Types for Safe Locking: Static Race Detection for Java

MARTIN ABADI and CORMAC FLANAGAN
University of California at Santa Cruz
and
STEPHEN N. FREUND
Williams College

This paper presents a static race-detection analysis for multithreaded shared-memory programs, focusing on the Java programming language. The analysis is based on a type system that captures many common synchronization patterns. It supports classes with internal synchronization, classes that require client-side synchronization, and thread-local classes. In order to demonstrate the effectiveness of the type system, we have implemented it in a checker and applied it to over 40,000 lines of hand-annotated Java code. We found a number of race conditions in the standard Java libraries and other test programs. The checker required fewer than 20 additional type annotations per 1,000 lines of code. This paper also describes two improvements that facilitate checking much larger programs: an algorithm for annotation inference and a user interface that clarifies warnings generated by the checker. These extensions have enabled us to use the checker for identifying race conditions in large-scale software systems with up to 500,000 lines of code.

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

General Terms: Languages, Verification, Reliability

Additional Key Words and Phrases: concurrent programs, race conditions, type system, type inference

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. Although some race conditions are benign and do not affect program correctness, in many cases race conditions result in program crashes, incorrect results, and other unintended program behavior. Race conditions

Martín Abadi's and Cormac Flanagan's address is Department of Computer Science, University of California at Santa Cruz, Santa Cruz, CA 95064. Stephen Freund's address is Department of Computer Science, Williams College, Williamstown, MA 01267.

can be avoided by careful programming discipline: protecting each data structure with a lock and acquiring that lock before manipulating the data structure [Birrell 1989]. Since a lock can be held by at most one thread at any time, 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 inadvertently neglects to perform 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 run-time testing techniques. A single synchronization error in an otherwise correct program may yield a race condition whose cause takes weeks to identify [Savage et al. 1997].

This paper investigates a static analysis system for detecting race conditions in Java programs. The analysis supports the lock-based synchronization discipline by associating a lock with each shared field and by verifying that the appropriate lock is held whenever a shared field is accessed (that is, read or updated). We express the reasoning and checks performed by this analysis as an extension of Java's type system. We start by presenting an initial type system that suffices to verify that some example programs in a core subset of Java are free of race conditions. In order to accommodate larger, more realistic, multithreaded programs, we extend the initial type system with a number of additional features. These features include:

—classes parameterized by locks, which allow the fields of a class to be protected by some lock external to the class;

—the notion of objects that are local to a particular thread and therefore safely accessible without synchronization; and

—mechanisms for escaping from the type system in places where it proves too restrictive, or where a particular race condition is considered benign.

In order to evaluate the utility of the resulting type system, we have implemented a typechecker 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 scripting language WebL; and Ambit, a mobile-ambient calculus implementation.

Checking these programs using our type system requires adding some type annotations. Typically, fewer than 20 annotations are required per 1,000 lines of code. Most of the annotations were inserted in response to feedback from the checker. The annotation process proceeded at a rate of roughly 1,000 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 the class `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.

Despite these successes, the annotation cost of our initial checker prevented its adoption for large-scale software development projects, particularly when a sizable code base already exists. The second half of this paper extends the race-condition checker with two key features:

—an annotation-inference algorithm that automatically computes annotations, and

—a user interface that reduces the burden of inspecting the warnings generated by the checker.

The annotation inference has enabled us to analyze programs with up to 500,000 lines of code, and the user interface helped us in identifying race conditions in those 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 type system, plus extensions for classes parameterized by locks and thread-local classes. Section 4 describes our prototype implementation, including the escape mechanisms. Section 5 discusses our experiences checking several Java programs. In Section 6, we turn our attention to issues of scale and extensions to check large programs, and present our annotation-inference algorithm and user interface. Section 7 describes our experiences applying these tools to large examples. We relate this work to other projects in Section 8, and we conclude in Section 9. The Appendix contains a formal definition of the type system and proofs.

This paper is based on work presented in preliminary form at conferences and workshops [Flanagan and Abadi 1999b; 1999a; Flanagan and Freund 2000; 2001]. Some of that work focused on lambda calculi and object calculi (rather than Java), and some relied on type constructions not considered here (for example, existential types). It also treated deadlocks, which, like race conditions, are infamously common in multithreaded programming.

2. A MULTITHREADED SUBSET OF JAVA

This section introduces ConcurrentJava, a small multithreaded subset of Java. This language is derived from ClassicJava [Flatt et al. 1998], a sequential subset of Java, and we adopt much of the type structure and semantics of ClassicJava.

2.1 Syntax and Informal Semantics

The syntax of ConcurrentJava is shown in Figure 1. A program is a sequence of class declarations together with an initial expression, which is the starting point for the program's execution. Each class declaration associates a class name with a body that consists of a super class, a sequence of field declarations, and a sequence of method declarations. The self-reference variable "`this`" is implicitly bound within these field and method declarations.

A field declaration includes an initializer expression and an optional `final` modifier; if this modifier is present, then the field cannot be updated after initialization. We use "$[X]_{\mathrm{opt}}$" in grammars to denote either "X" or the empty string. A field initializer may be an integer, `null`, or a variable. 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 ones for object allocation, field read and update, method invocation, and variable binding and reference. ConcurrentJava also

P	$::=$	$\mathit{defn}^*\ e$	(program)
defn	$::=$	$\texttt{class}\ \mathit{cn}\ \mathit{body}$	(class declaration)
body	$::=$	$\texttt{extends}\ c\ \texttt{\{}\ \mathit{field}^*\ \mathit{meth}^*\texttt{\}}$	(class body)
field	$::=$	$[\texttt{final}]_{\mathrm{opt}}\ t\ \mathit{fn}\ \texttt{=}\ v$	(field declaration)
meth	$::=$	$t\ \mathit{mn}\texttt{(}\mathit{arg}^*\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}$	(method declaration)
arg	$::=$	$t\ x$	(variable declaration)
s, t	$::=$	c	(type)
	$\mid$	$\texttt{int}$	
c	$::=$	cn	(class type)
	$\mid$	$\texttt{Object}$	
e, f	$::=$	v	(value)
	$\mid$	$\texttt{new}\ c\texttt{()}$	(allocate)
	$\mid$	$e.\mathit{fn}$	(field read)
	$\mid$	$e.\mathit{fn}\ \texttt{=}\ e$	(field update)
	$\mid$	$e.\mathit{mn}\texttt{(}e^*\texttt{)}$	(method call)
	$\mid$	$\texttt{let}\ \mathit{arg}\ \texttt{=}\ e\ \texttt{in}\ e$	(variable binding)
	$\mid$	$\texttt{synchronized}\ e\ e$	(synchronization)
	$\mid$	$e.\texttt{fork}$	(fork)
v	$::=$	x	(variable)
	$\mid$	n	(integer)
	$\mid$	$\texttt{null}$	(null reference)
cn	$\in$	class names	
fn	$\in$	field names	
mn	$\in$	method names	
x, y	$\in$	variable names	
n	$\in$	integers	

Fig. 1. The grammar for CONCURRENTJAVA.

supports multithreaded programs by including the operation $e.\texttt{fork}$. Here, e should evaluate to an object that includes a nullary method `run`. The fork operation spawns a new thread that calls that `run` method. This evaluation is performed only for its effect; the result of the method call 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 $\texttt{synchronized}\ e_1\ e_2$ is evaluated in a manner similar to Java's `synchronized` statement: the subexpression e_1 is evaluated first, and should yield an object, whose lock is then acquired; the subexpression e_2 is then evaluated; and finally the lock is released. The result of e_2 is returned as the result of the synchronized expression. While evaluating e_2, the current thread is said to hold the lock. Any other thread that attempts to acquire the lock blocks until the lock is released. A newly forked thread does not inherit locks held by its parent thread.

We present example programs in an extended language with integer operations and additional control structures such as conditional and loop statements, and we permit fields to be initialized with freshly created objects. Adding these extensions to the formal language would not introduce any major technical challenges.

We use $e_1; e_2$ to abbreviate $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$, where x does not occur free in e_2. We sometimes enclose expressions in parentheses or braces, and sometimes add

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

class DepositThread {
  final Account a = new Account();
  int run() {
    a.deposit(10);
  }
}

let DepositThread t = new DepositThread() in {
  t.fork;
  t.fork;
}
```

Fig. 2. Multithreaded bank-account program.

semicolons for clarity. We use the notation $[e_1/x_1, \ldots, e_n/x_n]e$ to denote the parallel capture-free substitution of e_i for all free occurrences of x_i within e, for each $i \in 1..n$.

ConcurrentJava is simpler than Java in a number of respects. For example, ConcurrentJava includes neither arrays nor the complex object initialization of Java. We deal with the full Java language starting in Section 4.

2.2 Locks Against Races

Multithreaded ConcurrentJava programs are prone to race conditions, as illustrated by the program in Figure 2, which allocates a new bank account and makes two deposits into the account in parallel. The program contains a race condition: two threads may attempt to manipulate the field `balance` simultaneously. In particular, suppose that two executions of `deposit` are interleaved as follows:

Thread 1	Thread 2
`int temp = this.balance;`	
	`int temp = this.balance;`
	`temp = temp + x;`
`temp = temp + x;`	
`this.balance = temp;`	
	`this.balance = temp;`

The final value of `balance` reflects only one of the two deposits made to the account. This behavior is not the intended one. Thus, the race condition leads to incorrect results.

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

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

The modified account implementation is race-free. It 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 of Section 2.2. We now present a type system that supports this programming discipline.

3.1 RaceFreeJava

In order to enforce lock-based synchronization, the type system requires that each field have an associated lock that is held whenever the field is read or updated. For this purpose, the type system:

—associates a lock with each field declaration, and

—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 lock l protects that field; the type system then verifies that whenever a thread accesses a field, that thread holds the field's lock. 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 uses this assumption when checking that the method body is race-free. In essence, we are using an effect system [Jouvelot and Gifford 1991] to determine statically where locks are held. We extend the syntax of field and method declarations to include these type annotations.

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

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* whose value does not change during program execution. For RaceFreeJava, a final expression is simply a reference to an immutable variable. When considering the full Java language in Section 4, we will generalize the set of final expressions. The type system also requires that the lock expressions in a `requires` clause be final for similar reasons.

Since field and method types now include final expressions, our type system supports a specialized notion of dependent types. In a previous paper [Flanagan and Abadi 1999b], we explored an alternative approach that avoids these dependent types, and instead uses singleton types (types with one single element) to enable the tracking of locks. These singleton types can be hidden when necessary using existential types. For example, if an object type A contains a lock with singleton lock type t_1, then we can hide this singleton type using $\exists t_1.A$.

However, the singleton-type approach is syntactically cumbersome, since it requires introducing two names for each lock; one name for the lock itself and a second name for the corresponding singleton type, like the type t_1 above. Moreover, the singleton type approach does not easily support the flexible subtyping relations needed in object-oriented languages. For example, suppose A has a subtype $B <: A$ that, in addition to the lock of type t_1, contains a second lock of type t_2. Then, after hiding these singleton types, the standard rules for existential types do not yield the desired subtyping relation:

$$(\exists t_1.\exists t_2.B) \not<: (\exists t_1.A)$$

Because of these two limitations of the singleton-type approach, in RaceFreeJava we chose to adopt specialized dependent types instead.

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

$$P; E; ls \vdash e : t$$

Here, P (the program being checked) is included in the judgment in order to provide information about class declarations in the program; E is an environment that provides types for the free variables of the expression e; ls is a set of final expressions that describe the locks that are held when the expression e is evaluated; and t is the inferred type of e. Thus, the type rules track the set of locks held at each program point.

Most of the type rules are straightforward and similar to those of ClassicJava. The complete set of judgments and rules, as well as a proof of soundness for the rules, is contained in the Appendix. Here we briefly explain some of the crucial rules.

—The rule [exp sync] for `synchronized` e_1 e_2 checks that e_1 is a final expression of some class type c, and checks e_2 with an extended lock set that includes e_1, since the lock e_1 is held when e_2 is evaluated.

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

The judgment $P; E \vdash_{\text{final}} e_1 : c$ in the hypotheses means that e_1 is a final expression of type c.

—The rule [EXP REF GUARDED] for $e.fn$ checks that e is a well-typed expression of some class type c, that c has a field fn of type t, guarded by lock l, and that l is held.

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fn : [e/\texttt{this}]t} \quad \text{[EXP REF GUARDED]}$$

The judgment $P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c$ in the hypotheses means that c declares or inherits a field fn of type t, guarded by l, where the self-reference variable "this" provides a means for the field to refer to its containing object. In the source syntax, self-reference bindings were implicit. However, in order to support α-renaming and avoid name collisions, this judgment makes the self-reference binding explicit via the construct "$\varsigma\texttt{this}.\ (\cdots)$". Note that, apart from being implicitly bound within field and method declarations in the source language, the variable "this" is otherwise a regular variable. In particular, we could rewrite the rule above using a different variable name instead of "this", but for clarity we follow Java's convention of using "this" as the self-reference variable.

In order to ensure that the protecting lock l is held, it suffices to establish that l denotes the same lock as some expression l' in the current lock set. In general, proving that l and l' denote the same lock may be undecidable. Therefore, we rely on a conservative approximation. In a first attempt to define such an approximation, one may simply check that l and l' are syntactically identical. However, requiring this syntactic identity is too restrictive in many cases. In particular, occurrences of `this` in the lock expression l refer to the object e being dereferenced. To account for the aliasing of `this` and e, the rule replaces all occurrences of `this` in l by e. Accordingly, the antecedent $P; E \vdash [e/\texttt{this}]l \in ls$ ensures that $[e/\texttt{this}]l$ is in the current lock set. This conservative approximation has been sufficient for most programs we have inspected.

A similar aliasing situation arises in the next section, where we introduce types that contain lock expressions. In order to accommodate this future extension, we include the appropriate substitutions for types here, and give $[e/\texttt{this}]t$ as the type of the field read. The antecedent $P; E \vdash [e/\texttt{this}]t$ ensures that $[e/\texttt{this}]t$ is a well-formed type.

—The rule [EXP ASSIGN] for $e.fn = e'$ ensures that the appropriate lock is held whenever a field is updated.

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

—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.

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

The judgment $P; E \vdash \varsigma\texttt{this}.\,(t\ mn(s_1\ y_1, \ldots, s_n\ y_n)\ \texttt{requires}\ ls_2\ \texttt{\{}\ e'\ \texttt{\}}) \in c$ in the hypotheses means that c has a method of the expected form, where the self-reference variable "`this`" provides a means for the method to refer to the containing object. The judgment $P; E \vdash [e/\texttt{this}]ls_2 \subseteq ls_1$ simply expresses set inclusion for lock sets.

—The rule [EXP FORK] for $e.\texttt{fork}$ checks that e refers to an object with a `run` method:

$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash \varsigma\texttt{this}.\,(t\ \texttt{run()}\ \texttt{requires}\ \emptyset\ \texttt{\{}\ e'\ \texttt{\}}) \in c \end{array}}{P; E; ls \vdash e.\texttt{fork} : \texttt{int}}\ [\text{EXP FORK}]$$

The fork expression always evaluates to the dummy result zero and has type `int`.

Race-Free Bank Accounts. We can use this type system to verify that the synchronized bank-account implementation is race-free by adding a type annotation that expresses 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`. The following class declaration replaces the corresponding one of Figure 2:

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

An alternative implementation of the bank account may rely on its clients to perform the necessary synchronization operations, as shown in Figure 3. In this example, the method signature

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

declares that the object's lock must be acquired before calls to `deposit`. Since the necessary lock is indeed held at each call-site, this program is well-typed and race-free.

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

class DepositThread {
  final Account a = new Account();
  int run() {
    synchronized (a) {
      a.deposit(10);
    }
  }
}
```

Fig. 3. Bank-account implementation with client-side locking.

3.2 External Locks

The type system of the previous section can verify the absence of race conditions in a number of interesting examples. However, larger programs frequently use additional synchronization patterns that cannot be captured by that system. In order to accommodate such programs, this section extends the RaceFreeJava type system with classes parameterized by locks, and Section 3.3 further extends the type system with thread-local classes.

