A Parameterized Type System for Race-Free Java Programs

Chandrasekhar Boyapati Martin Rinard

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

Abstract

This paper presents a new static type system for multithreaded programs; any well-typed program in our system is free of data races. Our type system is significantly more expressive than previous such type systems. In particular, our system lets programmers write generic code to implement a class, then create different objects of the same class that have different protection mechanisms. This flexibility enables programmers to reduce the number of unnecessary synchronization operations in a program without risking data races. We also support default types which reduce the burden of writing the extra type annotations. Our experience indicates that our system provides a promising approach to make multithreaded programs more reliable and efficient.

1 Introduction

The use of multiple threads of control is quickly becoming a mainstream programming practice. But interactions between threads can significantly complicate the software development process. Multithreaded programs typically synchronize operations on shared data to ensure that the operations execute atomically. Failure to correctly synchronize such operations leads to *data races*, which occur when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write.

Because data races can be one of the most difficult programming errors to detect, reproduce, and eliminate, many researchers have developed tools that help programmers detect or eliminate data races. These tools include systems that monitor the execution of a program to dynamically detect potential races [18, 33], static race detection systems [34, 26, 17], and formal type systems that ensure race-free programs [9, 21, 19, 20, 3].

This paper presents a new static type system for multithreaded object-oriented programs; this type system guarantees that any well-typed program is free of data races. Every object in our system is associated with a *protection mechanism* that ensures that accesses to the object never create data races. Our system enables programmers to specify the protection mechanism for each object as part of the type of the variables that refer to that object. The type can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses [4], 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 and is not shared between threads, or 3) the variable contains the unique reference to the object. The type checker then uses these type specifications to statically verify that a program uses objects only in accordance with their declared protection mechanisms.

Unlike previously proposed type systems for race-free programs [21, 19, 20, 3], our type system also 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.

Without this flexibility, programmers often must either write a program that acquires redundant locks just to satisfy the type checker, or unnecessarily duplicate code to produce multiple versions of the same classes; these versions differ only in the code that implements the protection mechanisms.

One of the challenges in designing an effective type system is to make it powerful enough to express common programming paradigms. One trivial way to guarantee race-free programs, for example, is to require every thread to acquire the lock on every object before accessing the object. But that would introduce an unnecessary synchronization overhead because programmers often know from the logic of their programs that acquiring certain locks is not necessary.

Our type system is expressive enough to verify the absence of races in many common situations where a thread accesses an object without acquiring the lock on that object. In particular, it accommodates the following cases:

The research was supported in part by DARPA/AFRL Contract F33615-00-C-1692, NSF Grant CCR00-86154, and NSF Grant CCR00-73513.

OOPSLA'01, October 14-18, 2001, Tampa, Florida, USA.

- **Thread-local objects:** If an object is accessed by only one thread, it needs no synchronization.

  Consider, for example, a Vector class. In our system, programmers can write a generic Vector implementation. Some Vector objects can then be created to be thread-local — these objects can be accessed without any synchronization. Other Vector objects can be shared between multiple threads — these objects will contain their own locks that must be acquired by a thread before the thread accesses the objects.

  Moreover, a program can also create thread-local Vector objects containing only thread-local elements, and thread-local Vector objects containing shared elements, all from the same generic Vector implementation.

  With previous systems, the only way to do this is to have different versions of the Vector class, one for each case. These versions contain the exact same code except for synchronization operations.

- **Objects contained within other objects:** Sometimes, an object is contained within an enclosing data structure. In such cases, it might be redundant to acquire the lock on that object since the same lock that protects the enclosing data structure also protects that object.

  Consider, for example, a Stack implementation that internally uses a Vector. In our system, a program can create a Vector object from the same generic Vector implementation such that the Vector object inherits the protection mechanism of the enclosing Stack object.

  If the program then creates a Stack object that is thread-local, no synchronization operations will be necessary to access the Stack or the Vector. If the program creates a shared Stack object, the same lock that protects the Stack will also protect the Vector.

  Previous systems needed multiple Vector and Stack implementations to support these different cases.

- **Objects migrating between threads:** Some programs use serially-shared objects that migrate from one thread to another. Although these objects are shared by multiple threads, they are accessed only by a single thread at a time. Operations on these objects can therefore execute without synchronization. Our type system uses the notion of unique pointers [29, 7, 23] to support this kind of sharing.

  Our system also supports a novel technique that lets the programs build collection classes that contain unique objects. For example, programmers can implement a generic Queue class and use it to create a Queue of unique objects. This is useful in a producer-consumer paradigm where producer threads insert items into the Queue and consumer threads extract them from the Queue.

- **Read-only objects:** Programs often use read-only objects that are initialized once by a single thread, then read by multiple threads. Because none of the parallel threads writes a read-only object after it is initialized, they can all access the object concurrently without synchronization and without data races. Our system supports this sharing pattern.

Because our type system is expressive and yet guarantees race-free programs, programmers can apply efficient protection mechanisms without risking synchronization errors. For example, the Java libraries contain two different classes to implement resizable arrays: the Vector class and the ArrayList class. The methods in the Vector class are synchronized, therefore, multiple threads can use Vector objects without creating data races. But Vectors always incur a synchronization overhead, even when used in contexts where synchronization is unnecessary. On the other hand, the methods in the ArrayList class are not synchronized, therefore, ArrayLists do not incur any unnecessary synchronization overhead. But programs that use ArrayLists risk data races because there is no mechanism in Java to ensure that ArrayLists are accessed with appropriate synchronization when used in multithreaded contexts.

Our system enables programmers to implement a single generic resizable array class. If a program creates a resizable array object to be concurrently shared between threads, our system ensures that accesses to the array are synchronized. If an array is not concurrently shared, our system allows the program to access the array without synchronization.

Our system also provides default types that reduce the burden of writing the extra type annotations. In particular, single-threaded programs incur almost no programming overhead. We implemented several multithreaded Java programs in our system. Our experience shows that our system is sufficiently expressive and requires little programming overhead.

Finally, we note in passing that any type system that guarantees race freedom also eliminates issues associated with the use of weak memory consistency models [31]. A detailed explanation of this issue can be found in [3].

The rest of this paper is organized as follows. We present our type system in the context of a core subset of Java called Concurrent Java [21]. Section 2 presents this subset. Sections 3, 4, and 5 describe the basic type system that supports thread-local objects and objects contained within other objects. Section 6 presents the type checking rules. Section 7 describes how we provide default types. Section 8 shows how the type system can be extended to handle unique pointers and read-only objects. Section 9 describes our experience in using our type system. Section 10 contains related work, and Section 11 presents our conclusions.

