Keep Off the Grass

Locking the Right Path for Atomicity

Dave Cunningham Khilan Gudka Susan Eisenbach

Imperial College London

CC 2008

Atomic blocks

Example:

```
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)

Two ways of safely interleaving more threads

	Transactional Memory	**Lock Inference**
IO	Hard	Easy
Reflection	Easy	Need JIT support
Native calls	Hard	Hard
Compiler machinery	Some	Lots
Runtime machinery	Lots	Some
Contention performance	Slow	Fast
Granularity	Perfect	Reasonable

Key: good, OK, bad

Two-phase lock Discipline

- Everyone uses *two-phase* discipline
- Known that two-phase discipline ⇒ atomicity (Eswaran et al, '76)
- Constraints:
  - Lock acquisitions precede lock releases
  - All accesses nested within appropriate locks

Example:

$$\ldots\ 4\ (\acute{2}\ 2\ 4\ \acute{1}\ 1\ 4\ \acute{3}\ \grave{1}\ 2\ 2\ 4\ \grave{2}\ 3\ \grave{3})\ 4\ \ldots$$

Example 1 - Single Access

Source	Target

`atomic {`
`    x = this.f`
`}`
	`lock(this);`
	`x = this.f;`
	`unlock(this);`

Example 1 - Single Access

Source	Target

atomic {
    x = this.f
}
	// if f is final
	// or this is thread-local
	x = this.f;

Example 1 - Single Access

Source	Target
`atomic {`	`// if f is final`
`    x = this.f`	`// or this is thread-local`
`}`	`x = this.f;`

Henceforth, everything is non-final and shared between threads.

Example 2 - Two accesses

Source

```
atomic {
    this.f = 42;
    x.f = 20;
}
```

Target

```
lock(this);
this.f = 42;
lock(x);
unlock(this);
x.f = 20;
unlock(x);
```

Example 2 - Two accesses

Source

```
atomic {
    this.f = 42;
    x.f = 20;
}
```

Target

```
while (true) {
    lock(this);
    if (lock(x)) {
        break; // yes, proceed
    } else {
        unlock(this);
    }
    // no, try again
}
this.f = 42;
unlock(this);
x.f = 20;
unlock(x);
```

Example 2 - Two accesses

Source

```
atomic {
    this.f = 42;
    x.f = 20;
}
```

Target

```
while (true) {
    lock(this);
    if (lock(x)) { // what if x==null?
        break; // yes, proceed
    } else {
        unlock(this);
    }
    // no, try again
}
this.f = 42;
unlock(this);
x.f = 20;
unlock(x);
```

Example 2 - Two accesses

Source

```
atomic {
    this.f = 42;
    x.f = 20;
}
```

Target

```
while (true) {
    lock(this);
    if (x==null || lock(x)) {
        break; // yes, proceed
    } else {
        unlock(this);
    }
    // no, try again
}
this.f = 42;
unlock(this);
x.f = 20;
unlock(x);
```

Example 2 - Two accesses

Source

```
atomic {
    this.f = 42;
    x.f = 20;
}
```

Target

```
// from now on, assume:
// - deadlock free
// - NPE free
lock(this,x);
this.f = 42;
unlock(this);
x.f = 20;
unlock(x);
```

Pause For Thought on Deadlock...

- We *cannot* insert locks that may deadlock
- Related work avoids deadlock by using ordering locks statically...
- ... but this seriously hurts granularity
- Our rollback strategy should have better granularity
- All lock acquisitions moved to top, this might hurt granularity a bit
- No transaction log required
- In our experience, rollback is actually very rare (minimal overhead)

Example 3 - Assign

Source	Target
`atomic {`	`lock(x);`
`    x = this;`	`x = this;`
`    x.f = 42;`	`x.f = 42;`
`}`	`unlock(x);`

Example 3 - Assign

Source	Target
`atomic {`	`lock(this);`
`    x = this;`	`x = this;`
`    x.f = 42;`	`x.f = 42;`
`}`	`unlock(x);`

Example 4 - Load

Source

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

Target

```
lock(this,this.g);
x = this.g
unlock(this);
x.f = 42;
unlock(x);
```

Example 5 - Store

Source

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

Target