In the type system of the previous section, each field of an object can be protected only by the object itself or by some field of the object. In some cases, however, we would like to allow an external lock to protect a field. For example, all of the fields in a linked list might be protected by some lock external to the list.

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

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

A class declaration 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 and they can appear only in type annotations and not in regular code. In particular, to preserve compatibility with existing Java compilers, they do not affect run-time program behavior. 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.

Checking of parameterized classes is handled via substitution. If

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

is a well-formed class declaration, then for any final expressions $l_1, \ldots, l_n$ of the appropriate types $t_1, \ldots, t_n$, the type $cn\langle l_1, \ldots, l_n \rangle$ is a valid instantiated class

```
class Node⟨ghost Dictionary d⟩ {
  String key = null guarded_by d;
  Object value = null guarded_by d;
  Node⟨d⟩ next = null guarded_by d;

  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 = null guarded_by this;

  void put(String k, Object v) {
    synchronized (this) {
      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;
        }
      }
    }
  }
  ...
}
```

Fig. 4. A synchronized dictionary.

type, with associated instantiated class declaration

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

Methods could be parameterized in a similar fashion, but we did not find parameterized methods necessary in most of the programs we studied.

Using External Locks. As an example of the use of external locks, consider the dictionary implementation of Figure 4. A dictionary maps keys to values. In our implementation, a dictionary is represented as an object that contains a linked list of `Nodes`, where each `Node` contains a key, a value, and a next pointer.

We may wish to protect the entire dictionary, including its linked list, with the lock of the dictionary. For this purpose, the class `Node` is parameterized by the enclosing dictionary; the fields of `Node` are guarded by the dictionary lock; and each method of `Node` requires that the dictionary lock be 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.

3.3 Thread-Local Classes

Large multithreaded programs typically include 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 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();
  final Queue todo = new Queue();
  void run() {
    while (true) {
      String url = todo.dequeue();
      if (!visited.add(url)) {
        let String text = loadPageText(url) in {
          let LinkEnumerator enum = new LinkEnumerator() in {
            enum.init(text);
            while (enum.hasMoreLinks()) {
              todo.enqueue(enum.nextLink());
            }
          }
        }
      }
    }
  }
  ...
}

let Crawler c = new Crawler() in {
  c.todo.enqueue("http://www.yahoo.com");
  c.fork;
  c.fork;
  c.fork;
}
```

Fig. 5. A concurrent web crawler with a thread-local enumeration class.

`thread_local` modifier on class declarations:

$$defn \ ::= \ [\texttt{thread\_local}]_{\text{opt}} \ \texttt{class} \ cn\langle garg^{*}\rangle \ body$$

We also modify the typing rules for fields so that thread-local classes can have non-final, unguarded fields.

An example of a thread-local class appears in Figure 5. The class `Crawler` defines a concurrent web crawler that forks new threads to process pages stored in a shared queue. Instances of the `LinkEnumerator` class, which parses the text of the page in order to find links, are 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. The typing rule for `fork` ensures that the new thread starts by calling the `run` method of a thread-shared class. The rules for thread-shared types and `fork` appear in Appendix C.

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. In particular, this restriction applies to the implicit downcasts that occur during dynamic dispatch. To avoid such implicit downcasts, our type system requires a thread-local class does not override any methods declared in a thread-shared superclass. In the context of the full Java language, this restriction also applies to explicit cast operations. (RaceFreeJava does not contain those operations.) A consequence of preventing these downcasts is that thread-local objects cannot be stored in collection classes. This is not a problem, however, in GJ [Bracha et al. 1998] or Java 1.5 [JavaSoft 2004], both of which support parametric polymorphism.

Alternatively, we could replace these static requirements with dynamic checks: a compiler could insert code to track the allocating thread of each object and dynamically check that thread-shared to thread-local downcasts are performed only by the appropriate thread.

4. THE RCCJAVA CHECKER

We extended the RaceFreeJava type system to the full Java language [Gosling et al. 1996] and implemented the resulting system. This race-condition checker, `rccjava`, extends the type system outlined in Section 3 to accommodate the additional features of Java, including arrays, interfaces, constructors, and static fields and methods. In addition, we extended the notion of final expressions to include a field read $e.fn$, where e is a final expression and fn is a final field. The additional type information required by `rccjava` is embedded in Java comments in order 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 6 for an example. To specify the lock guarding elements in an array, we introduced an `/*# elems_guarded_by` *l* `*/` annotation.

The `rccjava` tool was built on top of an existing Java front-end that includes a scanner, parser, and typechecker. 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 typechecking, implementing syntactic equality and substitution on abstract syntax trees, and incorporating classes parameterized by locks.

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` relaxes the formal type system in several ways, and also infers default annotations for unannotated code. These features are described below.

4.1 Escape Mechanisms

The `rccjava` checker 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, as in:

```
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. Our experience has shown that, while many such overrides exist in large programs, they are typically not sources of many race conditions. Therefore, this flag suppresses some errors but also many false alarms.

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 the current statement block. 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. An object is typically initialized in the object's constructor without synchronization. This initialization pattern is sound provided that no references to the object being initialized escape from the creating thread until after the constructor exits. Violations of this assumption are unlikely, and using it eliminates a large number spurious warnings. We believe that this command-line flag could be replaced with a sound escape analysis (such as those of [Choi et al. 1999; Salcianu and Rinard 2001]) without significant reduction in the expressiveness of the system.

4.2 Default Annotations

Although not originally designed to infer type information, `rccjava` does construct default annotations for unannotated classes and fields. The heuristics used to compute default annotations are:

Program	KLOC	Programmer time (hrs)	Annotations	Races found
`java.util.Hashtable`	0.4	0.5	60	0
`java.util.Vector`	0.4	0.5	10	1
`java.io.*`	16.0	16.0	139	4
Ambit	4.5	4.0	38	0
WebL	20.0	12.0	358	5

Table I. Programs analyzed using `rccjava`.

—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.

5. EVALUATION OF RCCJAVA

In order to test the effectiveness of `rccjava` as a static race-detection tool, we used it for checking several multithreaded Java programs. Our test cases include two representative single classes, `java.util.Hashtable` and `java.util.Vector`, and several larger programs, including `java.io`, the Java input/output package (version 1.1) [JavaSoft 1998]; Ambit, an implementation of a mobile-ambient calculus [Cardelli 1997]; and an interpreter and run-time environment for WebL, a language for automating web-based tasks [Kistler and Marais 1998].

These five programs use a variety of synchronization patterns, most of which were captured easily with `rccjava` annotations. We used the command-line flags "`-no_warn thread_local_override`" and "`-constructor_holds_lock`" for these tests (see Section 4.1). Although these flags may cause `rccjava` to miss some potential race conditions, 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 I 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. We added annotations to these programs until the tool reported warnings only for the race conditions listed in the last column.

Table II breaks down the annotation count into the different categories of annotations, normalized to the frequency with which they appear in 1,000 lines of code. For the large programs, fewer than 20 annotations were required per 1,000 lines. Most of these annotations were clustered in the small number of classes that directly manipulate thread-shared data. The majority of classes typically required

Program	Annotations per KLOC						
	guarded by	requires	class param or arg	thread local/shared	no warn	holds	Total
java.util.Hashtable	29.0	2.3	79.5	0.0	11.0	11.4	136.0
java.util.Vector	11.6	0.0	0.0	0.0	2.3	9.3	23.2
java.io.*	2.9	0.8	0.0	1.8	0.7	2.4	8.7
Ambit	2.7	0.2	0.0	2.2	2.2	1.1	8.4
WebL	4.1	1.7	5.2	0.9	3.1	3.1	17.9

Table II. Number of `rccjava` annotations added to each program.

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 4. Interestingly, restructuring `Hashtable` to declare the linked list as an inner class within the scope of the protecting lock reduces the number of annotations from 60 to 25.

We discovered problematic race conditions in three of the five example programs, even though these programs were well tested and relatively mature. Of the four race conditions found in `java.io`, one was fixed in JDK version 1.2. We also found benign race conditions in all test cases; those are not counted in Table I. Races conditions are benign if they do not affect program correctness. Examples include unsynchronized increments of global performance counters, where any resulting race conditions will not cause the resulting count to be substantially incorrect, and patterns like double-checked initialization [Schmidt and Harrison 1997], among others.

Figure 6 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` w. If one thread calls `lastIndexOf(elem)` for some `elem`, that method may read `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 reads the `elementData` array based on the old value of `elementCount`. Declaring both versions of `lastIndexOf` to be synchronized removes this race condition. Sun essentially adopted this solution for JDK version 1.4 in response to our bug report and the original publication of our results [Flanagan and Freund 2000].

6. HOUDINI/RCC

We now turn our attention to checking significantly larger programs. For such programs, the burden of manually inserting type annotations and understanding error messages becomes more significant. This section introduces `Houdini/rcc`, which makes the following two contributions:

Annotation inference: In practice, reliance on programmer-supplied annotations has restricted the application of `rccjava` to small to medium-sized (about 20 KLOC) programs for which the task of writing annotations is tolerable. To achieve

```
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!!!
  }
  ...
}
```

Fig. 6. Excerpt from `java.util.Vector`.

practical analysis of large programs, we developed an annotation-inference system for `rccjava` based on the Houdini framework [Flanagan et al. 2001].

User interface: Processing and understanding `rccjava`'s output is labor-intensive, particularly for large programs with many potential race conditions. To facilitate this process, we developed a simple but effective user interface that describes the potential race conditions. The interface also clusters race conditions together according to their probable cause so that related race conditions can be dealt with as a single unit. In addition, the user interface describes the reasoning of the annotation-inference system so that the programmer can identify the cause of each warning.

The rest of this section covers these two items in more detail, and the next section describes our experience using `Houdini/rcc` to catch race conditions in several large test programs.

6.1 Annotation Inference

`Houdini/rcc` infers annotations using the following algorithm.

> generate candidate annotation set;
> **repeat**
>   invoke `rccjava` to refute annotations;
>   remove the refuted annotations
> **until** quiescence

The simplicity of this algorithm is due to its reuse of `rccjava` to reason about the correctness of particular annotations.

The first step in the algorithm is to generate a finite set of *candidates annotations*. Each candidate annotation is a conjectured property of the locking discipline used by the program. For each class, the candidate annotation set includes an annotation `thread_local` conjecturing that all instances of that class are local to a particular thread. In addition, Houdini/rcc conjectures that each non-final field is guarded by a number of different candidate locks. The candidate locks include this and any final field declared in the same class or a superclass. We extend the typing rules to permit field declarations prefixed by multiple guarded_by annotations. Similarly, Houdini/rcc conjectures that each of these candidate locks are held on entry to each routine. Houdini/rcc does not conjecture requires clauses for methods that are called by the Java run-time systems with no lock held: such methods include main, the entry point of the program, and run, the entry point of a particular thread. Houdini/rcc also does not conjecture type parameter annotations.

Many candidate annotations will of course be incorrect. To identify incorrect annotations, the Houdini algorithm invokes rccjava on the annotated program. Like any invocation of rccjava, this invocation produces warnings about violations of the given annotations. Houdini/rcc interprets such warnings as identifying incorrect annotation guesses in the candidate set. In this sense, each invocation of rccjava has the effect of refuting some number of candidate annotations, and these annotations are then removed from the program.

Since removing one annotation may cause other annotations to become invalid, this check-and-refute cycle iterates until a fixed point is reached. At that point, all incorrect annotations have been removed from the program. The set of remaining annotations is a correct subset of the candidate set, and is in fact, the unique maximal such set [Flanagan et al. 2001].

We illustrate this annotation-inference process for a small example program that includes the Account class. This program, together with the candidate annotations conjectured by Houdini/rcc, is shown in Figure 7. Candidate annotations refuted by the Houdini/rcc algorithm are underlined. The new class Add100 is a subclass of java.lang.Thread. Therefore, invoking the start method of an Add100 object causes the object's run method to be executed in the object's thread. The Add100.main method spawns two new threads, both of which will add 100 to the balance of an Account object.

The Houdini/rcc algorithm performs four iterations through the loop:

(1) On the first iteration, rccjava refutes three annotations. The annotation thread_local on the class Add100 is refuted since each instance of this class is also an instance of its superclass java.lang.Thread. Any object of this class can be accessed by two threads: both itself and the parent thread that started it. In addition, rccjava sees that the run method calls deposit with no locks held, and hence refutes the two requires annotations on deposit.

(2) On the next iteration, rccjava sees that the (now thread-shared) class Add100 contains a reference to Account, and rccjava therefore refutes the annotation thread_local on Account. In addition, rccjava refutes the annotation requires this on update, since it is called from deposit when the lock this is not held.

(3) On the third iteration, rccjava refutes the annotation guarded_by this on the field balance, since it is accessed from update when the lock this is not held.

(4) On the fourth iteration, no further annotation are refuted, so the remaining annotations are all correct and quiescence is reached.

```
/*# thread_local */
class Account {
  final Object lock = new Object();

  /*# guarded_by lock */
  /*# guarded_by this */
  int balance = 0;

  /*# requires lock */
  /*# requires this */
  void update(int n) { balance = n; }

  /*# requires lock */
  /*# requires this */
  void deposit(int x) {
    synchronized(lock) {
      update(balance + x);
    }
  }
}
```

```
/*# thread_local */
class Add100 extends java.lang.Thread {
  final Account a;

  Add100(Account a) { this.a = a; }

  public void run() { a.deposit(100); }

  static public void main(String st[]) {
    Account a = new Account();
    (new Add100(a)).start();
    (new Add100(a)).start();
  }
}
```

Fig. 7. The `Houdini/rcc` candidate annotations for `Account`. The underlined annotations are refuted by the `Houdini/rcc` algorithm.

The `rccjava` checker is invoked one last time on the now-annotated program to determine the set of warnings to be reported to the user. For this example, the final run produces no warnings, indicating that the program is indeed free of race conditions. If, on the other hand, there was a race condition on a particular field, `Houdini/rcc` would refute all of the conjectured guarding locks on that field. Thus, the final run of `rccjava` would produce a warning that there is no lock guarding that field.

6.2 Extensions

While the basic `Houdini/rcc` algorithm can detect race conditions in unannotated programs, it produces many false alarms. This section describes two extensions to `Houdini/rcc` that help eliminate false alarms.

Read-Only Data Inference. Constant fields can be read safely by multiple threads without synchronization. Ideally, such fields would be declared as `final`, in which case `rccjava` would not warn about unsynchronized accesses. However, our experiments with `Houdini/rcc` indicated that a large number of shared constant fields are not declared as `final`, because of either programmer oversight or the use of initialization patterns that violate the restrictions on final fields.

To avoid false alarms in these cases, we have extended `rccjava` to allow a field to be annotated with the annotation `readonly`. This annotation behaves much like Java's `final` annotation, except that `readonly` is an `rccjava` annotation that can be inferred by `Houdini/rcc` in a preliminary pass.

Moreover, since `readonly` reference fields are constant values, they can be included in the set of candidate locks for a class, thus increasing the set of candidate annotations conjectured by `Houdini/rcc`.

```
class BadAccount {
  final Object lock = new Object();

  int balance = 0;

  void update(int n) { balance = n; }

  void deposit(int x) {
      update(balance + x);
  }
}
```

```
class Add100 extends java.lang.Thread {
  final BadAccount a;

  Add100(BadAccount a) { this.a = a; }

  public void run() { a.deposit(100); }

  static public void main(String st[]) {
    BadAccount a = new BadAccount();
    (new Add100(a)).start();
    (new Add100(a)).start();
  }
}
```

Fig. 8. A version of `Account` that contains a race condition on `balance`.

Main-Lock Inference. Since static fields are accessible by all threads, `rccjava` requires that every static field be protected by a lock. However, a number of the programs we examined exhibited a common pattern whereby a static field would be accessed exclusively by the main thread, without synchronization. To accommodate this programming pattern, we extended `rccjava` with the notion of a *main lock*, that is, a lock that is implicitly held by the main thread.

To infer annotations regarding the main lock, we extended `Houdini/rcc` so that it guesses the annotation `requires MainLock` for each method and the annotation `guarded_by MainLock` for each field. The refutation loop of `Houdini/rcc` then determines which fields are accessed exclusively by the main thread, thus avoiding false alarms on such fields.

