Keep Off the Grass

Locking the Right Path for Atomicity

Dave Cunningham Khilan Gudka Susan Eisenbach

Imperial College London

07/03/2008

Atomicity

In general...

- Semantics trivial to understand:

$$\frac{e, g \rightsquigarrow^* v, g'}{T \cup \{E[\texttt{atomic } e]\}, g \rightsquigarrow T \cup \{E[v]\}, g'}$$

- Naive implementation is inefficient
- Lots of research tries to interleave more threads...
- Deciding which threads can interleave is hard

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

Comparison with Transactional Memory

There are two main approaches to allowing more threads:

- **Transactional Memory**
  Disadvantages:
  - IO, JNI, etc is very hard
  - Performs badly under high contention
  - Lots of runtime machinery

- **Lock Inference**
  Disadvantages:
  - Reflection, JNI, etc is very hard
  - Worse granularity (common example: non-empty queue)
  - Requires more compiler machinery than TM

Related Work

Alternative ways of implementing atomic sections:

- Transactional Memory (more work than I can cite)
- Other lock inference approaches:
  - Associating Synchronisation Constraints w/Data (POPL'05)
  - Lock Inference for Atomic Sections (TRANSACT'06)
  - Lock Allocation (POPL'07)
  - Component-Based Lock Allocation (PACT'07)
  - Inferring Locks for Atomic Sections (PLDI'08)
- Also, verification of programmer-implemented atomicity
  - Cormac Flanagan et al (everywhere since 1999)
  - "Autolocker" (POPL'06)
  - Cunningham et al (VAMP'07)

Lock Inference

Inferred locks

- We try for at least the performance of manual locking
- Even if we are not as fast, we will at least be correct
- The accuracy of the inference determines the parallelism
- We avoid inferring locks for accesses to immutable data
- Knowing what objects are thread-local is also beneficial
- In theory, ownership types allow very good performance

Lock Discipline

- Usually require *two-phase* discipline
  - Lock acquisitions precede lock releases
  - All accesses nested within appropriate locks
- Example:

  4 (ź 2 4 í 1 4 ś ì 2 2 4 ò 3 ì) 4
- Known that two phase locking $\Rightarrow$ atomicity, due to Lipton (POPL'75)
- Everyone else uses two-phase

