Type Discovery for Parameterized Race-Free Java *

RAHUL AGARWAL
AMIT SASTURKAR

SCOTT D. STOLLER
Computer Science Dept., SUNY at Stony Brook, Stony Brook, NY 11794-4400

December 10, 2004

Abstract

Concurrent programs are notorious for containing data races that are difficult to reproduce and diagnose at run-time. This inspired the development of type systems that statically ensure the absence of data races. Boyapati and Rinard's Parameterized Race Free Java (PRFJ) is an extension of Java with such a type system. We give the first complete formal presentation of PRFJ; Boyapati and Rinard's paper gives only an informal sketch of an important part of the type system, namely, support for readonly objects and objects referenced by a unique pointer. We present a new method for producing the type annotations needed by the type checker to show that a program is race-free. This approach, called type discovery, uses a combination of run-time monitoring and static analysis to automatically obtain most of the annotations. We study the expressiveness of the type system and efficacy of type discovery on several programs. In our experiments, type discovery reduced the number of annotations that need to be supplied by the programmer to about 1.9 annotations/KLOC. In Boyapati and Rinard's experiments, the programmer needed to supply about 25 annotations/KLOC.

1 Introduction

Type systems are well established as an effective technique for ensuring at compile-time that programs are free from a wide variety of errors. New type systems are being developed by researchers at an impressive rate. Many of them are very elaborate and expressive.

Types provide valuable compile-time guarantees, but at a cost: the programmer must annotate the program with types. Annotating new code can be a significant burden on programmers. Annotating legacy code is a much greater burden, because of the vast quantity of legacy code, and because a programmer might need to spend a long time studying the legacy code before he or she understands the code well enough to annotate it.

Type inference reduces this burden by automatically determining types for some or all parts of the program. A type inference algorithm is *complete* if it can infer types for all typable programs. Unfortunately, complete type inference is impossible or infeasible for many expressive type systems. This motivates the development of incomplete type inference algorithms. These algorithms fall on a spectrum that embodies a

*This work was supported in part by NSF under Grants CCR-9876058 and CCR-0205376 and by ONR under Grants N00014-02-1-0363 and N00014-04-1-0722. Authors' Email: {ragarwal,amits,stoller}@cs.sunysb.edu Web: http://www.cs.sunysb.edu/~{ragarwal,amits,stoller}

trade-off between computational cost and power. Roughly speaking, we measure an algorithm's power by how many annotations the user must supply in order for the algorithm to successfully infer the remaining types for a program. For some potentially useful type systems, even incomplete algorithms designed to infer most types for most programs encountered in practice may have prohibitive exponential time complexity.

Traditional type inference is based on static analysis. A common approach is constraint-based type inference, which works by constructing a system of constraints that express relationships between the types of different parts of the program and then solving the resulting constraints.

We propose a novel approach to type inference. The main idea is to monitor executions of the program and then infer candidate types based on the observed behavior and the results of static analysis. We call this approach *type discovery*, to distinguish it from traditional type inference based solely on static analysis. Static analysis is used in type discovery to reduce run-time monitoring overhead and for intra-procedural type inference.

Type discovery is effective because, in our experiments, *monitoring a small number of simple executions of a program is usually sufficient to discover most or all of the types.* It is not necessary for the monitored executions collectively to achieve—or even come close to—full statement coverage. This is because simple and inexpensive intra-procedural static type inference algorithms exist for many powerful type systems. The hard problem is discovering type information for fields, method parameters, and return values; after that, intra-procedural static type inference can efficiently propagate the types throughout the code, including code in unexecuted branches.

This approach is not complete, because the process of generalizing from relationships between specific objects in a particular execution to static relationships between expressions or statements in the program is based in part on (incomplete) heuristics. Soundness is ensured by checking the discovered types with a type checker.

As a first step towards the empirical evaluation of the effectiveness of type discovery, we developed and implemented a type discovery algorithm for a type system for concurrent programs.

Concurrent programs are notorious for containing errors that are difficult to reproduce and diagnose at run-time. This inspired the development of type and effect systems (for brevity, we call them "type systems" hereafter) that statically ensure the absence of some common kinds of concurrent programming errors. Flanagan and Freund [FF00] developed a type system that ensures that a Java program is race-free, *i.e.*, contains no data races. A *data race* occurs when two threads concurrently access a shared variable and at least one of the accesses is a write. The resulting programming language (*i.e.*, Java with their extensions to the type system) is called *Race Free Java.* Boyapati and Rinard [BR01] modified and extended Flanagan and Freund's type system to make it more expressive. The resulting programming language is called *Parameterized Race Free Java* (PRFJ). [1]

This paper contains the first complete formal presentation of PRFJ. [BR01] contains only an informal sketch of an important part of the type system, namely, support for readonly objects and objects referenced by a unique pointer. Boyapati's thesis [Boy04] contains a separate type system for the `unique` and `readonly` properties, but does not combine it with the type system for race-freedom. As we will see in Section 4, combining them soundly involves some subtle special cases. We also study in detail the notion of uniqueness required to ensure absence of races.

Although the type systems is occasionally undesirably restrictive, (*i.e.*, the type checker produces warnings for some race-free programs), experience indicates that it is sufficiently expressive for many programs.

[1] Hereafter, we assume that programs contain all type information required by the standard Java type-checker, and we use the word "types" to refer only to the *additional* type information required by these extended type systems.

The cost of expressiveness is that type inference for Race Free Java and hence PRFJ is NP-complete [FFar].

Flanagan and Freund developed a simple and efficient type inference algorithm for a fragment of Race Free Java. Roughly speaking, it starts with a set of candidate types for each expression, runs the type checker, deletes some of the candidate types based on the errors (if any) reported by the type checker, and repeats this process until the type checker reports no errors [FF01]. However, this algorithm infers types only for a fragment of the type system (specifically, a fragment without external locks) with significantly reduced expressiveness.

Boyapati and Rinard [BR01] use carefully chosen defaults and intra-procedural type inference to reduce the annotation burden. The user provides type annotations on selected declarations of classes, fields, and methods, and selected object allocation sites (*i.e.*, calls to `new`). Default types are used for unannotated classes, fields, and methods. A simple constraint-based intra-procedural type inference algorithm is used to infer types for local variables and unannotated allocation sites. In their experiments with several small programs, users needed to supply about 25 annotations/KLOC [BR01].

We believe that type systems like PRFJ are a promising practical approach to verification of race-freedom for programs that use locks for synchronization, if the annotation burden can be reduced. The computational complexity of type inference for PRFJ motivated us to develop and implement a type discovery algorithm for PRFJ. The target program is instrumented by an automatic source-to-source transformation. The instrumented program writes relevant information (mainly information about which locks are held when various objects are accessed) to a log file. Analysis of the log, together with a simple static program analysis that identifies unique pointers, produces type annotations at selected program points. Boyapati and Rinard's simple intra-procedural type inference algorithm is then used to propagate the resulting types to other program points. This has the crucial effect of propagating type information into branches of the program that were not exercised in the monitored executions. Our experience, reported in detail in Section 7, is that type discovery significantly reduces the annotation burden, to about 1.9 annotations/KLOC on average.

One useful direction for future work is to consider partial typings, which ensure that parts of a program are free of races. This would be useful for programs with some races and for programs where a significant fraction of the methods are not invoked at all by available test suites, because type discovery will not produce complete typings for such programs. Another direction for future work is type discovery for other type systems, such as the type system for safe region-based memory management in [BSBR03].

2 Related Work

Type discovery is similar in spirit to Daikon [Ern00]. During execution of a program, Daikon evaluates a large syntactic class of predicates at specified program points and determines, for each of those program points, the subset of those predicates that always hold at that program point during the monitored executions. Among those predicates, those that satisfy some additional criteria are reported as candidate invariants. Daikon cannot infer PRFJ types, because the invariants expressed by PRFJ types are not expressible in Daikon's language for predicates. Daikon infers predicates that can be evaluated at a single program point. In contrast, a single PRFJ type annotation can express an invariant that applies to many program points. For example, if the declaration of a field f in a class C is annotated with the PRFJ type `self`, it means (roughly): for all instances o of class C, for all objects o' ever stored in field f of o, o' is protected by its own lock, *i.e.*, the built-in lock associated (by the Java language semantics) with o' is held whenever any field of o' is accessed. Such accesses may occur throughout the program.

Boyapati combines object encapsulation with unique pointers and readonly objects in the SafeJava type system [Boy04]. Boyapati's rules for uniqueness (in the context of object encapsulation) allow unique pointers to be transferred to other variables. He also allows temporary aliases to the unique pointers using a `borrow` construct. However, in the context of race-freedom, such temporary aliases may not always be safe, and may lead to races. To allow temporary aliases but only when safe to do so, we introduce the notion of active expressions and the `!e` ("not escaping") modifier. This makes our typing rules more complicated than the ones in [Boy04]. We also remain syntactically closer to Java and do not introduce new constructs like `borrow`. Boyapati and Rinard [BR01] give only informal sketches of the rules for `unique` and `readonly` types in the context of types for race-freedom and miss subtle issues, such as not allowing `readonly` objects to have `unique` fields, which (as discussed in Section 4) can make the type system unsound. Also, motivated by the experiments in Section 7, we allow instantiation of classes declared as having first owner `self` with `unique`.

Race Free Java [FF00] and Grossman's race-free type system for Cyclone [Gro03] lack uniqueness and escape information.

[RSH04] presents a dynamic type inference technique similar to ours for Race Free Java. Their algorithm can infer more context-sensitive typings than ours but does not handle the additional features of PRFJ.

Static analyses such as meta-compilation [HCXE02] and type qualifiers [FTA02] can check or verify simple lock-related properties of concurrent programs, *e.g.*, that a lock is not acquired twice by the same thread without an intervening release. Such analyses cannot easily be used to check more difficult properties such as race-freedom. RacerX uses data-flow analysis to find race conditions and deadlocks [EA03]. RacerX is useful for finding defects but is not a verification tool: to improve scalability and reduce false alarms, it relies on unsound heuristics and can miss some race conditions and deadlocks. Type discovery builds on a sound type system that guarantees absence of race conditions.

Tools that simply use run-time monitoring to detect indications of race conditions [SBN$^+$97, vPG01, JPr02, CLL$^+$02] in monitored executions of a program cannot guarantee absence of data races in other executions of the program. Type discovery can provide that guarantee. [CLL$^+$02] uses static analysis to show that some statements cannot be involved in data races and hence do not need to be instrumented. Their analysis is sound and can perhaps be regarded as equivalent to type inference for some race-free type system. However, there are many programs for which type discovery succeeds (showing that the entire program is race-free) while their analysis shows that some but not all statements in the program are race-free. For example, their analysis does not correlate memory accesses with the corresponding thread and synchronization information, so it is unable to show race-freedom for a statement that is executed by multiple threads but accesses a (different) unshared object in each thread. Also, their analysis cannot show race-freedom for a statement that is executed by multiple threads but accesses a thread-local object with an empty lockset in one of the threads and a shared object with proper synchronization in other thread. Their analysis does not recognize readonly objects and hence may report false alarms at accesses to readonly objects. Type discovery can prove absence of races in these cases.

An initial description of this work appeared in [AS04]. The main contributions of this paper relative to that one are the typing rules (in Section 4 and Appendix A), the technique for refining discovered types (in Section 5.8), and significantly more experimental results (in Section 7).

$$
\begin{array}{rcl}
P & ::= & \mathit{defn}^*\ \mathit{local}^*\ e \\
\mathit{defn} & ::= & \texttt{class}\ \mathit{cn}\langle \mathit{firstowner}\ [, f]^* \rangle\ [\texttt{where}\ [\mathit{wexpr}]^*]?\ \texttt{extends}\ c\ \mathit{body} \\
\mathit{firstowner} & ::= & f \mid \texttt{self} \mid \texttt{thisThread} \\
\mathit{wexpr} & ::= & f\ \neq \texttt{unique} \mid f\ \neq \texttt{readonly} \\
\mathit{body} & ::= & \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{field} & ::= & [\ \texttt{final}]?\ t\ fd = e \\
\mathit{meth} & ::= & t\ \mathit{mn}\ (\mathit{arg}^*)\ \texttt{requires}\ (e^*_{final})\ [\texttt{where}\ [\mathit{wexpr}]^*]?\ \{\ \mathit{local}^*\ e\ \} \\
\mathit{arg} & ::= & \mathit{argtype}\ x \\
\mathit{local} & ::= & \mathit{argtype}\ y \\
s,\ t & ::= & c_1 \mid \texttt{int} \\
c & ::= & \mathit{cn}\langle \mathit{firstowner}\ [, f]^* \rangle \mid \texttt{Object}\langle \mathit{firstowner} \rangle \\
c_1 & ::= & \mathit{cn}\langle \mathit{owner}\ [, \mathit{owner}]^* \rangle \mid \texttt{Object}\langle \mathit{owner} \rangle \\
\mathit{argtype} & ::= & \texttt{int} \mid \mathit{cn}\langle \mathit{owner}\ [, \mathit{owner}]^* \rangle\ [\mathit{mod}]? \mid \texttt{Object}\langle \mathit{owner} \rangle\ [\mathit{mod}]? \\
\mathit{owner} & ::= & f \mid \texttt{self} \mid \texttt{thisThread} \mid \texttt{readonly} \mid \texttt{unique} \mid e_{final} \\
e_{final} & ::= & e \\
e & ::= & \texttt{null} \mid n \mid \texttt{new}\ c_1 \mid \texttt{this} \mid e; e \mid x \mid x\ =\ e \mid \\
 & & e.fd \mid e.fd\ =\ e \mid e.mn([e]^*) \mid x\texttt{--} \mid e.fd\texttt{--} \mid \\
 & & \texttt{synchronized}\ (e)\ \{\ e\ \} \mid e.\texttt{fork} \\
 & & \\
n & \in & \texttt{integer constants} \\
\mathit{cn} & \in & \text{class names} \\
fd & \in & \text{field names} \\
\mathit{mn} & \in & \text{method names} \\
x, y & \in & \text{variable names} \\
f & \in & \text{formal owner names} \\
\mathit{mod} & \in & \text{type modifiers } (\texttt{!e},\texttt{!w},\texttt{!e!w})
\end{array}
$$

Figure 1: The grammar for mini PRFJ. $[X]?$ indicates 0 or 1 occurrences of X. $[X]^*$ indicates 0 or more occurrences of X separated by commas.

