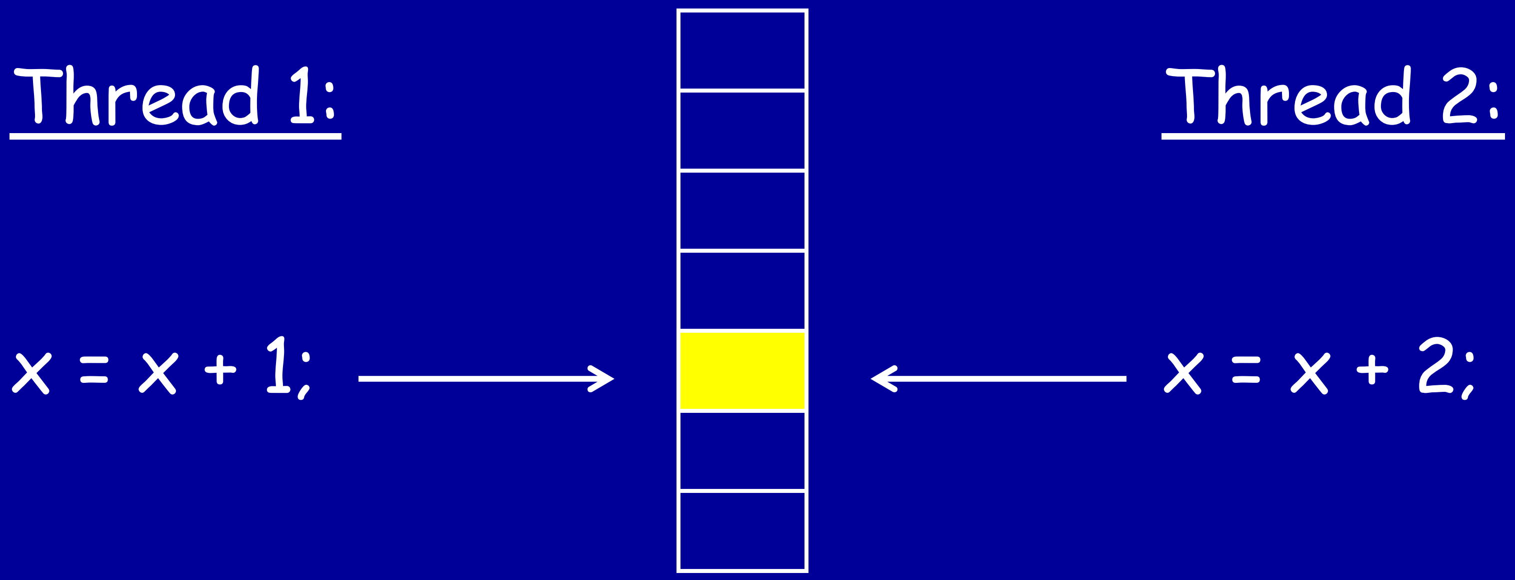
A Parameterized Type System for Race-Free Java Programs

Chandrasekhar Boyapati   Martin Rinard

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

Data races in multithreaded programs

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



Thread 1:

x = x + 1;

Thread 2:

x = x + 2;

Why data races are a problem

- Some correct programs contain data races

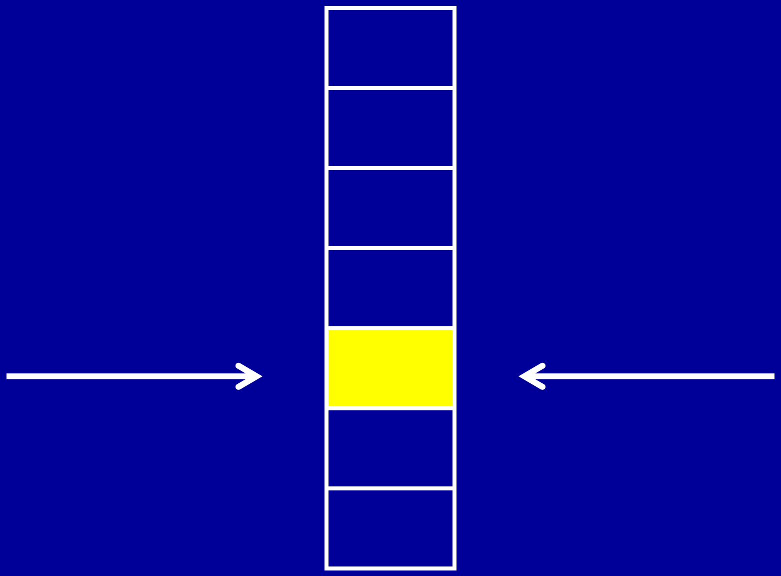
- But most races are programming errors
  - Code intended to execute atomically
  - Synchronization omitted by mistake

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

Avoiding data races

Thread 1:

x = x + 1;



