A Type System for Preventing Data Races and Deadlocks in Java Programs

Chandrasekhar Boyapati Robert Lee Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139
{chandra,rhlee,rinard}@lcs.mit.edu

Abstract

This paper presents a new static type system for multi-threaded programs; well-typed programs in our system are guaranteed to be free of data races and deadlocks. Our type system allows programmers to partition the locks into a fixed number of equivalence classes and specify a partial order among the equivalence classes. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order.

Our system also allows programmers to use recursive tree-based data structures to describe the partial order. For example, programmers can specify that nodes in a tree must be locked in the *tree-order*. Our system allows mutations to the data structure that change the partial order at runtime. The type checker statically verifies that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. We do not know of any other sound static system for preventing deadlocks that allows changes to the partial order at runtime.

1 Introduction

The use of multiple threads of control is quickly becoming a mainstream programming practice. But interactions between threads can significantly complicate the software development process. Multithreaded programs typically synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to *data races* or *deadlocks*. A data race occurs when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write. A deadlock occurs when there is a cycle of the form: $\forall i \in \{0..n-1\}$, Thread_i holds Lock_i and Thread_i is waiting for $\text{Lock}_{(i+1) \bmod n}$. Synchronization errors in multithreaded programs are among the most difficult programming errors to detect, reproduce, and eliminate.

The research was supported in part by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-63513.

This paper presents a new static type system for multi-threaded programs; well-typed programs in our system are guaranteed to be free of data races and deadlocks. In recent previous work, we presented a static type system to prevent data races [3]. In this paper, we extend the race-free type system to prevent both data races and deadlocks. The basic idea behind our system is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. Our system allows programmers to specify this locking discipline in their programs. The resulting specifications take the form of type declarations.

1.1 Deadlock Freedom

To prevent deadlocks, programmers partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. Our type system allows programmers to write code that is polymorphic in lock levels. Programmers can specify a partial order among formal lock level parameters using `where` clauses. This is somewhat similar to the use of `where` clauses in [11, 24].

Our system also allows programmers to use recursive tree-based data structures to further order the locks that belong to the same lock level. For example, programmers can specify that nodes in a tree must be locked in the *tree-order*. Our system allows mutations to the data structure that change the partial order at runtime. The type checker uses an intra-procedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. We do not know of any other sound static system for preventing deadlocks that allows changes to the partial order at runtime.

1.2 Data Race Freedom

To prevent data races, programmers associate every object with a *protection mechanism* that ensures that accesses to the object never create data races. The protection mechanism of an object can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses, or that threads can safely access the object without synchronization because either 1) the object is immutable, 2) the object is accessible to a single thread, or 3) the variable contains the unique pointer to the object. Unique pointers are useful to support object migration between threads. The type checker statically verifies that a program uses objects only in accordance with their declared protection mechanisms.

Our type system is significantly more expressive than previously proposed type systems for preventing data races [15, 2]. In particular, our type system lets programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. We do this by introducing a way of parameterizing classes that lets programmers defer the protection mechanism decision from the time when a class is defined to the times when objects of that class are created.

1.3 Contributions

This paper makes the following contributions:

- **Static Type System to Prevent Deadlocks:** This paper presents a new static type system to prevent deadlocks in Java programs. Our system allows programmers to partition all the locks into a fixed number of lock levels and specify a partial order among the lock levels. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order.
- **Formal Rules for Type Checking:** To simplify the presentation of key ideas behind our approach, this paper formally presents our type system in the context of a core subset of Java called Concurrent Java[3, 15, 16]. Our implementation, however, works for the whole of the Java language.
- **Type Inference Algorithm:** Although our type system is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information that our system requires. Instead, we use a combination of intra-procedural type inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. Our approach permits separate compilation.
- **Lock Level Polymorphism:** Our type system allows programmers to write code where the exact levels of some locks are not known statically—only some ordering constraints among the unknown lock levels are known statically. Our system uses lock level polymorphism to support this kind of programming. Programmers can specify a partial order among formal lock level parameters using where clauses. This is somewhat similar to the use of where clauses in [11, 24].
- **Support for Condition Variables:** In addition to mutual exclusion locks, our type system prevents deadlocks in the presence of condition variables. Our system statically enforces the constraint that a thread can invoke e.wait only if the thread holds no locks other than the lock on e. Since a thread releases the lock on e on executing e.wait, the above constraint implies that any thread that is waiting on a condition variable holds no locks. This in turn implies that there cannot be a deadlock that involves a condition variable. Our system thus prevents the nested monitor problem [22].
- **Partial-Orders Based on Mutable Trees:** Our system also allows programmers to use recursive tree-based data structures to further order the locks that belong to the same lock level. Our system allows mutations to the trees that change the partial order at runtime. The type checker uses an intra-procedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks.
- **Partial-Orders Based on Monotonic DAGs:** Our system also allows programmers to use recursive DAG-based data structures to order the locks that belong to the same lock level. DAG edges cannot be modified once initialized. Only newly created nodes may be added to a DAG by initializing the newly created nodes to contain DAG edges to existing DAG nodes.
- **Implementation:** We have a prototype implementation of our type system. Our implementation handles all the features of the Java language including threads, constructors, arrays, exceptions, static fields, and runtime downcasts. We also modified some Java server programs and implemented them in our system. These programs exhibit a variety of sharing patterns. Our experience indicates that our system is sufficiently expressive and requires little programming overhead.

1.4 Outline

The rest of this paper is organized as follows. Section 2 introduces our type system using two examples. Section 3 describes a core subset of Java that we use to formally describe our type system. Our implementation, however, works for the whole of the Java language. Sections 4 and 5 present our basic type system that prevents both data races and deadlocks. Section 6 describes some important rules for type checking. The full set of typing rules are presented in the appendix. Section 7 contains our type inference algorithm that significantly reduces the programming overhead. Section 8 shows how our type system supports lock level polymorphism, while Section 9 shows how our type system prevents deadlocks in the presence of condition variables. Section 10 presents tree-based partial orders and Section 11 presents DAG-based partial orders. Section 12 describes our implementation. Section 13 presents related work and Section 14 concludes.

2 Examples

This section introduces our type system with two examples. The later sections explain our type system in greater detail.

2.1 Combined Account Example

Figure 1 presents an example program implemented in our type system. The program has an Account class and a CombinedAccount class. To prevent data races, programmers associate every object in our system with a *protection mechanism.* In the example, the CombinedAccount class is declared to be immutable. A CombinedAccount may not be modified after initialization. The Account class is generic—different Account objects may have different protection mechanisms.

```
class Account {
  int balance = 0;

  int balance()        accesses (this) { return balance; }
  void deposit(int x)  accesses (this) { balance += x; }
  void withdraw(int x) accesses (this) { balance -= x; }
}

class CombinedAccount<readonly> {
  LockLevel savingsLevel = new;
  LockLevel checkingLevel < savingsLevel;
  final Account<self:savingsLevel> savingsAccount
    = new Account;
  final Account<self:checkingLevel> checkingAccount
    = new Account;

  void transfer(int x) locks(savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        savingsAccount.withdraw(x);
        checkingAccount.deposit(x);
  }}}
  int creditCheck() locks(savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        return savingsAccount.balance() +
               checkingAccount.balance();
  }}}
  ...
}
```

Figure 1: Combined Account Example

```
class BalancedTree {
  LockLevel l = new;
  Node<self:l> root = new Node;
}

class Node<self:v> {
  tree Node<self:v> left;
  tree Node<self:v> right;

  //   this              this
  //   /  \              /  \
  // ...   x           ...   v
  //      / \    -->        / \
  //     v   y             u   x
  //    / \                   / \
  //   u   w                 w   y

  synchronized void rotateRight() locks(this) {
    final Node x = this.right; if (x == null) return;
    synchronized (x) {
      final Node v = x.left; if (v == null) return;
      synchronized (v) {
        final Node w = v.right;
        v.right = null;
        x.left = w;
        this.right = v;
        v.right = x;
  }}}
  ...
}
```

Figure 2: Tree Example

The CombinedAccount class contains two Account fields—savingsAccount and checkingAccount. The key word self indicates that these two Account objects are protected by their own locks. The type checker statically ensures that a thread holds the locks on these Account objects before accessing the Account objects.

To prevent deadlocks, programmers associate every lock in our system with a lock level. In the example, the CombinedAccount class declares two lock levels—savingsLevel and checkingLevel. Lock levels are purely compile-time entities—they are not preserved at runtime. In the example, checkingLevel is declared to rank lower than savingsLevel in the partial order of lock levels. The checkingAccount belongs to checkingLevel, while the savingsAccount belongs to savingsLevel. The type checker statically ensures that threads acquire these locks in the descending order of lock levels.

Methods in our system may contain accesses clauses to specify assumptions that hold at method boundaries. The methods of the Account class each have an accesses clause that specifies that the methods access the this Account object without synchronization. To prevent data races, our type checker requires that the callers of Account methods must hold the locks that protect the corresponding Account object before the callers can invoke any of the Account methods. Without the accesses clauses, the Account methods would not have been well-typed.

Methods in our system may also contain locks clauses. The methods of the CombinedAccount class contain a locks clause to indicate to callers that they may acquire locks that belong to lock levels savingsLevel or lower. To prevent deadlocks, the type checker statically ensures that callers of CombinedAccount methods only hold locks that are of greater lock levels than savingsLevel. Like the accesses clauses, the locks clauses are useful to enable separate compilation.

