Analysis of Concurrent Software

Types for Race Freedom

Cormac Flanagan
UC Santa Cruz

Stephen N. Freund
Williams College

Shaz Qadeer
Microsoft Research

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)

More BSOD
Embarrassments
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 0.6 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, ...

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;
    }
}
```

**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;
```

**Lock-Based Synchronization**

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

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

Verifying Race Freedom with Types

```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;
```

Verifying Race Freedom with Types

```java
check:   this ∈ { this, r }
check:   this[this:=r] = r ∈ { this, r }
check:   {this,r}[this:=x,r:=y] ⊆ { x, y }
```

Soundness Theorem:
Well-typed programs are race-free
One Problem ...

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 (Halting Problem)

Lock Equality

• Approximate (undecidable) semantic equality by syntactic equality
  - two locks exprs are considered equal only if syntactically identical
• Conservative approximation
  
```java
class A {
    void f() requires this { ... }
}
A p = new A();
p = p;
synch(q) { p.f(); } this[this:=p] = p \in \{q\}
```

• Not a major source of imprecision

RaceFreeJava

• Concurrent extension of CLASSICJAVA
  [Flatt–Krishnamurthi–Felleisen 99]
• Judgement for typing expressions

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

Typing Rules

• Thread creation
  
```
P; E; \emptyset \vdash e : t
\hline
P; E; ls \vdash \text{fork } e : \text{int}
```

• Synchronization
  
```
P; E; \emptyset \vdash e_1 : c \quad \text{lock is constant}
P; E; ls \cup \{e_1\} \vdash e_2 : t \quad \text{add to lock set}
P; E; ls \vdash \text{synchronized } e_1 \text{ in } e_2 : t
```

Field Access

```
P; E; ls \vdash e : c
P; E \vdash (l \text{ fd guarded_by } l) \in c
P; E \vdash [e/\text{this}]l \in ls
P; E; ls \vdash c, \text{fd} : [e/\text{this}]l
```

java.util.Vector

```
0 1 2
2 a b
```

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

```
class Ref {
    int i guarded by this, m;
    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;
```

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

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

```
class Ref {
    int i,
    void add(Ref r) {
        j = 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;
```

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

```java
static final Object m = new Object();
class Ref {
    int i guarded_by this, m;
    void add(Ref r) requires this, r, m {
        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;
```

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

### Harder Example: External Locking

```java
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;
```

- 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

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

- Ref parameterized by external ghost lock g

Sound, complete, fast
But type system too basic
Ghost Parameters on Classes

```java
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 external ghost lock g
- Field i guarded by g

