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

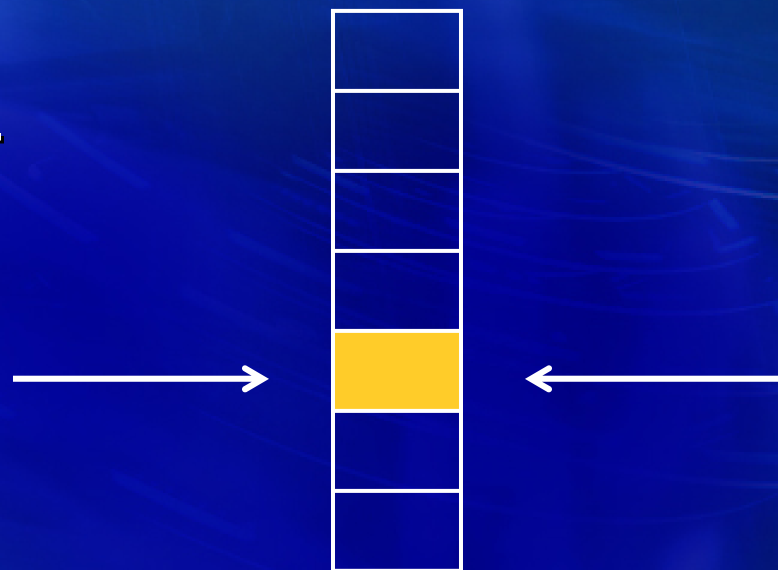
- Preventing other errors

Preventing Data Races

Data Races in Multithreaded Programs

Thread 1:

x = x + 1;



Thread 2:

x = x + 2;

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

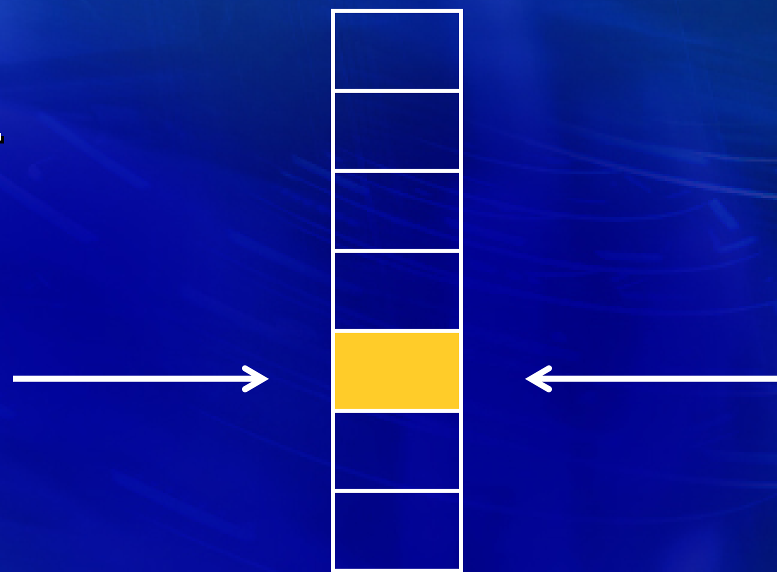
Why Data Races are a Problem

- 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;

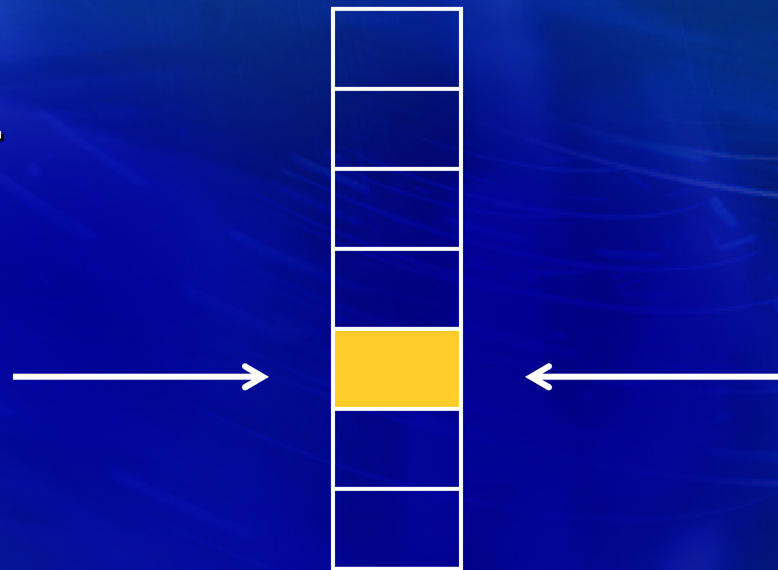
Avoiding Data Races

Thread 1:

lock(l);

x = x + 1;

unlock(l);



Thread 2:

lock(l);

x = x + 2;

unlock(l);

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

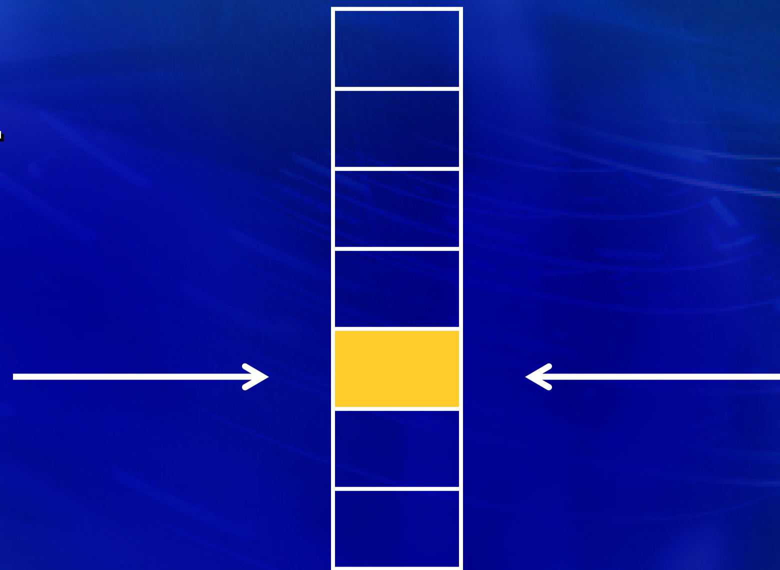
Avoiding Data Races

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

Preventing Data Races

```
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);
```

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);
```

Preventing Data Races

a1 is protected by its own lock
a2 is thread-local

```
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

deposit requires lock on “this”

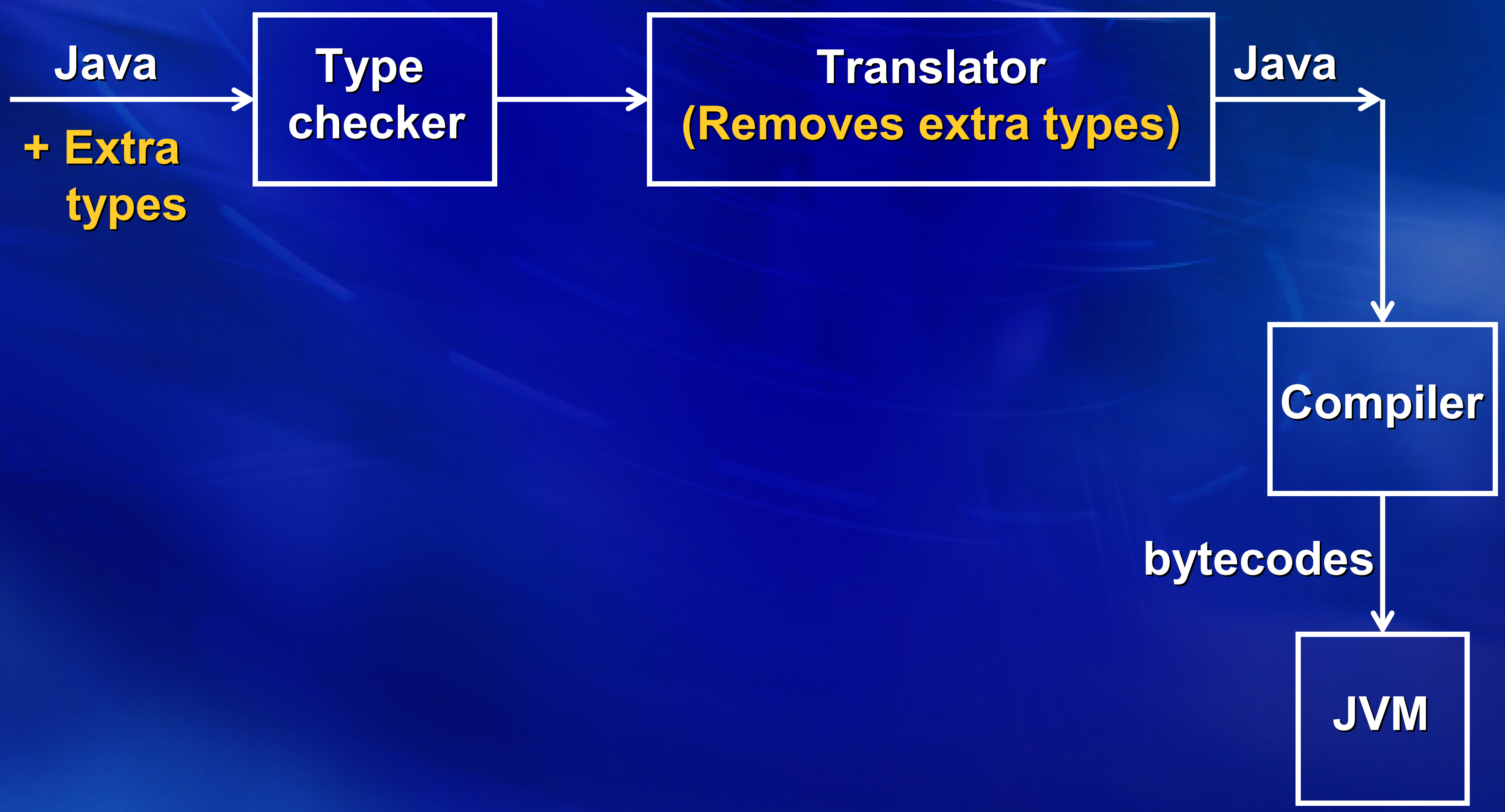
```
class Account {
    int balance = 0;
    void deposit(int x) requires (this)  { balance += x; }
}

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

a1 is locked before calling deposit
a2 need not be locked

