Ownership Types for Safe Programming: Preventing Data Races and Deadlocks

Chandrasekhar Boyapati
Robert Lee
Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
{chandra, rhlee, rinard}@lcs.mit.edu
Data Races in Multithreaded Programs

- Two threads access same data
- At least one access is a write
- No synchronization to separate accesses
Avoiding Data Races

Thread 1:

\[ x = x + 1; \]

Thread 2:

\[ x = x + 2; \]
Avoiding Data Races

- Associate locks with shared mutable data
- Acquire lock before data access
- Release lock after data access

Thread 1:
lock(l);
x = x + 1;
unlock(l);

Thread 2:
lock(l);
x = x + 2;
unlock(l);
Deadlocks in Multithreaded Programs

- **Cycle of the form**
  - Thread 1 holds Lock 1, waits for Lock 2
  - Thread 2 holds Lock 2, waits for Lock 3 ...
  - Thread n holds Lock n, waits for Lock 1
Avoiding Deadlocks

Thread 1

Lock 1

Thread n

Lock n

Thread 2

Lock 2

Thread 2

Lock 3

...
Avoiding Deadlocks

- Associate a partial order among locks
- Acquire locks in order
Problem With Current Practice

- Locking discipline is not enforced

- Inadvertent programming errors
  - Can cause data races and deadlocks

- Consequences can be severe
  - Non-deterministic, timing dependent bugs
  - Difficult to detect, reproduce, eliminate
Our Solution

- Static type system
  - Prevents both data races and deadlocks
Our Solution

- **Static type system**
  - Prevents both data races and deadlocks

- **Programmers specify**
  - How each object is protected from races
  - Partial order among locks

- **Type checker statically verifies**
  - Objects are used only as specified
  - Locks are acquired in order
Preventing Data Races

- Programmers specify for every object
  - Lock protecting the object, or
  - That the object needs no locks because
    - Object is immutable
    - Object is thread-local
    - Object has a unique pointer
Preventing Deadlocks

- Programmers specify lock ordering using
  - Static lock levels
  - Recursive data structures
    - Mutable trees
    - Monotonic DAGs
  - Runtime ordering

- Type checker statically verifies
  - Locks are acquired in descending order
  - Specified order is a partial order
Lock Level Based Partial Orders

- Lock levels are partially ordered
- Locks belong to lock levels
- Threads must acquire locks in descending order of lock levels
class CombinedAccount {

    final Account savingsAccount = new Account();
    final Account checkingAccount = new Account();

    int balance() {
        synchronized (savingsAccount) {
            synchronized (checkingAccount) {
                return savingsAccount.balance + checkingAccount.balance;
            }
        }
    }
}
class CombinedAccount {

  LockLevel savingsLevel;
  LockLevel checkingLevel < savingsLevel;

  final Account<self : savingsLevel> savingsAccount = new Account();
  final Account<self : checkingLevel> checkingAccount = new Account();

  int balance() locks (savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        return savingsAccount.balance + checkingAccount.balance;
      }
    }
  }
}
class CombinedAccount {

    LockLevel savingsLevel;
    LockLevel checkingLevel < savingsLevel;

    final Account<self : savingsLevel> savingsAccount = new Account();
    final Account<self : checkingLevel> checkingAccount = new Account();

    int balance() locks (savingsLevel) {
        synchronized (savingsAccount) {
            synchronized (checkingAccount) {
                return savingsAccount.balance + checkingAccount.balance;
            }
        }
    }
}
class CombinedAccount {

  LockLevel savingsLevel;
  LockLevel checkingLevel < savingsLevel;

  final Account<self : savingsLevel> savingsAccount = new Account();
  final Account<self : checkingLevel> checkingAccount = new Account();

  int balance() locks (savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        return savingsAccount.balance + checkingAccount.balance;
      }
    }
  }
}
Lock Level Based Partial Orders

```java
class CombinedAccount {
    LockLevel savingsLevel;
    LockLevel checkingLevel < savingsLevel;

    final Account<self : savingsLevel> savingsAccount = new Account();
    final Account<self : checkingLevel> checkingAccount = new Account();

    int balance() locks (savingsLevel) {
        synchronized (savingsAccount) {
            synchronized (checkingAccount) {
                return savingsAccount.balance + checkingAccount.balance;
            }
        }
    }
}
```
class CombinedAccount {

  LockLevel savingsLevel;
  LockLevel checkingLevel < savingsLevel;

  final Account<self : savingsLevel> savingsAccount = new Account();
  final Account<self : checkingLevel> checkingAccount = new Account();

  int balance() locks (savingsLevel) {
    synchronized (savingsAccount) {
      synchronized (checkingAccount) {
        return savingsAccount.balance + checkingAccount.balance;
      }
    }
  }
}
class CombinedAccount {