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

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

Example 2

Source	Target

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

Example 2

Source

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

Target

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

Example 2

Source

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

Target

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

Example 2

Source

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

Target

```
// from now on, assume:
// - deadlock free
// - NPE free
// - CCE 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
- The programmer cannot recover the situation
- Related work avoids deadlock by using ordering locks statically...
- ... but this seriously hurts granularity
- Our rollback strategy should have better granularity
- In our experience, rollback is actually very rare
- Overhead should be minimal
- All lock acquisitions moved to top, this might hurt granularity a bit
- No transaction log required

Example 3 - Assign

Source	Target

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

Example 3 - Assign

Source	Target

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

Example 4 - Load

Source	Target

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

Example 5 - Construction

Source	Target

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

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

Example 6 - 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);`

Example 7 - Store

Source	Target

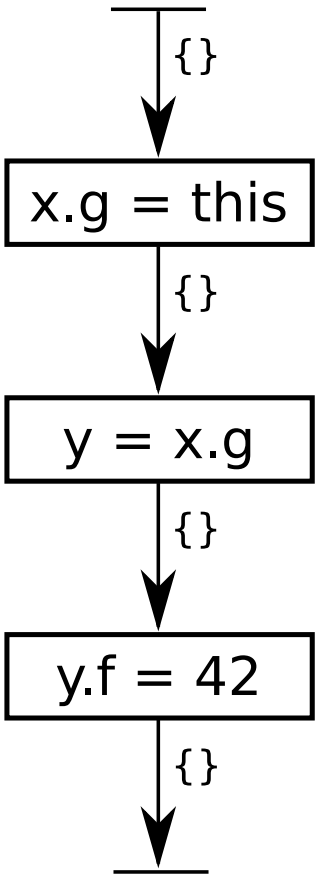
atomic {
  x.g = this;
  y = x.g;
  y.f = 42;
}
	lockw(x,this);
	x.g = this;
	lockr(x);
	unlockw(x);
	y = x.g;
	unlockr(x);
	y.f = 42;
	unlockw(y);

How Does This Work?

Backwards program analysis propogates locks to start of block

Source	CFG	Target

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

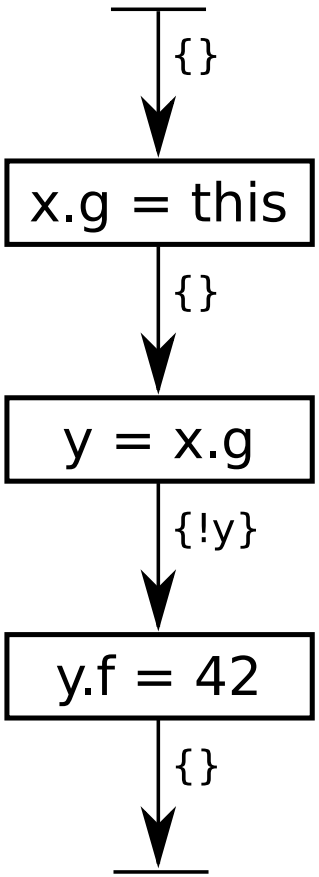


How Does This Work?

Backwards program analysis propogates locks to start of block

Source	CFG	Target

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

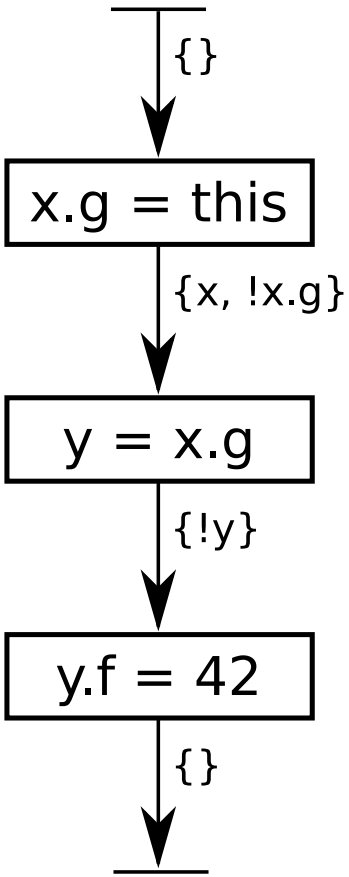


How Does This Work?

Backwards program analysis propogates locks to start of block

Source	CFG	Target

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

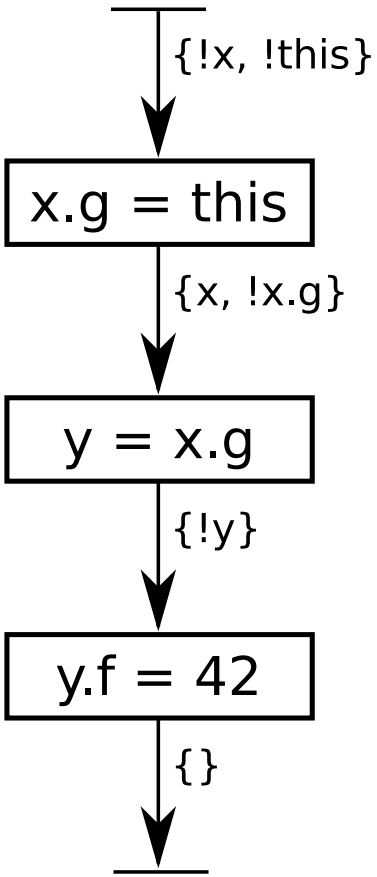


How Does This Work?

Backwards program analysis propogates locks to start of block

Source	CFG	Target

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

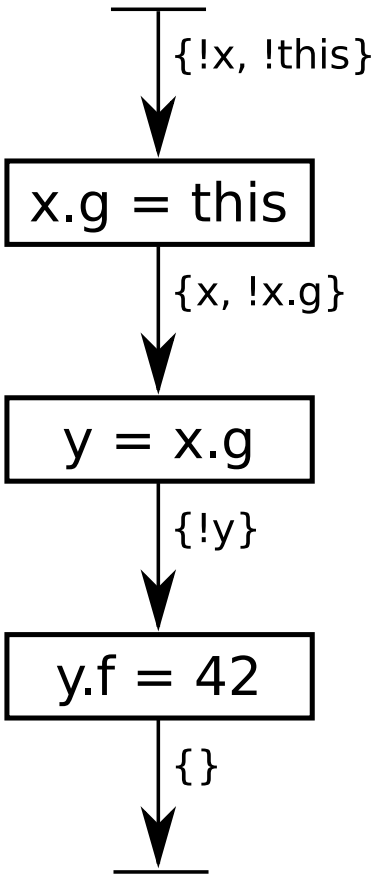


How Does This Work?

Backwards program analysis propogates locks to start of block

Source	CFG	Target

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




```
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) {
        vehicle = car;
    } else {
        vehicle = van;
    }
    vehicle.fuel = 42;
}
```

Target

```
lock(car,van);
if (b) {
    unlock(van);
    vehicle = car;
} else {
    unlock(car);
    vehicle = van;
}
vehicle.fuel = 42;
unlock(vehicle);
```

Example 9 - While

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

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

```
//lock(x, x.n, x.n.n, ...);
while (x.n) {
    x = x.n;
}
x.f = 42;
```

Example 9 - While

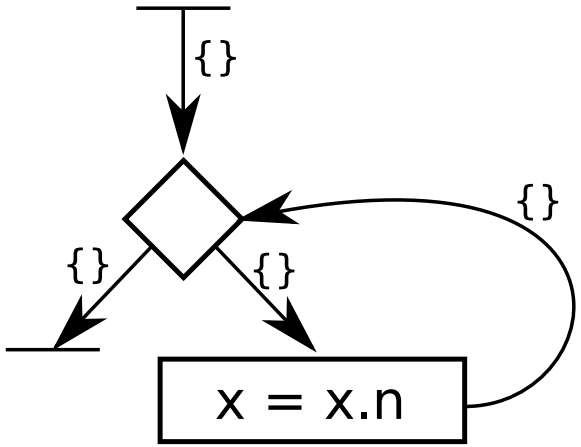
```
class Node {
    Node n;
    int f;
}
```

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