2.2 Tree Example

Figure 2 presents part of a BalancedTree implemented in our type system. A BalancedTree is a tree of Nodes. Every Node object is declared to be protected by its own lock. To prevent data races, the type checker statically ensures that a thread holds the lock on a Node object before accessing the Node object.

The Node class is parameterized by the formal lock level v. The Node class has two Node fields—left and a right. The Nodes left and right also belong to the same lock level v.

Our system also allows programmers to use recursive tree-based data structures to further order the locks that belong to the same lock level. In the example, the key word tree indicates that the Nodes left and right are ordered less than the this Node object in the partial order. To prevent deadlocks, the type checker statically verifies that the rotateRight method acquires the locks on Nodes this, x and v in the tree-order.

The rotateRight method in the example performs a standard rotation operation on the tree to restore the tree balance. The type checker uses an intra-procedural intra-loop flow-sensitive analysis to statically verify that the mutations do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks.

Our type system thus statically verifies the absence of both data races and deadlocks in the above examples.

$$\begin{array}{rcl}
P & ::= & \mathit{defn}^* \; e \\
\mathit{defn} & ::= & \mathsf{class}\ \mathit{cn}\ \mathsf{extends}\ c\ \mathit{body} \\
c & ::= & \mathit{cn} \mid \mathsf{Object} \\
\mathit{body} & ::= & \{\mathit{field}^*\ \mathit{meth}^*\} \\
\mathit{meth} & ::= & t\ \mathit{mn}(\mathit{arg}^*)\ \{e\} \\
\mathit{field} & ::= & [\mathsf{final}]_{\mathrm{opt}}\ t\ \mathit{fd} = e \\
\mathit{arg} & ::= & [\mathsf{final}]_{\mathrm{opt}}\ t\ x \\
t & ::= & c \mid \mathsf{int} \mid \mathsf{boolean} \\
 & & \\
e & ::= & \mathsf{new}\ c \mid x \mid x = e \mid e.\mathit{fd} \mid e.\mathit{fd} = e \mid e.\mathit{mn}(e^*) \mid \\
 & & e;e \mid \mathsf{let}\ (\mathit{arg} = e)\ \mathsf{in}\ \{e\} \mid \mathsf{if}\ (e)\ \mathsf{then}\ \{e\} \mid \\
 & & \mathsf{synchronized}\ (e)\ \mathsf{in}\ \{e\} \mid \mathsf{fork}\ (x^*)\ \{e\} \\
 & & \\
\mathit{cn} & \in & \text{class names} \\
\mathit{fd} & \in & \text{field names} \\
\mathit{mn} & \in & \text{method names} \\
x & \in & \text{variable names}
\end{array}$$

Figure 3: Grammar for Concurrent Java

3 Core Subset of Java

This section presents Concurrent Java [3, 15], a core subset of Java [17] with formal semantics. To simplify the presentation of key ideas behind our approach, we describe our type system formally in the context of Concurrent Java. Our implementation, however, works for the whole of the Java language. Concurrent Java is an extension to a sequential subset of Java known as Classic Java [16], and has much of the same type structure and semantics as Classic Java. Figure 3 shows the grammar for Concurrent Java.

Each object in Concurrent Java has an associated lock that has two states—locked and unlocked—and is initially unlocked. The expression fork(x*) {e} spawns a new thread with arguments (x*) to evaluate e. The evaluation is performed only for its effect; the result of e is never used. Note that the Java mechanism of staring threads using code of the form {Thread t=...; t.start();} can be expressed equivalently in Concurrent Java as {fork(t) {t.start();}}. The expression synchronized (e_1) in {e_2} works as in Java. e_1 should evaluate to an object. The evaluating thread holds the lock on object e_1 while evaluating e_2. The value of the synchronized expression is the result of e_2. While one thread holds a lock, any other thread that attempts to acquire the same lock blocks until the lock is released. A newly forked thread does not inherit locks held by its parent thread.

A Concurrent Java program is a sequence of class definitions followed by an initial expression. A predefined class Object is the root of the class hierarchy. Each variable and field declaration in Concurrent Java includes an initialization expression and an optional final modifier. If the modifier is present, then the variable or field cannot be updated after initialization. Other Concurrent Java constructs are similar to the corresponding constructs in Java.

4 Type System to Prevent Data Races

This section presents our type system for preventing data races in the context of Concurrent Java. Programmers associate every object with a *protection mechanism* that ensures that accesses to the object never create data races. Programmers specify the protection mechanism for each object as part of the type of the variables that refer to that object. The type can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses, or that threads can safely access the object without synchronization because either 1) the object is immutable, 2) the object is accessible to a single thread, or 3) the variable contains the unique pointer to the object. Unique pointers are useful to support object migration between threads. The type checker then uses these type specifications to statically verify that a program uses objects only in accordance with their declared protection mechanisms.

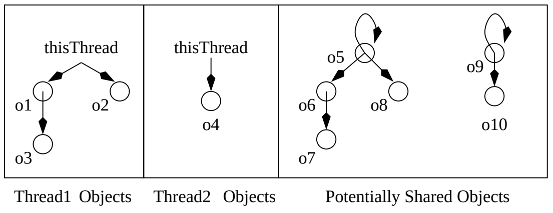

Figure 4: An Ownership Relation

1. The owner of an object does not change over time.
2. The ownership relation forms a forest of rooted trees, where the roots can have self loops.
3. The necessary and sufficient condition for a thread to access to an object is that the thread must hold the lock on the root of the ownership tree that the object belongs to.
4. Every thread implicitly holds the lock on the corresponding thisThread owner. A thread can therefore access any object owned by its corresponding thisThread owner without any synchronization.

Figure 5: Ownership Properties

This section only describes our basic type system that handles objects protected by mutual exclusion locks and thread-local objects that can be accessed without synchronization. Our race-free type system also supports unsynchronized accesses to immutable objects and objects with unique pointers that can migrate between threads. Our race-free type system is described in greater detail in [3]. The key to our basic race-free type system is the concept of object ownership. Every object in our system has an owner. An object can be owned by another object, by itself, or by a special per-thread owner called thisThread. Objects owned by thisThread, either directly or transitively, are local to the corresponding thread and cannot be accessed by any other thread. Figure 4 presents an example ownership relation. We draw an arrow from object x to object y in the figure if object x owns object y. Our type system statically verifies that a program respects the ownership properties shown in Figure 5.

Figure 6 shows how to obtain the grammar for Race-Free Java by extending the grammar for Concurrent Java. Figure 7 shows a TStack program in Race-Free Java. For simplicity, all the examples in this paper use an extended language that is syntactically closer to Java. A TStack is a stack

defn	::=	class $cn\langle owner\ formal*\rangle$ extends $c\ body$
c	::=	$cn\langle owner+\rangle$ | Object$\langle owner+\rangle$
owner	::=	*formal* | self | thisThread | e_{final}
meth	::=	$t\ mn(arg*)$ accesses $(e_{\text{final}}*)\ \{e\}$
e_{final}	::=	e
formal	::=	f
f	$\in$	owner names

Figure 6: Grammar Extensions for Race-Free Java

of T objects. A TStack is implemented using a linked list. A class definition in Race-Free Java is parameterized by a list of owners. This parameterization helps programmers write generic code to implement a class, then create different objects of the class that have different protection mechanisms. In Figure 7, the TStack class is parameterized by thisOwner and TOwner. thisOwner owns the this TStack object and TOwner owns the T objects contained in the TStack. In general, the first formal parameter of a class always owns the this object. In case of s1, the owner thisThread is used for both the parameters to instantiate the TStack class. This means that the main thread owns TStack s1 as well as all the T objects contained in the TStack. In case of s2, the main thread owns the TStack but the T objects contained in the TStack own themselves. The ownership relation for the TStack objects s1 and s2 is depicted in Figure 8 (assuming the stacks contains three elements each). This example illustrates how different TStacks with different protection mechanisms can be created from the same TStack implementation.

In Race-Free Java, methods can contain accesses clauses to specify the assumptions that hold at method boundaries. Methods specify the objects they access that they assume are protected by externally acquired locks. Callers are required to hold the locks on the root owners of the objects specified in the accesses clause before they invoke a method. In the example, the value and next methods in the TNode class assume that the callers hold the lock on the root owner of the this TNode object. Without the accesses clause, the value and next methods would not have been well-typed.

5 Type System to Prevent Deadlocks

This section presents our type system for preventing both data races and deadlocks in the context of Concurrent Java. To prevent deadlocks, programmers specify a partial order among all the locks. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. This section only describes our basic type system that allows programmers to partition the locks into a fixed number of equivalence classes and specify a partial order among the equivalence classes. Our system also allows programmers to use recursive tree-based data structures to describe the partial order—we describe how our type system handles tree-based partial orders in Section 10.

