Implementing Atomicity with Locks

Dave Cunningham

April 4, 2006
Motivation
  The Problem
  A New Solution
  Example

Analysis Of Accessed Objects
  Examples
  Formalism
  Correctness
  Termination

Conclusion
  Conclusion
  Future work
Atomic Section

Future concurrent programming languages may include the atomic section.

```python
atomic {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}
```
Atomic Section

Future concurrent programming languages may include the atomic section.

atomic {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}

Efficient implementations must understand *interference*.
Future concurrent programming languages may include the atomic section.

```plaintext
atomic {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}
```

- Efficient implementations must understand *interference*.
- What objects are accessed by atomic code?
Two Approaches

**Transactions**: Log object accesses at runtime.
- Concurrent logs with non-empty intersection $\Rightarrow$ interference.
- Interference avoided by undoing (or not committing) code.
- Atomic code re-executed.
Two Approaches

Transactions: Log object accesses at runtime.
- Concurrent logs with non-empty intersection $\Rightarrow$ interference.
- Interference avoided by undoing (or not committing) code.
- Atomic code re-executed.

Alternative: Statically infer what objects may be accessed
- Prevent interference using synchronisation.
- Dataflow analysis may be inaccurate (arbitrary pointers).
- But programmers already do this in their heads...
Two Approaches

**Transactions**: Log object accesses at runtime.
- Concurrent logs with non-empty intersection $\Rightarrow$ interference.
- Interference avoided by undoing (or not committing) code.
- Atomic code re-executed.

**Alternative**: Statically infer what objects may be accessed
- Prevent interference using synchronisation.
- Dataflow analysis may be inaccurate (arbitrary pointers).
- But programmers already do this in their heads...

*(Like Ethernet vs Token Ring)*
account.balance := account.balance - amount;
log.append("withdrew ...");
Interference Prevention

synchronized (account, log) {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}

Variables?

Path, sequences of field lookups:

me.brother.girlfriend.car

list.first.next.next.next
Interference Prevention

```java
synchronized (account, log) {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}
```

<table>
<thead>
<tr>
<th>Static</th>
<th>Inference returns {account, log}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Evaluate {account, log} to find out what to lock.</td>
</tr>
</tbody>
</table>
synchronized (account, log) {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}

| Static   | Inference returns \{account, log\} |
| Dynamic  | Evaluate \{account, log\} to find out what to lock. |

What should the analysis return in general?
Interference Prevention

```java
synchronized (account, log) {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}
```

<table>
<thead>
<tr>
<th>Static</th>
<th>Inference returns {account, log}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Evaluate {account, log} to find out what to lock.</td>
</tr>
</tbody>
</table>

What should the analysis return in general?
- Variables?
Interference Prevention

synchronized (account, log) {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}

| Static | Inference returns \{account, log\} |
| Dynamic | Evaluate \{account, log\} to find out what to lock. |

What should the analysis return in general?
- Variables?
- Arbitrary expressions?
synchronized (account, log) {
    account.balance := account.balance - amount;
    log.append("withdrew ...");
}

| Static  | Inference returns \{account, log\} |
| Dynamic | Evaluate \{account, log\} to find out what to lock. |

What should the analysis return in general?
- Variables?
- Arbitrary expressions?

**Paths:** sequences of field lookups:
- me.brother.girlfriend.car
- list.first.next.next.next.next
### Examples (1)

<table>
<thead>
<tr>
<th>e</th>
<th>( L(e) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>me.brother.car.fuel := 100;</code></td>
<td><code>{ me, me.brother, me.brother.car }</code></td>
</tr>
<tr>
<td><code>if (goodWeather) {</code></td>
<td><code>{ this, drawer, cloakroom }</code></td>
</tr>
<tr>
<td><code>    this.clothing := drawer.hat;</code></td>
<td></td>
</tr>
<tr>
<td><code>} else {</code></td>
<td></td>
</tr>
<tr>
<td><code>    this.clothing := cloakroom.umbrella;</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>
### Examples (2)

<table>
<thead>
<tr>
<th>$e$</th>
<th>$L(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>me.car := you.car; me.car.fuel := 100;</td>
<td>{ me, you, you.car }</td>
</tr>
<tr>
<td>me.car := you.car; dave.car.fuel := 100;</td>
<td>{ me, you, you.car, dave, dave.car }</td>
</tr>
</tbody>
</table>
Definition of $L$

Intuition:

$L(e) \approx \text{“objects that may be accessed by } e\text{”}$
Definition of $L$

Intuition:

$L(e) \approx \text{“objects that may be accessed by } e\text{”}$

(in terms of \textit{paths} through the \textit{initial heap})
Intuition:

\( L(e) \approx \text{“objects that may be accessed by } e \text{”} \)

(in terms of \textit{paths} through the \textit{initial heap})

