Lock Inference Proven Correct

Dave Cunningham    Sophia Drossopoulou    Susan Eisenbach

Imperial College London

FTfJP 2008
Why Atomic sections

Example:

```java
atomic {
    Node x = new Node();
    x.next = list.first;
    list.first = x;
}
```

- Semantics easy for programmers to understand
  - Guaranteed that threads don’t interfere
- Concurrency much easier
- Naive implementation is inefficient
- Lots of research tries to interleave more threads (which is hard)
Why Lock Inference?

One thread in an atomic section.
Why Lock Inference?

One thread in an atomic section.
Non-interfering threads allowed to proceed.
In CC’08 we published an algorithm that compiles atomic sections:

```
atomic {
  z.g = this;
  y = x.g;
  y.f = 42;
}
```

```
lock(x, x.g, z, this);
  z.g = this;
  y = x.g;
  y.f = 42;
unlockall();
```
While Loops

NFAs allow iterations:

```
atomic {
    while (...) {
        x = x.n;
    }
}
```

Termination is due to the use of CFG nodes as NFA states.
While Loops

NFAs allow iterations:

```java
atomic {
    while (...) {
        x = x.n;
    }
}
```

Termination is due to the use of CFG nodes as NFA states.
While Loops

NFAs allow iterations:

```plaintext
atomic {
    while (...) {
        x = x.n;
    }
}
```

Termination is due to the use of CFG nodes as NFA states.
While Loops

NFAs allow iterations:

```cpp
atomic {
    while (...) {
        x = x.n;
    }
}
```

Termination is due to the use of CFG nodes as NFA states.
While Loops

NFAs allow iterations:

```plaintext
atomic {
    while (...) {
        x = x.n;
    }
}
```

Termination is due to the use of CFG nodes as NFA states.
In This Paper We Prove Soundness

Our approach:

- Assume the two-phase locking discipline is sound
- Don’t have to worry about concurrency at all!
- Prove analysis correctly infers the objects accessed

In this talk, I will:

- Show analysis in more detail
- Formalise the meaning of the NFAs:
  \[
  \begin{array}{c}
  \bullet \\
  \xrightarrow{x} \\
  2 \\
  \xrightarrow{n}
  \end{array}
  \]
- Show soundness theorem
The Transfer Functions

How to formalise the analysis:

\[ G' = a(st^n) \cup t(st^n)(G) \]

**Addition function** \( a(st^n) \) inserts the accesses performed by \( st \)

**Translation function** \( t(st^n)(G) \) rewrites \( G \) to compensate for state change
Addition Function

(introduces new accesses into the CFG)

\[ x.f = y \]

\[ x = y.f \]
A standard kill/gen function

Translate the accesses to balance the effect of the statement:
Translation Function (load)

\[ x = y \cdot f \]
Translation Function (store)

\[ x.f = y \]
What is soundness?

Does the top NFA safely approximate the addresses accessed?

- Let execution start from the top of the atomic section.
- Let $A$ be the addresses accessed (define a semantics for this)
- Let $G$ be the NFA from the analysis

Want to show $A \subseteq G$

But we have no link between static world $G$ and dynamic world $A$.

- Static characterisation of a set of objects.
- Easily represent infinitely many accesses.
An **assignment** $\varphi$ interprets $G$ in a stack, heap. Assignment $\varphi$ stores a set of addresses at each NFA state.

**Table:**

<table>
<thead>
<tr>
<th>var</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addr</th>
<th>field f</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

**Diagram:**

```
NFA

x

y

2 f
{ }

3 f
{ }

4 { }
```
Linking NFAs to addresses

An assignment \( \varphi \) interprets \( G \) in a stack, heap. Assignment \( \varphi \) stores a set of addresses at each NFA state.

| var | addr | | addr | field f |
|-----|------||------|---------|
| x   | 10   | | 10   | 20      |
| y   | 30   | | 20   | 10      |
|     | 30   | | 30   | 40      |

NFA:

- Node 2 with input label \( x \) and output label \( \{10\} \)
- Node 3 with input label \( y \) and output label \( \{\} \)
- Node 4 with input label \( f \) and output label \( \{\} \)
An assignment $\varphi$ interprets $G$ in a stack, heap.
Assignment $\varphi$ stores a set of addresses at each NFA state.

<table>
<thead>
<tr>
<th>stack</th>
<th>heap</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>var</td>
<td>addr</td>
<td>addr</td>
</tr>
<tr>
<td>x</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>30</td>
<td>20</td>
</tr>
</tbody>
</table>

{10, 20}  
{}  
{}
Linking NFAs to addresses

An assignment $\varphi$ interprets $G$ in a stack, heap.
Assignment $\varphi$ stores a set of addresses at each NFA state.

<table>
<thead>
<tr>
<th>var</th>
<th>addr</th>
<th>addr</th>
<th>field f</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>y</td>
<td>30</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

NFA:
- States: 2, 3, 4
- Edges: x → 2, y → 3, f
- Fields: f
- Addresses: {10, 20}, {30}, {}
An assignment $\varphi$ interprets $G$ in a stack, heap. Assignment $\varphi$ stores a set of addresses at each NFA state.