```
lock(Node);
while (x.n) {
    x = x.n;
}
lock(x);
unlock(Node);
x.f = 42;
unlock(x);
```

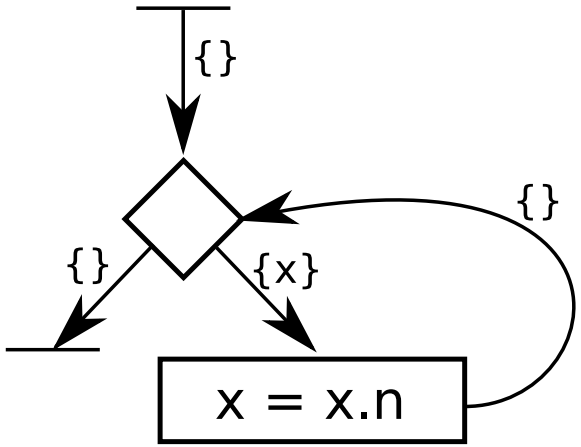
How does it work?

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



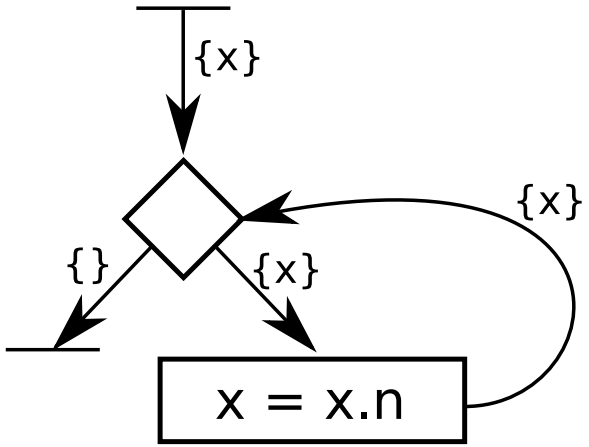
How does it work?

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



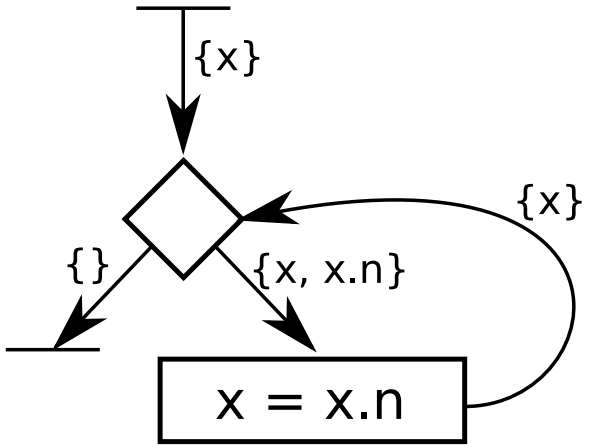
How does it work?

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



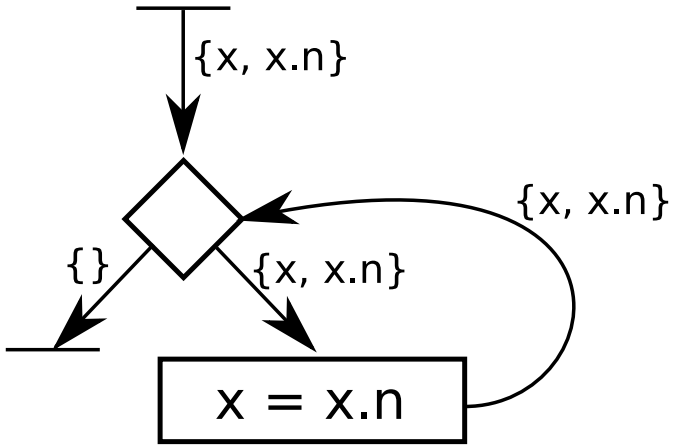
How does it work?

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



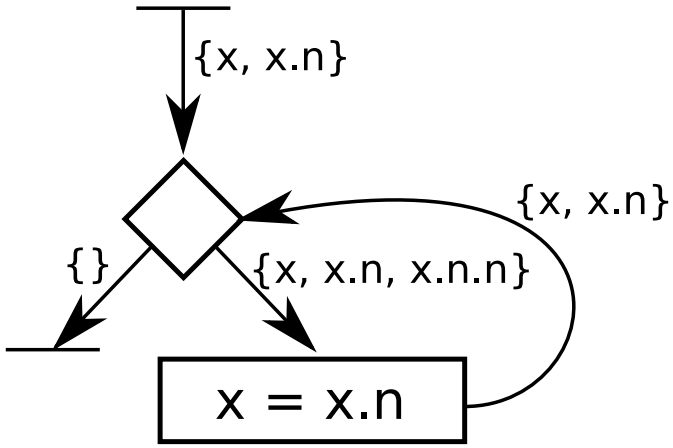
How does it work?

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