2 A Core Subset of Java

This section presents a variant of Concurrent Java [21], which is a multithreaded subset of Java [24, 27] with formal semantics. Our type system is designed in the context of Concurrent Java. Section 5 describes how the type system for Concurrent Java can be extended to guarantee race-free programs.

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

Figure 1: The Grammar for Concurrent Java

```
class Account {
  int balance = 0;
  int deposit(int x) {
    this.balance = this.balance + x;
  }
}
Account a1, a2;

a1 = new Account;
a1.deposit(10);

a2 = new Account;
fork (a2) { synchronized (a2) in { a2.deposit(10); } };
fork (a2) { synchronized (a2) in { a2.deposit(10); } };
```

Figure 2: A Program With an Account Class

Concurrent Java is an extension to a sequential subset of Java known as Classic Java [22], and has much of the same semantics as Classic Java. 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. Figure 1 shows the grammar for Concurrent Java. Figure 2 presents an example program written in Concurrent Java. For simplicity, all the examples in this paper use an extended language that is syntactically closer to Java.

The expression fork (e^*) e spawns a new thread with arguments (e^*) to evaluate e. The evaluation is performed only for its effect; the result of e is never used.

The expression synchronized e_1 in e_2 works as in Java. e_1 should evaluate to an object. The lock on that object is held while e_2 is evaluated. The result of evaluating e_2 is returned.

In the Account example in Figure 2, two threads access Account a2 concurrently, and must therefore synchronize their accesses to a2 to prevent data races. Account a1 is thread-local, therefore the main thread can access a1 without synchronization.

With previous type systems for race-free programs [21, 3], one has to either declare Account to be a shared class, in which case the main thread has to acquire the lock on a1 before accessing it, or one has to declare Account to be a thread-local class, in which case two threads cannot access a2 concurrently. But as we show in Section 4, our type system is expressive enough that one can add type annotations to make the above Account program be well-typed.

3 Object Ownership

The next few sections present our basic type system that supports thread-local objects and objects protected by mutual exclusion locks. Section 8 describes how our basic type system can be extended to handle unique pointers and read-only objects.

The key to our type system is the concept of object ownership. This resembles the notion of ownership types described in [13, 12], even though there it was motivated by software engineering principles and was used to restrict object aliasing.

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 3 presents an example. We draw an arrow from object x to object y if object x owns object y. In the figure, the thisThread owner of Thread 1 transitively owns objects o_1, o_2, and o_3, the thisThread owner of Thread 2 owns object o_4, object o_5 transitively owns objects o_5, o_6, o_7, and o_8, and object o_9 owns objects o_9 and o_{10}.

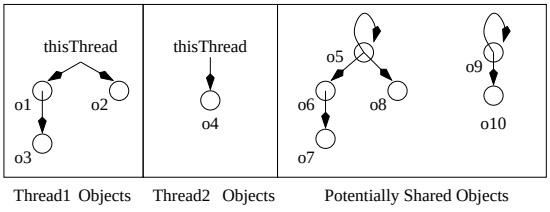

Figure 3: An Ownership Relation

Our ownership relation has the following properties:

1. The owner of an object does not change over time.
2. The ownership relation forms a forest of rooted trees, where the roots can have self loops.

In our system, it is necessary and sufficient for a thread to acquire the lock on the root of an ownership tree to gain exclusive access to all the members in the tree. Moreover, every thread implicitly holds the lock on the corresponding thisThread. Thus, a thread can access any object owned by the corresponding thisThread without explicitly acquiring any locks.

4 An Example

This section introduces our type system using an example. Section 5 describes the semantics of our type system in

```
// thisOwner owns the Account object
class Account<thisOwner> {
  int balance = 0;
  int deposit(int x) requires (this) {
    this.balance = this.balance + x;
  }
}

// Acount a1 is owned by this thread, so it is thread-local
Account<thisThread> a1 = new Account<thisThread>;
a1.deposit(10);

// Account a2 owns itself, so it can be shared between threads
final Account<self> a2 = new Account<self>;
fork (a2) { synchronized (a2) in { a2.deposit(10); } }
fork (a2) { synchronized (a2) in { a2.deposit(10); } }

final Account<self> a3 = new Account<self>;
Account<a3>         a4 = new Account<a3>;
```

Figure 4: A Parameterized Account Class

greater detail.

The language we use is an extension to Concurrent Java that we described in Section 2. We refer to our language as Parameterized Race Free Java, or PRFJ.

A class definition in PRFJ is parameterized by a list of owners. Our way of parameterizing is similar to the proposals for parametric types for Java [30, 8, 1, 10]. The difference is that the parameters are values and not other types.[1]

Figure 4 shows an example in which the Account class is parameterized by thisOwner. thisOwner owns the this object. In general, the first formal parameter of a class always owns the this object.

In the case of variable a1, the special thisThread owner is used to instantiate the Account class. Thus, thisThread owns a1, and hence a1 is local to the main thread. In the case of a2, the special self owner is used to instantiate the Account class. This means that a2 owns itself, so it can be potentially shared between threads.

In PRFJ, methods can require callers to hold locks on some objects using the requires clause. In the example, the deposit method requires every thread to hold the lock on the (root owner of the) Account object before calling the deposit method. This ensures that there will be no data races when the deposit method is called. Without the requires clause, the deposit method would not have been well-typed.

In the example, all the threads that call the deposit method on a2 first acquire the lock on a2. For a1, however, the main thread implicitly holds the lock on the thisThread owner that owns a1. Hence, it does not explicitly acquire any locks before calling the deposit method on a1.

In addition to object fields, we allow method-local variables also to be declared final. A final field or variable cannot be written into after it is initialized. In our system, an object field or a method-local variable can also be used to instantiate a class, but only if the field or variable is declared final and is already initialized. The example in Figure 4 shows how a class can be instantiated with a final variable. The Account object a4 is owned by the Account object a3.

Requiring fields and variables that instantiate a class to be declared final ensures that the owner of an object does not change over time. Requiring that they be already initialized guarantees that any object that is newly created using new is owned by an already existing object (or by itself, if we instantiate its class by self, or by thisThread). This ensures that there are no cycles in the ownership relation except for self loops. Thus, our type system preserves the ownership properties we presented earlier in Section 3.

In addition, a thread can access an object only if the thread can name the object's type. But since the type name contains the name of the owner, a thread cannot access an object without being able to name its owner. In particular, thisThread refers to different owners in different threads. Variables in one thread cannot name the thisThread of another thread, and hence cannot refer to objects local to another thread.

