Improving the performance of Atomic Sections

Khilan Gudka

Supervised by:
Prof. Susan Eisenbach

Background

- The multi-core revolution has made concurrency a hot topic
- Programmers are now forced to think about it for performance
- But shared memory concurrency is hard!

Where we are: we use locks

- Problems
  - Not composable
  - Introduce deadlock
  - Break modularity
  - Other problems: priority inversion, convoying, starvation...

Atomic sections

- What programmers probably can do is tell which parts of their program should not involve interferences
- Atomic sections [Lomet77]
  - Declarative concurrency control
  - Move responsibility for figuring out what to do to the compiler/runtime

```
atomic {
    ... access shared state ...
}
```

Atomic sections

- Simple semantics (no interference allowed)
- Naive implementation: one global lock
- But we want to allow parallelism without:
  - Interference
  - Deadlock

Transactional memory

- Very hot research area - lots of papers!
  [For review of work up until 2006, see Larus06]
- Advantages
  - No problems associated with locks
  - More concurrency
- Disadvantages
  - Irreversible operations (IO, System calls)
  - Run-time overhead

Lock inference

- Statically infer the locks that are needed to protect shared accesses
- Insert lock()/unlock() statements for them into the program to ensure atomic execution

```
atomic {
  x.f = 10;
}
```

→

```
synchronized(x) {
  x.f = 10;
}
```

Lock inference

- Challenges
  - Maximise concurrency
  - Minimise locking overhead
  - Avoid deadlock

Restriction for atomicity: Two-phase locking

```
atomic {
    ...
    lock(A);
    ...
    lock(B);
    ...
    unlock(B);
    ...
    unlock(A);
    ...
}
```

Correct

```
atomic {
    ...
    lock(A);
    ...
    unlock(A);
    ...
    lock(B);
    ...
    unlock(B);
    ...
}
```

Wrong

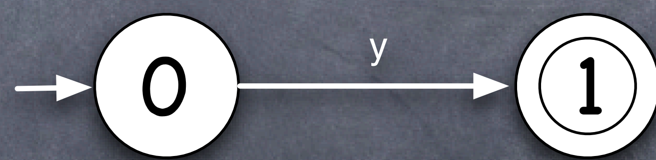
Locking granularity

- To maximise parallelism, locks should be as fine-grained as possible
- The granularity of locks depends on the compile-time representation of objects
- Lvalues (e.g. x.f) allow per-instance locks when each object has its own lock (e.g. Java)
- During my masters, we developed an analysis to infer lvalues and it was published in CC'08 [Cunningham08]

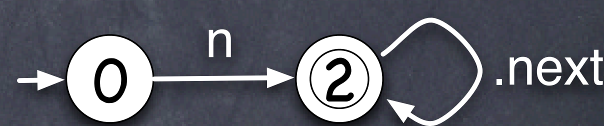
Finite State Automata

- A compact compile-time object representation
- Represents a possibly infinite set of lvalues
- Our analysis flows automata around the CFG