3 Overview of Parameterized Race Free Java (PRFJ)

This section presents the type system in the context of Concurrent Java, a multithreaded subset of Java [FF00]. We call the resulting language mini PRFJ. The grammar for this language is shown in Figure 1. e_{final} ranges over final expressions, which are expressions whose value does not change. Syntactically, final expressions are built from final variables (including `this`), final fields, and static final fields. [2] The expression e.`fork` evaluates e to an object o, spawns a new thread, invokes `o.run()` in the new thread and returns 0. We use Java-like syntactic sugar in examples. For example, a method declaration `synchronized` $t\ mn(arg^*)\ e$ abbreviates $t\ mn(arg^*)$ `synchronized (this)` $\{e\}$. Section 3.1 introduces the basic type system, which does not include `unique` and `readonly`. Section 3.2 introduces `unique` and `readonly`.

3.1 The basic type system

The PRFJ type system is based on the concept of object ownership. Each object is associated with an owner which is specified as part of its type. Each object is owned by another object or by special owner values `thisThread` or `self`. Since an object can be owned by another object which in turn could be owned by another object, the ownership relation can be regarded as a forest of rooted trees, where the roots may have

[2] In mini PRFJ, the only final variable is `this` (assignments to `this` are prohibited); in PRFJ, any parameter or local variable may be annotated as final.

```
public class Account<thisOwner> extends Object<thisOwner> {

  int balance = 0;

  int deposit(int x) requires (this) {
    this.balance = this.balance + x;
  }

  int run() requires() {
   synchronized(a2) { a2.deposit(10); }
  }

}

Account<thisThread> a1 = new Account<thisThread>;
a1.deposit(10);

Account<self> a2 = new Account<self>;
a2.fork;
a2.fork;
```

Figure 2: A sample PRFJ program.

self loops. The typing rules enforce the following synchronization discipline: to access an object o, a thread must hold the lock associated with the root r of the ownership tree containing o; r is called o's *root owner*. This implies that the program is race-free.

An object with root owner `thisThread` is unshared. Such objects can be accessed without synchronization. This is reflected in the type system by declaring that every thread implicitly holds the lock associated with `thisThread`. An object with owner `self` is owned by itself.

Every class in PRFJ is parameterized with one or more owner parameters. Parameterization allows the programmer to specify ownership information separately for each use of the class. The first parameter always specifies the owner of the `this` object. The remaining parameters, if any, may specify the owners of fields, method parameters, etc. The first parameter of a class can be a formal owner parameter or a special owner value; the remaining parameters must be formal owner parameters. When the class is used in the program, its formal owner parameters are instantiated with final expressions, special owner values, or owner parameters that are in scope at the use. Using final expressions to represent owners ensures that an object's owner does not change from one object to another.

Every method is annotated with a clause of the form "`requires` $(e_1, \ldots, e_n)$", where the e_i are final expressions. Locks on the root owners of the objects listed in a method's `requires` clause must be held at each call site.

To illustrate the basic PRFJ system, consider the program in Figure 2, copied from [BR01]. The program defines an `Account` class with a formal owner parameter, which is instantiated with `thisThread` for the thread-local instance stored in variable `a1` and with `self` for the instance stored in variable `a2`. The `deposit` method is annotated with "`requires (this)`". `a1`'s owner is `thisThread`, and every thread implicitly holds the imaginary `thisThread` lock, so this `requires` clause is satisfied at the call site involving `a1`. The `requires` clause is satisfied for the shared instance with rootowner `self`, because the call to `a2.deposit` occurs in the scope of `synchronized (a2)`. The program is well-typed and is therefore race free.

3.2 Unique and readonly

The full type system has two more special owner values: `unique` and `readonly`. An object with rootowner `readonly` cannot be updated. For an object o with rootowner `unique`, there is a single (unique) reference to o; only the thread currently holding that reference can access o. In both cases, the object cannot be involved in a data race, and no lock needs to be held when accessing o.

To support unique objects, additional annotations `!e` and `--` are needed. A `!e` annotation on a variable v of a method m means that, if v refers to an object o, then when m returns, no references to o created by m remain, and during execution of m, actions of m (including methods m calls) do not cause o to become reachable by other threads. For simplicity, the type system enforces a stronger requirement, namely, that m does not store references to o in any field of any object. The `--` annotation is used to transfer the unique reference to an object from one variable to another: $x = y$`--` is equivalent to $x = y$; y = `null`. Such an assignment may be accompanied by an ownership change from `unique` to another owner, as discussed in detail in Section 4.4. The operational semantics is that this expression transfers the reference from y to x, assigns null to y, and returns 0. If the entire expression returned the previous value of y, then there are two references to that value (e.g., $z = (x = y{-}{-}))$, violating the fact that x has owner `unique`. To avoid this, all assignment expressions ($x = e$ and $e'.fd = e$) return 0.

Our definition of uniqueness can now be stated more formally as: for an object o with rootowner `unique`, at most one field or variable without a `!e` annotation refers to o. Multiple variables with `!e` annotation may also refer to o, but at any given time, a single thread holds all the references to o.

To support `readonly` objects, an annotation `!w` is introduced. A `!w` annotation on a variable v of a method m means that, if v refers to an object o, then o is not updated through v or through any storage location to which a reference to o flows from v inside m or methods m calls. We refer to `!e` and `!w` annotations as type modifiers.

The use of `unique` and `readonly` is illustrated in Figure 3. The definition of class `ArrayList` is not shown, but it has two owner parameters: the first specifies the owner of the `ArrayList` itself, and the second specifies the owner of the objects stored in the `ArrayList`. The `Worker` constructor returns a unique reference to the newly allocated object. Thus, the main thread has unique references to the two instances of `Worker` until they are forked. After an instance of `Worker` is forked, it is accessed by only one thread and hence is unshared. Thus, the owner of each `Worker` object changes from `unique` to `thisThread`. The occurrences of `--` in the `main` method indicate that the main thread relinquishes its unique references to `m1` and `m2` when it forks them. These occurrences of `--` are required by the typing rules, and we consider them to be, in effect, type annotations. The operational semantics of e`--` is that it atomically returns the value of e and assigns `null` to e; thus, if e contains a unique reference to an object o, then e`--` returns a unique reference to o.

The two instances of `Worker` share a single `ArrayList` object a. The lock associated with a is held at every access to a, so a has owner `self`, and the first owner parameter of `ArrayList` is instantiated with `self`. Instances of the `Integer` class are immutable, so they have owner `readonly`. All objects stored in a have owner `readonly`, so the second owner parameter of `ArrayList` is instantiated with `readonly`.

The defaults in [BR01] are unable to determine the `unique`, `self`, and `readonly` owners used in this program. Our type discovery algorithm correctly discovers all of the types for this program.

```
public class Worker<thisThread> extends Object<thisThread> {

  public ArrayList<self,readonly> l;

  public Worker(ArrayList<self,readonly> l) {
    this.l = l;
  }

  public void run() {
   synchronized(this.l) { this.l.add(new Integer<readonly>(10)); }
  }

 public static void main(String args[]) {
  ArrayList<self,readonly> ls = new ArrayList<self,readonly>();
  Worker<unique> m1 = new Worker<unique>(ls);
  Worker<unique> m2 = new Worker<unique>(ls);

  m1--.fork;
  m2--.fork;
 }
}
```

Figure 3: A sample PRFJ program with unique and readonly variables.

Judgment	Meaning
$\vdash P : t$	program P is well-typed and its main expression has type t
$P \vdash \mathit{defn}$	defn is a well-formed class definition
$P \vdash E$	E is a well-formed typing environment
$P; E \vdash \mathit{meth}$	meth is a well-formed method
$P; E \vdash \mathit{field}$	field is a well-formed field
$P; E; cs \vdash t$	t is a well-formed type assuming classes in cs are well formed
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P \vdash \mathit{field} \in cn\langle f_1 \ldots f_n \rangle$	class cn with owner parameters $f_1 \ldots f_n$ declares or inherits $field$
$P \vdash \mathit{meth} \in cn\langle f_1 \ldots f_n \rangle$	class cn with owner parameters $f_1 \ldots f_n$ declares or inherits $meth$
$P; E \vdash_{final} e : t$	e is a final expression with type t
$P; E \vdash_{owner} o$	o is a well-formed owner
$P; E \vdash \mathrm{RootOwner}(e) = r$	r is the root owner of final expression e
$E \vdash \mathrm{mod}(e) = \mathit{mod}$	mod is the set of type modifiers associated with expression e
$P; E \vdash e : t$	expression e has type t provided all necessary locks are held
$P; E; ls \vdash e : t$	expression e has type t provided locks in ls are held

Table 1: Type judgments.

4 Typing rules

The type system is presented using the judgments in Table 1, which is based closely on [BR01]. Some important typing rules are presented in this section. The appendix contains the complete set of typing rules.

```
class C1<thisOwner> {
 int f;

 int run() {
   this.f = ..
  }
}

class C<self> {
...
 synchronized int m(C1<unique> y) {
   borrow (C1<f>  x  =y) { // x = y --;
    C1<f> z = x;
    z.fork ;
    x.f = ..;
   }                // y = x--
 }
}
```

Figure 4: A code fragment that shows that simple extensions to [Boy04]'s typing rules for uniqueness (in the context of object ownership) do not work in the context of race-freedom. This code fragment contains a race but would be typable in [Boy04] if `fork` was treated like other method calls.

4.1 Active expressions

Our type system allows temporary aliases to unique references in certain cases. Boyapati's rules for uniqueness (in the context of object encapsulation) also allow temporary aliases to the unique pointers using a `borrow` construct [Boy04]. However, simple extensions to [Boy04]'s typing rules for uniqueness (in the context of object ownership) do not work in the context of race-freedom. Figure 4 illustrates a code fragment that contains a race but would be typable if [Boy04]'s rules for uniqueness were naively extended to allow temporary aliases in the context of race-freedom. In the example, `x` temporarily borrows the pointer in `y`. `y` becomes unusable in the scope of `borrow` but `x` is allowed to create temporary aliases. `x` creates a temporary alias `z` which escapes. `z` escapes and thus the two temporary aliases to the unique reference can update the reference simultaneously leading to a data race. At the end of the scope of `borrow`, `x` is transferred back to `y`. Of course, this example is not typable in our race-free type system. In order to identify when a variable or field holding a unique reference needs to be nulled, we introduce the notion of active expressions. The typing rules (specifically [EXP REF] and [EXP VAR]) require that `--` is applied to each active occurrence of a variable or field with root owner `unique`.

Informally, an expression in a program P is active if a reference is created to the result of the expression. Formally, active is defined recursively over the structure of expressions (abstract syntax trees, abbreviated AST). The base cases are:

(b1) The body of a method (i.e., the root of the AST for the body) is active (because a reference to the return value is created in the calling context). [3]

(b2) Arguments (including `this`) to methods (i.e., roots of ASTs for all arguments of all method calls)

[3] For Java, this rule is replaced with: the argument of every `return` statement is active.

are active (because references from parameters to arguments are created), except arguments passed to parameters with non-escaping types.

(b3) Right sides of variable assignments and field assignments—including initializers— are active (i.e., in $x = e$ and $e'.fd = e$, the root of the AST for e is active) unless both left and right sides are non escaping (i.e., they have `!e` in their types) .

The "except" clause in (b2) and the "unless" clause in (b3) allow some temporary aliases to unique objects. Passing the same unique object as multiple arguments to the same method can create multiple temporary aliases to a unique object. For example, consider a method declaration `m(C<f1>!e x, C<f2>!e y) {...}`, and a program fragment `C<unique> o; m(o,o);`. In this example, passing the same unique object `o` as the first and second argument of method `m` creates temporary aliases (in `x` and `y`) to the same unique object.

The inductive cases of the definition propagate activeness down ASTs: if an expression is active, then subexpressions that may provide the return value of the expression are active.

(i1) If $e_1; e_2$ is active, then e_2 is active.

(i2) If `synchronized` (e_1) { e_2 } is active, then e_2 is active.

(i3) If e`--` is active, then e is active.

Note that the definition of active is top-down and therefore cannot easily be expressed in typing rules, which work bottom-up. The following program illustrates active expressions.

```
class C<thisOwner> {
  C<unique> f;

  m1(C<self> this) {
     synchronized (this) { this.f  = new C<unique>();}
     synchronized (this) { this.f-- ;}
  }

}
```

The body of `m1` is active by (b1). The second `synchronized` expression is active by (i1) and the occurrence of `this.f--` in it is active by (i2). The occurrence of `this.f` in it is active by (i3) and has owner `unique`, so `--` must be applied to it. `new C<unique>` is active by (b3), but it is not of the form v or $e.fd$, so `--` need not be applied. Without the concept of active expressions, it would be difficult to express the requirement that `--` needs to be applied to the second occurrence of `this.f`, since this depends on how the value is used multiple steps up the AST. For example, if the synchronized expression had been on the left of an assignment, `--` would not be needed.

