succeed. The resulting warnings from the type checker can be used to focus manual code inspections: users can inspect the lines of code that caused the warnings and check that correct synchronization is used. The warnings can also be used fully automatically to reduce the overhead of run-time data race detection tools, such as [16, 14], and run-time atomicity checking tools, such as [10, 17], by allowing them to avoid monitoring parts of the program for which type-checking succeeded. Section 9 describes our experience optimizing run-time atomicity checking. The warnings can also be used to guide scheduling heuristics in tools like ConTest [6], which perturb thread scheduling in ways intended to make synchronization errors manifest themselves.

In summary, the main contributions of our paper are (1) a significantly more expressive type system for analyzing data races, leading to fewer false alarms in the analysis of data races and atomicity; (2) one of the first type inference algorithms for atomicity types; (3) experiments demonstrating the effectiveness of our type system at verifying atomicity, catching subtle bugs and optimizing run-time atomicity checking of concurrent programs.

The rest of the paper is organized as follows. We give an overview of the EPAJ type system in Section 2 and describe the typing rules in Section 3. Sections 4 and 5 give the inference algorithms for race-free types and atomicity types. Section 6 briefly describes our implementation. We examine the expressiveness of EPAJ in Section 7, describe our experience in Section 8, and present our technique for optimizing run-time atomicity analysis in Section 9. Finally we survey related work in Section 10.

2. EXTENDED PARAMETERIZED ATOMIC JAVA

This section gives an overview of the EPAJ type system. We briefly review race-free types in PRFJ [4], describe our modifications to them and then describe atomicity types [11].

2.1 Overview of PRFJ

In PRFJ, types are extended to indicate the synchronization discipline (also called 'protection mechanism' or 'owner') used to co-ordinate accesses to objects. Each class

```
class Node<thisOwner> {
  int value;
  void setValue(int v) requires this {
    value = v;
  }
  void getValue() {
    return value;
  }
}

Node<thisThread> n1 = new Node<thisThread>();
Node<self> n2 = new Node<self>();
n1.setValue(10);
n2.setValue(20);
synchronized (n2) {
 n2.setValue(30);
}
```

Figure 1: An example program in PRFJ

is parameterized by owner parameters which may be instantiated with other formal owner parameters, final expressions (*i.e.* expressions whose value does not change) or special owners. This allows different instances of a class to use different protection mechanisms. The first owner parameter indicates the *owner* of the object, and the other owner parameters are used to propagate ownership information to the object's fields (an example appears in Section 2.2.1). There are four special owners : `thisThread`, `self`, `readonly` and `unique`. `readonly` indicates that the object is readonly and cannot be updated. `unique` means that there is a unique reference to the object. `thisThread` means that the object is thread-local (*i.e.*, unshared). `self` means that the object is protected by its own lock (*i.e.*, self-synchronized object). A final expression used as an owner specifies a lock that must be held when the object is accessed. The ownership relation forms a forest of trees. The root of each tree is referred to as the root owner of each object in the tree. Method declarations may have a `requires` clause that contains a set of final expressions; the locks on the root owners of these expressions must be held when the method is invoked. The locks on the special owners `thisThread`, `unique` and `readonly` are always assumed to be in the lockset. PRFJ ensures that whenever a field of an object is accessed, the lock on the root owner of the object is held, thus avoiding races. The root owner of an object is said to guard all of its fields.

The PRFJ program in Figure 1 illustrates how the PRFJ type system discovers potential race conditions. The program defines a `Node` class parameterized by a single owner `thisOwner`. The read access to the field `value` in the method `getValue` can lead to a data race because it can occur simultaneously with the write access to `value` in the method `setValue`. PRFJ detects this race condition by ensuring that whenever the field `value` is accessed, the lock on rootowner of `this` is held. Thus, the access to `value` in the method `setValue` typechecks because `setValue` has a `requires this` clause, whereas the access to `value` in the method `getValue` does not typecheck.

Since `n1` is parameterized by `thisThread`, it is a thread-local object; accesses to fields of `n1` cannot be involved in any data races. The method invocation `n1.setValue(10)` typechecks since the rootowner of `n1` is `thisThread` which

P	::=	$defn*\ \ local*\ \ e$
$defn$	::=	`class` $cn < formal+ >$ `extends` $c\ \ body$
c	::=	$cn < formal+ >$
$body$	::=	$\{\ field*\ \ meth*\ \}$
$field$	::=	`final`? $t\ \ fd\ \ [guardedby\ lock]? = e$
$meth$	::=	$t\ \ mn(arg*)$ `requires` $(lock*)$ `atomicity` $atom\ \ \{\ \ local*\ \ e\}$
arg	::=	$t\ \ x$
t	::=	$prim\_type\ \mid\ obj\_type$
obj_type	::=	$cn < owner+ >\ \mid\ Object < owner >$
$prim\_type$	::=	`int` | `bool` | `long`
$local$	::=	$t\ \ y\ \ =\ \ e$
$owner$	::=	$formal$ |`self` | `thisThread` | e_{final} | $owner\$e_{final}$
$guardedby$	::=	`guarded_by` | `write_guarded_by`
$lock$	::=	e_{final} | $formal\$e_{final}$
$atom$	::=	`const` | `mover` | `atomic` | `cmpd` | `error` | $owner\ ?\ atom\ :\ atom$
e	::=	`new` $obj\_type\ \mid\ e\,;e\ \mid\ x$ $\mid\ x = e\ \mid\ e.fd\ \mid\ e.fd = e$ $\mid\ e.mn(e*)\ \mid$ `synchronized` $(e)\ \{e\}$ $\mid\ e$.`fork` $\mid$ `if` $e\ e\ e\ \mid$ `while` $e\ e$
cn	$\in$	`classnames`
fd	$\in$	`fieldnames`
mn	$\in$	`methodnames`
x, y	$\in$	`variablenames`
$formal$	$\in$	`formalownernames`

Figure 2: Grammar for mini EPAJ. Asterisk (*), plus (+), and question mark (?) indicate any number, 1 or more, and 0 or 1 occurrences, respectively. Square brackets are used for grouping. e_{final} ranges over final expressions, as defined in the text.

is always assumed to be held. Since `n2` is parameterized by `self`, it is a self-synchronized object that can be shared by multiple threads. The method invocation `n2.setValue(20)`, which writes to the field `value`, can be involved in a data race since no lock is held when this method is invoked. PRFJ detects this race condition by checking that the `requires` clause on `setValue` is satisfied at all its invocation sites. Since the rootowner of `n2` is `n2` itself and the lock on `n2` is not held when `n2.setValue(20)` is called, this invocation of `setValue` does not typecheck. But, `n2.setValue(30)` typechecks since it is enclosed within a `synchronized(n2) {..}` block. Thus, by guarding all fields of an object by the rootowner of the object, and specifying requires clauses on methods, PRFJ can catch possible data races and verify their absence.

2.2 Race-Free Types in EPAJ

We present our type system in the context of a multi-threaded subset of Java, which is based on Concurrent Java [7]. We call the resulting language mini EPAJ. The grammar is given in Figure 2. Due to space constraints, and since support for `readonly` and `unique` is orthogonal to atomicity types, this grammar and the typing rules in Section 3 omit support for `readonly` and `unique`. The complete grammar and typing rules appear in [13]. Our implementation supports all four special owners.

