Atomicity Analysis of Concurrent Software

Cormac Flanagan
UC Santa Cruz

Stephen N. Freund
Williams College

Shaz Qadeer
Microsoft Research

Types Against Races

Moore’s Law

- Transistors per chip doubles every 18 months
- Single-threaded performance doubles every 2 years
  - faster clocks, deep pipelines, multiple issue
  - wonderful!

Moore’s Law is Over

- Sure, we can pack more transistors ...
  - ... but can’t use them effectively to make single-threaded programs faster
- Multi-core is the future of hardware
- Multi-threading is the future of software

Programming With Threads

- Decompose program into parallel threads
- Advantages
  - exploit multiple cores/processors
  - some threads progress, even if others block
- Increasingly common (Java, C#, GUIs, servers)

![Diagram of network and threads]
Concurrency is a problem
• Windows 2000 hot fixes
  – Concurrency errors most common defects among “detectable errors”
  – Incorrect synchronization and protocol errors most common defects among all coding errors
• Windows Server 2003 late cycle defects
  – Synchronization errors second in the list, next to buffer overruns

Security vulnerabilities involving race conditions
• Buffer overruns
• Phishing attacks

Economic Impact
• NIST study

Last year, a study commissioned by the National Institute of Standards and Technology found that software errors cost the U.S. economy about $39.5 billion annually, or about 1.6 percent of the gross domestic product. More than half the costs are borne by software users, the rest by developers and vendors.


Non-Determinism, Heisenbugs
• Multithreaded programs are non-deterministic
  – behavior depends on interleaving of threads
• Extremely difficult to test
  – exponentially many interleavings
  – during testing, many interleavings behave correctly
  – post-deployment, other interleavings fail
• Complicates code reviews, static analysis, ...

Mars, July 4, 1997
Lost contact due to real-time priority inversion bug

400 horses
100 microprocessors
Bank Account Implementation

```java
class Account {
    private int bal = 0;
    public void deposit(int n) {
        int j = bal;
        bal = j + n;
    }
}
```

Bank Account Implementation

```java
class Account {
    private int bal = 0;
    public void deposit(int n) {
        int j = bal;
        bal = j + n;
    }
}
```

A race condition occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write.

Race Conditions

```java
class Ref {
    int i;
    void add(Ref r) {
        i += r.i;
    }
}
```

```java
Ref x = new Ref(0);
Ref y = new Ref(3);
x.add(y);
x.add(y);
assert x.i == 6;
```

Race Conditions

```java
class Ref {
    int i;
    void add(Ref r) {
        i += r.i;
    }
}
```

```java
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    x.add(y); // two calls happen
    x.add(y); // in parallel
}
assert x.i == 6;
```

Race condition on x.i

Assertion may fail
Lock-Based Synchronization

- Every shared memory location protected by a lock
- Lock must be held before any read or write of that memory location

When Locking Goes Bad ...

- Hesienbugs (race conditions, etc) are common and problematic
  - forget to acquire lock, acquire wrong lock, etc
  - extremely hard to detect and isolate
- Traditional type systems are great for catching certain errors
- Type systems for multithreaded software
  - detect race conditions, atomicity violations, ...

Verifying Race Freedom with Types

```java
class Ref {
    int i;
    // guarded by this
    void add(Ref r) {
        i += r.i;
    }
    Ref x = new Ref(0);
    Ref y = new Ref(3);
    parallel {
        synchronized (x,y) { x.add(y); }
        synchronized (x,y) { x.add(y); }
    }
    assert x.i == 6;
}
```

```java
class Ref {
    int i guarded_by this;
    void add(Ref r) requires this, r {
        i += r.i;
    }
    Ref x = new Ref(0);
    Ref y = new Ref(3);
    parallel {
        synchronized (x,y) { x.add(y); }
        synchronized (x,y) { x.add(y); }
    }
    assert x.i == 6;
}
```

```
class Ref {
    int i guarded_by this;
    void add(Ref r) requires this, r {
        i += r.i;
    }
    Ref x = new Ref(0);
    Ref y = new Ref(3);
    parallel {
        synchronized (x,y) { x.add(y); }
        synchronized (x,y) { x.add(y); }
    }
    assert x.i == 6;
}
```
Verifying Race Freedom with Types

```java
class Ref {
    int i guarded_by this;
    void add(Ref r) requires this, r {
        i = i + r.i;
    }
}

Ref x = new Ref(0);
Ref y = new Ref(3);

parallel {
    synchronized (x,y) {
        x.add(y);
    }
    synchronized (x,y) {
        x.add(y);
    }
}

assert x.i == 6;
```

**Soundness Theorem:**
Well-typed programs are race-free.

---

**One Problem ...**

```java
Object o;
int x guarded_by o;

fork {
    sync(o) { x++; }
}
fork {
    o = new Object();
    sync(o) { x++; }
}
```

- Lock expressions must be constant

---

**Lock Equality**

- Type system checks if lock is in lock set
  - \( r \in \{ \text{this}, r \} \)
  - same as \( r = \text{this} \lor r = r \)

