Type-Based Race Detection for Java

Cormac Flanagan
Compaq Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301
cormac.flanagan@compaq.com

Stephen N. Freund*
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
freunds@cs.stanford.edu

Abstract

This paper presents a static race detection analysis for multithreaded Java programs. Our analysis is based on a formal type system that is capable of capturing many common synchronization patterns. These patterns include classes with internal synchronization, classes that require client-side synchronization, and thread-local classes. Experience checking over 40,000 lines of Java code with the type system demonstrates that it is an effective approach for eliminating races conditions. On large examples, fewer than 20 additional type annotations per 1000 lines of code were required by the type checker, and we found a number of races in the standard Java libraries and other test programs.

1 Introduction

Race conditions are common, insidious errors in multithreaded programs. A race condition occurs when two threads manipulate a shared data structure simultaneously, without synchronization. Race conditions often result in unexpected program behavior, such as program crashes or incorrect results. They can be avoided by careful programming discipline: protecting each data structure with a lock and acquiring that lock before manipulating the data structure [Bir89]. Since a lock can be held by at most one thread at any time, careful adherence to this lock-based synchronization discipline ensures a race-free program.

Current programming tools provide little support for this synchronization discipline. It is easy to write a program that, by mistake, neglects to perform certain crucial synchronization operations. These synchronization errors are not detected by traditional compile-time checks. Furthermore, because the resulting race conditions are scheduler dependent, they are difficult to catch using testing techniques. A single synchronization error in an otherwise correct program may yield a race condition whose cause may take weeks to identify [SBN+97].

This paper investigates a static analysis system for detecting race conditions in Java programs. The analysis supports the lock-based synchronization discipline by tracking the protecting lock for each shared field in the program and verifying that the appropriate lock is held whenever a shared field is accessed. We express the reasoning and checks performed by this analysis as an extension of Java's type system.

This work builds on an earlier paper that describes a race-free type system for a concurrent object calculus [FA99a]. We start by adapting that type system to a core subset of Java. This initial type system is sufficient to verify some example programs as race free. In order to accommodate larger, more realistic, multithreaded programs, we extend the initial type system with a number of additional features. These features include:

1. classes parameterized by locks, which allow the fields of a class to be protected by some lock external to the class;
2. the notion of objects that are local to a particular thread and therefore safely accessible without synchronization; and
3. mechanisms for escaping the type system in places where it proves too restrictive, or where a particular race condition is considered benign.

To evaluate the utility of the resulting type system, we implemented a type checker and tested it on a variety of Java programs totaling over 40,000 lines of code. These programs include the standard Java input/output package `java.io`; an interpreter for the web

*This work was completed while the author was employed at the Compaq Systems Research Center.

PLDI 2000, Vancouver, British Columbia, Canada.

scripting language WebL; and Ambit, a mobile object calculus implementation.

Checking these programs using our type system requires adding some additional type annotations. This annotation burden is not excessive; typically fewer than 20 annotations were required per 1000 lines of code. Most of the annotations were inserted based on feedback from the type checker. This annotation process proceeded at a rate of roughly 1000 lines of code per programmer-hour. During this process, we discovered a number of race conditions in the programs being checked, including one race condition in `java.util.Vector`, four in the `java.io` package, and five in the WebL implementation. Although it is far from complete, the type system proved sufficiently expressive to accommodate the majority of synchronization patterns present in these programs.

The presentation of our results proceeds as follows. Section 2 introduces a small concurrent subset of Java, which we use to provide a formal description of our type system. Section 3 describes an initial race-free type system. We extend the system to include classes parameterized by locks in Section 4 and thread-local classes in Section 5. Section 6 describes our prototype implementation, including the escape mechanisms. Section 7 discusses our experiences checking several Java programs. We relate this work to other projects in Section 8, and we conclude in Section 9. The Appendix contains a formal description of the type system.

2 A Multithreaded Subset of Java

This section introduces a small multithreaded subset of Java, ConcurrentJava. This language is derived from ClassicJava [FKF98], a sequential subset of Java, and we adopt much of the type structure and semantics of ClassicJava.

2.1 Syntax and Informal Semantics

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

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

$$
\begin{array}{rcl}
\mathit{cn} & \in & \text{class names} \\
\mathit{fd} & \in & \text{field names} \\
\mathit{mn} & \in & \text{method names} \\
x, y & \in & \text{variable names}
\end{array}
$$

Figure 1: The grammar for ConcurrentJava.

The syntax of the `synchronized` and `fork` expressions and the rest of ConcurrentJava is shown in Figure 1. A program is a sequence of class definitions together with an initial expression, which is the starting point for the program's execution. Each class definition associates a class name with a class body consisting of a super class, a sequence of field declarations, and a sequence of method declarations. A field declaration includes an initialization expression and an optional `final` modifier; if this modifier is present, then the field cannot be updated after initialization. A method declaration consists of the method name, its return type, number and types of its arguments, and an expression for the method body. Types include class types and integers. Class types include class names introduced by the program, as well as the predefined class `Object`, which serves as the root of the class hierarchy. Expressions include the typical operations for object allocation, field access and update, method invocation, and variable binding and reference, as well as the concurrency primitives.

2.2 Locks Against Races

We present example programs in an extended language with integer and boolean constants and operations, and the constant `null`. We use $e_1; e_2$ to abbreviate `let` x `=` e_1 `in` e_2, where x does not occur free in e_2, and we sometimes enclose expressions in braces for clarity.

Multithreaded ConcurrentJava programs are prone to race conditions, as illustrated by the following program, which allocates a new bank account, and makes two deposits into the account in parallel:

```
class Account {
  int balance = 0
  int deposit (int x) {
    this.balance = this.balance + x
  }
}

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

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

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

