ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), November 2002

Ownership Types for Safe Programming: Preventing Data Races and Deadlocks

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 multithreaded 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.

Our system uses a variant of ownership types to prevent data races and deadlocks. Ownership types provide a statically enforceable way of specifying object encapsulation. Ownership types are useful for preventing data races and deadlocks because the lock that protects an object can also protect its encapsulated objects. This paper describes how to use our type system to statically enforce object encapsulation as well as prevent data races and deadlocks. The paper also contains a detailed discussion of different ownership type systems and the encapsulation guarantees they provide.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs;
D.2.4 [**Software Engineering**]: Program Verification

General Terms

Languages, Verification

Keywords

Data Races, Deadlocks, Ownership Types, Encapsulation

1 Introduction

Multithreaded programming is becoming a mainstream programming practice. But multithreaded programming is difficult and error prone. Multithreaded programs 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.

This paper presents a new static type system for multithreaded programs; well-typed programs in our system are guaranteed to be free of data races and deadlocks. We recently presented a static type system to prevent data races [7]. This paper extends the race-free type system to prevent both data races and deadlocks. The basic idea 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 in the form of type declarations. Our system statically verifies that a program is consistent with its 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 [17, 41].

Our system also allows programmers to use recursive tree-based data structures to further order the locks within a given 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) there is a 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 [22, 4]. 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 Ownership Types

We use a variant of ownership types [14, 13] to prevent data races and deadlocks. Ownership types provide a statically enforceable way of specifying object encapsulation. Ownership types are useful for preventing data races and deadlocks because the lock that protects an object can also protect its encapsulated objects. In recent previous work we presented PRFJ [7], a type system that uses a variant of ownership types to statically prevent data races. PRFJ is the first type system to combine ownership types with unique pointers [38]. This enables PRFJ to express constructs that neither ownership types nor unique pointers alone can express. PRFJ is also the first type system to combine ownership types with effects clauses [37]. This paper extends PRFJ to prevent both data races and deadlocks.

We have recently developed an ownership type system [6] that statically enforces object encapsulation, while supporting subtyping and constructs like iterators. Other ownership type systems either do not enforce object encapsulation (they enforce weaker restrictions instead) [12, 7, 2], or they are not expressive (they do not support subtyping and constructs like iterators) [14, 13]. We present a detailed discussion of ownership types in Section 7. We also describe how the type system in this paper can be combined with the type system in [6] to statically enforce object encapsulation as well as prevent data races and deadlocks.

1.4 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 [7, 22, 23]. 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 write code that is polymorphic in lock levels. Our system also allows programmers to specify a partial order among formal lock level parameters using `where` clauses [17, 41]. This feature enables programmers to write code in which the exact levels of some locks are not known statically, but only some ordering constraints among the unknown lock levels are known statically.

- **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 [36].

- **Partial-Orders Based on Mutable Trees:** Our system allows programmers to use recursive tree-based data structures to further order the locks within a given lock level. Our system allows mutations 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 within a given 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.

- **Runtime Ordering of Locks:** Our system supports imposing an arbitrary linear order at runtime on locks within a given lock level. Our system also provides a primitive to acquire such locks in the linear order.

```
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:k> {
  tree Node<self:k> left;
  tree Node<self:k> 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

- **Experience:** We have a prototype implementation of our system in the context of Java. Our implementation handles all the features of Java including threads, constructors, arrays, exceptions, static fields, interfaces, runtime downcasts, and dynamic class loading. To gain preliminary experience, we modified several Java libraries and multithreaded server programs and implemented them in our system. These programs exhibit a variety of sharing patterns. We found that our system is sufficiently expressive to support these sharing patterns and requires little programming overhead.

1.5 Outline

The rest of this paper is organized as follows. Section 2 introduces our type system using two examples. Section 3 presents our basic type system for preventing data races and deadlocks. Section 4 describes inference techniques that significantly reduce programming overhead. Section 5 presents extensions to our basic type system to support lock level polymorphism, condition variables, tree-based partial orders, DAG-based partial orders, and runtime ordering of locks. Section 6 describes our experience in using our type system. Section 7 contains a discussion of ownership types. Section 8 presents other related work and Section 9 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. 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, the callers of an Account method must hold the lock that protects the corresponding Account object before the callers can invoke the Account method. Without the accesses clauses, the Account methods would not have been well-typed.

Methods in our system may also contain locks clauses. The

```
P     ::=  defn* e
defn  ::=  class cn extends c body
c     ::=  cn | Object
body  ::=  {field* meth*}
meth  ::=  t mn(arg*) {e}
field ::=  [final]opt t fd = e
arg   ::=  [final]opt t x
t     ::=  c | int | boolean

e     ::=  new c | x | x = e | e.fd | e.fd = e | e.mn(e*) |
           e;e | let (arg = e) in {e} | if (e) then {e} |
           synchronized (e) in {e} | fork (x*) {e}

cn    ∈    class names
fd    ∈    field names
mn    ∈    method names
x     ∈    variable names
```

Figure 3: Grammar for Concurrent Java

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 k. The Node class has two Node fields left and right. The Nodes left and right also belong to the same lock level k. Our system 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 lower 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 statically verifies the absence of both data races and deadlocks in the above examples.

3 Basic Type System

This section describes our basic type system. To simplify the presentation of key ideas behind our approach, we describe our type system formally in the context of a core subset of Java [24] known as Concurrent Java [7, 22]. 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 [23], and has much of the same type structure and semantics as Classic Java. Figure 3 shows the grammar for Concurrent Java.

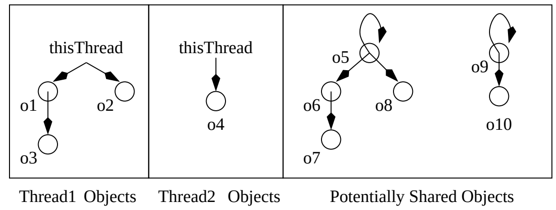

Figure 4: An Ownership Relation

O1. The owner of an object does not change over time.

O2. The ownership relation forms a forest of rooted trees, where the roots can have self loops.

O3. 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.

O4. 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

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 starting 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.

3.1 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 point to that object. The type can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent ac-

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

cesses, 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.

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 [7]. 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.[1]

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 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

[1]In our complete race-free type system [7], the owner of an object can change if there is a unique pointer to the object.