Definition:

\[
L(x) = \emptyset
\]
Definition of \( L \)

**Intuition:**

\( L(e) \approx \text{“objects that may be accessed by } e \text{”} \)

(in terms of paths through the initial heap)

**Definition:**

\[
\begin{align*}
L(x) & = \emptyset \\
L(\text{if } q \ e_1 \ e_2) & = L(q) \cup (L(e_1) \cup L(e_2))
\end{align*}
\]
Definition of $L$

**Intuition:**

$L(e) \approx \text{"objects that may be accessed by } e\text{"}$
(in terms of *paths* through the *initial heap*)

**Definition:**

\[
\begin{align*}
L(x) &= \emptyset \\
L(\text{if } q \ e_1 \ e_2) &= L(q) \cup (L(e_1) \cup L(e_2)) \\
L(q.f) &= L(q) \cup \{q\} \\
L(q.f := r) &= L(q) \cup L(r) \cup \{q\}
\end{align*}
\]
Definition of $L$

**Intuition:**

$L(e) \approx \text{“objects that may be accessed by } e\text{”}$

(in terms of paths through the initial heap)

**Definition:**

\[
\begin{align*}
L(x) &= \emptyset \\
L(\text{if } q \; e_1 \; e_2) &= L(q) \cup (L(e_1) \cup L(e_2)) \\
L(q.f) &= L(q) \cup \{q\} \\
L(q.f := r) &= L(q) \cup L(r) \cup \{q\} \\
L(e_1; e_2) &= L(e_1) \cup T_{e_1}(L(e_2))
\end{align*}
\]
Definition of $T$

**Intuition:**

$T_e(P') \approx "P' \text{ translated with respect to the side-effects of } e"$
Definition of $T$

**Intuition:**

$T_e(P') \approx \text{“} P' \text{ translated with respect to the side-effects of } e \text{”}$

**Definition:**

\[
\begin{align*}
T_x(P') &= P' \\
T_{q.f}(P') &= P'
\end{align*}
\]
Definition of T

**Intuition:**

$T_e(P') \approx "P' \text{ translated with respect to the side-effects of } e"$

**Definition:**

\[
\begin{align*}
T_x(P') &= P' \\
T_{q.f}(P') &= P' \\
T_{\text{if } q \ e_1 \ e_2}(P') &= T_{e_1}(P') \cup T_{e_2}(P') \\
T_{e_1;e_2}(P') &= T_{e_1}(T_{e_2}(P'))
\end{align*}
\]
Definition of \( T \)

**Intuition:**

\[ T_e(P') \approx "P' translated with respect to the side-effects of e" \]

**Definition:**

\[
\begin{align*}
T_x(P') &= P' \\
T_{q.f}(P') &= P' \\
T_{if \ q \ e_1 \ e_2}(P') &= T_{e_1}(P') \cup T_{e_2}(P') \\
T_{e_1:e_2}(P') &= T_{e_1}(T_{e_2}(P')) \\
T_{q.f:=r}(P') &= \text{something horrible...}
\end{align*}
\]
Definition of \( T \)

**Intuition:**

\( T_e(P') \approx "P' translated with respect to the side-effects of e" \)

**Definition:**

\[
\begin{align*}
T_x(P') &= P' \\
T_{q.f}(P') &= P' \\
T_{\text{if } q e_1 e_2}(P') &= T_{e_1}(P') \cup T_{e_2}(P') \\
T_{e_1;e_2}(P') &= T_{e_1}(T_{e_2}(P')) \\
T_{q.f:=r}(P') &= \text{something horrible...}
\end{align*}
\]

\[
\bigcup_{p' \in P'} \left( \{ r.g \mid p' = _f.g \} \cup \begin{cases} \\
\emptyset & \text{if } p' = q.f \ldots \\
\{p'\} & \text{otherwise} \\
\end{cases} \right)
\]
While loops

Infer a set of constraints and propagate solutions until fixed point.

\[ L(\text{while } p \ e) \supseteq L(p; e) \cup T_e L(\text{while } p \ e) \]

\[ T_{\text{while } p \ e}(P') \supseteq P' \cup T_e (T_{\text{while } p \ e}(P')) \]
Correctness

Can we prove that $L(e)$ always returns the right results?
Can we prove that \( L(e) \) always returns the right results?

- Define operational semantics. \( e, h \xrightarrow{A} v, h' \)
Correctness

Can we prove that \( L(e) \) always returns the right results?

- Define operational semantics. \( e, h \xrightarrow{A} v, h' \)
- Define correctness property for \( L, T \).
Correctness

Can we prove that $L(e)$ always returns the right results?

- Define operational semantics. $e, h \xrightarrow{A} v, h'$
- Define correctness property for $L, T$.
  $A \subseteq \{ h(p) \mid p \in L(e) \}$
Can we prove that $L(e)$ always returns the right results?