```
check: {g} [this=x,r=y, g=m] ⊑ [m] ✓
```

Type Checking Ghost Parameters

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

- Ref parameterized by external ghost lock g
- Field i guarded by g
- g held when add called
- Argument r also parameterized by g

Type Inference with Ghosts

- HARD
  - iterative GFP algorithm does not work
  - check may fail because of two annotations
    - which should we remove?
  - requires backtracking search
Type Inference with Ghosts

\[
\begin{align*}
\text{class } & \text{A} \\
& \{ \\
& \text{int } f; \\
& \} \\
\text{class } & \text{B}\text{g}\text{host } y; \\
\text{A a } = & \ldots; \\
\end{align*}
\]

\[
\begin{align*}
\text{class } & \text{A}\text{g}\text{host } g; \\
& \{ \\
& \text{int } f \text{ guarded by } g; \\
& \} \\
\text{class } & \text{B}\text{g}\text{host } y; \\
\text{A a } = & \ldots; \\
\end{align*}
\]

Boolean Satisfiability

\[
\begin{align*}
(t_1 \lor t_2 \lor t_3) & \land (t_2 \lor \neg t_1 \lor \neg t_4) \\
& \land (t_2 \lor \neg t_3 \lor t_4) \\
\end{align*}
\]

Reducing SAT to Type Inference

\[
\begin{align*}
\text{class } & \text{A}\text{g}\text{host } x, y, z; \\
& \ldots \\
\text{class } & \text{C}; \\
\text{A a } = & \ldots; \\
\text{B b } = & \ldots; \\
\text{C c } = & \ldots; \\
\end{align*}
\]

\[
\begin{align*}
\text{class } & \text{A}\text{g}\text{host } x, y, z; \\
& \ldots \\
\text{class } & \text{C}\text{g}\text{host } x, y, z; \\
\text{A p1, p2, p3 } a = & \ldots; \\
\text{B p1, n1, n4 } b = & \ldots; \\
\text{C p2, n3, p4 } c = & \ldots; \\
\end{align*}
\]

Rcc/Sat Type Inference Tool

\[
\begin{align*}
\text{class } & \text{A} \\
& \{ \\
& \text{int } f; \\
& \} \\
\text{class } & \text{A}\text{g}\text{host } g; \\
& \{ \\
& \text{int } f \text{ guarded by } g; \\
& \} \\
\text{A a } = & \ldots; \\
\text{B b } = & \ldots; \\
\text{C c } = & \ldots; \\
\end{align*}
\]

Reducing Type Inference to SAT

\[
\begin{align*}
\text{class } & \text{Ref} \\
& \{ \\
& \text{int } i; \\
& \text{void } \text{add}(\text{Ref } r) \\
& \{ \\
& \text{=} i; \\
& \} \\
& \} \\
\end{align*}
\]

\[
\begin{align*}
\text{class } & \text{Ref}\text{g}\text{host } p_1, p_2, \ldots, p_n \\
& \{ \\
& \text{int } i; \\
& \text{void } \text{add}(\text{Ref } r) \\
& \{ \\
& \text{=} i; \\
& \} \\
& \} \\
\end{align*}
\]
Reducing Type Inference to SAT

- Add ghost parameters `<ghost g>` to each class declaration
- Add guarded_by `g` to each field declaration
- Type inference resolves `g` to some lock
- Add `<g>` to each class reference

Constraints:

1. `c{ this, g }
2. `c{ this, g }
3. `{ b3 ? this, b4 ? g, b5 ? r }

Use boolean variables `b1,...,b5` to encode choices for `a1, a2, β`
Reducing Type Inference to SAT

Overview of Type Inference

Add unknowns.

Error: potential race on field i

Constraint Solution:

Satisfiable

Satisfiable

Unsatisfiable

Chaff

Constraint

SAT problem:

SAT solver

Annotated Program:

Add annotations to fields

Annotated Program:

Add unknown field i

Unannotated Program:
### Performance

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LOC)</th>
<th>Time (s)</th>
<th>Time/Field (s)</th>
<th>Number Constraints</th>
<th>Formula Vars</th>
<th>FormulaClauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>529</td>
<td>5.0</td>
<td>0.22</td>
<td>215</td>
<td>1,449</td>
<td>3,831</td>
</tr>
<tr>
<td>tsp</td>
<td>723</td>
<td>6.9</td>
<td>0.19</td>
<td>233</td>
<td>2,090</td>
<td>7,151</td>
</tr>
<tr>
<td>sor</td>
<td>687</td>
<td>4.5</td>
<td>0.15</td>
<td>130</td>
<td>562</td>
<td>1,205</td>
</tr>
<tr>
<td>raytracer</td>
<td>1,982</td>
<td>21.0</td>
<td>0.27</td>
<td>801</td>
<td>9,456</td>
<td>29,841</td>
</tr>
<tr>
<td>moldyn</td>
<td>1,408</td>
<td>12.6</td>
<td>0.12</td>
<td>904</td>
<td>4,011</td>
<td>10,036</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3,674</td>
<td>20.7</td>
<td>0.19</td>
<td>1,097</td>
<td>9,003</td>
<td>25,974</td>
</tr>
<tr>
<td>mpir</td>
<td>11,315</td>
<td>138.8</td>
<td>1.5</td>
<td>5,636</td>
<td>38,025</td>
<td>123,046</td>
</tr>
<tr>
<td>jpp</td>
<td>30,519</td>
<td>2,773.5</td>
<td>3.52</td>
<td>11,698</td>
<td>146,290</td>
<td>549,667</td>
</tr>
</tbody>
</table>

- Inferred protecting lock for 92–100% of fields
- Used preliminary read-only and escape analyses

### Complexity of Restricted Cases

<table>
<thead>
<tr>
<th># Params:</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>O(2^3)</td>
</tr>
<tr>
<td>2</td>
<td>O(2^2)</td>
</tr>
<tr>
<td>1</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>0</td>
<td>O(n)</td>
</tr>
<tr>
<td></td>
<td>O(n log n)</td>
</tr>
</tbody>
</table>

### Reducing SAT to Type Inference

- Formula
  - (t1 ∨ t2 ∨ t3) ∧ (t2 ∨ ¬t1 ∨ ¬t4) ∧ (t2 ∨ ¬t3 ∨ t4)

- Construct Program From Formula
- Construct Assignment From Annotations

### Summary

- Multithreaded heisenbugs notorious
- Rccjava
  - type system for race freedom
- Type inference is NP-complete
  - ghost parameters require backtracking search
  - Reduce to SAT
    - adequately fast up to 30,000 LOC
    - precise: 92–100% of fields verified race free

### Improved Error Reporting