6.3 User Interface

We now turn our attention to how a programmer can identify defects using the feedback from `Houdini/rcc`. The `Houdini/rcc` interface, based on the interface of Houdini for ESC/Java [Flanagan and Leino 2001], generates the following output for an input program:

—a collection of HTML pages containing the *source code view* for each Java file analyzed, which contains information about both the valid and the invalid candidate annotations guessed by `Houdini/rcc`, and

—a root HTML page listing the warnings produced by the final call to `rccjava`, where each warning message contains a hyperlink to the source view of the code at the location of the offending program line.

As an example of the process of finding errors with this tool, consider `BadAccount`, a broken version of the `Account` class shown in Figure 8. Note that the synchronization code from `deposit` is missing, so there is a potential race condition on the field `balance`. For this program, `Houdini/rcc` generates the following warning:

```
BadAccount.java:7: field ‘BadAccount.balance’ must be guarded
                   in a thread shared class
```

Clicking on this warning would open up the source code view for line 7 of the Account class (see Figure 9). The source code view displays all of the candidate

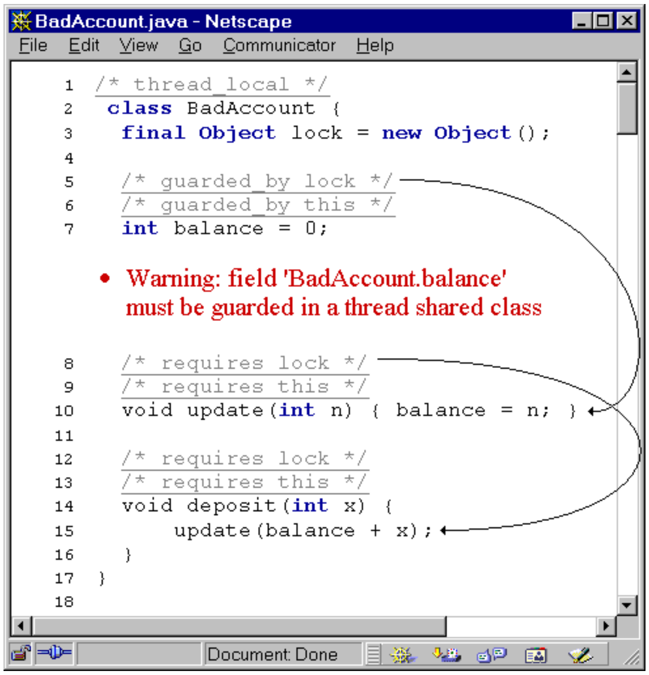

Fig. 9. Screen shot showing the `Houdini/rcc` user interface for the `BadAccount` class. The overlaid arrows indicate where two of the hyperlinks are pointing.

annotations guessed by `Houdini/rcc`. A refuted annotation is underlined, whereas a valid annotation is darkened. In this case, all annotations were refuted, but Figure 10, as described below, contains several valid annotations. `Houdini/rcc` also inserts the warning messages into the source code view.

In a situation like this, a programmer may wonder why the annotation `guarded_by lock` was not inferred for the field `balance`. Identifying the cause of `rccjava` warnings often boils down to answering such questions. To facilitate this process, each refuted annotation is a hyperlink to the line of code refuting that annotation. Figure 9 shows that the candidate annotations `guarded_by lock` for the field `balance` was refuted. Clicking on this refuted annotation brings the programmer to an access of field `balance` on line 10 where `rccjava` believes the lock `lock` is not held. To determine why the lock is not held on line 10, the programmer could click on the refuted annotation `requires lock` on line 8. This hyperlink then brings the programmer to line 15, where the required synchronization statement is missing. Surprisingly, our experience indicates that presenting the *refuted* annotations and the causes thereof is the most important aspect of the user interface.

Running `Houdini/rcc` on the correct version of `Account` produces no warnings and yields the source code view shown in Figure 10, where the inferred annotations appear in bold.

Clustering Warnings. During our experiments, we noticed several cases where `Houdini/rcc` would incorrectly characterize a thread-local class `C` as thread-shared,

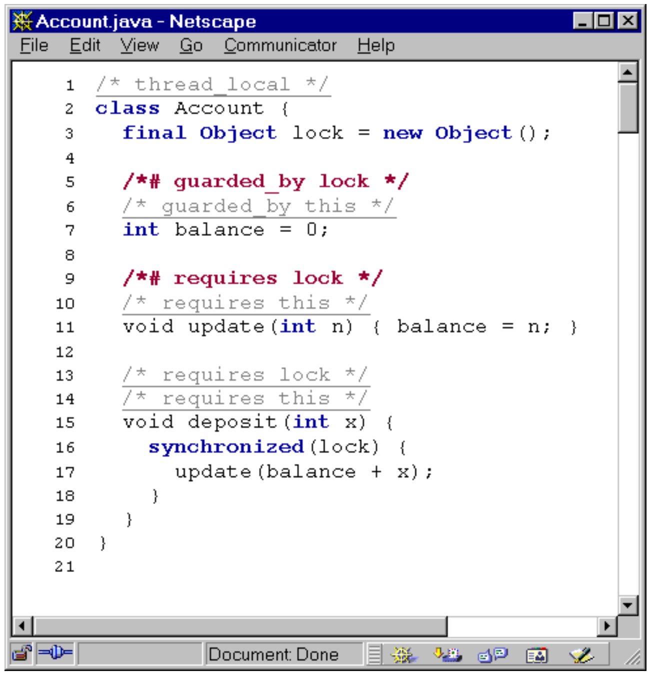

Fig. 10. Screen shot showing the `Houdini/rcc` user interface for the `Account` class.

because of conservative approximations introduced by the analysis. Unfortunately, in these cases, `Houdini/rcc` subsequently characterizes as thread-shared all classes reachable (transitively) from `C` and produces spurious warnings regarding race conditions on accesses to the fields of these classes.

To reduce this problem, we extended the user interface to group into a single cluster all of the warnings that were caused, either directly or indirectly, by `C` being characterized as thread-shared. The programmer can often deal with all the warnings in a cluster as a single unit. For example, a programmer who verifies that `C` is actually thread-local can easily ignore the entire cluster of warnings.

7. EVALUATION OF HOUDINI/RCC

We have evaluated `Houdini/rcc` on test programs with sizes that range from several thousand lines to a half million lines of code. The programs include Ambit and WebL, described in Section 5. They also include jbb2000, a Java SPEC benchmark that models a server application [Standard Performance Evaluation Corporation 2000]; TLC, a multi-threaded model checker for the TLA specification language [Yu et al. 1999]; jigsaw, a web server written in Java [World Wide Web Consortium 2001]; and orange and red, two large, proprietary Compaq systems.

Table III shows the results of running `rccjava` (without annotations) and various versions of `Houdini/rcc` on these programs. The table shows the size of each unannotated program and the number of warnings reported by running `rccjava` on that program. The column labeled `Basic Houdini/rcc` corresponds to running

Program	KLOC	Warnings per KLOC					
		`rccjava`	`Basic Houdini/rcc`	`-no_warn thread-local override`	`-constructor-holds_lock`	`-read only`	`-main lock`
Ambit	4.5	13.6	37.3	14.2	13.3	7.1	7.1
WebL	20.0	11.8	12.2	5.1	4.8	1.9	1.9
jbb2000	30.8	18.8	4.9	3.4	3.3	1.3	0.6
TLC	53.5	11.0	14.7	4.7	4.5	2.2	1.0
jigsaw	128.9	21.1	13.6	8.2	7.7	2.9	2.9
orange	28.0	17.7	33.3	14.0	13.6	6.0	3.6
red	445.0	16.6	9.0	5.2	4.8	2.2	2.2
Average		15.8	17.9	8.3	7.9	3.6	2.9

Table III. Number of warnings produced by `rccjava` and various versions of `Houdini/rcc`.

Program	KLOC	Annotations per KLOC		Time (min)	Warnings		Number of clusters	Races (found/ clusters examined)
		Candidate	Valid		per KLOC	Total		
Ambit	4.5	433	78	4	7.1	32	6	0/6
WebL	20.0	262	43	9	1.9	37	10	6/10
jbb2000	30.8	282	74	9	0.6	17	17	0/17
TLC	53.5	758	124	31	1.0	52	30	4/30
jigsaw	128.9	375	49	62	2.9	367	78	0/30
orange	28.0	863	135	74	3.6	100	84	1/84
red	445.0	358	64	286	2.2	957	340	5/70

Table IV. Statistics for `Houdini/rcc` with all flags enabled.

`Houdini/rcc` in its original form from Section 6. The four remaining columns show the number of warnings reported by `Houdini/rcc` with various extensions turned on. These extensions are the two flags (`-no_warn thread_local_override` and `-constructor_holds_lock`) described in Section 4 and the two inference modifications (`-read_only` and `-main_lock`) described in Section 6.2. Note that the additions to `Houdini/rcc` are cumulative across the columns so that, for example, the "`-constructor_holds_lock`" column reflects `Basic Houdini/rcc` run with both the "`-no_warn thread_local_override`" and "`-constructor_holds_lock`" extensions. All of the columns are normalized to show the number of warnings reported per 1,000 lines of code.

Excluding method override warnings reduced the number of warnings by roughly a factor of two across the test programs. The read-only field inference also decreased the number of warnings by another factor of two. Although the other two extensions were not as consistent in their effectiveness, there were some programs in which they also significantly reduced the number of warnings produced by `Houdini/rcc`.

Table IV shows more detailed statistics for running `Houdini/rcc` with all of the extensions described. From this table, it is clear that `Houdini/rcc` infers a nontrivial number of annotations. In general, it guesses roughly 350 candidate annotations per 1,000 lines of code, with roughly 1/4 of these annotations being valid. Our system ran in time proportional to the size of the program, processing approximately 2,000 lines per minute on a 667 MHz Alpha workstation. However, we have not optimized our tool for performance, and there are several architectural improvements that would significantly speed up the system. For instance, `Hou-`

dini/rcc makes heavy use of temporary files, and reading and writing these files accounts for a sizable fraction of its run time.

In most examples, the clustering algorithm was successful at grouping related warnings. A representative situation of this appears in jigsaw. In that program, a `DebugThread` object gathers and prints statistics about the program as it runs. This object accesses fields of a number of different objects both directly and through multiple levels of accessor methods without acquiring the necessary locks for those fields. The clustering algorithm identified the `DebugThread` class implementation as the common source of 92 such potential race conditions. All of these race conditions were deemed benign because the debugging code was intentionally designed to read data in this manner.

The last column in Table IV reflects how many non-benign race conditions we identified while studying the warnings reported by `Houdini/rcc`. Since we have not examined every cluster for the larger examples, this column shows both the number of real race conditions found and the number of warning clusters examined. For example, six race conditions were found in WebL while examining all 10 clusters. These race conditions include one that had been mistakenly considered benign when WebL was annotated by hand (hence the slight discepancy with Table I).

8. RELATED WORK

A number of tools have been developed for detecting race conditions, both statically and dynamically. We review many of them below. Several of these tools have been developed since the initial publication of our results.

Warlock [Sterling 1993] 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 race conditions. 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 race conditions 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 [Leino et al. 1999; Flanagan et al. 2002; Detlefs et al. 1998]. It uses an underlying automatic theorem prover to reason about program behavior and to verify the absence of certain kinds of errors, such as null dereferences and array-out-of-bounds errors. ESC/Java supports multithreaded programming via annotations similar to our `guarded_by` and `requires` clauses, and 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 occur only 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 [Aiken and Gay 1998] also investigate static race detection, in the context of SPMD programs. Since synchronization in these programs is performed using barriers, rather than 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 [Savage et al. 1997], rather than statically. This approach has the advantage of being applicable to unannotated programs, but it may fail to detect certain errors because of insufficient test coverage. The Eraser algorithm has been extended to handle features, such as constructors, that are common in object-oriented programming languages [von Praun and Gross 2001]. Another recent approach has combined dynamic analysis with a global static analysis to improve precision and performance [Choi et al. 2002].

Boyapati and Rinard have defined a type system much like RaceFreeJava but with a notion of object ownership [Boyapati and Rinard 2001]. Their system allows a single class declaration to yield both thread-local and thread-shared instances. Tracking thread locality on a per-object basis in this manner seems to be a promising alternative to the notion of thread-local classes presented in this paper (which forbid a class to yield both thread-local and thread-shared instances). In the system of Boyapati and Rinard, synchronization is expressed only at the object level, and not at the field level. They have extended their analysis to identify deadlocks [Boyapati et al. 2002] by using a partial ordering over the lock names manipulated by the program, much as in [Flanagan and Abadi 1999b].

Grossman has developed a type system for preventing race conditions in Cyclone, a statically safe variant of C [Grossman 2003], adapting our approach. The differences between class-based languages and Cyclone cause his system to differ from ours in several ways. Most significantly, Grossman permits functions to be polymorphic over the locks guarding the parameters of the function. In another interesting approach, Bacon *et al.* developed Guava [Bacon et al. 2001], an extension to the Java language with a form of monitor capable of sharing object state in a race-free manner.

RaceFreeJava has also been extended to verify atomicity [Flanagan and Qadeer 2003b; 2003a; Flanagan et al. 2004]. Whereas race conditions yield valuable indications of unintended interference between threads, atomicity guarantees the absence of such interference. In particular, a method is atomic if for every program execution, there is an equivalent serial execution where the actions of the method are executed contiguously, without interleaved actions of other threads. Atomic methods are amenable to sequential reasoning, which significantly simplifies subsequent (formal or informal) correctness arguments.

Vault [DeLine and Fähndrich 2001] is a system for checking resource management protocols, and is mainly focused on sequential programs. Capabilities in Vault are encoded using a combination of singleton types and existential types (much as in [Flanagan and Abadi 1999b]). While Vault has been used to reason about protocols involving mutual exclusion locks, it does not currently have a multithreaded execution model.

A variety of other approaches have been developed for race condition and deadlock prevention; these include model checking [Chamillard et al. 1996; Corbett 1996; Fajstrup et al. 1998; Yahav 2001], dataflow analysis [Dwyer and Clarke 1994], and type systems for process calculi [Kobayashi 1998; Kobayashi et al. 2000].

A number of formal calculi for Java have been presented in recent literature. These include attempts to model the entire Java language [Drossopoulou and Eisenbach 1997; Syme 1997; Nipkow and von Oheimb 1998] and, also, smaller systems designed to study specific features and extensions [Igarashi et al. 2001]. We chose to use the ClassicJava calculus of Flatt, Krishnamurthi, and Felleisen [Flatt et al. 1998] as the starting point for our study.

There have been many suggested language extensions for supporting Java classes parameterized by types [Odersky and Wadler 1997; Bracha et al. 1998; Agesen et al. 1997; Myers et al. 1997; Cartwright and Jr. 1998]. 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 [Bracha et al. 1998], in that information about class parameters is not preserved at run time.

The `requires` annotations used by `rccjava` are similar to effects [Jouvelot and Gifford 1991; Lucassen and Gifford 1988; Nielson 1996]. Thus, the analysis performed by `Houdini/rcc` includes a basic form of effect reconstruction [Tofte and Talpin 1994; 1997; Amtoft et al. 1997; Talpin and Jouvelot 1992], and the `Houdini/rcc` interface provides an explanation of why certain effects were inferred.

The `Houdini/rcc` algorithm can be viewed as an abstract interpretation [Cousot and Cousot 1977], where the abstract state space is the powerset lattice over the candidate annotations and `rccjava` is used to compute the abstract transition relation. `Houdini/rcc` may also be seen as a variant of predicate abstraction [Graf and Saidi 1997] in which each candidate annotation corresponds to a predicate. The Houdini algorithm finds the largest conjunction of these predicates that holds in all reachable states.

`Houdini/rcc` infers thread-local annotations for classes whose instances are never shared between threads. Other work on this escape-analysis problem [Choi et al. 1999; Blanchet 1999; Bogda and Hölzle 1999; Whaley and Rinard 1999; Aldrich et al. 1999] has primarily focused on optimizing synchronization operations. Because of its intended application, `Houdini/rcc` includes an interface that provides explanations.