5 Informal Semantics

Figure 5 shows how to obtain the grammar for Parameterized Race Free Java (PRFJ) by extending the grammar for Concurrent Java.

defn	::=	class $cn\langle firstowner\ \ formal^*\rangle$ extends c *body*
c	::=	$cn\langle owner+\rangle$ | Object$\langle owner\rangle$
meth	::=	$t\ mn(arg^*)$ requires $(e_{\text{final}}{}^*)$ $\{local^*\ e\}$
firstowner	::=	*formal* | self | thisThread
owner	::=	*formal* | self | thisThread | e_{final}
local	::=	$t\ y$ | $[\text{final}]_{\text{opt}}\ t\ y = e$
e_{final}	::=	e
formal	$\in$	formal owner name

Figure 5: Extensions to Concurrent Java to Obtain PRFJ

The rest of this section presents an informal semantics of our type system and motivates our design using examples. Figure 6 shows a stack[2] of objects of type T written in PRFJ. The stack is implemented using a linked list. Section 7.1 later describes how some of the type annotations can be automatically inferred by the system.

5.1 Parameterizing Classes

Every class in PRFJ is parameterized with one or more parameters. However, the first parameter always owns the this object. The TStack class in Figure 6 is parameterized by two owners — the owner of the stack itself, and the owner of the elements in the stack.

[1] Race Free Java [21] also uses types parameterized with values. We describe how it differs from our type system in Section 10.

[2] If we had parameterized types in the language [30, 8], then the Stack declaration would have looked like the following: `class Stack<thisOwner>[ T<TOwner> ] {...}`

```
1   // thisOwner owns the TStack object
2   // TOwner owns the T objects in the stack.
3
4   class TStack<thisOwner, TOwner> {
5
6     TNode<this, TOwner> head = null;
7
8     void push(T<TOwner> value) requires (this) {
9       TNode<this, TOwner> newNode =
10        new TNode<this, TOwner>;
11      newNode.init(value, head);
12      head = newNode;
13    }
14    T<TOwner> pop() requires (this) {
15      if (head == null) return null;
16      T<TOwner> value = head.value();
17      head = head.next();
18      return value;
19    }
20  }
21
22  class TNode<thisOwner, TOwner> {
23
24    T<TOwner> value;
25    TNode<thisOwner, TOwner> next;
26
27    void init(T<TOwner> v, TNode<thisOwner, TOwner> n)
28      requires (this) {
29      this.value = v;
30      this.next  = n;
31    }
32    T<TOwner> value() requires (this) {
33      return value;
34    }
35    TNode<thisOwner, TOwner> next() requires (this) {
36      return next;
37    }
38  }
39
40  class T<thisOwner> { int x=0; }
41
42  T<thisThread> t1 = new T<thisThread>;
43  T<self>       t2 = new T<self>;
44
45  TStack<thisThread, thisThread> s1 =
46    new TStack<thisThread, thisThread>;
47  TStack<thisThread, self>       s2 =
48    new TStack<thisThread, self>;
49  final TStack<self, self>       s3 =
50    new TStack<self, self>;
51
52  ...
53  s1.push(t1);
54  s2.push(t2);
55  fork (s3,t2) {synchronized (s3) in {s3.push(t2);}}
56  fork (s3,t2) {synchronized (s3) in {s3.push(t2);}}
```

Figure 6: A Stack of T Objects

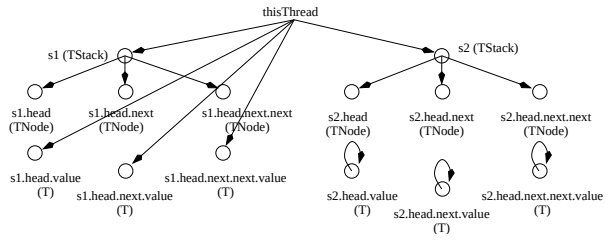

Figure 7: Ownership Relation for TStacks s1 and s2

5.2 Instantiating Classes

As we discussed in Section 4, we can instantiate a class with self, or thisThread, or final fields, or final variables. In addition, we can use the formal parameters of a class to instantiate other classes, and thus propagate the ownership information. This flexibility enables us to do coarse-grained locking, where all the objects in a compound data structure are guarded by a single lock.

In Figure 6, a TStack object owns the TNode object referred to by head. In the TNode class, the owner of the TNode object owns the next object. Thus, the TStack object owns all the nodes in the linked list. The ownership relation for the TStack s1 is depicted in Figure 7 (assuming the stack contains three elements).

5.3 Requires Clauses

Methods can require every thread to hold locks on zero or more objects before the method is invoked.

The value and next methods in the TNode class require every thread to hold the lock on the this (TNode) object before invoking those methods. However, as explained in Section 3, a thread needs to hold the lock on the root of an ownership tree to gain exclusive access to all the members in a tree. Thus, the right way to interpret the requires clause on the value and next methods is that they require every thread to hold the lock on the root of the ownership tree that contains the TNode object. This is true for all requires clauses.

The pop method in TStack assumes that the calling thread holds the lock on the (root) owner of the TStack object, as specified in its requires clause. The TStack object in turn owns all the TNode objects in it. Hence it is legal for pop to call the value and next methods of TNode without using synchronization operations.

5.4 The self Owner

Based on the discussion so far, all the T objects in any particular TStack are always owned by the same TOwner. However, we might sometimes want to create a stack of T objects that own themselves. To enable this kind of programming, we use the self owner.

When an object is owned by self, it means that the object owns itself. In Figure 6, the type for TStack s2 is instantiated with self for TOwner. The ownership relation for the TStack s2 is depicted in Figure 7 (assuming the stack contains

three elements). Since the formal TOwner is instantiated with actual self, all objects that were declared to be owned by TOwner are owned by themselves. TStack s2 thus is a thread-local stack containing potentially shared objects that own themselves. The self owner enables this kind of programming.

5.5 The thisThread Owner

Section 4 showed how a type could be instantiated with thisThread. The thisThread owner comes into scope at the beginning of each thread. It is meaningful to use thisThread only for local variables — the local variables before the initial expression of a thread, the local variables within methods, and method arguments. In particular, it would be illegal to instantiate the type of an object field with thisThread because thisThread does not refer to any particular thread at that point. Figure 8 shows an illegal usage of thisThread.

```
class CombinedAccount<thisOwner> {
  Account<thisThread> savingsAccount;  // illegal
  Account<thisThread> checkingAccount; // illegal
}
```

Figure 8: Illegal Usage of thisThread

5.6 Self-Synchronized Classes

Sometimes, we might want to specify in a class declaration that instances of the class always own themselves. Consider, for example, a SharedAccount class where the deposit method is synchronized, so that the callers of the deposit method do not have to acquire any locks. This is shown in Figure 9, where the SharedAccount class extends the Account class presented in Figure 4.