```
// 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

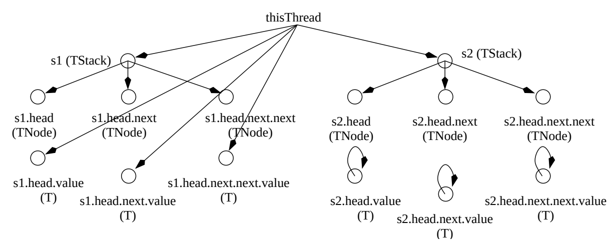

Figure 8: Ownership Relation for TStacks s1 and s2

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.

3.2 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

body	::=	$\{$*level** *field** *meth**$\}$
level	::=	LockLevel l = new | LockLevel l < *cn.l** > *cn.l**
owner	::=	*formal* | self:*cn.l* | thisThread | e_{final}
meth	::=	*t mn*(*arg**) accesses (e_{final}*) *locksclause* $\{e\}$
locksclause	::=	locks (*cn.l** $[lock]_{\text{opt}}$)
lock	::=	e_{final}
l	$\in$	lock level names

Figure 9: Grammar Extensions for Deadlock-Free Java

L1. The lock levels form a partial order.

L2. Objects that own themselves are locks. Every lock belongs to some lock level. The lock level of a lock does not change over time.

L3. 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.

L4. 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

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 extensions to our basic type system in Section 5.

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—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 belong to savingsLevel while the checkingAccount is declared to belong to checkingLevel. In the example, both the methods of CombinedAccount acquire locks in the descending

```
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

order by acquiring the lock on savingsAccount before acquiring the lock on checkingAccount.

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 a lock in addition to lock levels. If a locks clause contains an object l, then a thread that invokes the corresponding method may already hold the lock on object l. Re-acquiring the lock within the method would be redundant in that case. 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.[2] 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, 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.

3.3 Rules for 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. The core of our type system is a set of rules for reasoning about the typing judgment: $P; E; ls; l_{\min} \vdash e : t$. 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 before e is evaluated. $l_{\min}$ is the minimum

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

level among the levels of all the locks held before 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 $P; E; ls; l_{\min} \vdash e : t$ 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 and *locksclause* is the locks clause of a method.

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

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

$$ls ::= \mathsf{thisThread} \mid ls, \mathit{lock} \mid ls, \mathrm{RO}(e_{\mathrm{final}})$$

A minimum lock level $l_{\min}$ is defined as follows, where $\mathrm{LUB}(cn_1.l_1\ \ldots\ cn_k.l_k) > cn_i.l_i\ \forall_{i=1..k}$. 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.

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

The rule for acquiring a new lock using synchronized e_1 in e_2 checks that e_1 is a lock of some level $cn.l$ that is less than $l_{\min}$. If the enclosing method has a locks clause that contains a lock l, then the rule checks that either e_1 is the same object as l, or the level of e_1 is less than the level of l. 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 $e.fd$ where e is a final expression and fd is a final field.

[EXP SYNC]

$$\frac{\begin{array}{c} P; E \vdash_{\mathrm{final}} e_1 : cn'\langle \mathsf{self}{:}cn.l\ \ldots\rangle \quad P \vdash cn.l < l_{\min} \\ (E = E_1, \mathsf{locks}(\ldots\ l), E_2) \implies (P; E \vdash cn.l < \mathsf{level}(l)) \vee (l = e_1) \\ P; E; ls, e_1; cn.l \vdash e_2 : t_2 \end{array}}{P; E; ls; l_{\min} \vdash \mathsf{synchronized}\ e_1\ \mathsf{in}\ e_2 : t_2}$$

Before we proceed further with the rules, we give a formal definition for RootOwner(e). The root owner of an expression e that points 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]

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

[ROOTOWNER SELF]

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

[ROOTOWNER FINAL TRANSITIVE]

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

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 $\mathrm{RO}(e)$.

[ROOTOWNER FORMAL]

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

The rule for accessing field $e.fd$ checks that e is a well-typed expression of some 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. If the field is not final, the thread must hold 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, the rule renames this with the expression e, and the formal parameters with their corresponding actual parameters.

[EXP REF]

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

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. If the locks clause contains a lock l, the rule ensures that either the level of l is less than $l_{\min}$, or the level of l is equal to $l_{\min}$ and l is in the lock set (in which case re-acquiring l within the method is redundant). The rule appropriately renames expressions and types used outside their declared context.

[EXP INVOKE]

$$\mathrm{Renamed}(\alpha) \stackrel{\mathrm{def}}{=} \alpha[e/\mathsf{this}][o_1/f_1]..[o_n/f_n][e_1/y_1]..[e_k/y_k]$$

$$\frac{\begin{array}{c} P; E; ls; l_{\min} \vdash e : cn\langle o_{1..n}\rangle \\ P \vdash (t\ mn(t_j\ y_j\ ^{j \in 1..k})\ \mathsf{accesses}(e'*)\ \mathsf{locks}(cn.l*\ [l]_{\mathrm{opt}})\ \ldots) \\ \in cn\langle f_{1..n}\rangle \\ P; E; ls; l_{\min} \vdash e_j : \mathrm{Renamed}(t_j) \\ P; E \vdash \mathrm{RootOwner}(\mathrm{Renamed}(e'_i)) = r'_i \quad r'_i \in ls \\ P \vdash cn_i.l_i < l_{\min} \quad l_R = \mathrm{Renamed}(l) \\ P; E \vdash (\mathsf{level}(l_R) < l_{\min}) \vee (\mathsf{level}(l_R) = l_{\min}) \wedge (l_R \in ls) \end{array}}{P; E; ls; l_{\min} \vdash e.mn(e_{1..k}) : \mathrm{Renamed}(t)}$$

The rule for type checking a method assumes that the thread holds the locks on the root owners of all the final expressions specified in the accesses clause. The rules also assumes that for each lock held by the thread, the level of the lock is greater than all the levels specified in the locks clause. If the locks clause of the method contains a lock l, the rule assumes that for each lock held by the thread, either the level of the lock is greater than the level of l, or the lock is the same object as l. The rule then type checks the method body under these assumptions.

[METHOD]

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

3.4 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 [48] of type soundness can be constructed by defining an operational semantics for Safe Concurrent Java (by extending the operational semantics of Classic Java [23]) 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.

3.5 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 need not be 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. However, the extra type information available in our system can be 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 can be separately garbage collected, and when the thread dies, the space in a thread-local heap can be reclaimed at once.

