Implementing Atomicity with Locks

Dave Cunningham

April 4, 2006

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 ...");
}
```

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`

Examples (1)

e	$\mathbf{L}(e)$
`me.brother.car.fuel := 100;`	{ `me`, `me.brother`, `me.brother.car` }

`if (goodWeather) {`
    `this.clothing := drawer.hat;`
`} else {`
    `this.clothing := cloakroom.umbrella;`
`}`
	{ `this`,
	`drawer`,
	`cloakroom` }

Examples (2)

e	$\mathbf{L}(e)$
`me.car := you.car;` `me.car.fuel := 100;`	{ `me`, `you`, `you.car` }
`me.car := you.car;` `dave.car.fuel := 100;`	{ `me`, `you`, `you.car`, `dave`, `dave.car` }

Definition of L

Intuition:

$\mathbf{L}(e) \approx$ "objects that may be accessed by e"

Definition of $\mathbf{L}$

Intuition:

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

Definition of $\mathbf{L}$

Intuition:

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

Definition:

$$\mathbf{L}(x) = \emptyset$$

Definition of $\mathbf{L}$

Intuition:

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

Definition:

$$\begin{aligned}\mathbf{L}(x) &= \emptyset \\ \mathbf{L}(\texttt{if}\ q\ e_1\ e_2) &= \mathbf{L}(q) \cup (\mathbf{L}(e_1) \cup \mathbf{L}(e_2))\end{aligned}$$

Definition of $\mathbf{L}$

Intuition:

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

Definition:

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

Definition of **L**

Intuition:

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

Definition:

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

Definition of T

Intuition:

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

Definition of T

Intuition:

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

Definition:

$$\begin{aligned} \mathbf{T}_x(P') &= P' \\ \mathbf{T}_{q.f}(P') &= P' \end{aligned}$$

Definition of $\mathbf{T}$

Intuition:

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

Definition:

$$
\begin{aligned}
\mathbf{T}_x(P') &= P' \\
\mathbf{T}_{q.f}(P') &= P' \\
\mathbf{T}_{\texttt{if}\ q\ e_1\ e_2}(P') &= \mathbf{T}_{e_1}(P') \cup \mathbf{T}_{e_2}(P') \\
\mathbf{T}_{e_1;e_2}(P') &= \mathbf{T}_{e_1}(\mathbf{T}_{e_2}(P'))
\end{aligned}
$$

Definition of $\mathbf{T}$

Intuition:

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

Definition:

$$
\begin{aligned}
\mathbf{T}_x(P') &= P' \\
\mathbf{T}_{q.f}(P') &= P' \\
\mathbf{T}_{\texttt{if}\ q\ e_1\ e_2}(P') &= \mathbf{T}_{e_1}(P') \cup \mathbf{T}_{e_2}(P') \\
\mathbf{T}_{e_1;e_2}(P') &= \mathbf{T}_{e_1}(\mathbf{T}_{e_2}(P')) \\
\mathbf{T}_{q.f:=r}(P') &= \text{something horrible...}
\end{aligned}
$$

Definition of $\mathbf{T}$

Intuition:

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

Definition:

$$
\begin{array}{ll}
\mathbf{T}_x(P') & = P' \\
\mathbf{T}_{q.f}(P') & = P' \\
\mathbf{T}_{\texttt{if}\ q\ e_1\ e_2}(P') & = \mathbf{T}_{e_1}(P') \cup \mathbf{T}_{e_2}(P') \\
\mathbf{T}_{e_1;e_2}(P') & = \mathbf{T}_{e_1}(\mathbf{T}_{e_2}(P')) \\
\mathbf{T}_{q.f:=r}(P') & = \text{something horrible...}
\end{array}
$$

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

While loops

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

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

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

Correctness

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

Correctness

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

- Define operational semantics. $e, h \overset{A}{\rightsquigarrow} v, h'$

Correctness

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

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

Correctness

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

- Define operational semantics. $e, h \overset{A}{\rightsquigarrow} v, h'$
- 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. $e, h \overset{A}{\rightsquigarrow} v, h'$
- 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. $e, h \overset{A}{\rightsquigarrow} v, h'$
- 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. $\square$

Termination

Can we prove that the analysis finishes in finite time?

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$
`atomic { while (x.next) x := x.next; }`	

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$
`atomic { while (x.next) x := x.next; }`	{ x,

Termination

Can we prove that the analysis finishes in finite time?

e	**L**(e)

`atomic {`
`    while (x.next)`
`        x := x.next;`
`}`
	`{ x,`
	`  x.next,`

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$

`atomic {`
`    while (x.next)`
`        x := x.next;`
`}`
	`{ x,`
	`  x.next,`
	`  x.next.next,`

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$

`atomic {`
`    while (x.next)`
`        x := x.next;`
`}`
	`{ x,`
	`  x.next,`
	`  x.next.next,`
	`  x.next.next.next,`

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$

`atomic {`
`    while (x.next)`
`        x := x.next;`
`}`
	{ `x`,
	`x.next`,
	`x.next.next`,
	`x.next.next.next`,
	. . . }

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$

`atomic {`
`    while (x.next)`
`        x := x.next;`
`}`
	`{ x,`
	`  x.next,`
	`  x.next.next,`
	`  x.next.next.next,`
	`  ...}`

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

Termination

Can we prove that the analysis finishes in finite time?

e	$\mathbf{L}(e)$

`atomic {`
`    while (x.next)`
`        x := x.next;`
`}`
	`{ x,`
	`  x.next,`
	`  x.next.next,`
	`  x.next.next.next,`
	`  ...}`

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

(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` $\{e\} \longrightarrow$

Conclusion

We can implement `atomic` *without* transactions:

- $\texttt{atomic } \{e\} \longrightarrow \texttt{synchronized } (\mathbf{L}(e))\{e\}$

Conclusion

We can implement `atomic` *without* transactions:

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

Conclusion

We can implement `atomic` *without* transactions:

- `atomic` $\{e\} \longrightarrow$ `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 $\mathbf{L}(e)$ grows by field `next`.

Future Work

Widening of while loops.

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

Future Work

Widening of while loops.

- Suppose $\mathbf{L}(e)$ 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 $\mathbf{L}(e)$ 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.

- $\mathbf{L}(e)$ too big.

Future Work

Widening of while loops.

- Suppose $\mathbf{L}(e)$ 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.

- $\mathbf{L}(e)$ too big.
- Runtime test for aliasing before atomic section.

Future Work

Widening of while loops.

- Suppose $\mathbf{L}(e)$ 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.

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

Future Work

Widening of while loops.

- Suppose $\mathbf{L}(e)$ 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.

- $\mathbf{L}(e)$ 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 ($\mathbf{L}(e)$) {
    e
}

We actually need:

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