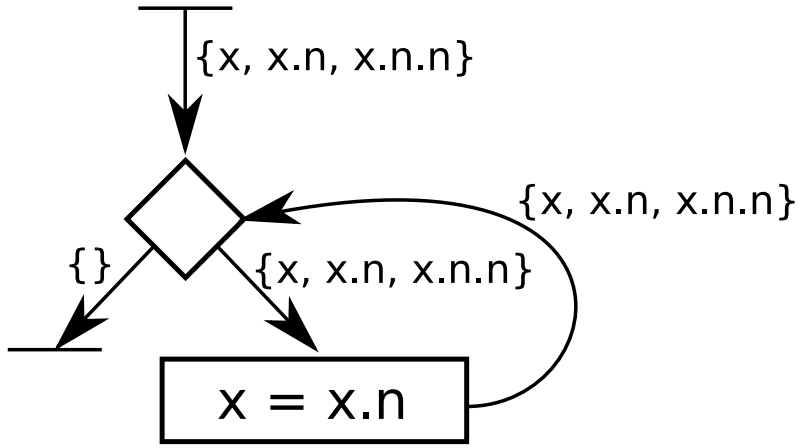
How does it work?

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



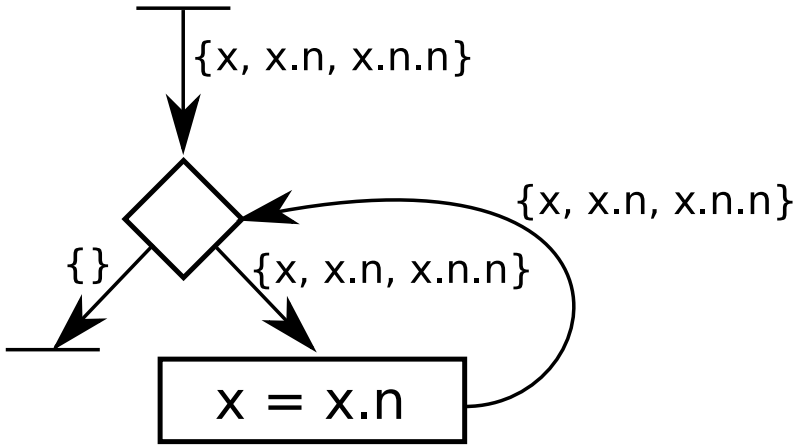
How does it work?

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



How does it work?

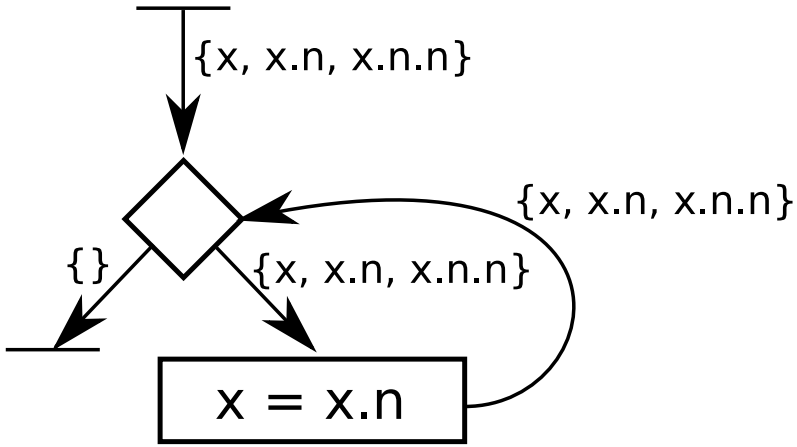
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



Analysis doesn't terminate :(

How does it work?

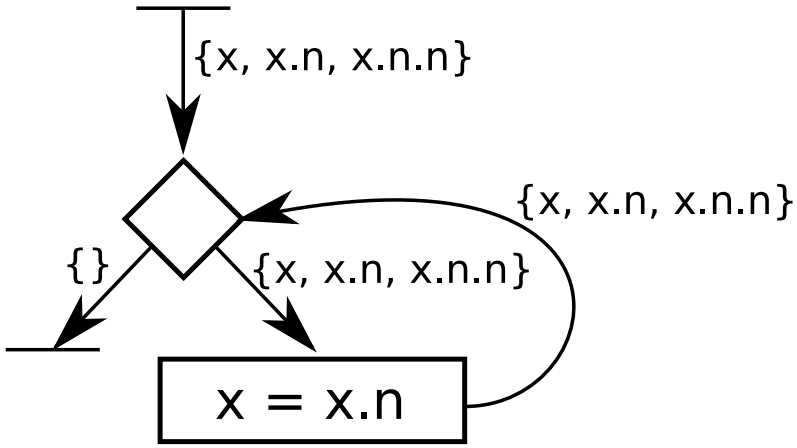
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



How do we solve this?

How does it work?

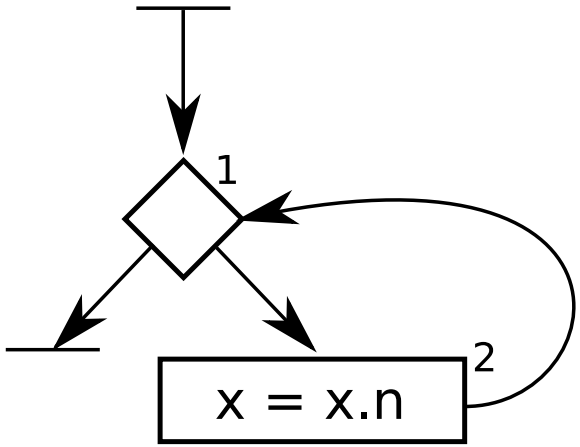
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



First, number the CFG nodes...

How does it work?