```
1   class A<oa1, oa2> {...};
2   class B<ob1, ob2, ob3> extends A<ob1, ob3> {...};
3
4   class C<oc1> {
5     void m(B<this, oc1, thisThread> b) {
6       A a1;
7       B b1;
8       b1 = b;
9       a1 = b1;
10    }
11  }
```

Figure 12: An Incompletely Typed Method

4 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. This approach permits separate compilation.

4.1 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 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 unknown parameters such that the program becomes 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 [15]. 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.

4.2 Anonymous Owners

Consider the code in Figure 7. The TStack class is parameterized by thisOwner and TOwner. However, the owner pa-

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

Figure 13: Types Augmented With Unknown Owners

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

Figure 14: Constraints on Unknown Owners

rameter thisOwner is not used in the static scope where it is visible. Similarly, the owner parameter thisOwner for class T is not used in the body of class T. If a class body or a method body does not use an owner parameter, it is unnecessary to name the parameter. Our system allows programmers to use ⟨-⟩ for such anonymous owner parameters. For example, the TStack class can be declared as class TStack⟨-,TOwner⟩ {...}. The T class can be declared as class T⟨-⟩ {...}.

4.3 Default Types

In addition to supporting intra-procedural type inference and anonymous owners, 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.

5 Extensions to the Basic Type System

This section presents extensions our basic type system.

5.1 Lock Level Polymorphism

This section describes how our type system supports polymorphism in lock levels. In the type system described in Section 3, the level of each lock is known at compile-time. But programmers may sometimes want 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. 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 the grammar extensions to Safe Concurrent Java to support lock level polymorphism.

defn	::=	class *cn*⟨*owner formal**⟩ *whereclause* extends *c body*
formal	::=	*f* | self:*v*
locklevel	::=	*cn.l* | *v*
whereclause	::=	where (*locklevel* > *locklevel*)*
locksclause	::=	locks (*locklevel** $[lock]_{opt}$)
v	∈	formal lock level names

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

Programmers can parameterize classes with formal lock level parameters in addition to formal owner parameters. Programmers can specify ordering constraints among the lock level parameters using where clauses [17, 41]. Figure 16 shows part of a self-synchronized Stack implemented using the self-synchronized Vector in Figure 11. 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 verifies that the program acquires the locks in the descending order.

5.2 Condition Variables

This section describes how our system prevents deadlocks in the presence of condition variables. Java provides condition variables in the form of 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 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 the grammar extensions to Safe Concurrent Java 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

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

$$e \;::=\; \ldots \mid e.\textsf{wait} \mid e.\textsf{notify}$$

Figure 17: Grammar Extensions for Condition Variables

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

Figure 18: Grammar Extensions for Tree Ordering

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 lock set only contains the lock on e. The rules for type checking are shown below.

[EXP WAIT]

$$\frac{\begin{array}{c} E = E_1,\ \textsf{locks}(\infty\ [e]_{\text{opt}}),\ E_2 \\ P;\ E \vdash_{\text{final}} e \quad ls = \{e\} \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.\textsf{wait} : \textsf{int}}$$

[EXP NOTIFY]

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

5.3 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.

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) that 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]

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

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

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 is the formal parameter k. A Node has two tree fields left and right. The Nodes left and right own themselves and also belong to lock level k. 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 vari-

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

Figure 20: Grammar Extensions for DAG Ordering

$$\begin{array}{rcl} \textit{defn} & ::= & \mathsf{class}\ cn\langle \textit{owner formal*}\rangle\ \textit{whereclause} \\ & & \mathsf{extends}\ c\ [\textit{dynamic}]_{\text{opt}}\ \textit{body} \\ \textit{dynamic} & ::= & \mathsf{implements\ Dynamic} \\ e & ::= & \ldots \mid \mathsf{synchronized}\ (e+)\ \mathsf{in}\ \{e\} \end{array}$$

Figure 21: Grammar Extensions for Runtime Ordering

```
class Account implements Dynamic {
  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; }
}