```
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



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);
```

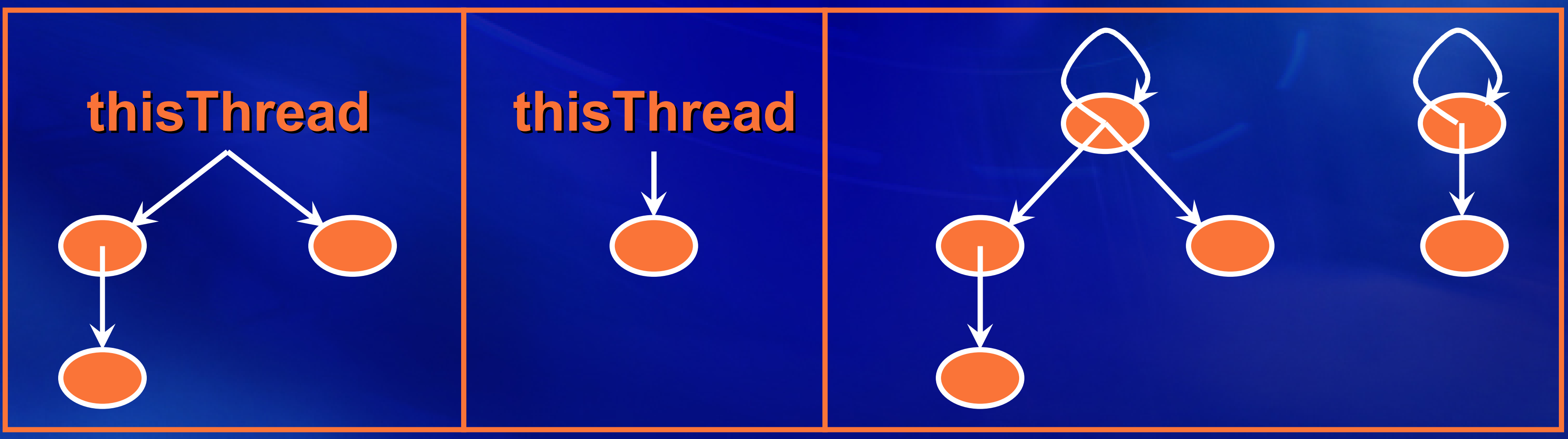
Preventing Data Races

```
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

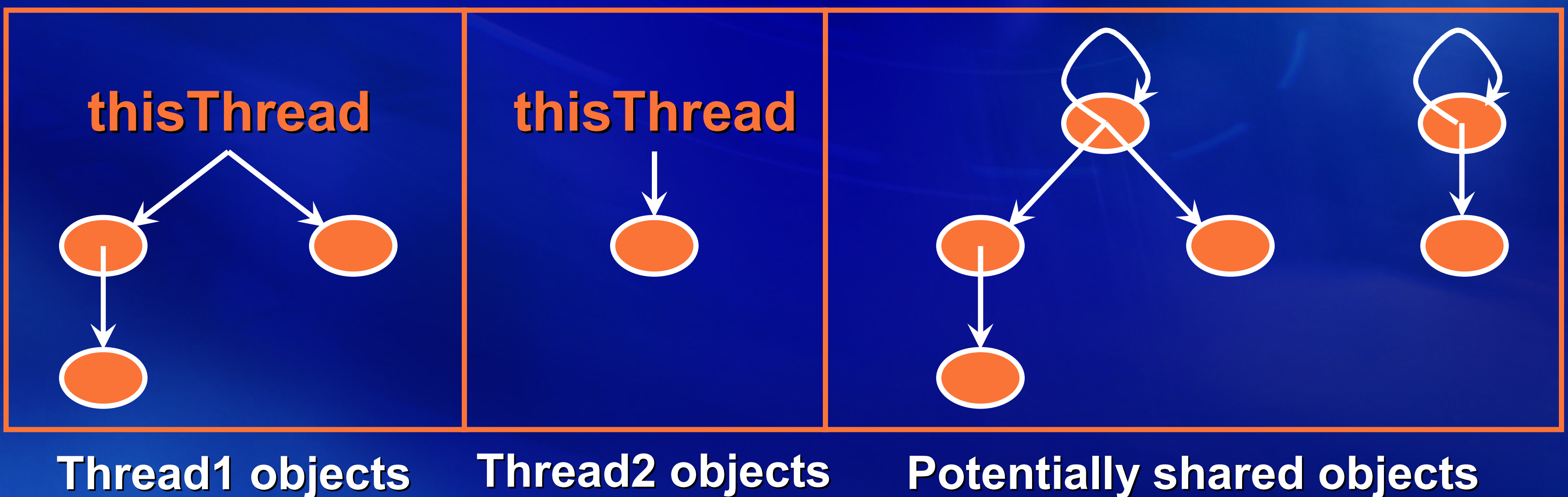
- Every object is owned by
  - Itself, or
  - Another object, or
  - Special per-thread owner called thisThread
- Ownership relation forms a forest of trees



Thread1 objects　Thread2 objects　Potentially shared objects

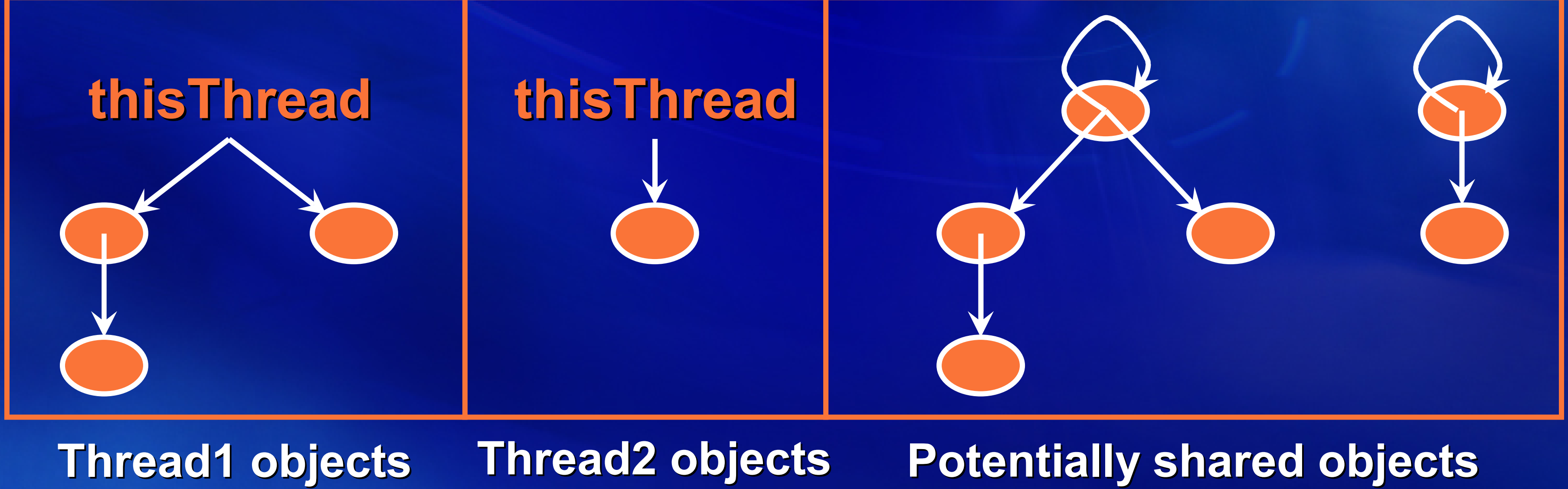
Object Ownership

- 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



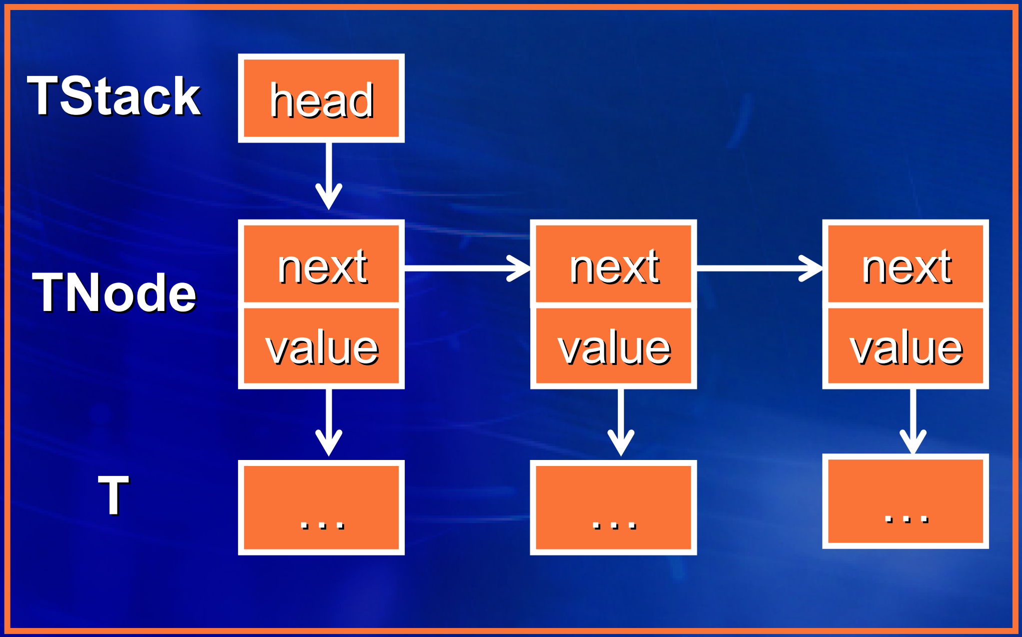
Object Ownership

- 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



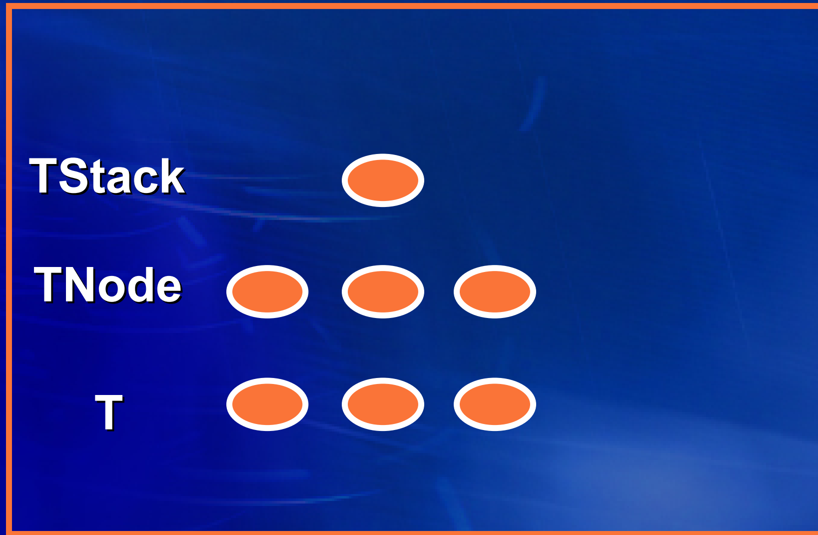
TStack Example

```
class TStack {
    TNode head;

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

class TNode {
    TNode next;
    T value;
    …
}

class T {…}
```