In the grammar in Figure 2, e_{final} ranges over final expressions which are built from final variables including `this`, final fields, and fields that are assigned only once in the enclosing class's constructor and are not accessed in the constructor before that assignment. In mini EPAJ, the only final variable is `this` (assignments to `this` are prohibited); in EPAJ, any local variable (including parameters) may be annotated as `final`. The expression e.`fork` evaluates e to an object `o`, creates a new thread, invokes `o.run` in the new thread, and returns 0. We use Java-like syntactic sugar in examples. For example, a method declaration `synchronized` $t\ mn(arg^*)\ e$ abbreviates $t\ mn(arg^*)$ `synchronized (this)` $\{e\}$.

In EPAJ, as in PRFJ, a class is parameterized by owner parameters. The main difference in expressiveness between EPAJ and PRFJ is that in EPAJ each field may have a different guard. A guard may be (1) empty or (2) a `guarded_by` clause or (3) a `write_guarded_by` clause. Since each field has a different guard, the concept of root owner of an object is no longer well-defined and hence EPAJ completely dispenses with it. Another difference is that in EPAJ, the `requires` clause may contain formal owners; this helps compensate for removing root ownership.

If a bare formal owner parameter appeared in a `requires` clause, its meaning when instantiated with `self` might be ambiguous. For example, consider the class Node for nodes in a linked list.

```
class Node<f1,f2> {

   Node<f1,f2> next guarded_by f1;
   int value guarded_by f2;

   setValue() requires (f2) {
    this.value = ...
   }
}
```

In PRFJ, both fields would be forced to have the same owner (and hence the same root owner), which is the owner of the entire object, and `setValue` would have a "`requires (this)`" clause, meaning that the lock on the root owner of `this` must be held when the method is called. In EPRFJ, it is natural to use `requires (f2)`, meaning that the lock on `f2` (if `f2` refers to an object, not a special owner) should be held when the method is called. However, when `f2` is instantiated with `self`, it is unclear which object is being referred to.

The problem is strikingly evident in the following example, which involves a *method owner parameter*, *i.e.*, a formal owner parameter that appears in a method declaration and not in the declaration of the enclosing class.

```
class Node<f1> {
   swap(Node<g1> a1, Node<g1> a2) requires (g1) {
      ....
  }
}
```

In this example, when `g1` is instantiated with `self`, it is unclear whether the lock on `a1` or `a2` should be held when the method is called. We eliminate such ambiguity by qualifying each formal owner parameter f with a final expression e

that indicates the object that f refers to when instantiated with `self`. The notation for this is $f\$e$. In practice, formal owner parameters almost always get qualified with the final expression `this`; by making this the default when a qualifier is omitted in a `requires`, `guarded_by` or `write_guarded_by` clause, few $\$e$ annotations are needed in most programs.

Formally, the meaning of $f\$e$ is expressed by the effect of substitutions (applied by the typing rules) on it. Let σ be a substitution that maps variables to expressions. The result of applying a substitution σ to $f\$e$ is

$$f\$e[\sigma] = \begin{cases} e[\sigma] & \text{if } f[\sigma] = \texttt{self} \\ f[\sigma]\$e[\sigma] & \text{if } f[\sigma] \text{ is a formal owner parameter} \\ f[\sigma] & \text{otherwise} \end{cases}$$

Here, $f[\sigma]$ denotes the effect of applying substitution σ to f. Note that applying the empty substitution [ ] may simplify an expression. For example, `thisThread`$\$e[\ ] =$ `thisThread`. Every substitution σ can be composed with the empty substitution without changing it. Thus, the result of applying a substitution never has the form $g\$e$ where g is a special owner.

A `guarded_by` clause contains a well-formed *lock expression*, which is either a final expression or $f\$e$ where e is a final expression and f is the first formal owner parameter in the type of e. A `guarded_by` clause cannot contain a special owner explicitly, but the formal owner parameter in it may be instantiated with a special owner, providing the same effect.

If a field `fd` of an object o is guarded by `thisThread`, it is accessed by only one thread and hence cannot be involved in a data race. Thus, no lock is required to access `fd`. If a field is guarded by a final expression e, then the lock on e must be held when `fd` is accessed. If a field is guarded by $f\$e$, it means that when f is instantiated with `self`, the field is guarded by e, otherwise it is guarded by whatever f is instantiated with.

A field can have a `write_guarded_by` clause instead of a `guarded_by` clause. A `write_guarded_by` clause has the same meaning as a `guarded_by` clause, except that the protection mechanism specified by a `write_guarded_by` clause is required only at write accesses to the field.

The `requires` clause on a method declaration contains lock expressions, and indicates the locks that must be held when the method is called.

2.2.1 Example

The following program from [4] is a typical example that demonstrates the usefulness of the root owner concept of PRFJ. We show that EPAJ, though lacking root owner, is expressive enough to prove that the program is race-free. Ignore the `atomicity` clauses for now; atomicity types are described below.

Figures 3 and 4 give EPAJ code for an implementation of a stack of type `T` objects using a linked list. The `TStack` class is parameterized by two formal owner parameters : the first one guards `TStack.head`, and the second one is used to instantiate a formal owner parameter in the type of `TStack.head` therby propagating ownership information to the guards of the fields of the elements in the stack. The substitutions described below reflect the instantiations of parameterized types as they are used in this program; the typing rules in Section 3 perform such substitutions.

The `head` field of `TStack` has different protection mechanisms in the stacks `s1`, `s2` and `s3`. `s1` and `s2` are thread-local and hence `s1.head` and `s2.head` can be accessed without any synchronization. The `requires` clauses of `push` and `pop` methods evaluate to `thisThread` under the substitution σ_1 = [`thisOwner`/`thisThread`] for stacks `s1` and `s2` and hence no lock needs to be held at their call sites. On the other hand, stack `s3` is self-synchronized, and `s3.head` is guarded by `thisOwner$this` which evaluates to `s3` under the substitution σ_3 = [`self`/`thisOwner`][`s3`/`this`]. The methods `push` and `pop` require `thisOwner$this`, which evaluates to `s3` under σ_3. Thus, the accesses to `s3.head` in these methods type-check and the lock on `s3` needs to be held at the call sites of `s3.push` and `s3.pop`.

The `value` and `next` fields of `TNode` have the guard `thisOwner$this`. For the stacks `s1` and `s2`, this guard evaluates to `thisThread` and hence the fields can be accessed without synchronization. The `requires` clause of the methods `init, value` and `next` contains `thisOwner$this` which evaluates to `thisThread` for `s1` and `s2`. Thus, no lock needs to be held when these methods are called in `push` and `pop`. In the case of `s3`, the guard of the above fields and the lock expression in the `requires` clause of the above methods both evaluate to `s3` (since `thisOwner$this` under the substitution [`thisOwner$this/thisOwner`] evaluates to `thisOwner$this` which under substitution σ_3 evaluates to `s3`). Thus, the lock on `s3` must be held when the methods `s3.init, s3.value` and `s3.next` are called, and we have already seen that the `requires` clause on `s3.push` and `s3.pop` ensures that this is indeed the case.

This example illustrates that programs whose PRFJ typings are based on root ownership are also typable in EPAJ. Informally, instead of implicitly defining a root owner, we explicitly specify the root owner in the type of the object and pass it through the type parameters, transforming it by applying substitutions (for *e.g.*, while evaluating the locks in a `requires` clause or a `guarded_by` clause) as needed.