```
lock(x,this);
x.g = this;
y = x.g;
unlock(x);
y.f = 42;
unlock(y);
```

Example 6 - Construction

Source	Target
`atomic {`	
`    x = new C;`	`x = new C;`
`    x.f = 42;`	`x.f = 42;`
`}`	

`atomic {`	
`    x = null;`	`x = null;`
`    x.f = 42;`	`x.f = 42;`
`}`	

Example 7 - Readers/Writers

Many threads may read concurrently.

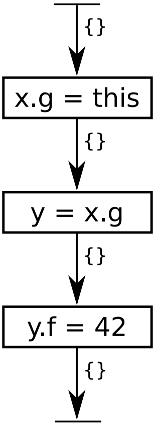
Source	Target
	`lockw(x);`
`atomic {`	`x.f = 10;`
`    x.f = 10;`	`lockr(x);`
`    y = x.g;`	`unlockw(x);`
`}`	`y = x.g;`
	`unlockr(x);`

How Does This Work?

Source

```
// "Store" example
// again.

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

// This time
// we will use
// r/w locks.
```

CFG



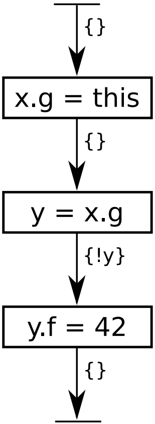
Target

How Does This Work?

Source

```
// "Store" example
// again.

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

// This time
// we will use
// r/w locks.
```

CFG



{}

x.g = this

{}

y = x.g

{!y}

y.f = 42

{}

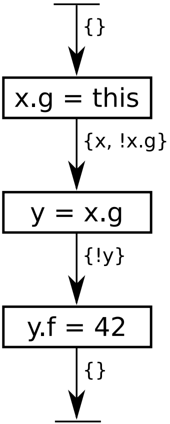
Target

How Does This Work?

Source

```
// "Store" example
// again.

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

// This time
// we will use
// r/w locks.
```

CFG

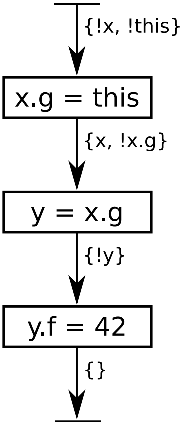


Target

How Does This Work?

Source	CFG	Target

```
// "Store" example
// again.

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

// This time
// we will use
// r/w locks.
```

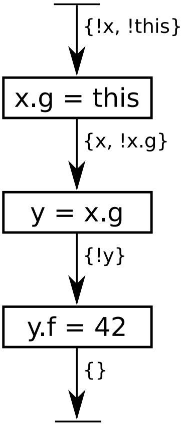


How Does This Work?

Source

```
// "Store" example
// again.

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

// This time
// we will use
// r/w locks.
```

CFG



Target

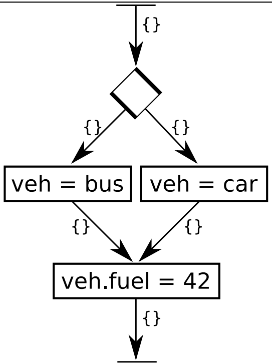
```
lockw(x,this);

x.g = this;
lockr(x);
unlockw(x);
//lockw(x.g);
//unlockw(this);

y = x.g;
//lockw(y);
//unlockw(x.g);
unlockr(x);

y.f = 42;
unlockw(y);
```

Example8 - If

Source

```
atomic {
    if (b) {
        veh = bus;
    } else {
        veh = car;
    }
    veh.fuel = 42;
}
```

Target



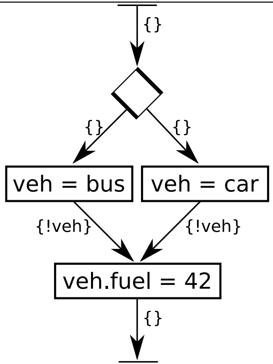
Example8 - If

Source	Target	

Source

```
atomic {
    if (b) {
        veh = bus;
    } else {
        veh = car;
    }
    veh.fuel = 42;
}
```

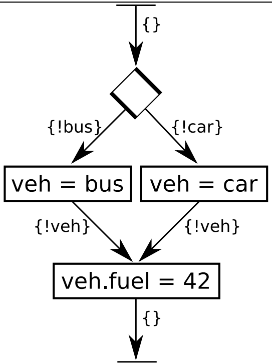
Target



Example8 - If

Source	Target	