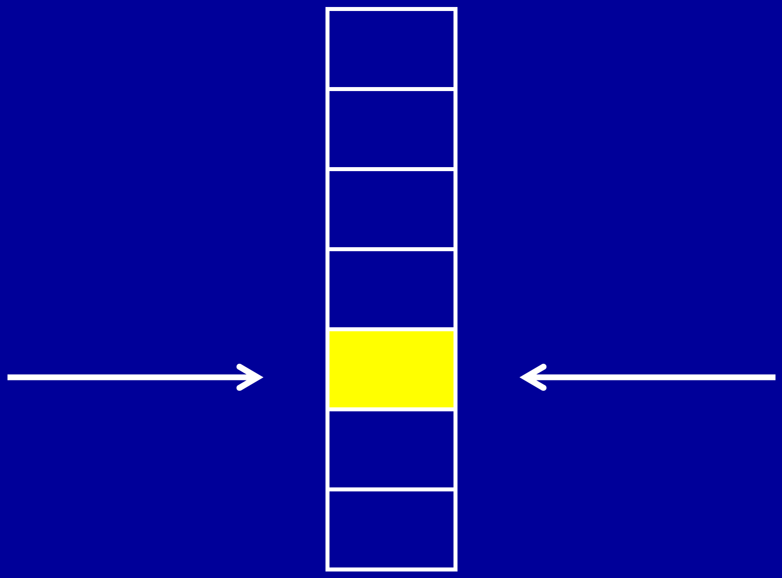
Thread 2:

x = x + 2;

Avoiding data races

Thread 1:

```
lock(l);
x = x + 1;
unlock(l);
```



Thread 2:

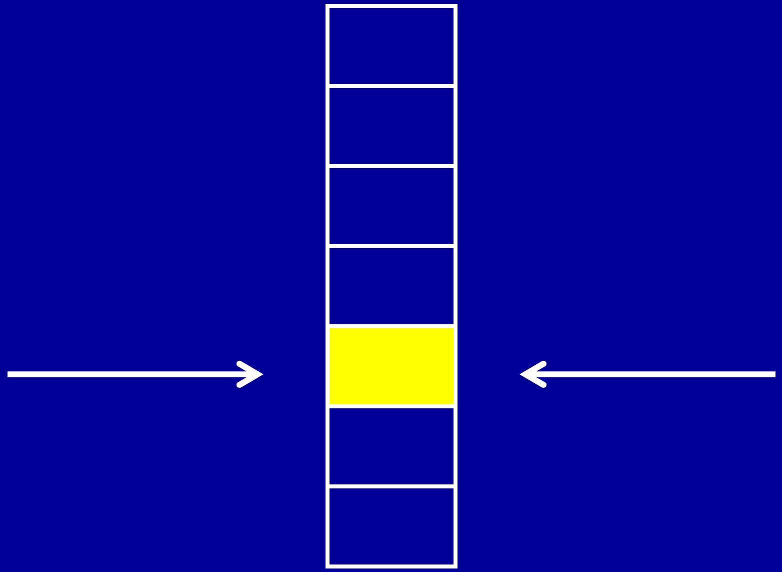
```
lock(l);
x = x + 2;
unlock(l);
```

- Associate a lock with every 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

- A static type system for OO programs
- Well-typed programs are free of races

Our solution

- A static type system for OO programs
- Well-typed programs are free of races

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

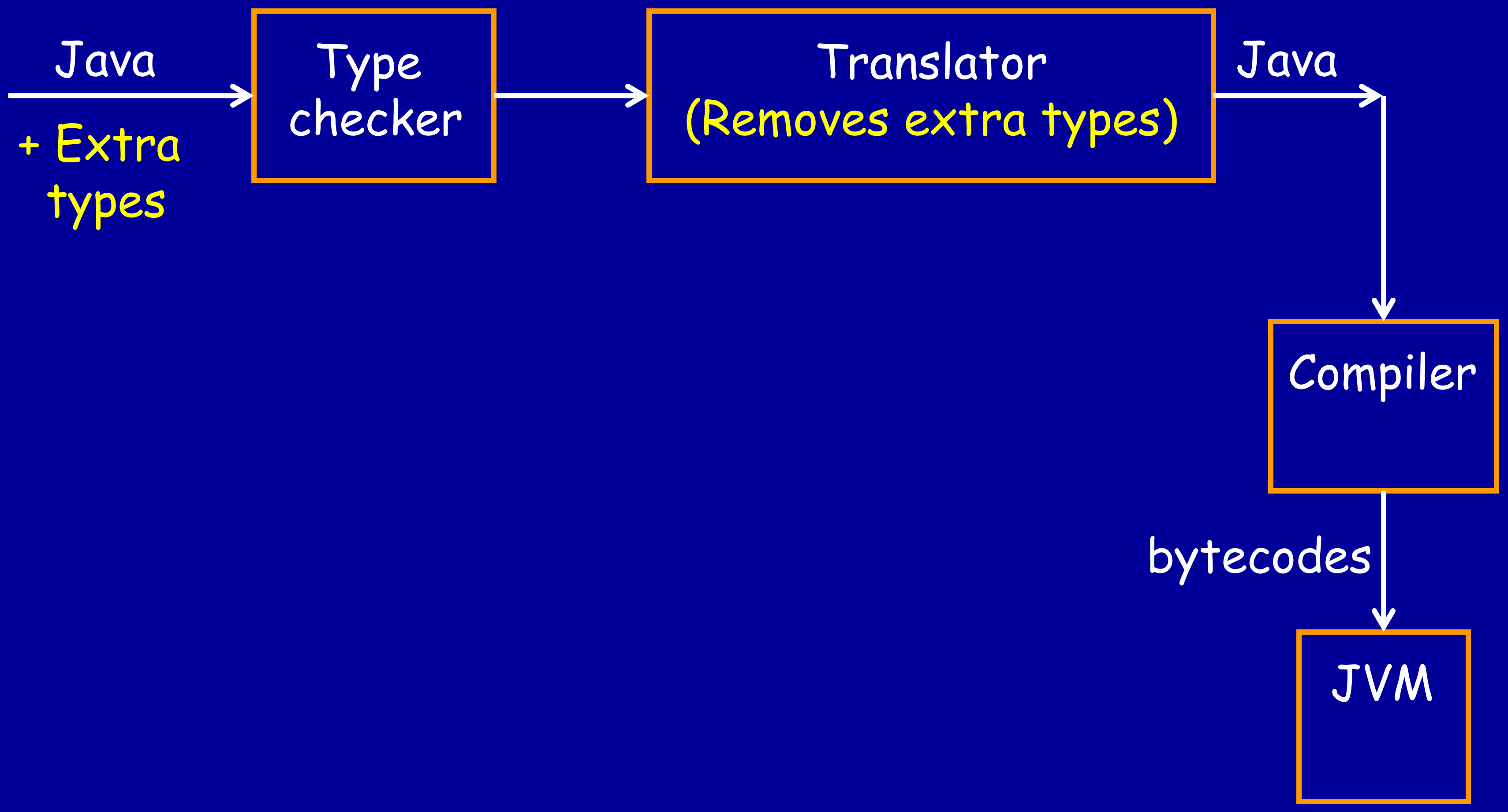
- Type checkers statically verify
  - That objects are used only as specified