9. CONCLUSIONS

Type systems have proven remarkably effective for preventing errors in sequential programs. In this work, we adapt type-based analysis techniques to multithreaded programs, and focus in particular on race conditions, a common source of errors in multithreaded programs.

In comparison to both traditional testing and dynamic analysis techniques such as Eraser [Savage et al. 1997], the main benefit of our type-based analysis is that it is not limited by the test coverage concerns of dynamic approaches. In particular, our core type system is provably sound, although in practice we use unsound (but useful) extensions to this core system.

Dependent types are a key feature of our type system, since the type of a field includes its protecting lock. Many dependent type systems are undecidable, since they require checking the semantic equality of two expressions [Cardelli 1988]. Our type system approximates semantic equality by syntactic equality modulo substitution; this approximation yields a decidable type system without substantially reducing expressiveness in practice. Our type system also supports dependent or parameterized classes, which significantly extend its expressiveness.

Further extensions to the type system could help reduce the number of false alarms. For example, an escape analysis would permit safe, synchronization-free initialization of an object that is not yet thread-shared, and it would remove the need for our implementation's unsound assumption that objects are never shared before being fully constructed. Similarly, a unique pointer analysis could statically guarantee the soundness of typecasts that change the ghost parameters in an object type. However, such extensions would also result in a more complex type system, and perhaps more complex type annotations.

Providing type annotations is perhaps the dominant obstacle to the wider adoption of RaceFreeJava. The experimental results with `Houdini/rcc` are promising, but are limited by `Houdini/rcc`'s inability to infer ghost parameters. We believe that developing a cost-effective type system for race freedom requires a type inference algorithm that infers ghost parameters. This more general type inference problem is NP-complete and we are tackling this problem via reduction to propositional satisfiability [Flanagan and Freund 2004].

Acknowledgments: Thanks to John Mitchell and Scott Stoller for comments on an earlier draft of this paper, to Hannes Marais, Mike Burrows, and Mark Lillibridge for several useful discussions, and to Andrew Gordon and Paul Hankin for permission to adapt their macros and inference rules. K. Rustan M. Leino developed the Houdini architecture for ESC/Java with Cormac Flanagan, and Yuan Yu helped us apply `Houdini/rcc` to the TLC source code. Martín Abadi's work was started at Compaq; it was also partly supported by the National Science Foundation under Grants CCR-0204162 and CCR-0341179, and by faculty research funds granted by the University of California, Santa Cruz. Cormac Flanagan's work was started at Compaq and was partly supported by faculty research funds granted by the University of California, Santa Cruz and by the National Science Foundation under Grant CCR-0341179. Stephen Freund's work was started at Compaq and was partly supported by faculty research funds granted by Williams College and by the National Science Foundation under Grants CCR-0306486 and CCR-0341387.

REFERENCES

Agesen, O., Freund, S. N., and Mitchell, J. C. 1997. Adding type parameterization to the Java language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 49–65.

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

Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. 1999. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the Static Analysis Symposium*, A. Cortesi and G. Filé, Eds. Lecture Notes in Computer Science, vol. 1694. Springer-Verlag, 19–38.

Amtoft, T., Nielson, F., and Nielson, H. R. 1997. Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming 7,* 3, 321–347.

Bacon, D. F., Strom, R. E., and Tarafdar, A. 2001. Guava: A dialect of Java without data races. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 382–400.

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

Blanchet, B. 1999. Escape analysis for object-oriented languages. Application to Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 20–34.

Bogda, J. and Hölzle, U. 1999. Removing unnecessary synchronization in Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 35–46.

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

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

Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 183–200.

Cardelli, L. 1988. Typechecking dependent types and subtypes. In *Lecture notes in computer science on Foundations of logic and functional programming*. Springer-Verlag New York, Inc., 45–57.

Cardelli, L. 1997. Mobile ambient synchronization. Tech. Rep. 1997-013, Digital Systems Research Center, Palo Alto, CA.

Cartwright, R. and Jr., G. L. S. 1998. Compatible genericity with run-time types for the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 201–215.

Chamillard, A. T., Clarke, L. A., and Avrunin, G. S. 1996. An empirical comparison of static concurrency analysis techniques. Technical Report 96-084, Department of Computer Science, University of Massachusetts at Amherst.

Choi, J.-D., Gupta, M., Serrano, M. J., Sreedhar, V. C., and Midkiff, S. P. 1999. Escape analysis for Java. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 1–19.

Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridhara, M. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 258–269.

Corbett, J. C. 1996. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering 22,* 3, 161–180.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 238–252.

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

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. Research Report 159, Compaq Systems Research Center.

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

Dwyer, M. B. and Clarke, L. A. 1994. Data flow analysis for verifying properties of concurrent programs. Technical Report 94-045, Department of Computer Science, University of Massachusetts at Amherst.

Fajstrup, L., Goubault, E., and Raussen, M. 1998. Detecting deadlocks in concurrent systems. In *Proceedings of the International Conference on Concurrency Theory*, D. Sangiorgi and R. de Simone, Eds. Lecture Notes in Computer Science, vol. 1466. Springer-Verlag, 332–347.

FLANAGAN, C. AND ABADI, M. 1999a. Object types against races. In *Proceedings of the International Conference on Concurrency Theory*, J. C. M. Baeten and S. Mauw, Eds. Lecture Notes in Computer Science, vol. 1664. Springer-Verlag, 288–303.

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

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

FLANAGAN, C. AND FREUND, S. N. 2001. Detecting race conditions in large programs. In *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering*. 90–96.

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

FLANAGAN, C., FREUND, S. N., AND QADEER, S. 2004. Exploiting purity for atomicity. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*. 221–231.

FLANAGAN, C., JOSHI, R., AND LEINO, K. R. M. 2001. Annotation inference for modular checkers. *Information Processing Letters 77,* 2-4, 91–108.

FLANAGAN, C. AND LEINO, K. R. M. 2001. Houdini, an annotation assistant for ESC/Java. In *Formal Methods Europe*, J. N. Oliveira and P. Zave, Eds. Lecture Notes in Computer Science, vol. 2021. Springer-Verlag, 500–517.

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

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

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

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

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

GRAF, S. AND SAIDI, H. 1997. Construction of abstract state graphs with PVS. In *Proceedings of the IEEE Conference on Computer Aided Verification*. 72–83.

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

IGARASHI, A., PIERCE, B., AND WADLER, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23,* 3, 396–450.

JAVASOFT. 1998. Java Developers Kit, version 1.1. available from `http://java.sun.com`.

JAVASOFT. 2004. Java Developer's Kit, version 1.5. available from `http://java.sun.com`.

JOUVELOT, P. AND GIFFORD, D. 1991. Algebraic reconstruction of types and effects. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 303–310.

KISTLER, T. AND MARAIS, J. 1998. WebL – a programming language for the web. In *Proceedings of the International World Wide Web Conference*. Computer Networks and ISDN Systems, vol. 30. Elsevier, 259–270.

KOBAYASHI, N. 1998. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems 20,* 2, 436–482.

KOBAYASHI, N., SAITO, S., AND SUMII, E. 2000. An implicitly-typed deadlock-free process calculus. In *Proceedings of the International Conference on Concurrency Theory*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 1877. Springer-Verlag, 489–503.

LEINO, K. R. M., SAXE, J. B., AND STATA, R. 1999. Checking Java programs via guarded commands. Tech. Rep. 1999-002, Compaq Systems Research Center, Palo Alto, CA. 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, Fernuniversität Hagen, 1999.

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

Myers, A. C., Bank, J. A., and Liskov, B. 1997. Parameterized types and Java. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 132–145.

Nielson, F. 1996. Annotated type and effect systems. *ACM Computing Surveys 28,* 2, 344–345. Invited position statement for the Symposium on Models of Programming Languages and Computation.

Nipkow, T. and von Oheimb, D. 1998. Java$_{light}$ is type-safe - definitely. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 161–170.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 146–159.

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

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

Schmidt, D. C. and Harrison, T. H. 1997. Double-checked locking – a optimization pattern for efficiently initializing and accessing thread-safe objects. In *Pattern Languages of Program Design 3*, R. Martin, F. Buschmann, and D. Riehle, Eds. Addison-Wesley.

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

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

Syme, D. 1997. Proving Java type soundness. Tech. Rep. 427, University of Cambridge Computer Laboratory Technical Report.

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

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

Tofte, M. and Talpin, J.-P. 1997. Region-based memory management. *Information and Computation 132,* 2, 109–176.

von Praun, C. and Gross, T. 2001. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 70–82.

Whaley, J. and Rinard, M. 1999. Compositional pointer and escape analysis for Java programs. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*. 187–206.

World Wide Web Consortium. 2001. Jigsaw. Available from `http://www.w3c.org`.

Yahav, E. 2001. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*. 27–40.

Yu, Y., Manolios, P., and Lamport, L. 1999. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Lecture Notes in Computer Science, vol. 1703. Springer-Verlag, 54–66.

APPENDIX

We present the syntax, semantics, and type system for RaceFreeJava without thread-local classes in Appendix A and the soundness proof for this system in Appendix B. We extend the type system to thread-local classes and discuss the modifications to the soundness proof needed for thread-local classes in Appendices C and D. We divide the formal development in this manner for simplicity of presentation, and also because we believe that, without thread-local classes, our semantics and type system should be a useful starting point for future work on other analysis problems that require dependent types and parameterized classes.

A. FORMAL DEFINITION OF RACEFREEJAVA

A.1 Syntax

The grammar for RaceFreeJava is presented in Figure 11. In order to support a substitution-based operational semantics, the set of values is extended to include addresses, which are described below. The set of expressions is also extended to include the construct `in-sync` p e, which indicates that the lock p has been acquired and e is being evaluated. Addresses and the `in-sync` construct should not appear in source programs. The grammar also defines class instantiations, which are instantiations of parameterized classes.

A.2 Semantics

We specify the operational semantics of RaceFreeJava using the abstract machine in Figure 12. The machine evaluates a program by stepping through a sequence of states. A state consists of two components: an object store and a sequence of expressions, each of which is a thread. The result of a program is the result of the initial thread, which will always appear at the beginning of a sequence of thread expressions. New threads are added to the end of the sequence. We use $T.T'$ to denote the concatenation of two sequences.

Objects are kept in an object store σ that maps addresses to objects. An object $\{\!|\, db \,|\!\}^m_c$ has three components: a map db from field names to values, a lock state m, and the class c of the object. The field map db is a list $fn_1 = v_1, \ldots, fn_n = v_n$, and we use ϵ to denote the empty field map. The lock state is either unlocked or locked, denoted by $\circ$ and $\bullet$, respectively.

Program evaluation begins in a state with an empty store and a single thread. Evaluation proceeds according to the evaluation rules, and terminates when all the threads are reduced to values. We use $\sigma[p \mapsto d]$ to denote the store that agrees with σ at all addresses except p, which is mapped to d. The store $\sigma[p.fn \mapsto v]$ denotes the store that agrees with σ at all addresses except p, which is mapped to the object $\sigma(p)$ updated so that field fn contains value v.

The reduction rules are mostly straightforward. They rely on the following auxiliary judgments. The rule [RED NEW] uses the auxiliary judgment $P; c \vdash_{\text{initial}} db$ to determine the initial field map db for a newly allocated object. The rule for method dispatch uses the auxiliary judgment $P; c \vdash_{\text{dispatch}} meth$ to determine the correct method to call. The judgment $P \vdash_{\text{inst}} ci$ states that the class declaration ci is a

$$
\begin{array}{rcll}
P & ::= & \mathit{defn}^*\ e & \text{(program)}\\
\mathit{defn} & ::= & \texttt{class}\ \mathit{cn}\langle \mathit{garg}^*\rangle\ \mathit{body} & \text{(class declaration)}\\
\mathit{garg} & ::= & \texttt{ghost}\ t\ x & \text{(ghost var)}\\
\mathit{body} & ::= & \texttt{extends}\ c\ \ \texttt{\{}\ \mathit{field}^*\ \mathit{meth}^*\texttt{\}} & \text{(class body)}\\
\mathit{field} & ::= & [\texttt{final}]_{\mathrm{opt}}\ t\ \mathit{fn}\ \texttt{=}\ v\ [\texttt{guarded\_by}\ l]_{\mathrm{opt}} & \text{(field declaration)}\\
\mathit{meth} & ::= & t\ \mathit{mn}\texttt{(}\mathit{arg}^*\texttt{)}\ \texttt{requires}\ \mathit{ls}\ \texttt{\{}\ e\ \texttt{\}} & \text{(method declaration)}\\
\mathit{arg} & ::= & t\ x & \text{(variable declaration)}\\
s, t & ::= & c\ \mid\ \texttt{int} & \text{(type)}\\
c & ::= & \mathit{cn}\langle l^*\rangle\ \mid\ \texttt{Object} & \text{(class type)}\\
\mathit{ls} & ::= & l^* & \text{(lock set)}\\
l & ::= & v & \text{(lock)}\\
 & & & \\
\mathit{ci} & ::= & \texttt{class}\ c\ \mathit{body} & \text{(class instantiation)}\\
 & & & \\
e, f & ::= & v & \text{(value)}\\
 & \mid & \texttt{new}\ c\texttt{()} & \text{(allocate)}\\
 & \mid & e.\mathit{fn} & \text{(field read)}\\
 & \mid & e.\mathit{fn}\ \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\ e & \text{(synchronization)}\\
 & \mid & e.\texttt{fork} & \text{(fork)}\\
 & \mid & \texttt{in-sync}\ p\ e & \text{(in sync)}\\
 & & & \\
v & ::= & x & \text{(variable)}\\
 & \mid & n & \text{(integer)}\\
 & \mid & \texttt{null} & \text{(null reference)}\\
 & \mid & p & \text{(address)}\\
 & & & \\
\mathit{cn} & \in & \text{class names} & \\
\mathit{fn} & \in & \text{field names} & \\
\mathit{mn} & \in & \text{method names} & \\
x, y & \in & \text{variable names} & \\
n & \in & \text{integers} & \\
p & \in & \text{addresses} &
\end{array}
$$

Fig. 11. The grammar for RaceFreeJava.