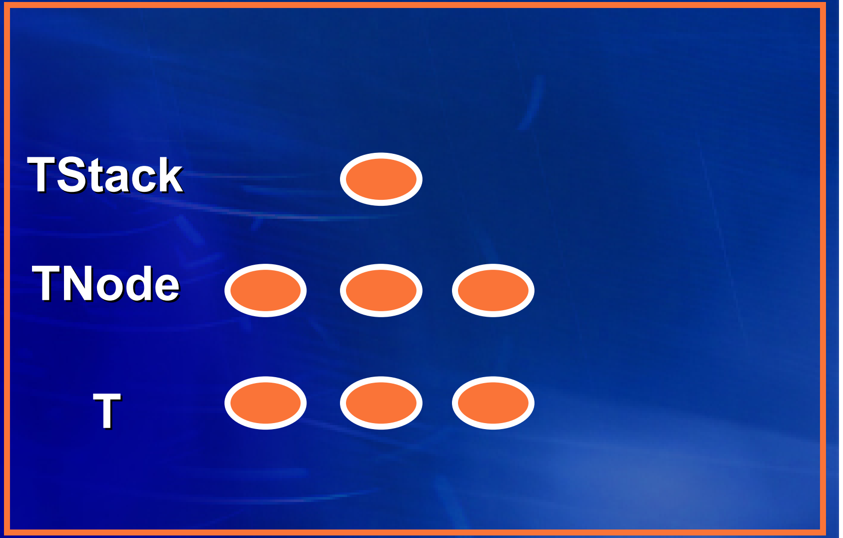
TStack Example

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
```



TStack Example

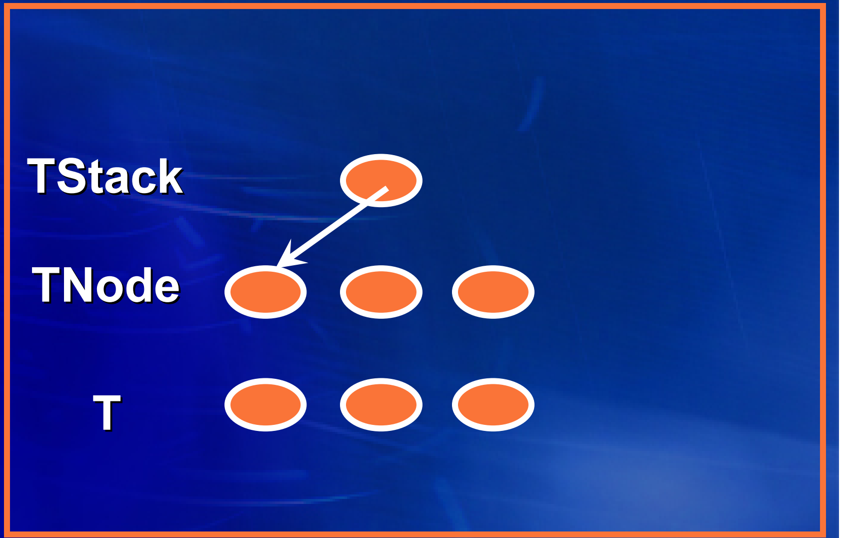
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
```



Classes are parameterized with owners
First owner owns the “this” object

TStack Example

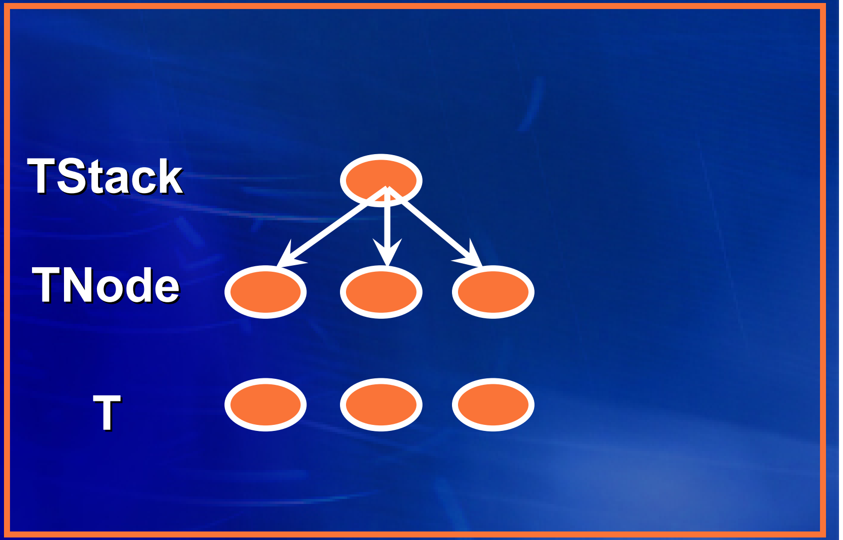
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
```



TStack owns the head TNode

TStack Example

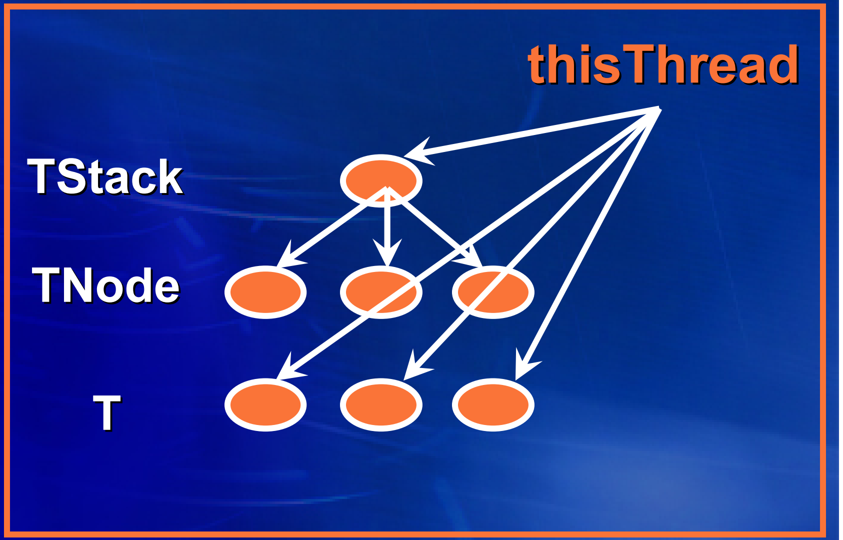
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
```



All TNodes have the same owner

TStack Example

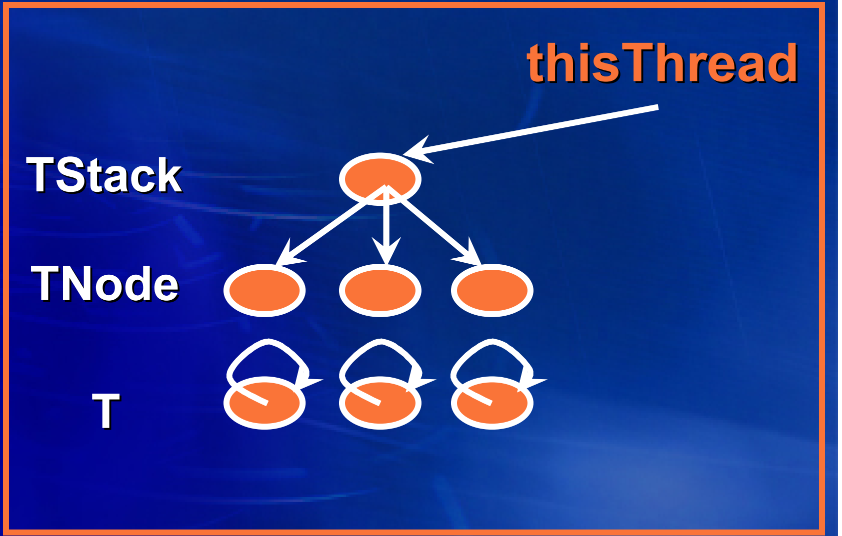
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
TStack⟨thisThread, thisThread⟩ s1;
TStack⟨thisThread, self⟩ s2;
TStack⟨self, self⟩ s3;
```



s1 is a thread-local stack with thread-local elements

TStack Example

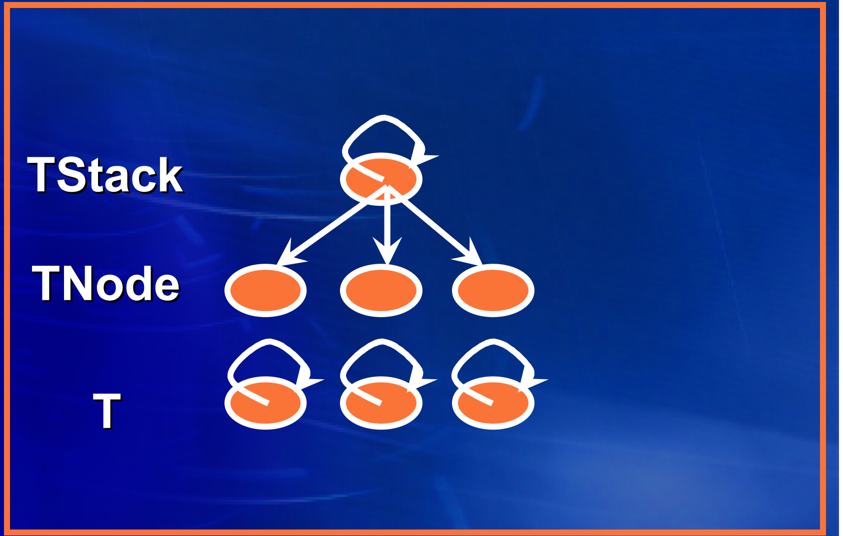
```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
TStack⟨thisThread, thisThread⟩ s1;
TStack⟨thisThread, self⟩ s2;
TStack⟨self, self⟩ s3;
```



s2 is a thread-local stack with shared elements

TStack Example

```
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

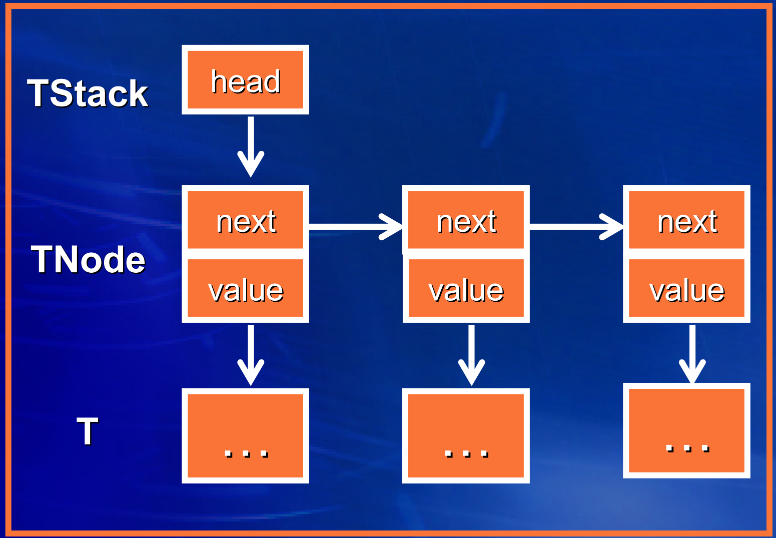
TStack Example

```
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) {…}
   …
}
```