If a SharedAccount object were owned by some other object, then it would have been necessary to hold the lock on the root owner of the SharedAccount object to access the SharedAccount object. This is because some other thread might acquire the lock on the root owner and access the balance field of the SharedAccount object directly.

Thus, the deposit method with the empty requires clause will type check only if the SharedAccount class is declared to be always owned by self. To enable this, we allow the first parameter in a class declaration to be self. This feature lets us implement self-synchronized classes in our system. Figure 9 shows an example.

```
class SharedAccount<self> extends Account<self> {
  int deposit(int x) requires () {
    synchronized (this) in { super.deposit(x); }
  }
}
SharedAccount<self> a = new SharedAccount<self>;
fork (a) { a.deposit(10); }
fork (a) { a.deposit(10); }
```

Figure 9: A Self-Synchronized Account Class

Note that it is highly unusual to have a type system where a constant value is used instead of a formal parameter. But it is necessary in our case because the first parameter in our system is special, in that it owns the this object.

5.7 Thread-Local Classes

We also allow the first parameter in a class declaration to be thisThread. Such a class can only be instantiated with thisThread as the first owner. All instances of such classes would be thread-local.

6 Type Checking PRFJ Programs

Section 5 presented the grammar for PRFJ. This section describes some of the important rules for type checking. The full set of rules can be found in the appendix.

6.1 Rules for Type Checking

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

The judgment $P; E \vdash e : t$ states that e is of type t, while the judgment $P; E; ls \vdash e : t$ states that e is of type t and that ls contains the necessary locks to safely evaluate e.

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

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

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

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

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

[EXP FORK]

$$\frac{P;\ E;\ ls \vdash e_i : t_i \quad g_i = \text{final}\ t_i\ e_i \qquad P;\ g_i\ [\text{otherThread/thisThread}],\ local_{1..l};\ \text{thisThread} \vdash e : t}{P;\ E;\ ls \vdash \text{fork}\ (e_{1..n})\ \{local_{1..l}\ e\} : \text{int}}$$

The rule for synchronized e_1 in e_2 checks that e_1 is a final expression of some type t_1 and then type checks e_2 in an extended lock set that includes e_1. A final expression is either a final variable, or a field access $e.fd$ where e is a final expression and fd is a final field.

[EXP SYNC]

$$\frac{P;\ E \vdash_{\text{final}} e_1 : t_1 \qquad P;\ E;\ ls,\ e_1 \vdash e_2 : t_2}{P;\ E;\ ls \vdash \text{synchronized}\ e_1\ \text{in}\ e_2 : t_2}$$

Note that if e_1 is not a root owner, that is, if e_1 does not own itself, then acquiring the lock on e_1 is redundant. Our type system does not prevent this, even though we could modify our type rules to disallow such redundant locking.

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

[ROOTOWNER SELF]

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

[ROOTOWNER THISTHREAD]

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

[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 RO(e). [3]

[ROOTOWNER FORMAL]

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

The rule for accessing field *e.fd* checks that e is a well-typed expression of some class type $cn\langle o_{1..n}\rangle$, where $o_{1..n}$ are actual owner parameters. It verifies that the class cn with formal parameters $f_{1..n}$ declares or inherits a field fd of type t and that we do have the lock on the root owner of e.

Since t is declared inside the class, it might contain occurrences of this and the formal class parameters. When t is used outside the class, we have to rename this with the expression e, and the formal parameters with their corresponding actual parameters.

[EXP REF]

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

The rule for invoking a method checks that the arguments are of the right type and ensures that we hold the locks on the root owners of all final expressions in the requires clause of the method. The expressions and types used inside the method have to be renamed appropriately when used outside their class.

[EXP INVOKE]

$$\frac{\begin{array}{c}P;\ E;\ ls \vdash\ e\ :\ cn\langle o_{1..n}\rangle \\ P;\ E \vdash (t\ mn(t_j\ x_j\ ^{j\in 1..k})\ \mathsf{requires}\ (e'_{1..m})\ ...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash\ e_j\ :\ t_j[e/\mathrm{this}][o_1/f_1]..[o_n/f_n] \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]) = r'_i \quad r'_i \in ls\end{array}}{P;\ E;\ ls \vdash\ e.mn(e_{1..k})\colon\ t[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]}$$

The rule for type checking a method verifies that the method body is well typed under the assumption that all the locks specified in the requires clause are held. When a method specifies that it requires the lock on some object o, it really means that it requires the lock on the root owner of o.

[METHOD]

$$\frac{\begin{array}{c}g_i = \mathrm{final}\ arg_i \quad P;\ E,\ g_{1..n} \vdash_{\mathrm{final}}\ e_i\ :\ t_i \\ P;\ E,\ g_{1..n} \vdash \mathrm{RootOwner}(e_i) = r_i \\ P;\ E,\ g_{1..n},\ local_{1..l};\ \mathrm{thisThread},\ r_{1..m} \vdash\ e\ :\ t\end{array}}{P;\ E \vdash\ t\ mn(arg_{1..n})\ \mathsf{requires}\ (e_{1..m})\{local_{1..l}\ e\}}$$

6.2 Soundness of the Type System

Our type checking rules ensure that for a program to be well-typed, an object can be read or written by a thread only if the thread holds the lock on the root owner of that object. But since a lock can be held by at most one thread at a time, a well typed program in our system will not have any data races.

A complete proof of this can be constructed by defining an operational semantics for PRFJ (by extending the operational semantics of Classic Java [22]) and then proving the generalized subject reduction theorem, that states that the semantic interpretation of a term's type is invariant under reduction. The proof is straight-forward but tedious, so it is omitted here.

6.3 Runtime Overhead

PRFJ is a statically typed system. The ownership relations are used only for compile-time type checking and are not preserved at runtime. Consequently, PRFJ programs have no runtime overhead when compared to regular (Concurrent) Java programs.

In fact, one way to compile and run a PRFJ program is to convert it into a (Concurrent) Java program after type checking, by removing the type parameters and the requires clauses from the program.

However, the extra type information in a PRFJ program 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.

