Ownership Types for Safe Programming

Chandrasekhar Boyapati

Laboratory for Computer Science
Massachusetts Institute of Technology
Motivation

Making software reliable

- Is important
  - Role in civil infrastructure
  - Effect on economy

- Is challenging because of complexity
This Talk

Type system to increase software reliability

- Statically prevents many classes of errors
  - Prevents data races and deadlocks
  - Prevents representation exposure
  - Enables region-based memory management
  - Enables upgrades in persistent object stores
- Checking is fast and scalable
- Requires little programming overhead
- Promising way for increasing reliability
Outline

- Preventing data races
- Preventing deadlocks
- Type inference
- Experience
- Preventing other errors
Preventing Data Races
Data Races in Multithreaded Programs

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

- Some correct programs contain data races
- But most races are programming errors
  - Code intended to execute atomically
  - Synchronization omitted by mistake
- Consequences can be severe
  - Nondeterministic timing-dependent bugs
  - Difficult to detect, reproduce, eliminate
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);
Avoiding Data Races

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

Thread 2:
lock(l);
\[ x = x + 2; \]
unlock(l);

Problem: Locking is not enforced!
Inadvertent programming errors...
Our Solution

- Type system for object-oriented languages
- Statically prevents data races
Our Solution

- Type system for object-oriented languages
- Statically prevents data races

Programmers specify
- How each object is protected from races
- In types of variables pointing to objects

Type checker statically verifies
- Objects are used only as specified
Protection Mechanism of an Object

- Specifies the lock protecting the object, or

- Specifies object needs no locks because
  - Object is immutable
  - Object is thread-local
  - Object has a unique pointer
Protection Mechanism of an Object

- Specifies the lock protecting the object, or

- Specifies object needs no locks because
  - Object is immutable
  - Object is thread-local
  - Object has a unique pointer
class Account {
    int balance = 0;
    void deposit(int x) { balance += x; }
}

Account a1 = new Account();
fork { synchronized (a1) { a1.deposit(10); } };
fork { synchronized (a1) { a1.deposit(10); } };

Account a2 = new Account();
a2.deposit(10);
class Account {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
}

Account(self) a1 = new Account();
fork  { synchronized (a1) { a1.deposit(10); } };
fork  { synchronized (a1) { a1.deposit(10); } };

Account(thisThread) a2 = new Account();
a2.deposit(10);
Preventing Data Races

class Account {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
}

Account(self) a1 = new Account();
fork { synchronized (a1) { a1.deposit(10); } };  
fork { synchronized (a1) { a1.deposit(10); } }

Account(thisThread) a2 = new Account();  
a2.deposit(10);
class Account {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
}

Account(self) a1 = new Account();
fork { synchronized (a1) { a1.deposit(10); } };  
fork { synchronized (a1) { a1.deposit(10); } };  

Account(thisThread) a2 = new Account();
a2.deposit(10);
class Account {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
}

Account(self) a1 = new Account();
fork { synchronized (a1) { a1.deposit(10); } };
fork { synchronized (a1) { a1.deposit(10); } };

Account(thisThread) a2 = new Account();
a2.deposit(10);
Types Impose No Dynamic Overhead

Java → Type checker → Translator (Removes extra types) → Java

+ Extra types

Compiler

bytcecodes

JVM
class Account {
    int balance = 0;
    void deposit(int x) requires (this) { balance += x; }
}

Account(self) a1 = new Account();
fork { synchronized (a1) { a1.deposit(10); } }; 
fork { synchronized (a1) { a1.deposit(10); } }; 

Account(thisThread) a2 = new Account();
a2.deposit(10);
class Account {
    int balance = 0;
    void deposit(int x) { balance += x; }
}

Account a1 = new Account();
fork { synchronized (a1) { a1.deposit(10); } };
fork { synchronized (a1) { a1.deposit(10); } };;

Account a2 = new Account();
a2.deposit(10);
Object Ownership
Object Ownership

- Every object is owned by
  - Itself, or
  - Another object, or
  - Special per-thread owner called thisThread