```
atomic {
    if (b) {
        veh = bus;
    } else {
        veh = car;
    }
    veh.fuel = 42;
}
```

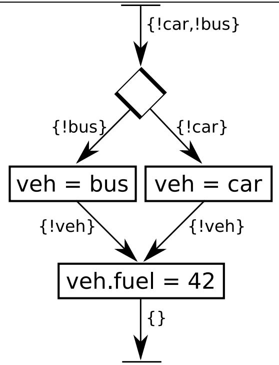


Example8 - If

Source

```
atomic {
    if (b) {
        veh = bus;
    } else {
        veh = car;
    }
    veh.fuel = 42;
}
```

Target

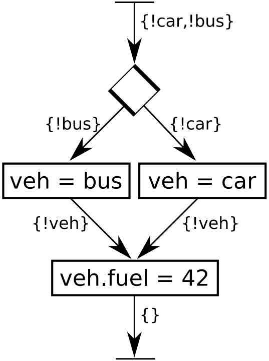


Example8 - If

Source

```
atomic {
    if (b) {
        veh = bus;
    } else {
        veh = car;
    }
    veh.fuel = 42;
}
```

Target



```
lockw(car,bus);
if (b) {
    unlockw(car);
    veh = bus;
} else {
    unlockw(bus);
    veh = car;
}
veh.fuel = 42;
unlockw(veh);
```

Example 9 - While

```
class Node {
    Node n;
    int f;
}
```

```
atomic {
    while (x.n!=null) {
        x = x.n;
    }
    x.f = 42;
}
```

```
//lockw(x, x.n, x.n.n, ...);
while (x.n!=null) {
    x = x.n;
}
x.f = 42;
```

Example 9 - While

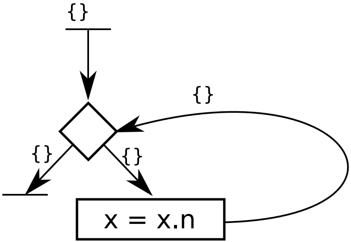
```
class Node {
    Node n;
    int f;
}
```