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



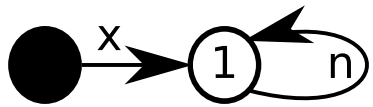
First, number the CFG nodes...

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 easily represent infinite sets of locks:



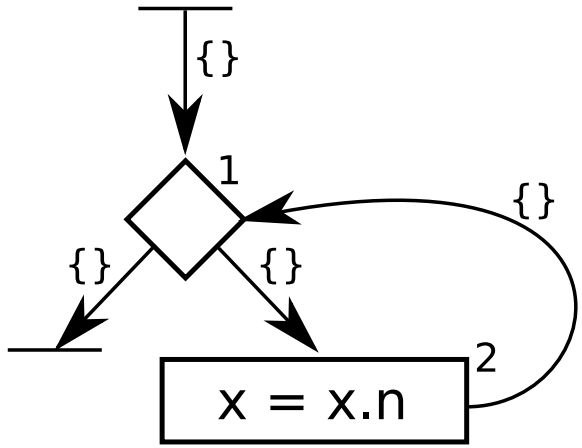
More formally, a set of edges (doubles and triples): $\{x \mapsto 1, 1 \rightarrow^n 1\}$

We constrain the set of automata nodes to the set of CFG nodes!

How does it work?

NFAs avoid the infinite loop:

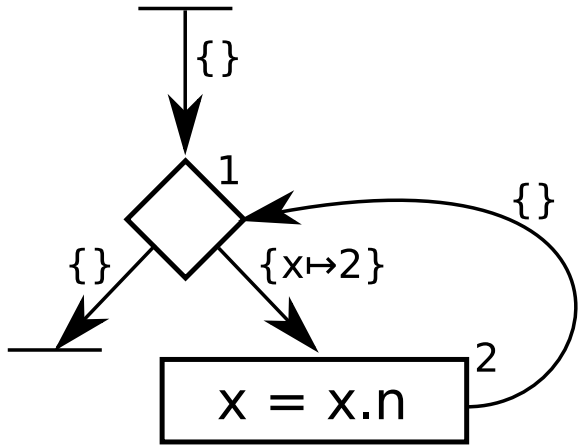
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

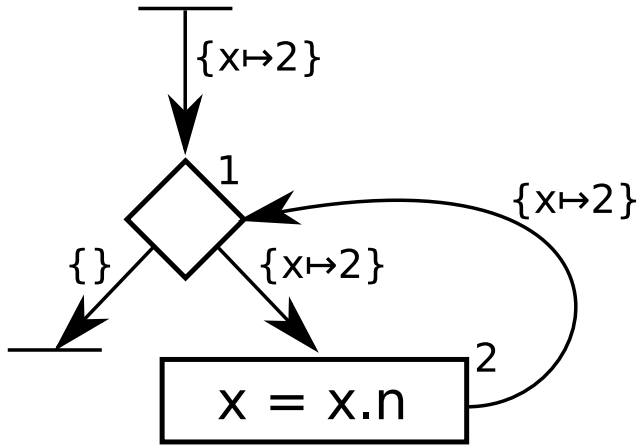
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

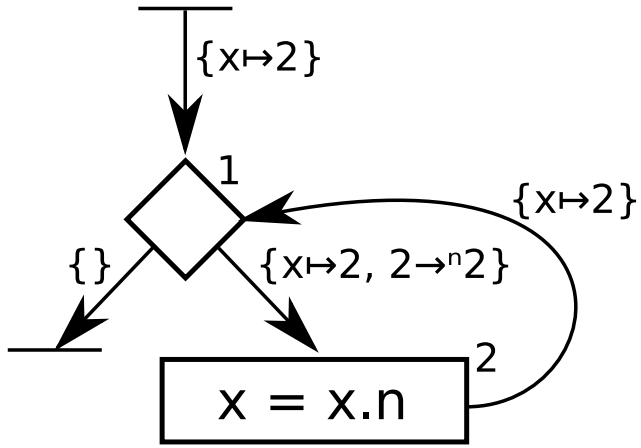
