A Type and Effect System for Atomicity

Corman Flanagan and Shaz Qadeer (PLDI ’03)

Presented by: Jonathan Deber
CSC2125F - October 30, 2008
Motivation
Motivation

• Concurrency is hard
Motivation

- Concurrency is hard
- Type checking is less hard
Motivation

- Concurrency is hard
- Type checking is less hard

Therefore, look at concurrency as a type checking problem:

1. Develop type-theoretic model of concurrency
2. Determine necessary language changes (to Java)
3. Develop static checking tools
String s = "foo";
...
int i = s++;

s cannot be cast to an int
Goal

String s = "foo";
... int i = s++;

int x guarded_by lock = 0;
...
... int x guarded_by lock = 0;
... x++;
Previous Work
Previous Work


Previous Work


Type Checking

1. Developer specifies type metadata
   - e.g., int i;

2. Type checker analyzes program, generates inferred types
   - e.g., i = i + 2;

3. Type checker compares declared type and inferred type
Race Conditions

• Previous work looked at race conditions
  • e.g., rccjava

• Formally:
  1. Two threads simultaneously access same variable
  2. At least one access is a write
Race Conditions

"However, verifying the absence of such simultaneous-access race conditions is neither necessary nor sufficient to ensure the absence of errors due to unexpected thread interactions."

```java
int x; // shared var guarded by lock l
void m() {
    int t;
    synchronized (l) { t = x; }
    t++;
    synchronized (l) { x = t; }
}
```
Atomicity

• “[A] stronger non-interference property...”
• If a method is atomic, interactions with other threads is benign (a.k.a “thread safe”)
• Can therefore reason at higher level of granularity
• Developers specify atomicity, system checks
Movers

• Based on Lipton’s concept of movers


• *Left movers* and *right movers*
Movers

- Thread A, action $a$
- Thread B, action $b$
- States $s_1$, $s_2$, and $s_3$

\[ s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3 \]
Movers

• Thread A, action \( a \)
• Thread B, action \( b \)
• States \( s_1, s_2, \) and \( s_3 \)

Adapted from: p. 3
Which Way Do They Move?

• Let’s say \texttt{a} is \texttt{acquire(lock)}, \texttt{b} anything else

• Which way can \texttt{a} move?
Which Way Do They Move?

- Let’s say a is acquire(lock), b anything else
- Which way can a move?
- acquire(lock) is a right mover
Which Way Do They Move?

• Let’s say a is acquire(lock), b anything else
• Which way can a move?

• acquire(lock) is a right mover
• release(lock) is a left mover
Which Way Do They Move?

- Let’s say $a$ is `acquire(lock)`, $b$ anything else
- Which way can $a$ move?
  - `acquire(lock)` is a *right mover*
  - `release(lock)` is a *left mover*
- Accessing a variable protected by a lock is both
void f() {
  int t;
  acq(lock);
  t = x;
  x = t + 1;
  rel(lock);
}

Adapted from: p. 3
void f() {
    int t;
    acq(lock);
    t = x;
    x = t + 1;
    rel(lock);
}

Adapted from: p. 3
void f() {
    int t;
    acq(lock);
    t = x;
    x = t + 1;
    rel(lock);
}

Adapted from: p. 3
void f() {
    int t;
    acq(lock);
    t = x;
    x = t + 1;
    rel(lock);
}

Adapted from: p. 3
Atomicities

- const
- mover
- atomic
- cmpd
- error
Notation

- \( e \) used for expressions; \( \alpha, a, b \) used for atomicities
- \( \alpha^* \) is iterative closure
- \( \alpha_1; \alpha_2 \) is sequential composition
- \( \alpha_1 \sqcup \alpha_2 \) is join (non-deterministic choice)
- \( \text{const} \sqsubseteq \text{mover} \sqsubseteq \text{atomic} \sqsubseteq \text{cmpd} \sqsubseteq \text{error} \)
- \( l ? a:b \) is conditional atomicity
- \( l ? a \) is equivalent to \( l ? a:\text{error} \)
More Notation