7 Default Types

The previous sections introduced our basic type system. This section shows how we can reduce the burden of writing these extra type annotations.

[3] Thus, while type checking the deposit method in the Account class in Figure 4, we use the term RO(this) as the root owner of this.

7.1 Inferring Owners of Local Variables

In our system, it is not necessary 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 PRFJ programs.

We illustrate our algorithm with an example. Figure 10 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.

```
 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 10: An Incompletely Typed Method

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 11 shows augmented types for the example in Figure 10. The goal of the inference algorithm is to find known owner parameters that can be used in place of the each of the unknown owner parameters to make the program be well-typed.

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

Figure 11: Types Augmented With Unknown Owner Parameters

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 12 shows the constraints imposed by Statements 8 and 9 in the example in Figure 10.

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

Figure 12: Constraints on Owner Parameters

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 [14].

If the solution is inconsistent, that is, if any two known owner parameters are constrained to be equal to one another by the solution, then the inference algorithm returns an error and the program does not type check. Otherwise, if the solution is incomplete, that is, if there is no known parameter that is equal to an unknown parameter, then the algorithm replaces all such unknown parameters with thisThread.

7.2 Single-Threaded Programs

If a class is declared to be default-single-threaded, then our system adds default type annotations wherever they are not explicitly specified by the programmer.

If the class is not explicitly parameterized, the system parameterizes the class with a single thisThread owner. If the type of any instance variable in the class 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 a requires clause, the system adds an empty requires clause to the method.

With these default types, single-threaded programs require almost no explicit type annotations.

7.3 Multithreaded Programs

Like in the case of single-threaded programs, if a class is declared to be default-multithreaded, then our system adds default type annotations wherever they are not explicitly specified by the programmer.

If the class is not explicitly parameterized, the system parameterizes the class with a single thisOwner owner. If the type of any instance variable in the class is not explicitly parameterized, the system augments the type with an appropriate number of this owner parameters. If a method in the class does not contain a requires clause, the system adds a requires clause that contains all the method arguments (including the implicit this argument) that are read or written by the method.

8 Extensions to the Type System

This section describes how our basic type system can be extended to handle objects with unique pointers and read-only objects, both of which can be accessed safely without synchronization.

8.1 Objects with Unique Pointers

Sometimes, objects migrate from one thread to another. For example, in a producer-consumer paradigm, one or more threads may produce data that is subsequently processed by one or more consumer threads. To enable an object to migrate from one thread to another, we use the notion of unique pointers [29, 7, 23].

If a variable (or field) x is declared to be the unique pointer to an object, then it means that there is no other variable (or field) that has a pointer to that object. If a thread has a unique pointer to an object, then the thread can access the object without acquiring any locks.

Regular variables cannot be assigned to unique variables. Unique variables can be transferred to other regular or unique variables only by using the following syntax. x and y are assumed to be unique variables in the example below.

```
y = x--;  // y = x; x = null;
m(y--);   // m(y);  y = null;
```

Guava [3] uses a similar syntax to transfer objects between variables. This syntax is inspired by the following C expression syntax. i and j are assumed to be integer variables below.

```
j = i--;  // j = i; i = i-1;
m(j--);   // m(j);  j = j-1;
```

In the above example, if the x is a method-local variable and is not subsequently used within the method, an optimizing compiler will eliminate `x=null` as dead code.

By using the `x--` construct, we are in effect shifting some checking from compile time to runtime. If x is subsequently dereferenced before being reinitialized, the system will raise a NullPointer exception at runtime.

A variable (or field) can be declared to be unique in our system if we instantiate its type by using unique as the first owner. For example, `Account<unique> a` declares that a is a unique pointer to an Account object. Like self, unique is thus a special owner in our system that can be used to instantiate classes.

Of course, not all class parameters can be instantiated with unique. The TOwner parameter in Figure 6 is an example. To be well-typed, the TStack class would have to declare that TOwner cannot be unique. We use where clauses to thus constrain class parameters. This is somewhat similar to the use of where clauses in [15, 30]. The code in Figure 13 shows the declaration of TStack with a where clause constraining TOwner.

```
class TStack<thisOwner, TOwner> where (TOwner != unique) {
   ...
}
```

Figure 13: Using a Where Clause

It is also possible to rewrite the TStack class so that TOwner can be instantiated with unique. Figure 14 shows how the code in Figure 6 can be changed to support this. Only the lines of code that are different from Figure 6 are shown here.

```
11 newNode.init(value--, head);
18 return value--;
29 this.value = v--;
33 T<TOwner> value() requires (this) {return value--;}
```

Figure 14: Changes to the TStack code in Figure 6

With the changes to TStack shown in Figure 14, we can instantiate a stack of unique objects. A producer thread can now insert unique items into the stack, while consumer threads can extract the items from the stack and process them without having to acquire locks on those items. Figure 15 shows an example.

A method-local variable containing a unique pointer can also be passed as an argument to a method provided the method declaration specifies that the argument does not escape. To enable this, types of local variables and method arguments

```
final TStack<self, unique> s = new TStack<self, unique>;

T<unique> t1 = new T<unique>;
T<unique> t2 = new T<unique>;

synchronized (s) in { s.push(t1--); }
synchronized (s) in { s.push(t2--); }
fork(s) {synchronized(s) in {T<unique> t=s.pop(); t.x=1;}}
fork(s) {synchronized(s) in {T<unique> t=s.pop(); t.x=1;}}
```

Figure 15: Using a Stack of Unique Objects

can be augmented with an optional !e. This is similar to the use of effects [28]. We refer to such types as non-escaping types.

If a variable has a non-escaping type, then it means that the reference stored in the variable will not escape to any object field or to another thread. A variable with a non-escaping type can be assigned only to other variables with non-escaping types. Similarly, it can passed as a method argument only if the type of the argument is specified to be non-escaping in the method declaration.

Figure 16 shows an example where a unique Message object is passed as an argument to a display method that declares that the Message argument will not escape.