- Define operational semantics. $e, h \xrightarrow{A} v, h'$
- Define correctness property for $L, T$.
  \[ A \subseteq \{ h(p) \mid p \in L(e) \} \]
  \[ \forall P'. \{ h'(p') \mid p' \in P' \} \subseteq \{ h(p) \mid p \in T_e(P') \} \]
Can we prove that $L(e)$ always returns the right results?

- Define operational semantics. $e, h \xrightarrow{A} v, h'$
- Define correctness property for $L, T$.
  
  $A \subseteq \{ h(p) \mid p \in L(e) \}$
  
  $\forall P'. \{ h'(p') \mid p' \in P' \} \subseteq \{ h(p) \mid p \in T_e(P') \}$

- Prove by induction over structure of execution.
Termination

Can we prove that the analysis finishes in finite time?
Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>e</th>
<th>L(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
</tr>
<tr>
<td>while (x.next)</td>
<td></td>
</tr>
<tr>
<td>x := x.next;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>$e$</th>
<th>$L(e)$</th>
</tr>
</thead>
</table>
| atomic {
    while (x.next)
        x := x.next;
} | \{ x, \} |

(No fixed point.)
Termination

Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>$e$</th>
<th>$L(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td>{ $x$,</td>
</tr>
<tr>
<td></td>
<td>$x.next$,</td>
</tr>
<tr>
<td>while ($x.next$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$x := x.next;$</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

($L(e)$ must be a fixed point.)

Example:

```
atomic {
    while ($x.next$)
        $x := x.next$;
    }
```
Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>$e$</th>
<th>$L(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
</tr>
<tr>
<td>while (x.next)</td>
<td></td>
</tr>
<tr>
<td>x := x.next;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>{ x,</td>
<td></td>
</tr>
<tr>
<td>x.next,</td>
<td></td>
</tr>
<tr>
<td>x.next.next,</td>
<td></td>
</tr>
</tbody>
</table>
Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>$e$</th>
<th>$L(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
</tr>
<tr>
<td>while (x.next)</td>
<td></td>
</tr>
<tr>
<td>x := x.next;</td>
<td>x,</td>
</tr>
<tr>
<td>}</td>
<td>x.next,</td>
</tr>
<tr>
<td></td>
<td>x.next.next,</td>
</tr>
<tr>
<td></td>
<td>x.next.next.next,</td>
</tr>
<tr>
<td></td>
<td>(No fixed point.)</td>
</tr>
</tbody>
</table>
Termination

Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>e</th>
<th>L(e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
</tr>
<tr>
<td>while (x.next)</td>
<td></td>
</tr>
<tr>
<td>x := x.next;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td>{ x,</td>
</tr>
<tr>
<td>x.next,</td>
<td></td>
</tr>
<tr>
<td>x.next.next,</td>
<td></td>
</tr>
<tr>
<td>x.next.next.next,</td>
<td></td>
</tr>
<tr>
<td>... }</td>
<td></td>
</tr>
</tbody>
</table>
Termination

Can we prove that the analysis finishes in finite time?

\[
\begin{array}{|c|c|}
\hline
\text{e} & \text{L(e)} \\
\hline
\text{atomic \{ \\
\quad \text{while (x.next)} \\
\quad \quad x := x.next; \\
\quad \} } & \{ x, \\
\quad x.next, \\
\quad x.next.next, \\
\quad x.next.next.next, \\
\quad \ldots \} \\
\hline
\end{array}
\]

\[ L(e) \supseteq \{ x \} \cup \{ p.next \mid p \in L(e) \} \]
Can we prove that the analysis finishes in finite time?

<table>
<thead>
<tr>
<th>$e$</th>
<th>$L(e)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic {</td>
<td></td>
</tr>
<tr>
<td>while (x.next)</td>
<td></td>
</tr>
<tr>
<td>x := x.next;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>{ x,</td>
<td></td>
</tr>
<tr>
<td>x.next,</td>
<td></td>
</tr>
<tr>
<td>x.next.next,</td>
<td></td>
</tr>
<tr>
<td>x.next.next.next,</td>
<td></td>
</tr>
<tr>
<td>... }</td>
<td></td>
</tr>
</tbody>
</table>

$L(e) \supseteq \{ x \} \cup \{ p.next \mid p \in L(e) \}$

(No fixed point.)
We can implement atomic \textit{without} transactions:
Conclusion

We can implement atomic *without* transactions:

- atomic \{ e \}
We can implement atomic \textit{without} transactions:

\begin{itemize}
  \item atomic \{e\} \rightarrow
\end{itemize}
We can implement atomic *without* transactions:

\[ \text{atomic } \{ e \} \rightarrow \text{synchronized } (L(e))\{e\} \]
We can implement atomic *without* transactions:

- atomic \{e\} \rightarrow \text{synchronized} (L(e))\{e\}

- (As long as \(e\) does not use any syntax not considered here.)
We can implement atomic *without* transactions:

- atomic \{e\} \rightarrow \text{synchronized} (L(e))\{e\}
- (As long as e does not use any syntax not considered here.)
- (Assuming a suitable widening to deal with non-termination.)
Future Work

Widening of while loops.