- Ownership relation forms a forest of trees
Object Ownership

- Objects with a thisThread as their root owner
  - Are local to the corresponding thread

- Objects with an object as their root owner
  - Are potentially shared between threads
Object Ownership

- Every object is protected by its root owner
- For race-free access to an object
  - A thread must lock its root owner
- A thread implicitly holds lock on its thisThread

```
Thread1 objects

Thread2 objects

Potentially shared objects
```
class TStack {
    TNode head;

    void push(T value) {...}
    T pop() {...}
}

class TNode {
    TNode next;
    T value;
    ...
}

class T {...}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}

class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}

Classes are parameterized with owners
First owner owns the “this” object
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}

All TNodes have the same owner
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}

TStack<thisThread, thisThread> s1;
TStack<thisThread, self> s2;
TStack<self, self> s3;

s1 is a thread-local stack with thread-local elements
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}
TStack<thisThread, thisThread> s1;
TStack<thisThread, self> s2;
TStack<self, self> s3;

s2 is a thread-local stack with shared elements
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}
TStack<thisThread, thisThread> s1;
TStack<thisThread, self> s2;
→ TStack<self, self> s3;

* s3 is a shared stack with shared elements
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() requires (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<nodeOwner, TOwner> {
    T<TOwner> value() requires (this) {...}
    TNode<nodeOwner, TOwner> next() requires (this) {...}
    ...
}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() requires (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<nodeOwner, TOwner> {
    T<TOwner> value() requires (this) {…}
    TNode<nodeOwner, TOwner> next() requires (this) {…}
    ...
}
class TStack<stackOwner, TOwner> { 
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() requires (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<nodeOwner, TOwner> {
    T<TOwner> value() requires (this) {...}
    TNode<nodeOwner, TOwner> next() requires (this) {...}
    ...
}
class TStack<stackOwner, TOwner> {  
    TNode<this, TOwner> head;  
    ...
    T<TOwner> pop() requires (this) {  
        if (head == null) return null;  
        T<TOwner> value = head.value();  
        head = head.next();  
        return value;  
    }  
}  
}
class TNode<nodeOwner, TOwner> {  
    T<TOwner> value() requires (this) {…}  
    TNode<nodeOwner, TOwner> next() requires (this) {…}  
    …  
}  

Locks held  
thisThread, RootOwner(this)  

Locks required  
RootOwner(this)
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() requires (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<nodeOwner, TOwner> {
    T<TOwner> value() requires (this) {
    }
    ...
    TNode<nodeOwner, TOwner> next() requires (this) {
    }
}
class TStack<stackOwner, TOwner> {  
    TNode<this, TOwner> head;  
    ...  
    T<TOwner> pop() requires (this) {  
        if (head == null) return null;  
        T<TOwner> value = head.value();  
        head = head.next();  
        return value;  
    }  
}  

class TNode<nodeOwner, TOwner> {  
    T<TOwner> value() requires (this) {...}  
    TNode<nodeOwner, TOwner> next() requires (this) {...}  
    ...  
}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() requires (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<nodeOwner, TOwner> {
    T<TOwner> value() requires (this) {...}
    TNode<nodeOwner, TOwner> next() requires (this) {...}
    ...
}
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head;
  ...
  T<TOwner> pop() requires (this) {
    if (head == null) return null;
    T<TOwner> value = head.value();
    head = head.next();
    return value;
  }
}

class TNode<nodeOwner, TOwner> {
  T<TOwner> value() requires (this) {...}
  TNode<nodeOwner, TOwner> next() requires (this) {...}
  ...
}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
    T<TOwner> pop() requires (this) {
        if (head == null) return null;
        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<nodeOwner, TOwner> {
    T<TOwner> value() requires (this) { ... }
    TNode<nodeOwner, TOwner> next() requires (this) { ... }
    ...
}
Preventing Data Races

- Data races make programming difficult
- Our type system prevents data races

- Programmers specify
  - How each object is protected from races

- Type checker statically verifies
  - Objects are used only as specified
Other Benefits of Race-free Types
Other Benefits of Race-free Types

- Data races expose the effects of
  - Weak memory consistency models
  - Standard compiler optimizations
Initially:

\[ x = 0; \]
\[ y = 1; \]

Thread 1:
\[ y = 0; \]
\[ x = 1; \]

Thread 2:
\[ z = x + y; \]

What is the value of \( z \)?
What is the value of z?

