Lock Inference Proven Correct

Dave Cunningham   Sophia Drossopoulou   Susan Eisenbach

Imperial College London

FTfJP 2008

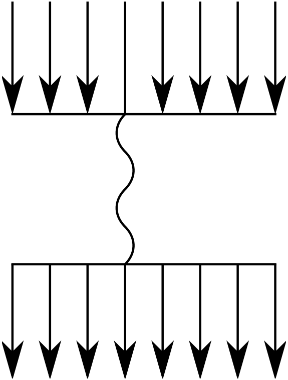
Why Atomic sections

Example:

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

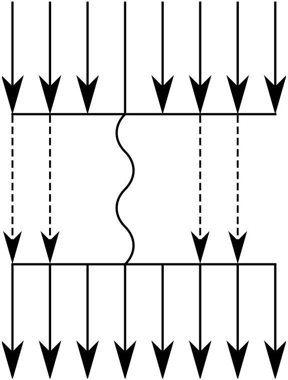
- Semantics easy for programmers to understand
    - Guaranteed that threads don't interfere
- Concurrency much easier
- Naive implementation is inefficient
- Lots of research tries to interleave more threads (which is hard)

Why Lock Inference?



One thread in an atomic section.

Why Lock Inference?



One thread in an atomic section.
Non-interfering threads allowed to proceed.

Our Algorithm

In CC'08 we published an algorithm that compiles atomic sections:

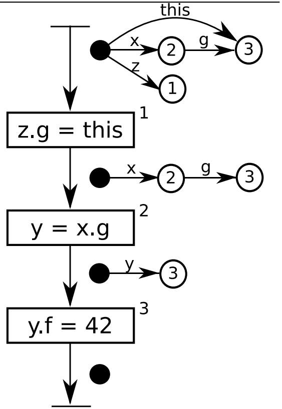
Source

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

Target

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

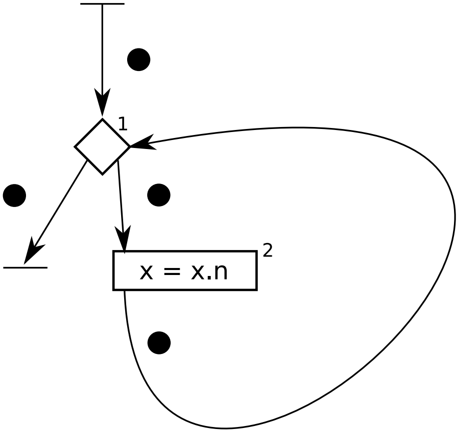
Analysis



While Loops

NFAs allow iterations:

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

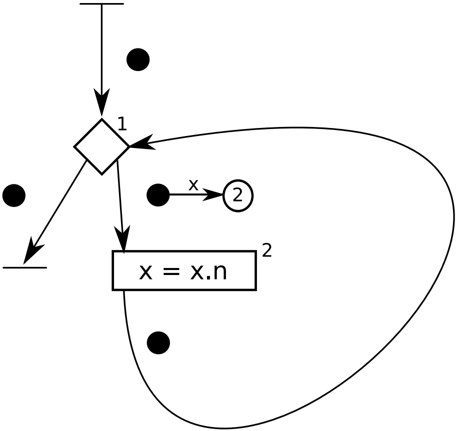


Termination is due to the use of CFG nodes as NFA states.

While Loops

NFAs allow iterations:

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

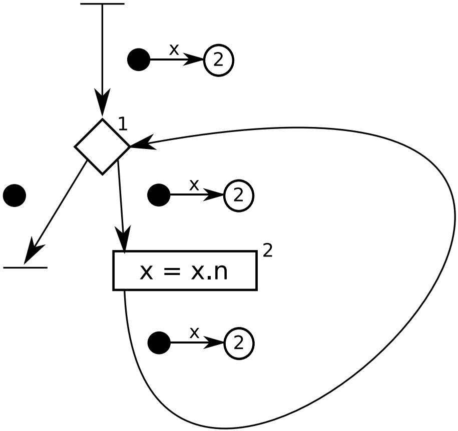


Termination is due to the use of CFG nodes as NFA states.