```
atomic {
    while (x.n!=null) {
        x = x.n;
    }
    x.f = 42;
}
```

```
lockw(Node);
while (x.n!=null) {
    x = x.n;
}
lockw(x);
unlockw(Node);
x.f = 42;
unlockw(x);
```

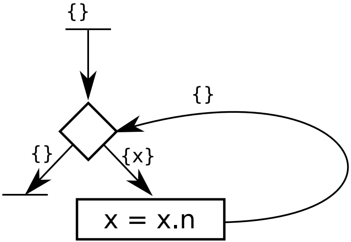
How does it work?

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



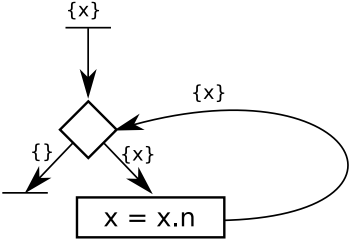
How does it work?

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



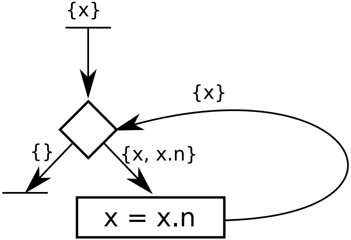
How does it work?

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



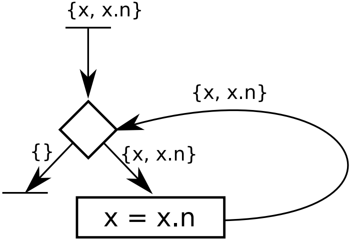
How does it work?

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



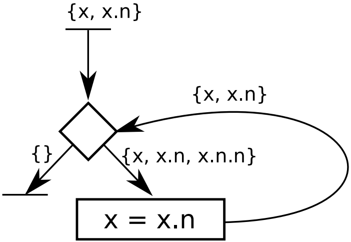
How does it work?

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



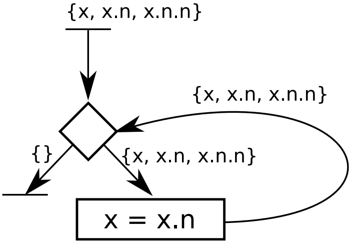
How does it work?

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



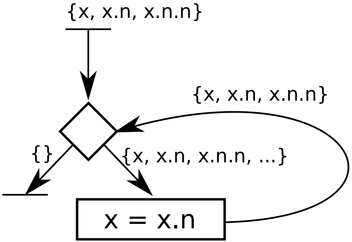
How does it work?

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



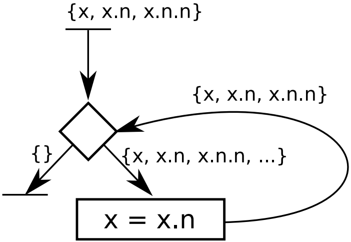
How does it work?

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



How does it work?

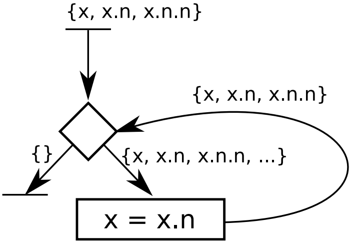
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



Analysis doesn't terminate :(

How does it work?

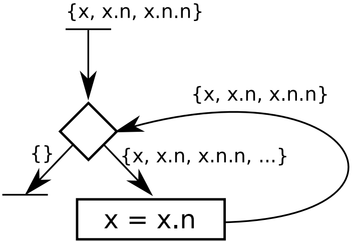
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



How do we solve this?

How does it work?

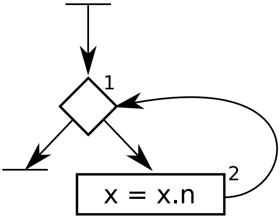
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



First, number the CFG nodes...

How does it work?

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



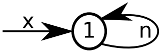
First, number the CFG nodes...

Nondeterministic Finite Automata

Recap:

- Propogating sets of "paths" through the graph.
- (This is a static characterisation of a set of objects.)
- We cannot represent an infinite set of paths: $\{x, x.n, x.n.n, \ldots\}$
- Use regular expressions? $\{x.n^*\}$ (sadly, hard to mechanise...)
- Use nondeterministic finite automata (NFAs)?

NFAs *easily* represent infinite sets of paths!



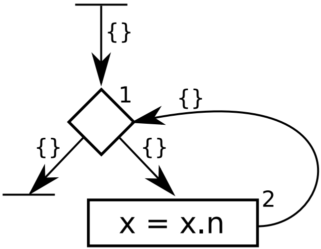
Represent with a set of edges: $\{x \mapsto 1, 1 \rightarrow^n 1\}$

Constrain the set of automata nodes to the set of CFG nodes...

How does it work?

NFAs avoid the infinite loop:

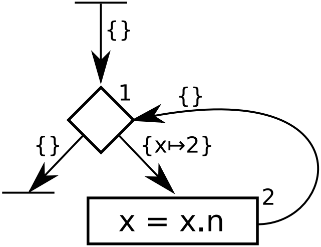
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

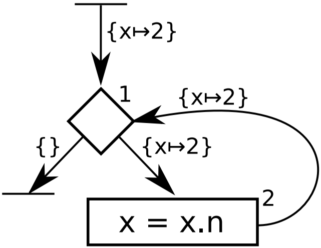
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

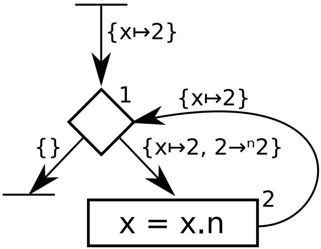
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

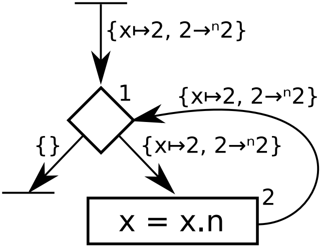
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

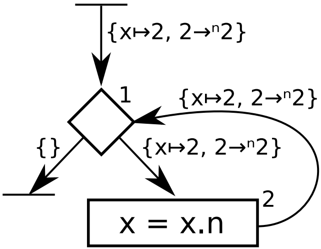
```
atomic {
    while (...) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

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

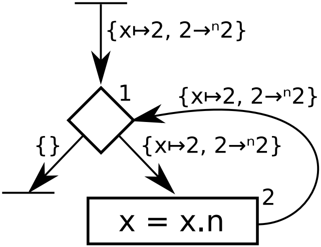


Analysis now terminates.