Evaluator

$$eval(P, v_1) \iff P \vdash \langle \emptyset, e \rangle \longmapsto^* \langle \sigma, v_1 . \ldots . v_n \rangle \quad \text{where } P = defn^*\ e$$

State Space

$$\begin{array}{rcl} S & \in & State = Store \times ThreadSeq \\ \sigma & \in & Store = p \rightharpoonup d \\ T & \in & ThreadSeq = e^* \\ d & ::= & \{\!|\, db \,|\!\}_c^m \\ db & ::= & fn_1 = v_1, \cdots, fn_n = v_n \\ m & ::= & \circ \mid \bullet \end{array}$$

Evaluation Contexts

$$\begin{array}{rcl} \mathcal{E} & = & [\,] \mid \mathcal{E}.fn \mid \mathcal{E}.fn\ \texttt{=}\ e \mid p.fn\ \texttt{=}\ \mathcal{E} \\ & \mid & \mathcal{E}.mn\texttt{(}e^*\texttt{)} \mid p.mn\texttt{(}v^*, \mathcal{E}, e^*\texttt{)} \mid \texttt{let}\ arg\ \texttt{=}\ \mathcal{E}\ \texttt{in}\ e \\ & \mid & \mathcal{E}.\texttt{fork} \mid \texttt{synchronized}\ \mathcal{E}\ e \mid \texttt{in-sync}\ p\ \mathcal{E} \end{array}$$

Transition Rules

[RED NEW]

$$P \vdash \langle \sigma, T.\mathcal{E}[\texttt{new}\ c\texttt{()}].T' \rangle \longmapsto \langle \sigma[p \mapsto \{\!|\, db \,|\!\}_c^\circ], T.\mathcal{E}[p].T' \rangle \quad \text{if } p \notin dom(\sigma) \text{ and } P; c \vdash_{\text{initial}} db$$

[RED READ]

$$P \vdash \langle \sigma, T.\mathcal{E}[p.fn].T' \rangle \longmapsto \langle \sigma, T.\mathcal{E}[v].T' \rangle \quad \text{if } \sigma(p) = \{\!|\, \ldots, fn = v, \ldots \,|\!\}_c^m$$

[RED ASSIGN]

$$P \vdash \langle \sigma, T.\mathcal{E}[p.fn\ \texttt{=}\ v].T' \rangle \longmapsto \langle \sigma[p.fn \mapsto v], T.\mathcal{E}[v].T' \rangle$$

[RED INVOKE]

$$P \vdash \langle \sigma, T.\mathcal{E}[p.mn\texttt{(}v_{1..n}\texttt{)}].T' \rangle \longmapsto \langle \sigma, T.\mathcal{E}[[v_i/x_i{}^{i \in 1..n}, p/\texttt{this}]e].T' \rangle \quad \text{if } \sigma(p) = \{\!|\, db \,|\!\}_c^m \text{ and } P; c \vdash_{\text{dispatch}} t\ mn\texttt{(}t_i\ x_i\ {}^{i \in 1..n}\texttt{)}\ \texttt{requires}\ ls\ \texttt{\{}\ e\ \texttt{\}}$$

[RED LET]

$$P \vdash \langle \sigma, T.\mathcal{E}[\texttt{let}\ t\ x\ \texttt{=}\ v\ \texttt{in}\ e].T' \rangle \longmapsto \langle \sigma, T.\mathcal{E}[[v/x]e].T' \rangle$$

[RED SYNC]

$$P \vdash \langle \sigma[p \mapsto \{\!|\, db \,|\!\}_c^\circ], T.\mathcal{E}[\texttt{synchronized}\ p\ e].T' \rangle \longmapsto \langle \sigma[p \mapsto \{\!|\, db \,|\!\}_c^\bullet], T.\mathcal{E}[\texttt{in-sync}\ p\ e].T' \rangle$$

[RED IN-SYNC]

$$P \vdash \langle \sigma[p \mapsto \{\!|\, db \,|\!\}_c^\bullet], T.\mathcal{E}[\texttt{in-sync}\ p\ v].T' \rangle \longmapsto \langle \sigma[p \mapsto \{\!|\, db \,|\!\}_c^\circ], T.\mathcal{E}[v].T' \rangle$$

[RED FORK]

$$P \vdash \langle \sigma, T.\mathcal{E}[p.\texttt{fork}].T' \rangle \longmapsto \langle \sigma, T.\mathcal{E}[0].T'.(p.\texttt{run()}) \rangle$$

Fig. 12. The semantics for RaceFreeJava.

well-formed instantation of a class appearing in the program P.

Judgment	Meaning
$P; c \vdash_{\text{initial}} db$	db are the initialized fields of an object of class c
$P; c \vdash_{\text{dispatch}} meth$	$meth$ is a resolved method for c
$P \vdash_{\text{inst}} ci$	ci is an instantiated class from P

These auxiliary judgments are defined as follows:

$\boxed{P; c \vdash_{\text{initial}} db}$

[INIT EMPTY]

$$\frac{}{P; \texttt{Object} \vdash_{\text{initial}} \epsilon}$$

[INIT OBJECT]

$$\frac{\begin{array}{c} P; c' \vdash_{\text{initial}} db \\ P \vdash_{\text{inst}} \texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{\{}\ \mathit{field}_{1..m}\ \mathit{meth}_{1..n}\ \texttt{\}} \\ \mathit{field}_i = ([\texttt{final}]_{\text{opt}}\ t_i\ \mathit{fn}_i\ \texttt{=}\ v_i\ [\texttt{guarded\_by}\ l_i]_{\text{opt}}) \quad \forall i \in 1..m \end{array}}{P; c \vdash_{\text{initial}} db, \mathit{fn}_1 = v_1, \ldots, \mathit{fn}_m = v_m}$$

$\boxed{P; c \vdash_{\text{dispatch}} meth}$

[LOOKUP]

$$\frac{\begin{array}{c} P \vdash_{\text{inst}} \texttt{class}\ c\ \texttt{extends}\ c' \\ \texttt{\{}\ldots\ \mathit{meth}\ \ldots\texttt{\}} \end{array}}{P; c \vdash_{\text{dispatch}} \mathit{meth}}$$

[LOOKUP SUPER]

$$\frac{\begin{array}{c} P \vdash_{\text{inst}} \texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{\{}\ldots\ \mathit{meth}_{1..k}\ \texttt{\}} \\ P; c' \vdash_{\text{dispatch}} \mathit{meth} \\ \mathit{meth} = t\ \mathit{mn}\texttt{(}\mathit{arg}_{1..n}\texttt{)}\ \texttt{requires}\ \mathit{ls}\ \texttt{\{}\ e\ \texttt{\}} \\ \mathit{meth}_i = t_i\ \mathit{mn}_i\texttt{(}\mathit{arg}_{i,1..n_i}\texttt{)}\ \texttt{requires}\ \mathit{ls}_i\ \texttt{\{}\ e_i\ \texttt{\}} \quad \forall i \in 1..k \\ \mathit{mn} \neq \mathit{mn}_i \quad \forall i \in 1..k \end{array}}{P; c \vdash_{\text{dispatch}} \mathit{meth}}$$

$\boxed{P \vdash_{\text{inst}} ci}$

[INST]

$$\frac{\texttt{class}\ \mathit{cn}\langle \texttt{ghost}\ t_i\ x_i^{\ i \in 1..n}\rangle\ \mathit{body} \in P}{P \vdash_{\text{inst}} \texttt{class}\ \mathit{cn}\langle l_{1..n}\rangle\ [l_i/x_i^{\ i \in 1..n}]\mathit{body}}$$

The rule [RED FORK] for p.`fork` creates a new thread to evaluate p.`run()`, and returns 0 as the (dummy) result of the fork expression. The rule [RED SYNC] evaluates the expression `synchronized` p e by acquiring the lock of p, and it yields the expression `in-sync` p e. The new expression denotes that the lock of p has been acquired and e is being evaluated. After e evaluates to some value v, the rule [RED IN-SYNC] releases the lock of p and returns the value v. We say that an expression e is in a *critical section* on p if $e = \mathcal{E}[\texttt{in-sync}\ p\ e']$ for some evaluation context $\mathcal{E}$ and expression e'.

We use the semantics to formalize the notion of a race condition. An expression e *accesses* $p.\mathit{fn}$ if $e = \mathcal{E}[p.\mathit{fn}]$ or $e = \mathcal{E}[p.\mathit{fn} = v]$ for some $\mathcal{E}$ and v. A state has *conflicting accesses* on $p.\mathit{fn}$ if its thread sequence contains two or more (top-level) expressions that access $p.\mathit{fn}$ and at least one of the accesses is a write. A program has a *race condition* if its evaluation may yield a state with conflicting accesses. In other words, a program $P = \mathit{defn}^*\ e$ has a race condition if $P \vdash \langle \emptyset, e\rangle \longmapsto^* S$ where S has conflicting accesses.

A.3 Type System

We first define a number of predicates used in the type system informally. These predicates are based on similar predicates from a previous paper [Flatt et al. 1998], to which we refer the reader 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 maps variables to types, addresses to classes, and ghost variables to types:

$$E \ ::= \ \emptyset \ \mid \ E, \mathit{arg} \ \mid \ E, c\ p \ \mid \ E, \mathit{garg}$$

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 declaration
$P \vdash E$	E is a well-formed typing environment
$P; E \vdash t$	t is a well-formed type
$P; E \vdash s <: t$	s is a subtype of t
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P; E \vdash \mathit{field}$	field is a well-formed field
$P; E \vdash \varsigma y.\ \mathit{field} \in c$	class c declares/inherits field, where y is the self-reference variable
$P; E \vdash \varsigma y.\ \mathit{meth} \in c$	class c declares/inherits meth, where y is the self-reference variable
$P; E \vdash_{\mathrm{final}} e : t$	e 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
$P \vdash S : E$	S is a well-formed state with environment E
$P \vdash \sigma : E$	σ is a well-formed store with environment E
$P; E; p \vdash \mathit{db} : c$	db is a well-formed object of class c at address p

The typing rules for these judgments are presented below.

$\boxed{\vdash P : t}$

[PROG]

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

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

[CLASS]

$$\frac{\begin{array}{c}P; \emptyset \vdash t_i \quad \forall i \in 1..n \\ \mathit{garg}_i = \texttt{ghost}\ t_i\ x_i \quad \forall i \in 1..n \\ E = \mathit{garg}_{1..n}, \mathit{cn}\langle x_{1..n}\rangle\ \texttt{this} \\ P; \mathit{garg}_{1..n} \vdash c \\ P; E \vdash \mathit{field}_i \quad \forall i \in 1..j \\ P; E \vdash \mathit{meth}_i \quad \forall i \in 1..k\end{array}}{\begin{array}{c}P \vdash \texttt{class}\ \mathit{cn}\langle \mathit{garg}_{1..n}\rangle\ \texttt{extends}\ c \\ \texttt{\{}\ \mathit{field}_{1..j}\ \mathit{meth}_{1..k}\texttt{\}}\end{array}}$$

$\boxed{P \vdash E}$

[ENV EMPTY]

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

[ENV X]

$$\frac{P; E \vdash t \qquad x \notin \mathit{dom}(E)}{P \vdash E, [\texttt{ghost}]_{\mathrm{opt}}\ t\ x}$$

[ENV ADDRESS]

$$\frac{P; E \vdash c \qquad p \notin \mathit{dom}(E)}{P \vdash E, c\ p}$$

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

[TYPE C]
$$\frac{\texttt{class}\ cn\langle \texttt{ghost}\ t_i\ x_i{}^{i \in 1..n}\rangle\ body \in P \qquad P; E \vdash_{\text{final}} l_i : t_i \quad \forall i \in 1..n}{P; E \vdash cn\langle l_{1..n}\rangle}$$

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

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

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

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

[SUBTYPE CLASS]
$$\frac{P; E \vdash c <: cn\langle l_{1..n}\rangle \qquad \texttt{class}\ cn\langle \texttt{ghost}\ t_i\ x_i{}^{i \in 1..n}\rangle\ \texttt{extends}\ c'\ \ldots \in P}{P; E \vdash c <: [l_i/x_i{}^{i \in 1..n}]c'}$$

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

[FIELD]
$$\frac{P; E \vdash_{\text{final}} l : c \qquad P; E; \emptyset \vdash v : t}{P; E \vdash t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l}$$

[FINAL FIELD]
$$\frac{P; E; \emptyset \vdash v : t}{P; E \vdash \texttt{final}\ t\ fn\ \texttt{=}\ v}$$

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

[METHOD]
$$\frac{P; E \vdash t \qquad P; E \vdash ls \qquad P; E, arg_{1..n}; ls \vdash e : t}{P; E \vdash t\ mn(arg_{1..n})\ \texttt{requires}\ ls\ \texttt{\{}\ e\ \texttt{\}}}$$

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

[FIELD C]
$$\frac{P; E \vdash c <: cn\langle l_{1..n}\rangle \qquad y \text{ does not appear in } l_{1..n} \qquad \texttt{class}\ cn\langle \texttt{ghost}\ t_i\ x_i{}^{i \in 1..n}\rangle\ \texttt{extends}\ c'\ \texttt{\{}\ \ldots\ \mathit{field}\ \ldots\ \texttt{\}} \in P}{P; E \vdash \varsigma y.\ ([l_i/x_i{}^{i \in 1..n}, y/\texttt{this}]\mathit{field}) \in c}$$

$$\boxed{P; E \vdash \varsigma y.\ \mathit{meth} \in c}$$

[METH C]
$$\frac{P; E \vdash c <: cn\langle l_{1..n}\rangle \qquad y \text{ does not appear in } l_{1..n} \qquad \texttt{class}\ cn\langle \texttt{ghost}\ t_i\ x_i{}^{i \in 1..n}\rangle\ \texttt{extends}\ c'\ \texttt{\{}\ \ldots\ \mathit{meth}\ \ldots\ \texttt{\}} \in P}{P; E \vdash \varsigma y.\ ([l_i/x_i{}^{i \in 1..n}, y/\texttt{this}]\mathit{meth}) \in c}$$

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

[FINAL VAR]
$$\frac{P \vdash E \qquad E = E_1, [\texttt{ghost}]_{\text{opt}}\ c\ x, E_2}{P; E \vdash_{\text{final}} x : c}$$

[FINAL ADDR]
$$\frac{P \vdash E \qquad E = E_1, c\ p, E_2}{P; E \vdash_{\text{final}} p : c}$$

[FINAL NULL]
$$\frac{P; E \vdash c}{P; E \vdash_{\text{final}} \texttt{null} : c}$$

[FINAL SUB]
$$\frac{P; E \vdash_{\text{final}} e : s \qquad P; E \vdash s <: t}{P; E \vdash_{\text{final}} e : t}$$

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

[LS EMPTY]
$$\frac{P \vdash E}{P; E \vdash \emptyset}$$

[LS ADD]
$$\frac{P; E \vdash ls \qquad P; E \vdash_{\text{final}} l : c}{P; E \vdash ls \cup \{l\}}$$

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

[LS ELEM]
$$\frac{l \in ls \qquad P; E \vdash ls}{P; E \vdash l \in ls}$$

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

[LS SUBSET]
$$\frac{P; E \vdash ls_1 \qquad P; E \vdash ls_2 \qquad ls_1 \subseteq ls_2}{P; E \vdash ls_1 \subseteq ls_2}$$

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

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

[EXP INT]
$$\frac{P; E \vdash ls \qquad n \in \text{integers}}{P; E; ls \vdash n : \texttt{int}}$$

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

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

[EXP ADDRESS]
$$\frac{P; E \vdash ls \qquad E = E_1, c\ p, E_2}{P; E; ls \vdash p : c}$$

[EXP NEW]
$$\frac{P; E \vdash ls \qquad P; E \vdash c}{P; E; ls \vdash \texttt{new}\ c\texttt{()} : c}$$

[EXP REF UNGUARDED]
$$\frac{P; E; ls \vdash e : c \qquad P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v) \in c \qquad P; E \vdash [e/\texttt{this}]t}{P; E; ls \vdash e.fn : [e/\texttt{this}]t}$$

[EXP REF GUARDED]
$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\mathrm{opt}}\ t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c \\ P; E \vdash [e/\texttt{this}]l \in ls \\ P; E \vdash [e/\texttt{this}]t \end{array}}{P; E; ls \vdash e.fn : [e/\texttt{this}]t}$$

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

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

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

[EXP FORK]
$$\frac{\begin{array}{c} P; E; ls \vdash e : c \\ P; E \vdash \varsigma\texttt{this}.\ (t\ \texttt{run()}\ \texttt{requires}\ \emptyset\ \texttt{\{}\ e'\ \texttt{\}}) \in c \end{array}}{P; E; ls \vdash e.\texttt{fork} : \texttt{int}}$$

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

[EXP IN-SYNC]
$$\frac{P; E \vdash_{\mathrm{final}} p : c \qquad P; E; ls \cup \{p\} \vdash e : t}{P; E; ls \vdash \texttt{in-sync}\ p\ e : t}$$

$\boxed{P \vdash S : E}$

[STATE]
$$\frac{\begin{array}{c} P \vdash \sigma : E \\ |T| > 0 \\ P; E; \emptyset \vdash T_i : t_i \quad \forall i \in 1..|T| \end{array}}{P \vdash \langle \sigma, T \rangle : E}$$

$\boxed{P \vdash \sigma : E}$

[STORE]
$$\frac{\begin{array}{c} \vdash P : t \\ dom(\sigma) = \{p_1, \ldots, p_n\} \\ \sigma(p_i) = \{\!|\ldots|\!\}_{c_i}^{m_i} \quad \forall i \in 1..n \\ E = c_1\ p_1, \ldots, c_n\ p_n \\ P; E; p_i \vdash \sigma(p_i) : c_i \quad \forall i \in 1..n \end{array}}{P \vdash \sigma : E}$$

$\boxed{P; E; p \vdash db : t}$

[VAL OBJECT]
$$\frac{}{P; E; p \vdash \{\!|\ \ |\!\}_{\texttt{Object}}^{m} : \texttt{Object}}$$

[VAL SUBCLASS]
$$\frac{\begin{array}{c} P; E \vdash c \\ P \vdash_{\mathrm{inst}} \texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{\{}\ field_{1..n}\ meth_{1..m}\ \texttt{\}} \\ field_i = \ldots\ t_i\ fn_i\ \ldots \quad \forall i \in 1..n \\ P; E; \emptyset \vdash v_i : [p/\texttt{this}]t_i \quad \forall i \in 1..n \\ P; E; p \vdash \{\!|\,db\,|\!\}_{c'}^{m'} : c' \end{array}}{P; E; p \vdash \{\!|\,db, fn_1 = v_1, \ldots, fn_n = v_n\,|\!\}_{c}^{m} : c}$$

B. SOUNDNESS

In this section, we prove that well-typed RaceFreeJava programs without thread-local classes cannot have race conditions. The preliminary lemmas are routine, and much of their structure is derived directly from previous work on similar systems [Flatt et al. 1998]. Therefore, we primarily focus on the novel aspects of RaceFreeJava, including how lock sets and dependent types are handled.