Protection mechanism of an object

- Specifies the lock protecting the object, or

- Specifies that object needs no locks b'cos
  - The object is immutable, or
  - The object is not shared, or
  - There is a unique pointer to the object

Types are proofs



Outline

- Motivation
- Type system
- Experience
- Related work
- Conclusions

Race-free Account program

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

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

Account a2 = new Account;
a2.deposit(10);
```

Race-free Account program

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

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

Account a2 = new Account;
a2.deposit(10);
```

Java:

```
Thread t;
t.start();
```

≡

Concurrent Java:

```
fork (t) { t.start(); }
```

Race-free Account program

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

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

Account a2 = new Account;
a2.deposit(10);
```

Statically verifiable race-free program

```
class Account⟨thisOwner⟩ {
    int balance = 0;
    int deposit(int x) requires (this) {
        this.balance += x;
    }
}

final Account⟨self⟩ a1 = new Account⟨self⟩;
fork (a1) { synchronized (a1) in { a1.deposit(10); } };
fork (a1) { synchronized (a1) in { a1.deposit(10); } };

Account⟨thisThread⟩ a2 = new Account⟨thisThread⟩;
a2.deposit(10);
```

Statically verifiable race-free program

thisOwner protects the Account

```
class Account⟨thisOwner⟩ {
    int balance = 0;
    int deposit(int x) requires (this) {
        this.balance += x;
    }
}

final Account⟨self⟩ a1 = new Account⟨self⟩;
fork (a1) { synchronized (a1) in { a1.deposit(10); } };
fork (a1) { synchronized (a1) in { a1.deposit(10); } };

Account⟨thisThread⟩ a2 = new Account⟨thisThread⟩;
a2.deposit(10);
```

Statically verifiable race-free program

a1 is protected by its lock
a2 is thread-local

```
    class Account⟨thisOwner⟩ {
        int balance = 0;
        int deposit(int x) requires (this) {
            this.balance += x;
        }
    }

➡   final Account⟨self⟩ a1 = new Account⟨self⟩;
    fork (a1) { synchronized (a1) in { a1.deposit(10); } };
    fork (a1) { synchronized (a1) in { a1.deposit(10); } };

➡   Account⟨thisThread⟩ a2 = new Account⟨thisThread⟩;
    a2.deposit(10);
```

Statically verifiable race-free program

deposit requires lock on “this”

```
class Account⟨thisOwner⟩ {
   int balance = 0;
   int deposit(int x) requires (this) {
      this.balance += x;
   }
}

final Account⟨self⟩ a1 = new Account⟨self⟩;
fork (a1) { synchronized (a1) in { a1.deposit(10); } };
fork (a1) { synchronized (a1) in { a1.deposit(10); } };

Account⟨thisThread⟩ a2 = new Account⟨thisThread⟩;
a2.deposit(10);
```

Statically verifiable race-free program

a1 is locked before calling deposit
a2 need not be locked

```
   class Account⟨thisOwner⟩ {
      int balance = 0;
      int deposit(int x) requires (this) {
         this.balance += x;
      }
   }

   final Account⟨self⟩ a1 = new Account⟨self⟩;
➡  fork (a1) { synchronized (a1) in { a1.deposit(10); } };
➡  fork (a1) { synchronized (a1) in { a1.deposit(10); } };