While Loops

NFAs allow iterations:

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

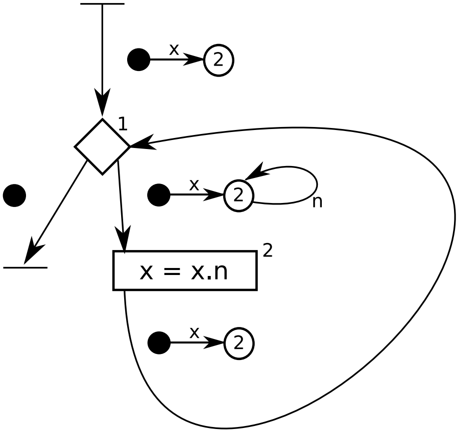


Termination is due to the use of CFG nodes as NFA states.

While Loops

NFAs allow iterations:

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

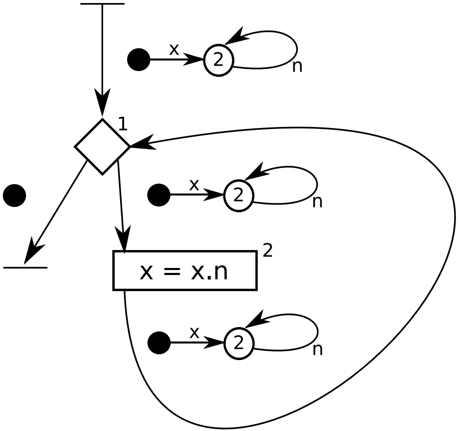


Termination is due to the use of CFG nodes as NFA states.

While Loops

NFAs allow iterations:

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



Termination is due to the use of CFG nodes as NFA states.

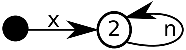
In This Paper We Prove Soundness

Our approach:

- Assume the two-phase locking discipline is sound
- Don't have to worry about concurrency at all!
- Prove analysis correctly infers the objects accessed

In this talk, I will:

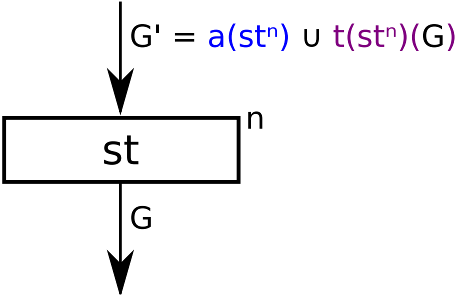
- Show analysis in more detail
- Formalise the meaning of the NFAs:



- Show soundness theorem

The Transfer Functions

How to formalise the analysis:

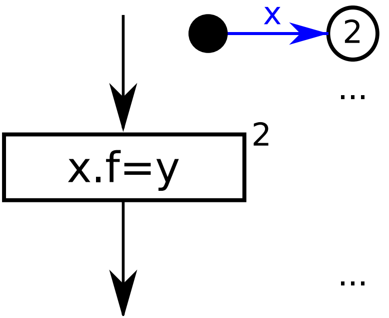


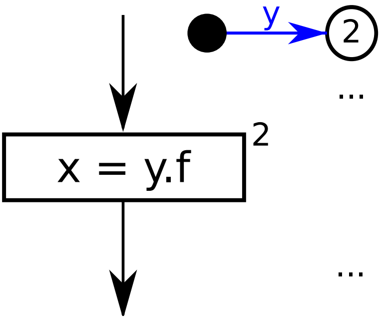
Addition function a(st^{n}) inserts the accesses performed by **st**

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

Addition Function

(introduces new accesses into the CFG)