We start by presenting a lemma that states that, given a well-typed expression, there is a deduction that yields that the expression has a type and that does not end with an application of [EXP SUB]. This lemma enables us to establish a number of properties, such as that subexpressions of a well-typed expression are well-typed.

Lemma 1. *Suppose there exists a deduction that concludes $P; E; ls \vdash e : t$. Then that deduction contains a subdeduction that concludes $P; E; ls \vdash e : t'$ and does not end with an application of rule* [EXP SUB] *for some type t' where $P; E \vdash t' <: t$.*

Proof. Induction over the derivation of $P; E; ls \vdash e : t$. □

Lemma 2 (Context subexpression). *Suppose there is a deduction that concludes $P; E; ls \vdash \mathcal{E}[e] : t$. Then that deduction contains, at a position corresponding to the hole in $\mathcal{E}$, a subdeduction that concludes $P; E; ls' \vdash e : t'$ for some t' and ls' such that $P; E \vdash ls \subseteq ls'$ and for all $l \in ls' \setminus ls$, $\mathcal{E}[e]$ is in a critical section on l.*

Proof. The proof is a routine induction on the structure of $\mathcal{E}$. We show the representative case where $\mathcal{E} = \mathcal{E}'.fn$. If $P; E; ls \vdash \mathcal{E}'[e].fn : t$, then Lemma 1 indicates that $P; E; ls \vdash \mathcal{E}'[e].fn : s$ for some s, where an application of [exp sub] is not the last step of the derivation. Suppose that [exp ref guarded] is used to conclude that $P; E; ls \vdash \mathcal{E}'[e].fn : s$, using the antecedent $P; E; ls \vdash \mathcal{E}'[e] : c$. By the induction hypothesis, $P; E; ls' \vdash e : t'$ for some t' and $\forall l \in ls' \setminus ls$. $\mathcal{E}[e]$ is in a critical section on l. The case for [exp ref unguarded] is similar. □

Note that the subexpression e is typed with the original lock set, extended only with locks for which $\mathcal{E}[e]$ is in a critical section.

Lemma 3. *Suppose there is a deduction that concludes $P; E; ls \vdash \mathcal{E}[e] : t$. Then that deduction contains, at a position corresponding to the hole in $\mathcal{E}$, a subdeduction that concludes $P; E; ls' \vdash e : t'$ for some t' and ls' such that $P; E \vdash ls \subseteq ls'$ and for all $l \in ls' \setminus ls$, $\mathcal{E}[e]$ is in a critical section on l. In addition, the subdeduction ends with an application of a rule other than* [exp sub].

Proof. Simple application of Lemmas 1 and 2. □

The next lemma shows that a subexpression e_1 may be replaced by a different subexpression with the same type, provided e_1 is not a value. (If e_1 is a value, it may appear in the type t or in the lock set ls, since our system supports dependent types that contain values.)

Lemma 4 (Context replacement). *Suppose a deduction concluding $P; E; ls \vdash \mathcal{E}[e_1] : t$ contains a deduction concluding $P; E; ls' \vdash e_1 : t'$ at a position corresponding to the hole in $\mathcal{E}$. If e_1 is not a value and $P; E; ls' \vdash e_2 : t'$, then $P; E; ls \vdash \mathcal{E}[e_2] : t$.*

Proof. Induction on the structure of $\mathcal{E}$. □

The next lemma shows that judgments from the formal system are preserved under capture-free variable substitution. We use Z as a place holder for the right-hand side of any judgment used in the type system.

Lemma 5 (Substitution). *If $P; E; ls' \vdash v : s$ then:*

(1) *if $P \vdash E, s\ x, E'$ then $P \vdash E, [v/x]E'$, and*
(2) *if $P; E, s\ x, E' \vdash Z$ then $P; E, [v/x]E' \vdash [v/x]Z$, and*
(3) *if $P; E, s\ x, E'; ls \vdash Z$ then $P; E, [v/x]E'; [v/x]ls \vdash [v/x]Z$, and*
(4) *if $P; E, s\ x, E' \vdash_{\text{final}} Z$ then $P; E, [v/x]E' \vdash_{\text{final}} [v/x]Z$.*

Proof. The proof is by a simultaneous induction on the four parts of the lemma. We present details of the proof of part 3 for expression typing judgments. In particular, we show that if $P; E; ls' \vdash v : s$ and $P; E, s\ x, E'; ls \vdash e : t$ then $P; E, [v/x]E'; [v/x]ls \vdash [v/x]e : [v/x]t$ by induction over the derivation of $P; E, s\ x, E'; ls \vdash e : t$. We consider three representative cases:

[EXP SUB]: To have applied this rule, it must be the case that $P; E, s\ x, E'; ls \vdash e : t'$ and $P; E, s\ x, E' \vdash t' <: t$. By the induction hypothesis:

$$P; E, [v/x]E'; [v/x]ls \vdash [v/x]e : [v/x]t'$$
$$P; E, [v/x]E' \vdash [v/x]t' <: [v/x]t$$

Thus, we apply [EXP SUB] to conclude $P; E, [v/x]E'; [v/x]ls \vdash [v/x]e : [v/x]t$.

[EXP REF GUARDED]: Let e be the expression $e'.fn$. Then:

$$P; E, s\ x, E'; ls \vdash e' : c$$
$$P; E, s\ x, E' \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t'\ fn\ \texttt{=}\ v'\ \texttt{guarded\_by}\ l) \in c$$
$$P; E, s\ x, E' \vdash [e'/\texttt{this}]l \in ls$$
$$P; E, s\ x, E' \vdash [e'/\texttt{this}]t'$$

where $t = [e'/\texttt{this}]t'$. We may assume $x \neq \texttt{this}$ and $v \neq \texttt{this}$, renaming the bound variable if necessary. By the induction hypothesis:

$$P; E, [v/x]E'; [v/x]ls \vdash [v/x]e' : [v/x]c$$
$$P; E, [v/x]E' \vdash [v/x](\varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t'\ fn\ \texttt{=}\ v'\ \texttt{guarded\_by}\ l) \in c)$$
$$P; E, [v/x]E' \vdash [v/x]([e'/\texttt{this}]l) \in [v/x]ls$$
$$P; E, [v/x]E' \vdash [v/x]([e'/\texttt{this}]t')$$

The second line above simplifies to:

$$P; E, [v/x]E' \vdash$$
$$\varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ [v/x]t'\ fn\ \texttt{=}\ [v/x]v'\ \texttt{guarded\_by}\ [v/x]l) \in [v/x]c$$

In addition, $[v/x]t = [v/x]([e'/\texttt{this}]t') = [([v/x]e')/\texttt{this}]([v/x]t')$, and we may transform lines 3 and 4 into:

$$P; E, [v/x]E' \vdash [([v/x]e')/\texttt{this}]([v/x]l) \in [v/x]ls$$
$$P; E, [v/x]E' \vdash [([v/x]e')/\texttt{this}]([v/x]t')$$

Using these conditions, we can apply the rule [EXP REF GUARDED] to conclude $P; E, [v/x]E'; [v/x]ls \vdash [v/x]e : [v/x]t$.

[EXP LET]: Let e be the expression $\texttt{let}\ t_1\ y\ \texttt{=}\ e_1\ \texttt{in}\ e_2$. It must be that:

$$P; E, s\ x, E'; ls \vdash e_1 : t_1$$
$$P; E, s\ x, E', t_1\ y; ls \vdash e_2 : t_2$$
$$P; E, s\ x, E' \vdash t$$

where $t = [e_1/y]t_2$. We know that the variable x is different from y because $E, s\ x, E', t_1\ y$ is a well-formed environment. By the induction hypothesis:

$$P; E, [v/x]E'; [v/x]ls \vdash [v/x]e_1 : [v/x]t_1$$
$$P; E, [v/x]E', [v/x]t_1\ y; [v/x]ls \vdash [v/x]e_2 : [v/x]t_2$$
$$P; E, [v/x]E' \vdash [v/x]t$$

Since x and y are distinct and y is not free in v, $[v/x]t = [v/x]([e_1/y]t_2) = [([v/x]e_1)/y]([v/x]t_2)$. Therefore, $P; E, [v/x]E' \vdash [([v/x]e_1)/y]([v/x]t_2)$. We may thus conclude $P; E, [v/x]E'; [v/x]ls \vdash [v/x]e : [v/x]t$ using rule [EXP LET].

□

A similar lemma is used to substitute values for ghost variables.

Lemma 6 (Ghost substitution). *If $P; E \vdash_{\text{final}} v : t$, then:*

(1) *if $P \vdash E, \texttt{ghost}\ t\ x, E'$ then $P \vdash E, [v/x]E'$, and*
(2) *if $P; E, \texttt{ghost}\ t\ x, E' \vdash Z$ then $P; E, [v/x]E' \vdash [v/x]Z$, and*
(3) *if $P; E, \texttt{ghost}\ t\ x, E'; ls \vdash Z$ then $P; E, [v/x]E'; [v/x]ls \vdash [v/x]Z$, and*
(4) *if $P; E, \texttt{ghost}\ t\ x, E' \vdash_{\text{final}} Z$ then $P; E, [v/x]E' \vdash_{\text{final}} [v/x]Z$.*

Proof. Proof is by simultaneous induction on the four parts. □

Next we show the conditions under which an environment can be strengthened with additional variable declarations.

Lemma 7 (Environment strengthening). *If $E = E', [\texttt{ghost}]_{\text{opt}}\ t\ v, E''$ and $P \vdash E$ then:*

(1) *if $P; E', E'' \vdash Z$ then $P; E \vdash Z$, and*
(2) *if $P; E', E''; ls \vdash Z$ then $P; E; ls \vdash Z$, and*
(3) *if $P; E', E'' \vdash_{\text{final}} Z$ then $P; E \vdash_{\text{final}} Z$.*

Proof. By simultaneous induction on the three parts of the lemma. □

The previous two lemmas are sufficient to prove that the fields, methods, and class names of instantiations are well-formed. We begin by showing that class names appearing in typing derivations are well-formed.

Lemma 8 (Class name instantiation). *If $\vdash P : t$ and $P; E; ls \vdash p : c$ then $P; E \vdash c$.*

Proof. If we deduce $P; E; ls \vdash p : c$ with rule [exp address], then $E = E_1, c\, p, E_2$, where $P \vdash E$. Well-formed environments contain only valid class names. Thus, $P; E \vdash c$.

If we deduce $P; E; ls \vdash p : c$ with rule [exp sub], then $P; E; ls \vdash p : c_1$ where $P; E \vdash c_1 <: c$ for some c_1. We proceed to show by induction on the deduction of $P; E \vdash c_1 <: c$ that if $P; E; ls \vdash p : c_1$ and $P; E \vdash c_1 <: c$, then $P; E \vdash c$. There are two cases:

[subtype refl]: A hypothesis of the rule is $P; E \vdash c$.

[subtype class]: In this case:

$$P; E \vdash c_1 <: cn\langle l_{1..n} \rangle$$
$$\texttt{class}\ cn\langle \texttt{ghost}\ t_i\ x_i\ ^{i \in 1..n} \rangle\ \texttt{extends}\ c'\ \texttt{\{ ... \}} \in P$$
$$c = [l_i/x_i\ ^{i \in 1..n}]c'$$

The inductive hypothesis indicates that $P; E \vdash cn\langle l_{1..n} \rangle$. Therefore, $P; E \vdash_{\text{final}} l_i : t_i$ for $i \in 1..n$. Since $\vdash P : t$, we know that the definition of cn is well-formed, which implies that $P; \texttt{ghost}\ t_i\ x_i\ ^{i \in 1..n} \vdash c'$ as a requirement of rule [class]. Assuming that $x_{1..n}$ do not appear in E (α-renaming if necessary), we may apply Lemma 7 to obtain

$$P; E, \texttt{ghost}\ t_i\ x_i\ ^{i \in 1..n} \vdash c'$$

Replacing each x_i with l_i using Lemma 6 gives us $P; E \vdash c$.

□

LEMMA 9 (FIELD AND METHOD INSTANTIATION). *If* $\vdash P : t$ *and* $P; E; ls \vdash p : c$ *then:*

(1) *if* $P; E \vdash \varsigma y.\ field \in c$ *then* $P; E \vdash [p/y]field$*, and*

(2) *if* $P; E \vdash \varsigma y.\ meth \in c$ *then* $P; E \vdash [p/y]meth$*.*

PROOF. We consider fields. We can handle methods in a similar fashion. If $P; E \vdash \varsigma y.\ field \in c$, then

$$\begin{array}{l} P; E \vdash c <: cn\langle l_{1..n}\rangle \\ y \text{ does not appear in } l_{1..n} \\ \texttt{class}\ cn\langle \texttt{ghost}\ t_i\ x_i\ {}^{i \in 1..n}\rangle\ \ldots\ \texttt{\{}\ \ldots\ field'\ \ldots\texttt{\}} \in P \\ field = [l_i/x_i\ {}^{i\in 1..n}, y/\texttt{this}]field' \end{array}$$

Since $P; E; ls \vdash p : cn\langle l_{1..n}\rangle$ by rule [EXP SUB], Lemma 8 indicates that $P; E \vdash cn\langle l_{1..n}\rangle$. Thus, $P; E \vdash_{\text{final}} l_i : t_i$ for $i \in 1..n$. Since $\vdash P : t$, we know that the declaration of cn is well-formed, which requires that $P; \texttt{ghost}\ t_i\ x_i\ {}^{i\in 1..n}, cn\langle x_{1..n}\rangle\ \texttt{this} \vdash field'$. Assuming that $x_{1..n}$ and y do not appear in E (α-renaming if necessary), we may apply Lemma 7 and rename `this` to obtain

$$P; E, \texttt{ghost}\ t_i\ x_i\ {}^{i\in 1..n}, cn\langle x_{1..n}\rangle\ y \vdash [y/\texttt{this}]field'$$

After replacing x_i with l_i for $i \in 1..n$, we have $P; E, cn\langle l_{1..n}\rangle\ y \vdash field$, which leaves us with $P; E \vdash [p/y]field$ after replacing y with p. □

We next show that the types assigned to addresses are supertypes of the allocated type of the corresponding heap objects.

LEMMA 10 (INFERRED TO EXACT TYPES). *If* $P \vdash \langle\sigma, T\rangle : E$ *and* $P; E; ls \vdash p : c$*, then* $P; E; \emptyset \vdash p : c$ *and there exists* c' *such that* $P; E \vdash c' <: c$*,* $E(p) = c'$*, and* $\sigma(p) = \{\!|\ db\ |\!\}^m_{c'}$*.*

PROOF. Lemma 1 indicates that there exists a c' such that $P; E \vdash c' <: c$ and $P; E; ls \vdash p : c'$ is derivable by a rule other than [EXP SUB]. Since $P; E; ls \vdash p : c'$ is derivable only by rule [EXP ADDRESS], $E(p) = c'$. In addition, to have concluded that $P \vdash \langle\sigma, T\rangle : E$, the object $\sigma(p)$ must have type c'. Thus, $\sigma(p) = \{\!|\ db\ |\!\}^m_{c'}$. Furthermore, $P; E \vdash \emptyset$ by rule [LOCK SET EMPTY], and rules [EXP ADDRESS] and [EXP SUB] allow us to conclude that $P; E; \emptyset \vdash p : c$. □

Every value stored in an object has the appropriate type.