2.3 Atomicity Types

The atomicity types proposed in [11] are adopted unchanged in EPAJ. The type system associates an atomicity with each expression. The atomicity of each method is declared in the program; atomicities of other expressions are implicit. An atomicity is a *basic atomicity* or a *conditional atomicity*. The basic atomicities and their meanings are: `const`: evaluation of the expression does not depend on or change any mutable state; `mover`: the expression left-commutes with every operation of another thread that could occur immediately before it and right-commutes with every operation of another thread that could occur immediately after it; `atomic`: evaluation of the expression is always equivalent to evaluation of the expression without interleaved actions of other threads; `cmpd` (compound): none of the preceding atomicities apply; `error`: evaluation of the expression violates the locking discipline specified by the `guarded_by` or the `write_guarded_by` clauses.

Conditional atomicities are used when the atomicity of an expression depends on the locks held by the thread evaluating it. A conditional atomicity $l\,?\,a : b$ is equivalent to atomicity a if lock l is held when the expression is evaluated, and is equivalent to atomicity b otherwise. $l\,?\,a$ abbreviates $l\,?\,a$: `error`.

```
class TStack<thisOwner,TOwner> {
  TNode<thisOwner$this,TOwner> head
     guarded_by thisOwner = null;

  void push(T<TOwner> value) requires (thisOwner)
  atomicity thisOwner ? mover {
    TNode<thisOwner$this,TOwner> newNode =
       new TNode<thisOwner$this,TOwner>();
    newNode.init(value,head);
    this.head = newNode;
  }
  T<TOwner> pop() requires (thisOwner)
  atomicity thisOwner ? mover {
    if (this.head == null) return null;
    T<TOwner> value = this.head.value();
    this.head = this.head.next();
    return value;
  }
}

class TNode<thisOwner,TOwner> {
  T<TOwner> value guarded_by thisOwner;
  TNode<thisOwner,TOwner> next guarded_by thisOwner;

  void init(T<TOwner> v, TNode<thisOwner,TOwner> n)
  requires (thisOwner)
  atomicity thisOwner ? mover {
    this.value = v;
    this.next = n;
  }

  T<TOwner> value() requires (thisOwner)
  atomicity thisOwner ? mover {
    return this.value;
  }

  TNode<thisOwner,TOwner> next()
  requires (thisOwner)
  atomicity thisOwner ? mover {
    return this.next;
  }
}

class T<thisOwner> {
  int x guarded_by thisOwner = 0;
}
```

Figure 3: EPAJ code for the TStack example (part 1).

Let α and a, b range over basic atomicities and atomicities, respectively. Each atomicity a is interpreted as a function $[\![a]\!]$ from the set ls of locks currently held to a basic atomicity: $[\![\alpha]\!](ls) = \alpha$ and $[\![l\,?\,a_1 : a_2]\!](ls) =$ if $l \in ls$ then $[\![a_1]\!](ls)$ else $[\![a_2]\!](ls)$. Following [11], we define a partial order $\sqsubseteq$ on atomicities. The ordering on basic atomicities is `const` $\sqsubseteq$ `mover` $\sqsubseteq$ `atomic` $\sqsubseteq$ `cmpd` $\sqsubseteq$ `error`. The ordering on conditional atomicities is the pointwise extension of the ordering on basic atomicites, *i.e.*, $a \sqsubseteq b$ iff $\forall ls : [\![a]\!](ls) \sqsubseteq [\![b]\!](ls)$. Rules for effectively determining the ordering on atomicities appear in [11].

```
class Worker<thisOwner> {
  final TStack<self,self> s3;

  Worker(TStack<self,self> s) { s3 = s; }

  int run() {
    T<self> t = new T<self>();
    synchronized (s3) { s3.push(t) }
  }
}

T<thisThread> t1 = new T<thisThread>();
T<self> t2 = new T<self>();

TStack<thisThread,thisThread> s1 =
 new TStack<thisThread,thisThread>();
TStack<thisThread,self> s2 =
 new TStack<thisThread,self>();
TStack<self,self> s3 =
 new TStack<self,self>();

s1.push(t1); s2.push(t2);
(new Worker<self>(s3)).fork;
(new Worker<self>(s3)).fork;
```

Figure 4: EPAJ code for the TStack example (part 2).

The typing rules express the atomicity of an expression in terms of the atomicities of its subexpressions using five operations on atomicities: sequential composition $a; b$, iterative closure a^*, join $a \sqcup b$ (based on the partial order $\sqsubseteq$ described above), conditional $l\,?\,a : b$, and the operation $S(l, a)$ described below. Sequential composition for basic atomicities is defined by: $\alpha_1; \alpha_2$ equals `cmpd` if α_1 and α_2 are both `atomic`, and equals $\alpha_1 \sqcup \alpha_2$ otherwise. The iterative closure a^* denotes the atomicity of an expression that repeatedly executes an expression with atomicity a. For basic atomicities, it is defined by: a^* equals `cmpd` if a is `atomic`, and equals a otherwise. Sequential composition and iterative closure for conditional atomicities are defined as follows [11].

$$(l\,?\,a : b)^* = l\,?\,a^* : b^*$$
$$(l\,?\,a_1 : a_2); b = l\,?(a_1; b) : (a_2; b)$$
$$\alpha; (l\,?\,b_1 : b_2) = l\,?(\alpha; b_1) : (\alpha; b_2)$$

The operation $S(l, a)$ defined in Figure 5 is used to simplify the atomicity a of an expression based on the fact that lock l is known to be held during evaluation of the expression [11]. S is used in the typing rule for synchronized expressions to simplify conditional atomicities in the atomicity of the body of the synchronized expression.

3. TYPING RULES

This section presents representative typing rules for EPAJ. Typing rules for EPRFJ can be obtained from typing rules for EPAJ by erasing the parts that mention atomicities, requiring that every field has a `guarded_by` clause (*i.e.*, the guarded-by clause may not be absent or `write_guarded_by`)

$$
\begin{array}{lcl}
S(l, \texttt{const}) & = & l\,?\,\texttt{const} : \texttt{atomic} \\
S(l, \texttt{mover}) & = & l\,?\,\texttt{mover} : \texttt{atomic} \\
S(l, \texttt{atomic}) & = & \texttt{atomic} \\
S(l, \texttt{cmpd}) & = & \texttt{cmpd} \\
S(l, \texttt{error}) & = & \texttt{error} \\
S(l, l\,?\,a : b) & = & S(l, a) \\
S(l, l_1\,?\,a : b) & = & l_1\,?\,S(l, a) : S(l, b) \quad \text{if } l \neq l_1
\end{array}
$$

Figure 5: Definition of S.

and deleting the rules [EXP REF RACE] and [EXP ASSIGN RACE]. This ensures that programs typable in EPRFJ are race-free.

The core of the type system is a set of rules for reasoning about type judgments of the form $P; E; ls \vdash e : t \;\&\; a$. Here P is the program being checked, E is the environment that contains types of local variables and formal owner parameters, e is the expression being checked, t is the type of e, a is the atomicity of e and ls is the set of locks that are known to be held during the evaluation of e. Several auxiliary judgments are also needed. Table 1 summarizes the judgments used in this paper.