Methods can require callers to hold locks on root owners

Type Checking Pop Method

```
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) {…}
    …
}
```



Type Checking Pop Method

```
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)

Type Checking Pop Method

```
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)

Type Checking Pop Method

```
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

?

Type Checking Pop Method

```
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(head)

Type Checking Pop Method

```
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(head)
= RootOwner(this)

Type Checking Pop Method

```
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),

RootOwner(head)
= RootOwner(this)

Type Checking Pop Method

```
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?

Initially:

x=0;

y=1;

Thread 1:

y=0;

x=1;

Thread 2:

z=x+y;

Possible Interleavings

z=x+y;	y=0;	y=0;
y=0;	z=x+y;	x=1;
x=1;	x=1;	z=x+y;
z=1	z=0	z=1

What is the value of z?

Initially:

x=0;

y=1;

Thread 1:

y=0;

x=1;

Thread 2:

z=x+y;

What is the value of z?

Possible Interleavings

z=x+y;	y=0;	y=0;
y=0;	z=x+y;	x=1;
x=1;	x=1;	z=x+y;
z=1	z=0	z=1

x=1;

z=x+y;

y=0;

z=2 !!!

Above instruction reordering legal in single-threaded programs

Violates sequential consistency in multithreaded programs

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

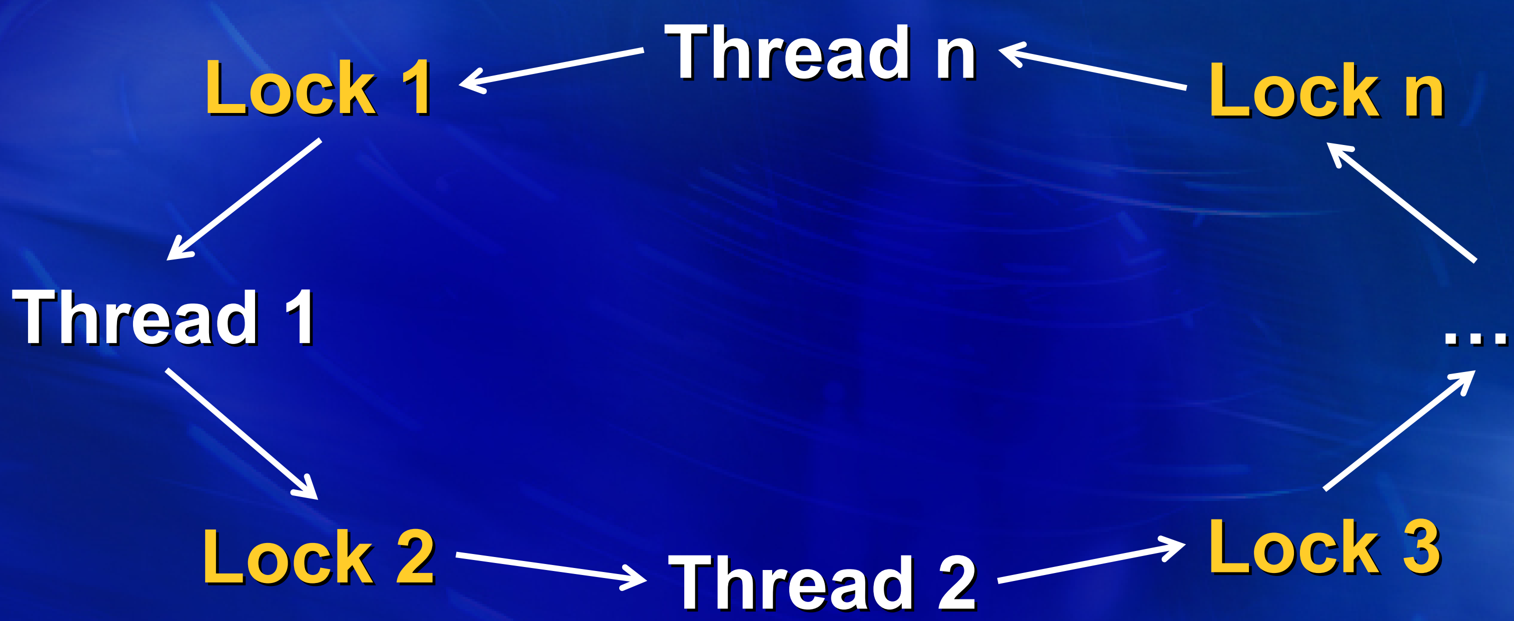
- Race-free languages
  - Eliminate these issues
  - Make multithreaded programming tractable

Outline

- Preventing data races
- Preventing deadlocks
- Type inference
- Experience

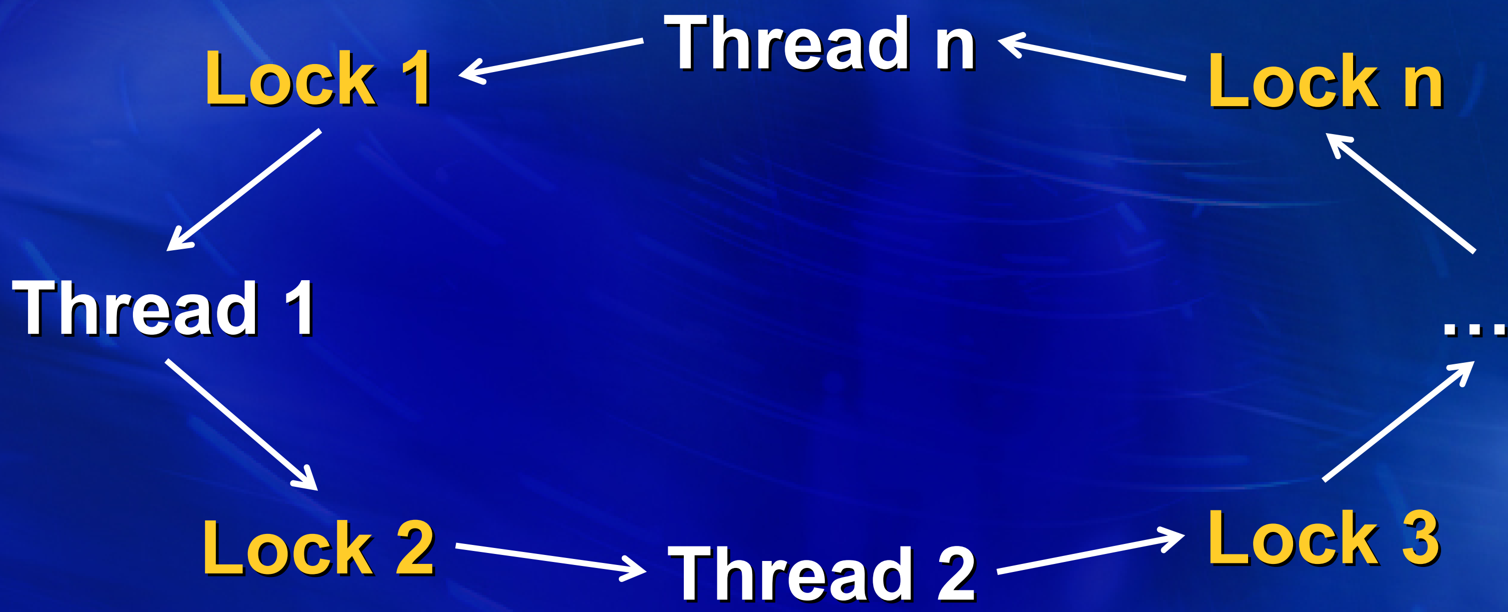
- Preventing other errors

Deadlocks in Multithreaded Programs

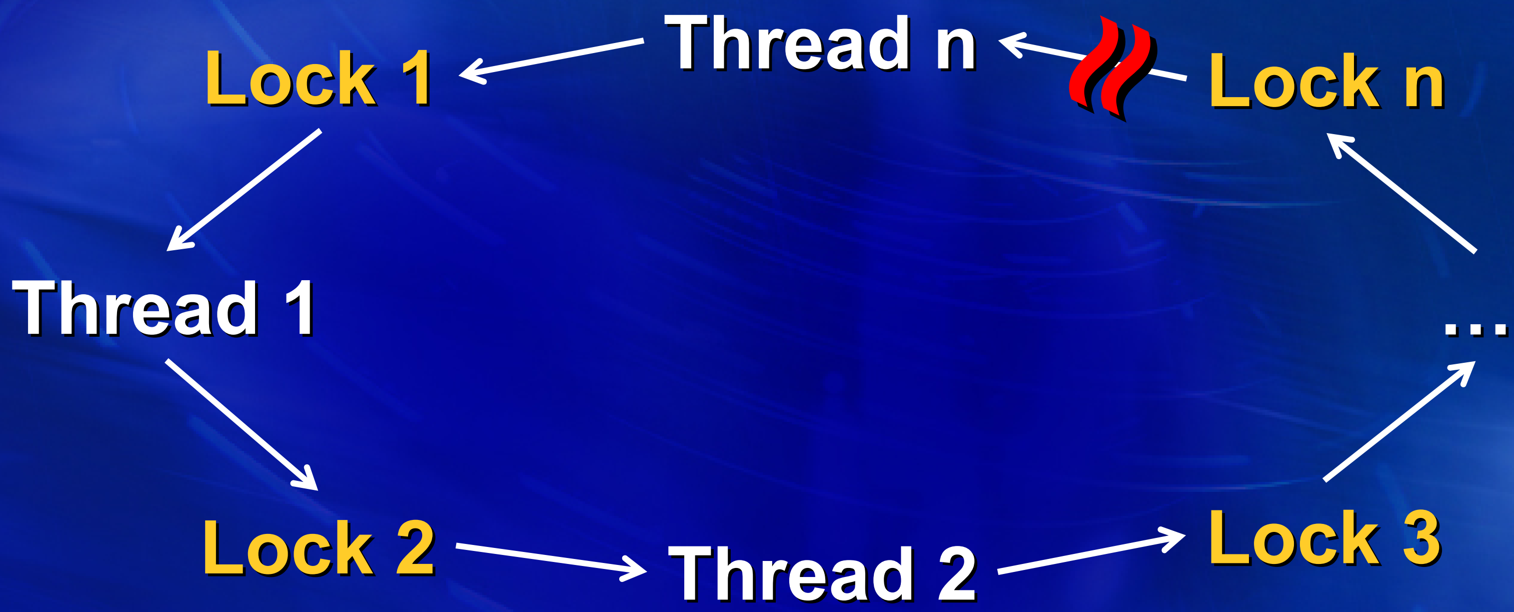


- Cycle of the form
  - Thread 1 holds Lock 1, waits for Lock 2
  - Thread 2 holds Lock 2, waits for Lock 3 …
  - Thread n holds Lock n, waits for Lock 1

Avoiding Deadlocks

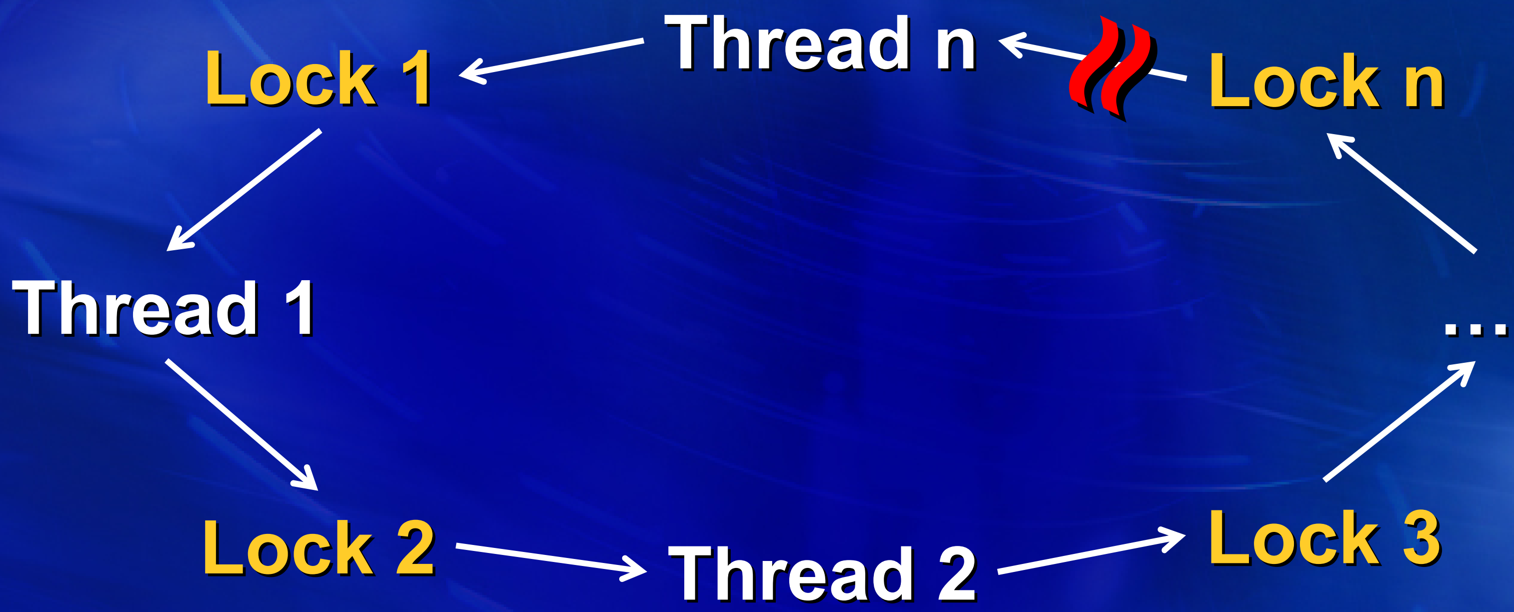


Avoiding Deadlocks



- Associate a partial order among locks
- Acquire locks in order

Avoiding Deadlocks



Problem: Lock ordering is not enforced!

Inadvertent programming errors…

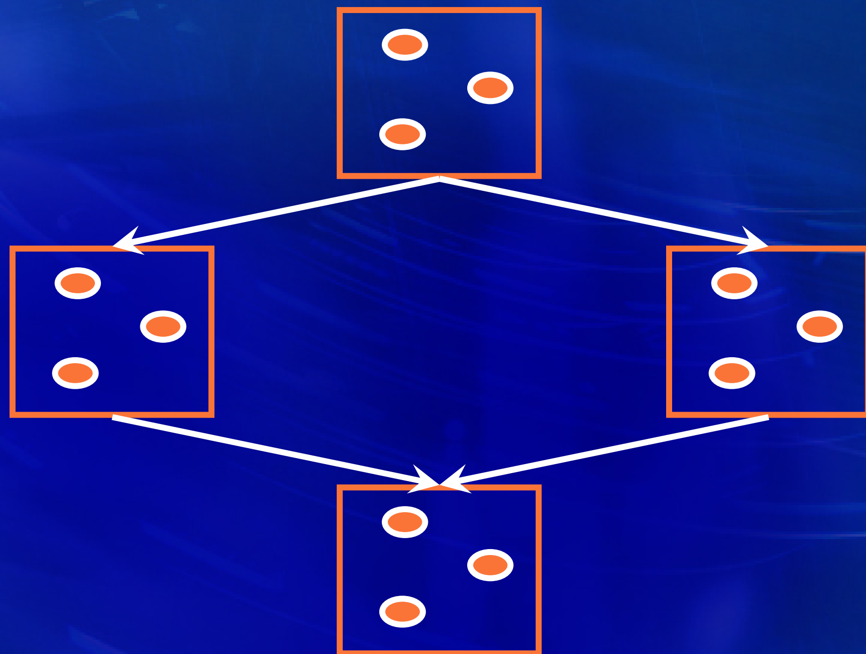
Our Solution

- 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

Lock Level Based Partial Orders