LEMMA 11 (OBJECT FIELDS WELL-TYPED). *If* $\sigma(p) = \{\!|\ \ldots, fn = v, \ldots\ |\!\}^m_c$ *and* $P \vdash \sigma : E$ *and* $P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v'\ [\texttt{guarded\_by}\ l]_{\text{opt}}) \in c$ *then* $P; E; \emptyset \vdash v : [p/\texttt{this}]t$*.*

PROOF. Proof is a straightforward induction on the subtyping deduction used to conclude $P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v'\ [\texttt{guarded\_by}\ l]_{\text{opt}}) \in c$. □

LEMMA 12 (LOCK SET STRENGTHENING). *If* $P; E; ls \vdash v : t$ *and* $P; E \vdash ls \subseteq ls'$*, then* $P; E; ls' \vdash v : t$*.*

PROOF. By induction on the derivation of $P; E; ls \vdash v : t$. □

The first of several subject-reduction lemmas states that types are preserved by reduction steps. Note that this itself does not prove that a program is free of race conditions, but it does allow us to conclude that all states reached from a well-typed state are also well-typed.

LEMMA 13 (TYPE SUBJECT REDUCTION). *If* $P \vdash S : E$ *and* $P \vdash S \longmapsto S'$ *then* $P \vdash S' : E'$ *for some* E'.

PROOF. We proceed by case analysis on the rule used to conclude $P \vdash S \longmapsto S'$.

In the first four cases presented, we assume that thread k is reduced, and let $S = \langle \sigma,\, e_1..e_k..e_n \rangle$ and $S' = \langle \sigma,\, e_1..e'_k..e_n \rangle$. Since σ and all e_i, for $i \neq k$, do not change, it is sufficient to show that $P; E; \emptyset \vdash e'_k : t_k$ follows from $P; E; \emptyset \vdash e_k : t_k$. Proving this guarantees that $P \vdash S' : E$ via [STATE].

[RED LET]: In this case:

$$e_k = \mathcal{E}[\texttt{let}\ t\ x\ \texttt{=}\ v\ \texttt{in}\ e]$$
$$e'_k = \mathcal{E}[[v/x]e]$$

Lemma 3 implies that there is a deduction of $P; E; \mathit{ls} \vdash \texttt{let}\ t\ x\ \texttt{=}\ v\ \texttt{in}\ e : s$ that does not end with an application of rule [EXP SUB] for some ls and s. The only possible rule to conclude that statement is [EXP LET]. Therefore, $P; E; \mathit{ls} \vdash v : t$ and $P; E \vdash s$ and $P; E, t\ x; \mathit{ls} \vdash e : s'$, where $s = [v/x]s'$. Given the first and third statements, Lemma 5 establishes that $P; E; [v/x]\mathit{ls} \vdash [v/x]e : [v/x]s'$. Since $E, t\ x$ is a well-formed environment, x does not appear in E. Therefore, x cannot appear in the well-formed lock set ls, and $[v/x]\mathit{ls} = \mathit{ls}$. Thus, $P; E; \mathit{ls} \vdash [v/x]e : s$, and $P; E; \emptyset \vdash e'_k : t_k$ follows by Lemma 4.

[RED READ]: In this case:

$$e_k = \mathcal{E}[p.\mathit{fn}]$$
$$e'_k = \mathcal{E}[v]$$

where $\sigma(p) = \{\!|\ \ldots, \mathit{fn} = v, \ldots\ |\!\}^{m_p}_{c_p}$. Lemma 3 implies that the derivation of $P; E; \emptyset \vdash e_k : t_k$ includes a derivation that concludes that $P; E; \mathit{ls} \vdash p.\mathit{fn} : s$ where the last rule is not rule [EXP SUB]. We proceed to show that $P; E; \mathit{ls} \vdash v : s$, and this shall allow us to establish that $P; E; \emptyset \vdash e'_k : t_k$ by Lemma 4. We consider each possible rule used to derive $P; E; \mathit{ls} \vdash p.\mathit{fn} : s$:

[EXP REF GUARDED]: It must be that

$$P; E; \mathit{ls} \vdash p : c$$
$$P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t_v\ \mathit{fn}\ \texttt{=}\ v'\ \texttt{guarded\_by}\ l) \in c$$
$$s = [p/\texttt{this}]t_v$$

Thus, we may apply Lemmas 10 and 11 to conclude that $P; E; \emptyset \vdash v : s$, which implies that $P; E; \mathit{ls} \vdash v : s$ by Lemma 12.

[EXP REF UNGUARDED]: This case is similar to the previous.

[RED INVOKE]: In this case,

$$e_k = \mathcal{E}[p.\mathit{mn}\texttt{(}v_{1..m}\texttt{)}]$$
$$e'_k = \mathcal{E}[[v_i/x_i{}^{i \in 1..m}, p/\texttt{this}]e]$$

where $\sigma(p) = \{\!|\ \mathit{db}\ |\!\}^{m}_{c}$ and $P; c \vdash_{\text{dispatch}} t\ \mathit{mn}\texttt{(}s_i\ x_i\ {}^{i \in 1..m}\texttt{)}\ \texttt{requires}\ \mathit{ls}'\ \texttt{\{}\ e\ \texttt{\}}$. Lemma 3 implies that there is a deduction of $P; E; \mathit{ls} \vdash p.\mathit{mn}\texttt{(}v_{1..m}\texttt{)} : s$ that does not end with an application of rule [EXP SUB] for some ls and s. The only possible rule to conclude that statement is [EXP INVOKE]. Therefore,

$$P; E; ls \vdash p : c$$
$$P; E \vdash \varsigma\mathtt{this}.\ (t\ mn\mathtt{(}s_j\ x_j\ {}^{j\in 1..m}\mathtt{)}\ \mathtt{requires}\ ls'\ \mathtt{\{}\ e'\ \mathtt{\}}) \in c$$
$$P; E; ls \vdash v_j : [p/\mathtt{this}]s_j \quad \forall j \in 1..m$$
$$P; E \vdash [p/\mathtt{this}]ls' \subseteq ls$$
$$P; E \vdash [p/\mathtt{this}]t$$

The method resolution in line 2 identifies a method with the same signature, but possibly a different implementation, than the dynamically dispatched version of mn. The predicate $OverridesOK(P)$ ensures that the signatures will match. From the deduction of $P; c \vdash_{\text{dispatch}} t\ mn\mathtt{(}t_i\ x_i\ {}^{i\in 1..m}\mathtt{)}\ \mathtt{requires}\ ls'\ \mathtt{\{}\ e\ \mathtt{\}}$ we can deduce $P; E \vdash \varsigma\mathtt{this}.\ (t\ mn\mathtt{(}s_j\ x_j\ {}^{j\in 1..m}\mathtt{)}\ \mathtt{requires}\ ls'\ \mathtt{\{}\ e\ \mathtt{\}}) \in c$ for the dynamically resolved method. We may then use Lemma 9 to show that $P; E \vdash [p/\mathtt{this}](t\ mn\mathtt{(}s_j\ x_j\ {}^{j\in 1..m}\mathtt{)}\ \mathtt{requires}\ ls'\ \mathtt{\{}\ e\ \mathtt{\}})$. Therefore, $P; E \vdash [p/\mathtt{this}]t$, and $P; E \vdash [p/\mathtt{this}]ls'$, and $P; E, [p/\mathtt{this}]s_j\ x_j\ {}^{j\in 1..m}; [p/\mathtt{this}]ls' \vdash [p/\mathtt{this}]e' : [p/\mathtt{this}]t$. The formal parameters $x_{1..m}$ cannot appear in t or ls', and substituting the values $v_{1..m}$ in for $x_{1..m}$ gives us $P; E; [p/\mathtt{this}]ls' \vdash [v_i/x_i\ {}^{i\in 1..m}, p/\mathtt{this}]e' : [p/\mathtt{this}]t$. We may then use Lemma 12 to conclude $P; E; ls \vdash [v_i/x_i\ {}^{i\in 1..m}, p/\mathtt{this}]e' : s$, and $P; E; \emptyset \vdash e'_k : t_k$ follows by Lemma 4.

[RED SYNC]: In this case $e_k = \mathcal{E}[\mathtt{synchronized}\ p\ e]$ and $e'_k = \mathcal{E}[\mathtt{in\text{-}sync}\ p\ e]$. As in the previous cases, we determine that there is a derivation that yields $P; E; ls \vdash \mathtt{synchronized}\ p\ e : s$ that ends in a rule other than [EXP SUB] for some ls and s. The rule must be [EXP SYNC], meaning that $P; E \vdash_{\text{final}} p : c$ and $P; E; ls \cup \{p\} \vdash e : t$, and these statements allow us to conclude that $P; E; ls \vdash \mathtt{in\text{-}sync}\ p\ e : s$ by rule [EXP IN-SYNC]. As before, $P; E; \emptyset \vdash e'_k : t_k$ follows by Lemma 4.

We examine [RED FORK] separately. Assume that thread k is reduced, and let $e_k = \mathcal{E}[p.\mathtt{fork}]$, $S = \langle\sigma, e_1..e_k..e_n\rangle$, $e'_k = \mathcal{E}[0]$, and $S' = \langle\sigma, e_1..e'_k..e_n.(p.\mathtt{run()})\rangle$. Since σ and all e_i, $i \neq k$, do not change, it is sufficient to show that, given $P; E; \emptyset \vdash e_k : t_k$, it follows that (1) $P; E; \emptyset \vdash e'_k : t_k$, and (2) $P; E; \emptyset \vdash e.\mathtt{run()} : t_e$ for some t_e. As before, we use Lemma 3 to conclude that $P; E; ls \vdash p.\mathtt{fork} : t$ is derivable by a rule other than [EXP SUB]. The rule must be [EXP FORK]; this means that t is $\mathtt{int}$ and there exists some class c such that $P; E; ls \vdash p : c$ and $P; E \vdash \varsigma\mathtt{this}.\ (t\ \mathtt{run()}\ \mathtt{requires}\ \emptyset\ \mathtt{\{}\ e'\ \mathtt{\}}) \in c$. Lemma 4 implies that $P; E; \emptyset \vdash e'_k : t_k$. By Lemma 10, $P; E; \emptyset \vdash p : c$, and $P; E; \emptyset \vdash p.\mathtt{run()} : t$. Thus, $P \vdash S' : E$.

The case for [RED NEW] is similar to the previous cases, although we must construct a new environment E' containing the new object address in order to apply [STATE] to S'. Lemma 7 shows that the new environment is sufficient to deduce that all threads are well-typed in the post-state.

All other cases are similar to the above. □

We now turn our attention to the relationship between lock sets and critical sections. We introduce a new judgment $ls \vdash_{\text{cs}} e$ to indicate that ls contains all locks for which e is in a critical section. In other words, $ls \vdash_{\text{cs}} e$ only if for all p such that $e = \mathcal{E}[\mathtt{in\text{-}sync}\ p\ e']$, the lock p is in the set ls. This property is stated as a lemma below.

$\boxed{ls \vdash_{\mathrm{cs}} f}$

$$\text{[CS EXP]}\quad \frac{e \text{ does not contain } \texttt{in-sync}}{\emptyset \vdash_{\mathrm{cs}} e} \qquad \text{[CS IN-SYNC]}\quad \frac{ls \vdash_{\mathrm{cs}} e \qquad p \notin ls}{ls \cup \{p\} \vdash_{\mathrm{cs}} \texttt{in-sync}\ p\ e} \qquad \text{[CS NOT IN-SYNC]}\quad \frac{\begin{array}{c} ls \vdash_{\mathrm{cs}} e \\ \mathcal{E} \text{ does not contain } \texttt{in-sync} \end{array}}{ls \vdash_{\mathrm{cs}} \mathcal{E}[e]}$$

We also extend the notion of well-formed critical sections to states with the judgment $\vdash_{\mathrm{cs}} S$. The set $ls_1 \uplus ls_2$ is the disjoint union of ls_1 and ls_2.

$\boxed{\vdash_{\mathrm{cs}} S}$

$$\text{[CS STATE]}\quad \frac{\begin{array}{c} ls_i \vdash_{\mathrm{cs}} T_i \quad \forall i \in 1..|T| \\ ls = ls_1 \uplus \ldots \uplus ls_n \\ \forall p \in ls.\ \sigma(p) = \{\!|\ldots|\!\}^{\bullet}_{c_p} \end{array}}{\vdash_{\mathrm{cs}} \langle \sigma, T \rangle}$$

LEMMA 14. *If e is in a critical section on p and $ls \vdash_{\mathrm{cs}} e$, then $p \in ls$.*

PROOF. The proof is a straightforward induction on the derivation of $ls \vdash_{\mathrm{cs}} e$. □

In order to conclude that a program state has well-formed critical sections, the [CS STATE] rule requires that a distinct set of locks be held by each thread. This property is preserved by reduction steps on well-typed states:

LEMMA 15 (MUTUAL EXCLUSION SUBJECT REDUCTION). *If $P \vdash S : E$ and $\vdash_{\mathrm{cs}} S$ and $P \vdash S \longmapsto S'$ then $\vdash_{\mathrm{cs}} S'$.*

PROOF. The proof is by case analysis on the reduction rule for $P \vdash S \longmapsto S'$. All cases are straightforward except [RED SYNC] and [RED IN-SYNC]. In each case, let $S = \langle \sigma, e_{1..n} \rangle$. Since $\vdash_{\mathrm{cs}} S$, we know that $ls_i \vdash_{\mathrm{cs}} e_i$ for all $i \in 1..n$, $ls = ls_1 \uplus \ldots \uplus ls_n$, and $\forall q \in ls.\ \sigma(q) = \{\!|\ldots|\!\}^{\bullet}_{c_q}$. Also, assume that the reduction happens in thread k and $S' = \langle \sigma', e_1..e'_k..e_n \rangle$.

[RED SYNC]: In this case, $e_k = \mathcal{E}[\texttt{synchronized}\, p\ e]$ and $\sigma(p) = \{\!|\ldots|\!\}^{\circ}_{c_p}$. Also, $e'_k = \mathcal{E}[\texttt{in-sync}\ p\ e]$ and $\sigma' = \sigma[p \mapsto \{\!|\ldots|\!\}^{\bullet}_{c_p}]$. Therefore, $ls_i \cup \{p\} \vdash_{\mathrm{cs}} e'_k$, $ls' = ls \cup \{p\}$, and $\forall q \in ls'.\ \sigma'(q) = \{\!|\ldots|\!\}^{\bullet}_{c_q}$, and we may conclude $\vdash_{\mathrm{cs}} S'$ by rule [CS STATE].

[RED IN-SYNC]: In this case, $e_k = \mathcal{E}[\texttt{in-sync}\ p\ v]$ and $\sigma(p) = \{\!|\ldots|\!\}^{\bullet}_{c_p}$. Also, $e'_k = \mathcal{E}[v]$ and $\sigma' = \sigma[p \mapsto \{\!|\ldots|\!\}^{\circ}_{c_p}]$. Therefore, $ls_i \setminus \{p\} \vdash_{\mathrm{cs}} e'_k$, $ls' = ls \setminus \{p\}$, and $\forall q \in ls'.\ \sigma'(q) = \{\!|\ldots|\!\}^{\bullet}_{c_q}$, and we may conclude $\vdash_{\mathrm{cs}} S'$ by rule [CS STATE].

□

Before proving that a well-typed program state does not have conflicting accesses, we state the conditions under which an object will be accessed. In particular, we show that a non-final field will be accessed by a thread only when that thread is in a critical section on the lock protecting the field.

Lemma 16 (Field access). *Suppose $P \vdash S : E$ and $P; E; \emptyset \vdash e : t$ and e accesses $p.fn$. Then $P; E; \emptyset \vdash p : c$, and either:*

(1) $P; E \vdash \varsigma\texttt{this}.\ (t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c$ *and e is in a critical section on* $[p/\texttt{this}]l$, *or*

(2) $P; E \vdash \varsigma\texttt{this}.\ (\texttt{final}\ t\ fn\ \texttt{=}\ v) \in c$.