   Account⟨thisThread⟩ a2 = new Account⟨thisThread⟩;
➡  a2.deposit(10);
```

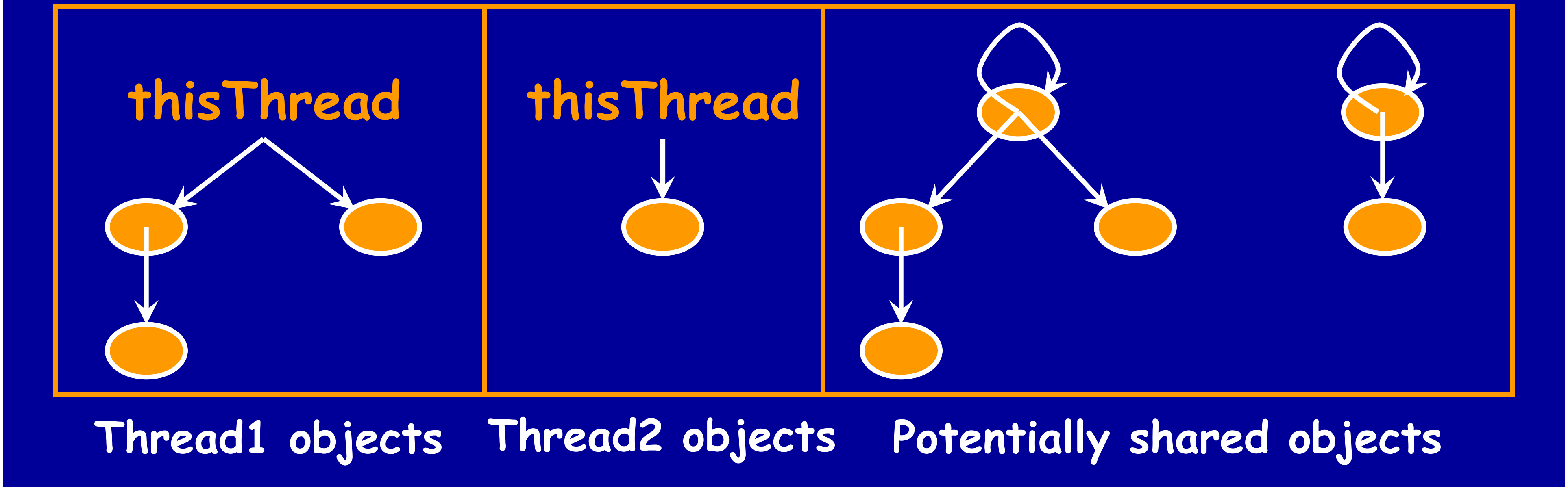
Type system

- Basic type system: Locks, thread-local objects
  - Object ownership
  - Type system
  - Type inference

- Extensions: Unique pointers, read-only objects

Object ownership

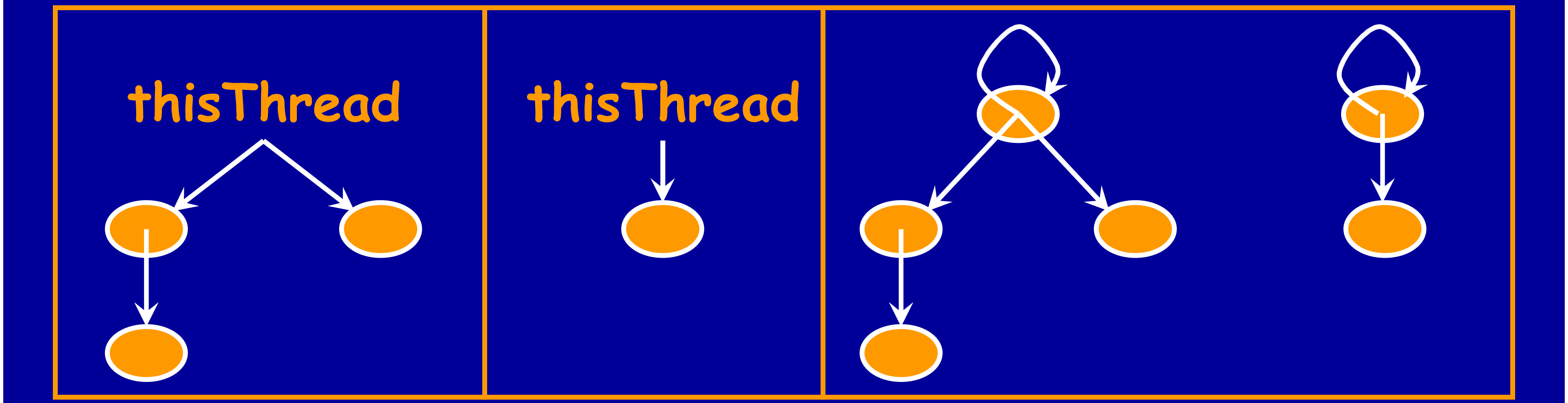
- Every object has an owner
- An object can be owned by
  - Itself
  - Another object
  - Special per-thread owner called thisThread



Ownership properties

- Owner of an object does not change over time