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

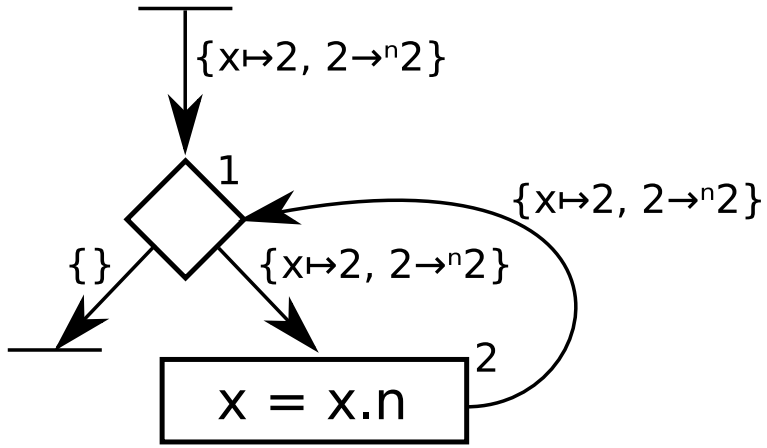
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



How does it work?

NFAs avoid the infinite loop:

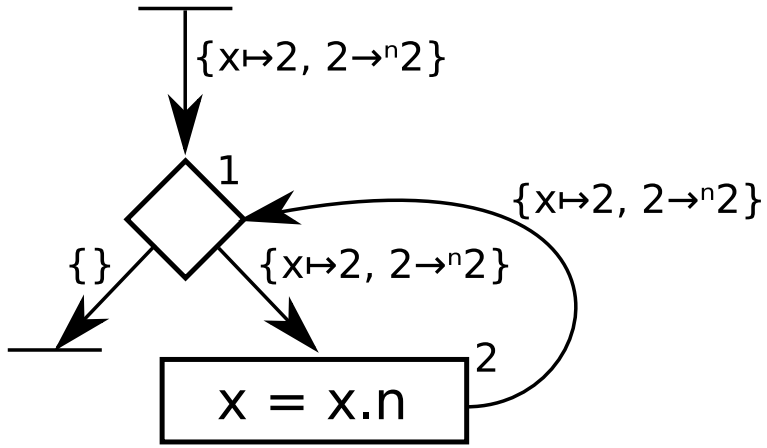
```
atomic {
    while (x.n) {
        x = x.n;
    }
}
```



How does it work?

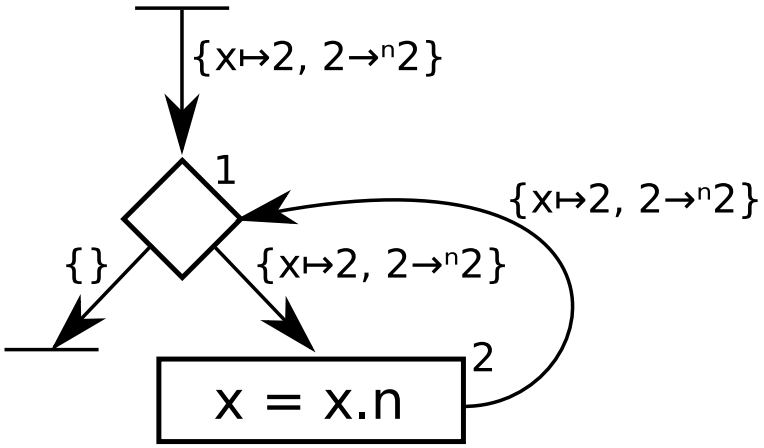
NFAs avoid the infinite loop:

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

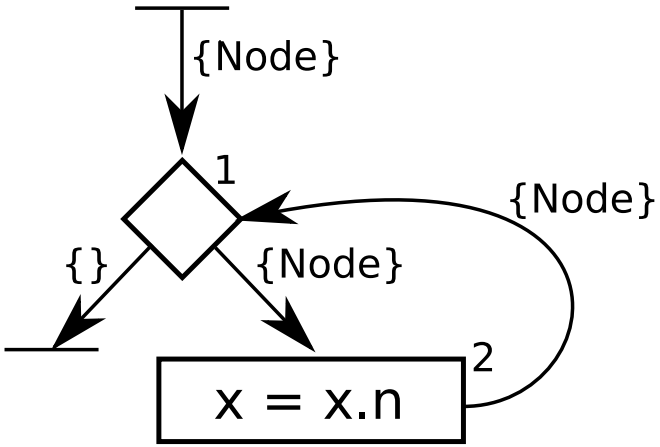


Analysis now terminates.

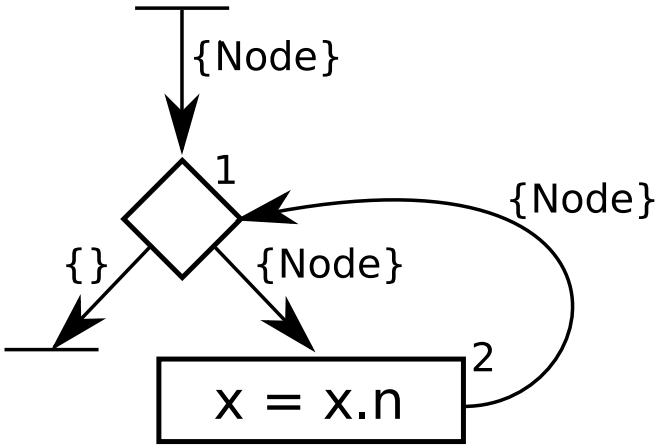
How Do We Lock an NFA?



How Do We Lock an NFA?



How Do We Lock an NFA?

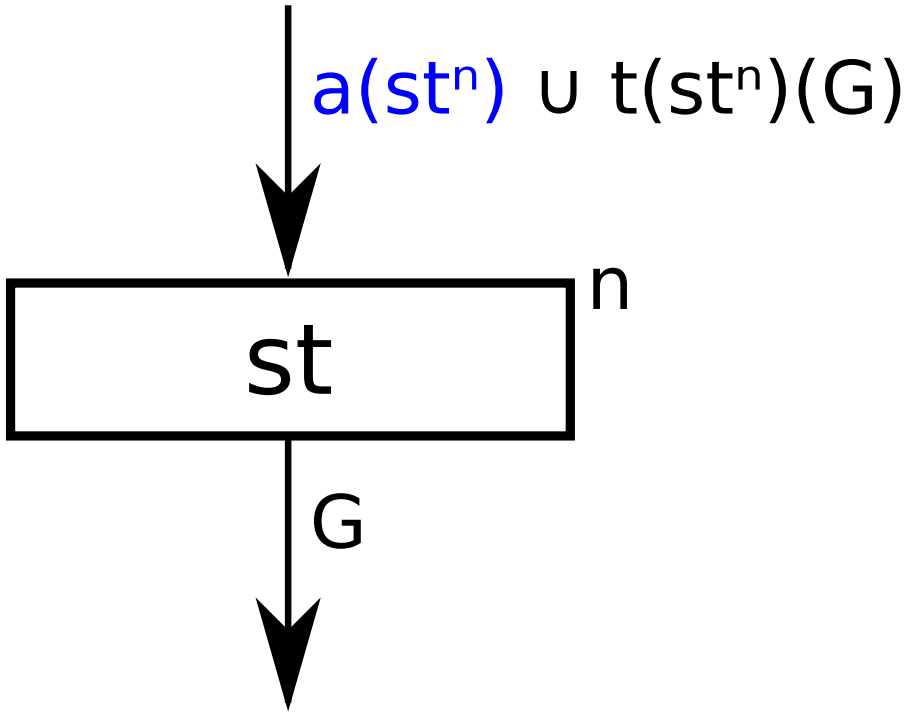