By using the `--` construct, some checking is shifted from compile time to runtime; for example, the following program typechecks but throws a `NullPointerException`.

```
C<unique> x;
C<f1> y = x--;                // x becomes null
C<self> z = x--;              // x and z are null
synchronized (z) { z.f =...} // NullPointerException
```

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
$\mathit{OverridesOK}(P)$	Each overriding method has the same return type and parameter types (including the owner parameters) as the methods being overridden, except for the `this` argument where the class names differ but the owner parameters are same. The `requires` clause of each overriding method is the same or a subset of the `requires` clause of the methods being overridden. Each parameter of the overriding method has the same or a superset of the modifiers (`!e`,`!w`) of the corresponding parameter of the overridden method. The `where` clause of the overriding method is a subset of the `where` clause of the method being overridden.

Table 2: Predicates that need to hold for a well-typed program.

To allow (b2) to be checked without virtual method call resolution, and to ensure that the checks involving `!e` and `!w` in the [EXP INVOKE] rule are strong enough, we require that for every method m_{sub} that overrides another method m_{super}, each parameter of m_{sub} has the same or a superset of the modifiers (`!e`,`!w`) of the corresponding parameter of m_{super}. This requirement is part of $\mathit{OverridesOK}(P)$ (shown in Table 2), which is a premise of rule [PROG].

4.2 Well-formed types and class definitions

Well-formed types are obtained by instantiating the formal owner parameters of a class. Recall that the first owner in a class declaration may be a constant, specifically, `self` or `thisThread`. [BR01] shows that allowing `self` there is necessary for typability of self-synchronized classes (also called callee-synchronized, i.e., all methods except constructors acquire the lock on `this`). Allowing `thisThread` there is necessary for similar reasons. Our type system allows these constants to be "instantiated" with `unique`.

FO(t) return null if t is a primitive type else it returns the first owner of the class type.

[TYPE THREAD-LOCAL CLASS UNIQUE]

$$\frac{\begin{array}{c} P \vdash \texttt{class}\ cn\langle\texttt{thisThread}\ f_{2\ldots n}\rangle\ \texttt{extends}\ c\ \{\ [\texttt{final}]?\ t_i\ fd_i\ =\ e_i{}^{i\in\ 1\ldots k}\ \ldots\} \\ P;E \vdash_{\text{owner}} o_{2\ldots n} \\ t'_i\ =\ t_i[o_2/f_2]\ldots[o_n/f_n] \\ \text{isFinal}(fd_i)\ \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ t'_i \in cs\ \vee P;E;cs, cn\langle\texttt{unique}\ o_{2\ldots n}\rangle\ \vdash\ t'_i \end{array}}{P;E;cs \vdash cn\langle\texttt{unique}\ o_{2\ldots n}\rangle}$$

Allowing classes with first owner `thisThread` to be instantiated with first owner `unique` supports programs in which the thread allocating the object has a unique reference to it before passing its unique reference to some other thread, after which the object is thread-local. The example in Figure 3 illustrates this. The relevant rule is [TYPE THREAD-LOCAL CLASS UNIQUE].

[BR01] states that if a variable or field x is declared to be the unique pointer to an object, then there is no other variable or field that has a pointer to that object, and hence the object can be accessed safely

```
class C<COwner> {
 C1<unique> f;
 .....

 int run() {
   synchronized(this) {this.f.f1 =...};
 }

 m1(C<self> x) {
   x.fork;
   x.fork;
 }
}
```

```
class C<COwner> {
 C1<unique> f;
 .....

 int run() {
   this.f.f1 =...;
 }

 m2(C<readonly> x) {
   x.fork;
   x.fork;
 }
}
```

Figure 5: A program to illustrate two threads accessing the `unique` object. We assume class `C1` has a field `f1`. Right side shows that allowing `readonly` objects to have `unique` fields can lead to races.

without any synchronization. However, two threads may simultaneously access the unique pointer. The method `m1` in Figure 5 illustrates this: two threads have reference to variable `x`, hence both threads can access the object pointed to by `x.f`, even though `x.f` has a unique pointer to it. Note that `this.f` and `this.f.f1` are not active and hence `--` is not applied. In this example, the access to `f1` is safe, because the expression `this.f.f1` is protected by lock on `this`.

In contrast, method `m2` contains a race condition, since two threads simultaneously update `x.f.f1`. To make `m2` untypable, we do not allow `readonly` objects to have fields with owner `unique`. A similar example can be constructed to show that final fields with owner `unique` can be involved in races.

The rules to check well-formed instantiation of classes ([TYPE SELF CLASS], [TYPE THREAD-LOCAL CLASS], [TYPE SELF CLASS UNIQUE], [TYPE THREAD-LOCAL CLASS UNIQUE], [TYPE C]) prohibit final fields and fields in `readonly` objects from having owner `unique`. The rules allow any field of a class to be instantiated with owner `thisThread` only if the corresponding class is also instantiated with owner `thisThread`. The rules also check that field types are instantiated legally and hence need to handle mutually recursive classes (*e.g.*, `class C<f1,f2>` {`D<f1,f2> x ...`} `class D<f1,f2>` {`C<f1,f2> y ...`}) correctly. For this they keep track of "class set" *cs*, the set of classes whose field types have already been checked. The rule does not generate premises for instantiation of the fields in classes already in set *cs*.

Figure 6 illustrates why instantiation of `self` with `unique` (see rule [TYPE SELF CLASS UNIQUE] in the Appendix) is useful. Class `C` is self-synchronized and hence is typable only if its first owner is `self`, not a formal owner parameter, as explained in [BR01, Section 5.6]. The `init` method contains no synchronization and has an empty `requires` clause, so the access to `this.f` is typable only if invoked with a unique pointer to `this`. Thus, the owner of `this` must be made explicit and equal to `unique` in the declaration of `init`. The `init` method declares `this` to be `!e`, so there is still a unique pointer to `this` after `init` returns. Since the `this` parameter must sometimes be declared explicitly to show its owner and type modifiers (as in this example), our typing rules require, for simplicity, that `this` is explicitly declared as the first parameter of every method and constructor. Thus, our rule [CLASS] for well-formed class definitions, unlike [BR01]'s, does not add `this` to the typing environment in the premise that checks the method is well-formed. As syntactic sugar, if the declaration of `this` is omitted in a constructor or method of a class $cn\langle f_1, .., f_n\rangle$, we

```
class C<self> {
  C1<..> f;

  void init(C<unique>!e this) requires () {
    this.f = ...
  }

  synchronized int m() requires () {
   this.f = ...
  }

  void run() {
   m();
  }
}
```

```
C<unique> o = new C<unique>;
o.init();
C<self> tmp = o--;
tmp.fork;
tmp.fork;
```

Figure 6: PRFJ program to show that allowing instantiation of `self` with `unique` is useful. A typical use of class C is shown on the right.

```
class C<thisOwner,fOwner> {
D<fowner> f;
 }

class C1<thisOwner,cOwner,fOwner> {

 copy(C<cOwner,fOwner> c, C<cOwner,fOwner> c1) where fOwner != unique requires (c,c1) {
 c1.f =c.f;
  }
}
```

Figure 7: A program to illustrate the need for `where f != unique` clause.

add the declaration $cn\langle$`unique`$, f_2, ..., f_n\rangle$ `this` or $cn\langle f_1, ..., f_n\rangle$ `this`, respectively.

A `where` clause in a method declaration constrains the values of formal owner parameters. Constraints of the form `f` $\neq$ `unique` and `f` $\neq$ `readonly` are allowed.

Figure 7 illustrates why `where f` $\neq$ `unique` clauses are needed. Instantiation of `fOwner` with `unique` must be prohibited (by a `where` clause on method `copy` or class `C1`), because the method creates (in `c1.f`) an additional (not unique) reference to `c.f`.

To see why `where f` $\neq$ `readonly` clauses are needed, consider the program in Figure 8; class `C` is the same as in Figure 7. Instantiation of `fOwner` with `readonly` must be prohibited, because it would make `c.f` have owner `readonly`, and `c.f` gets updated.

We require that if m_1 overrides m_2, then m_1's `where` constraints are the same or a subset of m_2's `where` constraints. This requirement is a part of $OverridesOK(P)$ (shown in Table 2), which is a premise of rule [PROG].

```
class C1<thisOwner,cOwner,fOwner> {

 m(C<cOwner,fOwner> c ) where fOwner != readonly requires (c,c.f) {
  c.f.x = ...;
  }
}
```

Figure 8: A program to illustrate the need for `where f != readonly` clause.

4.3 Well-formed method

The [METHOD] rule typechecks the method body assuming the locks in the `requires` clause are held.

Method owner parameters are formal owner parameters that appear in a method declaration and not in the declaration of the enclosing class. To see why they are useful, consider the `print(Object)` method of the `PrintStream` class [BR01, Section 9]. If parameterized methods were not allowed, then all objects that can be printed by a single instance of `PrintStream` must have the same owner, which is undesirably restrictive. The [METHOD] rule adds ordinary parameters, method owner parameters, and `where` constraints to the typing environment. (Formal owner parameters of the class are added to the typing environment by the [CLASS] rule.) Note that in class declarations, only the first owner parameter can be a special owner; all other owner parameters must be formal owner parameters. In declarations of method parameters, special owners may appear in all positions.

[METHOD]

$$\frac{\begin{array}{c} P \vdash\ t\ mn(arg_{0..n})\ [\ \texttt{where}\ f'_1 \neq o_1, \ldots, f'_p \neq o_p\ ]?\ \texttt{requires}\ (e_{1..m})\ \{local_{1..l}\ e\}\ \in\ cn\langle f_{1..k}\rangle \\ \text{each } f'_i \text{ appears in some } arg_i \\ \text{each formal owner in } t \text{ appears in some } arg_i \\ arg_0 \text{ matches } cn\langle\ldots\rangle\ldots\ \texttt{this} \\ E' \ = \ E,\ \texttt{final}\ arg_0, \ldots, \texttt{final}\ arg_n \\ E'' \ = \ E' \ \cup\ \{\ \text{owner}_{formal}\ f\ \mid\ f \text{ appears in some } arg_i\ \}\ \cup\ \{\ \texttt{where}\ f'_1 \neq o_1, \ldots, f'_p \neq o_p\ \} \\ P; E'' \vdash_{\text{final}}\ e_i\ :\ t_i \\ P; E'' \vdash\ \text{RootOwner}(e_i)\ =\ r_i \\ P; E'', local_{1..l}; \texttt{thisThread}, r_{1..m} \vdash\ e\ :\ t \end{array}}{P; E \vdash\ t\ mn(arg_{0..n})\ [\ \texttt{where}\ f'_1 \neq o_1, \ldots, f'_p \neq o_p\ ]?\ \texttt{requires}\ (e_{1..m})\ \{local_{1..l}\ e\}}$$

4.4 Well-formed expressions

The [EXP VAR] rule for reading the value of a variable x requires that `--` is applied to x if the occurrence of x is an active expression and the first owner in the type of x is `unique` or a formal owner parameter f without a `where f` $\neq$ `unique` restriction. The rule uses a predicate possiblyUnique(t, E), which holds if the first owner of t is `unique` or a formal owner parameter f without a `where` $f \neq$ `unique` restriction in E.

[EXP VAR]

$$\frac{\begin{array}{c} P \vdash E \quad E = E_1,\ [\texttt{final}]?\ t\ mod\ x, E_2 \\ x \text{ is active} \wedge \text{possiblyUnique}(t, E) \Rightarrow \texttt{--} \text{ is applied to } x \end{array}}{P; E; ls \vdash\ x\ :\ t}$$

The rule [EXP VAR ASSIGN] for assignments to variables (including method parameters) checks that an expression with non-escaping type is assigned only to a variable with non-escaping type. Similarly, an expression whose type modifier contains a `!w` can be assigned only to a variable whose rootowner is `readonly` or whose type contains a `!w` modifier. The type judgment $E \vdash \mathrm{mod}(e) = \mathit{mod}$ means that mod is the set of type modifiers associated with the result of expression e. For example, `Object !e x`, `Object y` $\vdash$ mod(`synchronized (y) { x }`) = { `!e` }.

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} P; E \vdash\ x\ :\ t \\ P; E; ls \vdash\ e\ :\ t \\ E \vdash\ \mathrm{mod}(e) = \mathit{mod} \quad E \vdash\ \mathrm{mod}(x) = \mathit{mod}' \\ \texttt{!e}\ \in\ \mathit{mod}\ \Rightarrow\ \texttt{!e}\ \in\ \mathit{mod}' \\ \texttt{!w}\ \in\ \mathit{mod}\ \Rightarrow\ \texttt{!w}\ \in\ \mathit{mod}' \vee \mathrm{RootOwner}(x)\ =\ \texttt{readonly} \end{array}}{P; E; ls \vdash\ x\ =\ e\ :\ \texttt{int}}$$

During initialization of an object, there is usually a unique reference to the object, regardless of subsequent ownership. To accommodate this, our type system uses subtyping to allow the owner of an object to change from `unique` to any other owner. Note that this does not introduce the possibility of race conditions.

[EXP SUB UNIQUE]

$$\frac{\begin{array}{c} P; E; ls \vdash\ e : cn\langle \texttt{unique}\ o^* \rangle \\ P; E \vdash_{\mathrm{owner}}\ o_1 \end{array}}{P; E; ls \vdash\ e : cn\langle o_1\ o^* \rangle}$$

For a dereference $e.fd$, the rule [EXP REF] checks that either the root owner of e is in the current lockset ls or the root owner of e is `readonly` or `unique` (since no synchronization is needed to access objects owned by `readonly` or `unique`). As an exception, if the field is final, then the dereference is always allowed.