Figure 9 describes how to obtain the grammar for Deadlock-Free Java by extending the grammar for Race-Free Java. We call the resulting language Safe Concurrent Java. Safe Concurrent Java allows programmers to define lock levels in class definitions. A lock level is like a static field in Java—

```
// thisOwner owns the TStack object
// TOwner owns the T objects in the stack.

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

  T<TOwner> pop() accesses (this) {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
  ...
}
class TNode<thisOwner, TOwner> {
  T<TOwner> value;
  TNode<thisOwner, TOwner> next;

  T<TOwner> value() accesses (this) {
    return value;
  }
  TNode<thisOwner, TOwner> next() accesses (this) {
    return next;
  }
  ...
}
class T<thisOwner> { int x=0; }

TStack<thisThread, thisThread> s1 =
  new TStack<thisThread, thisThread>;
TStack<thisThread, self>       s2 =
  new TStack<thisThread, self>;
```

Figure 7: Stack of T Objects in Race-Free Java

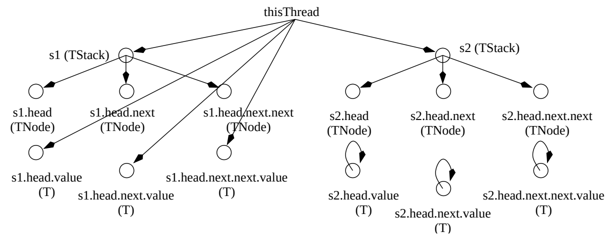

Figure 8: Ownership Relation for TStacks s1 and s2

a lock level is a per-class entity rather than a per-object entity. But unlike static fields in Java, lock levels are used only for compile-time type checking and are not preserved at runtime. Programmers can specify a partial order among the lock levels using the < and > syntax in the lock level declarations. Since a program has a fixed number of lock levels, our type checker can statically verify that the lock levels do indeed form a partial order. Every lock in Safe Concurrent Java belongs to some lock level. Note that the set of locks in Race-Free Java is exactly the set of objects that are the roots of ownership trees. A lock is, therefore, an object that has self as its first owner. In Safe Concurrent Java, every self owner is augmented with the lock level that the corresponding lock belongs to. The properties of our lock levels are summarized in Figure 10.

In the example shown in Figure 1, the CombinedAccount class defines two lock levels—savingsLevel and checkingLevel. checkingLevel is declared to be less than savingsLevel. A CombinedAccount contains a savingsAccount and a checkingAccount. These objects have self as their first owners—these objects are therefore locks. The savingsAccount is declared to

$$\begin{array}{rcl}
\textit{body} & ::= & \{\textit{level* field* meth*}\} \\
\textit{level} & ::= & \textsf{LockLevel}\ l = \textsf{new} \mid \textsf{LockLevel}\ l < \textit{cn.l*} > \textit{cn.l*} \\
\textit{owner} & ::= & \textit{formal} \mid \textsf{self}{:}\textit{cn.l} \mid \textsf{thisThread} \mid e_{\text{final}} \\
\textit{meth} & ::= & t\ \textit{mn}(\textit{arg*})\ \textsf{accesses}\ (e_{\text{final}}\textit{*})\ \textit{locksclause}\ \{e\} \\
\textit{locksclause} & ::= & \textsf{locks}\ (\textit{cn.l* lock*}) \\
\textit{lock} & ::= & e_{\text{final}} \\
 & & \\
l & \in & \text{lock level names}
\end{array}$$

Figure 9: Grammar Extensions for Deadlock-Free Java

1. The lock levels form a partial order.
2. Objects that own themselves are locks. Every lock belongs to some lock level. The lock level of a lock does not change over time.
3. The necessary and sufficient condition for a thread to acquire a new lock l is that the levels of all the locks that the thread currently holds are greater than the level of l.
4. A thread may also acquire a lock that it already holds. The lock acquire operation is redundant in that case.

Figure 10: Lock Level Properties

belong to savingsLevel while the checkingAccount is declared to belong to checkingLevel. In the example, both the methods of CombinedAccount acquire the lock on savingsAccount before they acquire the lock on checkingAccount to satisfy the condition that locks must be acquired in the descending order.

Methods in Safe Concurrent Java can have locks clauses in addition to accesses clauses to specify assumptions at method boundaries. A locks clause can contain a set of lock levels. These lock levels are the levels of locks that the corresponding method may acquire. To ensure that a program is free of deadlocks, a thread that calls the method can only hold locks that are of a higher level than the levels specified in the locks clause. In the example in Figure 1, both the methods of CombinedAccount contain a locks(savingsLevel) clause. A thread that invokes either of these methods can only hold locks whose level is greater than savingsLevel.

A locks clause can also contain locks in addition to lock levels. If a locks clause contains an object l, then a thread that invokes the corresponding method must either hold the lock on object l (in which case re-acquiring the lock within the method is redundant), or the thread can only hold locks whose level is greater than the level of l. This is useful to support the case where a synchronized method of a class calls another synchronized method of the same class. Figure 11 shows part of a self-synchronized Vector implemented in Safe Concurrent Java.[1] A self-synchronized class is a class that has self as its first owner instead of a formal owner parameter. Methods of a self-synchronized class can assume that the this object owns itself—the methods can therefore synchronize on this and access the this object without requiring external locks using the accesses clause. In the example,

[1]As we mentioned before, all the examples in this paper use an extended language that is syntactically closer to Java.

```
class Vector<self:Vector.l, elementOwner> {
  LockLevel l = new;

  int elementCount = 0;
  ...
  int size() locks (this) {
    synchronized (this) {
      return elementCount;
  }}

  boolean isEmpty() locks (this) {
    synchronized (this) {
      return (size() == 0);
  }}
}
```

Figure 11: Self-Synchronized Vector

the isEmpty method acquires the lock on this and invokes the size method which also acquires the lock on this. This does not violate our condition that locks must be acquired in the descending order because the second lock acquire is redundant.

6 Type Checking

The previous sections presented the grammar for Safe Concurrent Java in Figures 3, 6, and 9. This section describes some of the important rules for type checking. The full set of rules and the complete grammar can be found in the appendix.

6.1 Rules for Type Checking

The core of our type system is a set of rules for reasoning about the typing judgment: $P;\ E;\ ls;\ l_{\text{min}} \vdash e : t$. P, the program being checked, is included here to provide information about class definitions. E is an environment providing types for the free variables of e. ls describes the set of locks held when e is evaluated. l_{min} is the minimum level among the levels of all the locks held when e is evaluated. t is the type of e.

The judgment $P;\ E \vdash e : t$ states that e is of type t, while the judgment $P;\ E;\ ls;\ l_{\text{min}} \vdash e : t$ states that e is of type t provided ls contains the necessary locks to safely evaluate e and l_{min} is greater that the levels of all the locks that are newly acquired when evaluating e.

A typing environment E is defined as follows, where f is a formal owner parameter of a class.

$$E ::= \emptyset \mid E,\ [\textsf{final}]_{\text{opt}}\ t\ x \mid E,\ \text{owner}\ f$$

A lock set ls is defined as follows, where $\text{RO}(x)$ is the root owner of x.

$$ls ::= \textsf{thisThread} \mid ls,\ \textit{lock} \mid ls,\ \text{RO}(e_{\text{final}})$$

A minimum lock level l_{min} is defined as follows, where $\text{LUB}(cn_1.l_1\ \ldots\ cn_k.l_k) > cn_i.l_i\ \forall_{i=1..k}$. Note that LUB(...) is not computed—it is just an expression used as such for type checking. The lock level ∞ denotes that no locks are currently held.

$$l_{\text{min}} ::= \infty \mid cn.l \mid \text{LUB}(cn_1.l_1\ \ldots\ cn_k.l_k)$$

The rule for fork e checks the expression e using a lock set that contains thisThread and is otherwise empty. Since a new thread does not inherit locks held by its parent, $l_{\min}$ for the child is set to ∞. The environment E might have some types that contain thisThread. But the owner thisThread in the parent thread is not the same as the owner thisThread in the child thread. So, all the thisThread owners in the environment must be changed to something else; we use the special owner otherThread for that.

[EXP FORK]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash x_i : t_i \\ g_i = \text{final } t_i[\text{otherThread}/\text{thisThread}]\ x_i \\ P;\ g_{1..n};\ \mathsf{thisThread};\ \infty \vdash e : t \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \mathsf{fork}\ (x_{1..n})\ \{e\} : \mathsf{int}}$$

The rule for acquiring a new lock using synchronized e_1 in e_2 checks that e_1 is a lock of some level $cn.l$. The rule checks that $cn.l$ is less than $l_{\min}$. The rule then type checks e_2 in an extended lock set that includes e_1 and with $l_{\min}$ set to $cn.l$. A lock is a final expression that owns itself. A final expression is either a final variable, or a field access $e.fd$ where e is a final expression and fd is a final field.

[EXP SYNC]

$$\frac{\begin{array}{c} P;\ E \vdash_{\text{final}} e_1 : cn'\langle \mathsf{self}{:}cn.l\ ...\rangle \\ P \vdash cn.l < l_{\min} \\ P;\ E;\ ls,\ e_1;\ cn.l \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \mathsf{synchronized}\ e_1\ \mathsf{in}\ e_2 : t_2}$$

Before we proceed further with the rules, we give a formal definition for RootOwner(e). The root owner of an expression e that refers to an object is the root of the ownership tree to which the object belongs. It could be thisThread, or an object that owns itself.

[ROOTOWNER THISTHREAD]

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

[ROOTOWNER SELF]

$$\frac{P;\ E \vdash e : cn\langle \mathsf{self}{:}cn'.l'\ o*\rangle}{P;\ E \vdash \text{RootOwner}(e) = e}$$

[ROOTOWNER FINAL TRANSITIVE]

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

If the owner of an expression is a formal owner parameter, then we cannot determine the root owner of the expression from within the static scope of the enclosing class. In that case, we define the root owner of e to be RO(e).