The rest of this section presents some representative typing rules. A complete presentation of our version of PRFJ which includes support for `readonly` and `unique` appears in [2]. The typing rules for EPAJ extend the rules in [2] and add judgments for inferring atomicity types. Due to space constraints, and since support for `unique` and `readonly` objects is mostly orthogonal to atomicity types, the rules presented in this section are stripped down versions of the actual EPAJ rules with support for `unique` and `readonly` eliminated. The complete typing rules for EPAJ (with support for `unique` and `readonly`) appear in [13]. We have proved soundness of the core of our type system by extending the soundness proof in [1]. The proof can be found at http://www.cs.sunysb.edu/~amits/papers/atomicity-inference/.

3.1 Well-formed method

Figure 6 contains the [METHOD] rule for typechecking methods. We allow the `this` parameter to be declared explicitly, to allow its owner parameters to be instantiated. For uniformity, we require that `this` be declared in each method. As syntactic sugar, if the declaration of `this` is absent in a method in class $cn\langle f_{1\ldots n}\rangle$, the declaration "$cn\langle f_{1\ldots n}\rangle$ `this`" is inserted. The rule adds ordinary parameters, method owner parameters and local variables to the typing environment. It checks that the declared atomicity of the method and the locks in the `requires` clause are well-formed. It typechecks the method body assuming the locks in the `requires` clause are held and verifies that the type and the atomicity of the method body match the return type and the declared atomicity of the method.

3.2 Well-formed expressions

The rule [EXP SYNC] for `synchronized`(e_1) { e_2 } adds the acquired lock e_1 to the lockset and checks e_2 with this enlarged lockset. It uses the function S defined in Section 2.3 to simplify the atomicity a of the body e_2 based on the fact that e_1 is held during evaluation of the body.

[METHOD]

$$
\frac{\begin{array}{l}
P \vdash\ t\ mn(arg_{0..n})\ \texttt{requires}\ (e_{1..m}) \\
\quad \texttt{atomicity}\ a\ \{local_{1..l}\ e\}\ \in\ cn\langle f_{1..k}\rangle \\
\text{each formal owner in } t \text{ appears in some } arg_i \\
arg_0 \text{ matches } cn\langle\ldots\rangle\ldots\ \texttt{this} \\
E' = E,\ \texttt{final}\ arg_0, \ldots, \texttt{final}\ arg_n \\
E'' = E' \cup \{\ \text{owner}_{formal}\ f \mid f \text{ appears in some } arg_i\ \} \\
\forall i = 1 \ldots m : P; E'' \vdash_{\text{lock}}\ e_i \\
P; E'' \vdash_{\text{atom}}\ a \\
P; E'', local_{1..l}; \texttt{thisThread}, e_{1..m} \vdash\ e\ :\ t\ \&\ a
\end{array}}{\begin{array}{l}
P; E \vdash\ t\ mn(arg_{0..n})\ \texttt{requires}\ (e_{1..m}) \\
\quad \texttt{atomicity}\ a\ \{local_{1..l}\ e\}
\end{array}}
$$

Figure 6: Typing rule for methods.

[EXP SYNC]

$$
\frac{\begin{array}{l}
P; E; ls \vdash_{\text{final}} e_1 : cn < o_1, o_2, ..., o_n > \ \&\ \text{const} \\
P; E; ls, e_1 \vdash e_2 : t_2\ \&\ a
\end{array}}{P; E; ls \vdash \texttt{synchronized}(e_1)\{e_2\} : t_2\ \&\ S(e_1, a)}
$$

The lock l in a conditional atomicity $l\,?\,a : b$ may be a *lock expression* (defined in 2.2). Thus, l may be instantiated with the special owner `thisThread`. Since `thisThread` is always assumed to be in the lockset, $l\,?\,a : b$ can be simplified to a in this case. [1] The simplify function does this.

$$
\begin{array}{l}
\text{simplify}(\alpha) \qquad = \alpha \\
\text{simplify}(l\,?\,a : b) = \begin{cases} \text{simplify}(a) & \text{if l} = \texttt{thisThread} \\ l\,?\,\text{simplify}(a) : \text{simplify}(b) & \text{otherwise} \end{cases}
\end{array}
$$

The rule [EXP REF GUARD] in Figure 7 checks that a read $e.fd$ of a guarded field fd is well-typed. Suppose e has atomicity a. If field fd is a final field, the read is allowed, and the atomicity of $e.fd$ is a; `const`. If $e.fd$ has a `guarded_by` l clause, the rule checks that l is in the current lockset, and the atomicity of $e.fd$ is a; simplify($l\,?\,$`mover`) (since race-free accesses are movers and the access is an error if l is not held). If $e.fd$ has a `write_guarded_by` l clause, atomicity of $e.fd$ is a; simplify($l\,?\,$`mover` : $A(t)$); this says that if l is held, the read has atomicity `mover`, otherwise the atomicity is $A(t)$ where $A(t)$ is `cmpd` if t is `long` (since long integers occupy 2-words and are presumably read in separate instructions), and is `atomic` otherwise.

The rule [EXP REF RACE] is used for read accesses $e.fd$ to unguarded fields. [EXP REF RACE] is similar to [EXP REF GUARD], except that hypotheses related to the guard are omitted, and the atomicity in the conclusion is $a; A(t)$.

In the rule for method invocation in Figure 8, the index j ranges over $0 \ldots k$; note that y_0 is `this`. formalOwners($t_{0\ldots k}$) denotes the set of formal owner parameters that appear in the types $t_{0\ldots k}$. The rule uses a substitution σ to instantiate formal owner parameters; this ensures that a formal owner parameter that occurs multiple times is instantiated consistently. The hypothesis containing $P; E \vdash_{\text{owner}} f[\sigma]$ ensures that σ replaces f with a valid owner. The hypothesis $e_i'' \in ls$ where $i \in \{\ 1, 2, \ldots, m\ \}$ checks that all the locks in the `requires` clause of the method declaration are held at the call site. The atomicity of the method call is the sequen-

[1] `readonly` and `unique` are also allowed in the full type system, and are simplified similarly.

Judgment	Meaning
$P; E; ls \vdash e : t \ \& \ a$	expression e has type t and atomicity a provided locks in ls are held
$P \vdash meth \in cn\langle f_{1\ldots n}\rangle$	class cn with owner parameters $f_{1\ldots n}$ declares or inherits method $meth$
$P \vdash field \in cn\langle f_{1\ldots n}\rangle$	class cn with owner parameters $f_{1\ldots n}$ declares or inherits field $field$
$P; E; ls \vdash_{\text{final}} e : t \ \& \ a$	e is a final expression with type t and atomicity a
$P; E \vdash_{\text{owner}} o$	o is a well-formed owner
$P; E \vdash_{\text{lock}} e$	e is a well-formed lock
$P; E \vdash_{\text{atom}} a$	a is a well-formed atomicity
$P \vdash_{\text{final}} cn.fd$	$cn.fd$ is a final field declared or inherited in class cn

Table 1: Type judgments.

[EXP REF GUARD]

$$\frac{\begin{array}{l} P; E; ls \vdash e : cn\langle o_{1\ldots n}\rangle \ \& \ a \\ P \vdash \ [\texttt{final}]?\ t\ fd\ guardedby\ l\ \in\ cn\langle f_{1\ldots n}\rangle \\ t' \ = \ t[e/\texttt{this}][o_1/f_1][o_2/f_2]\ldots[o_n/f_n] \\ l' \ = \ l[e/\texttt{this}][o_1/f_1][o_2/f_2]\ldots[o_n/f_n] \\ P \vdash_{\text{final}} cn.fd \Rightarrow b \ = \ \texttt{const} \\ P \nvdash_{\text{final}} cn.fd \ \wedge \ (guardedby = \texttt{guarded\_by}) \Rightarrow \\ \quad l' \in ls \wedge b = l'\,?\,\texttt{mover} \\ P \nvdash_{\text{final}} cn.fd \ \wedge \ (guardedby = \texttt{write\_guarded\_by}) \Rightarrow \\ \quad b = l'\,?\,\texttt{mover} : A(t) \end{array}}{P; E; ls \vdash \ e.fd : t' \ \& \ a; \text{simplify}(b)}$$

Figure 7: Typing rule for reading from guarded fields.