<table>
<thead>
<tr>
<th>var</th>
<th>addr</th>
<th>addr</th>
<th>field f</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>y</td>
<td>30</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

NFA:

- State 2: $\{10, 20\}$
- State 3: $\{30\}$
- State 4: $\{40\}$
- Transitions:
  - x from 1 to 2
  - y from 1 to 3
  - f from 3 to 4
We can represent the NFAs as e.g.:

\[ G = \{ x \mapsto 2, 2 \xrightarrow{f} 2, y \mapsto 3, 3 \xrightarrow{f} 4 \} \]

We say \( h, \sigma \vdash G : \varphi \) if \( \varphi \) is consistent with heap \( h \), stack \( \sigma \), NFA \( G \)

i.e. iff

\[
x \mapsto n \in G \Rightarrow \sigma(x) \in \varphi(n)
\]

\[
n \xrightarrow{f} n' \in G \Rightarrow \{ h(a)(f) | a \in \varphi(n) \} \subseteq \varphi(n')
\]
Now we can define soundness:

If:

- $G$ is the NFA returned by the analysis
- $h, \sigma$ is the initial heap, stack
- $A$ is the addresses accessed (operational semantics in paper)
- $\varphi$ is the addresses at each node of $G \quad h, \sigma \vdash G : \varphi$

then we must have $A \subseteq squash(\varphi)$
We used Isabelle/HOL.

- Mostly just sets (with a few lists too)
- Definitions are exactly as presented except for:
  - Explicit quantifiers where they are needed
  - Explicit handling of null, and the undefinedness of partial functions
- Well-formed induction using length of $A$
- $\sim 800$ lines (including definitions)
- $\sim 30$ seconds for proofgeneral to verify on 3Ghz P4
- Proof assistants are cool!
Conclusions

Proved soundness of our lock inference algorithm:

• Use known facts of two-phase discipline
• Use transfer functions to formalise analysis
• Use operational semantics to formalise execution
• Assignments (\(\varphi\)) were the missing link
• Mechanically checked proof

Further work:

• Prove early unlocking
• Prove readers/writers
• Prove arrays, functions, exceptions, etc.
• Improve underlying analysis
Thankyou!
## The ‘atomicity via locks’ arena

<table>
<thead>
<tr>
<th>Papers</th>
<th>Granularity</th>
<th>Assigns</th>
<th>Deadlock</th>
<th>Early unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flanagan99-05</td>
<td>Ownership*</td>
<td>No</td>
<td>N/A</td>
<td>Yes*</td>
</tr>
<tr>
<td>Boyapati02</td>
<td>Ownership*</td>
<td>No</td>
<td>Static</td>
<td>Yes*</td>
</tr>
<tr>
<td>Vaziri05</td>
<td>Static</td>
<td>Yes*</td>
<td>Static</td>
<td>No</td>
</tr>
<tr>
<td>McCloskey06</td>
<td>Dynamic</td>
<td>No</td>
<td>Static</td>
<td>No</td>
</tr>
<tr>
<td>Hicks06</td>
<td>Static</td>
<td>Yes*</td>
<td>Static</td>
<td>No</td>
</tr>
<tr>
<td>Emmi07</td>
<td>Dynamic</td>
<td>Yes*</td>
<td>Static</td>
<td>No</td>
</tr>
<tr>
<td>Halpert07</td>
<td>Dynamic</td>
<td>Yes*</td>
<td>Static</td>
<td>No</td>
</tr>
<tr>
<td><strong>Our work</strong></td>
<td><strong>Multigrain</strong></td>
<td><strong>Yes</strong></td>
<td><strong>Dynamic</strong></td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td>Cherem08</td>
<td>Multigrain</td>
<td>Yes</td>
<td>Static?</td>
<td>No</td>
</tr>
</tbody>
</table>

**Key:** v.good, good, OK, bad
### Two ways of safely interleaving more threads

<table>
<thead>
<tr>
<th></th>
<th>Transactional Memory</th>
<th>Lock Inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>Hard</td>
<td>Easy</td>
</tr>
<tr>
<td>Reflection</td>
<td>Easy</td>
<td>Need JIT support</td>
</tr>
<tr>
<td>Native calls</td>
<td>Hard</td>
<td>Hard</td>
</tr>
<tr>
<td>Compiler machinery</td>
<td>Some</td>
<td>Lots</td>
</tr>
<tr>
<td>Runtime machinery</td>
<td>Lots</td>
<td>Some</td>
</tr>
<tr>
<td>Performance</td>
<td>Slow</td>
<td>Fast</td>
</tr>
<tr>
<td>Granularity</td>
<td>Perfect</td>
<td>Reasonable</td>
</tr>
</tbody>
</table>

Key: good, OK, bad
Operational Semantics

We need to know what addresses are accessed by a block of code. A big step operational semantics will suffice for this. We can define it on the CFG to keep it simple.

\[ P \vdash h, \sigma, n \leadsto^* \]

\[ P(n) = [x = y.f, n'] \]
\[ \sigma(y) = a \]
\[ P \vdash h, \sigma[x \mapsto h(a)(f)], n' \leadsto^* \]
\[ P \vdash h, \sigma \{a\} \cup A, n \leadsto^* \]

\[ P(n) = [x = y, n'] \]
\[ P \vdash h, \sigma[x \mapsto \sigma(y)], n' \leadsto^* \]
\[ P \vdash h, \sigma, n \leadsto^* \]