[ROOTOWNER FORMAL]

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

The rule for accessing field $e.fd$ checks that e is a well-typed expression of some class type $cn\langle o_{1..n}\rangle$, where $o_{1..n}$ are actual owner parameters. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t and that the thread holds the lock on the root owner of e. Since t is declared inside the class, it might contain occurrences of this and the formal class parameters. When t is used outside the class, we rename this with the expression e, and the formal parameters with their corresponding actual parameters. The rule for assigning to a field is similar.

[EXP REF]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \\ P \vdash ([\mathsf{final}]_{\text{opt}}\ t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \text{RootOwner}(e) = r \quad r \in ls \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.fd : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP ASSIGN]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \\ P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \text{RootOwner}(e) = r \quad r \in ls \\ P;\ E;\ ls;\ l_{\min} \vdash e' : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n] \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.fd = e' : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for invoking a method checks that the arguments are of the right type and that the thread holds the locks on the root owners of all final expressions in the accesses clause of the method. The rule ensures that $l_{\min}$ is greater than all the levels specified in the locks clause of the method. The rule also ensures that for all the locks specified in the locks clause, either the lock is in the lock set (in which case acquiring that lock within the method is redundant), or the level of that lock is less than $l_{\min}$. The expressions and types used inside the method are renamed appropriately when used outside their class.

[EXP INVOKE]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \\ P \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{accesses}(e'*) \\ \qquad\qquad \mathsf{locks}(cn.l*\ lock*)\ ...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls;\ l_{\min} \vdash e_j : t_j[e/\mathsf{this}][o_1/f_1]..[o_n/f_n] \\ P;\ E \vdash \text{RootOwner}(e'_i[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]) = r'_i \quad r'_i \in ls \\ P \vdash cn_i.l_i < l_{\min} \\ (P;\ E \vdash \text{level}(lock_i) < l_{\min}) \text{ or } (lock_i \in ls) \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.mn(e_{1..k}) : t[e/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for type checking a method assumes that the locks on the root owners of all the final expressions specified in the accesses clause are held. The rules also assumes that the levels of all the locks held by the thread are greater than the levels specified in the locks clause and the levels of the locks specified in the locks clause. The rule then type checks the method body under these assumptions.

[METHOD]

$$\frac{\begin{array}{c} P;\, E,\, arg_{1..n} \vdash_{\mathsf{final}} e_i : t_i \\ P;\, E,\, arg_{1..n} \vdash \mathrm{RootOwner}(e_i) = r_i \\ ls = \mathsf{thisThread},\, r_{1..r} \\ P;\, E,\, arg_{1..n} \vdash \mathrm{level}(lock_j) = cn'_j.l'_j \\ l_{\min} = \mathrm{LUB}(cn_j.l_j\ {}^{j \in 1..k}\ cn'_j.l'_j\ {}^{j \in 1..l}) \\ P;\, E,\, arg_{1..n};\, ls;\, l_{\min} \vdash e : t \end{array}}{\begin{array}{l} P;\, E \vdash t\ mn(arg_{1..n})\ \mathsf{accesses}(e_{1..r}) \\ \qquad\qquad \mathsf{locks}(cn_j.l_j\ {}^{j \in 1..k}\ lock_{1..l})\ \{e\} \end{array}}$$

Our type checker also statically verifies that the lock levels declared in the program do indeed form a partial order. Our type checker ensure that there is no cycle of the following form.

[PARTIAL ORDER]

$$\frac{\begin{array}{c} \exists_{cn_0.l_0..cn_{n-1}.l_{n-1}} \text{ such that} \\ \forall_{i=0..n-1}\ P \vdash cn_i.l_i < cn_j.l_j, \text{ where } j = (i+1) \bmod n \end{array}}{P \vdash \neg\, LockLevelsOK(P)}$$

6.2 Soundness of the Type System

Our type checking rules ensure that for a program to be well-typed, the program respects the properties described in Figures 5 and 10. In particular, our type checking rules ensure that a thread can read or write an object only if the thread holds the lock on the root owner of that object, and that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. The properties in Figure 5 imply that program is free of data races, while the properties in Figure 10 imply that a program is free of deadlocks. Well-typed programs in our system are therefore guaranteed to be free of both data races and deadlocks.

A complete syntactic proof [29] of type soundness can be constructed by defining an operational semantics for Safe Concurrent Java (by extending the operational semantics of Classic Java [16]) and then proving that well-typed programs do not reach an error state and that the generalized subject reduction theorem holds for well-typed programs. The subject reduction theorem states that the semantic interpretation of a term's type is invariant under reduction. The proof is straight-forward but tedious, so it is omitted here.

6.3 Runtime Overhead

The system described so far is a purely static type system. The ownership relations and the lock levels are used only for compile-time type checking and are not preserved at runtime. Consequently, Safe Concurrent Java programs have no runtime overhead when compared to regular Concurrent Java programs. In fact, one way to compile and run a Safe Concurrent Java program is to convert it into a Concurrent Java program after type checking, by removing the type parameters, the lock level declarations, the accesses clauses, and the locks clauses from the program.

```
class A<oa1, oa2> {...};
class B<ob1, ob2, ob3> extends A<ob2, ob3> {...};

class C<oc1> {
  void m(B<thisThread, this, oc1> b) {
    A a1;
    B b1;
    b1 = b;
    a1 = b1;
  }
}
```

Figure 12: An Incompletely Typed Method

The Java language, however, is not a purely statically-typed language. Java allows downcasts that are checked at runtime. Suppose an object with declared type Object⟨o⟩ is downcast to Vector⟨o,e⟩. Since the result of this operation depends on information that is only available at runtime, our type checker cannot verify at compile-time that e is the right owner parameter even if we assume that the object is indeed a Vector. To safely support the Java downcast operation, our implementation keeps some ownership information at runtime, but only for objects that can be potentially involved in downcasts into types with multiple parameters. Section 12 describes our implementation.

The extra type information available in our system can be also used to enable program optimizations. For example, objects that are known to be thread-local can be allocated in a thread-local heap instead of the global heap. A thread-local heap may be separately garbage collected, and when the thread dies, the entire space in the thread-local heap may be reclaimed at once.

7 Type Inference

Although our type system is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information that our system requires. Instead, we use a combination of inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. We emphasize that our approach to inference is purely intra-procedural and we do not infer method signatures or types of instance variables. Rather, we use a default completion of partial type specifications in those cases to minimize the required annotations. This approach permits separate compilation. Section 7.1 below describes our intra-procedural inference algorithm, while Section 7.2 describes our default types.

7.1 Intra-Procedural Type Inference

In our system, it is usually unnecessary to explicitly augment the types of method-local variables with their owner parameters. A simple inference algorithm can automatically deduce the owner parameters for otherwise well-typed programs. We illustrate our algorithm with an example. Figure 12 shows a class hierarchy and an incompletely-typed method m. The types of local variables a1 and b1 inside m do not contain their owner parameters explicitly. The inference algorithm works by first augmenting such incomplete types with the appropriate number of distinct, unknown owner

```
      A<x1, x2>     a1;
      B<x3, x4, x5> b1;
```

Figure 13: Types Augmented With Unknown Owners

```
Statement 8   ==>   x3 = thisThread, x4 = this, x5 = oc1
Statement 9   ==>   x1 = x4,         x2 = x5
```

Figure 14: Constraints on Unknown Owners

parameters. For example, since a1 is of type A, the algorithm augments the type of a1 with two owner parameters. Figure 13 shows augmented types for the example in Figure 12. The goal of the inference algorithm is to find known owner parameters that can be used in place of the each of the unknown owner parameters to make the program be well-typed.

The inference algorithm treats the body of the method as a bag of statements. The algorithm works by collecting constraints on the owner parameters for each assignment or function invocation in the method body. Figure 14 shows the constraints imposed by Statements 8 and 9 in the example in Figure 12. Note that all the constraints are of the form of equality between two owner parameters. Consequently, the constraints can be solved using the standard Union-Find algorithm in almost linear time [10]. If the solution is inconsistent, that is, if any two known owner parameters are constrained to be equal to one another by the solution, then the inference algorithm returns an error and the program does not type check. Otherwise, if the solution is incomplete, that is, if there is no known parameter that is equal to an unknown parameter, then the algorithm replaces all such unknown parameters with thisThread.

7.2 Default Types

In addition to the intra-procedural type inference, our system provides well-chosen defaults to reduce the number of annotations needed in many common cases. We are also considering allowing user-defined defaults to cover specific sharing patterns that might occur in user code. The following are some default types currently provided by our system.

If a class is declared to be default-single-threaded, our system adds the following default type annotations wherever they are not explicitly specified by the programmer. If the type of any instance variable in the class or any method argument or return value is not explicitly parameterized, the system augments the type with an appropriate number of thisThread owner parameters. If a method in the class does not contain an accesses or locks clause, the system adds an empty accesses or locks clause to the method. With these default types, single-threaded programs require no extra type annotations.