[EXP INVOKE]

$$\frac{\begin{array}{l} P; E; ls \vdash e_j : t'_j \ \& \ a_j \\ P \vdash \ (t\ mn(t_j\ mod_j\ y_j{}^{j\in 0\ldots k})\ \texttt{requires}(e'_{1\ldots m}) \\ \quad \texttt{atomicity}\ b) \in cn\langle f_{1\ldots n}\rangle \\ \text{dom}(\sigma) \ = \ \text{formalOwners}(t_{0\ldots k}) \\ \forall f \in \text{dom}(\sigma) : \ P; E \vdash_{\text{owner}} \ f[\sigma] \\ t'_j \ = \ t_j[\sigma][e_0/\texttt{this}] \\ e''_i \ = \ e'_i[\sigma][e_0/\texttt{this}][e_1/y_1]\ldots[e_k/y_k] \\ e''_i \ \in \ ls \\ b' \ = \ \text{simplify}(b[\sigma][e_0/\texttt{this}][e_1/y_1]\ldots[e_k/y_k]) \end{array}}{P; E; ls \vdash \ e_0.mn(e_{1\ldots k}) \ : \ t[\sigma][e_0/\texttt{this}] \ \& \ a_0; a_1; \cdots; a_k; b'}$$

Figure 8: Typing rule for method invocation.

tial composition of the atomicity of each of the arguments and the declared atomicity of the method.

The [EXP FORK] rule in Figure 9, when checking e.`fork`, checks the invocation `this.run()` with an environment E' that declares `this` to have the same type as e (except for the effect of the substitution described below) and with a lockset that contains only `thisThread`.

To ensure that fields protected by `thisThread` in objects reachable from e do not become shared as a result of the `fork`, we substitute `otherThread` for `thisThread` in the type of e when constructing E', as in [4]. Thus, when type-checking the invocation of `this.run()` in the third hypothesis, the guarded-by clauses of fields that are local to the old thread (*i.e.*, fields that were guarded by `thisThread`) now have owner `otherThread`, causing a type error if the new thread tries to access them, because `otherThread` is never in the lockset. Since guarded-by clauses cannot contain special owners explicitly, applying this substitution to the type of e ensures that the effect propagates to accesses of fields of objects reachable from e.

[EXP FORK]

$$\frac{\begin{array}{l} P; E; ls \vdash \ e \ : \ cn\langle o_{1\ldots n}\rangle \ \& \ a \\ E' = \texttt{final}\ cn\langle o_{1\ldots n}\rangle[\texttt{otherThread}/\texttt{thisThread}]\ \texttt{this} \\ P; E'; \texttt{thisThread} \vdash \ \texttt{this.run}() : \texttt{int} \ \& \ \texttt{cmpd} \end{array}}{P; E; ls \vdash \ e.\texttt{fork} : \texttt{int} \ \& \ a \sqcup \texttt{atomic}}$$

Figure 9: Typing rule for `fork`.

4. TYPE DISCOVERY FOR EPRFJ

This section describes an extension to the type discovery algorithm for PRFJ in [3, 2] to automatically discover EPRFJ types. Section 5 describes an algorithm that uses these annotations to infer atomicity types.

The type discovery algorithm has three main steps. First, the target program is instrumented by an automatic source-to-source transformation and executed on test inputs. The instrumented program monitors accesses to fields of certain objects and writes relevant information (mainly information about which locks were held when the field was accessed) to a log file. Second, the information in the log file is used to statically infer owners for fields, method parameters and return values, and owners in class declarations. Third, the intra-procedural type inference algorithm in [4], is used to infer the owners in the types of local variables and allocation sites whose types have not already been determined. Local type-inference has the crucial effect of propagating type information into branches of the program that were not exercised in the monitored executions.

Type discovery is not guaranteed to produce correct typings for all typable programs, but experience shows that it is very effective in practice. In our experiments in [2], 98% of the race-free types were automatically discovered. As described in Section 1, these partial typings can be useful even to users who never look at or fix them.

The main change to type discovery needed to handle EPRFJ is to maintain information on a per-field basis instead of a per-object basis. For example, during run-time monitoring, the type discovery algorithm for PRFJ keeps track of whether each object is shared (*i.e.*, has been accessed by multiple threads), while the type discovery algorithm for EPRFJ keeps track of whether each field of each object is shared. The elimination of root owner has no direct impact here, because our type discovery algorithm for PRFJ does not exploit the distinction between owner and root owner. Another minor change is extending the algorithm to track information for determining write-guarded-by relationships (in addition to guarded-by relationships).

Type discovery for EPRFJ works roughly as follows. (*i*) If, for all monitored instances of a class, a field `fd` of a class

C is guarded by the same special owner or final expression o, then `fd` gets annotated with `guarded_by` o (*ii*) Otherwise, `fd` gets annotated with `guarded_by` f`$this` where f is the first formal owner parameter of class C. For each use of class C in a field type, method parameter type, or method return type, f gets instantiated based on how fields of C that are `guarded_by` f are accessed for the instances of C stored in that field, method parameter, or return value. `write_guarded_by` is handled similarly.

The complexity of the type discovery algorithm is discussed below. Let $|P|$ denote the size of the original program and let $|T|$ denote the size of the execution traces for the test inputs. The complexity of the first step of the algorithm is measured in terms of the overhead incurred while executing the instrumented program. On an average, for the benchmarks in our experiments, the first step incurs an overhead of 20%. In the second step, the execution trace is scanned once to discover types, and this has a time complexity of $O(|T|)$. The third step (local type-inference) uses a standard UNION-FIND data structure to propagate the types inferred in the second step. This phase runs in time almost linear ($O(|P| \log^*(|P|))$ in the size of the program. Thus, the static phases of the type-discovery algorithm have an overall time complexity of $O(|T| + |P| \log^*(|P|))$.

In our experience, the result of the type discovery algorithm is mainly affected by which methods are exercised by the test inputs, and is otherwise mostly insensitive to the choice of inputs. In our experiments, we did not carefully choose the inputs to obtain correct types. We just used the test inputs provided with the benchmarks, and these were sufficient to discover accurate types.

5. INFERENCE OF ATOMICITY TYPES

This section describes our algorithm for inferring the atomicity of each method. It assumes that the program is annotated with race-free types (`requires` clauses in the race-free typing are ignored; required locks are computed as part of conditional atomicities). Soundness of this algorithm does not depend on correctness of the race-free typing. In other words, the algorithm does not assume that the input program is well-typed in EPRFJ, and it can determine atomicity of methods in programs with races. However, incorrect race-free typings generally lead to weak conclusions about atomicity. For example, an incorrect `guarded_by` clause on a field may cause the algorithm to infer that a method that uses the field has atomicity `cmpd` or `error`, even though the method might actually be `atomic`.