**Initially:**

- x = 0;
- y = 1;

**Possible Interleavings**

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 0;</td>
<td>z = x + y;</td>
</tr>
<tr>
<td>x = 1;</td>
<td>x = 1;</td>
</tr>
</tbody>
</table>

**z = 1**

<table>
<thead>
<tr>
<th>z = x + y;</th>
<th>y = 0;</th>
<th>y = 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>y = 0;</td>
<td>z = x + y;</td>
<td>x = 1;</td>
</tr>
<tr>
<td>x = 1;</td>
<td>z = 0</td>
<td></td>
</tr>
</tbody>
</table>

**z = 1**
Initially:

- $x = 0$
- $y = 1$

**Possible Interleavings**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Above instruction reordering legal in single-threaded programs</th>
<th>Violates sequential consistency in multithreaded programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y = 0$</td>
<td>$z = x + y$</td>
<td>$y = 0$</td>
<td>$y = 0$</td>
</tr>
<tr>
<td>$x = 1$</td>
<td></td>
<td>$x = 0$</td>
<td>$z = x + y$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$x = 1$</td>
<td>$x = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$z = 1$</td>
</tr>
</tbody>
</table>

What is the value of $z$?
Weak Memory Consistency Models

- Are complicated in presence of data races

- Original Java memory model was
  - Ambiguous and buggy

- Formal semantics still under development
  - Manson, Pugh (Java Grande/ISCOPE ’01)
  - Maessen, Arvind, Shen (OOPSLA ’00)
Other Benefits of Race-free Types

- Data races expose effects of
  - Weak memory consistency models
  - Standard compiler optimizations
- Races complicate program analysis
- Races complicate human understanding

- Race-free languages
  - Eliminate these issues
  - Make multithreaded programming tractable
Outline

- Preventing data races
- Preventing deadlocks
- Type inference
- Experience
- Preventing other errors
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
Avoiding Deadlocks

Problem: Lock ordering is not enforced!
Inadvertent programming errors...
Our Solution

- Static type system that prevents deadlocks

- Programmers specify
  - Partial order among locks

- Type checker statically verifies
  - Locks are acquired in descending order
  - Specified order is a partial order
Preventing Deadlocks

- Programmers specify lock ordering using
  - Locks levels
  - Recursive data structures
    - Tree-based data structures
    - DAG-based data structures
  - Runtime ordering
Lock Level Based Partial Orders

- Locks belong to lock levels
- Lock levels are partially ordered
- **Threads must acquire locks in order**
class CombinedAccount {
    final Account savingsAccount = new Account();
    final Account checkingAccount = new Account();

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

Lock Level Based Partial Orders
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;
      }
    }
  }
}
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

- 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
Preventing Deadlocks

- Programmers specify lock ordering using
  - Locks levels
  - Recursive data structures
    - Tree-based data structures
    - DAG-based data structures
  - 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 or 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;
                v.right = null;
                x.left = w;
                this.right = v;
                v.right = x;
            }
        }
    }
}