If a class is declared to be default-self-synchronized, our system adds the following default type annotations wherever they are not explicitly specified by the programmer. If the type of any instance variable is not explicitly parameterized, the system augments the type with an appropriate number of this owner parameters. If the type of any method argument or return value is not explicitly parameterized, the system augments the type with fresh formal owner parameters. If a method in the class does not contain an accesses clause, the system adds an accesses clause that contains all the method arguments. If a method in the class does not contain a locks clause, the system adds a locks(this) clause. With these default types, many self-synchronized classes require almost no extra type annotations.

$$
\begin{array}{rcl}
\mathit{defn} & ::= & \textsf{class}\ \mathit{cn}\langle \mathit{owner}\ \mathit{formal}^*\rangle\ \mathit{whereclause} \\
 & & \textsf{extends}\ c\ \mathit{body} \\
\mathit{formal} & ::= & f \mid \textsf{self}{:}v \\
\mathit{locklevel} & ::= & \mathit{cn}.l \mid v \\
\mathit{whereclause} & ::= & \textsf{where}\ (\mathit{locklevel} > \mathit{locklevel})^* \\
\mathit{locksclause} & ::= & \textsf{locks}\ (\mathit{locklevel}^*\ \mathit{lock}^*) \\
 & & \\
v & \in & \text{formal lock level names}
\end{array}
$$

Figure 15: Grammar Extensions for Level Polymorphism

```
class Stack<self:v, elementOwner> where (v > Vector.l) {
  Vector<self:Vector.l, elementOwner> vec = new Vector;
  ...
  int size() locks(this) {
    synchronized (this) {
      return vec.size();
  }}
}
```

Figure 16: Self-Synchronized Stack Using Vector

8 Lock Level Polymorphism

This section describes how our type system supports polymorphism in lock levels. In the type system described in Section 5, the level of each lock was known at compile-time. But programmers may sometimes want to write code where the exact levels of some locks may not be known statically—only some ordering constraints among the unknown lock levels may be known statically. Lock level polymorphism enables this kind of programming. To simplify the presentation, this section describes how our type system supports lock level polymorphism in the context of Safe Concurrent Java. Figure 15 shows how the grammar of Safe Concurrent Java can be extended to support lock level polymorphism.

Classes may be parameterized with formal lock level parameters in addition to formal owner parameters. Ordering constraints among the formal lock level parameters may be specified using where clauses. This is somewhat similar to the use of where clauses in [11, 24]. Figure 16 shows part of a self-synchronized Stack implemented using the self-synchronized Vector shown in Figure 11. In the example, the lock level of the this Stack object is a formal parameter v. The where clause constrains v to be greater than Vector.l. It is therefore legal for the synchronized Stack.size method to call the synchronized Vector.size method. The type checker statically verifies that locks are acquired in the descending order of lock levels.

9 Condition Variables

This section describes how our system prevents deadlocks in the presence of condition variables. Java provides condition variables in the form of the wait and notify methods on Object. Since a thread can wait on a condition variable as well as on a lock, it is possible to have a deadlock that involves

$$\textit{locksclause} \quad ::= \quad \textsf{locks}\ ([\infty]_{\text{opt}}\ \textit{locklevel* lock*})$$

$$e \quad ::= \quad ...\ |\ e.\textsf{wait}\ |\ e.\textsf{notify}$$

Figure 17: Grammar Extensions for Condition Variables

condition variables as well as locks. There is no simple rule like the ordering rule for locks that can avoid this kind of deadlock. The lock ordering rule depends on the fact that a thread must be holding a lock to keep another thread waiting for that lock. In the case of conditions, the thread that will notify cannot be distinguished in such a simple way.

To simplify the presentation, this section describes how our type system handles condition variables in the context of Safe Concurrent Java. Figure 17 shows how the grammar of Safe Concurrent Java can be extended to support condition variables. The expression e.wait and e.notify are similar to the wait and notifyAll methods in Java. e must be a final expression that evaluates to an object, and the current thread must hold the lock on e. On executing wait, the current thread releases the lock on e and suspends itself. The thread resumes execution when some other thread invokes notify on the same object. The thread re-acquires the lock on e before resuming execution after wait.

To prevent deadlocks in the presence of condition variables, our system enforces the following constraint. A thread can invoke e.wait only if the thread holds no locks other than the lock on e. Since a thread releases the lock on e on executing e.wait, the above constraint implies that any thread that is waiting on a condition variable holds no locks. This in turn implies that there cannot be a deadlock that involves a condition variable. To statically verify that a program respects the above constraint, our type system requires that any method m that contains a call to e.wait must have a locks (∞) clause or a locks (∞ e) clause. The former locks clause indicates that a thread holds no locks when it invokes m, while the later locks clause indicates that a thread can only hold the lock on e when it invokes m. Within the method, our type checker ensures when type checking e.wait that the lockset contains only the lock on e and no other lock. The rules for type checking are shown below, assuming the locks clause of the enclosing method is included in the environment E.

[EXP WAIT]

$$\frac{\begin{array}{c} P;\, E \vdash \textit{Method has}\ \textsf{locks}(\infty\ [e]_{\text{opt}})\ \textit{clause} \\ P;\, E \vdash_{\text{final}} e \quad ls = \{e\} \end{array}}{P;\, E;\, ls;\, l_{\text{min}} \vdash e.\textsf{wait} : \textsf{int}}$$

[EXP NOTIFY]

$$\frac{P;\, E \vdash_{\text{final}} e \quad e \in ls}{P;\, E;\, ls;\, l_{\text{min}} \vdash e.\textsf{notify} : \textsf{int}}$$

10 Tree-Based Partial Orders

This section describes how our type system supports tree-based partial orders. Figure 18 shows the grammar extensions to Safe Concurrent Java to support tree-based partial orders. Programmers can declare fields in objects to be tree fields. If object x has a tree field fd that contains a pointer to object y, we say that there is a tree edge fd from x to y. x is the parent of y and y is a child of x. Our type system ensures that the graph induced by the set of all tree edges in the heap is indeed a forest of trees. Any data structure that has a tree backbone can be used to describe the partial order in our system. This includes doubly linked lists, trees with parent pointers, threaded trees, and balanced search trees.

$$\textit{field} \quad ::= \quad [\textsf{final}]_{\text{opt}}\ [\textsf{tree}]_{\text{opt}}\ t\ \textit{fd} = e$$

Figure 18: Grammar Extensions for Tree Ordering

Stmt #	Information in Environment After Checking Statement in Figure 2		

23	x=this.right
	v=x.left
	w=v.right		

24	x=this.right
	v=x.left
		w is Root	this not in Tree(w)
			x not in Tree(w)
			v not in Tree(w)

25	x=this.right
	w=x.left
		v is Root	this not in Tree(v)
			x not in Tree(v)
			w not in Tree(v)

26	v=this.right
	w=x.left
		x is Root	this not in Tree(x)
			v not in Tree(x)

27	v=this.right
	w=x.left
	x=v.right		

Figure 19: Illustration of Flow-Sensitive Analysis

Locks that belong to the same lock level are further ordered according to the tree-order. Suppose x and y are two locks (that is, they are objects that own themselves) and that x and y belong to the same lock level. Suppose a thread t holds the lock on x and reads a tree field fd of x to get a pointer to y. So y is a child of x. Our type system then allows thread t to also acquire the lock on y while holding the lock on x. Note that as long as t holds the lock on x, no other thread can modify x, so no other thread can make y not a child of x. The type checking rule is shown below, assuming that for every pair of final variables x and y, environment E contains information about whether the objects x and y are related by tree edges.

[EXP SYNC CHILD]

$$\frac{\begin{array}{c} \forall_{y \in ls}\ (P;\, E \vdash \text{level}(y) > l_{\text{min}}\ \text{or}\ P;\, E \vdash y\ \text{is an ancestor of}\ x) \\ x' \in ls \quad P;\, E \vdash x\ \text{is a child of}\ x' \\ P;\, E \vdash \text{level}(x) = \text{level}(x') = l_{\text{min}} \\ P;\, E;\, ls,\, x;\, l_{\text{min}} \vdash e : t \end{array}}{P;\, E;\, ls;\, l_{\text{min}} \vdash \textsf{synchronized}\ x\ \textsf{in}\ e : t}$$

Figure 2 presents an example with a tree-based partial order. The Node class is self-synchronized, that is, the this Node object owns itself. The lock level of the this Node object

$$\textit{field} \quad ::= \quad [\mathsf{final}]_{\mathrm{opt}}\ [\mathsf{tree}]_{\mathrm{opt}}\ t\ fd = e \mid \mathsf{final\ dag}\ t\ fd = e$$

Figure 20: Grammar Extensions for DAG Ordering

is the formal parameter v. A Node has two tree fields left and right. The Nodes left and right own themselves and also belong to lock level v. Nodes left and right are therefore ordered less than the this Node object in the partial order. In the example, the rotateRight method acquires the locks on Nodes this, x and v in the tree-order.

Our type system allows a limited set of mutations on trees at runtime. The type checker uses a simple intra-procedural intra-loop flow-sensitive analysis to check that the mutations do not introduce cycles in the trees. We illustrate our flow-sensitive analysis using the example in Figure 2. The type checker keeps the following additional information in the environment E for every pair of final variables x and y: 1) If the objects x and y are related by a tree edge, 2) If x is the root of a tree, and 3) If x is a root and y is not in the tree rooted at x. Figure 19 contains the information stored in the environment after the type checking of various statements in the rotateRight method in Figure 2. Since the analysis is flow-sensitive, the environment changes after checking each statement.

The rules for mutating a tree are as follows. Deleting a tree edge (for example, setting a tree field to null or over-writing a tree field) requires no extra checking. A tree edge from x to x' may be added only if x' is the root of a tree and x is not in the tree rooted at x'. The rule is shown below. Note that if x' is a unique pointer to an object (for example, x' is newly created using the expression new c), then x' is trivially a root. Similarly, if x is a unique pointer, then x cannot be in the tree rooted at x'.