\[
[\alpha](ls) = \alpha
\]

\[
[l ? a_1 : a_2](ls) = \begin{cases} 
[\alpha_1](ls) & \text{if } l \in ls \\
[\alpha_2](ls) & \text{if } l \notin ls
\end{cases}
\]

\[
a \equiv b, \text{ if } a \subseteq b \text{ and } b \subseteq a.
\]
Axioms and Theorems

\[(l ? a : b)^* = l ? a^* : b^*\]
\[(l ? a_1 : a_2); b = l ? (a_1; b) : (a_2; b)\]
\[\alpha; (l ? b_1 : b_2) = l ? (\alpha; b_1) : (\alpha; b_2)\]
\[(l ? a_1 : a_2) \sqcap b = l ? (a_1 \sqcap b) : (a_2 \sqcap b)\]
\[\alpha \sqcap (l ? a_1 : a_2) = l ? (\alpha \sqcap a_1) : (\alpha \sqcap a_2)\]

“Iterative closure is monotonic and idempotent.”

“Sequential composition is monotonic and associative and \texttt{const} is a left and right identity of this operation.”

\textit{etc. etc.}
AtomicJava

- Small addition to CONCURRENTJAVA used to illustrate type rules

\[
\begin{align*}
\text{field} & ::= \text{[final]}_{\text{opt}} t f d [g]_{\text{opt}} = e \quad \text{(fields)} \\
\text{meth} & ::= a t m n (\text{arg}^*) \{ e \} \quad \text{(methods)} \\
\text{g} & ::= \text{guarded\_by} \ l \mid \text{write\_guarded\_by} \ l \quad \text{(guards)} \\
\text{l} & ::= e \quad \text{(lock expression)}
\end{align*}
\]
ATOMICAJAVA

- Small addition to CONCURRENTJAVA used to illustrate type rules

class Account {
    int balance write_guarded_by this = 0;
    atomic int deposit(int x) requires this { ... }
    atomic int readBalance() { ... }
    atomic int withdraw(int amt) requires this { ... }
}
Type Rules

\[ P; E \vdash e : t \& a \]
Example

Adapted from: p. 6
Example

\[
\begin{align*}
\text{[EXP WHILE]} & \\
& \begin{array}{c}
P; E \vdash e_1 : \text{int} \ & a_1 \quad P; E \vdash e_2 : t \ & a_2 \\
\hline
P; E \vdash \text{while} \ e_1 \ e_2 : \text{int} \ & (a_1;(a_2;a_1)^*)
\end{array}
\end{align*}
\]

Adapted from: p. 6
Example

```java
while (i < 0) { i = nextNumber();}
```

\[
\begin{align*}
\text{[EXP WHILE]} & \\
&P; E \vdash e_1 : \text{int} \ & a_1 & P; E \vdash e_2 : t \ & a_2 \\
&P; E \vdash \text{while } e_1 \ e_2 : \text{int} \ & (a_1; (a_2; a_1)^*)
\end{align*}
\]

Adapted from: p. 6
Example

while (i < 0) { i = nextNumber();}

\[
\begin{align*}
\text{[EXP WHILE]} \quad P; E \vdash e_1 : \text{int} & \quad P; E \vdash e_2 : t & \quad \text{\& } a_1 \\
& \quad P; E \vdash e_2 : t & \quad \text{\& } a_2 \\
\end{align*}
\]

\[
\begin{align*}
P; E \vdash \text{while } e_1 \ e_2 : \text{int} & \quad \text{\& } (a_1; (a_2; a_1)^*)
\end{align*}
\]

Adapted from: p. 6
Example

```
while (i < 0) { i = nextNumber(); }

while e₁ e₂
```

[EXP WHILE]

\[
P; E \vdash e₁ : \text{int} \& a₁ \quad P; E \vdash e₂ : t \& a₂
\]

\[
P; E \vdash \text{while } e₁ e₂ : \text{int} \& (a₁;(a₂;a₁)^*)
\]
Example

while \( i < 0 \) { \( i = \) nextNumber() \}

while \( e_1 e_2 \)

