Ownership Types for Safe Programming: Preventing Data Races and Deadlocks
(Chandrasekhar, et al.)

Presented by Yi Huang
Fall 2008

A paper presentation for CSE891, *Formal Methods in Software Development: Reliable Computing with Threads*
Overview

Ownership type

Motivation

Type extension for preventing data races

Type extension for preventing deadlocks

Issues and fixes

Summary
Ownership

Intention
• To enforce object encapsulation

Usage
• Specify ownership information by annotating the types of variables, classes, and methods

Rule
• An object can be accessed only by its owner object and all other objects that its owner object owns, transitively
Ownership type

For a variable
- Specify the owner object for the object that the variable points to

For a class
- Specify the owner object for an instance of the class

For a method
- Specify the owner object for the object that the method returns
Example 1

class Car{
    Driver dvr = new Driver;
    Engine egn = new Engine;

    void go(){
        dvr.drink(); //OK but should be disallowed
    }
}

class Main{
    void main(){
        Car car = new Car;

        car.go(); //OK
        car.egn.start(); //OK but should be disallowed
    }
}
Example 2

class Car<owner>{
    Driver<self> dvr = new Driver<self>;
    Engine<this> egn = new Engine<this>;

    void go(){
        dvr.drink(); //FAIL
    }
}

class Main<self>{
    void main(){
        Car<this> car = new Car<this>;

        car.go(); //OK
        car.egn.start(); //FAIL
    }
}
Example 3

class Car<owner>{
    Driver<self> dvr = new Driver<self>;
    Engine<owner> egn = new Engine<owner>;

    void go(){
        dvr.drink(); //FAIL
    }
}

class Main<self>{
    void main(){
        Car<this> car = new Car<this>;

        car.go(); //OK
        car.egn.start(); //OK
    }
}
Owner

Facts

• Every object has only one owner
• An object’s owner cannot be changed
• An object’s owner can be itself
• If object A has a reference to object B, then A is not necessarily B’s owner
• Ownership relation forms a forest
Root owner

Definition
  • An object whose owner is itself

Facts
  • Every object has a root owner, which is not necessarily the same with its owner
  • An object’s root owner is its owner’s root owner
Safe programming

Goal
- Data race and deadlock free

Scope
- Lock-based multi-threaded systems

Solution
- Detecting at runtime
  - Pros: accurate
  - Cons: runtime overhead, incomplete
- Preventing while developing
  - Pros: no runtime overhead
  - Cons: false positives, less expressiveness
Data race

Detection
• Dynamic
  • Happens-before, lockset, hybrid
• Static
  • Flow-sensitive (static version of lockset)

Prevention
• Flow-insensitive type systems
Data race

Detection
- Dynamic
  - Happens-before, lockset, hybrid
- Static
  - Flow-sensitive (static version of lockset)

Prevention
- Flow-insensitive type systems

All we have learned so far are detection techniques
Today we will learn a data race prevention technique
Deadlock

Detection

- Dynamic
  - Timeout, resource-dependency graph
- Static
  - Lock-order graph

Prevention

- Break at least one of the four deadlock conditions (usually “wait-for” cycle)
Deadlock

Detection
- Dynamic
  - Timeout, resource-dependency graph
- Static
  - Lock-order graph

Prevention
- Break at least one of the four deadlock conditions (usually “wait-for” cycle)

So far we have not learned anything above
Today we will learn a deadlock prevention technique
A simple strategy

Every object is associated with a lock

Every lock has a unique priority

There is a complete order among priorities

Every thread acquires/releases the lock associated with an object before/after accessing the object

Every thread acquires locks according to their priorities
Example

Extreme 1: different objects are associated with different locks
  • Awkward programs

Extreme 2: all objects are associated with the same lock
  • Inefficient

Tradeoff: partition objects into groups and each group is associated with a distinct lock
  • More efficient but not that awkward
Challenges

How to group objects?

How to assign lock priorities?

How to supervise the strategy so that a program uses it appropriately?
Proposed solution

How to group objects?
According to ownership (objects owned by the same root owner are of the same group)

How to assign lock priorities?
By programmers

How to supervise the strategy so that a program uses it appropriately?
Through a well-designed type system
Overview of the solution

Programmers annotate types using an extended ownership type system to specify

- Ownership information
- Priorities for root owner objects

Programmers use a provided tool to type check programs

The tool statically verifies that

- Every thread holds an object’s root owner when it accesses the object
- Every thread acquires root owner objects according to the specified priorities
Type extension for preventing data races

A method can have an `accesses` clause to specify what root owners a caller must hold.

Example

- `int pop() accesses (this){
    return queue.last();
}`

- `int pop(){
    synchronized(this){
        return queue.last();
    }
} `
Example 1

class Account<Owner>{
    int _balance = 0;

    void deposit(int x) accesses (this){
        _balance += x;
    }

    void withdraw(int x) accesses (this){
        _balance -= x;
    }
}

class CombinedAccount<self>{
    Account<self> savingsAccount = new Account<self>;
    Account<self> checkingAccount = new Account<self>;
    ...
}
Example 1 (continued)

class Account<Owner>{{
    int _balance = 0;

    void deposit(int x) accesses (this){
        _balance += x;
    }

    void withdraw(int x) accesses (this){
        _balance -= x;
    }
}

class CombinedAccount<self>{
    Account<self> savingsAccount = new Account<self>;
    Account<self> checkingAccount = new Account<self>;

    void goodTransfer(int x){
        synchronized(savingsAccount){
            synchronized(checkingAccount){
                savingsAccount.withdraw(x);
                checkingAccount.deposit(x);
            }
        }
    }