```
class Account {
  int balance = 0
  int deposit(int x) {
    synchronized this in {
      this.balance = this.balance + x
    }
  }
}
```

The modified account implementation is race free and will behave correctly even when multiple deposits are made to the account concurrently.

3 Types Against Races

In practice, race conditions are commonly avoided by the lock-based synchronization discipline used in the example above. This section presents a type system that supports this programming discipline. The type system needs to verify that each field has an associated protecting lock that is held whenever the field is accessed or updated. In order to verify this property, the type system:

1. associates a protecting lock with each field declaration, and
2. tracks the set of locks held at each program point.

We rely on the programmer to aid the verification process by providing a small number of additional type annotations. The type annotation `guarded_by` l on a field declaration states that the field is protected by the lock expression l; the type system then verifies that this lock is held whenever the field is accessed or updated. The type annotation `requires` $l_1, \ldots, l_n$ on a method declaration states that the locks $l_1, \ldots, l_n$ are held on method entry; the type system verifies that these locks are indeed held at each call-site of the method, and checks that the method body is race-free given this assumption. We extend the syntax of field and method declarations to include these type annotations.

$$
\begin{array}{rcll}
\mathit{field} & ::= & [\texttt{final}]_{\mathrm{opt}}\ t\ \mathit{fd}\ \texttt{guarded\_by}\ l\ \texttt{=}\ e & \\
\mathit{meth} & ::= & t\ \mathit{mn}\texttt{(}\mathit{arg}^*\texttt{)}\ \texttt{requires}\ \mathit{ls}\ \{\ e\ \} & \\
\mathit{ls} & ::= & l^* & \text{(lock set)} \\
l & ::= & e & \text{(lock expression)}
\end{array}
$$

We refer to the extended language as RaceFreeJava.

To ensure that each field is consistently protected by a particular lock, irrespective of any assignments performed by the program, the type system requires that the lock expression in a `guarded_by` clause be a *final expression.* A final expression is either a reference to an immutable variable[1], or a field access $e.\mathit{fd}$, where e is a final expression and fd is a final field. The type system also requires that the lock expressions in a `requires` clause be final for similar reasons.

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

$$P; E; \mathit{ls} \vdash e : t\,.$$

Here, P (the program being checked) is included in the judgment to provide information about class definitions in the program; E is an environment providing types for the free variables of e; ls is a set of final expressions describing the locks that are held when the expression e is evaluated; and t is the type of e. Thus, the type rules track the set of locks held each program point. The rule [exp fork] for `fork` e checks the expression e using the empty lock set since new threads do not inherit locks held by their parent.

[exp fork]
$$\frac{P; E; \emptyset \vdash e : t}{P; E; \mathit{ls} \vdash \texttt{fork}\ e : \texttt{int}}$$

[1] All variables are immutable in RaceFreeJava, but only final variables are in Java.

The rule [EXP SYNC] for `synchronized` e_1 `in` e_2 checks that e_1 is a final expression of some class type c, and then type checks e_2 in an extended lock set that includes e_1, reflecting the fact that the lock e_1 is always held when evaluating e_2.

[EXP SYNC]
$$\frac{P; E \vdash_{\text{final}} e_1 : c \qquad P; E; ls \cup \{e_1\} \vdash e_2 : t}{P; E; ls \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t}$$

The antecedent $P; E \vdash_{\text{final}} e_1 : c$ checks that e_1 is a final expression of type c.

The rule [EXP REF] for $e.fd$ checks that e is a well-typed expression of some class type c and that c declares or inherits a field fd of type t, guarded by lock l.

[EXP REF]
$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash ([\texttt{final}]_{\text{opt}}\ t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e') \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fd : [e/\texttt{this}]t}$$

It remains to check that the lock l guarding fd is held at this program point; i.e., that l denotes the same lock as some expression l' in the current lock set. This requires checking the semantic equivalence of the two expressions l and l', which is in general undecidable.

One approach is to conservatively approximate semantic equivalence by syntactic equivalence, and simply to check that $l \equiv l'$. This approximation is overly conservative in many cases however. In particular, occurrences of `this` in the lock expression l refer to the object being dereferenced, which is the same object as that denoted by e. To account for the aliasing of `this` and e, the type system replaces all occurrences of `this` in l by e, and then checks that $[e/\texttt{this}]l \equiv l'$ for some l' in the current lock set. This check is performed by the antecedent $P; E \vdash [e/\texttt{this}]l \in ls$. It is a sound approximation of semantic equivalence and has been sufficient for all programs we have inspected.

A similar aliasing situation arises in the next section, where we introduce types containing lock expressions. To accommodate this future extension, we include the appropriate substitutions for types here, and yield $[e/\texttt{this}]t$ as the type of the field access after checking that it is a well-formed type.

The rule [EXP ASSIGN] for $e.fd = e'$ ensures that the appropriate lock is held whenever a field is updated. The rule [EXP INVOKE] for a method invocation ensures that all locks in the `requires` clause of a method declaration are held at each call site of the method.

[EXP ASSIGN]
$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash (t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e'') \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E; ls \vdash e' : [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fd = e' : [e/\texttt{this}]t}$$

[EXP INVOKE]
$$\frac{\begin{array}{c} P; E; ls_1 \vdash e : c \\ P; E \vdash (t\ mn(s_j\ y_j\ {}^{j \in 1 \ldots n})\ \texttt{requires}\ ls_2\ \{\ e'\ \}) \in c \\ P; E; ls_1 \vdash e_j : [e/\texttt{this}]s_j \\ P; E \vdash [e/\texttt{this}]ls_2 \subseteq ls_1 \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls_1 \vdash e.mn(e_{1 \ldots n}) : [e/\texttt{this}]t}$$

The remaining type rules are straightforward and similar to those of CLASSICJAVA. The complete set of type judgments and rules is contained in Appendix A.

3.1 Race-Free Bank Accounts

We can use this type system to verify that the synchronized bank account implementation is race free. We must first add a type annotation stating that the field `balance` is guarded by `this`. When no locks are required to invoke a method we omit its `requires` clause, as we have done for `deposit`.

```
class Account {
  int balance guarded_by this = 0
  int deposit(int x) {
    synchronized this in {
      this.balance = this.balance + x
    }
  }
}

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

An alternative implementation of the bank account may rely on its clients to perform the necessary synchronization operations:

```
class Account {
  int balance guarded_by this = 0
  int deposit(int x) requires this {
    this.balance = this.balance + x
  }
}

let Account a = new Account in {
  fork { synchronized a in a.deposit(10) }
  fork { synchronized a in a.deposit(10) }
}
```

In this example, the method signature

```
int deposit(int x) requires this
```

explicates the requirement that the object's lock must be acquired before calling `deposit`. Since the necessary lock is indeed held at each call site, this program is also well typed and race free.

4 External Locks

The type system of the previous section can verify the absence of races in a number of interesting examples. However, larger, more realistic programs frequently use a variety of synchronization patterns, some of which cannot be captured by the system presented so far. To accomodate such programs, we extend the RaceFreeJava type system with additional features. This section presents classes parameterized by locks, and Section 5 introduces thread-local classes.

The type system requires that every field be guarded by a final expression of the form $x.fd_1.\cdots.fd_n$. Since the only variable in scope at a field declaration is `this`, the fields of an object must be protected by a lock that is accessible from the object. In some cases, however, we would like to protect the fields of an object by some lock external to the object. For example, all of the fields in a linked list might naturally be protected by some object external to the list.