- Ownership relation forms a forest of rooted trees
    - Roots can have self loops



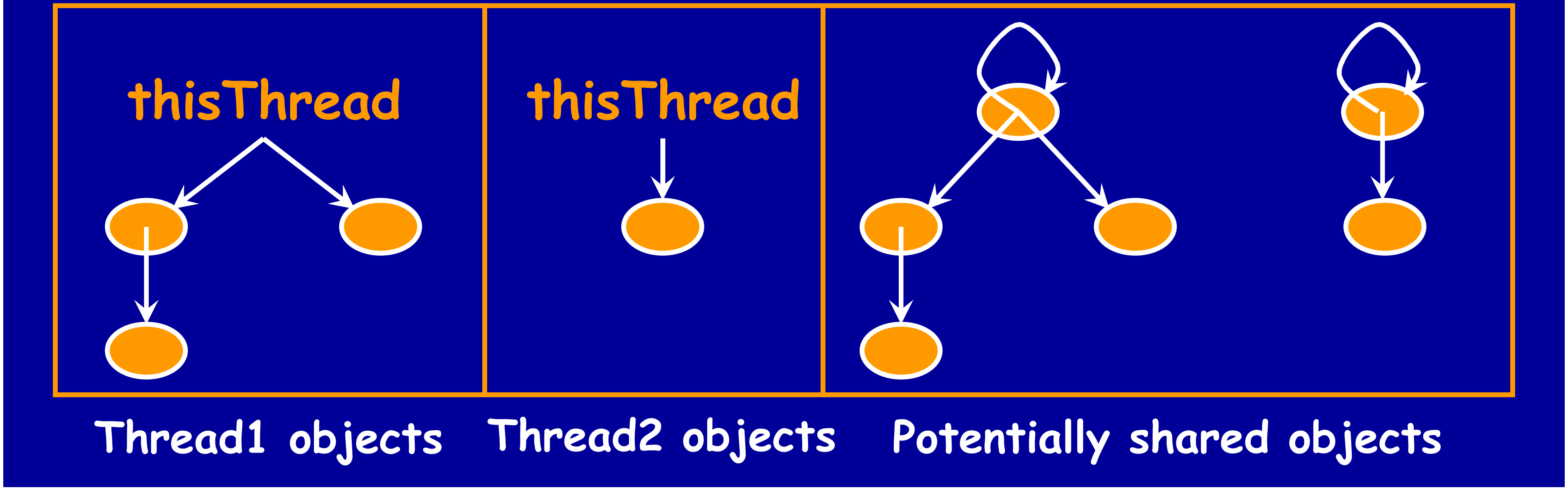
Thread1 objects Thread2 objects Potentially shared objects

Ownership properties

- Every object is protected by its root owner

- To gain exclusive access to an object, it is
  - Necessary and sufficient to lock its root owner

- A thread implicitly holds the lock on its thisThread

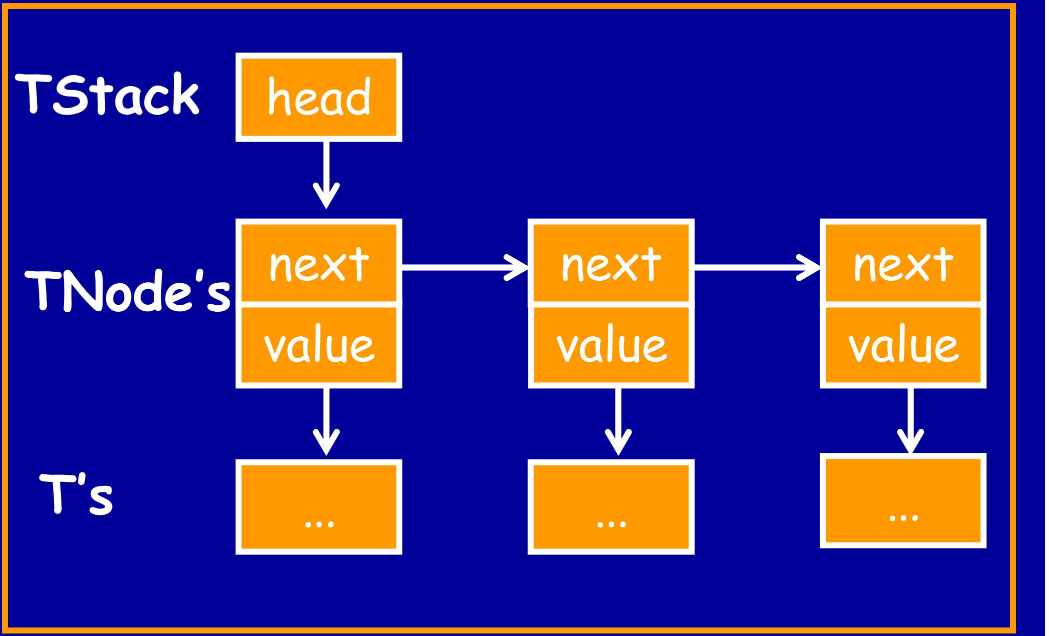


Basic type system

- Object ownership
- Type system
- Type inference

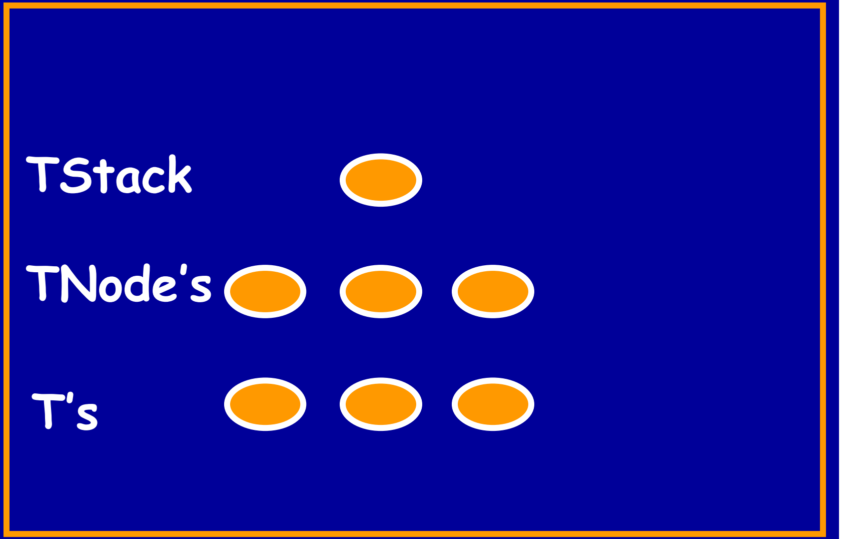
TStack program