```
class Message<thisOwner> {...};

class Util<thisOwner, MsgOwner> {
 void display(Message<MsgOwner>!e m) requires(m) {...}
}

Util<self, unique> u = new Util<self, unique>;
Message<unique>    m = new Message<unique>;
u.display(m);
```

Figure 16: Using Effects

Issues Related to the Java Memory Model

Synchronization operations in Java are used not just for mutual exclusion, but also to enforce visibility in multiprocessor machines [31]. Therefore, if Thread 1 creates or updates an object x and passes the unique reference to x to Thread 2 without using synchronization, then updates made by Thread 1 are not guaranteed to be visible to Thread 2.

But this is not a problem in our system because the only way Thread 1 can pass the unique reference to x to Thread 2 is by writing the unique reference into a shared data structure that can be subsequently read by Thread 2. But since the shared data structure can only be accessed with synchronization, the updates made by Thread 1 will be visible to Thread 2.

8.2 Read-Only Objects

A variable (or field) can be declared to be read-only if we instantiate its type using readonly as the first owner. Read-only objects can only be read but cannot be written into; hence they can be accessed without using any synchronization operations.

Figure 17 shows an example where the program first creates

and initializes a Message object that has a unique reference. The program then transfers the reference to a read-only variable. In general, this is the mechanism our system uses to support the initialization of read-only objects.

A read-only object can be passed as an argument to a method only if the method declares that the argument will not be written into by augmenting the type of the argument with !w. In the example in Figure 17, the read-only Message object rm is passed as an argument to a read method that declares that the Message argument will not escape and it will not be written into.

```
class Message<thisOwner> {...}

class Util<thisOwner, MsgOwner> {
 void init(Message<MsgOwner>!e  m) requires(m) {...}
 void read(Message<MsgOwner>!ew m) requires(m) {...}
}
Util<self, unique> u = new Util<self, unique>;

Message<unique>    m = new Message<unique>;
u.init(m);