void transfer(Account<self:v> a1, Account<self:v> a2, int x)
  locks(v) {
  synchronized (a1, a2) { a1.withdraw(x);  a2.deposit(x); }
}
```

Figure 22: Runtime Ordered Accounts

ous 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), then x' is trivially a root. Similarly, if a local variable x contains a unique pointer, then x cannot be in the tree rooted at x'.

[EXP TREE ASSIGN]

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

5.4 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.

5.5 Runtime Ordering of Locks

In the type system we described so far, the partial order between locks is known statically. However, programmers may sometimes want to write code where the order cannot be determined statically. For example, consider a transfer method that receives two self-synchronized Account objects a1 and a2. The transfer method acquires the locks on a1 and a2 and transfers money from a1 to a2. But the ordering between a1 and a2 may not be known statically within the transfer method. To avoid deadlocks in such programs, our system supports imposing an arbitrary linear order at runtime on a group of unordered locks. Our system also provides a primitive to acquire such locks in the linear order.

Figure 21 shows the grammar extensions to Safe Concurrent Java to support runtime ordering of locks. Programmers can declare a class to be a subtype of Dynamic. Objects of such classes cannot contain tree or dag edges to other objects. The runtime imposes an arbitrary linear order on Dynamic objects by assigning a unique id to each of them. For example, a runtime can choose the time of creation of an object to be its unique id. The runtime stores the unique id in every Dynamic object.

Locks of type Dynamic that belong to the same lock level are further ordered based on the linear order. Our system provides a primitive to acquire multiple Dynamic locks of the same lock level: synchronized(l_1, ..., l_n). To prevent deadlocks, the runtime sorts the locks $l_1 ... l_n$ based on the linear order and acquires the locks in the sorted order.[3] For example, in Figure 22, the locks a1 and a2 are of type Dynamic and belong to the same lock level. The synchronized statement acquires the locks in the linear order and thus avoids causing deadlocks.

6 Experience

We have a prototype implementation of our type system. Our implementation is JVM-compatible [35]. We translate well-typed programs in our system into bytecodes that can run on regular JVMs. Our implementation handles all the features of the Java language including threads, constructors, arrays, exceptions, static fields, interfaces, runtime downcasts, and dynamic class loading. 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 [7].

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 Ob-

[3] Our implementation of this feature runs on regular JVMs. We translate a synchronized statement with multiple locks into code that acquires the locks individually in the linear order. We also translate the code in constructors of Dynamic objects to store the unique ids in the objects.

jectOwner. 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 [45] to support safe runtime downcasts, but we only keep runtime ownership and lock level information for objects that are potentially involved in downcasts to types with multiple parameters. A companion technical report [5] describes how to do this efficiently without much space or time overhead. Note that our implementation of the type passing approach is JVM-compatible.

To gain preliminary experience, we implemented a number of Java programs in our system including several classes from the Java libraries. We also implemented some multithreaded server programs including *elevator*, a real time discrete event simulator [46, 11], 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 is expressive enough to support these programs. In each case, once we determined the sharing pattern of the program, adding the extra type annotations was a fairly straight forward process. On average, we had to change about one in thirty lines of code.

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.

7 Ownership Types and Encapsulation

We use a variant of ownership types [14, 13] to prevent data races and deadlocks. Ownership types provide a statically enforceable way of specifying object encapsulation. The idea is that an object may *own* other subobjects that are part of its representation. Ownership types are useful for preventing data races and deadlocks because the lock that protects an object can also protect its subobjects.

We have recently developed an ownership type system [6] that statically enforces object encapsulation, while supporting subtyping and constructs like iterators. Other ownership type systems either do not enforce object encapsulation (they enforce weaker restrictions instead) [12, 7, 2], or they are not expressive (they do not support subtyping and constructs like iterators) [14, 13]. This section presents a detailed discussion of ownership types. This section also describes how the type system in this paper can be combined with the type system in [6] to statically enforce object encapsulation as well as prevent data races and deadlocks.

7.1 Object Encapsulation

Object encapsulation gives programmers the ability to reason locally about program correctness. Reasoning about a class in an object-oriented program involves reasoning about the behavior of objects belonging to the class. Typically objects point to other *subobjects*, which are used to represent the containing object. Local reasoning about class correctness is possible if the subobjects are *fully encapsulated*, that is, if all subobjects are accessible only within the containing object. This condition supports local reasoning because it ensures that outside objects cannot interact with the subobjects without calling methods of the containing object. The containing object is thus in control of its subobjects.

However, full encapsulation is often more than is needed. Encapsulation is only required for subobjects that the containing object *depends* on [33]. An object a *depends* on subobject b if a calls methods of b and furthermore these calls expose mutable behavior of b in a way that affects the invariants of a. Thus, if a stack of items is implemented using a linked list, the stack only depends on the list but not on the items contained in the list. This is because if code outside could manipulate the list, it could invalidate the correctness of the stack implementation. But code outside can safely access the items contained in the stack because the stack does not call their methods; it only depends on the identities of the items and the identities never change. Similarly, a set of immutable elements does not depend on the elements even if it invokes a.equals(b) to ensure that no two elements a and b in the set are equal, because the elements are immutable.

Ownership types provide a statically enforceable way of specifying object encapsulation. If an object a depends on an object b, programmers can declare that a owns b. An ownership type system enforces object encapsulation if it enforces the following property:

E1. **Owners as encapsulating objects:** If object z owns object y, but z does not own object x directly or transitively, then x cannot access y.

Property E1 says that if y is *inside* the encapsulation boundary of z and x is *outside* the encapsulation boundary, then x cannot access y. An object x *accesses* an object y if methods of x obtain a pointer to y and can invoke methods of y. The pointer to y may be stored in a field of x, or in a local variable of a method of x. Consider Figure 4 for an illustration. o9 owns o10. But o9 does not own o6 directly or transitively. So o6 cannot access o10. The only objects that o6 can access are: o6 and its children, the ancestors of o6 and their children, and objects globally accessible within the thread, namely objects owned by self and thisThread.[4]

7.2 Ownership Type Systems

Ownership type systems use naming to enforce encapsulation. The type of an object includes the name of its owner. To access an object, a program fragment must name the type of that object, and hence must name the owner of that object. This section presents a discussion of the various ownership type systems and the encapsulation guarantees they provide. It also shows how to extend our type system to statically enforce object encapsulation as well as prevent data races and deadlocks.

Ownership Types [14, 13]: [14] is one of the first systems to introduce ownership types. [13] presents a formalization of the type system. These systems enforce object encapsulation, but do so by significantly limiting expressiveness. In these systems, a subtype must have the same owners as a super type. So TStack⟨thisOwner,TOwner⟩ cannot be a subtype of Object⟨thisOwner⟩. Moreover, one cannot express constructs like iterators in these systems.

Ownership Types With Subtyping [12]: JOE [12] builds on previous work in [14, 13]. JOE supports a natural form of subtyping that is similar to subtyping in parametric type systems [41, 8, 1, 45]. A subtype can have different owners than a super type. However, the first owners must match because the first owners own the corresponding object. To support subtyping, JOE enforces the constraint that in every type $T\langle o_1, ..., o_n\rangle$ with multiple owners, $(o_1 \preceq o_i)$ for all $i \in \{1..n\}$. Recall from Figure 5 that the ownership relation forms a forest of trees. The notation $(x \preceq y)$ means that either x is the same as y, or x is a descendant of y in the ownership tree, or y is the special owner self. The type TStack⟨self, this⟩ is thus illegal because (self $\not\preceq$ this). Without this constraint and with subtyping, JOE would not have provided any meaningful encapsulation guarantees. Figure 24 illustrates this with an example.

To support constructs like iterators, JOE allows programs to temporarily violate object encapsulation (Property E1). Figure 23 presents example code in JOE that violates object encapsulation. (We adopted the example from the JOE paper [12]. But we present this and other examples in our syntax, that is slightly different from the syntax in the original papers.) The example shows an iterator for the TStack

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TStackEnum<this, TOwner> elements() {
    return new TStackEnum<this, TOwner>(head);
  }
}
class TStackEnum<enumOwner, TOwner> {
  TNode<enumOwner, TOwner> curr;
  TStackEnum(TNode<enumOwner, TOwner> head)  {curr = head;}
  T<TOwner> getNext() {...} boolean hasMoreElements() {...}
}
class TStackClient<clientOwner> {
  void test() {
    TStack<this,  this> s = new TStack<this, this>;
    TStackEnum<s, this> e = s.elements(); /* Violates E1 */
  }
}   /* owner of e is instantiated with a local variable! */
```