To accommodate this programming pattern, we extend RaceFreeJava to allow classes to be parameterized by external locks:

$$\begin{array}{rcll} \mathit{defn} & ::= & \texttt{class}\ \mathit{cn}\texttt{<}\mathit{garg}^*\texttt{>}\ \mathit{body} & \\ \mathit{garg} & ::= & \texttt{ghost}\ t\ x & \text{(ghost decl)} \\ c & ::= & \mathit{cn}\texttt{<}l^*\texttt{>} \ \mid\ \texttt{Object} & \end{array}$$

A class definition now contains a (possibly empty) sequence of formal parameters or *ghost variables*. These ghost variables are used by the type system to verify that the program is race free; they do not affect the run-time behavior of the program. In particular, they can appear only in type annotations and not in regular code. A class type c consists of a class name cn parameterized by a sequence of final expressions. The number and type of these expressions must match the formal parameters of the class.

Type checking of parameterized classes is handled via substitution. If

$$\texttt{class}\ \mathit{cn}\texttt{<ghost}\ t_1\ x_1, \ldots, \texttt{ghost}\ t_n\ x_n\texttt{>}\ \mathit{body}$$

is a well-formed class definition, then for any final expressions $l_1, \ldots, l_n$ of the appropriate types, we consider $cn\texttt{<}l_1, \ldots, l_n\texttt{>}$ to be a valid instantiated class type, with associated instantiated class definition

$$\texttt{class}\ \mathit{cn}\texttt{<}l_1, \ldots, l_n\texttt{>}\ \ [l_1/x_1, \ldots, l_n/x_n]\mathit{body}$$

A few modifications to the type rules are necessary to accommodate parameterized classes. These modifications are described in Appendix B.

4.1 Using External Locks

To illustrate the use of external locks, consider the dictionary implementation of Figure 2. A dictionary maps keys to values. In our implementation, a dictionary is represented as an object containing a linked list of `Nodes`, where each `Node` contains a key, a value, and a next pointer.

For efficiency reasons, we would like to protect the entire dictionary, including its linked list, with the lock of the dictionary. To accomplish this, the class `Node` is parameterized by the enclosing dictionary `d`; the fields of `Node` are guarded by `d`; and each method of `Node` requires that `d` is held on entry. Each method of `Dictionary` first acquires the dictionary lock and then proceeds with the appropriate manipulation of the linked list. Since all fields of the linked list are protected by the dictionary lock, the type system verifies that this program is well typed and race free.

5 Thread-Local Classes

Large multithreaded programs typically have sections of code that operate on data that is not shared across multiple threads. For example, only a single thread in a concurrent web server may need to access the information about a particular request. Objects used in this fashion require no synchronization and should not need to have locks guarding their fields. To accommodate this situation, we introduce the concept of thread-local classes. We extend the grammar to allow an optional `thread_local` modifier on class definitions and to make the `guarded_by` clause on field declarations optional in a thread-local class:

$$\begin{array}{rcl} \mathit{defn} & ::= & [\texttt{thread\_local}]_{\mathrm{opt}}\ \texttt{class}\ \mathit{cn}\texttt{<}\mathit{garg}^*\texttt{>}\ \mathit{body} \\ \mathit{field} & ::= & [\texttt{final}]_{\mathrm{opt}}\ t\ \mathit{fd}\ [\texttt{guarded\_by}\ l]_{\mathrm{opt}}\ \ \texttt{=}\ e \end{array}$$

An example of a thread-local class appears in Figure 3. The class `Crawler` is a concurrent web crawler that processes a page by iterating over its links and forking new threads to process the linked pages. The `LinkEnumerator` class, which parses the text of the page to find links, is not shared among threads. Therefore, it is declared as a `thread_local` class and contains unguarded fields.

A simple form of escape analysis is used to enforce single-threaded use of thread-local objects. A type is *thread-shared* provided it is not a thread-local class type. The type system must ensure that thread-local objects are not accessible from thread-shared objects. Therefore, a thread-shared class declaration must (1) have a thread-shared superclass and (2) contain only shareable fields. A field is *shareable* only if it has a thread-shared type and is either final or protected by a lock. Also, the free variables of a forked expression must be of a thread-shared type. The rules for thread-shared types and `fork` appear in Appendix C.

```
class Node<ghost Dictionary d> {
  String key guarded_by d = null
  Object value guarded_by d = null
  Node<d> next guarded_by d = null

  void init(String k, Object v, Node<d> n)
      requires d {
    node.key = k;
    node.value = v;
    node.next = n
  }
  void update(String k, Object v) requires d {
    if (this.key.equals(k)) {
      this.value = v
    } else if (this.next != null) {
      this.next.update(k,v)
    }
  }
  ...
}
```

```
class Dictionary {
  Node<this> head guarded_by this = null

  void put(String k, Object v) {
    synchronized this in {
      if (this.contains(k)) {
        this.head.update(k,v)
      } else {
        let Node<this> node =
                new Node<this> in {
          node.init(k,v,this.head);
          this.head = node
        }
      }
    }
  }
  ...
}
```

Figure 2: A synchronized dictionary.

```
thread_local class LinkEnumerator {
  String text = null
  int index = 0

  void init(String t) {
    this.text = t
  }
  boolean hasMoreLinks() { ... }
  String nextLink() { ... }
}
```

```
class Crawler {
  final Set visited = new Set
  void process(String url) {
    if (!visited.add(url)) {
      let String text = loadPageText(url) in
        let LinkEnumerator enum =
            new LinkEnumerator in {
          enum.init(text);
          while (enum.hasMoreLinks()) {
            let String link = enum.nextLink() in
              fork { this.process(link) }
          }
        }
    }
  }
  ...
}

let Crawler c = new Crawler in
  c.process("http://www.research.compaq.com")
```

Figure 3: A concurrent web crawler using a thread-local enumeration class.

Interestingly, our type system permits a thread-local class to have a thread-shared superclass. This design permits us to maintain `Object` (which is thread-shared) as the root of the class hierarchy, as it is in Java. However, it also permits a thread-local object to be viewed as an instance of a thread-shared class and hence to be shared between threads. This sharing does not cause a problem unless the object is downcast back to the thread-local type in a thread other than the one in which it was created. This downcast would make unguarded fields in the subclass visible to more than one thread.

To eliminate this possibility, our type system forbids downcasts from a thread-shared type to a thread-local type. This restriction applies to explicit cast operations[2] and, also, to the implicit downcasts that occur during dynamic dispatch. To avoid such implicit downcasts, our type system requires a thread-local class not to override any methods declared in a thread-shared superclass.

Alternatively, if these static requirements are too restrictive, a compiler could insert code to track the allocating thread of each object and dynamically check that thread-shared to thread-local downcasts are only performed by the appropriate thread.