[EXP TREE ASSIGN]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash x : cn\langle o_{1..n}\rangle \\ P \vdash (\mathsf{tree}\ t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \mathrm{RootOwner}(x) = r \qquad r \in ls \\ P;\ E;\ ls;\ l_{\min} \vdash x' : t[x/\mathsf{this}][o_1/f_1]..[o_n/f_n] \\ P;\ E \vdash x' \text{ is Root} \\ P;\ E \vdash x \text{ not in Tree}(x') \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash x.fd = x' : t[x/\mathsf{this}][o_1/f_1]..[o_n/f_n]}$$

11 DAG-Based Partial Orders

Our type system also allows programmers to use directed acyclic graphs (DAGs) to describe the partial order. Figure 20 shows the grammar extensions to Safe Concurrent Java to support DAG-based partial orders. Programmers can declare fields in objects to be dag fields. Our type system ensures that no object can be both part of a tree and part of a DAG. Locks that belong to the same lock level are further ordered according to the DAG-order. DAGs used for partial orders are monotonic. DAG fields cannot be modified once initialized. Only newly created nodes may be added to a DAG by initializing the newly created nodes to contain DAG edges to existing DAG nodes.

Program	Lines of Code	Lines Changed
elevator	523	15
http	563	26
chat	308	22
stock quote	242	12
game	087	11
phone	302	10

Figure 21: Programming Overhead

12 Implementation

We have a prototype implementation of our type system. Our implementation handles all the features of the Java language including threads, constructors, arrays, exceptions, static fields, interfaces, and runtime downcasts. The type system we implemented is also more expressive than the type system we described formally in earlier sections of this paper. Our implementation supports unsynchronized accesses to immutable objects and objects with unique pointers [3].

Our implementation also supports parameterized methods in addition to parameterized classes. This is useful in many cases. For example, the PrintStream class has a print(Object) method. Let us say, the Object argument is owned by ObjectOwner. If we did not have parameterized methods, then the PrintStream class would have to have an ObjectOwner parameter. Not only would this be unnecessarily tedious, but it would also mean that all objects that can be printed by a PrintStream must have the same protection mechanism. Having parameterized methods allows us to implement a generic print(Object) method.

We also support safe runtime downcasts in our implementation. This is important because Java is not a fully statically-typed language. It allows downcasts that are checked at runtime. Suppose an object with declared type Object⟨o⟩ is downcast to Vector⟨o,e⟩. We cannot verify at compile-time that e is the right owner parameter even if we assume that the object is indeed a Vector. We use type passing to support safe runtime downcasts. Our technique is similar to the technique for implementing parametric polymorphism in Java described in [27]. We only keep runtime ownership and lock level information for objects that are potentially involved in downcasts to types with multiple parameters.

To gain preliminary experience, we implemented several Java programs in our system. These include *elevator*, a real time discrete event simulator [28, 7], an *http* server, a *chat* server, a *stock quote* server, a *game* server, and *phone*, a database-backed information sever. These programs exhibit a variety of sharing patterns. Our type system was expressive enough to support these programs. In each case, once the sharing pattern of the program was known, adding the extra type annotations was a fairly straight forward process. Figure 21 presents a measure of the programming overhead involved. The figure shows the lines of code that needed type annotations.

In our experience, we found that threads rarely need to hold multiple locks at the same time. In cases where threads do hold multiple locks simultaneously, the threads usually acquire the multiple locks as they cross abstraction boundaries. For example, in *elevator*, threads acquire the lock on a Floor object and then invoke synchronized methods on a Vector object. Even though such programs use an unbounded number of locks, these locks can be classified into a small number of lock levels. These programs are therefore easily expressed in our type system.

We also note that in cases where threads do hold multiple locks simultaneously, it is usually because of conservative programming. In the *elevator* example mentioned above, the Vector object is contained within the Floor object. Acquiring the lock on the Vector object is thus unnecessary. In fact, programmers can use an ArrayList instead of a Vector. The reason many Java programs are conservative is because there is no mechanism in Java to prevent data races or deadlocks. For example, Java programs that use ArrayLists risk data races because ArrayLists may be accessed without appropriate synchronization in shared contexts. But since our type system guarantees data race freedom and deadlock freedom, programmers can employ aggressive locking disciplines without sacrificing safety.

13 Related Work

There has been much research on approaches that help programmers detect data races and deadlocks in multithreaded programs.

13.1 Static Tools

Tools like Warlock [26] and Sema [20] use annotations supplied by programmers to statically detect potential data races and deadlocks in a program. The Extended Static Checker for Java (Esc/Java) [21, 12] is another annotation based system that uses a theorem prover to statically detect many kinds of errors including data races and deadlocks. Another recent system [14] assumes bugs to be deviant behavior to statically extract and check correctness conditions that a system must obey without requiring programmer annotations. While these tools are useful in practice, they are not sound, in that they do not certify that a program is race-free or deadlock-free. For example, ESC/Java does not always verify that a partial order of locks declared in a program is indeed a partial order.

13.2 Dynamic Tools

There are many systems that detect data races and deadlocks dynamically. These include systems developed in the scientific parallel programming community like [13, 18], tools like Eraser [25], and tools for detecting data races in Java programs like [28, 7]. Eraser dynamically monitors all lock acquisitions to test whether a linear order exists among the locks that is respected by every thread. Dynamic tools have the advantage that they can check unannotated programs. However, these tools are not comprehensive—they may fail to detect certain errors due to insufficient test coverage.

13.3 Language Mechanisms

To our knowledge, Concurrent Pascal is the first race-free programming language [5]. Programs in Concurrent Pascal used synchronized monitors to prevent data races. But monitors in Concurrent Pascal were restricted in that threads could share data with monitors only by copying the data. A thread could not pass a reference to an object to a monitor.

More recently, researchers have proposed type systems to prevent data races in object-oriented programs. Race Free Java [15] extends the static annotations in Esc/Java into a formal race-free type system. Guava [2] is another dialect of Java for preventing data races. Our race-free type system published earlier [3] lets programmers write generic code to implement a class, and create different objects of the same class that have different protection mechanisms. Our type system described in this paper extends the race-free type system [3] to prevent both data races and deadlocks.

13.4 Message Passing Systems

There are several systems that statically check for races and deadlocks in message passing systems [19, 6]. The programming model used in these systems is different from the Java programming model. Unlike Java programs, these programs do not access shared objects in a heap.

13.5 Related Type Systems

Our type system is related to several other type systems discussed in literature, even though their goals and techniques were different. The concept of object ownership used in this paper is similar to the one in ownership types [9, 8]. Ownership types were motivated by software engineering principles and were used to restrict object aliasing. Our way of parameterizing classes is similar to the proposals for parametric types for Java [24, 4, 1, 27], except that the parameters in our system are values and not types. Our use of where clauses is somewhat similar to the use of where clauses in [11, 24]. Our accesses clauses and locks clauses are similar to type and effect systems [23].

14 Conclusions

Multithreaded programming is difficult and error prone. This paper presents a new static type system for multithreaded programs; well-typed programs in our system are guaranteed to be free of both data races and deadlocks. Our type system allows programmers to partition the locks into a fixed number of lock levels and specify a partial order among the lock levels. Our system also allows programmers to use recursive tree-based data structures to further order locks that belong to the same lock level. The type checker then statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. The type checker also uses an intra-procedural intra-loop flow-sensitive analysis to check that mutations to trees used for describing partial orders do not introduce cycles in the partial order, and that the changing of the partial order does not lead to deadlocks. We do not know of any other sound static system for preventing deadlocks that allows changes to the partial order at runtime. We implemented our type system for Java. Our preliminary experience indicates that our type system is sufficiently expressive and requires little programming overhead.

References

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[2] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[3] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[5] P. Brinch-Hansen. The programming language Concurrent Pascal. In *IEEE Transactions on Software Engineering SE-1(2)*, June 1975.

[6] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: Model checking message-passing programs. In *Principles of Programming Languages (POPL)*, January 2002.

[7] J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[8] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *European Conference for Object-Oriented Programming (ECOOP)*, June 2001.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1991.

[11] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1995.

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

[13] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging (AOWPDD)*, May 1991.

[14] D. R. Engler, D. Y. Chen, S. Hallem, A. Chon, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, October 2001.

[15] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[16] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, January 1998.

[17] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[18] G. Ien Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1998.

[19] A. Igarashi and N. Kobayashi. A generic type system for the Pi-calculus. In *Principles of Programming Languages (POPL)*, January 2001.

[20] J. A. Korty. Sema: A lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *USENIX Winter Technical Conference*, January 1989.

[21] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Research Report 002, Compaq Systems Research Center, 1999.

[22] A. Lister. The problem of nested monitor calls. In *Operating Systems Review 11(3)*, July 1977.

[23] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[24] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Symposium on Operating Systems Principles (SOSP)*, October 1997.

[26] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, January 1993.

[27] M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[28] C. von Praun and T. Gross. Object-race detection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[29] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. In *Information and Computation 115(1)*, November 1994.

Appendix

A Type System for Safe Concurrent Java

This appendix presents the type system described in Section 5. The grammar for the type system is shown below.