Tree Based Partial Orders
Tree Based Partial Orders

```java
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 must be locked in tree order
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;  
                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;
        }
    }
}

flow sensitive analysis checks that tree order is preserved
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
class Node\{self : l\} \{
    tree Node\{self : l\} left;
    tree Node\{self : l\} right;

    synchronized void rotateRight() \{ \hspace{1cm} \text{locks (this)} \{ \hspace{1cm} \text{Node x = this.right; \hspace{1cm} synchronized (x) \{ \hspace{1cm} \text{Node v = x.left; \hspace{1cm} synchronized (v) \{ \hspace{1cm} \text{Node w = v.right; \hspace{1cm} v.right = null; \hspace{1cm} x.left = w; \hspace{1cm} this.right = v; \hspace{1cm} 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() {
  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<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;        synchronized (v) {
                    Node w = v.right;        synchronized (v) {
                        x.right = v;
                        v.right = x;
                    }
                    x.left = w;
                    v.left = w;
                    this.right = v;
                    v.right = x;
                }
                v.right = x;
                this.right = v;
            }
            x.right = v;
            v.right = x;
        }
        x = this.right
        w = x.left
    }
}

Checking Tree Mutations

x  = this.right
w  = x.left
v is Root
x  not in Tree(v)
w  not in Tree(v)
this not in Tree(v)
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\{self : I\} {
    tree Node\{self : I\} left;
    tree Node\{self : I\} 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

Node w = v.right;
Node v = x.left;
Node x = v.right;
Node w = x.left;
Node x = v.right;
Preventing Deadlocks

- Programmers specify lock ordering using:
  - Locks levels
  - Recursive data structures
    - Tree-based data structures
    - DAG-based data structures
  - Runtime ordering
DAG Based Partial Orders