6 Implementation

We have implemented the RaceFreeJava type system for the full Java language [GJS96]. This race condition checker, `rccjava`, extends the type system outlined so far with the missing Java features, including arrays, interfaces, constructors, static fields and methods, inner classes, and so on. Only thread-local arrays posed any technical challenges, but space considerations prohibit a full discussion of those challenges here.

The additional type information required by `rccjava` is embedded in Java comments to preserve compatibility with existing Java tools, such as compilers. Specifically, comments that start with the character "`#`" are treated as type annotations by `rccjava`. See Figure 5 for an example.

The `rccjava` tool was built on top of an existing Java front-end that includes a scanner, parser, and type checker. The extensions for race detection were relatively straightforward to add to the existing code base and required approximately 5,000 lines of new code. The major additions were maintaining the lock set during type checking, implementing syntactic equality and substitution on abstract syntax trees, and incorporating classes parameterized by locks.

[2]RaceFreeJava does not contain explicit casts, but the Java language does.

An important goal in the design of `rccjava` was to provide a cost-effective way to detect race conditions statically. Thus, it was important to minimize both the number of annotations required and the number of false alarms produced. In order to attain this goal, `rccjava` was designed to be able to relax the formal type system in several ways and, also, to infer default annotations for unannotated code. These features are described below.

6.1 Escape mechanisms

`Rccjava` provides mechanisms for escaping from the type system when it proves too restrictive. The simplest escape mechanism is the `no_warn` annotation, which turns off certain kinds of warnings on a particular line of code, e.g.

```
f.a = 3; //# no_warn race
```

This annotation is commonly used if a particular race condition is considered benign.

Also, `rccjava` may be configured with a command line flag to ignore all errors of a particular kind. For example, the "`-no_warn thread_local_override`" flag turns off the restrictions whereby a thread-local class cannot override a method of its thread-shared superclass.

The `holds` annotation asserts that a particular lock is held at a given program point:

```
//# holds f
f.a = 3;
```

This annotation puts `f` into the lock set for the remainder of block of statements in which it appears. As with the `no_warn` annotations, `rccjava` may be configured to make global assumptions about when locks are held. For instance, when run with the command line flag "`-constructor_holds_lock`", `rccjava` assumes that the lock `this` is held in constructors. This is sound as long as references to `this` are not passed to other threads before the constructor call returns. Violations of this assumption are unlikely, and using it eliminates a large number spurious warnings.

6.2 Default Annotations

Although type inference for the `rccjava` type system remains for future work, `rccjava` does construct default annotations for unannotated classes and fields. The heuristics used to compute default annotations are:

- A class with no annotations and no synchronized methods is thread local by default, unless the class is `java.lang.Object` or a subclass of `java.lang.Thread`.

- Unguarded non-final instance fields in thread shared classes are guarded by `this`.
- Unguarded non-final static fields are guarded by the class object for the class to which they belong.
- A `guarded_by` annotation is permitted on a class declaration, and it applies to all fields of the class.

These heuristics are not guaranteed to produce the correct annotations, but experience has shown that they save a significant amount of time while annotating large programs. Roughly 90% of the classes in the test programs described below are treated correctly by these heuristics.

7 Evaluation

To test the effectiveness of `rccjava` as a static race detection tool, we used it to check several multithreaded Java programs. Our test cases include two representative single classes, `java.util.Hashtable` and `java.util.Vector`, and several large programs, including `java.io`, the Java input/output package (version 1.1) [Jav98]; Ambit, an implementation of a mobile object calculus [Car97]; and an interpreter and run-time environment for WebL, a language for automating web-based tasks [KM98].

These five programs use a variety of synchronization patterns, most of which were captured easily with `rccjava` annotations. `Rccjava` was run with the command line flags "`-no_warn thread_local_override`" and "`-constructor_holds_lock`" for these tests (see Section 6.1). Although these flags may cause `rccjava` to miss some potential races, they significantly reduce the number of false alarms reported and provide the most effective way to deal with existing programs that were not written with this type system in mind. Table 1 summarizes our experience in checking these programs. It shows the number of annotations and time required to annotate each program, as well as the number of race conditions found in each program. The time includes both the time spent by the programmer inserting annotations and the time to run the tool.

Figure 4 breaks down the annotation count into the different categories of annotations, normalized to the frequency with which they appear in 1000 lines of code. For the large programs, fewer than 20 annotations were required per 1000 lines. Most of these annotations were clustered in the small number of classes manipulated from different threads. The majority of classes typically required very few or no annotations. Evidence of this pattern is reflected in the statistics for the single class examples, which have higher annotation frequencies than the larger programs. `Hashtable` has a high occurrence of annotations concerning class parameters and arguments because it contains a linked list similar to that of Figure 2. Interestingly, restructuring `Hashtable` to declare the linked list as an inner class within the scope of the protecting lock reduces the number of annotations to 25.

We discovered race conditions in three of the five case studies, despite most of the code in these examples being well tested and relatively mature. Of the four races found in `java.io`, one was fixed in JDK version 1.2. We also found benign race conditions in all test cases.

Figure 5 contains an excerpt from `java.util.Vector` that illustrates a typical race condition caught during our experiments. Suppose that there are two threads manipulating a shared `Vector` object. If one thread calls `lastIndexOf(elem)` for some `elem`, that method may access `elementCount` without acquiring the lock of the `Vector` object. However, the other thread may call `removeAllElements` (which sets `elementCount` to 0) and then call `trimToSize` (which resets `elementData` to an array of length 0). Thus, an array out of bounds exception will be triggered when the first thread enters the binary version of `lastIndexOf` and accesses the `elementData` array based on the old value of `elementCount`. Declaring both versions of `lastIndexOf` to be synchronized removes this race condition.

8 Related Work

A number of tools have been developed for detecting race conditions, both statically and dynamically. Warlock [Ste93] is a static race detection system for ANSI C programs. It supports the lock-based synchronization discipline through annotations similar to ours. However, Warlock uses a different analysis mechanism; it works by tracing execution paths through the program, but it fails to trace paths through loops or recursive function calls, and thus may not detect certain races. In addition, Warlock assumes, but does not verify, the thread-local annotations introduced by the programmer. However, these soundness issues have not prevented Warlock from being a practical tool. It has been used to catch races in several programs, including an X-windows library.