For a set S and number k, let S^* denote the set of finite sequences of elements of S, and $S^k = \{s \in S^* : |s| = k\}$. For a sequence s, let $s[i]$ denote the i^{th} element of the sequence. Let $\langle\langle x_0, x_1, \ldots\rangle\rangle$ denote a sequence with elements x_0, x_1, etc.

Let M denote the set of all methods in the program. Let Invoke : $M \rightarrow M^*$ be a function such that Invoke(m) is a sequence containing the methods that m invokes; the order of elements in this sequence is arbitrary but fixed (*e.g.*, lexicographic order). Let *Atom* denote the set of atomicities.

For each method m, the definition of m and the typing rules define an atomicity transfer function f_m, with type $f_m : Atom^{|\text{Invoke}(m)|} \rightarrow Atom$. The arguments to f_m are the atomicities of the methods that m invokes. f_m returns the atomicity of method m. Basically, $f_m(a_1, a_2, \ldots, a_n)$ applies the atomicity typing rules to the body of m, using a_i to determine the atomicity of a call to a method Invoke(m)[i], and constructs the atomicity of m.

For example, for the method `TStack.push` in Figure 3, Invoke(`TStack.push`) = $\langle\langle$`TNode.init`$\rangle\rangle$, and $f_{\texttt{TStack.push}}(a_{\texttt{init}})$ = `mover`; $a_{\texttt{init}}$; (`thisOwner$this` ? `mover`), where the first `mover` corresponds to the `new` expression, $a_{\texttt{init}}$ corresponds to the invocation of `TNode.init`, and the conditional atomicity corresponds to the assignment to `this.head`. For brevity, occurrences of `mover`; $\cdots$ and `const`; $\cdots$ corresponding to accesses to local variables (including parameters) have been simplified away.

The type inference algorithm determines an atomicity for each method. This is expressed as an atomicity assignment atom : $M \rightarrow Atom$. An atomicity assignment is `consistent` if, for all methods m, the atomicity assigned to m is greater than or equal to the atomicity computed for m using the transfer function f_m, *i.e.*,

$$f_m(\text{atom}(\text{Invoke}(m)[1]), \text{atom}(\text{Invoke}(m)[2]), \ldots) \sqsubseteq \text{atom}(m)$$

The partial order on atomicity assignments is the pointwise extension of the partial order on atomicities. Smaller atomicity assignments provide stronger guarantees, thus our aim is to compute the least consistent atomicity assignment for the program. This can be done using a simple fixed-point calculation since all atomicity transfer functions are monotonic and continuous. Recall that every atomicity transfer function is composed from the five operations listed in Section 2.3. It is easy to check that all of these operations, and therefore all compositions of them, are monotonic. Continuity follows from the fact that, for a given program, the set of relevant atomicities is finite.[2] Thus, the least solution to the above constraints can be computed using a standard work-list algorithm based on Tarski's fixed-point theorem [15].

Simplified pseudo-code for the algorithm appears in Figure 10. It uses the Invoke function defined above. It maintains two data structures, atom and tmp_atom, both of which are indexed by method name. The algorithm starts by instantiating every method's atomicity with `const`. In every iteration of the while loop, the transfer function f_m is used to compute a new atomicity for each method using the previously computed atomicities of all methods. If some method's atomicity changed, another iteration of the loop is started; otherwise the algorithm terminates, and atom(m) contains the inferred atomicity of method m. The algorithm we implemented is similar to this pseudo-code but is more efficient, because it avoids re-computing the atomicity of a method in iterations where the atomicities of methods it invokes have not changed.

The time complexity of the atomicity inference algorithm is discussed below. Let $|P|$ denote the size of the program and L_m denote the set of lock expressions that are in scope at the declaration of the method m. L_m includes the method's formal parameters, the the enclosing class's owner parameters, fresh owner parameters in the method declaration, and final fields - in the enclosing class and static fields in the entire program - that appear in a **guarded_by** or a **write_guarded_by** clause. Let A_m denote the set of atom-

[2]There are an infinite number of conditional atomicities if one allows arbitrarily deep nesting, but a given program contains only a finite number of lock expressions, and application of a few simplification rules reduces any conditional atomicity to an equivalent one from a finite set.

```
/** instantiate atomicities **/
foreach method m in the program do
  atom(m) = const
done
/** fix point computation **/
while (change) do
  change = false
  /** compute new atomicity using old atomicities **/
  foreach method m in the program do
    tmp_atom(m) = f_m(atom(Invoke(m)[1]), ...)
  done
  /** set atomicities and check if any change **/
  foreach method m in the program do
    if (atom(m) ⊏ tmp_atom(m))
      change = true
      atom(m) = tmp_atom(m)
    endif
  done
done
```

Figure 10: Atomicity inference algorithm

icities constructed from the lock expressions in L_m. Then, if $atom_m$ is m's atomicity, $atom_m \in A_m$. Let B denote the set of basic atomicities ($|B| = 5$ in this paper). A conditional atomicity evaluates to one of the basic atomicities for each subset $L'_m \subseteq L_m$ of locks held. Thus, $|A_m| = O(|B|^{2^{|L_m|}})$. Let $L_{max} = max\{|L_m|\}$ and $A_{max} = max\{|A_m|\}$ Then $A_{max} = O(|B|^{2^{L_{max}}})$.

Each iteration of the while loop of atomicity inference algorithm in Figure 10 has a time complexity of $O(|P|)$, since applying the transfer function to a method and computing its new atomicity entails scanning the method once. In each iteration, at least one method's atomicity increases by at least one unit. If the number of methods is N, the number of iterations necessary to increase every method's atomicity by at least one unit is $O(N)$. The height of the lattice of atomicities is bounded by $O(A_{max})$. Thus, the time complexity of the atomicity inference algorithm is $O(N \times |P| \times A_{max})$.

Though $L_{max} = O(|P|)$ in the worst case, typically L_{max} is much smaller than $|P|$. For example, in our benchmarks described in Section 8, each class has at most one owner parameter, at most three different final fields of each class are used as guards, very few static fields are used as guards, and the number of formal parameters and owner parameters in method declarations is quite small. We observed that for our benchmarks, the atomicity-inference algorithm took, on an average, $4N$ iterations to terminate. This observation demonstrates that for typical programs, A_{max} can be bounded by a small constant and the complexity of atomicity inference can be approximated to $O(N \times |P|)$.

6. IMPLEMENTATION

Our implementation of type discovery for PRFJ is based on the Kopi compiler (http://www.dms.at/kopi/); we simulated by hand the effects of the changes needed for EPRFJ. Our type checker for EPAJ is based on Flanagan and Freund's Rccjava. We implemented only partial support for owners of the form `owner$e`$_{\texttt{final}}$, sufficient for the benchmarks described in Section 8. The type checker fully supports special owners `unique` and `readonly` which are frequently used in the benchmarks in Section 8. Our implementation of the atomicity type inference algorithm is based on Rccjava and uses Soot (http://www.sable.mcgill.ca/soot) to compute the Invoke function.