```
class TStack {
    TNode head;

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

class TNode {
    TNode next;
    T value;
    …
}

class T {…}
```



TStack program

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



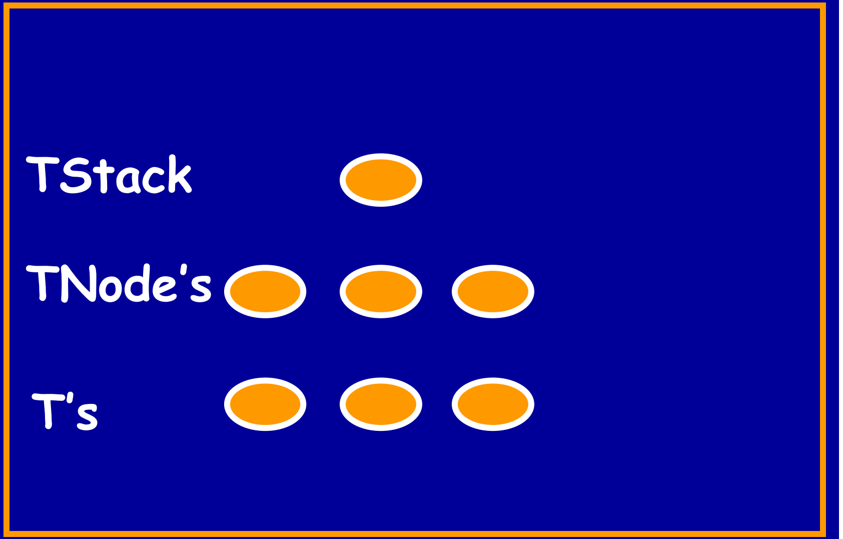
TStack

TNode's

T's

Parameterizing classes

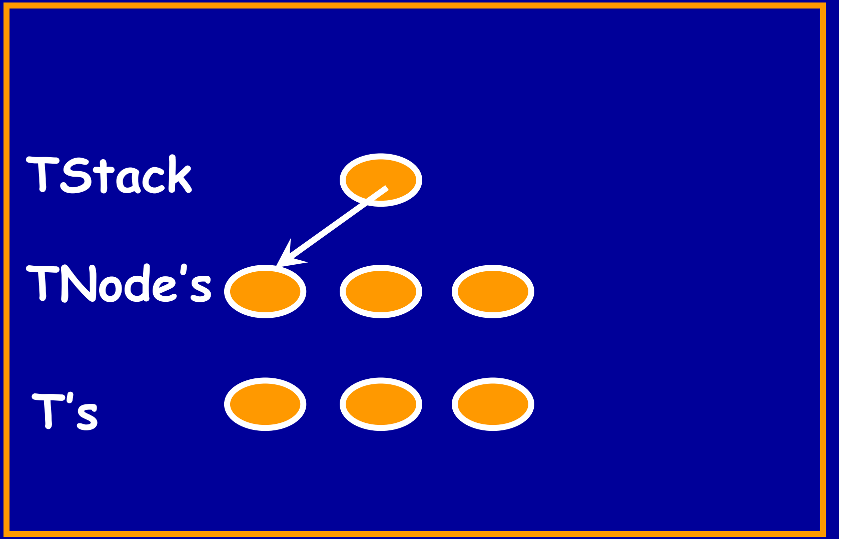
```
class TStack⟨thisOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨thisOwner, TOwner⟩ {
    TNode⟨thisOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
TStack⟨thisThread, thisThread⟩ s1;
TStack⟨thisThread, self⟩ s2;
```



Classes are parameterized with one or more owners
First owner owns the "this" object

Instantiating classes

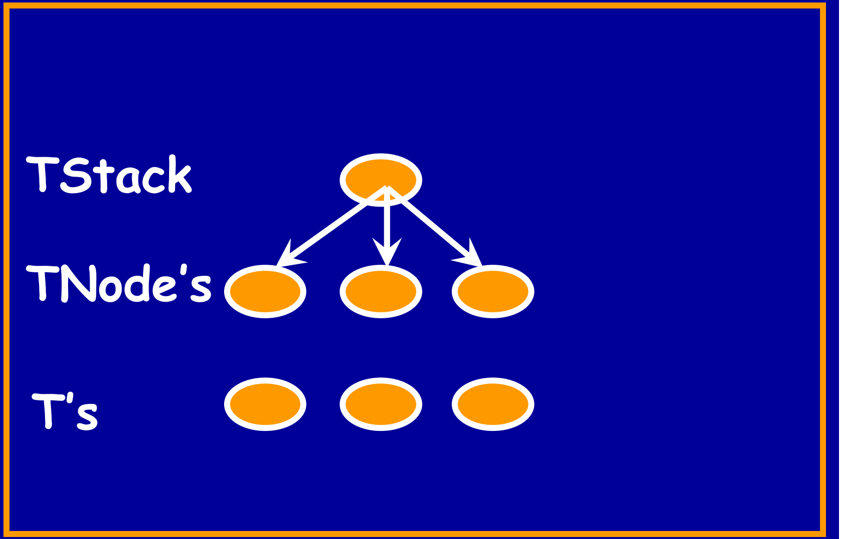
```
class TStack⟨thisOwner, TOwner⟩ {
➡   TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨thisOwner, TOwner⟩ {
    TNode⟨thisOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