How does it work?

NFAs avoid the infinite loop:

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



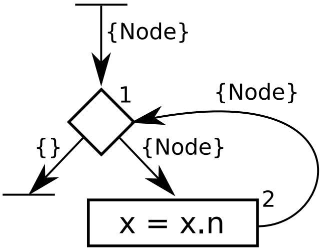
Analysis now terminates.

How do we insert locks?

How does it work?

NFAs avoid the infinite loop:

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



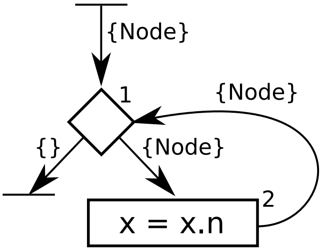
Analysis now terminates.

How do we insert locks?

How does it work?

NFAs avoid the infinite loop:

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



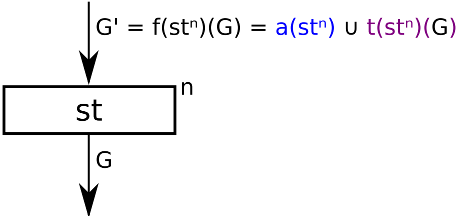
```
lockr(Node);
while (...) {
    x = x.n;
}
unlockr(Node);
```

Analysis now terminates.

How do we insert locks?

The Transfer Functions

Program analyses defined by transfer functions f(st^n)

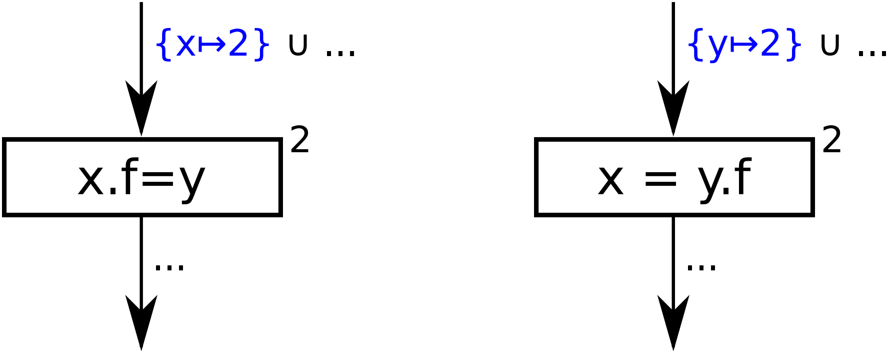


Addition function a(st^n) inserts the accesses performed by **st**

Translation function t(st^n)(G) rewrites G to compensate for state change

Addition function

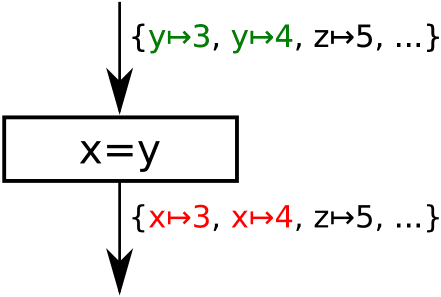
(introduces new accesses into the CFG)



Translation Function (easy cases)

A standard kill/gen function

$$\begin{aligned}
t[x = y]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\} \cup \{y \mapsto n' | x \mapsto n' \in G\} \\
t[x = \texttt{null}]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\} \\
t[x = \texttt{new}]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\}
\end{aligned}$$



{y↦3, y↦4, z↦5, ...}

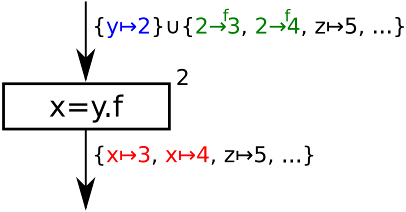
x=y

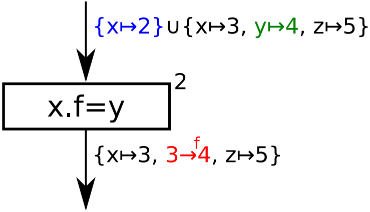
{x↦3, x↦4, z↦5, ...}

Translation Function (harder cases)

$$\begin{aligned} t[x = y.f]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\} \\ &\quad \cup \{n \to^f n' | x \mapsto n' \in G\} \end{aligned}$$

$$\begin{aligned} t[x.f = y]^n(G) &= G \setminus \{n' \to^f \_ \mid x \mapsto n' \in G, \\ &\qquad\qquad (\nexists z \neq x : z \mapsto n' \in G), \\ &\qquad\qquad (\nexists n''' : n''' \to_{\_} n' \in G)\} \\ &\quad \cup \{y \mapsto n' \mid \_ \to^f n' \in G\} \end{aligned}$$





