Safe Concurrent Programming in Java

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

1. Introduction

Multithreaded programming is difficult and error prone. Multithreaded programs typically synchronize operations on shared mutable data to ensure that the operations execute atomically. Failure to correctly synchronize such operations can lead to *data races* or *deadlocks*. A data race occurs when two threads concurrently access the same data without synchronization, and at least one of the accesses is a write. A deadlock occurs when there is a cycle of the form: $\forall i \in \{0..n-1\}$, Thread$_i$ holds Lock$_i$ and Thread$_i$ is waiting for Lock$_{(i+1) \bmod n}$.

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. In recent previous work, we presented a static type system to prevent data races. In this paper, we extend the race-free type system to prevent both data races and deadlocks. The basic idea behind our system is as follows. When programmers write multithreaded programs, they already have a locking discipline in mind. Our system allows programmers to specify this locking discipline in their programs. The resulting specifications take the form of type declarations.

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

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

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. We have a prototype implementation of our type system that handles all the features of the Java language. We also modified several multithreaded Java programs and implemented them in our system. These programs exhibit a variety of sharing patterns. Our experience shows that our system is sufficiently expressive and requires little programming overhead.

2. The Type System

This section introduces our type system with two examples. Figure 1 presents an example program that has an `Account` class and a `CombinedAccount` class. To prevent data races, programmers associate every object 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.

```
class Account {
  int balance = 0;

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

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

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

Figure 1. Combined Account Example

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

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

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

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

Figure 2. Tree Example

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

Methods in our system may contain accesses clauses to specify assumptions that hold at method boundaries. The methods of the Account class each have an accesses clause that specifies that the methods access the this Account object without synchronization. To prevent data races, our type checker requires that the callers of an Account method must hold the lock that protects the corresponding Account object before the callers can invoke the Account method.

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

Figure 2 presents part of a BalancedTree implemented in our type system. A BalancedTree is a tree of Nodes. Every Node object is declared to be protected by its own lock. To prevent data races, the type checker statically ensures that a thread holds the lock on a Node object before accessing the Node object. The Node class is parameterized by the formal lock level v. The Node class has two Node fields—left and a right. The Nodes left and right also belong to the same lock level v.

Our system also allows programmers to use recursive tree-based data structures to further order the locks that belong to the same lock level. In the example, the key word tree indicates that the Nodes left and right are ordered 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 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. More details about the type system will appear in (Boyapati et al., November 2002).

References

Boyapati, C., Lee, R., & Rinard, M. (November 2002). Ownership types for safe programming: Preventing data races and deadlocks. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).*