```
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;
    }}}
}
```

Lock Level Based Partial Orders

checkingLevel < savingsLevel

```
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

savingsAccount belongs to savingsLevel
checkingAccount belongs to checkingLevel

```
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

locks are acquired in descending order

```
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

locks held by callers > savingsLevel

```
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

balance can acquire these locks

```
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

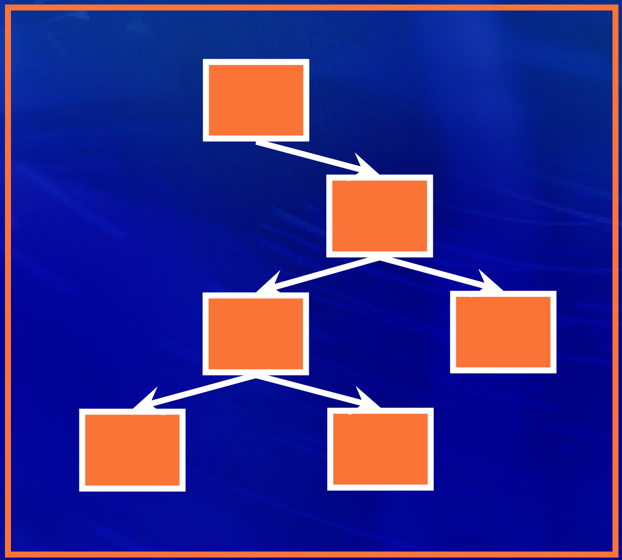
- Lock levels support programs where the maximum number of locks simultaneously held by a thread is bounded