Message<readonly> rm = m--;
fork (u, rm) { u.read(rm); }
fork (u, rm) { u.read(rm); }
```

Figure 17: A Read-Only Message Object

9 Experience

We implemented a number of Java programs in our extended language including several classes from the Java libraries. We also modified some server programs and implemented them in our system. These include 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.

We implemented the library classes in our system to be externally synchronized. This gives the users of the classes the flexibility to create different instances of the classes with different protection mechanisms. In fact, this also helped us eliminate some unnecessary synchronization operations from Sun's implementation of those classes. For example, in the PrintStream class in Sun's implementation, the print(String) method acquires the lock on the PrintStream object and then calls a method that acquires the lock on a BufferedWriter object contained within the PrintStream object. Acquiring the second lock was unnecessary and our implementation avoids this.

It is also possible to implement self-synchronized versions of these classes in our system, just like the way they are implemented in the Java libraries. The self-synchronized classes can be implemented as subtypes of the regular classes, similar to the way the SharedAccount class is implemented as a subtype of the Account class in Figure 9.

One important lesson we learned while implementing programs in our system is that it is often convenient and sometimes even necessary to have parameterized methods in addition to parameterized classes. For example, the PrintStream class has a print(Object) method. Let us say, the Object argument is owned by ObjectOwner. If we did not have parameterized methods, then the PrintStream class would have to have an ObjectOwner parameter. Not only would this be unnecessarily tedious, but it would also mean that all objects that can be printed by this PrintStream have to be owned by the same owner (or by self).

Figure 18 shows the lines of code that needed explicit type annotations for some of the programs we implemented in our system. As described in Section 7.1, our system infers the owner parameters of method-local variables. Moreover, the default requires clauses provided by the system (described in Section 7.3) were sufficient in most cases. Thus, most of the type annotations we had to write involved augmenting the types of method arguments and return values with appropriate owner parameters.

Program	Lines of Code	Lines Changed
Collection Classes		
java.util.Vector	0992	35
java.util.ArrayList	0533	18
java.util.Hashtable	1011	53
java.util.HashMap	0852	46
Other Library Classes		
java.io.PrintStream	568	14
java.io.FilterOutputStream	148	05
java.io.OutputStream	134	03
java.io.BufferedWriter	253	09
java.io.OutputStreamWriter	266	11
java.io.Writer	177	06
Multithreaded Server Programs		
http	563	26
chat	308	21
stock quote	242	12
game	087	10
phone	302	10

Figure 18: Programming Overhead

9.1 Limitations of the Type System

Our experience suggests that our type system is sufficiently expressive to accommodate the commonly used protection mechanisms, and that it can be extended to the cover all the features in the Java language. However, we did encounter the following limitations of our type system.

Runtime Casts: 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. And since we do not keep ownership information at runtime, we cannot verify this during runtime either.

In general, whenever an object is downcast to a type containing more than one owner parameter, there is no safe way to execute the downcast in our system. We provide a mechanism for escaping the type system in such cases.

It is also possible to design a compromise solution that would enable programmers to explicitly preserve some ownership information at runtime for objects that may be involved in dynamic casts. The system can then use this runtime information to safely execute some of the downcast operations. Other downcast operations that cannot be safely executed will be disallowed by the system.

Static Variables: Java has global (static) variables that are accessible to all threads. If a program accesses static variables without synchronization, our system cannot verify that this will not lead to data races. Therefore, in our system, a thread can access a static variable only when it holds the lock on the Java class that contains the static variable.

Multithreaded Scientific Programs: We looked at some scientific programs like barnes and water from the SPLASH-2 benchmark set [36]. These programs proceed through phases that are separated by barriers. Within each phase, there are unsynchronized accesses to disjoint elements of the same array by different threads. Our type system does not support these synchronization patterns. To accommodate such programs, a type system would have to provide a way for expressing temporal properties — like the fact that two consecutive phases in the program do not overlap in time.

10 Related Work

There has been much research on approaches that help programmers detect or prevent data races.

Tools like the Extended Static Checker for Java (Esc/Java) [26, 17] and Warlock [34] use programmer-supplied annotations to statically detect potential data races in a program. While these tools are useful in practice, they are not sound, in that they do not certify that a program is race-free.

Other systems developed mostly in the scientific parallel programming community [18, 25], and tools like Eraser [33], detect races dynamically. These tools have the advantage that they can check unannotated programs. However, they are also not comprehensive, in that they may fail to detect certain errors due to insufficient test coverage.

To our knowledge, Concurrent Pascal is the first race-free programming language [9]. Programs in Concurrent Pascal used synchronized monitors to prevent data races. But monitors in Concurrent Pascal were restricted in that threads could share data with monitors only by copying the data. A thread could not pass a reference to an object to a monitor.

More recently, researchers have proposed type systems for object-oriented programs that guarantee that any well-typed program is free of data races [21, 19, 20, 3]. The work on Race Free Java [21] is closest to ours. Race Free Java extends the static annotations in Esc/Java into a formal type system. It also introduces a way of parameterizing types with values that lets programmers use a single lock to guard an entire compound data structure like a linked list. Race Free Java also supports the use of thread-local objects by providing thread-local classes. Instances of thread-local classes need no synchronization.

Our work builds on this type system by letting programmers write generic code to implement a class, and create different objects of the same class that have different protection mechanisms. For example, in our system, programmers can write a generic Queue implementation, then create Queue objects that have different protection mechanisms. These different objects could include thread-local Queue objects, shared Queue objects, Queue objects contained within other enclosing data structures, Queue objects containing thread-local items, Queue objects containing shared items, and Queue objects containing unique items. In Race Free Java, one needed a different Queue implementation to support each of the above cases. Race Free Java also does not support unique pointers or read-only objects. But while our system only supports locking at the granularity of individual objects, Race Free Java allows more fine-grained locking where different fields of the same object can be guarded by different locks.

Guava [3] is another dialect of Java for preventing data races. It allows programmers to access objects without synchronization in many common cases where the absence of synchronization does not lead to data races. Guava splits the class hierarchy into three distinct sub-hierarchies. Instances of Monitor classes are self-synchronized shared objects that correspond to the roots of ownership trees in our system. Instances of Object classes are either thread-local or contained within some Monitor. These instances correspond to objects that are either owned by thisThread or by some other object in our system. Instances of Value classes are somewhat analogous to objects with unique pointers in our system. Again, the primary difference between the Guava approach and our approach is that our system lets programmers to write generic code, then create objects that have different protection mechanisms from the same generic code.

Vault [16] is a new programming language that is designed to enforce high-level protocols in low-level software. Besides verifying absence of date races, the Vault type checker can also verify other properties like absence of resource leaks and absence of dangling pointers. But the Vault type system is not specifically designed to prevent data races and it does not support some commonly used protection mechanisms. For example, it does not have a notion of thread-local objects that can be safely accessed without synchronization.

There has been a lot of work recently on compiler analysis techniques to eliminate unnecessary synchronizations [2, 35, 11, 5, 6, 32]. In our system, the natural way to implement most library classes (like a Hashtable, for example) is to require external synchronization. This has the effect of moving synchronization operations up the call chain. This in turn helps programmers structure their programs such

that locks are acquired only when necessary. Syntactic sugar can be provided to make it more convenient to acquire the lock on an object before invoking a method on it. Thus, our system provides an alternate way to reduce the number of unnecessary synchronization operations in a program without risking data races.

The concept of object ownership used in this paper is similar to the one in ownership types [13, 12], even though there it was motivated by software engineering principles and was used to restrict object aliasing.

Our way of parameterizing classes is similar to the proposals for parametric types for Java [30, 8, 1, 10], except that the parameters in our system are values and not types.

11 Conclusions

We presented a type system that guarantees that well-typed programs are free of data races. Our type system is significantly more expressive than previous such type systems. Unlike previous type systems, our type system lets programmers write generic code to implement a class, then create different objects from the same class that have different protection mechanisms. This flexibility lets programmers use a variety of protection mechanisms and acquire locks only when necessary; thus making programs more efficient without sacrificing reliability.