```
lockr(Node);
while (...) {
        x = x.n;
}
unlockr(Node);
```

The Transfer Functions

Addition function introduces new accesses into the CFG

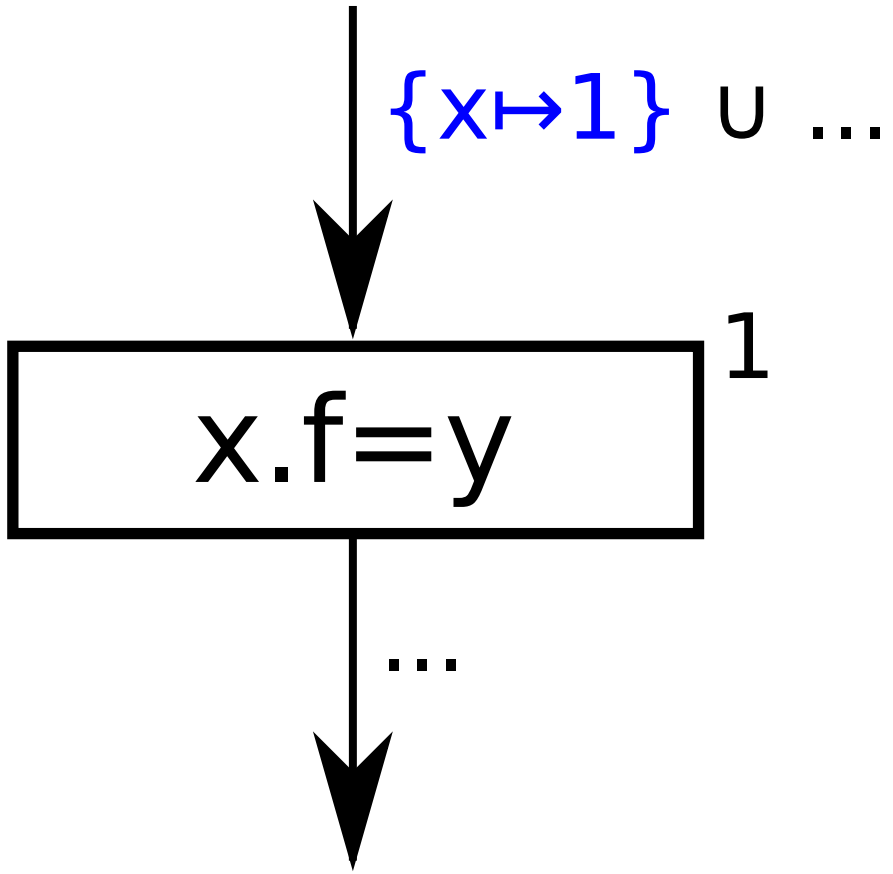
Translation function translates NFAs to compensate for state change



Addition function

(introduces new accesses into the CFG)

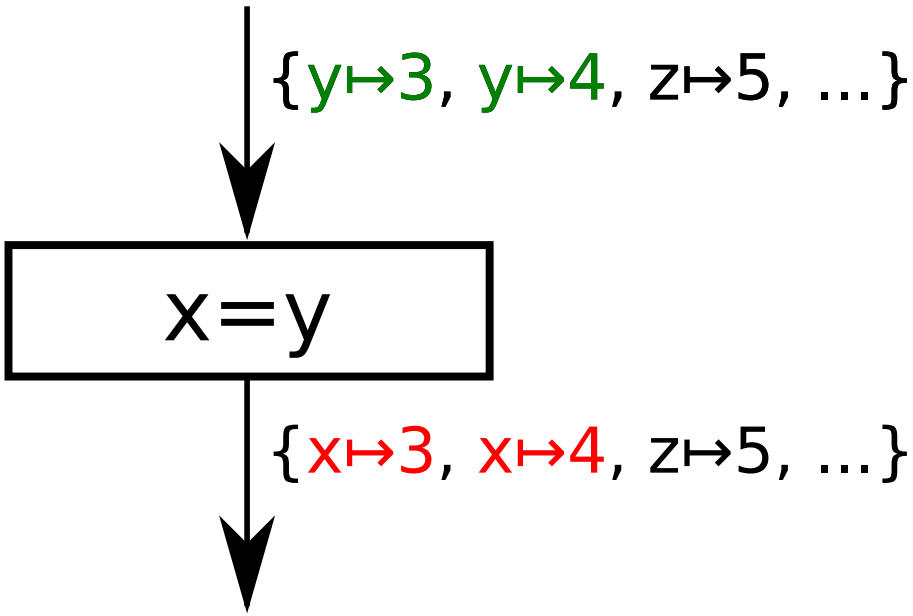
$$
\begin{aligned}
a[x = y]^n &= \emptyset \\
a[x = \texttt{null}]^n &= \emptyset \\
a[x = \texttt{new}]^n &= \emptyset \\
a[x = y.f]^n &= \{y \mapsto n\} \\
a[x.f = y]^n &= \{x \mapsto n\}
\end{aligned}
$$



Translation Function (1)

A standard kill/gen function

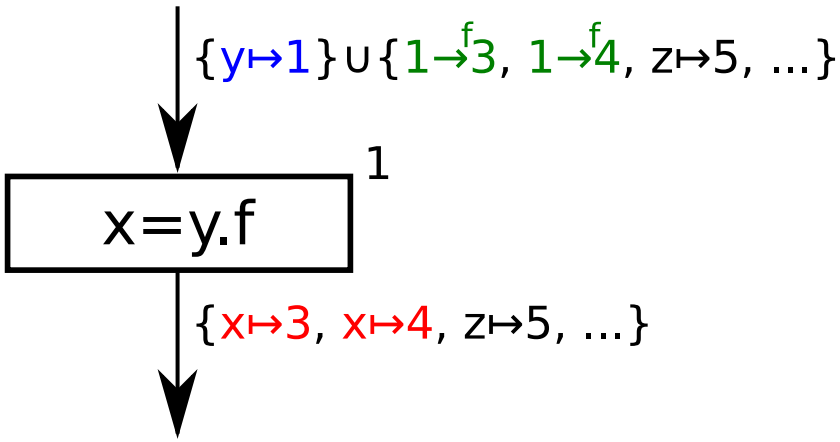
$$
\begin{aligned}
t[x = y]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\} \cup \{y \mapsto n' | x \mapsto n' \in G\} \\
t[x = \texttt{null}]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\} \\
t[x = \texttt{new}]^n(G) &= G \setminus \{x \mapsto n' | x \mapsto n' \in G\}
\end{aligned}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

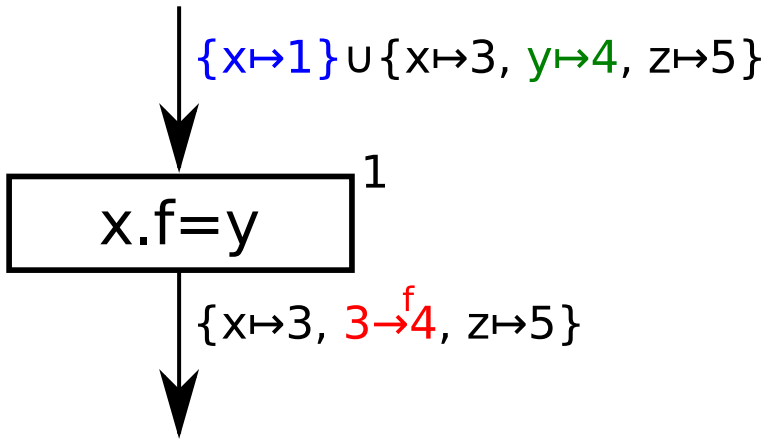


Translation Function (2)

$$t[x = y.f]^n(G) = G \setminus \{x \mapsto n' | x \mapsto n' \in G\} \cup \{n \rightarrow^f n' | x \mapsto n' \in G\}$$

$$t[x.f = y]^n(G) = G \setminus \{n' \rightarrow^f \_ | x \mapsto n' \in G, (\nexists z \neq x : z \mapsto n' \in G), (\nexists n''' : n''' \rightarrow_- n' \in G)\} \cup \{y \mapsto n' | \_ \rightarrow^f n' \in G\}$$