Figure 23: Violation of Object Encapsulation in [12]

in Figure 7. In the example, the TStack object owns the iterator object. But a TStackClient object that is outside the encapsulation boundary of the TStack object accesses the iterator object, thus violating object encapsulation (Property E1). However, note that type of the iterator contains the TStack object. So the TStackClient object can access the iterator only when the TStack object is in scope. This ensures that the violation of object encapsulation is temporally bounded. JOE enforces the following weak property:

E2. **Owners as dominators:** All paths in the heap from the root object to object x must pass through x's owner.

Property E2 implies that an application thread must first access the owner o of an object x before it can access x. Furthermore, in JOE, if the thread creates a path from a local variable v to x, then either the path must go through o, or the thread must have a local variable pointing to o and the type of v must contain o.

Ownership Types for Safe Concurrent Programming: In recent previous work we described PRFJ [7], a type system that uses a variant of ownership types to statically prevent data races in multithreaded programs. In this paper, we extend the type system to also prevent deadlocks. These type systems support subtyping and constructs like iterators. Unlike JOE, they do not have the constraint that the first owner $\preceq$ all other owners. The absence of this constraint allows a program to create a path to a subobject that does not go through its owner. However, these systems have effects clauses [37] that ensure that, even though such a path may exist, the program cannot exploit the path to access the subobject unless its owner is in scope. The effects clauses require every thread to hold the lock on the root owner of an object before the thread accesses the object. The effects clauses ultimately enable these type systems to enforce the following weak encapsulation property:

E3. **Owners as capabilities:** The owner of object x must be in scope when an application accesses x.

Property E3 states that when an application accesses x, the owner of x must be accessible either through a local variable l, or through a field access $e.fd$. The application must be

[4]Note the analogy with nested procedures: proc P_1 {var x_2; proc P_2 {var x_3; proc P_3 {...}}}. Say x_{n+1} and P_{n+1} are children of P_n. Then P_n can only access: P_n and its children, the ancestors of P_n and their children, and global variables and procedures.

```
class Foo<o> { int x = 0;  void accessMe() { x++; } }

class SuperType<o> { void some_method() {} }

class SubType<o,c> extends SuperType<o> {
  Foo<c>              owner_parameter_c_owns_me;
  SubType(Foo<c> x)  {owner_parameter_c_owns_me = x;}
  void some_method() {owner_parameter_c_owns_me.accessMe();}
}

class SomeClass<o> {
  Foo<this>       f = new Foo<this>;
  SuperType<self> s = new SubType<self,this>(f);
  SuperType<self> get() {return s;}
}

class Main<o> {
  void m() {
    SuperType<self> s = null;
    {SomeClass<this> c = new SomeClass<this>; s = c.get();}
    s.some_method(); // Violates E1, E2, E3
  }
}

// SubType s is not encapsulated within SomeClass
// but some_method of SubType accesses Foo object
// owned by SomeClass:                    Therefore Violates E1

// There is path to owner_parameter_c_owns_me
// through s that does not go through c:  Therefore Violates E2

// some_method accesses owner_parameter_c_owns_me
// whose owner c is now garbage:          Therefore Violates E3
```

Figure 24: Violation of Encapsulation in [2]

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TEnumeration<enumOwner, TOwner> elements<enumOwner>()
    where (enumOwner <= stackOwner) {
    return new TStackEnum<enumOwner>;
  }
  class TStackEnum<enumOwner>
    implements TEnumeration<enumOwner, TOwner> {

    TNode<TStack.this, TOwner> current;

    TStackEnum() {
      current = TStack.this.head;
    }
    T<TOwner> getNext() {
      if (current == null) return null;
      T<TOwner> t = current.value();
      current = current.next();
      return t;
    }
    boolean hasMoreElements() {
      return (current != null);
    }
  }
}