$t[\sigma]$ denotes the result of applying substitution σ to type t. Our definition of substitution is standard except that special owners in the domain of the substitution are ignored. For example, `C`⟨`thisThread,f`⟩ [`unique/thisThread`] [`self/f`] equals `C`⟨`thisThread,self`⟩, not `C`⟨`unique,self`⟩. To see why this is needed, consider the program fragment:

```
class C<self> {
 C1<self> fd;
}

C<unique> o;
... = o.fd;
```

If the [EXP REF] rule unconditionally substituted o_1 for f_1 in the type of the expression (as in [BR01]), we would conclude that the type of `o.fd` is `C1`⟨`self`⟩[`unique/self`], which is `C1`⟨`unique`⟩, which is not what we want.

The [TYPE THREAD-LOCAL CLASS UNIQUE] rule allows a class declared with first owner `thisThread` to be instantiated with first owner `unique`. The fields of such a class may have owner `thisThread`, and

```
class D<thisThread> {
  Object<thisThread> f1;
  ...
}

class C<thisThread> extends Thread<thisThread>{
  D<thisThread> f ;

  C(C<unique> this) {
    f = new D();
  }

  void run() {
    this.f.f1 = ...
  }

}

C<unique> c = new C<unique>;
D<thisThread> d = c.f;

c--.fork;
d.f1 =...;
```

Figure 9: A program to illustrate that `unique` objects with `thisThread` fields can be involved in a race.

allowing objects with owner `unique` to have `thisThread` fields can cause a race, as illustrated in Figure 9. In the example, an instance o of class `C` is created with owner `unique`. However, a thread-local instance o_1 of `D` gets a reference to the thread-local instance o_2 of `D` stored in o.`f`. After the unique reference to o is passed to the spawned thread, simultaneous dereference of object o_1 in the main thread and o.`f` in the spawned thread can cause a race. To avoid this, the type of an expression $e.fd$, where e has owner `unique` and fd has been declared with first owner `thisThread`, is treated as `unique` in [EXP REF] (the tfu function defined below accomplishes this). In the example in Figure 9 this means that, to make the program typable, `d = c.f` must be replaced with `d = c.f--`, so `this.f.f1` will cause a `NullPointerException` instead of a race.

If a field access expression $e.fd$ is active and its owner is `unique` or a formal owner parameter f without a `where` $f \neq$ `unique` restriction, `--` must be applied to $e.fd$, as discussed in Section 4.1. We define tfu(o, t) ("thisThread field of unique") as: if o = `unique` and t has the form $cn\langle \texttt{thisThread}, f_{2\ldots n}\rangle$ then $cn\langle \texttt{unique}, f_{2\ldots n}\rangle$ else t.

[EXP REF]

$$\frac{\begin{array}{c} P; E; ls \vdash \ e \ : \ cn\langle o_{1\ldots n}\rangle \\ P; E \vdash \ \text{RootOwner}(e) \ = \ r \\ (P \vdash \ (t \ fd) \ \in \ cn\langle f_{1\ldots n}\rangle \wedge r \ \in \ ls \cup \{\texttt{unique}, \texttt{readonly}\}) \vee (P \vdash \ (\texttt{final} \ t \ fd) \ \in \ cn\langle f_{1\ldots n}\rangle) \\ t' \ = \ \text{tfu}(o_1, t)[e/\texttt{this}][o_1/f_1][o_2/f_2]\ldots[o_n/f_n] \\ e.fd \text{ is active} \wedge \text{possiblyUnique}(t', E) \Rightarrow \texttt{--} \text{ is applied to } e.fd \end{array}}{P; E; ls \vdash \ e.fd : t'}$$

For an assignment $e.fd = e'$, the typing rule [EXP ASSIGN] checks that the root owner of e is in the current lockset or is `unique`. If e has root owner `readonly`, then typechecking fails (as it should), because ls never contains `readonly`. If e' is a variable then it should not have `!e` modifier, because this assignment makes e' escape to a field of e. If e is a variable, it should not have `!w` modifier. The typing rule also checks that fd is not final. The type of a `thisThread` field of a `unique` object is changed to `unique`, as in [EXP REF].

isROFormal(r) is true if r is of the form RO(e).

[EXP ASSIGN]

$$\frac{\begin{array}{c} P; E; ls \vdash\ e\ :\ cn\langle o_1 \ldots o_n\rangle \\ P \vdash\ (t\ fd)\ \in\ cn\langle f_1 \ldots f_n\rangle \text{ and } fd \text{ is not } \texttt{final} \\ P; E \vdash\ \mathrm{RootOwner}(e) = r \\ r\ \in\ ls\ \cup \{\texttt{unique}\}\ \wedge\ \mathrm{isROFormal}(r)\ \Rightarrow\ \texttt{where}\ r \neq \texttt{readonly} \in E \\ E \vdash\ \mathrm{mod}(e) = mod \quad E \vdash\ \mathrm{mod}(e') = mod' \\ \texttt{!w} \notin mod\ \wedge\ \texttt{!e} \notin mod' \\ t' =\ \mathrm{tfu}(o_1, t)[e/\texttt{this}][o_1/f_1][o_2/f_2] \ldots [o_n/f_n] \\ P; E; ls \vdash\ e'\ :\ t' \end{array}}{P; E; ls \vdash\ e.fd\ =\ e' : \texttt{int}}$$

The [EXP INVOKE] rule checks that all locks in the `requires` clause of the method declaration are held at the call site. The rule works as follows. Let t_p be the type of a method parameter (possibly `this`), and let t_a be the type of the corresponding argument. t_p and t_a must be related as follows for the invocation to type-check. (i) If t_p is a primitive type, then $t_a\ =\ t_p$. (ii) If t_p is a class type, then t_a is a class type for the same class. Let f be a formal owner parameter in t_p. If there is a constraint `where f` $\neq$ `unique` (or `where f` $\neq$ `readonly`) in the method declaration then the corresponding owner in t_a cannot be `unique` (or `readonly`). If f is a class parameter (i.e., f appears in the declaration of the class containing the method), then the corresponding owner in t_a is the same as the instantiation of f in the type of the target (receiver) object. If t_p contains special owners, then the corresponding owners in t_a must be the same special owners. (iii) If t_a has owner `unique` or modifier `!e`, then t_p has modifier `!e`. If t_a has owner `readonly` or modifier `!w`, then t_p has modifier `!w`.

In this rule, σ is the substitution that instantiates formal owner parameters. It is used to ensure that a formal owner parameter that occurs multiple times (in the parameter list and return type) is instantiated consistently.

In this rule j ranges over $0 \ldots k$; note that y_0 is `this`. FormalSatisfiesWhere(f, E, g) is defined as: `where` constraints on f in the environment E are a superset of the `where` constraints on g. formalOwners(t) $=$ formal owner parameters that appear in t. isFormal(f) is true iff f is a formal owner parameter.

[EXP INVOKE]

$$\frac{\begin{array}{c} P;E;ls \vdash\ e_j : t'_j \\ e_j \text{ does not contain } \texttt{synchronized} \\ P \vdash\ (t\ mn(t_j\ mod_j\ y_j{}^{j\in 0\ldots k})\ [\ \texttt{where}\ g_1 \neq o_1, \ldots, g_l \neq o_l\ ]?\ \texttt{requires}\ (e'_{1\ldots m})) \in cn\langle f_{1\ldots n}\rangle \\ t'_j\ =\ t_j[\sigma][e_0/\texttt{this}] \\ \mathrm{dom}(\sigma)\ =\ \mathrm{formalOwners}(t_{0\ldots k}) \\ g_i[\sigma]\ \neq\ o_i \wedge \mathrm{isFormal}(g_i[\sigma])\ \Rightarrow \mathrm{FormalSatisfiesWhere}(g_i[\sigma], E, g_i) \\ \forall f \in \mathrm{dom}(\sigma) :\ P;E \vdash_{\mathrm{owner}}\ f[\sigma] \\ \mathrm{possiblyUnique}(t'_j, E)\ \Rightarrow\ \texttt{!e} \in mod_j\ \vee\ e_j \text{ is of the form } e'{-}{-} \\ \mathrm{possiblyReadonly}(t'_j, E)\ \Rightarrow\ \texttt{!w} \in mod_j \\ E \vdash\ \mathrm{mod}(e_j) = mod'_j \\ mod_j\ \supseteq\ mod'_j \\ P;E \vdash\ \mathrm{RootOwner}(\mathrm{e'_i}[\mathrm{e_0}/\texttt{this}][\mathrm{e_1}/\mathrm{y_1}]\ldots[\mathrm{e_k}/\mathrm{y_k}])\ =\ \mathrm{r'_i} \\ r'_i\ \in\ ls \cup \{\texttt{unique}, \texttt{readonly}\} \end{array}}{P;E;ls \vdash\ e_0.mn(e_{1\ldots k})\ :\ t[\sigma][e_0/\texttt{this}]}$$

`synchronized` expressions in method arguments are prohibited; this implies that all locks held during the evaluation of the arguments are held during the invocation. This prevents some subtle races as illustrated in Figure 10. In the example, the instance o_c of class `C` has a unique field `f` that refers to an instance o_d of class `D`. o_c and o_d are passed as arguments to method `m`. The main thread spawns a new thread inside the method body. The main thread and spawned thread both access o_d`.f1`; the spawned thread holds the lock held on o_c, whereas the main thread accesses it without holding any locks, causing a race. This happens because the second argument to the method was a `synchronized` expression, and the lock on o_c needed to access o_c`.f` was held only during the evaluation of the argument and not during the method call.

The [EXP FORK] rule checks the invocation e.`run` with a lockset that contains `thisThread` and is otherwise empty. This check (the second premise of the rule) ensures that e has a `run` method, the owner of e is compatible with the type of the `this` parameter of the `run` method, and the `where` constraints on the `run` method are satisfied. The object to which e evaluates escapes to the new thread, so e should not have a `!e` modifier. Also the first owner in the type of e should not be `thisThread` unless e has first owner `unique`.

[EXP FORK]

$$\frac{\begin{array}{c} P;E;ls \vdash\ e\ :\ cn\langle o_{1\ldots n}\rangle \\ P;E;\texttt{thisThread} \vdash\ e.\texttt{run()} : \texttt{int} \\ E \vdash\ \mathrm{mod}(e) = mod \wedge \texttt{!e} \notin mod \\ o_1\ =\ \texttt{thisThread} \Rightarrow e \text{ has type } cn\langle \texttt{unique} \ldots f_n\rangle \end{array}}{P;E;ls \vdash\ e.\texttt{fork}\ :\ \texttt{int}}$$

4.5 Typing Rules for Additional Language Features

This section briefly sketches how several additional language features present in Java—specifically, static fields and static methods, arrays, and exceptions—are handled.

Static fields and static methods. The typing rule for dereferencing a static field of class cn checks whether cn (representing the lock associated with the class) is in the lockset. A static field must be protected by the lock associated with its class. There is currently no provision to specify a different owner, although this could be accommodated using `guarded_by` annotations [FF00]. The type of a static field cannot have owner `thisThread`, because static fields are accessible to all threads. A static method cannot use formal owner parameters of the class, because those parameters normally get instantiated based on the type of the target object, and static methods do not have a target object.

```
class C<self> {
 D<unique> f  = new D<unique>;

 int run() {
  synchronized(this) { this.f.f1 = ... ;}
 }

}


class M<thisThread> {

 void m(C<self> x, D<fOwner>!e y) requires (fOwner) {
   x.fork ;
   y.f1 = ... ;

 }
}

M<thisThread> m = new M<thisThread> ;
C<self> c = new C<self>;
m.m(c,synchronized(c) {c.f} );
```

Figure 10: A program to illustrate that `synchronized` expressions in method arguments can lead to races. Assume class `D` has a field `f1`.

Arrays. An example array type is `C<unique>[]<self>`. This type means that the owner of the array is `self`, and elements of the array are `unique` references to instances of `C`. Only arrays with owner `thisThread` may have elements with owner `thisThread`. `readonly` arrays cannot have `unique` elements for reasons similar to those discussed in Section 4.4. `--` must be applied to active array access expressions. For example, if `a` has the above array type, then `synchronized(a) {C<unique> c = a[0]--}` is typable.

Exceptions. To check `throws` clauses of methods, typing judgments must be extended to keep track of the set of exceptions that might be thrown by an expression. For example, the judgment $P; E; ls \vdash\ e : t, X$, where X is a set of exception types (i.e., types that extend `Throwable`) means that in the context described by $P; E; ls$, evaluation of e either returns a value of type t or throws an exception with a type in X. This requires changes to the typing rules [METHOD] (to deal with `throws` clauses), [EXP INVOKE] (to deal with `throws` clauses and `NullPointerException`), and [EXP REF] and [EXP ASSIGN] (to deal with `NullPointerException`). Most of the other rules change simply to propagate sets of exceptions. For simplicity, we require that all exception types are declared with exactly one owner parameter, which is a formal owner parameter, and that all `throw` statements throw unique references to exception objects. Note that implicit throws of `NullPointerException` and other run-time errors also throw unique references. Thus, the types of arguments to `catch` handlers and `throw` statements, and types in `throws` clauses are instantiated with owner `unique`.

5 Type Discovery for PRFJ

Our algorithm has three main steps.

First, static analysis is used to infer `unique` owners and `!e` annotations for fields, method parameters, return values, and local variables. We use static analysis for this to avoid the run-time overhead of tracking unique references.

Second, run-time information is used to infer owners for fields, method parameters and return values, and owners in class declarations.