- 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

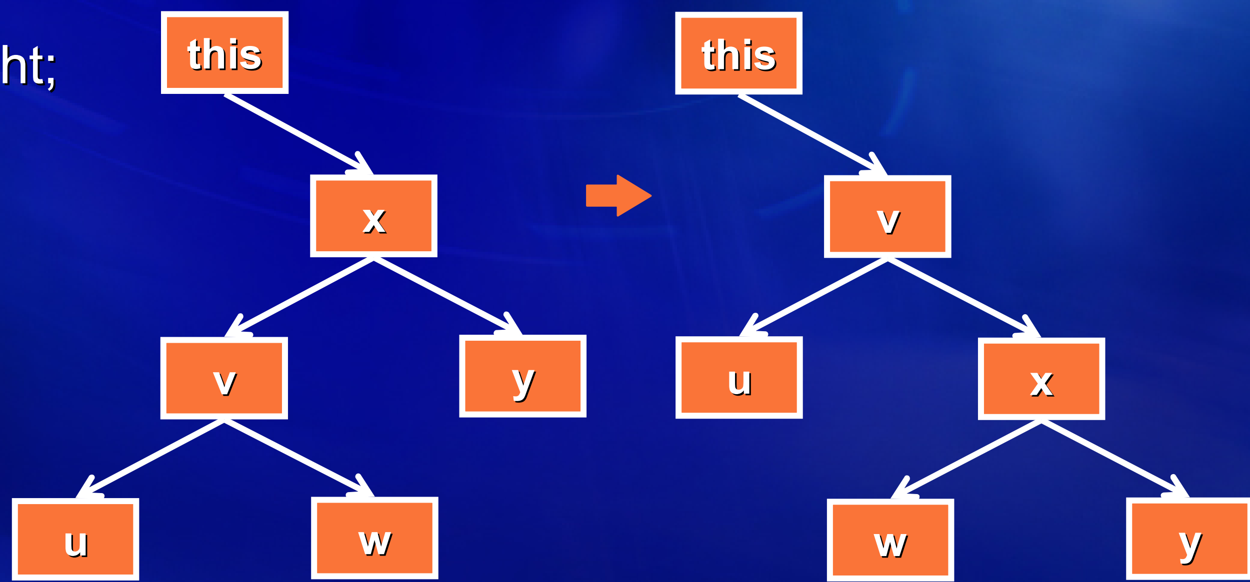
Tree Based Partial Orders



- 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…

Tree Based Partial Orders

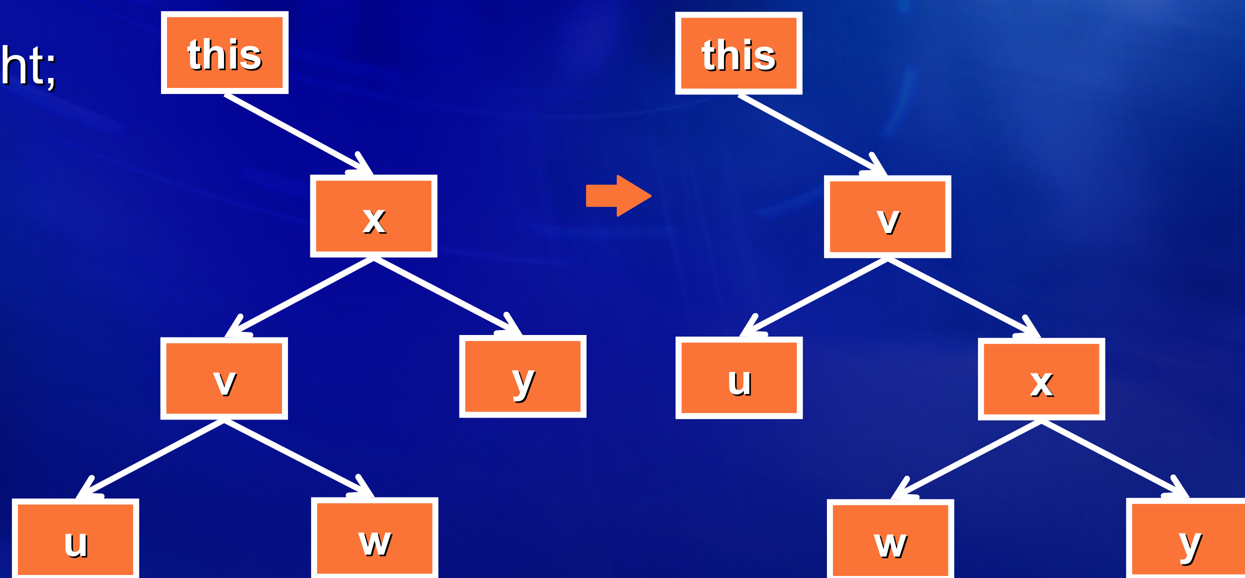
```
class Node {
    Node left;
    Node right;

    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

nodes must be locked in tree order

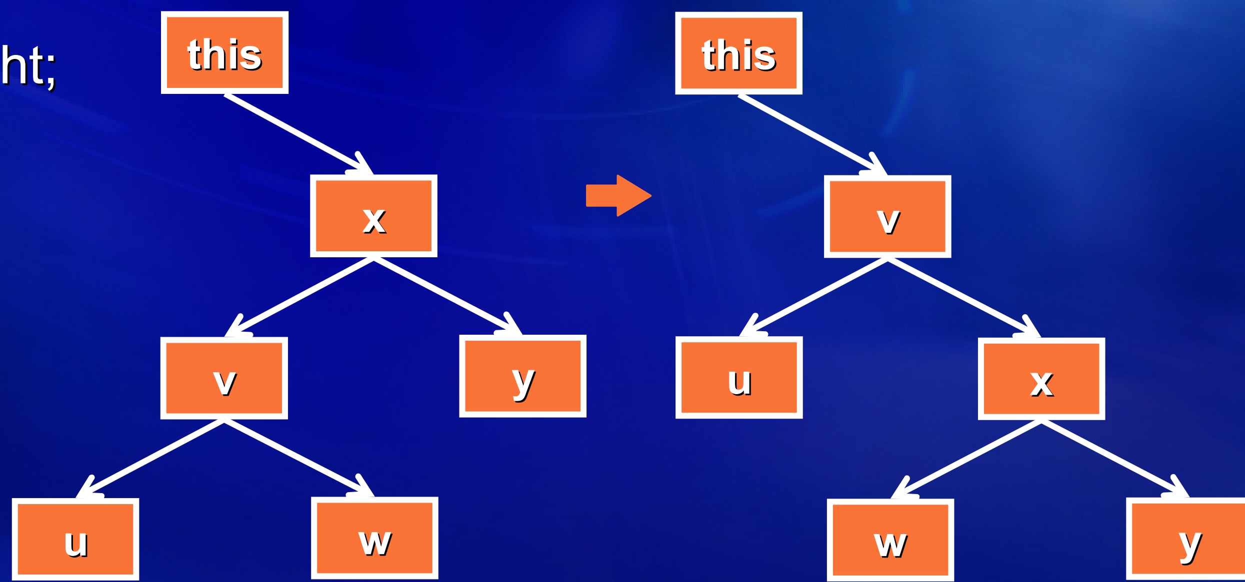
```
class Node⟨self : l⟩ {
    tree Node⟨self : l⟩ left;
    tree Node⟨self : l⟩ right;

    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;
    }}}
}
```



Tree Based Partial Orders

nodes are locked in tree order

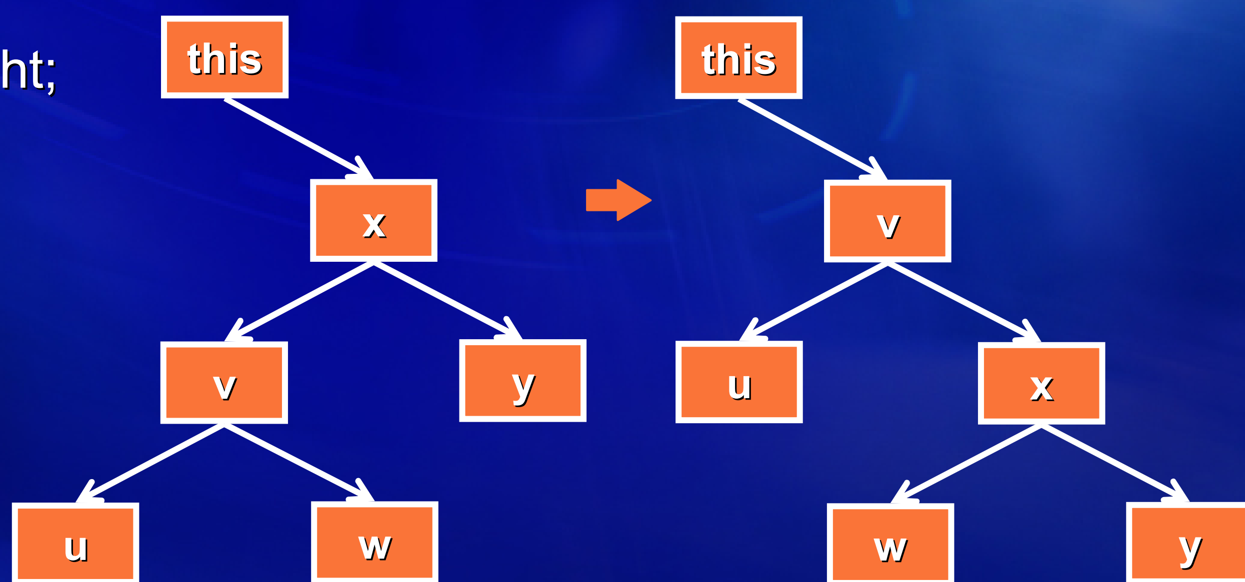
```
class Node⟨self : l⟩ {
    tree Node⟨self : l⟩ left;
    tree Node⟨self : l⟩ right;

    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;
    }}}
}
```



Tree Based Partial Orders

```
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;
    }}}
}
```

flow sensitive analysis checks that tree order is preserved

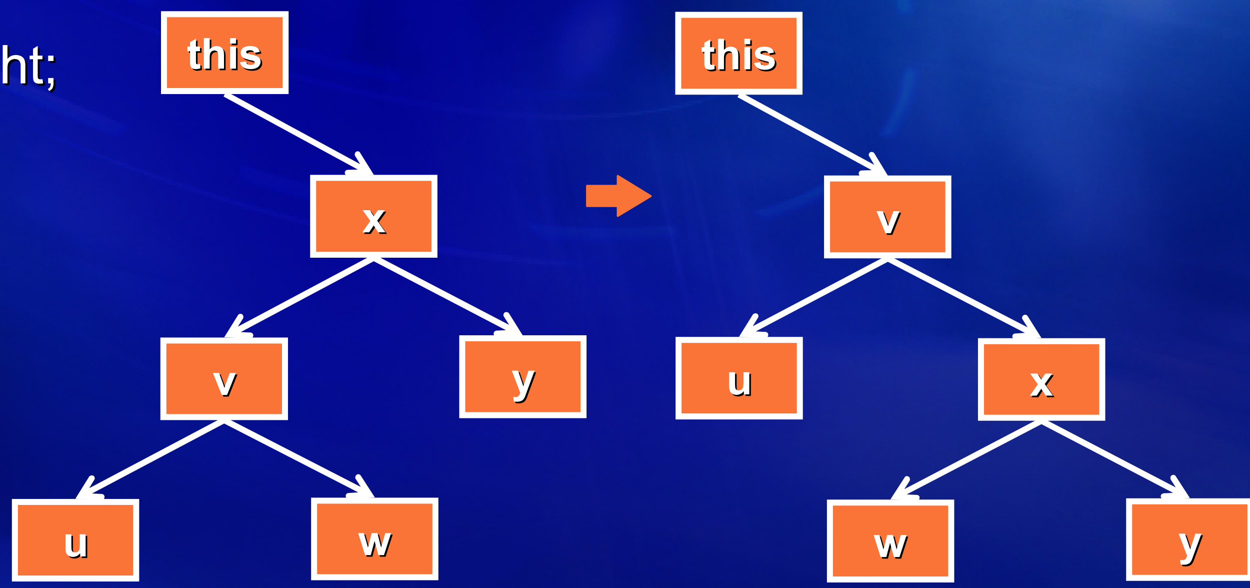


Checking Tree Mutations

- 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

Checking Tree Mutations

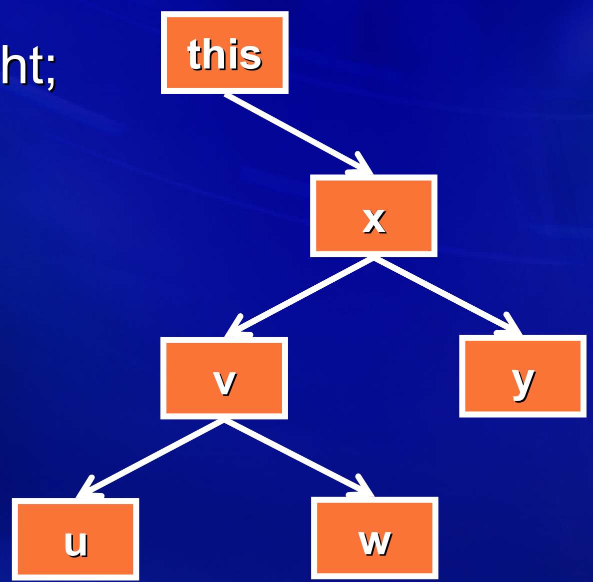
```
class Node⟨self : l⟩ {
  tree Node⟨self : l⟩ left;
  tree Node⟨self : l⟩ right;

  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;
  }}}
}
```



Checking Tree Mutations