class TStackClient<clientOwner> {
  void test() {
    TStack<this, this>       s = new TStack<this, this>;
    TEnumeration<this, this> e = s.elements();
  }
}
```

Figure 25: TStack With Iterator in [6]

able to call methods on the owner of x, or acquire the lock on the owner of x. (Property E3 thus helps us prevent data races.) The owner must be accessible either in the current stack frame or in a preceding stack frame. In the later case, an application may use a formal owner parameter to name the owner of x in the current stack frame. Note that JOE [12] also enforces Property E3. Property E3 couples the right to access a subobject with the ability to name its owner.

AliasJava [2]: AliasJava [2] uses ownership types to aid program understanding. Like other ownership type systems, AliasJava allows programmers to use ownership information to reason about aliasing. For example, if variables v_1 and v_2 are of types $\mathsf{T}\langle\mathsf{this}\rangle$ and $\mathsf{T}\langle\mathsf{x}\rangle$ respectively, where x is a formal owner parameter of the enclosing class, then one can locally infer that v_1 and v_2 are definitely not aliased because they refer to objects with different owners. Moreover, by transitively tracing the flow of the owner annotation of a variable v across method calls, one can identify all the variables that can refer to objects with the same owner as v, and thus identify all the variables that are potential aliases of v.

However, unlike other ownership type systems, AliasJava does not enforce properties like E1, E2, or E3 which either disallow violations of object encapsulation entirely or temporally limit such violations. This is because AliasJava has subtyping, but it neither has the constraint that the first owner $\preceq$ all other owners as in JOE [12], nor does it have effects clauses as in PRFJ [7] and this paper. Figure 24 presents AliasJava code that violates E1, E2, and E3. (Again, the syntax in the original paper is slightly different.) In the example, SomeClass passes its encapsulated object f to a publicly accessible object s, leading to a violation of object encapsulation (Property E1). The interaction between subtyping and ownership enables the creation of a path to f through s that does not go through f's owner. Other parts of the program can then access f using this path even if they have no relationship with f's owner. The decoupling of f from its owner is further illustrated by the fact that the program can access f even after f's owner becomes garbage.

Because AliasJava does not enforce Properties E1, E2, or E3, it is more flexible than other ownership type systems. For example, in AliasJava, an iterator object that accesses encapsulated subobjects of a collection can outlive the collection object. AliasJava thus trades off encapsulation guarantees such as E1, E2, or E3 in favor of added flexibility, while still allowing programmers to reason about aliasing.

Ownership Types With Subtyping and Iterators [6]: The ownership type systems described above either do not enforce object encapsulation (they enforce weaker restrictions instead), or they are not expressive (they do not support subtyping and constructs like iterators). Enforcing object encapsulation, while supporting subtyping and constructs like iterators, was an open problem. In a recent work [6], we provide a satisfactory solution to this problem. Consider an implementation of a stack and an iterator over the stack. The stack and the iterator *cannot* be in an ownership relation. If the stack owns the iterator, one cannot use the iterator object outside its stack object. If the iterator owns the stack, one cannot have more than one iterator object for a given stack object. In [6], we solve this problem by implementing the iterator as an inner class of the stack and allowing objects of inner classes to have privileged access to the representations of the corresponding objects of the outer classes. This approach allows programmers to express constructs like iterators and yet allows them to reason locally about the correctness of their classes. Our system allows local reasoning because programmers can reason about a class and its inner classes together as a module. Figure 25 shows an iterator implementation for the TStack in Figure 7. [6] enforces the following encapsulation property:

E1′. **Owners as encapsulating objects:** If object z owns object y, but z does not own object x directly or transitively, then x cannot access y, unless x is an inner class object of y.

Ownership Types for Concurrency and Encapsulation: The type system in this paper can be combined with the type system in [6] to statically enforce object encapsulation (Property E1′) as well as prevent data races and deadlocks. The type system in this paper must be modified as follows to enforce object encapsulation. A formal owner parameter can only be instantiated with: 1) another formal owner parameter, 2) thisThread, 3) this, 4) C.this, where C is an outer class, or 5) a lock. The relation $(x \preceq y)$ must be extended to handle thread-local variables and unique pointers as follows: either 1) x is the same as y, or 2) x is a descendant of y in the ownership tree, or 3) y is the special owner self, or thisThread, or unique.

7.3 Related Type Systems

Euclid [31] is one of the first languages that considered the problem of aliasing. [27] stressed the need for better treatment of aliasing in object-oriented programs. Early work on Islands [26] and Balloons [3] focused on *fully encapsulated* objects where all subobjects an object can access are not accessible outside the object. Universes [40] also enforces full encapsulation, except for read-only references. However, full encapsulation significantly limits expressiveness, and is often more than is needed. The work on ESC/Java pointed out that encapsulation is required only for subobjects that the containing object *depends* on [33], but ESC/Java was unable to always enforce encapsulation.

Unique Pointers: Linear types [47] and unique pointers [38] can also be used to control object aliasing. Linear types have been used in low level languages to support safe explicit memory deallocation [16] and to track resource usage [18]. Linear types and unique pointers are orthogonal to ownership types, but the two can be used in conjunction to provide more expressive type systems. PRFJ [7] is the first system to combine ownership types with unique pointers. The type system in this paper extends PRFJ. AliasJava [2] also combines ownership types with unique pointers. A type system with ownership types and unique pointers can express constructs that neither ownership types nor unique pointers alone can express, while enforcing object encapsulation. Figure 26 provides an illustration. The example is adopted from a *stock quote* server we had implemented in PRFJ [7]. Type systems without unique pointers such as JOE [12] can also express the example in Figure 26, but not without violating object encapsulation (Property E1 or E1′).

Region Types: Our ownership type system is related to the type systems for doing region-based memory management [16, 25]. In our system, objects are protected by locks. In region types, objects belong to regions. However, our system contains more information about the structure of the object graph. In our system, objects own (contain) other objects forming ownership trees. Programmers specify locks only for the roots of ownership trees. The lock that protects a root also protects all the objects in the tree. In region types, programmers directly specify the regions for all objects. Thus, the information in region types corresponds to a flattening of the ownership trees. Region types can be combined with ownership types to keep information about regions as well as object containment.