The extended static checker for Java (Esc/Java) is a tool for static detection of software defects [LSS99, DLNS98]. It uses an underlying automatic theorem prover to reason about the program's behavior and to verify the absence of certain kinds of errors, such as null dereferences and index out of bounds errors. ESC/Java supports multithreaded programming via annotations similar to our `guarded_by` and `requires` clauses, and

Program	Lines Of Code	Programmer Time (hrs)	Annotations	Races Found
`java.util.Hashtable`	440	0.5	60	0
`java.util.Vector`	430	0.5	10	1
`java.io.*`	16,000	16.0	139	4
Ambit	4,500	4.0	38	0
WebL	20,000	12.0	358	5

Table 1: Programs analyzed using `rccjava`.

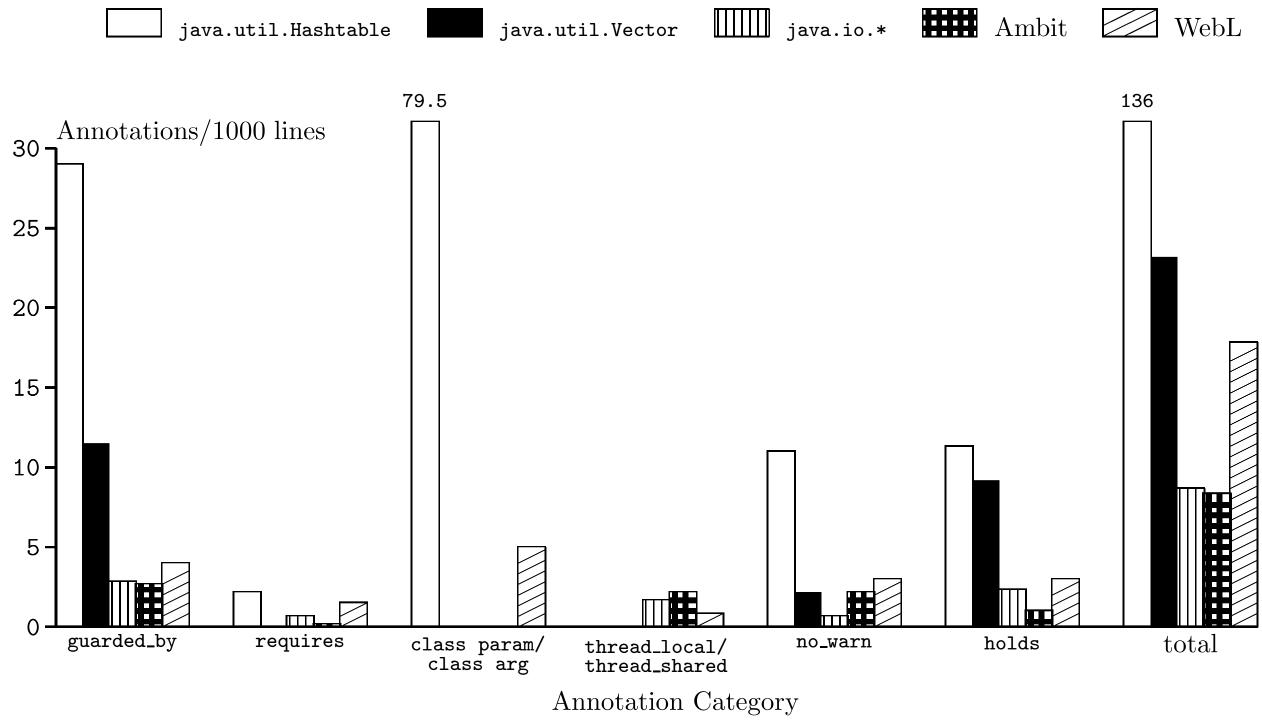

Figure 4: Number of `rccjava` annotations added to each program.

verifies that the appropriate lock is held whenever a guarded field is accessed. However, it may still permit race conditions on unguarded fields, since it does not verify that such unguarded fields only occur in thread-local classes. Overall, Esc/Java is a complex but powerful tool capable of detecting many kinds of errors, whereas `rccjava` is a lightweight tool tuned specifically for detecting race conditions.

Aiken and Gay [AG98] also investigate static race detection, in the context of SPMD programs. Since synchronization in these programs is performed using barriers, as opposed to locks, their system does not need to track the locks held at each program point or the association between locks the fields they protect. Their system has been used successfully on a number of SPMD programs.

Eraser is a tool for detecting race conditions and deadlocks dynamically [SBN+97], rather than statically. This approach has the advantage of being able to check unannotated programs, but it may fail to detect certain errors because of insufficient test coverage.

A variety of other approaches have been developed for race and deadlock prevention; they are discussed in more detail in an earlier paper [FA99b].

A number of formal calculi for Java have been presented in recent literature. These include attempts to model the entire Java language [DE97, Sym97, NvO98] and, also, smaller systems designed to study specific features and extensions [IPW99]. We chose to use the ClassicJava calculus of Flatt, Krishnamurthi, and Felleisen [FKF98] as the starting point for our study.

There have been many suggested language extensions for supporting Java classes parameterized by types [OW97, BOSW98, AFM97, BLM96, CJ98].

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

  synchronized void trimToSize() { ... }
  synchronized boolean removeAllElements() { ... }

  synchronized int lastIndexOf(Object elem, int n) {
     for (int i = n ; --i >= 0 ; )
       if (elem.equals(elementData[i])) { ... }
  }