Name

Preconditions

[EXP WHILE]

\[ P; E \vdash e_1 : \text{int} \& a_1 \]

\[ P; E \vdash e_2 : t \& a_2 \]

\[ P; E \vdash \text{while } e_1 e_2 : \text{int} \& (a_1; (a_2; a_1)*) \]

Adapted from: p. 6
Implementation

- Tool-without-a-name
- Built on top of rccjava
- Annotations in special comments
  - e.g., /*# guarded_by this */
- Default atomicity is cmpd
- Checker infers atomicity of each expression, compares it to stated value
Evaluation

• Ran it on several JDK 1.4 classes:
  • java.util.zip.Inflater
  • java.util.zip.Deflater
  • java.io.PrintWriter
  • java.util.Vector
  • java.net.URL
  • java.lang.StringBuffer
  • java.lang.String
Number of Annotations

<table>
<thead>
<tr>
<th>Class</th>
<th>LOC</th>
<th>Annotations per KLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total</td>
</tr>
<tr>
<td>java.util.zip.Inflater</td>
<td>296</td>
<td>20.3</td>
</tr>
<tr>
<td>java.util.zip.Deflater</td>
<td>364</td>
<td>24.7</td>
</tr>
<tr>
<td>java.io.PrintWriter</td>
<td>557</td>
<td>35.9</td>
</tr>
<tr>
<td>java.util.Vector</td>
<td>1029</td>
<td>13.6</td>
</tr>
<tr>
<td>java.net.URL</td>
<td>1269</td>
<td>33.1</td>
</tr>
<tr>
<td>java.lang.StringBuffer</td>
<td>1272</td>
<td>18.9</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>2399</td>
<td>21.7</td>
</tr>
<tr>
<td>All benchmarks</td>
<td>7186</td>
<td>23.3</td>
</tr>
</tbody>
</table>

Source: p. 9
public final class StringBuffer ... {

    private int count /* guarded_by this */;

    /\*\# atomic */ // does not type check
    public synchronized StringBuffer append(StringBuffer sb) {
        if (sb == null) { sb = NULL; }
        int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length) { expandCapacity(newcount); }
        sb.getChars(0, len, value, count);
        count = newcount;
        return this;
    }

    /\*\# atomic */
    public synchronized int length() { return count; }

    /\*\# atomic */
    public synchronized void getChars(…) { … }
public final class StringBuffer ... {
    ...
    private int count /* guarded_by this */;

    /*# atomic */ // does not type check
    public synchronized StringBuffer append(StringBuffer sb){
        if (sb == null) { sb = NULL; }
        int len = sb.length(); // len may be stale
        int newcount = count + len;
        if (newcount > value.length) { expandCapacity(newcount); }  
        sb.getChars(0, len, value, count); // use of stale len
        count = newcount;
        return this;
    }

    /*# atomic */
    public synchronized int length() { return count; }

    /*# atomic */
    public synchronized void getChars(...){ ... }
}
Real World Experience

- `__constructor_holds_lock` flag
- “Typecasts”

- “Adding appropriate type annotations to these classes was mostly straightforward, once the synchronization discipline of each class was understood.”
Future Work

- Escape analysis
- Rep-exposure
- Sound “typecasts”
Current Status

• This tool is mostly defunct
• Ongoing work in similar areas
• Other papers today
• *TOPLAS* journal article

Types for Atomicity: Static Checking and Inference for Java.
Summary

Concurrency is very hard, type checking less so

1. Developed a type-theoretic model for atomicity

2. Introduced Java language extensions, e.g.
   - /*# guarded_by this */
   - /*# atomic */

3. Developed tool to perform static atomicity checking

4. Evaluated it by adding atomicity metadata to seven existing JDK classes, and found several bugs
Utility?

- Still requires explicit metadata (i.e., effort)
- Easier to use it from beginning
- In order to retrofit, need to understand
- Too onerous?

- Is “zero-effort” too much to ask?
Questions?