P	$::=$	$\mathit{defn}^*\ e$
defn	$::=$	$\mathsf{class}\ \mathit{cn}\langle \mathit{owner}\ \mathit{formal}^*\rangle\ \mathsf{extends}\ c\ \{\mathit{level}^*\ \mathit{field}^*\ \mathit{meth}^*\}$
c	$::=$	$\mathit{cn}\langle \mathit{owner}+\rangle \mid \mathsf{Object}\langle \mathit{owner}+\rangle$
owner	$::=$	$\mathit{formal} \mid \mathsf{self}{:}\mathit{cn}.l \mid \mathsf{thisThread} \mid e_{\text{final}}$
level	$::=$	$\mathsf{LockLevel}\ l = \mathsf{new} \mid \mathsf{LockLevel}\ l < \mathit{cn}.l^* > \mathit{cn}.l^*$
meth	$::=$	$t\ \mathit{mn}(\mathit{arg}^*)\ \mathsf{accesses}\ (e_{\text{final}}{}^*)\ \mathsf{locks}\ (\mathit{cn}.l^*\ \mathit{lock}^*)\ \{e\}$
field	$::=$	$[\mathsf{final}]_{\text{opt}}\ t\ \mathit{fd} = e$
arg	$::=$	$[\mathsf{final}]_{\text{opt}}\ t\ x$
t	$::=$	$c \mid \mathsf{int} \mid \mathsf{boolean}$
formal	$::=$	f
e	$::=$	$\mathsf{new}\ c \mid x \mid x = e \mid e.\mathit{fd} \mid e.\mathit{fd} = e \mid e.\mathit{mn}(e^*) \mid e;e \mid \mathsf{let}\ (\mathit{arg}{=}e)\ \mathsf{in}\ \{e\} \mid \mathsf{if}\ (e)\ \mathsf{then}\ \{e\} \mid \mathsf{synchronized}\ (e)\ \mathsf{in}\ \{e\} \mid \mathsf{fork}\ (x^*)\ \{e\}$
e_{final}	$::=$	e
lock	$::=$	e_{final}
cn	$\in$	class names
fd	$\in$	field names
mn	$\in$	method names
x	$\in$	variable names
f	$\in$	owner names
l	$\in$	lock level names

We first define a number of predicates used in the type system informally. These predicates (except the last one) are based on similar predicates from [16] and [15]. We refer the reader to those papers for their precise formulation.

Predicate	Meaning
ClassOnce(P)	No class is declared twice in P
WFClasses(P)	There are no cycles in the class hierarchy
FieldsOnce(P)	No class contains two fields with the same name, either declared or inherited
MethodsOnce(P)	No class contains two methods with the same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden
	The accesses clause of an overriding method must be the same or a subset of the overridden methods
	The locks clause of an overriding method must be the same or a subset of the overridden methods
LockLevelsOK(P)	There are no cycles in the lock levels

A typing environment is defined as $E ::= \emptyset \mid E,\ [\mathsf{final}]_{\text{opt}}\ t\ x \mid E,\ \text{owner}\ f$

A lock set is defined as $ls ::= \mathsf{thisThread} \mid ls,\ \mathit{lock} \mid ls,\ \mathrm{RO}(e_{\text{final}})$; where $\mathrm{RO}(e)$ is the root owner of e

A minimum lock level is defined as $l_{\min} ::= \infty \mid \mathit{cn}.l \mid \mathrm{LUB}(\mathit{cn}_1.l_1\ \ldots\ \mathit{cn}_k.l_k)$; where $\mathrm{LUB}(\mathit{cn}_1.l_1\ \ldots\ \mathit{cn}_k.l_k) > \mathit{cn}_i.l_i\ \forall_{i=1..k}$

Note that RO(e) and LUB(...) are not computed—they are just expressions used as such for type checking.

We define the type system using the following judgments. We present the typing rules for these judgments after that.

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash \mathit{defn}$	*defn* is a well-formed class definition
$P;\, E \vdash wf$	E is a well-formed typing environment
$P;\, E \vdash t$	t is a well-formed type
$P;\, E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P;\, E \vdash_{\text{owner}} o$	o is an owner
$P \vdash_{\text{level}} \mathit{cn}.l$	$\mathit{cn}.l$ is a well-formed lock level
$P \vdash \mathit{cn}_1.l_1 < \mathit{cn}_2.l_2$	$\mathit{cn}_1.l_1$ is less than $\mathit{cn}_2.l_2$ in the partial order formed by lock levels
$P \vdash \mathit{cn}.l < l_{\min}$	$\mathit{cn}.l$ is less than $l_{\min}$ in the partial order formed by lock levels
$P;\, E \vdash \text{level}(e) = \mathit{cn}.l$	e is a final expression that owns itself and the lock level of e is $\mathit{cn}.l$
$P;\, E \vdash \text{level}(e) < l_{\min}$	e is a final expression that owns itself and the lock level of e is less than $l_{\min}$
$P;\, E \vdash_{\text{final}} e : t$	e is a final expression with type t
$P;\, E \vdash \mathit{field\ init}$	*field init* is a well-formed field initializer
$P \vdash \mathit{field} \in \mathit{cn}\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits *field*
$P \vdash \mathit{meth} \in \mathit{cn}\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits *meth*
$P;\, E \vdash \mathit{meth}$	*meth* is a well-formed method
$P;\, E \vdash \text{RootOwner}(e) = r$	r is the root owner of the final expression e
$P;\, E \vdash e : t$	expression e has type t
$P;\, E;\, ls;\, l_{\min} \vdash e : t$	expression e has type t and evaluating e will not create data races or deadlocks

$\boxed{\vdash P : t}$

[PROG]

$$\dfrac{\begin{array}{c} \mathit{ClassOnce(P)} \quad \mathit{WFClasses(P)} \quad \mathit{FieldsOnce(P)} \\ \mathit{MethodsOnce(P)} \quad \mathit{OverridesOK(P)} \quad \mathit{LockLevelsOK(P)} \\ P = \mathit{defn}_{1..n}\ e \quad P \vdash \mathit{defn}_i \quad P;\ \emptyset;\ \mathsf{thisThread};\ \infty \vdash e : t \end{array}}{\vdash P : t}$$

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

[CLASS]

$$\dfrac{\begin{array}{c} \text{if } (f_1 \neq \mathsf{self}{:}cn'.l' \mid \mathsf{thisThread}) \text{ then } g_1 = \text{owner } f_1 \\ \forall_{i=2..n}\ g_i = \text{owner } f_i \quad E = g_{1..n},\ \text{final } cn\langle f_{1..n}\rangle\ \mathsf{this} \\ P;E \vdash c \quad P;E \vdash \mathit{field}_i \quad P;E \vdash \mathit{meth}_i \end{array}}{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c\ \{\mathit{field}_{1..j}\ \mathit{meth}_{1..k}\}}$$

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

[ENV $\emptyset$]

$$\dfrac{}{P;\ \emptyset \vdash wf}$$

[ENV OWNER]

$$\dfrac{P;\ E \vdash wf \quad f \notin \mathrm{Dom}(E)}{P;\ E,\ \text{owner } f \vdash wf}$$

[ENV X]

$$\dfrac{P;\ E \vdash t \quad x \notin \mathrm{Dom}(E)}{P;\ E,\ [\text{final}]_{\text{opt}}\ t\ x \vdash wf}$$

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

[TYPE INT]

$$\dfrac{}{P;\ E \vdash \mathsf{int}}$$

[TYPE BOOLEAN]

$$\dfrac{}{P;\ E \vdash \mathsf{boolean}}$$

[TYPE OBJECT]

$$\dfrac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash \mathsf{Object}\langle o\rangle}$$

[TYPE SHARED CLASS]

$$\dfrac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle \mathsf{self}{:}cn'.l'\ f_{2..n}\rangle\ ... \\ o_1 = \mathsf{self}{:}cn'.l' \quad P;\ E \vdash_{\text{owner}} o_{1..n} \end{array}}{P;\ E \vdash cn\langle o_{1..n}\rangle}$$

[TYPE THREAD-LOCAL CLASS]

$$\dfrac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle \mathsf{thisThread}\ f_{2..n}\rangle\ ... \\ o_1 = \mathsf{thisThread} \quad P;\ E \vdash_{\text{owner}} o_{1..n} \end{array}}{P;\ E \vdash cn\langle o_{1..n}\rangle}$$

[TYPE C]

$$\dfrac{\begin{array}{c} P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ ... \\ f_1 \neq \mathsf{self}{:}cn'.l' \mid \mathsf{thisThread} \quad P;\ E \vdash_{\text{owner}} o_{1..n} \end{array}}{P;\ E \vdash cn\langle o_{1..n}\rangle}$$

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

[SUBTYPE REFL]

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

[SUBTYPE TRANS]

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

[SUBTYPE CLASS]

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

$\boxed{P;\ E \vdash_{\text{owner}} o}$

[OWNER THISTHREAD]

$$\dfrac{}{P;\ E \vdash_{\text{owner}} \mathsf{thisThread}}$$

[OWNER OTHERTHREAD]

$$\dfrac{}{P;\ E \vdash_{\text{owner}} \mathsf{otherThread}}$$

[OWNER SELF]

$$\dfrac{P \vdash_{\text{level}} cn.l}{P;\ E \vdash_{\text{owner}} \mathsf{self}{:}cn.l}$$

[OWNER EXP]

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

[OWNER FORMAL]

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

$\boxed{P \vdash_{\text{level}} cn.l}$

[LEVEL]

$$\dfrac{P \vdash \mathsf{class}\ cn...\ \{...\ \text{Locklevel } l\ ...\}}{P \vdash_{\text{level}} cn.l}$$