Proof. We examine two cases, starting with the case where $e = \mathcal{E}[p.fn]$. Using Lemma 3, we know that there is a derivation of $P; E; ls \vdash p.fn : s$ that ends with the application of a rule other than [exp sub] such that for all l in ls, e is in a critical section on l. Note that E has the form $c_1\ p_1, \ldots, c_n\ p_n$ since it is the environment used to type a store. This means that ls contains only addresses. There are two possible rules used for deducing $P; E; ls \vdash p.fn : s$:

[exp ref guarded]: For this case, it must be that $P; E; ls \vdash p : c$ and $P; E \vdash \varsigma\texttt{this}.\ (t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c$. Also, $P; E \vdash [p/\texttt{this}]l \in ls$. Hence e is in a critical section on $[p/\texttt{this}]l$.

[exp ref unguarded]: For this case, it must be that $P; E; ls \vdash p : c$ and $P; E \vdash \varsigma\texttt{this}.\ ([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v) \in c$. Lemma 9 indicates that $P; E \vdash [p/\texttt{this}]([\texttt{final}]_{\text{opt}}\ t\ fn\ \texttt{=}\ v)$. This could only be derived with [final field], and so fn must be declared as `final`.

In the case where $e = \mathcal{E}[p.fn\ \texttt{=}\ v]$, we may follow reasoning similar to the case for [exp ref guarded] above to conclude that $P; E \vdash \varsigma\texttt{this}.\ (t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c$ and that e is in a critical section on $[p/\texttt{this}]l$. □

Since a non-final field will be accessed only when the appropriate lock is held, and two different threads cannot hold a lock at the same time, well-formed program states cannot have conflicting accesses.

Lemma 17 (No conflicting accesses). *If $P \vdash S : E$ and $\vdash_{\text{cs}} S$ then S does not have conflicting accesses.*

Proof. Suppose $S = \langle \sigma, e_{1..n} \rangle$. Rule [state] requires that $P; E; \emptyset \vdash e_i : t_i$, and rule [cs state] requires that $ls = ls_1 \uplus \ldots \uplus ls_n$, where $ls_i \vdash_{\text{cs}} e_i$ for each thread e_i. Suppose S has conflicting accesses on $p.fn$. Then there exist distinct j and k, such that e_j and e_k both access some field $p.fn$ and at least one of the accesses is a write. According to Lemma 16, $P; E; \emptyset \vdash p : c$ and one of the following two cases holds:

$P; E \vdash \varsigma\texttt{this}.\ (t\ fn\ \texttt{=}\ v\ \texttt{guarded\_by}\ l) \in c$ and e_j, e_k are in critical sections on $[p/\texttt{this}]l$: Lemma 14 indicates that $[p/\texttt{this}]l \in ls_j$ and $[p/\texttt{this}]l \in ls_k$. However, this contradicts the assumption that $ls = ls_1 \uplus \ldots \uplus ls_n$. In other words, the sets of locks held by each thread are not disjoint, and a contradiction exists. Therefore, no such e_j and e_k exist.

$P; E \vdash \varsigma\texttt{this}.\ (\texttt{final}\ t\ fn\ \texttt{=}\ v) \in c$: In this case, neither access to $p.fn$ can be a write because fn is a final field. Therefore, e_j and e_k do not have conflicting accesses on $p.fn$.

□

We extend the previous lemma to cover all states reachable during the execution of a well-typed program.

Theorem 18 (Race-freedom). *If* $\vdash P : t$ *then* P *does not have a race condition.*

Proof. Let $P = \mathit{defn}^*\ e$ and $P \vdash \langle\emptyset,\ e\rangle \longmapsto^* S$. To have concluded $\vdash P : t$, rule [PROG] requires that $P;\emptyset;\emptyset \vdash e : t$, which is sufficient to derive $P \vdash \langle\emptyset, e\rangle : \emptyset$ with rule [STATE]. Using induction over the execution sequence and Lemma 13, we may conclude that that $P \vdash S : E$ for some E. Clearly, $\vdash_{\mathrm{cs}} \langle\emptyset,\ e\rangle$ holds, and $\vdash_{\mathrm{cs}} S$ follows from Lemma 15 using induction over the execution steps from $\langle\emptyset,\ e\rangle$ to S. By Lemma 17, S does not have conflicting accesses. This argument holds for any S reached during the execution of P, so P does not have a race condition. □

C. THREAD-LOCAL CLASSES

This section extends the type system presented in Appendix A with thread-local classes, and Appendix D sketches the corresponding extension to the correctness proof.

C.1 Syntax

First, we extend the grammar to include an optional `thread_local` modifier on class declarations and instantiations:

$$\begin{array}{rcl} \mathit{defn} & ::= & [\texttt{thread\_local}]_{\mathrm{opt}}\ \texttt{class}\ \mathit{cn}\langle \mathit{garg}^*\rangle\ \mathit{body} \\ \mathit{ci} & ::= & [\texttt{thread\_local}]_{\mathrm{opt}}\ \texttt{class}\ c\ \mathit{body} \end{array}$$

C.2 Semantics

We extend class instantiation to thread-local classes in a straightforward manner.

$\boxed{P \vdash_{\mathrm{inst}} ci}$

[INST LOCAL]
$$\frac{\texttt{thread\_local class}\ \mathit{cn}\langle \texttt{ghost}\ t_i\ {x_i}^{i\in 1..n}\rangle\ \mathit{body} \in P}{P \vdash_{\mathrm{inst}} \texttt{thread\_local class}\ \mathit{cn}\langle l_{1..n}\rangle\ [l_i/{x_i}^{i\in 1..n}]\mathit{body}}$$

Other semantic rules from Appendix A.2 that match class instantiations of the form `class` c ... are extended to match instantiations of the form `thread_local class` c ... as well. For example, the rule [LOOKUP] is extended as follows:

$\boxed{P; c \vdash_{\mathrm{dispatch}} \mathit{meth}}$

[LOOKUP]
$$\frac{P \vdash_{\mathrm{inst}} [\texttt{thread\_local}]_{\mathrm{opt}}\ \texttt{class}\ c\ \texttt{extends}\ c'\ \texttt{\{}\ldots\ \mathit{meth}\ \ldots\texttt{\}}}{P; c \vdash_{\mathrm{dispatch}} \mathit{meth}}$$

C.3 Type System

The following judgments are added to the system:

Judgment	Meaning
$P; E \vdash t\ \mathit{shared}$	values of type t can be shared between threads
$P; E \vdash \mathit{field}\ \mathit{shared}$	*field* has a shared type and is guarded by a lock or is final
$P; E \vdash \mathit{field}\ \mathit{local}$	*field* has no guarding lock
$P; E; c \vdash \mathit{meth}$	*meth* does not override a method from any thread-shared super class of c

The type rules for these judgments and modified type rules for earlier judgments are presented below. Rules with subscripted names like [CLASS$_2$] replace earlier rules with the same base name, like [CLASS].

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

[CLASS SHARED]
$$\frac{P; E \vdash c \qquad P \vdash_{\text{inst}} \texttt{class}\ c\ \mathit{body}}{P; E \vdash c\ \mathit{shared}}$$

[OBJECT SHARED]
$$\frac{P \vdash E}{P; E \vdash \texttt{Object}\ \mathit{shared}}$$

[INT SHARED]
$$\frac{P \vdash E}{P; E \vdash \texttt{int}\ \mathit{shared}}$$

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

[SHARED FIELD]
$$\frac{\begin{array}{c}\mathit{field} = [\texttt{final}]_{\text{opt}}\ t\ \mathit{fn}\ \texttt{=}\ v\ [\texttt{guarded\_by}\ l]_{\text{opt}} \\ P; E \vdash \mathit{field} \\ P; E \vdash t\ \mathit{shared}\end{array}}{P; E \vdash \mathit{field}\ \mathit{shared}}$$

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

[LOCAL FIELD]
$$\frac{P; E; \emptyset \vdash v : t}{P; E \vdash ([\texttt{final}]_{\text{opt}}\ t\ \mathit{fn}\ \texttt{=}\ v)\ \mathit{local}}$$

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

[CLASS$_2$]
$$\frac{\begin{array}{c}P; \emptyset \vdash t_i\ \mathit{shared} \quad \forall i \in 1..n \\ \mathit{garg}_i = \texttt{ghost}\ t_i\ x_i \quad \forall i \in 1..n \\ E = \mathit{garg}_{1..n}, \mathit{cn}\langle x_{1..n}\rangle\ \texttt{this} \\ P; \mathit{garg}_{1..n} \vdash c\ \mathit{shared} \\ P; E \vdash \mathit{field}_i\ \mathit{shared} \quad \forall i \in 1..j \\ P; E \vdash \mathit{meth}_i \quad \forall i \in 1..k\end{array}}{\begin{array}{l}P \vdash \texttt{class}\ \mathit{cn}\langle \mathit{garg}_{1..n}\rangle \\ \qquad \texttt{extends}\ c\ \texttt{\{}\ \mathit{field}_{1..j}\ \mathit{meth}_{1..k}\texttt{\}}\end{array}}$$

[LOCAL CLASS]
$$\frac{\begin{array}{c}P; \emptyset \vdash t_i\ \mathit{shared} \quad \forall i \in 1..n \\ \mathit{garg}_i = \texttt{ghost}\ t_i\ x_i \quad \forall i \in 1..n \\ E = \mathit{garg}_{1..n}, \mathit{cn}\langle x_{1..n}\rangle\ \texttt{this} \\ P; \mathit{garg}_{1..n} \vdash c \\ P; E \vdash \mathit{field}_i\ \mathit{local} \quad \forall i \in 1..j \\ P; E \vdash \mathit{meth}_i \quad \forall i \in 1..k \\ P; E; c \vdash \mathit{meth}_i \quad \forall i \in 1..k\end{array}}{\begin{array}{l}P \vdash \texttt{thread\_local class}\ \mathit{cn}\langle \mathit{garg}_{1..n}\rangle \\ \qquad \texttt{extends}\ c\ \texttt{\{}\ \mathit{field}_{1..j}\ \mathit{meth}_{1..k}\texttt{\}}\end{array}}$$

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

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

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

[EXP FORK$_2$]
$$\frac{\begin{array}{c}P; E; \mathit{ls} \vdash e : c \\ P; E \vdash c\ \mathit{shared} \\ P; E \vdash \varsigma\texttt{this}.\,(t\ \texttt{run()}\ \texttt{requires}\ \emptyset\ \texttt{\{}\ e'\ \texttt{\}}) \in c\end{array}}{P; E; \mathit{ls} \vdash e.\texttt{fork} : \texttt{int}}$$

[EXP ASSIGN UNGUARDED]
$$\frac{\begin{array}{c}P; E; \mathit{ls} \vdash e : c \\ P; E \vdash \varsigma\texttt{this}.\,(t\ \mathit{fn}\ \texttt{=}\ v) \in c \\ P; E; \mathit{ls} \vdash e' : [e/\texttt{this}]t\end{array}}{P; E; \mathit{ls} \vdash e.\mathit{fn}\ \texttt{=}\ e' : [e/\texttt{this}]t}$$

The typing rules from Appendix A.3 that match class declarations and instantiations are extended to match `thread_local` declarations and instantiations. Two rules in this category are the following:

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

[SUBTYPE CLASS]
$$\frac{\begin{array}{c} P; E \vdash c <: cn\langle l_{1..n}\rangle \\ \texttt{thread\_local class } cn\langle \texttt{ghost } t_i\ x_i^{\ i \in 1..n}\rangle \texttt{ extends } c'\ \ldots \in P \end{array}}{P; E \vdash c <: [l_i/x_i^{\ i\in 1..n}]c'}$$

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

[TYPE LOCAL C]
$$\frac{\begin{array}{c} \texttt{thread\_local class } cn\langle \texttt{ghost } t_i\ x_i^{\ i \in 1..n}\rangle\ body \in P \\ P; E \vdash_{\mathrm{final}} l_i : t_i \quad \forall i \in 1..n \end{array}}{P; E \vdash cn\langle l_{1..n}\rangle}$$

D. SOUNDNESS WITH THREAD-LOCAL CLASSES

The soundness proof in Appendix B can be extended to thread-local classes. Rather than repeat many of the details already presented, we give a brief, informal overview of the main technical issues involved with this extension.

The primary challenge is to ensure that two threads never have access to the thread-local part of an object, since those fields are accessed without synchronization. To do this, we require that only the thread that allocates a thread-local object has access to its thread-local fields. Other threads may access the shared part of the object because subsumption allows it to be treated as a thread-shared object. The typing rules prevent a thread from downcasting a thread-shared object to a thread-local subtype, ensuring that this type of sharing does not permit two threads to access the thread-local part of an object. The case in which both thread-local and thread-shared data is stored in the same object does not occur in similar systems [Grossman 2003], and a simpler proof strategy suffices for such systems.

We must change rule [STATE] in several substantial ways in order to preserve the necessary invariants. First, we type each thread i in a separate environment E_i, where E_i is derived from E, the environment constructed from the store. In other words, we have

$$P; E_i; \emptyset \vdash T_i : t_i$$

To construct E_i, we first partition the store into disjoint sets of addresses $\mathcal{L}_1, \ldots, \mathcal{L}_n$, and $\mathcal{G}$. The set $\mathcal{L}_i$ contains all objects treated as thread-local by thread i and the set $\mathcal{G}$ contains all thread-shared objects. We then require that the judgment $P \vdash E \triangleleft_{\mathcal{L}_i} E_i$ holds; this judgment is defined by the rule [TL ENV]:

[TL ENV]
$$\frac{\begin{array}{c} dom(E) = dom(E_i) \\ \forall p \in (\mathcal{L}_i \cup \mathcal{G}).\ E(p) = E_i(p) \\ \forall p \in dom(E) \setminus (\mathcal{L}_i \cup \mathcal{G}).\ E_i(p) \text{ is the closest ancestor of } E(p) \text{ with shared type} \end{array}}{P \vdash E \triangleleft_{\mathcal{L}_i} E_i}$$

Thus, any object treated as local by thread i is assigned the most specific type possible (its allocated type), and any other object is assigned the most specific shared type possible. Thread i will be unable to access any thread-local parts of the objects assigned shared types. In essence, using the most specific types allows us to show that if thread i reads from the store, the value obtained is consistent with E_i, and if thread i updates the store, the resulting store is consistent with E (and hence with E_j, for all j).

Since the sets of addresses $\mathcal{L}_i$ are disjoint, we can extend Lemma 17 to prove that two threads never have conflicting accesses on fields declared in thread-local classes. In order to ensure that this property is closed under reduction, we require that $\mathcal{L}_i$ be closed under thread-local field access. The following two hypotheses capture this requirement for thread i:

$$\begin{gathered}\forall p \in \mathrm{FP}(T_i).\ P; E_i \not\vdash E_i(p)\ \mathit{shared} \Rightarrow p \in \mathcal{L}_i \\ \forall p \in \mathcal{L}_i.\ \forall p'.\ p' \text{ is stored in a thread-local field of } p \Rightarrow p' \in \mathcal{L}_i\end{gathered}$$

where $\mathrm{FP}(T_i)$ is the set of all addresses occurring in the expression T_i. Thus, the new rule [STATE$_2$] is as follows:

[STATE$_2$]

$$\frac{\begin{gathered}P \vdash \sigma : E \\ |T| > 0 \\ P; E_i; \emptyset \vdash T_i : t_i \quad \forall i \in 1..|T| \\ \mathcal{L}_1, \ldots, \mathcal{L}_{|T|} \text{ are disjoint} \\ P \vdash E \lhd_{\mathcal{L}_i} E_i \quad \forall i \in 1..|T| \\ \forall p \in \mathrm{FP}(T_i).\ P; E_i \not\vdash E_i(p)\ \mathit{shared} \Rightarrow p \in \mathcal{L}_i \quad \forall i \in 1..|T| \\ \forall p \in \mathcal{L}_i.\ \forall p'.\ p' \text{ is stored in a thread-local field of } p \Rightarrow p' \in \mathcal{L}_i \quad \forall i \in 1..|T|\end{gathered}}{P \vdash \langle \sigma, T \rangle : E}$$

Lemmas 13, 15, and 17 and the Race-freedom Theorem (Theorem 18) can all be extended to cover thread-local classes using this new invariant.