

# 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.

```
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*.

# 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*.
- ▶ 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)*

# Interference Prevention

```
account.balance := account.balance - amount;  
log.append("withdrew ...");
```

# Interference Prevention

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

# 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.

# 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?

# 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?

# 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?

# Interference Prevention

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

<b>Static</b>	Inference returns $\{account, log\}$
<b>Dynamic</b>	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`

# Examples (1)

e	L(e)
<pre>me.brother.car.fuel := 100;</pre>	<pre>{ me,   me.brother,   me.brother.car }</pre>
<pre>if (goodWeather) {     this.clothing := drawer.hat; } else {     this.clothing := cloakroom.umbrella; }</pre>	<pre>{ this,   drawer,   cloakroom }</pre>

## Examples (2)

e	L(e)
<pre>me.car := you.car; me.car.fuel := 100;</pre>	<pre>{ me,   you,   you.car }</pre>
<pre>me.car := you.car; dave.car.fuel := 100;</pre>	<pre>{ me,   you,   you.car,   dave,   dave.car }</pre>

## Definition of $L$

### Intuition:

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

## Definition of $L$

### Intuition:

$L(e) \approx$  “objects that may be accessed by  $e$ ”  
(in terms of *paths* through the *initial heap*)

# Definition of $L$

## Intuition:

$L(e) \approx$  “objects that may be accessed by  $e$ ”  
(in terms of *paths* through the *initial heap*)

## Definition:

$$L(x) = \emptyset$$

# Definition of $L$

## Intuition:

$L(e) \approx$  “objects that may be accessed by  $e$ ”  
 (in terms of *paths* through the *initial heap*)

## Definition:

$$L(x) = \emptyset$$

$$L(\text{if } q \ e_1 \ e_2) = L(q) \cup (L(e_1) \cup L(e_2))$$

# Definition of $L$

## Intuition:

$L(e) \approx$  “objects that may be accessed by  $e$ ”  
 (in terms of *paths* through the *initial heap*)

## Definition:

$$\begin{aligned}
 L(x) &= \emptyset \\
 L(\text{if } q \ e_1 \ e_2) &= L(q) \cup (L(e_1) \cup L(e_2)) \\
 L(q.f) &= L(q) \cup \{q\} \\
 L(q.f := r) &= L(q) \cup L(r) \cup \{q\}
 \end{aligned}$$

# Definition of $L$

## Intuition:

$L(e) \approx$  “objects that may be accessed by  $e$ ”  
(in terms of *paths* through the *initial heap*)

## Definition:

$$\begin{aligned}L(x) &= \emptyset \\L(\text{if } q \ e_1 \ e_2) &= L(q) \cup (L(e_1) \cup L(e_2)) \\L(q.f) &= L(q) \cup \{q\} \\L(q.f := r) &= L(q) \cup L(r) \cup \{q\} \\L(e_1; e_2) &= L(e_1) \cup T_{e_1}(L(e_2))\end{aligned}$$

## Definition of $T$

### Intuition:

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

# Definition of $T$

## Intuition:

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

## Definition:

$$T_x(P') = P'$$

$$T_{q.f}(P') = P'$$

# Definition of $T$

## Intuition:

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

## Definition:

$$\begin{aligned}
 T_x(P') &= P' \\
 T_{q.f}(P') &= P' \\
 T_{\text{if } q \ e_1 \ e_2}(P') &= T_{e_1}(P') \cup T_{e_2}(P') \\
 T_{e_1;e_2}(P') &= T_{e_1}(T_{e_2}(P'))
 \end{aligned}$$

# Definition of $T$

## Intuition:

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

## Definition:

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

# Definition of $T$

## Intuition:

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

## Definition:

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

$$\bigcup_{p' \in P'} \left( \{ r.\bar{g} \mid p' = \_ . f . \bar{g} \} \cup \begin{cases} \emptyset & \text{if } p' = q.f \dots \\ \{p'\} & \text{otherwise} \end{cases} \right)$$

# While loops

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