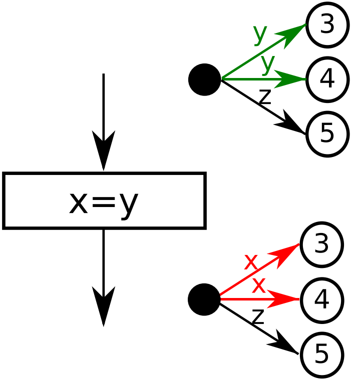




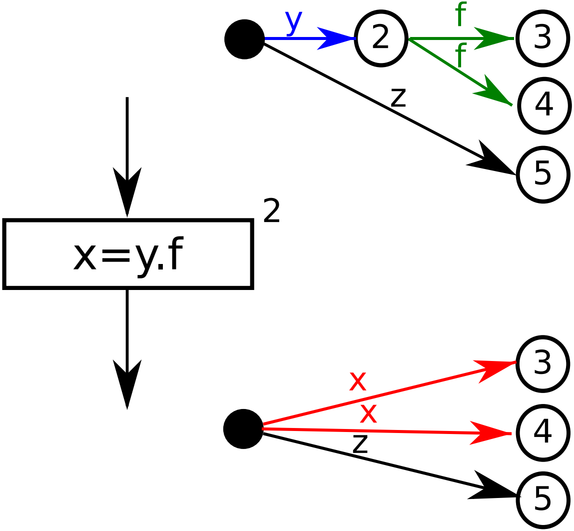
Translation Function (copy)

A standard kill/gen function

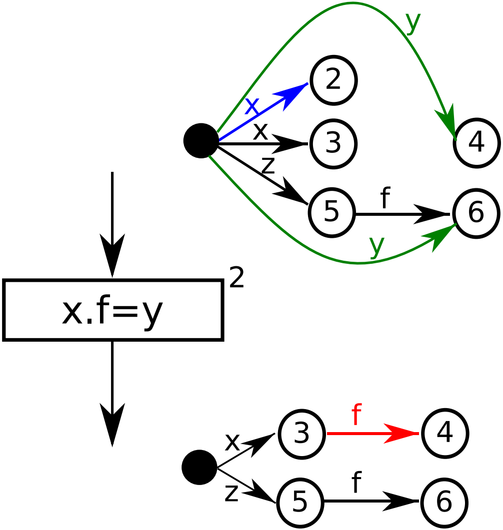
Translate the accesses to balance the effect of the statement:



Translation Function (load)



Translation Function (store)

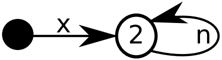


What is soundness?

Does the top NFA safely approximate the addresses accessed?

- Let execution start from the top of the atomic section.
- Let A be the addresses accessed (define a semantics for this)
- Let G be the NFA from the analysis

Want to show $A \subseteq G$
But we have no link between static world G and dynamic world A.



- Static characterisation of a set of objects.
- Easily represent infinitely many accesses.

Linking NFAs to addresses

An **assignment** φ interprets G in a stack,heap.
Assignment φ stores a set of addresses at each NFA state

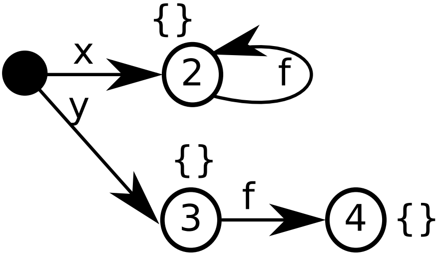
stack

var	addr
x	10
y	30

heap

addr	field f
10	20
20	10
30	40

NFA



Linking NFAs to addresses

An **assignment** φ interprets G in a stack,heap.
Assignment φ stores a set of addresses at each NFA state

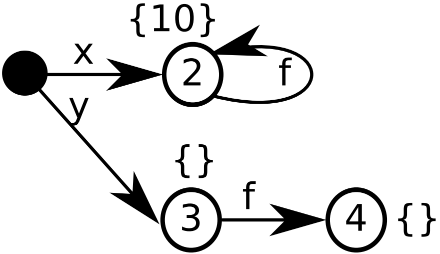
stack

var	addr
x	10
y	30

heap

addr	field f
10	20
20	10
30	40

NFA



Linking NFAs to addresses

An **assignment** φ interprets G in a stack,heap.
Assignment φ stores a set of addresses at each NFA state