Conclusions

Implemented atomic sections using lock inference:

- Two-phase discipline
- Locks are multi-granularity, read/write, reentrant, deadlock-free
- Unlock as early as possible for better granularity
- Implemented for a subset of Java in custom interpreter
- Currently implementing for full Java using soot

Further work:

- Better precision (ownership types?)
- Better runtime performance
- Better compiletime performance (JIT possible?)
- Nested atomicity would be nice
- Thread-local type system

The ‘atomicity via locks’ arena

Papers (chron. order)	**Granularity** (* locks not) inferred)	**Assigns** (* inside domain)	**Deadlock**	**Early unlock** (* sync block)
Flanagan99-05	Ownership*	No	N/A	Yes*
Boyapati02	Ownership*	No	Static	Yes*
Vaziri05	Static	Yes*	Static	No
McCloskey06	Dynamic	No	Static	No
Hicks06	Static	Yes*	Static	No
Emmi07	Dynamic	Yes*	Static	No
Halpert07	Dynamic	Yes*	Static	No
This paper	Multigrain	Yes	Dynamic	Yes
Cherem08	Multigrain	Yes	Static?	No

Key: v.good, good, OK, bad

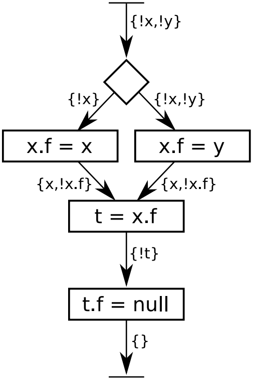
Questions

Balancing Example

Source

```
atomic {
    if (b) {
        x.f = x;
    } else {
        x.f = y;
    }
    t = x.f;
    t.f = null;
}
```

CFG



Target

```
lockw(x,y);
if (b) {
    unlockw(y);
    x.f = x;
    lockr(x);
} else {
    x.f = y;
    lockr(x);
    unlockw(x);
}
t = x.f;
unlockr(x);
t.f = null;
unlockw(t);
```

What Should we Prove?

Already known that two-phase locking $\Longrightarrow$ atomicity.
Therefore sufficient to show we are two-phase.

- Clearly the acquires precede the releases
- Locking a class can be thought of as locking every instance
- We are locking everything in the NFA.

We need to prove the NFA inferred by the analysis represents the accesses actually performed by the code...

Soundness?

Let's invent some notation for the ideas:

- h, σ is the initial heap, stack
- $P \vdash h, \sigma, n \overset{A}{\leadsto}^{*}$ means an incomplete execution from CFG node n can access the set of addresses A
- X maps every CFG node n to an NFA G
- $P \vdash X$ means that X is the fixed point of the analysis of CFG P

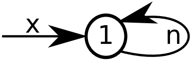
Soundness:

$$\left.\begin{array}{l} P \vdash h, \sigma, n \overset{A}{\leadsto}^{*} \\ P \vdash X \\ X(n) = G \end{array}\right\} \Longrightarrow A \subseteq G?$$

Not quite, but almost...

Assigning Meaning to NFAs

Recall the earlier NFA: (let's call it G)



- G is a static repepresention of a set of objects
- When combined with a h, σ, it resolves into a set of objects
- We must formalise this...