  int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount); // race!!!
  }
  ...
}
```

Figure 5: Excerpt from `java.util.Vector`.

Our work uses a different notion of parameterization, namely, classes parameterized by *values* (more specifically, lock expressions). Apart from this distinction, our class parameterization approach most closely follows that of GJ [BOSW98], in that information about class parameters is not preserved at run time.

The lock sets used in our type systems are similar to *effects* [JG91, LG88, Nie96] since the locks held on entry to an expression constrain the effects that it may produce. It may be possible to adapt existing techniques for effect reconstruction [TT94, TT97, ANN97, TJ92] to our setting to reduce the number of type annotations required.

Our type system verifies that objects of a `thread_local` type are never shared between threads. Much work has been done on the related problem of inferring which objects are not shared between threads [CGS+99, Bla99, BH99, WR99, ACSE99]. This work has primarily focused on optimizing synchronization operations, but it may be possible to adapt this work to reduce or eliminate the need for `thread_local` annotations.

9 Conclusions and Future Work

Race conditions are difficult to catch using traditional testing techniques. They persist even in common, relatively mature Java programs. In this paper, we have presented a type system for catching race conditions statically and described `rccjava`, an implementation of this system for Java. Our experience with `rccjava` indicates that this technique is a promising approach for building more reliable multithreaded software.

Because the type system is modular, it enables race conditions to be detected early in the development cycle, before the entire program has been written. The type system does require the programmer to write additional type annotations, but these annotations also function as documentation of the locking strategies used by the program.

To reduce the annotation overhead further, we are currently studying the issue of type inference. We are also considering `rccjava` extensions to support additional synchronization patterns. These extensions may include methods parameterized by locks and support for reader-writer locks.

Availability: We intend to make the `rccjava` prototype implementation available for download from `http://www.research.compaq.com`.

Acknowledgments: Thanks to John Mitchell for comments on an earlier draft of this paper, and to Hannes Marais and Martín Abadi for several useful discussions.

References

[ACSE99] Jonathan Aldrich, Craig Chambers, Emin Gun Sirer, and Susan Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Sixth International Static Analysis Symposium*, September 1999.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, October 1997.

[AG98] Alexander Aiken and David Gay. Barrier inference. In *Proceedings of the 25th Symposium on Principles of Programming Languages*, pages 243–354, 1998.

[ANN97] Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, 1997.

[BH99] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[Bla99] Bruno Blanchet. Escape analysis for object-oriented languages. Application to Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[BLM96] J. Bank, B. Liskov, and A. Myers. Parameterized Types and Java. Technical Report MIT/LCS/TM-553, Massachussetts Institute of Technology, 1996.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, October 1998.

[Car97] Luca Cardelli. Mobile ambient synchronization. Technical Report 1997-013, Digital Systems Research Center, Palo Alto, CA, July 1997.

[CGS+99] J.D. Choi, M. Gupta, M. Serrano, V.C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[CJ98] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, October 1998.

[DE97] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *European Conference On Object Oriented Programming*, pages 389–418, 1997.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA, December 1998.

[FA99a] Cormac Flanagan and Martín Abadi. Object types against races. In *Proceedings of CONCUR*, August 1999.

[FA99b] Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proceedings of European Symposium on Programming*, March 1999.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[IPW99] Atsushi Igarishi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

[Jav98] JavaSoft. Java Developers Kit, version 1.1. `http://java.sun.com`, 1998.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[KM98] Thomas Kistler and Johannes Marais. WebL – a programming language for the web. *Computer Networks and ISDN Systems*, 30:259–270, April 1998.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 47–57, 1988.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Report 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999. Also appeared in Formal Techniques for Java Programs, workshop proceedings. Bart Jacobs, Gary T. Leavens, Peter Muller, and Arnd Poetzsch-Heffter, editors. Technical Report 251, Fernuniversitat Hagen, 1999.

[Nie96] Flemming Nielson. Annotated type and effect systems. *ACM Computing Surveys*, 28(2):344–345, 1996. Invited position statement for the Symposium on Models of Programming Languages and Computation.

[NvO98] Tobias Nipkow and David von Oheimb. Java_{light} is type-safe - definitely. In *Proc. 25th ACM Symposium on Principles of Programming Languages*, January 1998.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Ste93] Nicholas Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[Sym97] Don Syme. Proving Java type soundness. Technical Report 427, University of Cambridge Computer Laboratory Technical Report, 1997.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of ACM Conference on Object Oriented Languages and Systems*, November 1999.

A The Initial Type System

This appendix presents the type system described in Section 3. We introduce class parameters and thread-local classes in Appendices B and C, respectively.

We first define a number of predicates used in the type system informally. These predicates are based on similar predicates from [FKF98], and we refer the reader to that paper for their precise formulation.