```
class StockQuoteHandler ... {
    Socket<this> s;
    StockQuoteHandler(Socket<unique> s) ... {
        this.s = s--;  // this.s = s; s = null;
    } ...
}
class Main {
    void serveQuotes(...) {
      Socket<unique> s = ...;
      StockQuoteHandler h = new StockQuoteHandler(s--);
      ...
    }
}
```

Figure 26: Quote Server That Preserves Object Encapsulation Using Ownership Types and Unique Pointers

Effects: Effects clauses [37] are useful for specifying assumptions that must hold at method boundaries. Effects enable modular checking of programs. PRFJ [7] is the first system to combine effects with ownership types to statically prevent data races. This paper uses effects with ownership types to prevent data races and deadlocks. [12] and [6] also combine effects with ownership types for program understanding and supporting safe software upgrades respectively.

Data Groups: Data groups [32, 34] can be used to name groups of objects in an effects clause to write modular specifications in the presence of subtyping. Ownership types provide an alternate way of writing modular specifications. Ownership types can also be used to name groups of objects in an effects clause—the name of an owner can be used to name all the objects transitively owned by the owner. However, because data groups are implemented using a theorem prover, data groups can be used reason more precisely about effects. *Pivot uniqueness* in [34] is similar to unique pointers [38]. Ownership types combined with unique pointers are more flexible than a system with *pivot uniqueness* because they allow arbitrarily many pointers to an encapsulated object from objects within the encapsulation boundary.

Shape Analysis: Systems such as TVLA [42], PALE [39], and Roles [30] specify the shape of a local object graph in more detail than ownership types. TVLA can verify properties such as when the input to the program is a tree, the output is also a tree. PALE can verify all the data structures that can be expressed as graph types. Roles can verify global properties such as the participation of objects in multiple data structures. In contrast to these systems that take exponential time for verification, ownership types provide a lightweight and practical way to constrain aliasing.

Parametric Types: Our ownership type system is similar to parametric type systems for Java [41, 8, 1, 45], except that our parameters are values and not types. Our type system fits naturally in a language with parameterized types.

8 Other Related Work

There has been much research on approaches to detect or prevent data races and deadlocks in multithreaded programs.

Static Tools: Tools like Warlock [44] and Sema [29] use annotations supplied by programmers to statically detect potential data races and deadlocks in a program. The Extended Static Checker for Java (ESC/Java) [19] is another annotation based system that uses a theorem prover to statically detect many kinds of errors including data races and deadlocks. [21] 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.

Dynamic Tools: There are many systems that detect data races and deadlocks dynamically. These include systems developed in the scientific parallel programming community [20], tools like Eraser [43], and tools for detecting data races in Java programs [46, 11]. 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. Besides, annotated programs are easier to understand and maintain because they explicitly contain the design decisions made by programmers.

Language Mechanisms: To our knowledge, Concurrent Pascal is the first race-free programming language [9]. Programs in Concurrent Pascal use synchronized monitors to prevent data races. But monitors in Concurrent Pascal are restricted in that threads can share data with monitors only by copying the data. A thread cannot pass a pointer to an object to a monitor. More recently, researchers have proposed type systems to prevent data races in object-oriented programs. Race Free Java [22] extends the static annotations in ESC/Java into a formal race-free type system. Guava [4] is another dialect of Java for preventing data races. Our race-free type system published earlier [7] lets programmers write generic code to implement a class, and create different objects of the same class that have different protection mechanisms. But the above systems do not prevent deadlocks. The type system in this paper extends our race-free type system [7] to prevent both data races and deadlocks.

Message Passing Systems: There are several systems that statically check for data races and deadlocks in message passing systems [28, 10]. These systems, however, use a different programming model. For example, programs in these systems do not access shared objects in a heap.

9 Conclusions

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 within a given lock level. The type checker statically verifies that whenever a thread holds more than one lock, the thread acquires the locks in the descending order. The type checker uses an intra-procedural intra-loop flow-sensitive analysis to check that mutations to trees used for describing the partial order 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. This paper also describes how to extend our type system to statically enforce object encapsulation as well as prevent data races and deadlocks. We have implemented our type system for Java. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead.

Acknowledgments

We are grateful to Viktor Kuncak, Jinyang Li, Rodrigo Rodrigues, Alexandru Salcianu, and the anonymous referees for their comments. We also thank Jonathan Aldrich for having lengthy discussions with us on AliasJava. This research was supported in part by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, NSF Grant CCR00-73513, and the Singapore-MIT Alliance.

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] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[3] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1997.

[4] 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.

[5] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types and safe lazy upgrades in object-oriented databases. Technical Report TR-858, MIT Laboratory for Computer Science, July 2002.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[13] 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.

[14] 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.

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

[16] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Principles of Programming Languages (POPL)*, January 1999.

[17] 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.

[18] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation (PLDI)*, June 2001.

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

[20] 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.

[21] 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.

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

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

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

[25] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[26] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1991.

[27] J. Hogg, D. Lea, A. Wills, and D. de Champeaux. The Geneva convention on the treatment of object aliasing. In *OOPS Messenger 3(2)*, April 1992.

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

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

[30] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Principles of Programming Languages (POPL)*, January 2002.

[31] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. In *Sigplan Notices, 12(2)*, February 1977.

[32] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[33] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.

[34] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[35] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

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

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

[38] N. Minsky. Towards alias-free pointers. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1996.