7. EXPRESSIVENESS OF EPAJ

Any program typable in either Race Free Java or PRFJ is also typable in EPAJ. EPAJ is more expressive than PRFJ primarily because it allows different protection mechanisms for different fields of a class, and because it allows one field of a class to protect other fields in the same class. Figure 11 illustrates a fragment of `java.io.Writer` that is not typable in PRFJ, because the `lock` field protects the other fields. Other examples include 12 classes in `java.io` and some classes in `java.lang`, `java.lang.ref`, `java.util` and `java.rmi.server`. The class `TspSolver` in `tsp` [16], described in Section 8, is not typable in PRFJ because its fields `TourStackTop` and `MinTourLen` have different guards. `Jigsaw`, a web-server [12], contains a class `ThreadCache` which accesses fields of the class `CachedThread`. The field `alive` of `CachedThread` is protected by the `CachedThread` object, whereas the fields `prev` and `next` of `CachedThread` are protected by the enclosing `ThreadCache` object. Since different fields have different guards, `Jigsaw` is not typable in PRFJ. EPAJ can handle these examples since it allows different fields to be guarded by different locks.

Race Free Java allows different protection mechanisms for different fields of a class, but, as discussed in [4], is less expressive in several other respects relative to PRFJ and hence EPAJ. EPAJ provides better support for parameterization, in particular for self-synchronized objects and readonly objects. For example, in the `TStack` program in Figures 3 and 4 the `TStack` class is parameterized by a `thisOwner` parameter, which is instantiated with `thisThread` in some places and with `self` in others. In the program `raytracer` described in Section 8, the class `Vec` is parameterized by a `thisOwner` parameter, which is instantiated with `thisThread` in some places and with `readonly` in others. Race Free Java does not support such parameterization. As a result, these programs are not typable in Race Free Java but are typable in PRFJ and EPAJ. In addition, EPAJ incorporates a uniqueness analysis to track objects accessed through unique pointers. Race Free Java does not incorporate any uniqueness analysis. For example, in the `elevator` program described in Section 8, the constructor for the `Elevator` class creates instances of the `Lift` class and initializes its fields. Each `Lift` object then spawns a new thread, which is the only thread that accesses fields of the `Lift` object during later execution. EPAJ gives these `Lift` instances a `unique` annotation. The EPAJ typing rules support transfer of unique references from one thread to another, allowing fields of the `Lift` objects to be accessed without holding any locks, in both the instantiating thread and the newly spawned thread. These `Lift` objects are not typable in Race Free Java. They remain untypable in the extension of Race Free Java described in [8], which incorporates an escape analysis but not a uniqueness analysis. EPAJ could easily be made strictly more expressive than that type system, by incorporating the same escape analysis.

```
public abstract class Writer {

 protected Object lock;
 private char[] writeBuffer;
 private final int writeBufferSize = 1024;

 protected Writer(Object lock) {
   if (lock == null)
      throw new NullPointerException();
   this.lock = lock;
 }
 public void write(int c) throws IOException {
   synchronized (lock) {
      if (writeBuffer == null)
         writeBuffer = new char[writeBufferSize];
      writeBuffer[0] = (char) c;
      write(writeBuffer, 0, 1);
   }
 }
}
```

Figure 11: A fragment of java.io.Writer

8. EXPERIENCE WITH TYPE INFERENCE

We tested our atomicity inference algorithm on 6 benchmark programs. Three of the programs (`elevator`, `tsp` and `hedc`) were developed at ETH Zürich and used as benchmarks in [16]. The other three programs (`moldyn`, `raytracer` and `montecarlo`) are part of the Java Grande Forum Benchmark Suite, available at http://www.epcc.ed.ac.uk/.

Our type checker signals a warning for all methods that are non-atomic at some call site. A method is said to be non-atomic at a call site if its atomicity does not simplify to a basic atomicity less than or equal to `atomic` taking into account the locks held at the call site. For example, a method `m` with conditional atomicity `l ? mover : error` is labeled as non-atomic at a call site if the lock `l` is not held at the call site. Some methods labeled non-atomic might actually be atomic for reasons not recognized by the type checker, *e.g.*, methods that are only executed when no other threads exist. Such warnings are counted as false alarms. In summary, our system verified that 91% (640 out of 701) of the methods in the benchmarks are atomic at all call sites.

We reduce the number of warnings reported to the user by clustering related warnings from the type checker into groups and reporting only those warnings that seem to reflect the root cause of errors. For each lock l, we calculate the set of methods whose atomicity possibly becomes non-atomic if the lock l is not held. We partition this set into clusters (with label l) such that m_1 and m_2 belong to the same cluster if one is an ancestor of the other in the program's method invocation graph. Two equal clusters with different labels can be merged into a single cluster. For each cluster, warnings only on the lowest methods in the cluster are reported, where the ordering is defined by $m_1 \geq m_2$ if there is a path from m_1 to m_2 in the method invocation graph.

Table 2 lists the results of running our atomicity checker on the benchmarks. The columns are: LOC (lines of code), Meth. (number of methods), False Alar. (number of false alarms, i.e., warnings signaled by the checker that do not correspond to actual violations of atomicity), Ben. Viol. (number of benign violations, *i.e.*, atomicity violations that do not lead to incorrect program behavior) and Bugs (number of atomicity violations that can lead to incorrect program behavior). The numbers reflect the effect of clustering. Although the number of false alarms is fairly high (about 76% of the warnings), many of them are due to imprecise analysis of race-free static fields and start-join synchronization. This can be addressed either by enhancing the type system or by using types together with run-time atomicity checking, which can already analyze these aspects with fewer false alarms (but gives weaker guarantees).

Benchmark	LOC	Meth.	False Alar.	Ben. Viol.	Bugs
elevator	535	24	4	0	0
tsp	736	24	1	1	0
moldyn	730	23	3	0	0
raytracer	1308	72	2	0	1
montecarlo	3198	179	4	0	0
hedc	7072	379	9	3	2
Total	13581	701	23	4	3

Table 2: Experimental Results

`elevator` is a simple discrete event simulator. Our atomicity type checker produces 4 warnings on `elevator`, all of which are false alarms. 2 false alarms are reported on methods that are executed before any new threads are created. The remaining 2 are in methods `Controls.claimUp` and `Controls.claimDown`. `Controls.claimUp` has the structure

```
claimUp() {
  if (Controls.checkUp()) {
    synchronized(floors[floor]){
      ...
    } } }
```

Both `Controls.checkUp` and the synchronized statement are atomic, so `claimUp` gets atomicity `cmpd`. However, `claimUp` is effectively atomic, because the synchronized block re-checks the condition checked by `checkUp`. The analysis for `Controls.claimDown` is similar.

`tsp` solves the traveling salesman problem. The type-checker produces 2 warnings on `tsp`, of which 1 is a benign atomicity violation and the other is a false alarm. The benign violation is in the method `TspSolver.set_best`, which gets atomicity `cmpd` because it reads `TspSolver.MinTourLen` without holding the lock that write-guards it and then enters a synchronized block. The false alarm is on `Tsp.main`, which gets atomicity `cmpd` because it starts multiple threads.

`moldyn` simulates molecular dynamics. The type checker gives 3 false alarms for `moldyn` due to imprecise analysis of possible races on some static fields (*i.e.* the type checker does not recognize that they are race-free).

`montecarlo` is a financial simulation. The type checker produces 4 false alarms for `montecarlo` due to imprecise analysis of start-join synchronization.

`raytracer` implements a parallel 3-dimensional ray tracing algorithm. There are 2 false alarms in `raytracer` and 1 atomicity bug. The false alarms are due to imprecise analysis of possible races on the static fields `JGFRayTracerBench.staticnumobjects` and

`JGFRayTracerBench.checksum1` The bug is due to a possible race on `JGFRayTracerBench.checksum1` in the method `JGFRayTracerRunner.run` which results in the method getting atomicity `error` and can lead to incorrect program behaviour.