Assignments φ

An assignment φ maps a consistent set of addresses to each node in G. (with respect to the h, σ)

$$G = \{x \mapsto 1, 1 \rightarrow^{next} 1\}$$

We say $h, \sigma \vdash G : \varphi$ if φ is consistent with h, σ, G

Example: If
$\sigma(x) = a_1$
$h(a_1)(next) = a_2$
$h(a_2)(next) = a_1$
$h(a_3)(next) = a_3$
$\varphi(1) = \{a_1, a_2\}$
then
$h, \sigma \vdash G : \varphi$

Assignments φ

An assignment φ maps a consistent set of addresses to each node in G. (with respect to the h, σ)

$$G = \{x \mapsto 1, 1 \rightarrow^{next} 1\}$$

We say $h, \sigma \vdash G : \varphi$ if φ is consistent with h, σ, G

Example: If
$\sigma(x) = a_1$
$h(a_1)(next) = a_2$
$h(a_2)(next) = a_1$
$h(a_3)(next) = a_3$
$\varphi'(1) = \{a_1, a_2, a_3\}$
then
$h, \sigma \vdash G : \varphi'$

Assignments φ

An assignment φ maps a consistent set of addresses to each node in G. (with respect to the h, σ)

$$G = \{x \mapsto 1, 1 \rightarrow^{next} 1\}$$

We say $h, \sigma \vdash G : \varphi$ if φ is consistent with h, σ, G

Example: If
$\sigma(x) = a_1$
$h(a_1)(next) = a_2$
$h(a_2)(next) = a_1$
$h(a_3)(next) = a_3$
$\varphi'(1) = \{a_1, a_2, a_3\}$
then
$h, \sigma \vdash G : \varphi'$

$$\frac{\begin{array}{l} x \mapsto n \in G \Rightarrow \sigma(x) \in \varphi(n) \\ n \rightarrow^{f} n' \in G \Rightarrow \{h(a)(f) | a \in \varphi(n)\} \subseteq \varphi(n') \end{array}}{h, \sigma \vdash G : \varphi}$$

Assignments φ

An assignment φ maps a consistent set of addresses to each node in G. (with respect to the h, σ)

$$G = \{x \mapsto 1, 1 \rightarrow^{next} 1\}$$

We say $h, \sigma \vdash G : \varphi$ if φ is consistent with h, σ, G

Example: If
$\sigma(x) = a_1$
$h(a_1)(next) = a_2$
$h(a_2)(next) = a_1$
$h(a_3)(next) = a_3$
$\varphi'(1) = \{a_1, a_2, a_3\}$
then
$h, \sigma \vdash G : \varphi'$

$$\frac{\begin{array}{l} x \mapsto n \in G \Rightarrow \sigma(x) \in \varphi(n) \\ n \rightarrow^{f} n' \in G \Rightarrow \{h(a)(f) | a \in \varphi(n)\} \subseteq \varphi(n') \end{array}}{h, \sigma \vdash G : \varphi}$$

squash(φ) gets the addresses from φ

Soundness?

Now we can define soundness properly:

- h, σ is the initial heap, stack
- $P \vdash h, \sigma, n \overset{A}{\rightsquigarrow}^*$ means an incomplete execution from CFG node n can access the set of addresses A
- X maps every CFG node n to an NFA G
- $P \vdash X$ means that X is the fixed point of the analysis of CFG P
- φ is the addresses represented by the static G.

Soundness:

$$\left.\begin{array}{l} P \vdash h, \sigma, n \overset{A}{\rightsquigarrow}^* \\ P \vdash X \\ X(n) = G \\ h, \sigma \vdash G : \varphi \end{array}\right\} \Longrightarrow A \subseteq \mathit{squash}(\varphi)$$

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.

$$\frac{}{P \vdash h, \sigma, n \overset{\{\}}{\rightsquigarrow}^{*}}$$

$$\frac{\begin{array}{l} P(n) = [x = y.f, n'] \\ \sigma(y) = a \\ P \vdash h, \sigma[x \mapsto h(a)(f)], n' \overset{A}{\rightsquigarrow}^{*} \end{array}}{P \vdash h, \sigma, n \overset{\{a\} \cup A}{\rightsquigarrow}^{*}}
\qquad
\frac{\begin{array}{l} P(n) = [x = y, n'] \\ P \vdash h, \sigma[x \mapsto \sigma(y)], n' \overset{A}{\rightsquigarrow}^{*} \end{array}}{P \vdash h, \sigma, n \overset{A}{\rightsquigarrow}^{*}}$$

Soundness!

Proved with 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
    - A few concessions so we could use primitive recursion:
        - Convenient to make A a list of "addr option"
        - Convenient to store set of constructed objects C
- Induction over structure of A
- $\sim$ 940 lines (including definitions)
- $\sim$ 30 seconds for proofgeneral to verify on an early P4
- The 2 big theorems were 443 and 75 steps
- Proof assistants are cool!