```
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;
   }}}
}
```

x = this.right
v = x.left
w = v.right



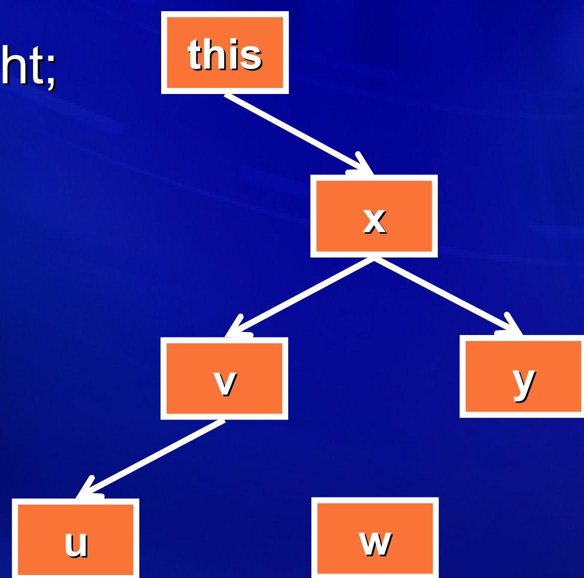
Checking Tree Mutations

```
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;
  }}}
}
```

x = this.right
v = x.left

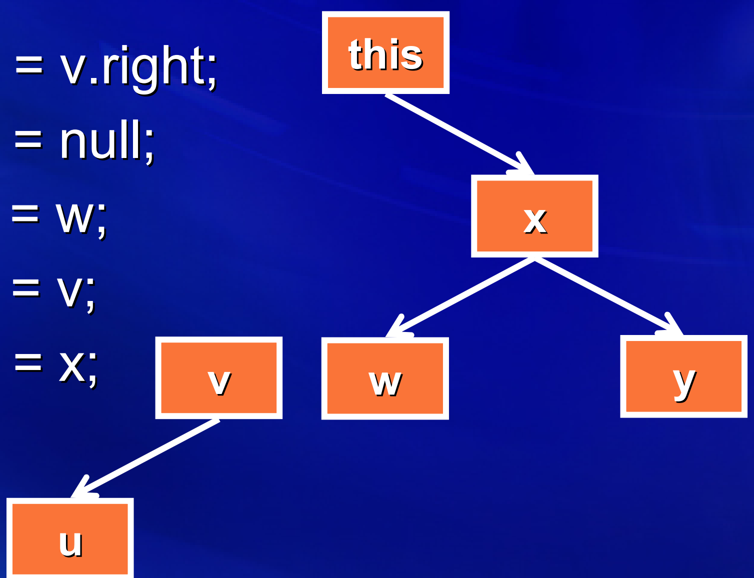
w is Root

v not in Tree(w)
x not in Tree(w)
this not in Tree(w)



Checking Tree Mutations

```
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;
  }}}
}
```



x = this.right
w = x.left

v is Root

x not in Tree(v)
w not in Tree(v)
this not in Tree(v)

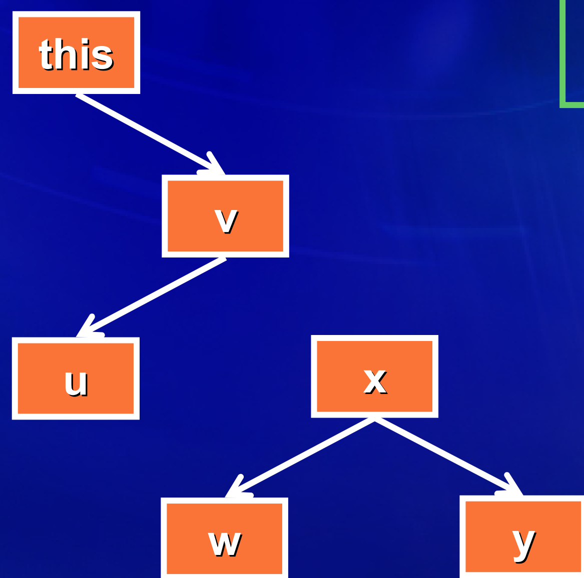
Checking Tree Mutations

```
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;
  }}}
}
```

v = this.right
w = x.left

x is Root

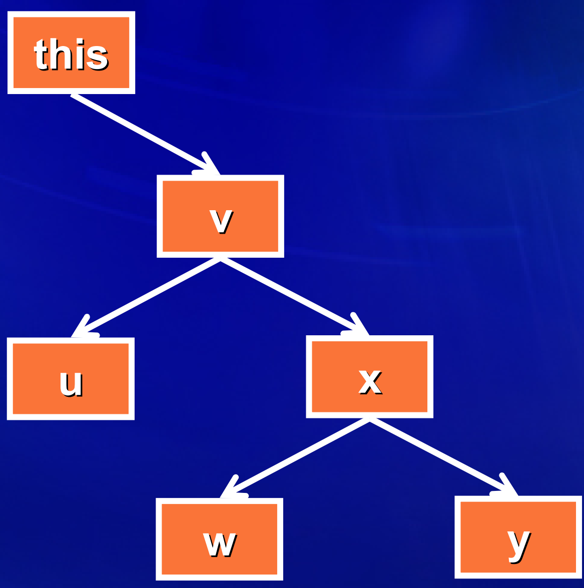
this not in Tree(x)
v not in Tree(x)



Checking Tree Mutations

```
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;
    }}}
}
```

v = this.right
w = x.left
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

```
class Node⟨self : l⟩ {
    dag Node⟨self : l⟩ left;
    dag Node⟨self : l⟩ 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

Runtime Ordering of Locks

```
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); }
}
```

Runtime Ordering of Locks

```
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); }
}
```

Runtime Ordering of Locks

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); }
}
```

Runtime Ordering of Locks

locks are acquired in runtime order

```
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

x3 = this
x4 = oc1
x5 = thisThread
x1 = x3
x2 = x5

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

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

Program		# Lines of code	# Lines annotated
SMTP Server	(Apache)	2105	46
POP3 Mail Server	(Apache)	1364	31
Discrete Event Simulator	(ETH Zurich)	523	15
HTTP Server		563	26
Chat Server		308	22
Stock Quote Server		242	12
Game Server		87	11
Database Server		302	10

Java Libraries

Program	# Lines of code	# Lines annotated
java.util.Hashtable	1011	53
java.util.HashMap	852	46
java.util.Vector	992	35
java.util.ArrayList	533	18
java.io.PrintStream	568	14
java.io.FilterOutputStream	148	5
java.io.BufferedWriter	253	9
java.io.OutputStreamWriter	266	11

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

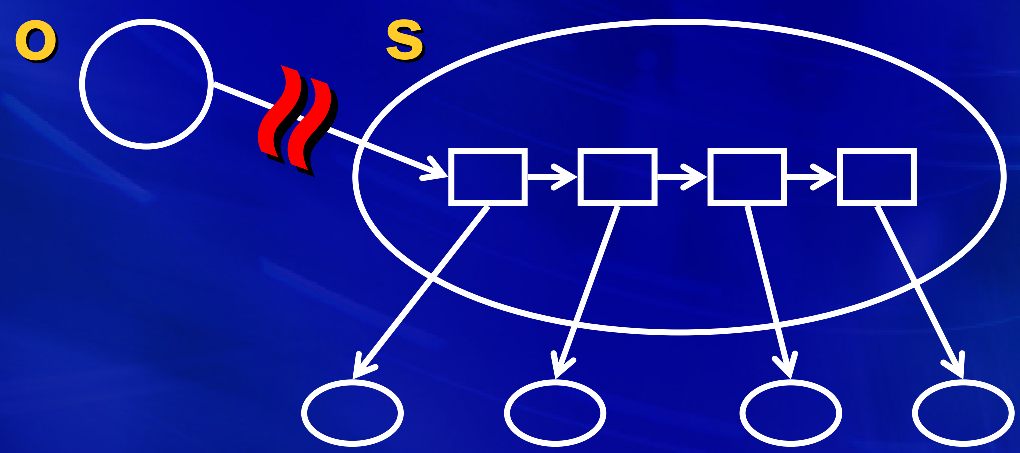
- Prevent data races and deadlocks
  - Boyapati, Rinard (OOPSLA ’01)
  - Boyapati, Lee, Rinard (OOPSLA ’02)
- Prevent representation exposure
  - Boyapati, Liskov, Shrira (POPL ’03)
- Enable safe region-based memory management
  - Boyapati, Salcianu, Beebee, Rinard (PLDI ’03)
- Enable safe upgrades in 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

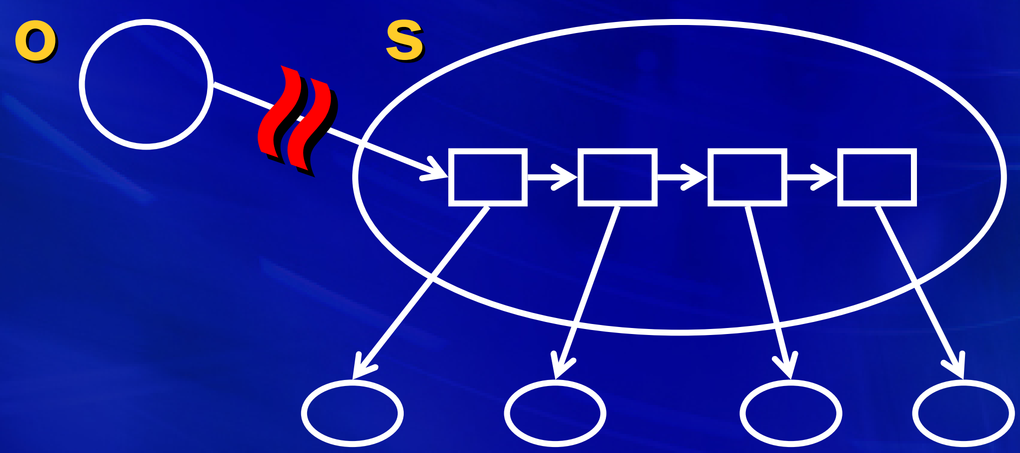
Preventing Representation Exposure

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



Preventing Representation Exposure

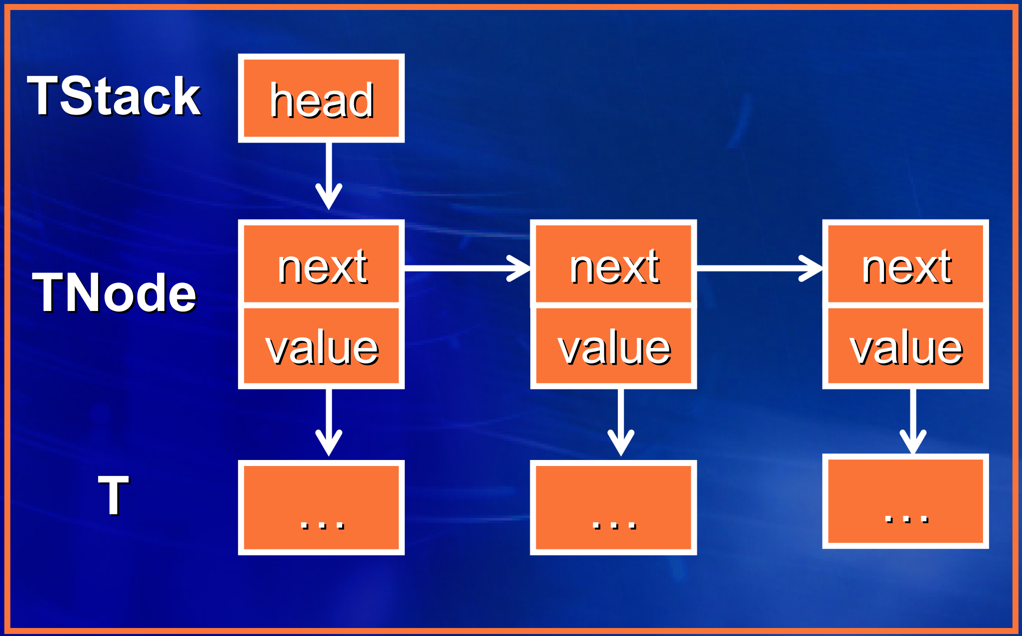
- Say Stack s is implemented with linked list
- Outside objects must not access list nodes