`hedc` is a meta-crawler for searching Internet archives in parallel. It uses Doug Lea's synchronization library, which we treat as part of the program. The type-checker reports 14 warnings of which 9 are false alarms, 3 are benign violations and 2 are bugs. The bugs are due to races on the fields `Task.valid` and `MetaSearchResult.request` as a result of which the methods `Task.run` and `Worker.run` get atomicity `error`. The race on `Task.valid` might result in a task being executed even after it has been cancelled. The race on `MetaSearchResult.request` might lead to a NullPointerException. The false alarms are mostly due to imprecise analysis of possible races.

9. OPTIMIZATION OF RUN-TIME ATOMICITY ANALYSIS

Our type system can be used to improve the efficiency of algorithms for run-time detection of atomicity violations [17, 18, 10]. Runtime techniques might miss some atomicity violations, but they produce very few false alarms (*e.g.*, the algorithms in [18] produce zero false alarms for the benchmarks in Table 2). Thus, users might want to use both run-time and type-based atomicity analysis and give higher priority to the investigation of warnings from the run-time checker. Run-time atomicity checking might slow down the program by a factor of 10 to 100. Types can be used to decrease this overhead drastically.

We use types to optimize the run-time atomicity analysis algorithms in [18]. The algorithms work in part by classifying race-free accesses to fields as movers and other accesses as non-movers. A multi-lockset algorithm based on the lockset algorithm in Eraser [14] monitors all field accesses and synchronization events to determine which fields can be involved in data races. [18] gives two reduction-based algorithms : online (*i.e.*, atomicity is checked on-the-fly as the program executes) and offline (*i.e.*, atomicity is checked after execution of the program). The online algorithm avoids the overhead of storing and retrieving data but may miss atomicity violations by misclassifying an access to a field as race-free, since a subsequent access might lead to a possible race. The offline algorithm augments the online algorithm by incorporating dynamic escape analysis and start-join analysis. Therefore, the offline algorithm is more precise, but slower than the online algorithm.

The EPAJ type-checker can list all the fields in a program that are not involved in a data race (race-free fields). The type-checker can also list the methods that are verified to be atomic. We optimize the reduction based algorithm by monitoring only the fields that are not verified to be race-free by the type-checker and by analyzing atomicity only for methods that are not verified to be atomic by the type-checker.

The results of our experiments are summarized in Table 3. 'Base' gives the execution time of the uninstrumented benchmark. 'Online Slowdown' and 'Offline Slowdown' give the slowdown for the online and offline reduction-based algorithms compared to the Base time. 'OptOnl Slowdown' and 'OptOffl Slowdowm' are similar but reflect the effect of the above optimization. The table demonstrates that the optimization reduces the median slowdown for the online algorithm from 15.7 to 1.5 and the median slowdown for the offline algorithm from 38.1 to 2.1.

Benchmark	Base (sec)	Slowdown			
		Online	Offline	OptOnl	OptOffl
elevator	0.2	1.25	3.30	1.25	1.5
tsp	1.8	119.17	421.11	2.56	2.57
moldyn	25.49	22.97	73.88	1.73	4.37
raytracer	13.88	111.07	45.82	6.87	6.31
montecarlo	16.07	8.47	30.42	1.12	1.12
hedc	0.53	1.13	1.89	1.04	1.51
median	7.84	15.72	38.12	1.49	2.09

Table 3: Optimization of Run-time atomicity analysis

Type discovery uses a lockset algorithm, but the overhead for type discovery is low (less than 20%) because it monitors only a sampling of objects. Furthermore, types discovered after running a program once can be used to optimize run-time atomicity checking for an entire test suite, making the amortized cost of type-discovery negligible.

10. RELATED WORK

The EPAJ type system combines the useful features of *Race Free Java* [7] (different protection mechanisms for different fields) and *Parameterized Race Free Java* (parameterization of classes, special owners). A comparison of EPAJ with these type systems appears in Section 7

In independent work, done concurrently with ours, Flanagan, Freund and Lifshin [9] combine *Race Free Java 2* (RFJ2) [8] with the atomicity types of [11] extended with *protected locks*. Support for protected locks makes their atomicity type system more expressive than ours; this feature can be added to our type system. However, their race-free type system is less expressive than EPRFJ. Specifically, it does not support `unique` as a protection mechanism and it does not allow classes to be parameterized by the protection mechanisms `readonly` and `self` (for self-synchronized objects). It allows a field to be guarded by `final` (corresponding to `readonly`) or `this` (corresponding to `self`), but then all instances of the class must use the same protection mechanism for the field. For example, the program in section 2.2.1 is not typable in their type system. EPRFJ fully supports `unique` and `readonly` and their use as parameters although this significantly complicates the type system [2]. Flanagan *et al.* also present a two step atomicity inference algorithm [9]; race-free types are inferred statically by solving constraints [8], and then atomicity types are inferred using an inference algorithm similar to ours. Their race-free type inference is complete (*i.e.* succeeds for all typable programs) and has a worst case time complexity exponential in the size of the program; this seems unavoidable since the type inference problem is NP-complete. Our run-time type discovery algorithm for race-free types discovers most of the types (98%, in our experiments [2]) and has a worst case time complexity that is linear in the size of the program and the length of the monitored runs.

Choi *et al.* [5] use static analysis to optimize their dynamic data race detection algorithm. This is similar in spirit to our use of types to optimize run-time atomicity checking.

Their static analysis is less effective than type-discovery in some cases, as discussed in [2]. Also, they perform some optimizations that are not possible in the context of atomicity checking.

Acknowledgement. We thank Cormac Flanagan, Stephen Freund and Chandra Boyapati for many helpful comments.

11. REFERENCES

[1] M. Abadi, C. Flanagan, and S. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, to appear.

[2] R. Agarwal, A. Sasturkar, and S. D. Stoller. Type discovery for parameterized race-free Java. Technical Report DAR-04-16, Computer Science Department, SUNY at Stony Brook, Sept. 2004. Available at http://www.cs.sunysb.edu/~stoller/type-discovery/.

[3] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, Jan. 2004.

[4] C. Boyapati and M. C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, 2001.

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

[6] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[7] C. Flanagan and S. Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[8] C. Flanagan and S. Freund. Type inference against races. In *Proc. 11th International Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*. Springer-Verlag, Aug. 2004.

[9] C. Flanagan, S. Freund, and M. Lifshin. Type inference for atomicity. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. ACM Press, 2005.

[10] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 2004.

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

[12] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *Proc. 24th international Conference on Software Engineering (ICSE)*, pages 453–463. ACM Press, May 2002.

[13] A. Sasturkar, R. Agarwal, and S. D. Stoller. Typing rules for Extended Parameterized Atomic Java. Technical Report DAR-05-21, Computer Science Department, SUNY at Stony Brook, Sept. 2004. Available at http://www.cs.sunysb.edu/~amits/papers/atomicity-inference/.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[15] A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific J. of Math*, 5:285–309, 1955.

[16] C. von Praun and T. R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, Oct. 2001.

[17] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Proc. Third Workshop on Runtime Verification (RV)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2003. Available at http://www.cs.sunysb.edu/~stoller/.

[18] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multi-threaded programs. Technical Report DAR-04-2, SUNY at Stony Brook, Computer Science Dept., July 2004. Available at http://www.cs.sunysb.edu/~liqiang/atomicity.html.