Third, the intra-procedural type inference algorithm in [BR01, Section 7.1] is applied, to infer the types of local variables and allocation sites whose types have not already been determined.

Our algorithm does not infer which classes C need multiple owner parameters or how those parameters should be used in the declarations of fields and methods of C; we assume this information is given. This is acceptable because in our experiments, relatively few classes need multiple owner parameters, and most of the classes that do are library classes, which can be annotated once and re-used. Our algorithm does try to discover how to instantiate those owner parameters in all uses of C.

5.1 Inferring Unique Owners and `!e` and `--` Annotations

Aldrich et al.'s static analyses for `!e` (they call it `lent`) and uniqueness are flow-insensitive context-insensitive inter-procedural data-flow analyses whose running times are linear in the size of the program [AKC02]. To infer `!e` annotations, we use [AKC02]'s `lent` analysis as is. To infer `unique` annotations, we use the following modified version of [AKC02]'s uniqueness analysis.

First a directed value flow graph is constructed. The graph contains a node for each field, local variable and method parameter. We refer to the node corresponding to x as $\mathrm{N}(x)$. In addition, for each method m, for each call site k of m and each parameter p of m, there is a special node denoted $\mathrm{S}_k(p)$. We extend the domain of N (without changing the set of nodes) by defining: $\mathrm{N}(e.fd) \ = \ \mathrm{N}(fd)$. For each assignment of the form $e_l = e_r$ such that e_l and e_r are variables or field accesses, there is a directed edge from $\mathrm{N}(e_r)$ to $\mathrm{N}(e_l)$. For the k^{th} callsite of method m, for each argument e that is a variable or field access, there is a directed edge from $\mathrm{N}(x)$ to $\mathrm{S}_k(p)$, where p is the method parameter corresponding to that argument, and there is a directed edge from $\mathrm{N}(p)$ to $\mathrm{N}(x)$.

The algorithm labels each program variable and expression with `unique` or `non-unique`. We define an ordering on the labels: `unique` $\prec$ `non-unique`. Let $\mathrm{U}(n)$ denote the label of node n. Initially, everything is optimistically labeled `unique`, except for final fields and arguments and results of native methods. Final fields cannot have owner `unique` for reasons discussed in Section 4.4. Live variable analysis annotates the last use of each variable and field access. For each assignment of the form $e_l = e_r$ such that e_l and e_r are variable or field accesses, if e_r is a variable and last use of e_r, then $\mathrm{U}(\mathrm{N}(e_l)) \ = \ \mathrm{U}(\mathrm{N}(e_l)) \bigsqcup \mathrm{U}(\mathrm{N}(e_r))$, else $\mathrm{U}(\mathrm{N}(e_l)) \ =$ `non-unique` because it aliases e_r which is still live and if either e_l or e_r are not marked `!e`, then $\mathrm{U}(\mathrm{N}(e_r)) \ =$ `non-unique`.

For a call site k of the form $m(\ldots, e, \ldots)$, if e is a variable or field access and is not the last use of a variable, and if the corresponding parameter p of method m is not marked `!e`, then $\mathrm{U}(\mathrm{N}(e)) \ =$ `non-unique` and $\mathrm{U}(\mathrm{S}_k(p)) \ =$ `non-unique`. If this is the last use of a variable, then $\mathrm{U}(\mathrm{S}_k(p)) \ = \mathrm{U}(\mathrm{S}_k(p)) \bigsqcup \mathrm{U}(\mathrm{N}(e))$.

Starting from the non-unique base cases generated above, we propagate non-unique forward along the edges of the directed value flow graph. At the end, if any of the special nodes corresponding to a method parameter p is labeled `non-unique` then p is not given owner `unique`, even if $\mathrm{N}(p)$ is labeled `unique`, although p's owner may be a formal owner parameter that is instantiated with `unique` at some call sites.

This notion of uniqueness is subtly different than in [AKC02] which does not consider concurrency. [AKC02] always allows a `unique` reference to be assigned to a non-escaping variable. Our notion of uniqueness allows this only if the `unique` pointer is itself non-escaping. Figure 11 illustrates why this change is

```
class C<thisOwner> {
 C1<thisOwner> f;

 int run() {
   synchronized(this) { this.f = ...;}
  }
}

class C2<thisOwner> {
 ...

 int m() {
  C<unique> y;  // y is unique but is allowed to escape
  C<unique>!e x;  // x is not escaping
  C<self> t;

  x =y ; // a unique variable is allowed to be assigned to !e variable
       // allowed by [AKC02] even though y is not dead.

  t = y--; //ownership transfer
  t.fork;

  x.f =... // done without lock, race here
 }
```

Figure 11: Program illustrating that using [AKC02]'s notion of uniqueness would lead to races. `y` is `unique` according to [AKC02]'s analysis but not according to ours.

necessary. Since the typing rules check `unique` owners, unsoundness of this static uniqueness analysis would cause the type checker to emit type errors. We insert `--` on active expressions that are the last use of a `unique` variable or field.

5.2 Discovering Owners for Fields, Method Parameters and Return Values

Let d denote a field, method parameter, or method return type with class type (*i.e.*, not a primitive type). To infer the owner of d, we monitor accesses to a set $\mathrm{S}(d)$ of objects associated with d. If d is a field of some class C, $\mathrm{S}(d)$ contains objects stored in that field of instances of C. For a method parameter d, $\mathrm{S}(d)$ contains arguments passed through that parameter. For a method return type d, $\mathrm{S}(d)$ contains objects returned by the method. Let $\mathrm{FE}(d)$ denote the set of final expressions that are syntactically legal at the declaration of d.

After an object o is added to $\mathrm{S}(d)$, every access to o is intercepted and the following information is updated: $\mathrm{lkSet}(d, o)$, the set of locks that were held at every access to o after o was inserted in $\mathrm{S}(d)$ [SBN+97]; $\mathrm{rdOnly}(d, o)$, a boolean that is true iff no field of o was written (updated) after o was inserted in $\mathrm{S}(d)$; $\mathrm{shar}(d, o)$, a boolean that is true iff o is "shared", *i.e.*, multiple threads accessed non-final fields of o after o was inserted in $\mathrm{S}(d)$; $\mathrm{val}(o, e)$, the value of final expression e for object o (*e.g.*, if e is the final parameter `this`, then $\mathrm{val}(o, e)$ is o.`this`) for each $e \in \mathrm{FE}(d)$.

The owner of d is determined by the first applicable rule below.

1. If the type of d is an immutable class (*e.g.*, `String` or `Integer`), then $\mathrm{owner}(d)$=`readonly`.

2. If $(\forall o \in \mathrm{S}(d) : \neg \mathrm{shar}(d, o))$ and d is not a static field, then owner(d)=`thisThread`.

3. If $(\forall o \in \mathrm{S}(d) : \mathrm{rdOnly}(d, o))$, then owner($d$)=`readonly`.

4. If $(\forall o \in \mathrm{S}(d) : o \in \mathrm{lkSet}(d, o))$, then owner($d$)=`self`.

5. Let L(d) be the set of final expressions e in FE(d) such that, for each object o in S(d), val(o, e) is a lock that protects o; that is, $\mathrm{L}(d) = \{e \in \mathrm{FE}(d) \mid \forall o \in \mathrm{S}(d) : \mathrm{val}(o, e) \in \mathrm{lkSet}(d, o)\}$. If L($d$) is non-empty, select an arbitrary element e in L(d), and take owner(d) = e.

6. If d is a field or a method return type, take owner(d) to be the formal owner parameter `thisOwner`, which will also be used as the first owner parameter of the class containing d. If d is a method parameter, take owner(d) to be a fresh formal owner parameter.

To reduce the run-time overhead, in our implementation, S(d) contains only selected objects associated with d. This typically does not affect the discovered types. We currently use the following heuristics to restrict S(d). For a field d with type C, S(d) contains at most one object created at each allocation site for C. For a method parameter or return type d, S(d) contains at most one object per call site of the method. Also, we restrict FE(d) to contain only the values of final expressions of the form `this` or `this.`f, where f is a final field.

To infer the owner of elements of an array a, we monitor the objects stored in $a[0 \ldots 2]$ and use the rules above.

5.3 Discovering Values of Non-First Owner Parameters

If the type of d is a class C with multiple owner parameters, for each formal owner parameter P of C other than the first, we need to infer the value with which P should be instantiated for d, denoted $\mathrm{owner}_P(d)$. Let $\mathrm{S}_P(d)$ denote a set of objects o' associated with P for d and such that P denotes the first owner of o'. In particular, for each o in S(d): (1) for each field f of C such that the first owner of f is P (*i.e.*, `class C<..., P, ...> { ... D<P> f; ... }`), objects stored in $o.f$ are added to $\mathrm{S}_P(d)$; (2) for each parameter p of a method m of C such that the first owner of p is P (*i.e.*, `class C<..., P, ...> { ... m(..., D<P> p,...) ... }`), add to $\mathrm{S}_P(d)$ arguments passed through parameter p when $o.m$ is invoked; (3) for each method m of C whose return type has first owner P, add to $\mathrm{S}_P(d)$ objects returned from invocations of $o.m$. We instrument the program to monitor accesses to objects in $\mathrm{S}_P(d)$ and infer an owner based on that, just as in Section 5.2, and instantiate P with that owner for d. For efficiency, we may restrict $\mathrm{S}_P(d)$ to contain a subset of the objects described above.

5.4 Discovering Owners in Class Declarations

Let owner(C) denote the first owner parameter in the declaration of class C (*e.g.*, it denotes o in `class C<o, ...> {...}`). Let S(C) contain instances of C stored in or passed through fields, method parameters or method returns that have not been inferred to be `unique` by static uniqueness analysis. We monitor accesses to elements of S(C) as in Section 5.2 and then use the following rules to determine owner(C).

1. If C is a subclass of a class C' with owner(C')=`self`, then owner(C)=`self`.

2. If $\mathrm{S}(C) = \emptyset$ (*i.e.*, there are no instances of C), then owner(C)=`thisThread`.[4]

[4]For example, in many programs, the class containing the `main` method is never instantiated.

3. If $(\forall o \in \mathrm{S}(C) : \neg\mathrm{shar}(d, o))$, then owner$(C)$=`thisThread`.

4. If $(\forall o \in \mathrm{S}(C) : o \in \mathrm{lkSet}(d, o))$, then owner$(C)$=`self`.

5. owner(C)=`thisOwner`.

For efficiency, we restrict $\mathrm{S}(C)$ to contain only a few instances of C. Currently, we arbitrarily pick two fields or method parameters or method returns of type C and take $\mathrm{S}(C)$ to contain the objects stored in or passed through them. For a class whose first owner parameter is inferred to be a constant (*i.e.*, not a formal owner parameter), all occurrences of that class in the program are instantiated with that constant as the owner.

5.5 Discovering `where` Clauses

For each method m and each parameter p of m, if owner(p) (from Section 5.2) is a formal owner parameter f and $\mathrm{N}(p)$ is labeled `non-unique` (from Section 5.1), then a `where` $f \neq$ `unique` clause is added to the declaration of m.

To infer `where` $f \neq$ `readonly` annotations, the following information is also recorded at runtime: for each write to a field of a monitored object, the name of the executing method is recorded. For each method m and each parameter p of m, if m performs a write to an object in $\mathrm{S}(p)$, and owner(p) is a formal owner parameter f, then add a `where` $f \neq$ `readonly` to the method declaration. This technique is also used to infer `!w` annotations. However, the analysis above cannot discover `where` constraints for method m where the updates to the parameters with formal owners are not done in m but in methods that m calls. (*e.g.*, `m1(C<g> x)` $\{$`m2(x)`$\}$ `m2(C<f> y)` $\{$`y.f1 =...`$\}$). To discover constraints on the formal owner parameters of parameters of methods like `m1` above, a simple static analysis is done following the runtime analysis. All call sites of the methods with a `where` $f \neq$ `readonly` (like `m2` above) constraints (inferred from runtime analysis) are checked to see if the corresponding argument's owner is a formal owner g . If so, it is also constrained with a `where` $g \neq$ `readonly` clause. These constraints are then propagated up the call chain till a fixed point is reached.

5.6 Discovering `requires` Clauses

We infer `requires` clauses basically as in [BR01], except we use type discovery (in Section 5.4) instead of user input to determine which classes C have owner `thisThread`.

Each method declared in a class with owner `thisThread` is given an empty `requires` clause. For other classes, the `requires` clause for a method m contains all method parameters p (including the implicit `this` parameter) such that m contains a field access $p.f$ (for some field f) outside the scope of a `synchronized(`p`)` statement; as an exception, the `run()` method of a class that implements `Runnable` is given an empty `requires` clause, because a new thread holds no locks.

5.7 Static Intra-Procedural Type Inference

The last step is to infer the types of local variables and allocation sites whose types have not already been determined, using the intra-procedural type inference algorithm in [BR01, Section 7.1]. Each incomplete type (*i.e.*, each type for which the values of some owner parameters are undetermined) is filled out with an appropriate number of fresh distinct formal owner parameters. Equality constraints between owners are constructed in a straightforward way from each assignment statement and method invocation. [EXP

SUB UNIQUE] rule allows unique variables to transfer ownership to variables with any valid owner. To accommodate this, no equality constraints are generated when an active occurrence of a variable or field with owner `unique` is assigned to a variable or field or passed as an argument. The constraints are solved in almost linear time using the standard union-find algorithm [CLR90]. For each of the resulting equivalence classes E, if E contains one known owner o (*i.e.*, an owner other than the fresh formal owner parameters), then replace the fresh owner parameters in E with o. If E contains multiple known owners, then report failure. If E contains only fresh formal owner parameters, then replace them with `thisThread`. This last rule is a heuristic that is adequate for the examples we have seen. If necessary, we could instrument accesses to local variables and discover their owners directly.