- Program can declare s owns list nodes
- System ensures list is encapsulated in s

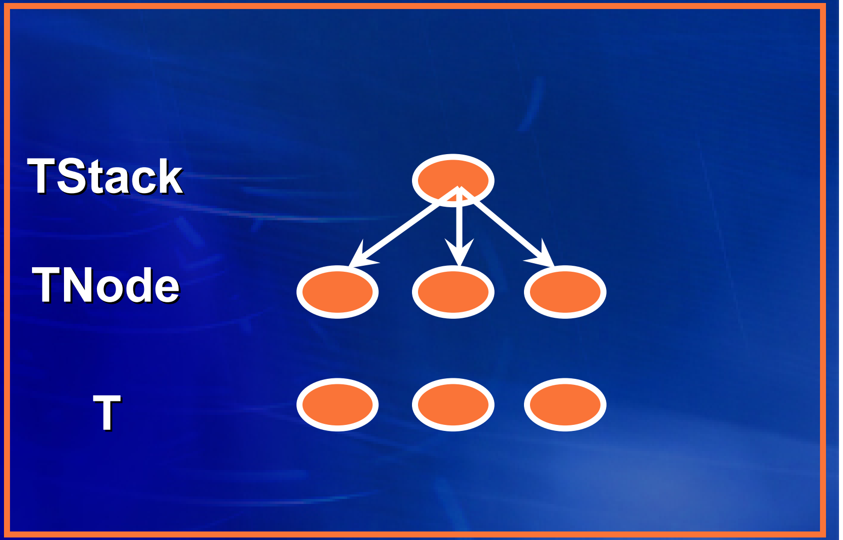
Preventing Representation Exposure

```
class TStack {
    TNode head;

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

class TNode {
    TNode next;
    T value;
    …
}

class T {…}
```



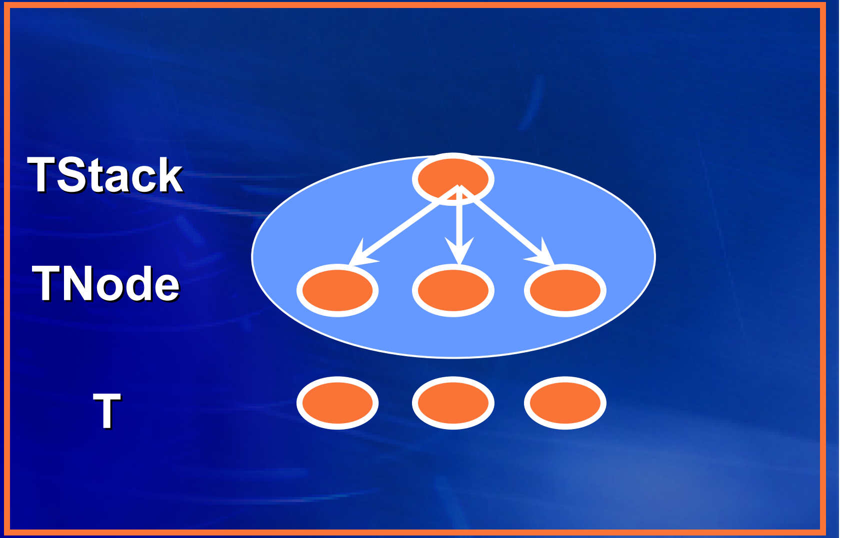
Preventing Representation Exposure

```
   class TStack⟨stackOwner, TOwner⟩ {
➡     TNode⟨this, TOwner⟩ head;
      …
   }
   class TNode⟨nodeOwner, TOwner⟩ {
➡     TNode⟨nodeOwner, TOwner⟩ next;
      T⟨TOwner⟩ value;
      …
   }
```



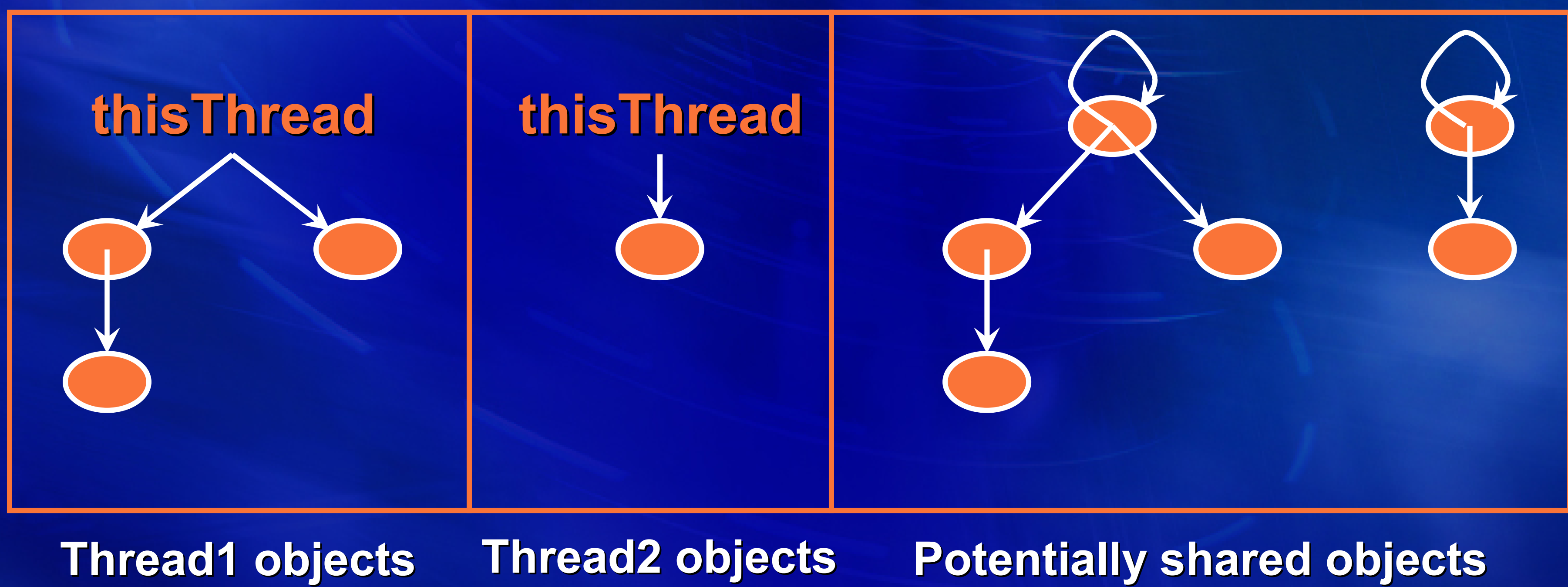
Preventing Representation Exposure

```
class TStack⟨stackOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨nodeOwner, TOwner⟩ {
    TNode⟨nodeOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
```

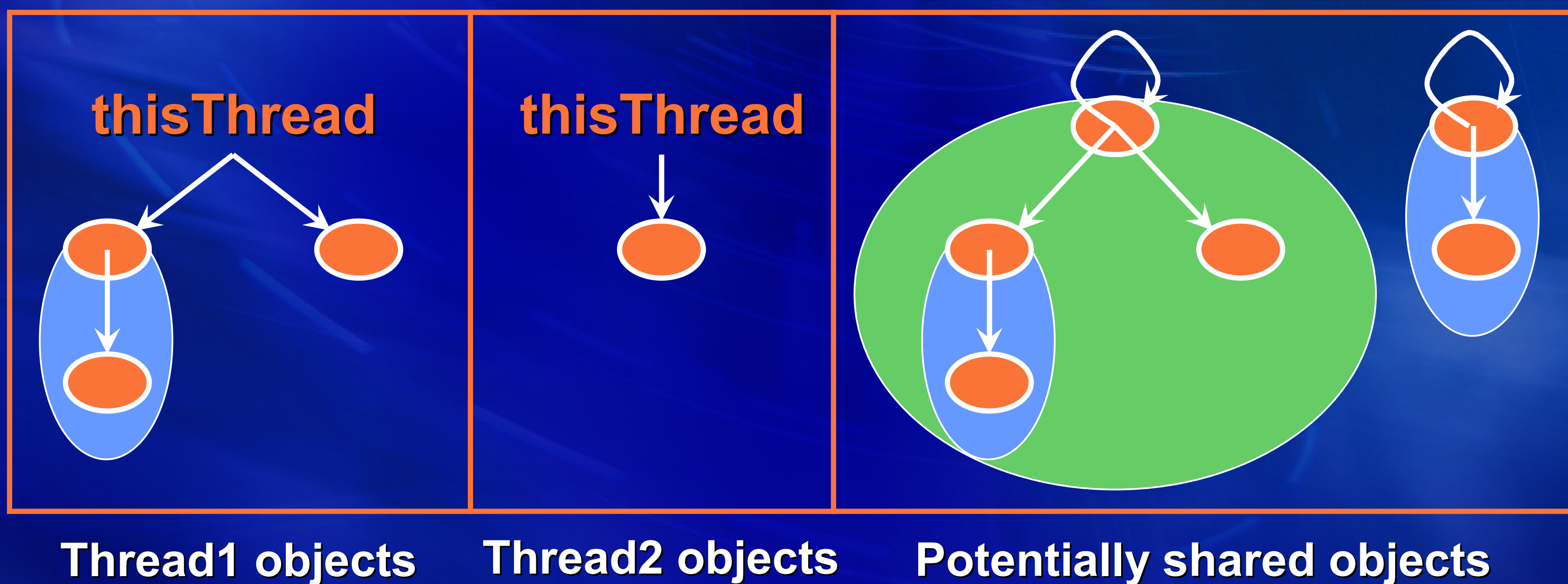


TNode objects are encapsulated in TStack object

Preventing Representation Exposure



Preventing Representation Exposure



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?

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() );
}
```

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