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

A typing environment is defined as

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

We define the type system using the following judgments.

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash \mathit{defn}$	defn is a well-formed class definition
$P; E \vdash \mathit{wf}$	E is a well-formed typing environment
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P; E \vdash \mathit{field}$	field is a well-formed field
$P; E \vdash t$	t is a well-formed type
$P; E \vdash s <: t$	s is a subtype of t
$P; E \vdash \mathit{defn}$	defn is a class defined in P
$P; E \vdash \mathit{field} \in c$	class c declares/inherits field
$P; E \vdash \mathit{meth} \in c$	class c declares/inherits meth
$P; E \vdash_{\mathrm{final}} l : t$	l is a final expression with type t
$P; E \vdash \mathit{ls}$	ls is a well-formed lock set
$P; E \vdash l \in \mathit{ls}$	l appears in ls
$P; E \vdash \mathit{ls}_1 \subseteq \mathit{ls}_2$	lock set ls_1 is contained in ls_2
$P; E; \mathit{ls} \vdash e : t$	expression e has type t

The typing rules for these judgments are presented below.

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{ClassOnce}(P) \quad \mathit{WFClasses}(P) \\ \mathit{FieldsOnce}(P) \quad \mathit{MethodsOncePerClass}(P) \\ \mathit{OverridesOK}(P) \\ P = \mathit{defn}_{1\ldots n}\ e \quad P \vdash \mathit{defn}_i \quad P; \emptyset; \emptyset \vdash e : t \end{array}}{\vdash P : t}$$

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

[CLASS]

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

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

[ENV EMPTY]

$$\frac{}{P; \emptyset \vdash \mathit{wf}}$$

[ENV X]

$$\frac{P; E \vdash t \quad x \notin \mathit{Dom}(E)}{P; E, t\ x \vdash \mathit{wf}}$$

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

[CLASS DEFINITION]

$$\frac{P; E \vdash \mathit{wf} \quad \texttt{class}\ c\ \ldots\ \in P}{P; E \vdash \texttt{class}\ c\ \ldots}$$

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

[TYPE C]

$$\frac{P; E \vdash \texttt{class}\ c\ \ldots}{P; E \vdash c}$$

[TYPE Object]

$$\frac{P; E \vdash \mathit{wf}}{P; E \vdash \texttt{Object}}$$

[TYPE INT]

$$\frac{P; E \vdash \mathit{wf}}{P; E \vdash \texttt{int}}$$

$\boxed{P; E \vdash t_1 <: t_2}$

[SUBTYPE REFL]

$$\frac{P; E \vdash t}{P; E \vdash t <: t}$$

[SUBTYPE CLASS]

$$\frac{\begin{array}{c} P; E \vdash c_1 <: c_2 \\ P; E \vdash \texttt{class}\ c_2\ \texttt{extends}\ c_3 \ldots \end{array}}{P; E \vdash c_1 <: c_3}$$

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

[FIELD]

$$\frac{\begin{array}{c} P; E \vdash_{\mathrm{final}} l : c \\ P; E; \emptyset \vdash e : t \end{array}}{P; E \vdash [\texttt{final}]_{\mathrm{opt}}\ t\ \mathit{fd}\ \texttt{guarded\_by}\ l\ \texttt{=}\ e}$$

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

[FIELD DECLARED]

$$\frac{P; E \vdash \texttt{class}\ c\ \ldots\ \{\ \ldots\ \mathit{field}\ \ldots\ \}}{P; E \vdash \mathit{field} \in c}$$

[FIELD INHERITED]

$$\frac{\begin{array}{c} P; E \vdash \texttt{class}\ c\ \texttt{extends}\ c'\ \ldots \\ P; E \vdash \mathit{field} \in c' \end{array}}{P; E \vdash \mathit{field} \in c}$$

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

[METHOD]

$$\frac{\begin{array}{c} P;E \vdash t \qquad P;E \vdash ls \\ P;E,arg_{1\ldots n};ls \vdash e : t \end{array}}{P;E \vdash t\ mn(arg_{1\ldots n})\ \texttt{requires}\ ls\ \{\ e\ \}}$$

$\boxed{P;E \vdash meth \in c}$

[METHOD DECLARED]

$$\frac{P;E \vdash \texttt{class}\ c\ \ldots\ \{\ \ldots\ meth\ \ldots\ \}}{P;E \vdash meth \in c}$$

[METHOD INHERITED]

$$\frac{\begin{array}{c} P;E \vdash \texttt{class}\ c\ \texttt{extends}\ c'\ \ldots \\ P;E \vdash meth \in c' \end{array}}{P;E \vdash meth \in c}$$

$\boxed{P;E \vdash_{\text{final}} e : t}$

[FINAL VAR]

$$\frac{\begin{array}{c} P;E \vdash wf \\ E = E_1, t\ x, E_2 \end{array}}{P;E \vdash_{\text{final}} x : t}$$

[FINAL REF]

$$\frac{\begin{array}{c} P;E \vdash_{\text{final}} e : c \\ P;E \vdash (\texttt{final}\ t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e') \in c \end{array}}{P;E \vdash_{\text{final}} e.fd : t}$$

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

[LOCK SET]

$$\frac{\begin{array}{c} P;E \vdash wf \\ \forall l \in ls.\ \exists c.\ P;E \vdash_{\text{final}} l : c \end{array}}{P;E \vdash ls}$$

$\boxed{P;E \vdash l \in ls}$

[LOCK SET ELEM]

$$\frac{l \in ls \qquad P;E \vdash ls}{P;E \vdash l \in ls}$$

$\boxed{P;E \vdash ls_1 \subseteq ls_2}$

[LOCK SET SUBSET]

$$\frac{P;E \vdash ls_1 \qquad P;E \vdash ls_2 \qquad ls_1 \subseteq ls_2}{P;E \vdash ls_1 \subseteq ls_2}$$

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

[EXP SUB]

$$\frac{P;E;ls \vdash e : s \qquad P;E \vdash s <: t}{P;E;ls \vdash e : t}$$

[EXP NEW]

$$\frac{P;E \vdash ls \qquad P;E \vdash c}{P;E;ls \vdash \texttt{new}\ c : c}$$

[EXP VAR]

$$\frac{P;E \vdash ls \qquad E = E_1, t\ x, E_2}{P;E;ls \vdash x : t}$$

[EXP REF]

$$\frac{\begin{array}{c} P;E;ls \vdash e : c \\ P;E \vdash ([\texttt{final}]_{\text{opt}}\ t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e') \in c \\ P;E \vdash [e/\texttt{this}]l \in ls \\ P;E \vdash [e/\texttt{this}]t \end{array}}{P;E;ls \vdash e.fd : [e/\texttt{this}]t}$$

[EXP ASSIGN]

$$\frac{\begin{array}{c} P;E;ls \vdash e : c \\ P;E \vdash (t\ fd\ \texttt{guarded\_by}\ l\ \texttt{=}\ e'') \in c \\ P;E \vdash [e/\texttt{this}]l \in ls \\ P;E;ls \vdash e' : [e/\texttt{this}]t \end{array}}{P;E;ls \vdash e.fd\ \texttt{=}\ e' : [e/\texttt{this}]t}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P;E;ls_1 \vdash e : c \\ P;E \vdash (t\ mn(s_j\ y_j\ ^{j \in 1\ldots n})\ \texttt{requires}\ ls_2\ \{\ e'\ \}) \in c \\ P;E;ls_1 \vdash e_j : [e/\texttt{this}]s_j \\ P;E \vdash [e/\texttt{this}]ls_2 \subseteq ls_1 \\ P;E \vdash [e/\texttt{this}]t \end{array}}{P;E;ls_1 \vdash e.mn(e_{1\ldots n}) : [e/\texttt{this}]t}$$

[EXP LET]

$$\frac{\begin{array}{c} P;E;ls \vdash e_1 : t \\ P;E,t\ x;ls \vdash e_2 : s \\ P;E \vdash [e_1/x]s \end{array}}{P;E;ls \vdash \texttt{let}\ t\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : [e_1/x]s}$$