- Suppose $L(e)$ grows by field $\text{next}$.

- Suppose $L(e)$ grows by field $\text{next}$.

- If type of $\text{next}$ is $\text{C}$, then lock all instances of $\text{C}$.

- If $\text{next}$ is owned by $o$, then lock all objects owned by $o$.

- (Programmers implicitly do this in Java.)

- Potential for aliasing makes analysis inaccurate.

- $L(e)$ too big.

- Runtime test for aliasing before atomic section.

- Use ownership types to restrict aliasing.

- More concrete languages.

- Recursive methods with dynamic binding.

- Inheritance, exceptions, object construction, arrays, . . .
Future Work

Widening of while loops.

- Suppose \( L(e) \) grows by field `next`.
- If type of `next` is `C`, then lock all instances of `C`.

Potential for aliasing makes analysis inaccurate.

- \( L(e) \) too big.
- Runtime test for aliasing before atomic section.
- Use ownership types to restrict aliasing.

More concrete languages.

- Recursive methods with dynamic binding.
- Inheritance, exceptions, object construction, arrays, . . .
Future Work

**Widening of while loops.**

- Suppose $L(e)$ grows by field `next`.  
- If type of `next` is `C`, then lock all instances of `C`.  
- If `next` is `owned` by `o`, then lock all objects owned by `o`.  
- (Programmers implicitly do this in Java.)
Future Work

Widening of while loops.

- Suppose $L(e)$ grows by field `next`.
- If type of `next` is $C$, then lock all instances of $C$.
- If `next` is owned by $o$, then lock all objects owned by $o$.
- (Programmers implicitly do this in Java.)

Potential for aliasing makes analysis inaccurate.

- $L(e)$ too big.
Future Work

Widening of while loops.

- Suppose $L(e)$ grows by field `next`.
- If type of `next` is $C$, then lock all instances of $C$.
- If `next` is owned by $o$, then lock all objects owned by $o$.
- (Programmers implicitly do this in Java.)

Potential for aliasing makes analysis inaccurate.

- $L(e)$ too big.
- Runtime test for aliasing before atomic section.
Future Work

Widening of while loops.

- Suppose \( L(e) \) grows by field \( \text{next} \).
- If type of \( \text{next} \) is \( C \), then lock all instances of \( C \).
- If \( \text{next} \) is \( \text{owned} \) by \( o \), then lock all objects owned by \( o \).
- (Programmers implicitly do this in Java.)

Potential for aliasing makes analysis inaccurate.

- \( L(e) \) too big.
- Runtime test for aliasing before atomic section.
- Use ownership types to restrict aliasing.
Future Work

Widening of while loops.

- Suppose $L(e)$ grows by field `next`.
- If type of `next` is $C$, then lock all instances of $C$.
- If `next` is owned by $o$, then lock all objects owned by $o$.
- (Programmers implicitly do this in Java.)

Potential for aliasing makes analysis inaccurate.

- $L(e)$ too big.
- Runtime test for aliasing before atomic section.
- Use ownership types to restrict aliasing.

More concrete languages.

- Recursive methods with dynamic binding.
- Inheritance, exceptions, object construction, arrays, ...
Related Work

Cormac Flanagan et al

- Type systems (uses ownership-esque parameters)
- Programs have both atomic and locking primitives
- The latter is verified against the former.
Related Work

Cormac Flanagan et al

- Type systems (uses ownership-esque parameters)
- Programs have both atomic and locking primitives
- The latter is verified against the former.

Mandana Vaziri, Frank Tip, Julian Dolby

- “Atomic sets” (subdivision of an object’s fields)
- All methods are atomic
- Dataflow analysis to infer the atomic sets (objects) accessed.
Related Work

Cormac Flanagan et al

- Type systems (uses ownership-esque parameters)
- Programs have both atomic and locking primitives
- The latter is verified against the former.

Mandana Vaziri, Frank Tip, Julian Dolby

- “Atomic sets” (subdivision of an object’s fields)
- All methods are atomic
- Dataflow analysis to infer the atomic sets (objects) accessed.

Both parties have formalised atomicity. Cormac uses “reduction” (Lipton’75), Vaziri uses serializability (from databases).
Instead of:

```java
synchronized (L(e)) {
    e
}
```

We actually need:

```java
start: let x₁...xₙ = L(e) in
    synchronized (x₁...xₙ) {
        if (x₁...xₙ != L(e)) goto start;
        e
    }
```