    LockLevel savingsLevel;
    LockLevel checkingLevel < savingsLevel;

    final Account<self : savingsLevel> savingsAccount = new Account();
    final Account<self : checkingLevel> checkingAccount = new Account();

    int balance() locks (savingsLevel) {
        synchronized (savingsAccount) {
            synchronized (checkingAccount) {
                return savingsAccount.balance + checkingAccount.balance;
            }
        }
    }
}
Types Impose No Dynamic Overhead

Java

+ Extra types

Type checker

Translator (Removes extra types)

Java

Compiler

bytcodes

JVM
Lock Level Based Partial Orders

- Bounded number of lock levels
- Unbounded number of locks

- Lock levels support programs where the maximum number of locks simultaneously held by a thread is bounded

- We use other mechanisms for other cases
Type System

- Preventing data races
- Preventing deadlocks using
  - Static lock levels
  - Recursive data structures
    - Mutable trees
    - Monotonic DAGs
- Runtime ordering
Tree Based Partial Orders

- Locks in a level can be tree-ordered
- Using data structures with tree backbones
  - Doubly linked lists
  - Trees with parent/sibling pointers
  - Threaded trees...
class Node {
    Node left;
    Node right;

    synchronized void rotateRight() {
        Node x = this.right;  synchronized (x) {
            Node v = x.left;        synchronized (v) {
                Node w = v.right;        synchronized (w) {
                    v.right = null;
                    x.left = w;
                    this.right = v;
                    v.right = x;
                }
            }
        }
    }
}
class Node\{self : l}\ {  
  tree Node\{self : l\} left;
  tree Node\{self : l\} right;

  synchronized void rotateRight() \{locks (this)\ {  
    Node x = this.right;  synchronized (x) {  
      Node v = x.left;  synchronized (v) {  
        Node w = v.right;  
        v.right = null;
        x.left = w;
        this.right = v;
        v.right = x;
      }  
    }  
  }  
}
class Node {
    tree Node left;
    tree Node right;
}

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

nodes are locked in tree order

Tree Based Partial Orders
Checking Tree Mutations

- A tree edge may be deleted
- A tree edge from x to y may be added iff
  - y is a Root
  - x is not in Tree(y)
- For onstage nodes x & y, analysis tracks
  - If y is a Root
  - If x is not in Tree(y)
  - If x has a tree edge to y
- Lightweight shape analysis
Checking Tree Mutations

class Node
{
    tree Node left;
    tree Node right;

    synchronized void rotateRight()
    {
        Node x = this.right;  synchronized (x) {
            Node v = x.left;        synchronized (v) {
                Node w = v.right;        synchronized (v) {
                    v.right     = null;
                    x.left       = w;
                    this.right = v;
                    v.right     = x;
                } } }
    }
}
Checking Tree Mutations

class Node {  
  tree Node left;
  tree Node right;

  synchronized void rotateRight() locks (this) {  
    Node x = this.right;  synchronized (x) {  
      Node v = x.left;  synchronized (v) {  
        Node w = v.right;  
        v.right = null;
        x.left = w;
        this.right = v;
        v.right = x;
      }  
    }  
  }
}
class Node{
    tree Node left;
    tree Node right;

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

Checking Tree Mutations

x = this.right
v = x.left

w is Root

v not in Tree(w)
x not in Tree(w)
this not in Tree(w)
class Node { 
    tree Node left; 
    tree Node right; 
}

synchronized void rotateRight() locks (this) { 
    Node x = this.right;  synchronized (x) { 
        Node v = x.left;        synchronized (v) { 
            Node w = v.right;        synchronized (v) { 
                v.right = null; 
                x.left = w; 
                this.right = v; 
                v.right = x; 
            } 
            }} 
    } 
}
class Node{
    tree Node left;
    tree Node right;

    synchronized void rotateRight() locks (this) {
        Node x = this.right;  synchronized (x) {
            Node v = x.left;        synchronized (v) {
                Node w = v.right;
                v.right = null;
                x.left = w;
                this.right = v;
                v.right = x;
            }
        }
    }
}
class Node{
    tree Node self : l} left;
    tree Node self : l} right;

    synchronized void rotateRight() locks (this) {
        Node x = this.right;  synchronized (x) {
            Node v = x.left;        synchronized (v) {
                Node w = v.right;
                v.right = null;
                x.left = w;
                this.right = v;
                v.right = x;
            }
        }
    }}}}
Type System

- Preventing data races
- Preventing deadlocks using
  - Static lock levels
  - Recursive data structures
    - Mutable trees
    - Monotonic DAGs
- Runtime ordering
DAG Based Partial Orders

- Locks in a level can be DAG-ordered
- DAGs cannot be arbitrarily modified
- DAGs can be built bottom-up by
  - Allocating a new node
  - Initializing its DAG fields

```class Node(self : l) {
  dag Node(self : l) left;
  dag Node(self : l) right;
  ...
}
```
Type System