5.8 Refining Discovered Types

The preceding steps produce an initial candidate typing for the program. That typing is now refined based on the error messages from the typechecker.

If the type of a field d is statically inferred to be `unique` but the typechecker reports a warning because the class containing d is instantiated as `readonly` (a combination prohibited by the type system), re-run type discovery and monitor field d as well, and then use the discovered type for d.

If the type of a field, method parameter or a method return type d is discovered to be a formal owner parameter and the typechecker reports a possible race on d, change the discovered owner to `self` and proceed with discovering owners in class declarations, discovering `requires` clauses and static intra-procedural type inference. If this results in fewer warnings from the typechecker, use `self` as the owner for d, otherwise revert to the original annotation. This refinement is beneficial for objects that are self-synchronized except for a few accesses protected by some other synchronization (*e.g.*, start/join synchronization). Other owners—notably `this`—could also be tried, and the owner leading to the fewest warnings selected; this was not necessary for the benchmarks we considered. Note that the tentative change to each error-producing annotation is evaluated independently; this works well in practice. In principle, better results could be obtained by evaluating sets of simultaneous changes to multiple annotations, but this would be expensive.

6 Implementation

A source-to-source transformation, implemented in the Kopi compiler [Kop02], instruments programs to record the information needed for type discovery. The transformation is parameterized by the set of classes for which types should be discovered.

All instances of `Thread` are replaced with `ThreadwithLockSet`, a new class that extends `Thread` and declares a field `locksHeld`. Synchronized statements and synchronized methods are instrumented to update `locksHeld` appropriately; a `try/finally` statement is used in the instrumentation to ensure that exceptions are handled correctly. For each field, method parameter and return type x being monitored, a distinct `IdentityHashMap` is added to the source code. The hashmap for x is a map from objects o in $S(x)$ to the information recorded for o, as described in Section 5, except the lockset. We store all locksets in a single `IdentityHashMap`. Thus, even if an object o appears in $S(x)$ for multiple x, we maintain a single lockset for o. Object allocation sites, method invocation sites, and field accesses are instrumented to update the hashmaps appropriately.

We do not instrument Java API classes. Instrumenting them creates new dependencies among the bootstrap classes, and since the VM loads those classes in a fixed order, an initialization error occurs. Instead, the following heuristic is used: for every method invoked on the library class, we assume all the parameters including the implicit `this` parameter are accessed. The locks held during these accesses is the set of locks held at the method call site and if the method is a synchronized method then the object invoking the method is also added to the set.

Boyapati and Rinard's typechecker for PRFJ is not available, so we implemented our own typechecker, by modifying rccjava, Flanagan and Freund's typechecker for Race Free Java [FF00].

Neither Kopi nor rccjava handles inner classes correctly, so for our experiments, we systematically manually transformed inner classes into regular classes.

7 Experience

We evaluated the expressiveness of our type system and the efficacy of our type discovery technique on eleven multi-threaded programs. The first five are multithreaded server programs used in [BR01]. The next three programs (elevator, tsp and hedc) were developed at ETH Zürich and used as benchmarks in [vPG01]. The last three programs (moldyn, raytracer and montecarlo) are part of the Java Grande Forum Benchmark Suite, available at http://www.epcc.ed.ac.uk/ .

A significant number of annotations in these programs are `unique` and `readonly`. Race-Free Java lacks a notion of uniqueness, and [FF00, FF01] rely on potentially unsafe escapes to reduce the number of resulting false alarms. For example, they give an option to the typechecker that effectively causes it to ignore accesses to `this` in constructors, and they explain that this is safe assuming constructors do not allow `this` to escape from the current thread. They claim that violations of this assumption are unlikely [FF00]. Using a simple static analysis, we found violations of this assumption in the Sun JDK 1.4 standard library and W3C's Jigsaw web server [Jig]; furthermore, the constructor accesses `this` after it escapes. This indicates the importance of extending the type system with `unique` types.

We also looked into W3C's Jigsaw web server [Jig] but we did not pursue it, because with the simple testcases we used, most methods did not get invoked at all. Type discovery is not effective in such cases.

7.1 Expressiveness of the type system

To evaluate the expressiveness of the type system, we compare the number of races reported by the type checker to the actual number of races in the program. The results are shown in Table 3 and discussed in detail below. For about 15.5 KLOC involving 568 fields, the typechecker reported races involving 97 fields whereas there are bugs on 7 fields and benign races on 7 fields. The typechecker reports which field accesses are involved in races, to help the user check whether the reported races are false alarms. 59 of the false alarms can easily be removed by adding a `final` modifier to some fields; this yields the results in the "modified program" column in Table 3. In some of these programs, static fields are used in a thread-local manner, causing false alarms, because static fields cannot have owner `thisThread` (see Section 4.5). Other false alarms result from use of synchronization constructs not analyzed by the type system, specifically barriers and `Thread.join`. A few false alarms are reported because different fields in a class have different protection mechanisms, which PRFJ does not allow. Besides this, a few lines of code needed to be changed in `elevator` and `hedc` to get them to typecheck.

The five multithreaded server examples (game, chat, phone, stock quote, and http) from [BR01] are race-free and are typable with no false alarms.

`elevator` is a simple discrete event simulator [vPG01]. Three lines of code had to be changed in the `elevator` example. The `Lift` class extends `Thread` and invokes the `start` method inside its constructor. Essentially, after the instance of `Lift` is started, it is accessed by only one thread (namely, itself) and hence

Program	LOC	# fields	# false alarms (original pgm)	# false alarms (modified pgm)	# bugs	# benign races
game	87	3	0	0	0	0
chat	308	7	0	0	0	0
phone	302	11	0	0	0	0
stock quote	242	6	0	0	0	0
http	563	19	0	0	0	0
elevator	523	21	1	1	0	0
tsp	706	36	14	4	4	3
hedc	7072	206	31	10	2	3
jgfutil	376	10	0	0	0	0
Barrier classes	134	3	2	0	0	1
moldyn	730	91	7	5	0	0
raytracer	1308	61	2	1	1	0
montecarlo	3198	94	26	3	0	0
total	15549	568	83	24	7	7

Table 3: Experimental results on the expressiveness of the type system

is unshared. Thus, the owner of `Lift` object changes from `unique` to `thisThread`. But `--` cannot be applied to `this` inside the constructor (because `this` is final), so we change the code to call start immediately after (instead of during) the call to the constructor. Then our system discovers that `l.start()` can be annotated as `l--.start()`, similar to the example in Figure 3. The one remaining false alarm is on the static field `Lift.count`, which is accessed only by the main thread, but our type system does not allow static fields to be owned by `thisThread`.

`tsp` solves the travelling salesman problem [vPG01]. We added final declarations to 6 static fields of `TspSolver` class and 4 static fields of `Tsp` class to eliminate false alarms on them.

`tsp` contains races that are defects (bugs) on the fields `TspSolver.MinTourLen`, `TourElement.last`, `TourElement.prefix`, and `TourElement.prefix_weight`.

There are benign races on `TspSolver.PrioQLast`, `PrioQElement.index`, and `PrioQElement.priority`. There are read accesses to these fields without synchronization in the `DumpPrioQ()` method, which dumps debugging information, so the races do not affect the execution of the program.

The main thread starts several worker threads, whose owners change from `unique` to `thisThread` when they are started, except that the main thread later calls `Thread.join` on the worker threads, causing our typechecker to produce a false alarm.

Warnings reported (of possible races) on fields `TourElement.conn` and `TourElement.lower_bound` are false alarms as these fields are accessed without any synchronization but only by the main thread before any new threads are created. There are also false alarms on two non-final static fields of the `TspSolver` class, namely, `TourStackTop` and `Done`, which are protected by two final fields of the `TspSolver` class. These false alarms could be eliminated by extending the type system to allow different owners for different fields, as in Race Free Java [FF00]. This extension is non-trivial because of its interactions with special owners `unique` and `thisThread`.

`hedc` [vPG01] is a meta-crawler for searching multiple Internet archives in parallel. We received an updated version that differs slightly from the one used in [vPG01]: some redundant code was eliminated, and a data race reported in [CLL+02] was fixed. We commented out some unreachable code. `hedc` uses part of Doug Lea's synchronization library; we treat it as part of the application (*i.e.*, we discover types for it). The typechecker identified 2 bugs and 3 benign races in the program. It also produced 31 false alarms.

Twenty one of them can easily be eliminated by making the corresponding fields final; this sometimes requires moving the initialization code for the field into an initializer for the field.

False alarms are reported because some static fields are accessed without synchronization before other threads have started or after they have terminated, or because a field is not updated when shared. Warnings on fields `poolSize_`, `maximumPoolSize_` and `minimumPoolSize_` of class `PooledExecutorWithInvalidate` reflect benign races; they are benign because each field is volatile and whenever the field is written into, it is protected by a lock.

A possible race is also reported on the `prev` field of class `Regexp`. Each `Regexp` object acts as a node in a doubly linked list and has `prev` and `next` fields which point to previous and next objects in the list. All instances of `Regexp` are thread-local except one, which is `readonly`. However, that readonly instance is used as the last node in every list, and this sharing pattern is not typable in our type system, causing this false alarm.

Two races reported by the typechecker are defects (bugs) in `hedc`. The first race is on the field `MetaSearchRequest.request`. It is a defect because `request` can be set to `null` in `MetaSearchResult.cancel()` just as the Worker thread completes execution and `request.countDownInterrupt()` is called in `MetaSearchResult.run()` leading to a `NullPointerException`. This defect was also discovered by the analysis in [CLL+02]. The typechecker also reported a previously unreported defect, which escaped detection by run-time monitoring [CLL+02], because it involves a code path that is not exercised by the test cases that accompany the program, which are also the test cases we used for type discovery. There is a race on the `valid` field of class `Task`. The `valid` field can be accessed in `Worker.run()` and `MetaSearchResult.cancel()` simultaneously. If a task is cancelled just after it is removed from the handoff queue and checked for validity, the `run()` method on the task object will be called even though the task has been cancelled. Tasks are placed in the handoff queue only if all worker threads are busy, which does not occur in the supplied test cases.

The Java Grande Forum benchmarks `raytracer`, `moldyn` and `montecarlo` use the `jgfutil` package that provides a `Timer` for measuring program execution time and an `Instrumentor` for manipulating Timers (starting, stopping, resetting etc). Since all three bechmarks use this package, we present results for it separately and do not count it in the statistics for these three benchmarks. Also, `raytracer` and `moldyn` use `Barrier` and `TournamentBarrier` classes for barrier-based synchronization. We separate the results for these two classes too.

`TournamentBarrier` implements barrier-based synchronization. Our typechecker reports races on fields `numThreads`, `isDone[]` and `maxBusyIter`. `maxBusyIter` and `numThreads` can be made final. There is a benign race on the `isDone` field.

`moldyn` simulates molecular dynamics. Each thread is assigned a unique identifier when it is started. The typechecker produces false alarms on the static fields `md.PARTSIZE`, `md.epot[]`, `md.vir[]`, `md.ek[]`, `md.interactions`, `md.interacts[]` and `JGFMolDynBench.nthreads`. There is no race on `PARTSIZE` or `nthreads` because the only write to it occurs in the main thread before any other threads are started; these fields can be made final after moving their initialization. The static int field `md.interactions` is accessed only in the thread with `id = 0`. Barriers are used to prevent races on the int arrays in the static fields `epot`, `vir`, `ek`, `interacts`. Our type system does not analyze use of barriers, so the typechecker produces false alarms on accesses to these arrays.

`raytracer` implements a parallel 3-dimensional ray tracing algorithm. The typechecker signals possible races on three static fields, `nthreads`, `staticnumobjects` and `checksum1` in `JGFRayTracerBench`. The warnings on `nthreads` and `staticnumobjects` are false alarms. For `nthreads`, the situation is exactly the same

Program	LOC	# annotations	# annotations changed manually
game	87	24	0
chat	308	51	0
phone	302	55	0
stock quote	242	50	0
http	563	127	0
elevator	523	56	0
tsp	706	61	0
hedc	7072	503	21
jgfutil	376	27	0
Barrier classes	134	3	1
moldyn	730	64	0
raytracer	1308	263	7
montecarlo	3198	230	0
total	15549	1514	29

Table 4: Experimental results on the efficacy of type discovery

as for `nthreads` in `moldyn`. The warning on `staticnumobjects` is a false alarm since `staticnumobjects` is written into only in the constructor `RayTracerRunner()`. `RayTracerRunner()` is called several times from the main thread sequentially. Thus the write accesses to `staticnumobjects` are sequential. The only read access to `staticnumobjects` occurs in the main thread after all the worker threads have terminated. The race on `JGFRayTracerBench.checksum1` is a bug, also noted in [OC03].

`montecarlo` is a financial simulation. The typechecker produces false alarms on 26 static fields. 23 of these fields can be made final after moving their initialization. The typechecker produces false alarms on `AppDemo.initAllTasks`, `AppDemo.JGFavgExpectedReturnRateMC` and objects that are stored in the static vector `AppDemo.tasks`. `initAllTasks` is initialized only once, before new threads are started. However, it cannot be made final since the only valid annotation for the field is `unique`, and our type system does not allow a final field to have a `unique` annotation. The accesses to `AppDemo.JGFavgExpectedReturnRateMC` occur after all threads have joined. The typechecker also produces a false alarm on accesses to `ToTask` objects that are stored in the static vector `tasks`. These objects are accessed by only one thread after initialization, but cannot be typed with owner `thisThread` because they are reachable from a static field.