$\boxed{P \vdash cn_1.l_1 < cn_2.l_2}$

[LEVEL <]

$$\dfrac{P \vdash \mathsf{class}\ cn_1...\ \{...\ \text{LockLevel } l_1 < ...\ cn_2.l_2\ ...\}}{P \vdash cn_1.l_1 < cn_2.l_2}$$

[LEVEL >]

$$\dfrac{P \vdash \mathsf{class}\ cn_2...\ \{...\ \text{LockLevel } l_2 > ...\ cn_1.l_1\ ...\}}{P \vdash cn_1.l_1 < cn_2.l_2}$$

$\boxed{P \vdash cn.l < l_{\text{min}}}$

[LEVEL < INFTY]

$$\dfrac{\begin{array}{c} l_{\text{min}} = \infty \\ P \vdash_{\text{level}} cn.l \end{array}}{P \vdash cn.l < l_{\text{min}}}$$

[LEVEL < LUB]

$$\dfrac{\begin{array}{c} l_{\text{min}} = \mathrm{LUB}(...\ cn.l\ ...) \\ P \vdash_{\text{level}} cn.l \end{array}}{P \vdash cn.l < l_{\text{min}}}$$

[LEVEL < CN.L]

$$\dfrac{\begin{array}{c} l_{\text{min}} = cn'.l' \\ P \vdash cn.l < cn'.l' \end{array}}{P \vdash cn.l < l_{\text{min}}}$$

[LEVEL TRANS]

$$\dfrac{\begin{array}{c} P \vdash cn'.l' < l_{\text{min}} \\ P \vdash cn.l < cn'.l' \end{array}}{P \vdash cn.l < l_{\text{min}}}$$

$\boxed{P;\ E \vdash \text{level}(e) = cn.l}$

[LEVEL(EXP)]

$$\dfrac{P;\ E \vdash_{\text{final}} e : cn'\langle \mathsf{self}{:}cn.l\ ...\rangle}{P;\ E \vdash \text{level}(e) = cn.l}$$

$\boxed{P;\ E \vdash \text{level}(e) < l_{\text{min}}}$

[LEVEL < LEVEL MIN]

$$\dfrac{\begin{array}{c} P;\ E \vdash \text{level}(e) = cn.l \\ P \vdash cn.l < l_{\text{min}} \end{array}}{P;\ E \vdash \text{level}(e) < l_{\text{min}}}$$

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

[FINAL VAR]

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

[FINAL REF]

$$\dfrac{\begin{array}{c} P \vdash (\text{final } t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash_{\text{final}} e : cn\langle o_{1..n}\rangle \end{array}}{P;\ E \vdash_{\text{final}} e.fd : t[o_1/f_1]..[o_n/f_n]}$$

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

[FIELD INIT]

$$\dfrac{P;\ E;\ \mathsf{thisThread};\ \infty \vdash e : t}{P;\ E \vdash [\text{final}]_{\text{opt}}\ t\ fd = e}$$

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

[FIELD DECLARED]

$$\dfrac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle...\ \{...\ \mathit{field}\ ...\}}{P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

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

$\boxed{P \vdash \mathit{meth} \in c}$

[METHOD DECLARED]

$$\dfrac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle...\ \{...\ \mathit{meth}\ ...\}}{P \vdash \mathit{meth} \in cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

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

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

[METHOD]

$$\frac{\begin{array}{c} P;\ E,\ arg_{1..n} \vdash_{\textsf{final}} e_i : t_i \quad P;\ E,\ arg_{1..n} \vdash \mathrm{RootOwner}(e_i) = r_i \\ P;\ E,\ arg_{1..n} \vdash \mathrm{level}(lock_j) = cn'_j.l'_j \quad l_{\min} = \mathrm{LUB}(cn_j.l_j\ {}^{j\in 1..k}\ cn'_j.l'_j\ {}^{j\in 1..l}) \\ P;\ E,\ arg_{1..n};\ \textsf{thisThread},\ r_{1..r};\ l_{\min} \vdash e : t \end{array}}{P;\ E \vdash t\ mn(arg_{1..n})\ \textsf{accesses}(e_{1..r})\ \textsf{locks}(cn_j.l_j\ {}^{j\in 1..k}\ lock_{1..l})\ \{e\}}$$

$\boxed{P;\ E \vdash \mathrm{RootOwner}(e) = r}$

[ROOTOWNER THISTHREAD]

$$\frac{P;\ E \vdash e : cn\langle \textsf{thisThread}\ o*\rangle}{P;\ E \vdash \mathrm{RootOwner}(e) = \textsf{thisThread}}$$

[ROOTOWNER OTHERTHREAD]

$$\frac{P;\ E \vdash e : cn\langle \textsf{otherThread}\ o*\rangle}{P;\ E \vdash \mathrm{RootOwner}(e) = \textsf{otherThread}}$$

[ROOTOWNER SELF]

$$\frac{P;\ E \vdash e : cn\langle \textsf{self}{:}cn'.l'\ o*\rangle}{P;\ E \vdash \mathrm{RootOwner}(e) = e}$$

[ROOTOWNER FINAL TRANSITIVE]

$$\frac{\begin{array}{c} P;\ E \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash_{\mathrm{final}} o_1 : c_1 \quad P;\ E \vdash \mathrm{RootOwner}(o_1) = r \end{array}}{P;\ E \vdash \mathrm{RootOwner}(e) = r}$$

[ROOTOWNER FORMAL]

$$\frac{\begin{array}{c} P;\ E \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash_{\mathrm{owner}} o_1 \end{array}}{P;\ E \vdash \mathrm{RootOwner}(e) = \mathrm{RO}(e)}$$

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

[EXP TYPE]

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

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

[EXP SUB]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : t' \\ P;\ E;\ ls;\ l_{\min} \vdash t' <: t \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e : t}$$

[EXP NEW]

$$\frac{P;\ E \vdash c}{P;\ E;\ ls;\ l_{\min} \vdash \textsf{new}\ c : c}$$

[EXP VAR]

$$\frac{\begin{array}{c} P;\ E \vdash wf \\ E = E_1,\ [\textsf{final}]_{\mathrm{opt}}\ t\ x,\ E_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash x : t}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} P;\ E \vdash wf \\ E = E_1,\ t\ x,\ E_2 \quad P;\ E;\ ls;\ l_{\min} \vdash e : t \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash x = e : t}$$

[EXP SEQ]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e_1 : t_1 \\ P;\ E;\ ls;\ l_{\min} \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e_1;\ e_2 : t_2}$$

[EXP LET]

$$\frac{\begin{array}{c} arg = [\textsf{final}]_{\mathrm{opt}}\ t\ x \quad P;\ E;\ ls;\ l_{\min} \vdash e : t \\ P;\ E,\ arg;\ ls;\ l_{\min} \vdash e' : t' \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \textsf{let}\ (arg = e)\ \textsf{in}\ \{e'\} : t'}$$

[EXP IF]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e_1 : \textsf{boolean} \\ P;\ E;\ ls;\ l_{\min} \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \textsf{if}\ (e_1)\ \textsf{then}\ \{e_2\} : t_2}$$

[EXP REF]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \quad P \vdash ([\textsf{final}]_{\mathrm{opt}}\ t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \mathrm{RootOwner}(e) = r \quad r \in ls \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.fd : t[e/\textsf{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP ASSIGN]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \mathrm{RootOwner}(e) = r \quad r \in ls \\ P;\ E;\ ls;\ l_{\min} \vdash e' : t[e/\textsf{this}][o_1/f_1]..[o_n/f_n] \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.fd = e' : t[e/\textsf{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP SYNC]

$$\frac{\begin{array}{c} P;\ E \vdash \mathrm{level}(e_1) = cn.l \\ P \vdash cn.l < l_{\min} \\ P;\ E;\ ls,\ e_1;\ cn.l \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \textsf{synchronized}\ e_1\ \textsf{in}\ e_2 : t_2}$$

[EXP SYNC REDUNDANT]

$$\frac{\begin{array}{c} e_1 \in ls \\ P;\ E;\ ls;\ l_{\min} \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \textsf{synchronized}\ e_1\ \textsf{in}\ e_2 : t_2}$$

[EXP FORK]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash x_i : t_i \\ g_i = \mathrm{final}\ t_i[\mathrm{otherThread}/\mathrm{thisThread}]\ x_i \\ P;\ g_{1..n};\ \textsf{thisThread};\ \infty \vdash e : t \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \textsf{fork}\ (x_{1..n})\ \{e\} : \textsf{int}}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \\ P \vdash (t\ mn(t_j\ y_j\ {}^{j\in 1..k})\ \textsf{accesses}(e'*)\ \textsf{locks}(cn.l*\ lock*)\ ...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls;\ l_{\min} \vdash e_j : t_j[e/\textsf{this}][o_1/f_1]..[o_n/f_n] \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[e/\textsf{this}][o_1/f_1]..[o_n/f_n]) = r'_i \quad r'_i \in ls \\ P \vdash cn_i.l_i < l_{\min} \quad (P;\ E \vdash \mathrm{level}(lock_i) < l_{\min})\ \mathrm{or}\ (lock_i \in ls) \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.mn(e_{1..k}) : t[e/\textsf{this}][o_1/f_1]..[o_n/f_n]}$$