- Preventing data races
- Preventing deadlocks using
  - Static lock levels
  - Recursive data structures
    -Mutable trees
    - Monotonic DAGs
- Runtime ordering
class Account {
    int balance = 0;
    void deposit(int x)  { balance += x; }
    void withdraw(int x) { balance -= x; }
}

void transfer(Account a1, Account a2, int x) {
    synchronized (a1, a2) in  { a1.withdraw(x); a2.deposit(x); }
}
class Account implements Dynamic {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
    void withdraw(int x) requires (this) { balance -= x; }
}

void transfer(Account(self : v) a1, Account(self : v) a2, int x) locks(v) {
    synchronized (a1, a2) in { a1.withdraw(x); a2.deposit(x); }
}
class Account implements Dynamic {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
    void withdraw(int x) requires (this) { balance -= x; }
}

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

Account objects are dynamically ordered
class Account implements Dynamic {
    int balance = 0;
    void deposit(int x)  requires (this)  { balance += x; }
    void withdraw(int x) requires (this)  { balance -= x; }
}

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

- Type inference and default types significantly reduce programming overhead
- Single threaded programs need no annotations
- Our approach supports separate compilation
Experience
## Multithreaded Server Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of code</th>
<th>Lines changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>523</td>
<td>15</td>
</tr>
<tr>
<td>http server</td>
<td>563</td>
<td>26</td>
</tr>
<tr>
<td>chat server</td>
<td>308</td>
<td>22</td>
</tr>
<tr>
<td>stock quote server</td>
<td>242</td>
<td>12</td>
</tr>
<tr>
<td>game server</td>
<td>87</td>
<td>11</td>
</tr>
<tr>
<td>phone (database) server</td>
<td>302</td>
<td>10</td>
</tr>
</tbody>
</table>
Java Libraries

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines of code</th>
<th>Lines changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.Hashtable</td>
<td>1011</td>
<td>53</td>
</tr>
<tr>
<td>java.util.HashMap</td>
<td>852</td>
<td>46</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>992</td>
<td>35</td>
</tr>
<tr>
<td>java.util.ArrayList</td>
<td>533</td>
<td>18</td>
</tr>
<tr>
<td>Program</td>
<td>Lines of code</td>
<td>Lines changed</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>java.io.PrintStream</td>
<td>568</td>
<td>14</td>
</tr>
<tr>
<td>java.io.FilterOutputStream</td>
<td>148</td>
<td>5</td>
</tr>
<tr>
<td>java.io.OutputStream</td>
<td>134</td>
<td>3</td>
</tr>
<tr>
<td>java.io.BufferedWriter</td>
<td>253</td>
<td>9</td>
</tr>
<tr>
<td>java.io.OutputStreamWriter</td>
<td>266</td>
<td>11</td>
</tr>
<tr>
<td>java.io.Writer</td>
<td>177</td>
<td>6</td>
</tr>
</tbody>
</table>
Related Work
Related Work

- **Static tools**
  - Korty (USENIX ’89)
  - Sterling (USENIX ’93)
  - Detlefs, Leino, Nelson, Saxe (SRC ’98)
  - Engler, Chen, Hallem, Chou, Chelf (SOSP ’01)

- **Dynamic tools**
  - Steele (POPL ’90)
  - Dinning, Schonberg (PPoPP ’90)
  - Savage, Burrows, Nelson, Sobalvarro, Anderson (SOSP ’97)
  - Praun, Gross (OOPSLA ’01)
  - Choi, Lee, Loginov, O’Callahan, Sarkar, Sridharan (PLDI ’02)
Related Work

- Type systems
  - Flanagan, Freund (PLDI ’00)
  - Bacon, Strom, Tarafdar (OOPSLA ’00)
Related Work

- **Ownership types**
  - Clarke, Potter, Noble (OOPSLA ’98), (ECOOP ’01)
  - Clarke, Drossopoulou (OOPSLA ’02)
  - Aldrich, Kostadinov, Chambers (OOPSLA ’02)
  - Boyapati, Rinard (OOPSLA ’01)
  - Boyapati, Lee, Rinard (OOPSLA ’02)
  - Boyapati, Liskov, Shrira (MIT ’02)
  - Boyapati, Salcianu, Beebee, Rinard (MIT ’02)
Ownership Types

- We have used ownership types for
  - Object encapsulation
  - Constraining heap aliasing
  - Modular effects clauses with subtyping
  - Preventing data races and deadlocks
  - Safe lazy upgrades in OODBs
  - Safe region-based memory management

- Ownership types can serve as a foundation for future OO languages
Conclusions

- Data races and deadlocks make multithreaded programming difficult
- We presented a static type system that prevents data races and deadlocks
- Our type system is expressive
- Programs can be efficient and reliable
Ownership Types for Safe Programming: Preventing Data Races and Deadlocks

Chandrasekhar Boyapati
Robert Lee
Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
{chandra, rhlee, rinard}@lcs.mit.edu