```python
class Node{
    dag Node left;
    dag Node right;
    ...
}
```

- **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
Preventing Deadlocks

- Programmers specify lock ordering using
  - Locks levels
  - Recursive data structures
    - Tree-based data structures
    - DAG-based data structures
  - 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); }
}
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); }
}
Preventing Deadlocks

- Static type system that prevents deadlocks

- Programmers specify
  - Partial order among locks

- Type checker statically verifies
  - Locks are acquired in descending order
  - Specified order is a partial order
Reducing Programming Overhead
Inferring Owners of Local Variables

class A<oa1, oa2> {...}
class B<ob1, ob2, ob3> extends A<ob1, ob3> {...}

class C {
    void m(B<this, oc1, thisThread> b) {
        A a1;
        B b1;
        b1 = b;
        a1 = b1;
    }
}
Inferring Owners of Local Variables

class A<oa1, oa2> {...}
class B<ob1, ob2, ob3> extends A<ob1, ob3> {...}

class C {
    void m(B>this, oc1, thisThread> b) {
        A<x1, x2> a1;
        B<x3, x4, x5> b1;
        b1 = b;
        a1 = b1;
    }
}

Augment unknown types with owners
Inferring Owners of Local Variables

class A<oa1, oa2> {...}
class B<ob1, ob2, ob3> extends A<ob1, ob3> {...}

class C {
  void m(B<this, oc1, thisThread> b) {
    A<x1, x2> a1;
    B<x3, x4, x5> b1;
    b1 = b;
    a1 = b1;
  }
}

Gather constraints
  x3 = this
  x4 = oc1
  x5 = thisThread
Inferring Owners of Local Variables

class A<oa1, oa2> {...}
class B<ob1, ob2, ob3> extends A<ob1, ob3> {...}

class C {
    void m(B<this, oc1, thisThread> b) {
        A<x1, x2> a1;
        B<x3, x4, x5> b1;
        b1 = b;
        a1 = b1;
    }
}

Gather constraints

x3 = this
x4 = oc1
x5 = thisThread
x1 = x3
x2 = x5
Inferring Owners of Local Variables

class A<oa1, oa2> {...}
class B<ob1, ob2, ob3> extends A<ob1, ob3> {...}

class C {
    void m(B<this, oc1, thisThread> b) {
        A<this, thisThread> a1;
        B>this, oc1, thisThread> b1;
        b1 = b;
        a1 = b1;
    }
}

Solve constraints

x3 = this
x4 = oc1
x5 = thisThread
x1 = x3
x2 = x5
Inferring Owners of Local Variables

class A\{oa1, oa2\} {...}
class B\{ob1, ob2, ob3\} extends A\{ob1, ob3\} {...}

class C {
    void m(B\{this, oc1, thisThread\} b) {
        A\{this, thisThread\} a1;
        B\{this, oc1, thisThread\} b1;
        b1 = b;
        a1 = b1;
    }
}

Solve constraints

- Only equality constraints between owners
- Takes almost linear time to solve

\[
x3 = this \\
x4 = oc1 \\
x5 = thisThread \\
x1 = x3 \\
x2 = x5
\]
Reducing Programming Overhead

- Type inference for method local variables
- Default types for method signatures & fields
- User defined defaults as well
- Significantly reduces programming overhead
- Approach supports separate compilation
Experience
## Multithreaded Server Programs

<table>
<thead>
<tr>
<th>Program</th>
<th># Lines of code</th>
<th># Lines annotated</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMTP Server (Apache)</td>
<td>2105</td>
<td>46</td>
</tr>
<tr>
<td>POP3 Mail Server (Apache)</td>
<td>1364</td>
<td>31</td>
</tr>
<tr>
<td>Discrete Event Simulator (ETH Zurich)</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>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 annotated</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>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.BufferedWriter</td>
<td>253</td>
<td>9</td>
</tr>
<tr>
<td>java.io.OutputStreamWriter</td>
<td>266</td>
<td>11</td>
</tr>
</tbody>
</table>
Java Libraries

- Java has two classes for resizable arrays
  - `java.util.Vector`
    - Self synchronized, do not create races
    - Always incur synchronization overhead
  - `java.util.ArrayList`
    - No unnecessary synchronization overhead
    - Could be used unsafely to create races

- We provide generic resizable arrays
  - Safe, but no unnecessary overhead

- Programs can be both reliable and efficient
Ownership Types

- Prevent data races and deadlocks
  - Boyapati, Rinard (OOPSLA ’01)
  - Boyapati, Lee, Rinard (OOPSLA ’02)

- Prevent representation exposure
  - Boyapati, Liskov, Shrira (POPL ’03)

- Enable safe region-based memory management
  - Boyapati, Salcianu, Beebee, Rinard (PLDI ’03)

- Enable safe upgrades in persistent object stores
  - Boyapati, Liskov, Shrira, Moh, Richman (OOPSLA ’03)
Preventing Representation Exposure

- Goal is local reasoning about correctness
  - Prove a class meets its specification, using specifications but not code of other classes

- Crucial when dealing with large programs

- Requires no interference from outside
  - Internal sub-objects must be encapsulated
Preventing Representation Exposure

- Say Stack s is implemented with linked list
- Outside objects must not access list nodes
Preventing Representation Exposure

- Say Stack $s$ is implemented with linked list
- Outside objects must not access list nodes