TStack⟨thisThread, thisThread⟩ s1;
TStack⟨thisThread, self⟩ s2;
```



Classes can be instantiated with final expressions
E.g., with "this"

Instantiating classes

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



TStack

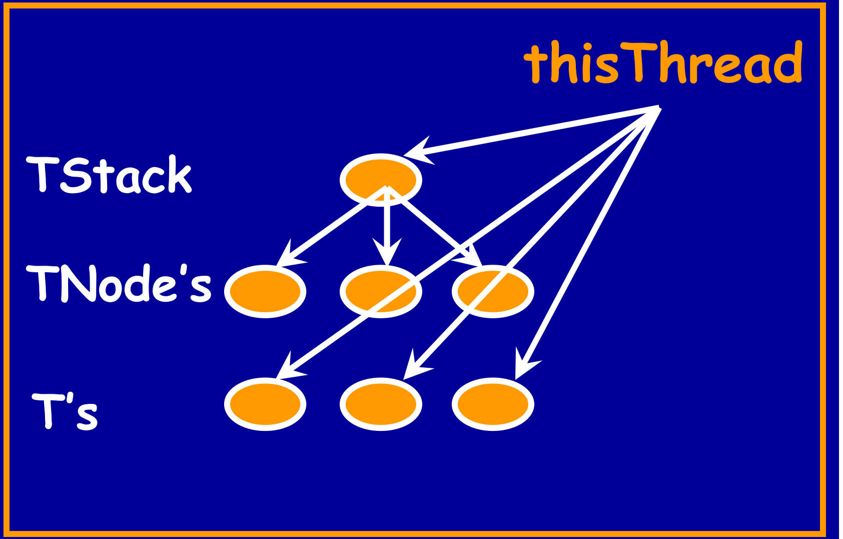
TNode’s

T’s

Classes can be instantiated with formal parameters
E.g., with “thisOwner” or “TOwner”

Instantiating classes

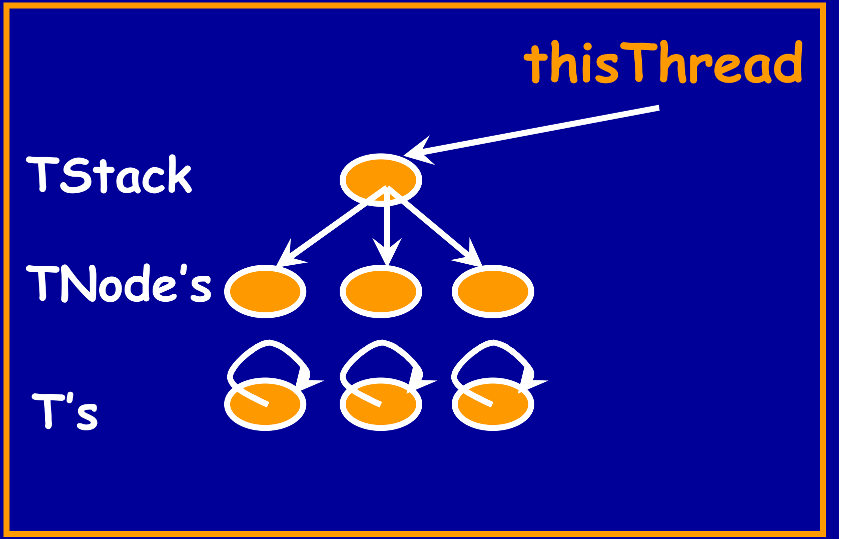
```
class TStack⟨thisOwner, TOwner⟩ {
    TNode⟨this, TOwner⟩ head;
    …
}
class TNode⟨thisOwner, TOwner⟩ {
    TNode⟨thisOwner, TOwner⟩ next;
    T⟨TOwner⟩ value;
    …
}
➡ TStack⟨thisThread, thisThread⟩ s1;
  TStack⟨thisThread, self⟩ s2;
```



Classes can be instantiated with "thisThread"

Instantiating classes

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



Classes can be instantiated with "self"

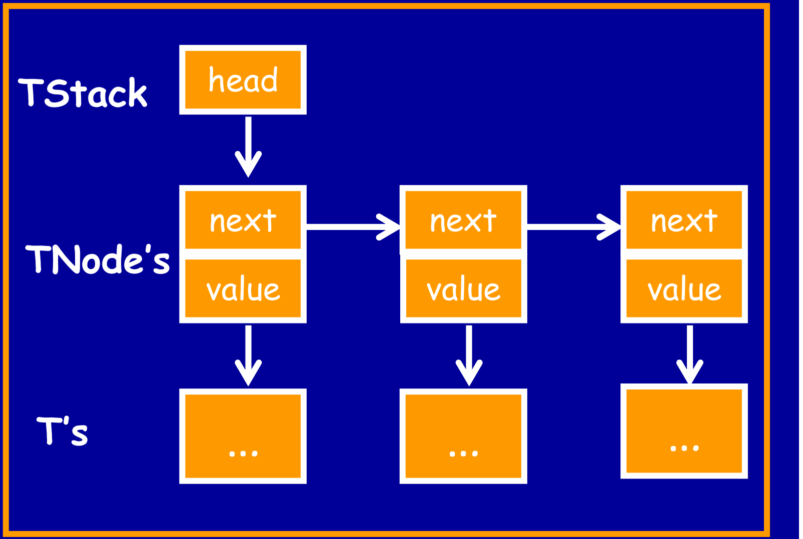
Requires clauses

