Types for Atomicity

Race Conditions

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

Race Conditions

class Ref {
    int i;
    void inc() {
        int t = i;
        i = t + 1;
    }
}

Ref x = new Ref(0);
parallel {
    x.inc(); // two calls happen
    x.inc(); // in parallel
}
assert x.i == 2;

Lock-Based Synchronization

class Ref {
    int i; // guarded by this
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t + 1;
        }
    }
}

Ref x = new Ref(0);
parallel {
    x.inc(); // two calls happen
    x.inc(); // in parallel
}
assert x.i == 2;

Limitations of Race-Freedom

class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t + 1;
        }
    }
    synchronized void read() { return i; }
}

Ref.inc()
- race-free
- behaves correctly in a multithreaded context

Limitations of Race-Freedom

class Ref {
    int i;
    void inc() {
        int t;
        synchronized (this) {
            t = i;
            i = t + 1;
        }
    }
    synchronized void read() { return i; }
}

Ref.inc()
- race-free
- behaves incorrectly in a multithreaded context

Limitations of Race-Freedom

Race freedom does not prevent errors due to unexpected interactions between threads

Analysis of Concurrent Software
Types for Atomicity

Cormac Flanagan
UC Santa Cruz
Stephen N. Freund
Williams College
Shaz Qadeer
Microsoft Research
### Limitations of Race–Freedom

```java
class Ref {
    int i;
    void inc() {
        synchronized (this) {
            int t = i;
            i = t + 1;
        }
        void read() { return i; }
    }
}
```

**Ref.read()**
- has a race condition
- behaves correctly in a multithreaded context

**Race freedom is not necessary to prevent errors due to unexpected interactions between threads**

### Race–Freedom

- Race–freedom is neither necessary nor sufficient to ensure the absence of errors due to unexpected interactions between threads

### Atomicity

- The method `inc()` is atomic if concurrent threads do not interfere with its behavior
- Guarantees that for every execution
  
  ![Diagram](acq(this) t=i i=t+1 rel(this))
  
  there is a *serial* execution with same behavior
  
  ![Diagram](acq(this) x y t=i i=t+1 z rel(this))

### Motivations for Atomicity

1. Stronger property than race freedom
2. Enables sequential reasoning

### Sequential Program Execution

```java
void inc() {
    ...
    ...
    ...
}
```
Multithreaded Execution

void inc() {
  ...
  ...
}

Motivations for Atomicity
1. Stronger property than race freedom
2. Enables sequential reasoning
3. Simple, powerful correctness property

Model Checking of Software Models

Specification for filesystem.c

Model Checker

Model Construction

Model Checking of Software Models

Specification for filesystem.c

Model Checker

Model Construction
Experience with Calvin Software Checker

Sequential case: code inspection & testing mostly ok

The Need for Atomicity

1. Stronger property than race freedom
2. Enables sequential reasoning
3. Simple, powerful correctness property
Atomicity

- Canonical property
  - (cmp. serializability, linearizability, ...)
- Enables sequential reasoning
  - simplifies validation of multithreaded code
- Matches practice in existing code
  - most methods (80%+) are atomic
  - many interfaces described as “thread-safe”
- Can verify atomicity statically or dynamically
  - atomicity violations often indicate errors
  - leverages Lipton’s theory of reduction

Reduction [Lipton 75]

Movers

- right-mover
  - lock acquire

Movers

- right-mover
  - lock acquire
- left-mover
  - lock release

Movers

- right-mover
  - lock acquire
- left-mover
  - lock acquire
- both-mover
  - race-free field access
- non-mover (atomic)
  - access to “racy” fields
Code Classification

<table>
<thead>
<tr>
<th>Right</th>
<th>Left</th>
<th>Mover</th>
<th>Atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock acquire</td>
<td>Lock release</td>
<td>Race-free variable</td>
<td>Conflicting variable</td>
</tr>
</tbody>
</table>

Composition rules:
- Right, mover = right; Right, left = atomic
- Right, atomic = atomic; Atomic, atomic = cmpd

Conditional Atomicity

```java
atomic void deposit(int n) {
    synchronized(this) { j = bal; }
    synchronized(this) { bal = j + n; }
}
```

Conditional Atomicity Details

- In conditional atomicity (x?b_1:b_2), x must be a lock expression (ie, constant)
- Composition rules
  \[ a : (x?b_1:b_2) = x ? (a;b_1) : (a;b_2) \]
java.lang.StringBuffer

/**
 * ... used by the compiler to implement the binary string concatenation operator ...
 */

public atomic class StringBuffer {...}

java.lang.StringBuffer is not Atomic!

public atomic StringBuffer {
    private int count guarded_by this;
    public synchronized int length() { return count; }
    atomic void getChars(...) {...}
    atomic void append(StringBuffer sb) {
        int len = sb.length();
        sb.length() acquires the lock on sb,
        gets the length, and releases lock
        use of stale len may yield
        StringIndexOutOfBoundsException
        inside getChars(...)
    }
    append(...) is not atomic
}

java.lang.Vector

interface Collection {
    atomic int length();
    atomic void toArray(Object a[]);
}

class Vector {
    int count;
    Object data[];
    atomic Vector(Collection c) {
        count = c.length();
        data = new Object[count];
        mover
        c.toArray(data);
    }
}