- Semantic equality
  - \( e_1 = e_2 \) if \( e_1 \) and \( e_2 \) refer to same object
  - need to test whether two program expressions evaluate to same value
  - undecidable in general

---

**Lock Equality**

- Approximate (undecidable) semantic equality by syntactic equality
  - two locks expressions are considered equal only if syntactically identical

- Conservative approximation

```java
class A {
    void f() requires this ({ ... })
}

A p = new A();
q = p;
synch(q) { p.f(); }
```

- Not a major source of imprecision

---

**RaceFreeJava**

- Concurrent extension of CLASSICJAVA
  - [Flatt-Krishnamurthi-Felleisen 99]

- Judgement for typing expressions

```
P; E; Is ⊢ e : t
```

- Program
- Environment
- Lock set

---

**Typing Rules**

- Thread creation

```
P; E; \emptyset \vdash e : t
```

- Synchronization

```
P; E; Is \vdash e_1 : c \quad \text{lock is constant}
P; E; Is \cup \{ e_1 \} \vdash e_2 : t \quad \text{add to lock set}
P; E; Is \vdash \text{synchronized} e_1 \text{ in } e_2 : t
```
Field Access

\[ P; E; ls \vdash c : e \quad \text{has class } e \]
\[ P; E \vdash (t /fd \text{ guarded by } l) \in c \quad \text{fd is declared in } c \]
\[ P; E \vdash c[/thin]l \in ls \quad \text{lock } l \text{ is held} \]

```java
package java.util;

public class Vector {
  Object elementData[] /* guarded_by this */;
  int elementCount /* guarded_by this */;
  synchronized int lastIndexOf(Object elem, int n) {
    for (int i = n; i >= 0; i--)
      if (elem.equals(elementData[i]))
        return i;
    return -1;
  }
  int lastIndexOf(Object elem) {
    return lastIndexOf(elem, elementCount - 1);
  }
  synchronized void trimToSize() {
    ...
  }
  synchronized boolean remove(int index) {
    ...
  }
}
```

Validation of rccjava

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (lines)</th>
<th>Number of annotations</th>
<th>Annotation time (hrs)</th>
<th>Races Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hashtable</td>
<td>434</td>
<td>60</td>
<td>0.5</td>
<td>0</td>
</tr>
<tr>
<td>Vector</td>
<td>440</td>
<td>10</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>java.io</td>
<td>16,000</td>
<td>139</td>
<td>16.0</td>
<td>4</td>
</tr>
<tr>
<td>Ambit</td>
<td>4,500</td>
<td>38</td>
<td>4.0</td>
<td>4</td>
</tr>
<tr>
<td>WebL</td>
<td>20,000</td>
<td>358</td>
<td>12.0</td>
<td>5</td>
</tr>
</tbody>
</table>

Basic Type Inference

```java
class Ref {
  int i;
  void add(Ref r) {
    i = i + r.i;
  }
}
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
  synchronized (x:y) { x.add(y); }
  synchronized (x:y) { x.add(y); }
}
assert x.i == 6;
```

Basic Type Inference

```java
static final Object m = new Object();
```

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE’01]
- Start with maximum set of annotations
Basic Type Inference

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

Basic Type Inference

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations

Basic Type Inference

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations
- Iteratively remove all incorrect annotations

Basic Type Inference

Iterative GFP algorithm:
- [Flanagan-Freund, PASTE'01]
- Start with maximum set of annotations
- Iteratively remove all incorrect annotations
- Check each field still has a protecting lock

Sound, complete, fast
But type system too basic

Harder Example: External Locking

- Field i of x and y protected by external lock m
- Not typable with basic type system
  - m not in scope at i
- Requires more expressive type system with ghost parameters

Ghost Parameters on Classes

class Ref {  
  int i;  
  void add(Ref r) {  
    i = i + x;  
  }  
}  
Object m = new Object();  
Ref x = new Ref();  
Ref y = new Ref();  
parallel {  
  synchronized (m) {  
    synchronized (m) {  
      assert x == y;  
    }  
  }  
}
Ghost Parameters on Classes

class Ref<ghost g> {
    int i;
    void add(Ref r) {
        i = i + r.i;
    }
}
Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    synchronized (m) { x.add(y); }
    synchronized (m) { x.add(y); }
}
assert x.i == 6;

- Ref parameterized by
e external ghost lock g
- Field i guarded by g
- g held when add called

Type Checking Ghost Parameters

class Ref<ghost g> {
    int i guarded by g;
    void add(Ref r) {
        i = i + r.i;
    }
}
Object m = new Object();
Ref x = new Ref(0);
Ref y = new Ref(3);
parallel {
    synchronized (m) { x.add(y); }
    synchronized (m) { x.add(y); }
}
assert x.i == 6;

- Ref parameterized by
e external ghost lock g
- Field i guarded by g
- g held when add called
- Argument r also parameterized by g
- x and y parameterized by
- lock m

check: (g); (this=x,r→y, g=m)∈ (m)
Type Inference with Ghosts

- Type inference is NP-complete
  - iterative GFP algorithm does not work
  - ghost parameters require backtracking search
- RccSAT: Reduce to SAT
  - works up to 30 KLOC
  - precise: 92-100% of fields verified race free