[39] A. Moeller and M. I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[40] P. Muller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming.* 1999.

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

[42] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Transactions on Programming Languages and Systems (TOPLAS) 20(1)*, January 1998.

[43] 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.

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

[45] 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.

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

[47] P. Wadler. Linear types can change the world. In M. Broy and C. Jones, editors, *Programming Concepts and Methods.* 1990.

[48] 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 3. The grammar for the type system is shown below.

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

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 [23] and [22]. 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 $E ::= \emptyset \mid E,\ [\textsf{final}]_{\text{opt}}\ t\ x \mid E,\ \textsf{owner}\ f \mid E,\ \mathit{locksclause}$
A lock set is defined as $ls ::= \textsf{thisThread} \mid ls,\ \mathit{lock} \mid ls,\ \mathrm{RO}(e_{\text{final}})$; where $\mathrm{RO}(e)$ is the root owner of e
A minimum lock level is defined as $l_{\min} ::= \infty \mid cn.l \mid \mathrm{LUB}(cn_1.l_1\ \ldots\ cn_k.l_k)$; where $\mathrm{LUB}(cn_1.l_1\ \ldots\ cn_k.l_k) > cn_i.l_i\ \forall_{i=1..k}$
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
$P \vdash_{\text{level}} cn.l$	$cn.l$ is a well-formed lock level
$P \vdash cn_1.l_1 < cn_2.l_2$	$cn_1.l_1$ is less than $cn_2.l_2$ in the partial order formed by lock levels
$P \vdash cn.l < l_{\min}$	$cn.l$ is less than $l_{\min}$ in the partial order formed by lock levels
$P; E \vdash \mathrm{level}(e) = cn.l$	e is a final expression that owns itself and the lock level of e is $cn.l$
$P; E \vdash \mathrm{level}(e) < l_{\min}$	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
$P \vdash \mathit{field} \in cn\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits *field*
$P \vdash \mathit{meth} \in cn\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits *meth*
$P; E \vdash \mathit{meth}$	*meth* is a well-formed method
$P; E \vdash \mathrm{RootOwner}(e) = r$	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

$\vdash P : t$

[PROG]

$$\frac{\begin{array}{c} ClassOnce(P) \quad WFClasses(P) \quad FieldsOnce(P) \\ MethodsOnce(P) \quad OverridesOK(P) \quad LockLevelsOK(P) \\ P = defn_{1..n}\ e \quad P \vdash defn_i \quad P;\ \emptyset;\ \mathsf{thisThread};\ \infty \vdash e : t \end{array}}{\vdash P : t}$$

$P \vdash defn$

[CLASS]

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

$P;\ E \vdash wf$

[ENV $\emptyset$]

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

[ENV OWNER]

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

[ENV X]

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

$P;\ E \vdash t$

[TYPE INT]

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

[TYPE BOOLEAN]

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

[TYPE OBJECT]

$$\frac{P;\ E \vdash_{\mathrm{owner}} o}{P;\ E \vdash \mathsf{Object}\langle o\rangle}$$

[TYPE SHARED CLASS]

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

[TYPE THREAD-LOCAL CLASS]

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

[TYPE C]

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

$P;\ E \vdash t_1 <: t_2$

[SUBTYPE REFL]

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

[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]

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

$P;\ E \vdash_{\mathrm{owner}} o$

[OWNER THISTHREAD]

$$\frac{}{P;\ E \vdash_{\mathrm{owner}} \mathsf{thisThread}}$$

[OWNER OTHERTHREAD]

$$\frac{}{P;\ E \vdash_{\mathrm{owner}} \mathsf{otherThread}}$$

[OWNER SELF]

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

[OWNER EXP]

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

[OWNER FORMAL]

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

$P \vdash_{\mathrm{level}} cn.l$

[LEVEL]

$$\frac{P \vdash \mathsf{class}\ cn...\ \{...\ \mathsf{Locklevel}\ l\ ...\}}{P \vdash_{\mathrm{level}} cn.l}$$

$P \vdash cn_1.l_1 < cn_2.l_2$

[LEVEL <]

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

[LEVEL >]

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

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

[LEVEL < INFTY]

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

[LEVEL < LUB]

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

[LEVEL < CN.L]

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

[LEVEL TRANS]

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

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

[LEVEL(EXP)]

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

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

[LEVEL < LEVEL MIN]

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

$P;\ E \vdash_{\mathrm{final}} e$

[FINAL VAR]

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

[FINAL REF]

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

$P;\ E \vdash field\ init$

[FIELD INIT]

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

$P \vdash field \in c$

[FIELD DECLARED]

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

[FIELD INHERITED]

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

$P \vdash meth \in c$

[METHOD DECLARED]

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

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

[METHOD INHERITED]

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

[METHOD]

$$\frac{\begin{array}{c} E' = E,\ arg_{1..n},\ \mathsf{locks}(cn_j.l_j\ ^{j\in 1..k}\ [lock]_{\mathrm{opt}}) \\ P;\ E' \vdash_{\mathsf{final}} e_i : t_i \quad P;\ E' \vdash \mathrm{RootOwner}(e_i) = r_i \quad ls = \mathsf{thisThread},\ r_{1..r} \\ l_{\min} = \mathrm{LUB}(cn_j.l_j\ ^{j\in 1..k}) \quad P;\ E';\ ls;\ l_{\min} \vdash e : t \end{array}}{P;\ E \vdash t\ mn(arg_{1..n})\ \mathsf{accesses}(e_{1..r})\ \mathsf{locks}(cn_j.l_j\ ^{j\in 1..k}\ [lock]_{\mathrm{opt}})\ \{e\}}$$

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

[ROOTOWNER THISTHREAD]

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

[ROOTOWNER OTHERTHREAD]

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

[ROOTOWNER SELF]

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

[ROOTOWNER FINAL TRANSITIVE]

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

[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]

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

[EXP VAR]

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

[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]

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

[EXP IF]

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

[EXP REF]

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

[EXP ASSIGN]

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

[EXP SYNC]

$$\frac{\begin{array}{c} P;\ E \vdash \mathrm{level}(e_1) = cn.l < l_{\min} \\ (E = E_1,\ \mathsf{locks}(...\ l),\ E_2) \implies (P;\ E \vdash cn.l < \mathrm{level}(l)) \vee (l = e_1) \\ P;\ E;\ ls,\ e_1;\ cn.l \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash \mathsf{synchronized}\ e_1\ \mathsf{in}\ e_2 : t_2}$$

[EXP SYNC REDUNDANT]

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

[EXP INVOKE]

$$\frac{\begin{array}{c} \mathrm{Renamed}(\alpha) \stackrel{\mathrm{def}}{=} \alpha[e/\mathsf{this}][o_1/f_1]..[o_n/f_n][e_1/y_1]..[e_k/y_k] \\ P;\ E;\ ls;\ l_{\min} \vdash e : cn\langle o_{1..n}\rangle \\ P \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{accesses}(e'*)\ \mathsf{locks}(cn.l*\ [lock]_{\mathrm{opt}})\ ...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls;\ l_{\min} \vdash e_j : \mathrm{Renamed}(t_j) \\ P;\ E \vdash \mathrm{RootOwner}(\mathrm{Renamed}(e'_i)) = r'_i \quad r'_i \in ls \\ P \vdash cn_i.l_i < l_{\min} \\ lock_R = \mathrm{Renamed}(lock) \\ P;\ E \vdash (\mathrm{level}(lock_R) < l_{\min}) \vee (\mathrm{level}(lock_R) = l_{\min}) \wedge (lock_R \in ls) \end{array}}{P;\ E;\ ls;\ l_{\min} \vdash e.mn(e_{1..k}) : \mathrm{Renamed}(t)}$$

[EXP FORK]

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