```java
class Safeghost {  
    int c guarded_by a;  
    void f1() requires y { c = 1; }  
    void f2() requires y { c = 2; }  
    void f3() requires this { c = 3; }  
}  ```
Improved Error Reporting

```java
class Ref<ghost y> {
    int c guarded_by y;
    void f1() requires y { c = 1; }
    void f2() requires y { c = 2; }
    void f3() requires this { c = 3; }
}
```

Possible Error Messages:

- `n = y`: Lock 'y' not held on access to 'c' in f3().
- `a = this`: Lock 'this' not held on access to 'c' in f1() & f2().
- `a = no_lock`: No consistent lock guarding 'c'.

Weighted Constraints

```java
class Ref<ghost y> {
    int c guarded_by y;
    void f1() requires y { c = 1; }
    void f2() requires y { c = 2; }
    void f3() requires this { c = 3; }
}
```

• Find solution that:
  - satisfies all un-weighted constraints, and
  - maximizes weighted sum of satisfiable weighted constraints
### Weighted Constraints

```cpp
class Ref<ghost y> {
  int c guarded_by y;
  void f1() requires y { c = 1; }
  void f2() requires y { c = 2; }
  void f3() requires y { c = 3; }
  void f4() requires this { c = 1; }
  void f5() requires this { c = 2; }
  void f6() requires this { c = 3; }
}
```

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>y[.]c</td>
<td>2</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
<tr>
<td>y[.]c</td>
<td>1</td>
</tr>
</tbody>
</table>

Solution: `=no_lock:` No consistent lock guarding `c`.

### Implementation

- Translate weighted constraints into a MAX-SAT problem
  - Example:
    
    ```
    (t1 ∨ t2 ∨ t3) 2
    (t2 ∨ ¬t1 ∨ ¬t4) 1
    (t2 ∨ ¬t3 ∨ t4) 1
    (t5 ∨ ¬t1 ∨ ¬t6)
    (t2 ∨ ¬t4 ∨ ¬t5)
    ```

  - Find solution with PBS [Aloul et al 02]

- Typical weights:
  - Field access: 1
  - Declaration: 2-4

  - MAX-SAT intractable if more than ~100 weighted clauses
  - Check one field at a time (compose results)
  - Only put weights on field constraints

### Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - Other applications:
    - Type systems [Flanagan-Qadeer 03, Flanagan-Freund-Qadeer 04]
    - Model checking [Stoller-Cohen 03, Flanagan-Qadeer 03]
    - Dynamic analysis [Flanagan-Freund 04, Wang-Stoller 04]

- Atomicity inference
  - Type and effect inference [Talpin-Jouvelot 92, ...]
  - Dependent types [Cardelli 88]
  - Ownership, dynamic [Sastakur-Agarwal-Stoller 04]