7.2 Effectiveness of Type Discovery

To study the efficacy of type discovery, we study the number of annotations that were not discovered correctly (for the "modified pgm", described above) ignoring untypable parts of the programs, corresponding to the false alarms and actual races described above. Results are shown in Table 4 and discussed in the following paragraphs. In summary, for all of the benchmarks together, out of 1514 annotations, type discovery automatically produced 1485 (98%) of them. We manually changed 29 annotations, or 1.9 annots/KLOC. If `hedc` (a program with unusually complex synchronization) is excluded, we changed only 8 annotations in 8477 LOC, or 0.9 annotations/KLOC.

Boyapati and Rinard's experiments with the five multithreaded server examples (game, chat, phone, stock quote, and http) required the programmer to supply about 25 annots/KLOC [BR01]. Our algorithm discovers all of the annotations in these examples. We discover types only for the application (i.e., server) code; we assume PRFJ types are given for Java API classes used by the servers. Four of the server programs did not come with clients, so we wrote very simple clients for them. The examples use Boyapati's modified versions of Java API classes (*e.g.*, `Vector`), from which synchronization has been removed. The benefit is that synchronization can be omitted in contexts where it is not needed; the downside is that, when synchronization is necessary, it must be included explicitly in the application code. We also considered variant of these examples that use unmodified Java library classes. Our algorithm infers complete and correct typings for both variants with no annotations.

Note that it would be misleading to compare the 0.9 annots/KLOC required by our system across all benchmarks excluding hedc with the 25 annots/KLOC required in [BR01] for these 5 server benchmarks, because some of the other benchmarks that we use are more complicated and would probably require more than 25 annots/KLOC using the defaults in [BR01].

All the discovered annotations are also correct for `elevator`, `tsp`, `moldyn` and `montecarlo`.

In `raytracer` example, we changed some annotations from `readonly` and `unique` to `thisThread`. Some fields that should be owned by `thisThread` are discovered as `readonly` because they are accessed by two threads. Annotating the field as `readonly` creates problems as the corresponding class then needs to be parameterized with a formal owner which prohibits it from having `thisThread` fields. Changing the owner to `thisThread` works, since we allow `thisThread` to be instantiated with `unique`, and the field is `unique` while the first thread is accessing it. We changed some `unique` annotations for local variables and fields to `thisThread`, because instantiating the class C in the type of the local variable or field with `unique` causes the fields of C owned by `thisThread` to be instantiated with `unique`, which is incorrect.

In `hedc`, 21 annotations were inferred incorrectly. Almost all of the annotations that were inferred incorrectly are for fields that were not accessed in our traces and hence were given a default annotation of `thisThread`. However, the correct annotation is `self`. Also, we changed the owner in the return type of a method from `thisThread` to `readonly`, because the return value was assigned to a static field, and static fields can not have owner `thisThread`.

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

References

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proc. 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, 2002.

[AS04] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, January 2004.

[Boy04] Chandrasekar Boyapati. *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, Laboratory for Computer Science, MIT, February 2004. Available at http://www.eecs.umich.edu/~bchandra/.

[BR01] Chandrasekar Boyapati and Martin C. Rinard. A parameterized type system for race-free Java programs. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 56–69. ACM Press, 2001.

[BSBR03] Chandrasekhar Boyapati, Alexandru Salcianu, W. Beebee Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. pages 324–337, June 2003. Available at http://www.eecs.umich.edu/˜bchandra/.

[CLL+02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269. ACM Press, 2002.

[CLR90] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[EA03] Dawson R. Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. pages 237–252. ACM Press, October 2003. Available at http://www.stanford.edu/˜engler/.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 2000.

[FF00] Cormac Flanagan and Stephen Freund. Type-based race detection for Java. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232. ACM Press, 2000.

[FF01] Cormac Flanagan and Stephen Freund. Detecting race conditions in large programs. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96. ACM Press, June 2001.

[FFar] Corman Flanagan and Stephen Freund. Partial type and effect inference for Rcc/Java is NP-complete. Technical note, Williams College, to appear. Will be available at http://www.cs.williams.edu/˜freund/.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2002.

[Gro03] Dan Grossman. Type-safe multithreading in Cyclone. In *Proc. ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 13–25. ACM Press, 2003.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82. ACM Press, 2002.

[Jig] Jigsaw — W3C's Server. Available at http://www.w3.org/Jigsaw/.

[JPr02] JProbe 4.0, 2002. http://www.quest.com/jprobe/.

[Kop02] Kopi 2.1B, 2002. http://www.dms.at/kopi/.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proc. ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2003.

[RSH04] James Rose, Nikhil Swamy, and Michael Hicks. Dynamic inference of polymorphic lock types, April 2004. Submitted for publication.

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

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proc. 16th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 36(11) of *SIGPLAN Notices*, pages 70–82. ACM Press, October 2001.

A Typing Rules

[CLASS]

$$\frac{\begin{array}{c} g_i = \text{owner}_{\text{formal}}\ f_i \\ \text{if isFormal}(f_1) \text{ then } E = g_{1\ldots n} \text{ else } E = g_{2\ldots n} \\ P; E; \emptyset \vdash\ c \\ P; E, \texttt{final}\ cn\langle f_{1\ldots n}\rangle\ \texttt{this} \vdash\ field_i \\ P; E \vdash\ meth_i \end{array}}{P \vdash\ \texttt{class}\ cn\langle f_{1\ldots n}\rangle\ \texttt{extends}\ c\ \{field_{1\ldots j}\ meth_{1\ldots k}\}}$$

[PROG]

$$\frac{\begin{array}{c} ClassOnce(P)\ WFClasses(P)\ FieldsOnce(P) \\ MethodsOncePerClass(P)\ OverridesOK(P) \\ P = defn_{1\ldots n}\ local_{1\ldots l}\ e \\ P \vdash defn_i \quad P; local_{1\ldots l}; \texttt{thisThread} \vdash e : t \end{array}}{\vdash P : t}$$

[OWNER UNIQUE]

$$\frac{P \vdash E}{P; E \vdash_{\text{owner}} \texttt{unique}}$$

[OWNER READONLY]

$$\frac{P \vdash E}{P; E \vdash_{\text{owner}} \texttt{readonly}}$$

[OWNER THISTHREAD]

$$\frac{P \vdash E}{P; E \vdash_{\text{owner}} \texttt{thisThread}}$$

[OWNER SELF]

$$\frac{P \vdash E}{P; E \vdash_{\text{owner}} \texttt{self}}$$

[OWNER FINAL]

$$\frac{P; E \vdash_{\texttt{final}} e : t}{P; E \vdash_{\text{owner}} e}$$

[OWNER FORMAL]

$$\frac{\begin{array}{c} P \vdash E \\ E = E_1,\ \text{owner}_{\text{formal}}\ f,\ E_2 \end{array}}{P; E \vdash_{\text{owner}} f}$$

[ENV $\emptyset$]

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

[ENV OWNER]

$$\frac{\begin{array}{c} P \vdash\ E \\ f \notin \text{Dom}(E) \end{array}}{P \vdash\ E,\ \text{owner}_{\text{formal}}\ f}$$

[ENV X]

$$\frac{P; E; \emptyset \vdash\ t \qquad x \notin \text{Dom}(E)}{P \vdash E,\ [\texttt{final}]?\ t\ mod?\ x}$$

[ENV WHERE]

$$\frac{\begin{array}{c} P \vdash\ E \quad f \in \text{Dom}(E) \\ o \in\ \{\texttt{unique}, \texttt{readonly}\} \end{array}}{P \vdash\ E,\ \texttt{where}\ f \neq o}$$

[TYPE INT]

$$\frac{P \vdash\ E}{P; E; \emptyset \vdash\ \texttt{int}}$$

[TYPE OBJECT]

$$\frac{P; E \vdash_{\text{owner}}\ o}{P; E; \emptyset \vdash\ \texttt{Object}\langle o\rangle}$$

[TYPE SELF CLASS]

$$\frac{\begin{array}{c} P \vdash \texttt{class}\ cn\langle \texttt{self}\ f_{2\ldots n}\rangle\ \texttt{extends}\ c\ \{\ [\texttt{final}]?\ t_i\ fd_i\ =\ e_i^{\,i\in\ 1\ldots k} \ldots\} \\ P; E \vdash_{\text{owner}}\ o_{2\ldots n} \\ t'_i\ =\ t_i[o_2/f_2]\ldots[o_n/f_n] \\ \text{isFinal}(fd_i)\ \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ \text{FO}(t'_i)\ \neq\ \texttt{thisThread} \\ t'_i \in cs\ \vee P; E; cs, cn\langle \texttt{self}\ o_{2\ldots n}\rangle\ \vdash\ t'_i \end{array}}{P; E; cs \vdash cn\langle \texttt{self}\ o_{2\ldots n}\rangle}$$

[TYPE THREAD-LOCAL CLASS]

$$\frac{\begin{array}{c} P \vdash \texttt{class}\ cn\langle \texttt{thisThread}\ f_{2\ldots n}\rangle\ \texttt{extends}\ c\ \{\ [\texttt{final}]?\ t_i\ fd_i\ =\ e_i^{\,i\in\ 1\ldots k} \ldots\} \\ P; E \vdash_{\text{owner}}\ o_{2\ldots n} \\ t'_i\ =\ t_i[o_2/f_2]\ldots[o_n/f_n] \\ \text{isFinal}(fd_i)\ \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ t'_i \in cs\ \vee P; E; cs, cn\langle \texttt{thisThread}\ o_{2\ldots n}\rangle\ \vdash\ t'_i \end{array}}{P; E; cs \vdash cn\langle \texttt{thisThread}\ o_{2\ldots n}\rangle}$$

[TYPE SELF CLASS UNIQUE]

$$\frac{\begin{array}{c} P \vdash \texttt{class}\ cn\langle \texttt{self}\ f_{2\ldots n}\rangle\ \texttt{extends}\ c\ \{\ [\texttt{final}]?\ t_i\ fd_i\ =\ e_i^{\,i\in\ 1\ldots k} \ldots\} \\ P; E \vdash_{\text{owner}}\ o_{2\ldots n} \\ t'_i\ =\ t_i[o_2/f_2]\ldots[o_n/f_n] \\ \text{isFinal}(fd_i)\ \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ \text{FO}(t'_i)\ \neq\ \texttt{thisThread} \\ t'_i \in cs\ \vee P; E; cs, cn\langle \texttt{unique}\ o_{2\ldots n}\rangle\ \vdash\ t'_i \end{array}}{P; E; cs \vdash cn\langle \texttt{unique}\ o_{2\ldots n}\rangle}$$

[TYPE THREAD-LOCAL CLASS UNIQUE]

$$\frac{\begin{array}{c} P \vdash \texttt{class}\ cn\langle \texttt{thisThread}\ f_{2\ldots n}\rangle\ \texttt{extends}\ c\ \{\ [\texttt{final}]?\ t_i\ fd_i\ =\ e_i^{\,i\in\ 1\ldots k} \ldots\} \\ P; E \vdash_{\text{owner}}\ o_{2\ldots n} \\ t'_i\ =\ t_i[o_2/f_2]\ldots[o_n/f_n] \\ \text{isFinal}(fd_i)\ \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ t'_i \in cs\ \vee P; E; cs, cn\langle \texttt{unique}\ o_{2\ldots n}\rangle\ \vdash\ t'_i \end{array}}{P; E; cs \vdash cn\langle \texttt{unique}\ o_{2\ldots n}\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c} f_1 \text{ is a formal owner parameter} \\ P \vdash \texttt{class}\ cn\langle f_{1\ldots n}\rangle\ \texttt{extends}\ c\ \{\ [\texttt{final}]?\ t_i\ fd_i\ =\ e_i{}^{i\in\ 1\ldots k}\ \ldots\} \\ P; E \vdash_{\text{owner}} o_{1\ldots n} \\ t'_i\ =\ t_i[o_1/f_1]\ldots[o_n/f_n] \\ \text{isFinal}(fd_i)\ \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ o_1\ =\ \texttt{readonly} \Rightarrow \text{FO}(t'_i) \neq \texttt{unique} \\ \text{FO}(t'_i)\ =\ \texttt{thisThread} \Rightarrow o_1 = \texttt{thisThread} \\ t'_i \in cs\ \vee P; E; cs, cn\langle o_{1\ldots n}\rangle\ \vdash\ t'_i \end{array}}{P; E; cs \vdash cn\langle o_1 \ldots o_n\rangle}$$

[SUBTYPE REFL]

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

[SUBTYPE TRANS]

$$\frac{P; E \vdash\ t_1 <: t_2 \quad P; E \vdash t_2 <: t_3}{P; E \vdash\ t_1 <: t_3}$$

[SUBTYPE CLASS]

$$\frac{\begin{array}{c} P; E \vdash\ cn_1\langle o_{1\ldots n}\rangle \\ P \vdash\ \texttt{class}\ cn_1\langle f_{1\ldots n}\rangle\ \texttt{extends}\ cn_2\langle f_1\ o{*}\rangle\ \ldots \end{array}}{P; E \vdash\ cn_1\langle o_{1\ldots n}\rangle <: cn_2\langle f_1\ o{*}\rangle[o_1/f_1]\ldots[o_n/f_n]}$$

[FINAL VAR]

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

[FINAL REF]

$$\frac{\begin{array}{c} P \vdash\ (\texttt{final}\ t\ fd) \in\ cn\langle f_{1\ldots n}\rangle \\ P; E \vdash_{\texttt{final}}\ e : cn\langle o_{1\ldots n}\rangle \\ t'\ =\ \text{tfu}(o_1, t)[e/\texttt{this}][o_1/f_1][o_2/f_2]\ldots[o_n/f_n] \end{array}}{P; E \vdash_{\texttt{final}}\ e.fd : t'}$$