[EXP SYNC]

$$\frac{P;E \vdash_{\text{final}} e_1 : c \qquad P;E;ls \cup \{e_1\} \vdash e_2 : t}{P;E;ls \vdash \texttt{synchronized}\ e_1\ \texttt{in}\ e_2 : t}$$

[EXP FORK]

$$\frac{P;E \vdash ls \qquad P;E;\emptyset \vdash e : t}{P;E;ls \vdash \texttt{fork}\ e : \texttt{int}}$$

B Parameterized Classes

This section extends the type system with classes parameterized by lock expressions. We extend typing environments to include ghost variables:

$$E \quad ::= \quad \emptyset \ \mid\ E, arg \ \mid\ E, garg$$

An instantiated class definition has the form:

$$ci ::= \texttt{class}\ c\ \texttt{extends}\ c\ \{\ field^*\ meth^*\ \}$$

There is one new judgment form.

Judgment	Meaning
$P;E \vdash ci$	ci is a valid instantiated class definition for P

We redefine well-typed classes to include ghost parameters, and we also introduce new rules for constructing environments and instantiating parameterized classes. Rules that have subscripts in their names, such as [CLASS_2], replace earlier rules of the same name. Rules that do not have subscripts are used in addition to the previous rules.

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

[CLASS$_2$]

$$\frac{\begin{array}{c} P; \emptyset \vdash t_i \\ \mathit{garg}_i = \texttt{ghost}\ t_i\ x_i \\ E = \mathit{garg}_{1\ldots n}, \mathit{cn}\texttt{<}x_{1\ldots n}\texttt{>}\ \texttt{this} \\ P; E \vdash c \\ P; E \vdash \mathit{field}_i \qquad P; E \vdash \mathit{meth}_i \end{array}}{\begin{array}{c} P \vdash \texttt{class}\ \mathit{cn}\texttt{<}\mathit{garg}_{1\ldots n}\texttt{>}\ \texttt{extends}\ c \\ \{\ \mathit{field}_{1\ldots j}\ \ \mathit{meth}_{1\ldots k} \} \end{array}}$$

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

[ENV GHOST]

$$\frac{\begin{array}{c} P; E \vdash t \\ x \notin \mathit{Dom}(E) \end{array}}{P; E, \texttt{ghost}\ t\ x \vdash \mathit{wf}}$$

$\boxed{P; E \vdash_{\text{final}} e : t}$

[FINAL VAR$_2$]

$$\frac{\begin{array}{c} P; E \vdash \mathit{wf} \\ E = E_1, [\texttt{ghost}]_{\text{opt}}\ t\ x, E_2 \end{array}}{P; E \vdash_{\text{final}} x : t}$$

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

[CLASS INSTANTIATION]

$$\frac{\begin{array}{c} \texttt{class}\ \mathit{cn}\texttt{<}\texttt{ghost}\ t_i\ x_i{}^{i \in 1\ldots n}\texttt{>}\ \mathit{body} \in P \\ P; E \vdash_{\text{final}} l_i : s_i \qquad P; E \vdash s_i <: t_i \end{array}}{P; E \vdash \texttt{class}\ \mathit{cn}\texttt{<}l_{1\ldots n}\texttt{>}\ [l_i/x_i{}^{i \in 1\ldots n}]\mathit{body}}$$

C Thread-Local Classes

This section extends the type system with thread-local classes. The following judgments are added to the system:

Judgment	Meaning
$P \vdash t\ \mathit{shared}$	values of type t can be shared between threads
$P \vdash \mathit{field}\ \mathit{shareable}$	*field* has a thread-shared type and is guarded by a lock or is final
$P; E; c \vdash \mathit{meth}$	*meth* does not override a method from any thread-shared super type of c

The typing rules for these judgments are presented below. Rules [EXP REF] and [EXP ASSIGN] must also be updated to support unguarded fields.

$\boxed{P \vdash t\ \mathit{shared}}$

[CLASS SHARED]

$$\frac{\texttt{class}\ \mathit{cn}\texttt{<}\mathit{garg}_{1\ldots n}\texttt{>}\ \texttt{extends}\ c\ \{\ \mathit{field}_{1\ldots j}\ \ \mathit{meth}_{1\ldots k} \} \in P}{P \vdash \mathit{cn}\texttt{<}x_{1\ldots n}\texttt{>}\ \mathit{shared}}$$

[OBJECT SHARED]

$$\frac{}{P \vdash \texttt{Object}\ \mathit{shared}}$$

[INT SHARED]

$$\frac{}{P \vdash \texttt{int}\ \mathit{shared}}$$

$\boxed{P \vdash \mathit{field}\ \mathit{shareable}}$

[SHAREABLE GUARDED FIELD]

$$\frac{P \vdash t\ \mathit{shared}}{P \vdash [\texttt{final}]_{\text{opt}}\ t\ \mathit{fd}\ \texttt{guarded\_by}\ l\ \texttt{=}\ e\ \mathit{shareable}}$$

[SHAREABLE FINAL FIELD]

$$\frac{P \vdash t\ \mathit{shared}}{P \vdash \texttt{final}\ t\ \mathit{fd}\ \texttt{=}\ e\ \mathit{shareable}}$$

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

[CLASS$_3$]

$$\frac{\begin{array}{c} P; \emptyset \vdash t_i \\ \mathit{garg}_i = \texttt{ghost}\ t_i\ x_i \\ E = \mathit{garg}_{1\ldots n}, \mathit{cn}\texttt{<}x_{1\ldots n}\texttt{>}\ \texttt{this} \\ P; E \vdash c \\ P; E \vdash \mathit{field}_i \qquad P; E \vdash \mathit{meth}_i \\ P \vdash c\ \mathit{shared} \qquad P \vdash \mathit{field}_i\ \mathit{shareable} \end{array}}{\begin{array}{c} P \vdash \texttt{class}\ \mathit{cn}\texttt{<}\mathit{garg}_{1\ldots n}\texttt{>} \\ \texttt{extends}\ c\ \{\ \mathit{field}_{1\ldots j}\ \ \mathit{meth}_{1\ldots k} \} \end{array}}$$

[LOCAL CLASS]

$$\frac{\begin{array}{c} P; \emptyset \vdash t_i \\ \mathit{garg}_i = \texttt{ghost}\ t_i\ x_i \\ E = \mathit{garg}_{1\ldots n}, \mathit{cn}\texttt{<}x_{1\ldots n}\texttt{>}\ \texttt{this} \\ P; E \vdash c \\ P; E \vdash \mathit{field}_i \qquad P; E \vdash \mathit{meth}_i \\ P; E; c \vdash \mathit{meth}_i \end{array}}{\begin{array}{c} P \vdash \texttt{thread\_local class}\ \mathit{cn}\texttt{<}\mathit{garg}_{1\ldots n}\texttt{>} \\ \texttt{extends}\ c\ \{\ \mathit{field}_{1\ldots j}\ \ \mathit{meth}_{1\ldots k} \} \end{array}}$$

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

[OVERRIDE OK]

$$\frac{\forall c'.\ \left[\begin{array}{l} P; E \vdash c <: c' \\ \wedge\ P; E \vdash (\ldots\ \mathit{mn}(\ \ldots\ )\ \texttt{requires}\ \ldots) \in c' \end{array}\right] \Rightarrow P \nvdash c'\ \mathit{shared}}{P; E; c \vdash t\ \mathit{mn}(\mathit{arg}_{1\ldots n})\ \texttt{requires}\ \mathit{ls}\ \{\ e\ \}}$$

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

[LOCAL CLASS INSTANTIATION]

$$\frac{\begin{array}{c} \texttt{thread\_local class}\ \mathit{cn}\texttt{<}\texttt{ghost}\ t_i\ x_i{}^{i \in 1\ldots n}\texttt{>}\ \mathit{body} \in P \\ P; E \vdash_{\text{final}} l_i : s_i \qquad P; E \vdash s_i <: t_i \end{array}}{P; E \vdash \texttt{class}\ \mathit{cn}\texttt{<}l_{1\ldots n}\texttt{>}\ [l_i/x_i{}^{i \in 1\ldots n}]\mathit{body}}$$

$\boxed{P; E; \mathit{ls} \vdash e : t}$

[EXP FORK$_2$]

$$\frac{\begin{array}{c} P; E; \emptyset \vdash e : t \\ \forall x \in \mathrm{FV}(e).\ P \vdash E(x)\ \mathit{shared} \end{array}}{P; E; \mathit{ls} \vdash \texttt{fork}\ e : \texttt{int}}$$