```
    class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
        T⟨TOwner⟩ value() requires (this) {…}
        TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
        …
    }
```

Methods can require threads to have locks on root owners of objects

Type checking pop method

```
   class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
       T⟨TOwner⟩ value() requires (this) {…}
       TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
       …
   }
```



Type checking pop method

```
   class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
      T⟨TOwner⟩ value() requires (this) {…}
      TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
      …
   }
```

Locks held

thisThread,
RootOwner(this)

Type checking pop method

```
class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
    T⟨TOwner⟩ value() requires (this) {…}
    TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
    …
}
```

Locks held

thisThread,
RootOwner(this)

Locks required

RootOwner(this)

Type checking pop method

```
class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
    T⟨TOwner⟩ value() requires (this) {…}
    TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
    …
}
```

Locks held

thisThread,
RootOwner(this)

Locks required

?

Type checking pop method

```
   class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
➡     T⟨TOwner⟩ value() requires (this) {…}
      TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
      …
   }
```

Locks held

thisThread,
RootOwner(this)

Locks required

RootOwner(head)

Type checking pop method

```
   class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
➡      T⟨TOwner⟩ value() requires (this) {…}
       TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
       …
   }
```

Locks held

thisThread,
RootOwner(this)

Locks required

RootOwner(head)
= RootOwner(this)

Type checking pop method

```
   class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
      T⟨TOwner⟩ value() requires (this) {…}
➡     TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
      …
   }
```

Locks held

thisThread,
RootOwner(this)

Locks required

RootOwner(this),

RootOwner(head)
= RootOwner(this)

Type checking pop method

```
  class TStack⟨thisOwner, 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⟨thisOwner, TOwner⟩ {
      T⟨TOwner⟩ value() requires (this) {…}
      TNode⟨thisOwner, TOwner⟩ next() requires (this) {…}
      …
  }
```

Type checking client code

```
class TStack⟨thisOwner, TOwner⟩ {
    T⟨TOwner⟩ pop() requires (this) {…}
    …
}

final TStack⟨self, self⟩ s = …;

fork (s) {
    synchronized (s) in {
        s.pop();
    }
};
```

Type checking client code

```
class TStack⟨thisOwner, TOwner⟩ {
    T⟨TOwner⟩ pop() requires (this) {…}
    …
}

final TStack⟨self, self⟩ s = …;

fork (s) {
    synchronized (s) in {
        s.pop();
    }
};
```

Locks held

thisThread, s

Type checking client code

```
class TStack⟨thisOwner, TOwner⟩ {
    T⟨TOwner⟩ pop() requires (this) {…}
    …
}

final TStack⟨self, self⟩ s = …;

fork (s) {
    synchronized (s) in {
        s.pop();
    }
};
```

Locks held

thisThread, s

Locks required

RootOwner(s) = s

Basic type system

- Object ownership
- Type system
- Type inference

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**

Default types

- To further reduce programming overhead
- Single threaded programs require almost no programming overhead

Outline

- Motivation
- Type system
- Experience
- Related work
- Conclusions

Multithreaded server programs

Program	Lines of code	Lines changed
http server	563	26
chat server	308	21
stock quote server	242	12
game server	87	10
phone (database) server	302	10

Java libraries

Program	Lines of code	Lines changed
java.util.Vector	992	35
java.util.ArrayList	533	18
java.io.PrintStream	568	14
java.io.FilterOutputStream	148	05
java.io.OutputStream	134	03
java.io.BufferedWriter	253	09
java.io.OutputStreamWriter	266	11
java.io.Writer	177	06

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

Java libraries

- Java programs contain unnecessary locking
- Much analysis work to remove unnecessary locking
  - Aldrich, Chambers, Sirer, Eggers (SAS '99)
  - Whaley, Rinard (OOPSLA '99)
  - Choi, Gupta, Serrano, Sreedhar, Midkiff (OOPSLA '99)
  - Blanchet (OOPSLA '99)
  - Bogda, Holzle (OOPSLA '99)
  - Ruf (PLDI '00)
- Our implementation
  - Avoids unnecessary locking
  - Without sacrificing safety

Additional 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;
```

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

Initially:

x=0;

y=1;

Thread 1:	Thread 2:
y=0;	z=x+y;
x=1;	

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

Additional benefits of race-free types

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

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

Outline

- Motivation
- Type system
- Experience
- Related work
- Conclusions

Tools to detect races

- Static race detection systems
  - Sterling (USENIX '93)
  - Detlefs, Leino, Nelson, Saxe (SRC '98)
  - Engler, Chen, Hallem, Chou, Chelf (SOSP '01)

- Dynamic race detection systems
  - Steele (POPL '90)
  - Dinning, Schonberg (PPoPP '90)
  - Savage, Burrows, Nelson, Sobalvarro, Anderson (SOSP '97)
  - Praun, Gross (OOPSLA '01)

Type systems to prevent races

- Race-free Java
  - Flanagan and Freund (PLDI '00)

- Guava
  - Bacon, Strom, Tarafdar (OOPSLA '00)

Other related type systems

- Ownership types
  - Clarke, Potter, Noble (OOPSLA '98), (ECOOP '01)
- Region types
  - Grossman, Morrisett, Jim, Hicks, Wang, Cheney (Cornell'01)
- Parameterized types for Java
  - Myers, Bank, Liskov (POPL '97)
  - Agesen, Freund, Mitchell (OOPSLA '97)
  - Bracha, Odersky, Stoutamire, Wadler (OOPSLA '98)
  - Cartwright, Steele (OOPSLA '98)

Conclusions

- Data races make programs hard to debug

- We presented race-free static type system

- Our type system is expressive
- Programs can be reliable and efficient

A Parameterized Type System for Race-Free Java Programs

Chandrasekhar Boyapati Martin Rinard

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