[ROOTOWNER UNIQUE]

$$\frac{P; E \vdash e : cn\langle\texttt{unique}\ o{*}\rangle\ \ |\ \texttt{Object}\langle\texttt{unique}\rangle}{P; E \vdash \text{RootOwner}(e) = \texttt{unique}}$$

[ROOTOWNER READONLY]

$$\frac{P; E \vdash e : cn\langle\texttt{readonly}\ o{*}\rangle\ \ |\ \texttt{Object}\langle\texttt{readonly}\rangle}{P; E \vdash \text{RootOwner}(e) = \texttt{readonly}}$$

[ROOTOWNER THISTHREAD]

$$\frac{P; E \vdash e : cn\langle\texttt{thisThread}\ o{*}\rangle\ \ |\ \texttt{Object}\langle\texttt{thisThread}\rangle}{P; E \vdash \text{RootOwner}(e) = \texttt{thisThread}}$$

[ROOTOWNER SELF]

$$\frac{P; E \vdash e : cn\langle\texttt{self}\ o{*}\rangle\ \ |\ \texttt{Object}\langle\texttt{self}\rangle}{P; E \vdash \text{RootOwner}(e) = \texttt{self}}$$

[ROOTOWNER FINAL TRANSITIVE]

$$\frac{\begin{array}{c} P; E \vdash e : cn\langle o_{1\ldots n}\rangle\ \ |\ \texttt{Object}\langle o_1\rangle \\ P; E \vdash_{\texttt{final}} o_1 : c_1 \\ P; E \vdash \text{RootOwner}(o_1) = r \end{array}}{P; E \vdash \text{RootOwner}(e) = r}$$

[ROOTOWNER FORMAL]

$$\frac{\begin{array}{c} P; E \vdash e : cn\langle o_{1\ldots n}\rangle\ \ |\ \texttt{Object}\langle o_1\rangle \\ E = E_1, \text{owner}_{\text{formal}} o_1, E_2 \end{array}}{P; E \vdash \text{RootOwner}(e) = \text{RO}(e)}$$

[FIELD INIT]

$$\frac{P; E; \texttt{thisThread} \vdash\ e : t}{P; E \vdash\ [\texttt{final}]?\ t\ fd = e}$$

[FIELD DECLARED]

$$\frac{P \vdash\ \texttt{class}\ cn\langle f_{1\ldots n}\rangle\ldots\{\ldots field\ldots\}}{P \vdash\ \ field\ \in\ cn\langle f_{1\ldots n}\rangle}$$

[FIELD INHERITED]

$$\frac{\begin{array}{c} P \vdash\ \ field\ \in\ cn\langle f_{1\ldots n}\rangle \\ P \vdash\ \texttt{class}\ cn'\langle g_{1\ldots m}\rangle\ \texttt{extends}\ cn\langle o_{1\ldots n}\rangle\ldots \end{array}}{P \vdash\ field[o_1/f_1]\ldots[o_n/f_n]\ \in cn'\langle g_{1\ldots m}\rangle}$$

[METHOD]

$$\frac{\begin{array}{c} P \vdash\ t\ mn(arg_{0..n})\ [\ \texttt{where}\ f'_1 \neq o_1, \ldots, f'_p \neq o_p\ ]?\ \texttt{requires}\ (e_{1..m})\ \{local_{1..l}\ e\}\ \in\ cn\langle f_{1..k}\rangle \\ \text{each } f'_i \text{ appears in some } arg_i \\ \text{each formal owner in } t \text{ appears in some } arg_i \\ arg_0 \text{ matches } cn\langle\ldots\rangle\ldots\ \texttt{this} \\ E'\ =\ E,\ \texttt{final}\ arg_0, \ldots, \texttt{final}\ arg_n \\ E''\ =\ E'\ \cup\ \{\ \text{owner}_{formal}\ f\ \mid\ f \text{ appears in some } arg_i\ \}\ \cup\ \{\ \texttt{where}\ f'_1 \neq o_1, \ldots, f'_p \neq o_p\ \} \\ P; E'' \vdash_{\text{final}}\ e_i\ :\ t_i \\ P; E'' \vdash\ \text{RootOwner}(e_i)\ =\ r_i \\ P; E'', local_{1..l}; \texttt{thisThread}, r_{1..m} \vdash\ e\ :\ t \end{array}}{P; E \vdash\ t\ mn(arg_{0..n})\ [\ \texttt{where}\ f'_1 \neq o_1, \ldots, f'_p \neq o_p\ ]?\ \texttt{requires}\ (e_{1..m})\ \{local_{1..l}\ e\}}$$

[METHOD DECLARED]

$$\frac{P \vdash\ \texttt{class}\ cn\langle f_{1\ldots n}\rangle\ldots\{\ldots meth\ldots\}}{P \vdash\ \ meth\ \in\ cn\langle f_{1\ldots n}\rangle}$$

[METHOD INHERITED]

$$\frac{\begin{array}{c} P \vdash\ \ meth\ \in\ cn\langle f_{1\ldots n}\rangle \\ P \vdash\ \texttt{class}\ cn'\langle g_{1\ldots m}\rangle\ \texttt{extends}\ cn\langle o_{1\ldots n}\rangle\ldots \end{array}}{P \vdash\ meth[o_1/f_1]\ldots[o_n/f_n]\ \in cn'\langle g_{1\ldots m}\rangle}$$

[MOD VAR]

$$\frac{E = E_1,\ [\texttt{final}]?\ t\ mod\ x, E_2}{E \vdash\ \text{mod}(x) = mod}$$

[MOD SEQ]

$$\frac{E \vdash\ \text{mod}(e_2) = mod}{E \vdash\ \text{mod}(e_1; e_2) = mod}$$

[MOD –]

$$\frac{E \vdash\ \text{mod}(e) = mod}{E \vdash\ \text{mod}(e{-}{-}) = mod}$$

[MOD SYNC]

$$\frac{E \vdash \ \mathrm{mod}(e_2) = mod}{E \vdash \ \mathrm{mod}(\texttt{synchronized}\ (e_1)\ \{e_2\}) = mod}$$

[MOD OTHER]

$$\frac{e \text{ is not a variable, sequence expression, } \texttt{--} \text{ expression or a synchronized expression.}}{E \vdash \ \mathrm{mod}(e) = \{\}}$$

[EXP TYPE]

$$\frac{\exists_{ls} \quad P; E; ls \vdash \ e : t}{P; E \vdash \ e : t}$$

[EXP SUB]

$$\frac{P; E; ls \vdash \ e : t' \quad P; E; ls \vdash \ t' <: t}{P; E; ls \vdash \ e : t}$$

[EXP SUB UNIQUE]

$$\frac{\begin{array}{c} P; E; ls \vdash \ e : cn\langle \texttt{unique}\ o^* \rangle \\ P; E \vdash_{\mathrm{owner}} \ o_1 \end{array}}{P; E; ls \vdash \ e : cn\langle o_1 \ o^* \rangle}$$

[EXP --]

$$\frac{\begin{array}{c} e \text{ is a variable or a field access} \\ P; E; ls \vdash \ e : cn\langle \texttt{unique}\ o^* \rangle \end{array}}{P; E; ls \vdash \ e{-}{-} : cn\langle \texttt{unique}\ o^* \rangle}$$

[EXP NEW]

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

[EXP SEQ]

$$\frac{P; E; ls \vdash \ e_1 : t_1 \qquad P; E; ls \vdash \ e_2 : t_2}{P; E; ls \vdash \ e_1; e_2 : t_2}$$

[EXP SYNC]

$$\frac{\begin{array}{c} P; E \vdash_{\texttt{final}} \ e_1 : t_1 \\ P; E; ls, e_1 \vdash \ e_2 : t_2 \end{array}}{P; E; ls \vdash \texttt{synchronized}\ (e_1)\ \{e_2\} : t_2}$$

[EXP VAR]

$$\frac{\begin{array}{c} P \vdash E \quad E = E_1,\ [\texttt{final}]?\ t\ mod\ x, E_2 \\ x \text{ is active} \wedge \text{possiblyUnique}(t, E) \Rightarrow \texttt{--} \text{ is applied to } x \end{array}}{P; E; ls \vdash \ x \ : \ t}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} P; E \vdash \ x \ : \ t \\ P; E; ls \vdash \ e \ : \ t \\ E \vdash \ \mathrm{mod}(e) = mod \quad E \vdash \ \mathrm{mod}(x) = mod' \\ \texttt{!e} \ \in \ mod \ \Rightarrow \ \texttt{!e} \ \in \ mod' \\ \texttt{!w} \ \in \ mod \ \Rightarrow \ \texttt{!w} \ \in \ mod' \vee \text{RootOwner}(x) \ = \ \texttt{readonly} \end{array}}{P; E; ls \vdash \ x \ = \ e \ : \ \texttt{int}}$$

[EXP REF]

$$\frac{\begin{array}{c} P; E; ls \vdash \ e \ : \ cn\langle o_{1\ldots n} \rangle \\ P; E \vdash \ \text{RootOwner}(e) \ = \ r \\ (P \vdash \ (t\ fd) \ \in \ cn\langle f_{1\ldots n} \rangle \wedge r \ \in \ ls \cup \{\texttt{unique}, \texttt{readonly}\}) \vee (P \vdash \ (\texttt{final}\ t\ fd) \ \in \ cn\langle f_{1\ldots n} \rangle) \\ t' \ = \ \text{tfu}(o_1, t)[e/\texttt{this}][o_1/f_1][o_2/f_2]\ldots[o_n/f_n] \\ e.fd \text{ is active} \wedge \text{possiblyUnique}(t', E) \Rightarrow \texttt{--} \text{ is applied to } e.fd \end{array}}{P; E; ls \vdash \ e.fd : t'}$$

[EXP ASSIGN]

$$\frac{\begin{array}{c} P; E; ls \vdash \ e \ : \ cn\langle o_1 \ldots o_n \rangle \\ P \vdash \ (t\ fd) \ \in \ cn\langle f_1 \ldots f_n \rangle \text{ and } fd \text{ is not } \texttt{final} \\ P; E \vdash \ \text{RootOwner}(e) = r \\ r \ \in \ ls \ \cup \{\texttt{unique}\} \ \wedge \ \text{isROFormal}(r) \ \Rightarrow \ \texttt{where}\ r \neq \texttt{readonly} \in E \\ E \vdash \ \mathrm{mod}(e) = mod \quad E \vdash \ \mathrm{mod}(e') = mod' \\ \texttt{!w} \notin mod \ \wedge \ \texttt{!e} \notin mod' \\ t' = \ \text{tfu}(o_1, t)[e/\texttt{this}][o_1/f_1][o_2/f_2]\ldots[o_n/f_n] \\ P; E; ls \vdash \ e' \ : \ t' \end{array}}{P; E; ls \vdash \ e.fd \ = \ e' : \texttt{int}}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P; E; ls \vdash \ e_j : t'_j \\ e_j \text{ does not contain } \texttt{synchronized} \\ P \vdash \ (t\ mn(t_j\ mod_j\ y_j{}^{j \in 0\ldots k})\ [\ \texttt{where}\ g_1 \neq o_1, \ldots, g_l \neq o_l\ ]?\ \texttt{requires}\ (e'_{1\ldots m})) \in cn\langle f_{1\ldots n} \rangle \\ t'_j \ = \ t_j[\sigma][e_0/\texttt{this}] \\ \mathrm{dom}(\sigma) \ = \ \text{formalOwners}(t_{0\ldots k}) \\ g_i[\sigma] \ \neq \ o_i \wedge \text{isFormal}(g_i[\sigma]) \ \Rightarrow \text{FormalSatisfiesWhere}(g_i[\sigma], E, g_i) \\ \forall f \in \mathrm{dom}(\sigma) : \ P; E \vdash_{\mathrm{owner}} \ f[\sigma] \\ \text{possiblyUnique}(t'_j, E) \ \Rightarrow \ \texttt{!e} \in mod_j \ \vee \ e_j \text{ is of the form } e'{-}{-} \\ \text{possiblyReadonly}(t'_j, E) \ \Rightarrow \ \texttt{!w} \in mod_j \\ E \vdash \ \mathrm{mod}(e_j) = mod'_j \\ mod_j \ \supseteq \ mod'_j \\ P; E \vdash \ \text{RootOwner}(\mathrm{e}'_\mathrm{i}[\mathrm{e}_0/\texttt{this}][\mathrm{e}_1/\mathrm{y}_1]\ldots[\mathrm{e}_\mathrm{k}/\mathrm{y}_\mathrm{k}]) \ = \ \mathrm{r}'_\mathrm{i} \\ r'_i \ \in \ ls \cup \{\texttt{unique}, \texttt{readonly}\} \end{array}}{P; E; ls \vdash \ e_0.mn(e_{1\ldots k}) \ : \ t[\sigma][e_0/\texttt{this}]}$$

[EXP FORK]

$$\frac{\begin{array}{c} P; E; ls \vdash \ e \ : \ cn\langle o_{1\ldots n} \rangle \\ P; E; \texttt{thisThread} \vdash \ e.\texttt{run()} : \texttt{int} \\ E \vdash \ \mathrm{mod}(e) = mod \wedge \texttt{!e} \notin mod \\ o_1 \ = \ \texttt{thisThread} \Rightarrow e \text{ has type } cn\langle \texttt{unique} \ldots f_n \rangle \end{array}}{P; E; ls \vdash \ e.\texttt{fork} \ : \ \texttt{int}}$$