Our system also provides default types that reduce the burden of writing the extra type annotations. In particular, single-threaded programs require almost no programming overhead.

We implemented several multithreaded Java programs in our system. Our experience shows that our system is sufficiently expressive, and requires little programming overhead.

12 Acknowledgments

We are grateful to Robert Lee for his work on implementing a prototype type checker for our system. We would also like thank Jonathan Babb, Michael Ernst, Kyle Jamieson, Dina Katabi, Viktor Kuncak, and the anonymous referees for providing useful comments on this paper.

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, C. Chambers, E. G. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronizations from Java programs. In *Static Analysis Symposium (SAS)*, September 1999.

[3] 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.

[4] A. D. Birrel. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.

[5] B. Blanchet. Escape analysis for object-oriented languages. Application to Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[6] J. Bogda and U. Holzle. Removing unnecessary synchronization in Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalization of uniqueness and sharing. In *European Conference for Object-Oriented Programming (ECOOP)*, June 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. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[10] R. Cartwright and G. L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[11] J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

[19] C. Flanagan and M. Abadi. Object types against races. In *Conference on Concurrent Theory (CONCUR)*, August 1999.

[20] C. Flanagan and M. Abadi. Types for safe locking. In *European Symposium on Programming (ESOP)*, March 1999.

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

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

[23] J.-Y. Girard. Linear logic. In *Theoretical Computer Science*, 1987.

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

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

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

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

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

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

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

[31] W. Pugh. Fixing the Java memory model. In *ACM Java Grande Conference*, June 1999.

[32] E. Ruf. Effective synchronization removal for Java. In *Programming Language Design and Implementation (PLDI)*, June 2000.

[33] 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.

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

[35] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1999.

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture (ISCA)*, June 1995.

A The Type System

This appendix presents the type system described in Section 5. The grammar for the type system was shown in beginning of Section 5. We first define a number of predicates used in the type system informally. These predicates are based on similar predicates from [22] and [21]. 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-PerClass(P)	No method name appears more than once per class
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden. The requires clause of the overriding method must be the same or a subset of the requires clause of the methods being overridden

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

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

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
$P \vdash defn$	$defn$ is a well-formed class definition
$P; E \vdash wf$	E is a well-formed typing environment
$P; E \vdash meth$	$meth$ is a well-formed method
$P; E \vdash field$	$field$ is a well-formed field
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 <: t_2$	t_1 is a subtype of t_2
$P; E \vdash field \in cn\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits $field$
$P; E \vdash meth \in cn\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits $meth$
$P; E \vdash_{\text{final}} e : t$	e is a final expression with type t
$P; E \vdash_{\text{owner}} o$	o can be an owner
$P; E \vdash \text{RootOwner}(e) = r$	r is the root owner of the final expression e
$P; E \vdash e : t$	expression e has type t, provided we have all the necessary locks
$P; E; ls \vdash e : t$	expression e has type t
$P; E \vdash e : t_1 \mid t_2$	expression e has type either t_1 or t_2, provided we have all the necessary locks
$P; E; ls \vdash e : t_1 \mid t_2$	expression e has type either t_1 or t_2

$\boxed{\vdash P : t}$

[PROG]

$$\frac{ClassOnce(P)\ \ WFClasses(P)\ \ FieldsOnce(P)\ \ MethodsOncePerClass(P)\ \ OverridesOK(P)\ \ P = defn_{1..n}\ \ local_{1..l}\ \ e \quad P \vdash defn_i \quad P;\ local_{1..l};\ \text{thisThread} \vdash e : t}{\vdash P : t}$$

$\boxed{P \vdash defn}$

[CLASS]

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

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

[OWNER THISTHREAD]

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

[OWNER OTHERTHREAD]

$$\frac{P;\ E \vdash wf}{P;\ E \vdash_{\text{owner}} \text{otherThread}}$$

[OWNER SELF]

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

[OWNER FINAL]

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

[OWNER FORMAL]

$$\frac{P;\ E \vdash wf \quad E = E_1, \text{ owner}_{\text{formal}}\ f,\ E_2}{P;E \vdash_{\text{owner}} f}$$

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

[FINAL VAR]

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

[FINAL REF]

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

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

[ENV ∅]

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

[ENV OWNER]

$$\frac{P;\ E \vdash wf \quad f \notin Dom(E)}{P;\ E,\ \text{owner}_{\text{formal}}\ f \vdash wf}$$

[ENV X]

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

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

[TYPE INT]

$$\frac{P;\ E \vdash wf}{P;\ E \vdash \text{int}}$$

[TYPE OBJECT]

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

[TYPE SHARED CLASS]

$$\frac{P \vdash \text{class } cn\langle \text{self } f_{2..n}\rangle\ \ldots \quad P;\ E \vdash wf \quad P;E \vdash_{\text{owner}} o_{2..n}}{P;E \vdash cn\langle \text{self } o_{2..n}\rangle}$$

[TYPE THREAD-LOCAL CLASS]

$$\frac{P \vdash \text{class } cn\langle \text{thisThread } f_{2..n}\rangle\ \ldots \quad P;\ E \vdash wf \quad P;E \vdash_{\text{owner}} o_{2..n}}{P;E \vdash cn\langle \text{thisThread } o_{2..n}\rangle}$$

[TYPE C]

$$\frac{f_1 \neq \text{self} \mid \text{thisThread} \quad P \vdash \text{class } cn\langle f_{1..n}\rangle\ \ldots \quad P;E \vdash_{\text{owner}} o_{1..n}}{P;E \vdash cn\langle o_{1..n}\rangle}$$

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

[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]

$$\frac{P;E \vdash cn_1\langle o_{1..n}\rangle \quad P \vdash \text{class } cn_1\langle f_{1..n}\rangle \text{ extends } cn_2\langle f_1\ o*\rangle\ \ldots}{P;\ E \vdash cn_1\langle o_{1..n}\rangle <: cn_2\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

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

[FIELD INIT]

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

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

[FIELD DECLARED]

$$\frac{P \vdash \text{class } cn\langle f_{1..n}\rangle \ldots\ \{\ \ldots\ field\ \ldots\ \}}{P;\ E \vdash field \in cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{P;\ E \vdash field \in cn\langle f_{1..n}\rangle \quad P \vdash \text{class } cn'\langle g_{1..m}\rangle \text{ extends } cn\langle o_{1..n}\rangle \ldots}{P;\ E \vdash field[o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

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

[METHOD]

$$\frac{g_i = \text{final } arg_i \quad P;\ E,\ g_{1..n} \vdash_{\text{final}} e_i : t_i \quad P;\ E,\ g_{1..n} \vdash \text{RootOwner}(e_i) = r_i \quad P;\ E,\ g_{1..n},\ local_{1..l};\ \text{thisThread},\ r_{1..m} \vdash e : t}{P;\ E \vdash t\ mn(arg_{1..n})\ \text{requires } (e_{1..m})\{local_{1..l}\ e\}}$$

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

[METHOD DECLARED]

$$\frac{P \vdash \text{class } cn\langle f_{1..n}\rangle \ldots\ \{\ \ldots\ meth\ \ldots\ \}}{P;\ E \vdash meth \in cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{P;\ E \vdash meth \in cn\langle f_{1..n}\rangle \quad P \vdash \text{class } cn'\langle g_{1..m}\rangle \text{ extends } cn\langle o_{1..n}\rangle \ldots}{P;\ E \vdash meth[o_1/f_1]..[o_n/f_n] \in cn'\langle g_{1..m}\rangle}$$

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

[ROOTOWNER THISTHREAD]

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

[ROOTOWNER OTHERTHREAD]

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

[ROOTOWNER SELF]

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

[ROOTOWNER FINAL TRANSITIVE]

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

[ROOTOWNER FORMAL]

$$\frac{P;\ E \vdash e : cn\langle o_{1..n}\rangle \mid \text{Object}\langle o_1\rangle \quad E = E_1,\ owner_{\text{formal}}\ o_1,\ E_2}{P;\ E \vdash \text{RootOwner}(e) = \text{RO}(e)}$$

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

[EXP TYPE]

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

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

[EXP SUB]

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

[EXP NEW]

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

[EXP SEQ]

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

[EXP VAR]

$$\frac{P;\ E \vdash wf \quad E = E_1, [\text{final}]_{\text{opt}}\ t\ x,\ E_2}{P;\ E;\ ls \vdash x : t}$$

[EXP VAR INIT]

$$\frac{P;\ E;\ ls \vdash e : t}{P;\ E;\ ls \vdash [\text{final}]_{\text{opt}}\ t\ x = e}$$

[EXP VAR ASSIGN]

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

[EXP FORK]

$$\frac{P;\ E;\ ls \vdash e_i : t_i \quad g_i = \text{final } t_i\ e_i \quad P;\ g_i\ [\text{otherThread}/\text{thisThread}],\ local_{1..l};\ \text{thisThread} \vdash e : t}{P;\ E;\ ls \vdash \text{fork } (e_{1..n})\ \{local_{1..l}\ e\} : \text{int}}$$

[EXP SYNC]

$$\frac{P;\ E \vdash_{\text{final}} e_1 : t_1 \quad P;\ E;\ ls,\ e_1 \vdash e_2 : t_2}{P;\ E;\ ls \vdash \text{synchronized } e_1 \text{ in } e_2 : t_2}$$

[EXP INVOKE]

$$\frac{P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \quad P;\ E \vdash (t\ mn(t_j\ y_{j\ j\in 1..k})\ \text{requires}(e'_{1..m})\ldots) \in cn\langle f_{1..n}\rangle \quad P;\ E;\ ls \vdash e_j : t_j[e/\text{this}][o_1/f_1]..[o_n/f_n] \quad P;\ E \vdash \text{RootOwner}(e'_i[e/\text{this}][o_1/f_1]..[o_n/f_n]) = r'_i \quad r'_i \in ls}{P;\ E;\ ls \vdash e.mn(e_{1..k}) : t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP REF]

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

[EXP ASSIGN]

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