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

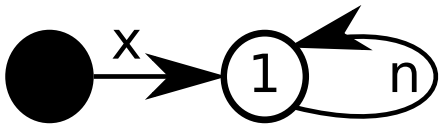
Soundness:

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

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
$h, \sigma \vdash G : \varphi'$

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

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
$h, \sigma \vdash G : \varphi'$

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

$squash(\varphi)$ gets the addresses from φ

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.

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

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

$$\frac{P(n) = [x = y, n'] \quad P \vdash h, \sigma[x \mapsto \sigma(y)], n' \overset{A}{\rightsquigarrow}^*}{P \vdash h, \sigma, n \overset{A}{\rightsquigarrow}^*}$$

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:

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

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!

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:

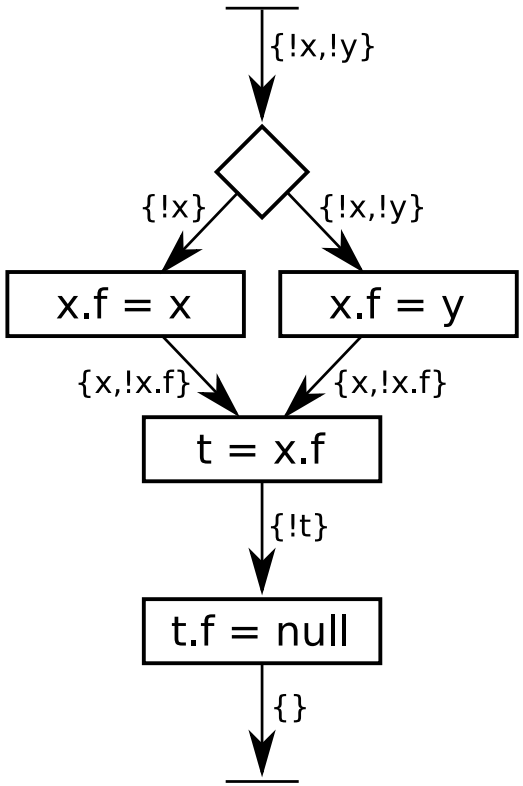
- Better precision (ownership types?)
- Better runtime performance
- Better compiletime performance (JIT possible?)
- Nested atomicity would be nice
- Thread-local type system

Questions

Balancing Example

Source	CFG	Target

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




```
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);
```