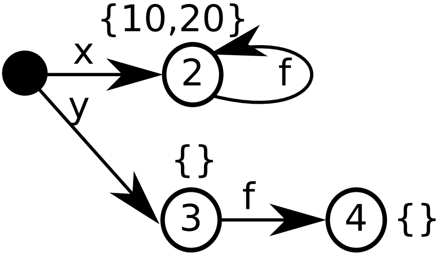
stack

var	addr
x	10
y	30

heap

addr	field f
10	20
20	10
30	40

NFA



Linking NFAs to addresses

An **assignment** φ interprets G in a stack,heap.
Assignment φ stores a set of addresses at each NFA state

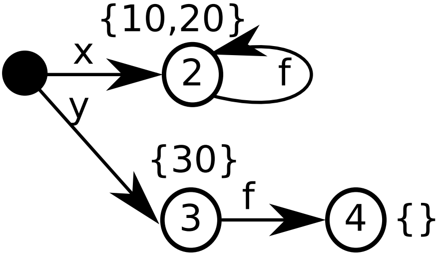
stack

var	addr
x	10
y	30

heap

addr	field f
10	20
20	10
30	40

NFA



Linking NFAs to addresses

An **assignment** φ interprets G in a stack,heap.
Assignment φ stores a set of addresses at each NFA state

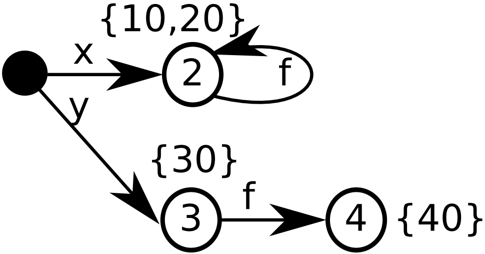
stack

var	addr
x	10
y	30

heap

addr	field f
10	20
20	10
30	40

NFA



Token Maths Slide

We can represent the NFAs as e.g.:

$$G = \{x \mapsto 2, 2 \rightarrow^{f} 2, y \mapsto 3, 3 \rightarrow^{f} 4\}$$

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

i.e. iff

$$x \mapsto n \in G \Rightarrow \sigma(x) \in \varphi(n)$$

$$n \rightarrow^{f} n' \in G \Rightarrow \{h(a)(f) | a \in \varphi(n)\} \subseteq \varphi(n')$$

Soundness Revisited

Now we can define soundness :

If:

- G is the NFA returned by the analysis
- h, σ is the initial heap, stack
- A is the addresses accessed (operational semantics in paper)
- φ is the addresses at each node of G $\quad h, \sigma \vdash G : \varphi$

then we must have $A \subseteq \mathit{squash}(\varphi)$

Soundness Proved

We used 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
- Well-formed induction using length of A
- $\sim$ 800 lines (including definitions)
- $\sim$ 30 seconds for proofgeneral to verify on 3Ghz P4
- Proof assistants are cool!

Conclusions

Proved soundness of our lock inference algorithm:

- Use known facts of two-phase discipline
- Use transfer functions to formalise analysis
- Use operational semantics to formalise execution
- Assignments (φ) were the missing link
- Mechanically checked proof

Further work:

- Prove early unlocking
- Prove readers/writers
- Prove arrays, functions, exceptions, etc.
- Improve underlying analysis

The End

Thankyou!

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
Our work	Multigrain	Yes	Dynamic	Yes
Cherem08	Multigrain	Yes	Static?	No

Key: v.good, good, OK, bad

Two ways of safely interleaving more threads

	Transactional Memory	**Lock Inference**
I/O	Hard	Easy
Reflection	Easy	Need JIT support
Native calls	Hard	Hard
Compiler machinery	Some	Lots
Runtime machinery	Lots	Some
Performance	Slow	Fast
Granularity	Perfect	Reasonable

Key: good, OK, bad

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.

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

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

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