    void badTransfer(int x){
        synchronized(savingsAccount){
            savingsAccount.withdraw(x);
            checkingAccount.deposit(x);
        }
    }
}
Example 2

class Account<Owner>{
    int _balance = 0;
    
    void deposit(int x) accesses (this){
        _balance += x;
    }
    
    void withdraw(int x) accesses (this){
        _balance -= x;
    }
}

class CombinedAccount<self>{
    Account<this> savingsAccount = new Account<this>;
    Account<this> checkingAccount = new Account<this>;
    ...
}
Example 2 (continued)

class Account<Owner>{
    int _balance = 0;

    void deposit(int x) accesses (this){
        _balance += x;
    }

    void withdraw(int x) accesses (this){
        _balance -= x;
    }
}

class CombinedAccount<self>{
    Account<this> savingsAccount = new Account<this>;
    Account<this> checkingAccount = new Account<this>;

    void goodTransfer(int x){
        synchronized(this){
            synchronized(checkingAccount){
                savingsAccount.withdraw(x);
                checkingAccount.deposit(x);
            }
        }
    }

    void badTransfer(int x) accesses(this){
        synchronized(savingsAccount){
            savingsAccount.withdraw(x);
            checkingAccount.deposit(x);
        }
    }
}
Type extension for preventing deadlocks

A class can define a set of locklevels and a partial order among them

Each root owner is associated with a locklevel

A thread must acquire root owners according to the partial order among the locklevels they belong to

A method can have a locks clause to specify the “lowest” locklevels that a caller must hold
Example 1

class Account<Owner>
{
    int _balance = 0;

    void deposit(int x) accesses (this)
    {
        _balance += x;
    }

    void withdraw(int x) accesses (this)
    {
        _balance -= x;
    }
}

class CombinedAccount<self:v>
{
    LockLevel savingsLevel = new;
    LockLevel checkingLevel < savingsLevel;

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

    void goodTransfer(int x)
    locks (savingsLevel)
    {
        synchronized(savingsAccount)
        {
            synchronized(checkingAccount)
            {
                savingsAccount.withdraw(x);
                checkingAccount.deposit(x);
            }
        }
    }

    void badTransfer(int x)
    locks (savingsLevel)
    {
        synchronized(checkingAccount)
        {
            synchronized(savingsAccount)
            {
                savingsAccount.withdraw(x);
                checkingAccount.deposit(x);
            }
        }
    }
}
Example 2

class Account<Owner>{
    int _balance = 0;

    void deposit(int x) accesses (this){
        _balance += x;
    }

    void withdraw(int x) accesses (this){
        _balance -= x;
    }
}

class CombinedAccount<self:v>{
    LockLevel savingsLevel = new;
    LockLevel checkingLevel = savingsLevel

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

    void goodTransfer(int x)
        locks (savingsLevel){
            synchronized(savingsAccount){
                synchronized(checkingAccount){
                    savingsAccount.withdraw(x);
                    checkingAccount.deposit(x);
                }
            }
        }

    void badTransfer(int x)
        locks (savingsLevel){
            synchronized(checkingAccount){
                synchronized(savingsAccount){
                    savingsAccount.withdraw(x);
                    checkingAccount.deposit(x);
                }
            }
        }
}
Example 2

This program can pass type checking, but it has deadlock!

```java
class Account<Owner>
{
    int _balance = 0;

    void deposit(int x) accesses (this){
        _balance += x;
    }

    void withdraw(int x) accesses (this){
        _balance -= x;
    }
}

class CombinedAccount<self:v>{}
LockLevel savingsLevel = new;
LockLevel checkingLevel = new;

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

void goodTransfer(int x) locks (savingsLevel){
    synchronized(savingsAccount){
        synchronized(checkingAccount){
            savingsAccount.withdraw(x);
            checkingAccount.deposit(x);
        }
    }
}

void badTransfer(int x) locks (savingsLevel){
    synchronized(checkingAccount){
        synchronized(savingsAccount){
            savingsAccount.withdraw(x);
            checkingAccount.deposit(x);
        }
    }
}
```

This program can pass type checking, but it has deadlock!
Issues and fixes

Reentrant locking

Ordering of root owners within a locklevel
Reentrant locking

Fix
- **Use locks clause to also specify what other root owners a caller may already hold**

Example
- **void transfer(int x) locks(savingsLevel, this){**
  
  if(balance_check(x)){ ... }
  
  ...}

  boolean balance_check(int x) locks(\texttt{this}){**
  
  ...}
  
  ...}
Ordering of root owners within a locklevel

Fix

- Tree-based partial order
- DAG-based partial order
- Runtime ordering
Example 1

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

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

    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;
            }
        }
    }
}
A problem

Using tree order may introduce cycle in the partial order when the tree structure is changed

Example

```java
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;
}
}
}
```

```
this
/  \
...  x  ...  v
/ \  -->  / \
v  y  u  x
/ \        / \
u  w        w  y
v.right = x;  //Cycle
x.left = w;
this.right = v;
```
Solution

Allow only the root of a tree to be added as a child of another tree

Deficiency

• Less expressive
Example 2

class Account implements Dynamic{
    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;
    }

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

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

Hint: the tool removes Dynamic and modifies the constructor of Account to store a unique ID in the Account object.
Summary

Using the proposed tool, programmers can partition objects into groups, according to ownership relations

Objects of each group is protected by the root owner

Programmers can partition root owners into locklevels and specify a partial order among the locklevels

Programmers can specify partial orders among root owners that belong to the same locklevel

The tool statically verifies that

- Whenever a thread accesses an object, it holds the object’s root owner
- Threads acquire root owners according to the specified partial order