Bohr

- Type inference for atomicity
  - finds smallest atomicity for each method

Atomicity Inference

Program w/ Locking Annotations

```java
atomic class A<ghost x> {
    int f guarded_by this;
    int g guarded_by x;
    void m() {...}
}
```

Program w/ Atomicity Annotations

```java
atomic class A<ghost x> {
    int f guarded_by this;
    int g guarded_by x;
    atomic void m() {...}
}
```
Partial order of atomicities

- error
- compound
- atomic
- mover
- x?mover:atomic
- const

1. Add atomicity variables

```java
class Account {
    int bal guarded_by this;

    α
    1 void deposit(int n) {
        synchronized(this) {
            int j = this.bal;
            j = this.bal + n;
        }
    }
}
class Bank {

    α
    2 void double(final Account c) {
        synchronized(c) {
            int x = c.bal;
            c.deposit(x);
        }
    }
}
```

2. Generate constraints over atomicity variables

```java
S(this, ((const; this?mover:error); (const; this?mover:error)))
```

3. Find assignment A

```java
S(this, ((const; this?mover:error); (const; this?mover:error)))
```

Atomicity expression

- `s ::= const | mover | atomic | cmpd | error
- `| α
- `| S1; S2
- `| x? S1; S2
- `| S(l,s)
- `| WFA(E, s)

```

S(l,a): atomicity of synchronized(l) { a }
where a has atomicity α
S(l, mover) = l ? mover : atomic
S(l, atomic) = atomic
S(l, compound) = compound
S(l, l(b1,b2)) = S(l,b2)
S(l, m(b1,b2)) = m ? S(l,b2) : S(l,b1) if l ≠ m
class Account {
    int bal guarded_by this;
    α
    void deposit(int n) {
        synchronized(this) {
            int j = this.bal;
            j = this.bal + n;
        }
    }
}

class Bank {
    α
    void double(final Account c) {
        synchronized(c) {
            int x = c.bal;
            c.deposit(x);
        }
    }
}

Delayed Substitutions

- Given α[x := e]
  - suppose e becomes x?mover:atomic
  - and e does not have const atomicity
  - then (e?mover:atomic) is not valid
- WFA(E, b) = smallest atomicity b' where
  - b ≤ b'
  - b' is well-typed and constant in E
- WFA(E, (e?mover:atomic)) = atomic
3. Compute Least Fixed Point

- Initial assignment $A$: $\alpha_1 = \alpha_2 = \text{const}$

- Algorithm:
  - pick constraint $s \leq \alpha$ such that $A(s) \not< A(\alpha)$
  - set $A(\alpha)$ to $A(\alpha) \cup A(s)$
  - repeat until quiescence

Validation

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (KLOC)</th>
<th>Time (s)</th>
<th>Time (s/KLOC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>0.5</td>
<td>0.6</td>
<td>1.1</td>
</tr>
<tr>
<td>tsp</td>
<td>0.7</td>
<td>1.4</td>
<td>2.0</td>
</tr>
<tr>
<td>sor</td>
<td>0.7</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>raytracer</td>
<td>2.0</td>
<td>1.7</td>
<td>0.9</td>
</tr>
<tr>
<td>moldyn</td>
<td>1.4</td>
<td>4.9</td>
<td>3.5</td>
</tr>
<tr>
<td>montecarlo</td>
<td>3.7</td>
<td>1.5</td>
<td>0.4</td>
</tr>
<tr>
<td>mtrt</td>
<td>11.3</td>
<td>7.8</td>
<td>0.7</td>
</tr>
<tr>
<td>jbb</td>
<td>30.5</td>
<td>11.2</td>
<td>0.4</td>
</tr>
</tbody>
</table>

(excludes Rcc/Sat time)

Inferred Atomicities

<table>
<thead>
<tr>
<th>Program</th>
<th>% Atomic</th>
<th>Methods</th>
<th>Synch Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>elevator</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>tsp</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>sor</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>raytracer</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>moldyn</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>montecarlo</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>mtrt</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>jbb</td>
<td>60%</td>
<td>80%</td>
<td>20%</td>
</tr>
</tbody>
</table>

Thread-Safe Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>% Atomic</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>80%</td>
</tr>
<tr>
<td>StringBuffer</td>
<td>80%</td>
</tr>
<tr>
<td>Vector</td>
<td>80%</td>
</tr>
<tr>
<td>Inflater</td>
<td>80%</td>
</tr>
<tr>
<td>Deflater</td>
<td>80%</td>
</tr>
<tr>
<td>ZipFile</td>
<td>80%</td>
</tr>
<tr>
<td>ObservableSynchList</td>
<td>80%</td>
</tr>
<tr>
<td>URL</td>
<td>80%</td>
</tr>
<tr>
<td>PrintWriter</td>
<td>80%</td>
</tr>
<tr>
<td>Boolean</td>
<td>80%</td>
</tr>
<tr>
<td>Double</td>
<td>80%</td>
</tr>
</tbody>
</table>

Related Work

- Reduction
  - [Lipton 75, Lamport-Schneider 89, ...]
  - other applications:
    - 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]
Conclusions And Future Directions

- Atomicity a fundamental concept
  - improves over race freedom
  - matches programmer intuition and practice
  - simplifies reasoning about correctness
  - enables concise and trustable documentation

- Many approaches for verifying atomicity
  - static type systems
  - dynamic checking (tomorrow)
  - ... hybrid checkers ...
  - ... model checkers ...