A Type System for Preventing Data Races

Chandrasekhar Boyapati CHANDRA@LCS.MIT.EDU
Martin Rinard RINARD@LCS.MIT.EDU
MIT Laboratory for Computer Science, 200 Technology Square, Cambridge MA, 02139 USA

1. Introduction

The use of multiple threads of control is quickly becoming a mainstream programming practice. But threads can significantly complicate the software development process. Multi-threaded 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 are one of the most insidious programming errors to detect, reproduce, and eliminate, many researchers have developed tools that help programmers detect or eliminate data races. These tools range from those that monitor the execution of the program to dynamically detect potential races, to static race detection systems, to more formal type systems that ensure race-free programs.

This paper presents a new static type system for writing multi-threaded programs. This type system guarantees that any well-typed program is free of data races. We focus on object-oriented programs that use lock-based synchronization. In such programs, every shared data structure is protected by a lock that must be acquired before the data structure is accessed. Since at most one thread can hold a lock at any time, the correct use of lock-based synchronization guarantees that the program is race-free.

Like previous such type systems (Flanagan & Freund, June 2000; Bacon et al., October 2000), we provide a way for programmers to augment their programs with type declarations that make the locking discipline explicit. These declarations associate locks with the data they protect. The type checker uses these declarations to verify that a thread holds the appropriate locks when it accesses shared data.

But unlike previous systems, we let programmers write generic code to implement a class, and create different objects of the class that follow different locking disciplines. We do this by introducing a way of parameterizing classes that lets programmers defer decisions about which locks protect which objects from the time when a class is defined to the time when objects of that class are created.

Without this flexibility, in previous systems, programmers sometimes had to acquire redundant locks just to ensure that their programs were well-typed. Or, they had to unnecessarily duplicate code by implementing multiple versions of the same classes; these versions contained the exact same code except for synchronization operations.

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

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

2. The Type System

The key to our type system is the concept of object ownership. This resembles the notion of ownership types described in (Clarke et al., October 1998), 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.

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 thisThread without explicitly acquiring any locks.

The language we use is an extension to Concurrent Java, which is a variant of Java and has format semantics. 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. The difference is that

```
// thisOwner owns the Account object
class Account<thisOwner> {
  int balance = 0;
  int deposit(int x) requires (this) {
    this.balance = this.balance + x;
  }
}
// a1 is owned by this thread, so it is thread-local
Account<thisThread> a1 = new Account<thisThread>;
a1.deposit(10);

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

Figure 1. An Account Program With no Data Races

the parameters are values and not other types.

Figure 1 shows an example where 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, a1 is owned by thisThread and hence 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.

Methods can now require callers to hold locks on some objects using the requires clause. In the example, the deposit method on Account requires that every thread hold the lock on the (root owner of the) Account object before the thread calls 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.

It is important to note that 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.

More details on the type system, including the formal rules for type checking and how the system automatically infers most of the extra type declarations, will appear in (Boyapati & Rinard, October 2001).

3. 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. Figure 2 shows the lines of code that needed explicit type annotations for some of the programs we implemented in our system.

Program	**Lines of Code**	**Lines Changed**
Collection Classes		
java.util.Vector	0992	35
java.util.Hashtable	1011	53
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
Multi-threaded Server Programs		
http	563	26
chat	308	21
stock quote	242	12
game	087	10
phone	302	10

Figure 2. Programming Overhead

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 locking disciplines. 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.

References

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

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

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

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