$$\mathbf{L}(\text{while } p \ e) \supseteq \mathbf{L}(p; e) \cup \mathbf{T}_e \mathbf{L}(\text{while } p \ e)$$

$$\mathbf{T}_{\text{while } p \ e}(P') \supseteq P' \cup \mathbf{T}_e(\mathbf{T}_{\text{while } p \ e}(P'))$$

# Correctness

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

# Correctness

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  $\mathbf{L}(e)$  always returns the right results?

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

# Correctness

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

- ▶ Define operational semantics.  $e, h \xrightarrow{A} v, h'$
- ▶ Define correctness property for  $\mathbf{L}$ ,  $\mathbf{T}$ .

$$A \subseteq \{ h(p) \mid p \in \mathbf{L}(e) \}$$

$$\forall P' . \{ h'(p') \mid p' \in P' \} \subseteq \{ h(p) \mid p \in \mathbf{T}_e(P') \}$$

# Correctness

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

- ▶ Define operational semantics.  $e, h \xrightarrow{A} v, h'$
- ▶ Define correctness property for  $\mathbf{L}$ ,  $\mathbf{T}$ .

$$A \subseteq \{ h(p) \mid p \in \mathbf{L}(e) \}$$

$$\forall P' . \{ h'(p') \mid p' \in P' \} \subseteq \{ h(p) \mid p \in \mathbf{T}_e(P') \}$$

- ▶ Prove by induction over structure of execution. □

# Termination

Can we prove that the analysis finishes in finite time?

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {   while (x.next)     x := x.next; }</pre>	

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {     while (x.next)         x := x.next; }</pre>	<pre>{ x,</pre>

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {     while (x.next)         x := x.next; }</pre>	<pre>{ x,   x.next,</pre>

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {   while (x.next)     x := x.next; }</pre>	<pre>{ x,   x.next,   x.next.next,</pre>

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {     while (x.next)         x := x.next; }</pre>	<pre>{ x,   x.next,   x.next.next,   x.next.next.next,</pre>

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {     while (x.next)         x := x.next; }</pre>	<pre>{ x,   x.next,   x.next.next,   x.next.next.next,   ... }</pre>

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {   while (x.next)     x := x.next; }</pre>	<pre>{ x,   x.next,   x.next.next,   x.next.next.next,   ... }</pre>

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

# Termination

Can we prove that the analysis finishes in finite time?

$e$	$L(e)$
<pre>atomic {     while (x.next)         x := x.next; }</pre>	<pre>{ x,   x.next,   x.next.next,   x.next.next.next,   ... }</pre>

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

(No fixed point.)

# Conclusion

We can implement atomic *without* transactions:

# Conclusion

We can implement atomic *without* transactions:

- ▶ atomic {e}

# Conclusion

We can implement atomic *without* transactions:

▶ atomic {e} →

# Conclusion

We can implement atomic *without* transactions:

▶ `atomic {e} → synchronized (L(e)){e}`

# Conclusion

We can implement atomic *without* transactions:

- ▶  $\text{atomic } \{e\} \longrightarrow \text{synchronized } (\mathbf{L}(e))\{e\}$
- ▶ (As long as  $e$  does not use any syntax not considered here.)

# Conclusion

We can implement atomic *without* transactions:

- ▶  $\text{atomic } \{e\} \longrightarrow \text{synchronized } (\mathbf{L}(e))\{e\}$
- ▶ (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 next.

## Future Work

### Widening of while loops.

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

# 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 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.

# 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).

# Emergency Slide!

**Instead of:**

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

**We actually need:**

```
start: let  $x_1 \dots x_n = L(e)$  in
    synchronized ( $x_1 \dots x_n$ ) {
        if ( $x_1 \dots x_n \neq L(e)$ ) goto start;
        e
    }
```