- Program can declare $s$ owns list nodes
- System ensures list is encapsulated in $s$
class TStack {
    TNode head;

    void push(T value) {...}
    T pop() {...}
}

class TNode {
    TNode next;
    T value;
    ...
}

class T {...}
class TStack<stackOwner, TOwner> {
    TNode<this, TOwner> head;
    ...
}
class TNode<nodeOwner, TOwner> {
    TNode<nodeOwner, TOwner> next;
    T<TOwner> value;
    ...
}
TNode objects are encapsulated in TStack object
Preventing Representation Exposure

- **Thread1 objects**
- **Thread2 objects**
- **Potentially shared objects**
Preventing Representation Exposure

Thread1 objects

Thread2 objects

Potentially shared objects
Example of Local Reasoning

class IntVector {
    int size () {...} ...
}

class IntStack {
    void push (int x) {...} ...
}

void m (IntStack s, IntVector v) {
    int n = v.size();  s.push(3);  assert( n == v.size() );
}

Is the condition in the assert true?
Is the condition in the assert true?
Example of Local Reasoning

class IntVector {

  int size () reads (this) {...} ...

}

class IntStack {

  void push (int x) writes (this) {...} ...

}

void m (IntStack s, IntVector v) where !(v <= s) !(s <= v) {

  int n = v.size();  s.push(3);  assert( n == v.size() );

}

size only reads v and its encapsulated objects
push only writes s and its encapsulated objects
Example of Local Reasoning

class IntVector {
    int size () reads (this) {...} ...
}

class IntStack {
    void push (int x) writes (this) {...} ...
}

void m (IntStack s, IntVector v) where !(v <= s) !(s <= v) {
    int n = v.size(); s.push(3); assert( n == v.size() );
}

s is not encapsulated in v, and v is not encapsulated in s
Example of Local Reasoning

class IntVector {
    int size () reads (this) {...} ... 
}

class IntStack {
    void push (int x) writes (this) {...} ... 
}

void m (IntStack s, IntVector v) where !(v <= s) !(s <= v) {
    int n = v.size(); s.push(3); assert( n == v.size() );
}

So size and push cannot interfere
So the condition in the assert must be true
Ownership Types

- **Prevent data races and deadlocks**
  - Boyapati, Rinard (OOPSLA ’01)
  - Boyapati, Lee, Rinard (OOPSLA ’02)

- **Prevent representation exposure**
  - Boyapati, Liskov, Shrira (POPL ’03)

- **Enable safe region-based memory management**
  - Boyapati, Salcianu, Beebee, Rinard (PLDI ’03)

- **Enable safe upgrades in persistent object stores**
  - Boyapati, Liskov, Shrira, Moh, Richman (OOPSLA ’03)
Related Work
Related Work

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

- **Dynamic tools for preventing races and deadlocks**
  - Steele (POPL ’90)
  - Dinning, Schonberg (PPoPP ’90)
  - Savage, Burrows, Nelson, Sobalvarro, Anderson (SOSP ’97)
  - Cheng, Feng, Leiserson, Randall, Stark (SPAA ’98)
  - Praun, Gross (OOPSLA ’01)
  - Choi, Lee, Loginov, O’Callahan, Sarkar, Sridharan (PLDI ’02)

Useful but unsound
Related Work

- **Types for preventing data races**
  - Flanagan, Freund (PLDI ’00)
  - Bacon, Strom, Tarafdar (OOPSLA ’00)
Related Work

- **Types for preventing data races**
  - Flanagan, Freund (PLDI ’00)
  - Bacon, Strom, Tarafdar (OOPSLA ’00)

- **Types for preventing representation exposure**
  - Clarke, Potter, Noble (OOPSLA ’98), (ECOOP ’01)
  - Clarke, Drossopoulou (OOPSLA ’02)
  - Aldrich, Kostadinov, Chambers (OOPSLA ’02)
Related Work

- **Types for preventing data races**
  - Flanagan, Freund (PLDI ’00)
  - Bacon, Strom, Tarafdar (OOPSLA ’00)

- **Types for preventing representation exposure**
  - Clarke, Potter, Noble (OOPSLA ’98), (ECOOP ’01)
  - Clarke, Drossopoulou (OOPSLA ’02)
  - Aldrich, Kostadinov, Chambers (OOPSLA ’02)

- **Types for region-based memory management**
  - Tofte, Talpin (POPL ’94)
  - Christiansen, Henglein, Niss, Velschow (DIKU ’98)
  - Crary, Walker, Morrisett (POPL ’99)
  - Grossman, Morrisett, Jim, Hicks, Wang, Cheney (PLDI ’02)
Related Work

- **Types for preventing data races**
  - Flanagan, Freund (PLDI ’00)
  - Bacon, Strom, Tarafdar (OOPSLA ’00)

- **Types for preventing representation exposure**
  - Clarke, Potter, Noble (OOPSLA ’98), (ECOOP ’01)
  - Clarke, Drossopoulou (OOPSLA ’02)
  - Aldrich, Kostadinov, Chambers (OOPSLA ’02)

- **Types for region-based memory management**
  - Tofte, Talpin (POPL ’94)
  - Christiansen, Henglein, Niss, Velschow (DIKU ’98)
  - Crary, Walker, Morrisett (POPL ’99)
  - Grossman, Morrisett, Jim, Hicks, Wang, Cheney (PLDI ’02)

Our work unifies these areas
Conclusions

Ownership types for object-oriented programs

- Statically prevent several classes of errors
  - Prevent data races and deadlocks
  - Prevent representation exposure
  - Enable region-based memory management
  - Enable upgrades in persistent object stores
- Provide documentation that lives with code
- Require little programming overhead
- Promising way to make programs reliable
Ownership Types for Safe Programming

Chandrasekhar Boyapati

Laboratory for Computer Science
Massachusetts Institute of Technology