{ y } =



{ n, n.next, n.next.next, ... } =

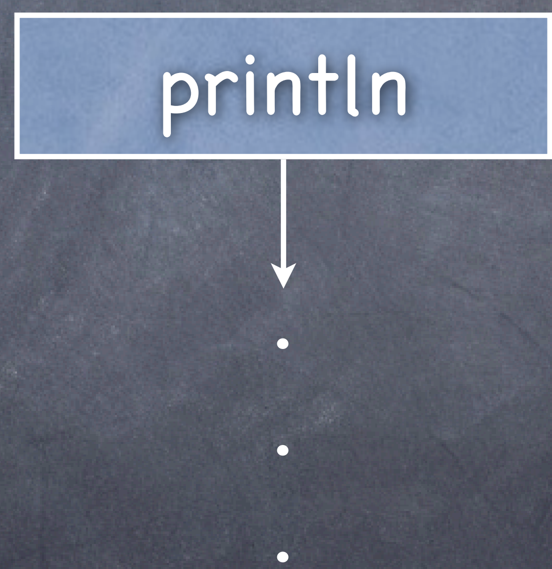


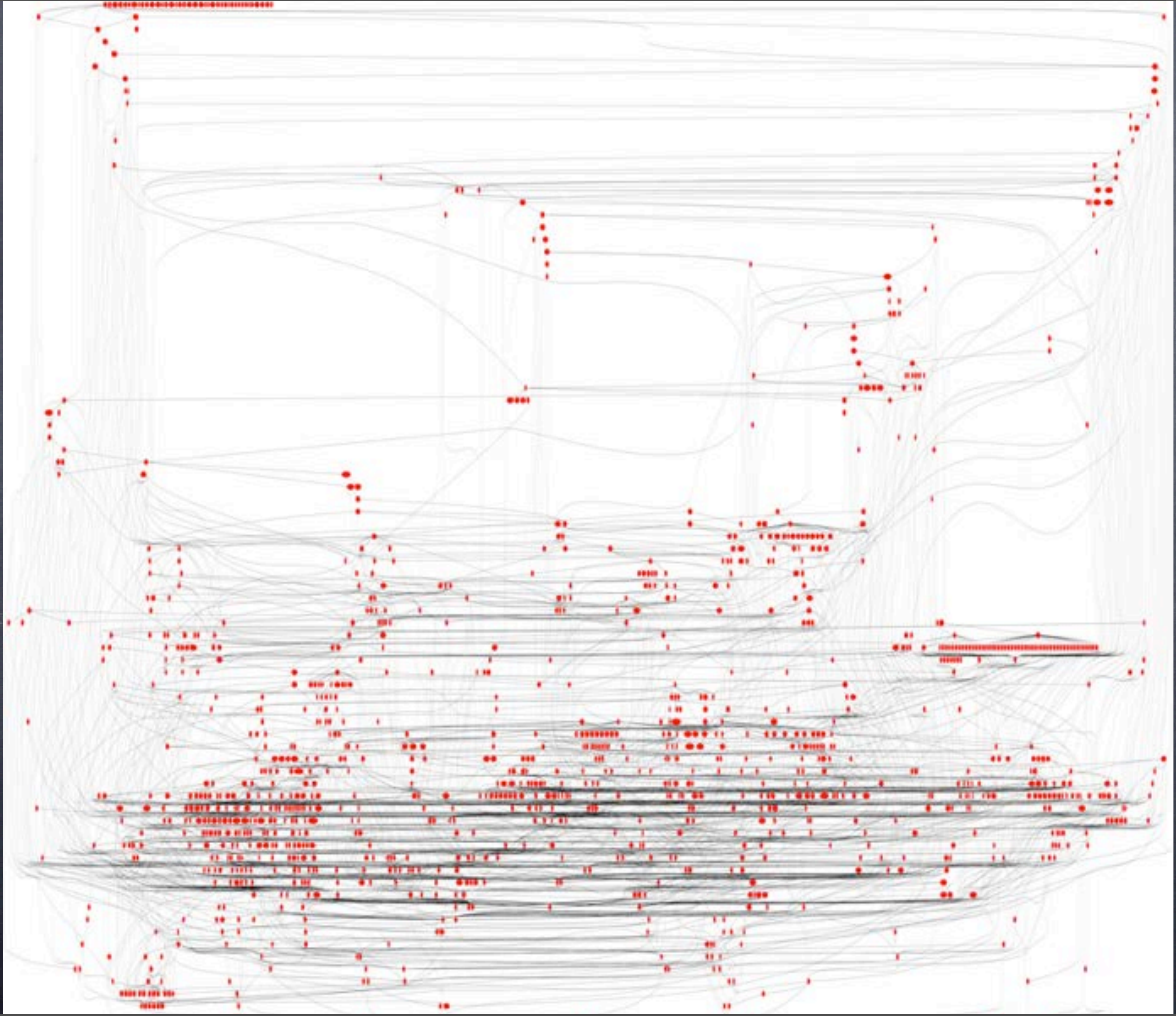
Scaling to Java: "Hello world"

```
atomic {
    System.out.println("Hello World");
}
```

Scaling to Java: "Hello world"

Call graph:

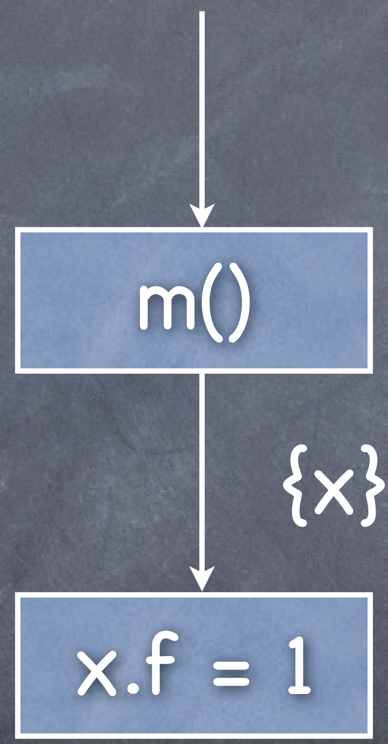




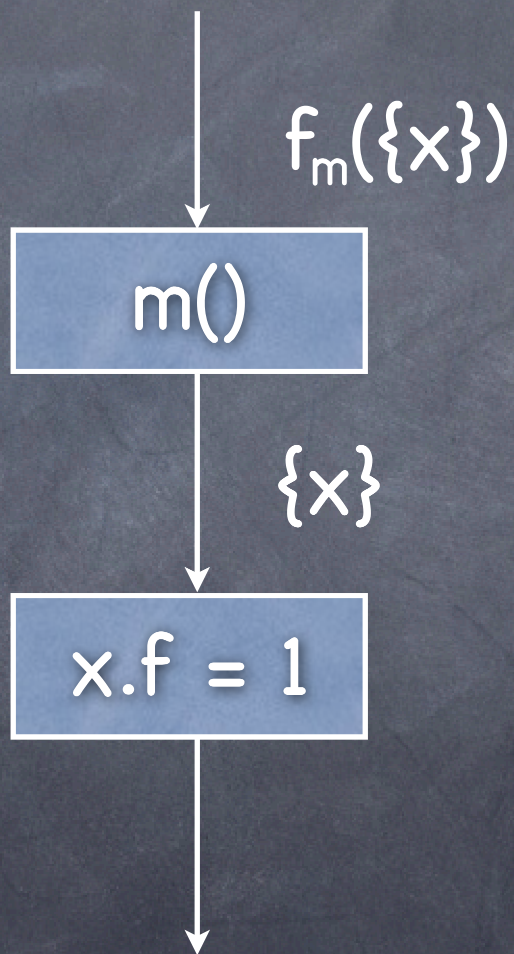
Scaling to Java: "Hello world"

- This work doesn't scale
- We switch to computing summaries
- A summary is a function that describes how a method as a whole translates dataflow information
- Summaries are also context-sensitive but can scale better

Method summaries



Method summaries



f_m is m's summary

Computing summaries

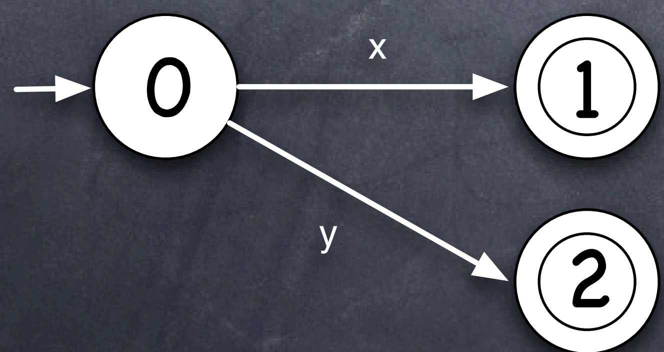
- Define, for each statement, transfer functions describing how they translate dataflow information
- Compose them into one large transfer function for the entire method by flowing them through the CFG using a normal dataflow analysis
- Summaries can get large: challenge is to find a representation of transfer functions that allows fast composition and meet operations

IDE Analyses

- Interprocedural Distributive Environment [Sagiv96]
- Dataflow facts are functions of type D -> L, called environments
- Transfer functions are called environment transformers
- Advantage: efficient graph representation of environment transformers exists that allows fast composition and meet [Reps95,Sagiv96,Rountev08]

Reformulate our lvalue analysis

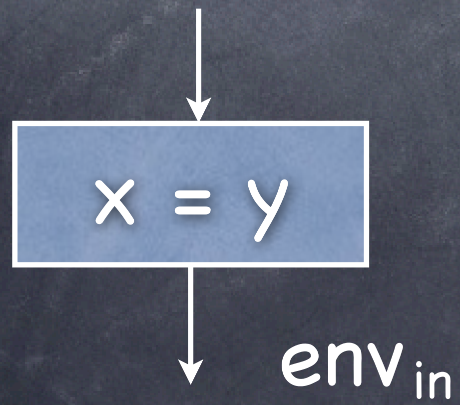
- Step 1: Express automata as environments (functions of type D -> L)
- We represent automata as functions from transition labels to sets of pairs of states (of the transitions for those labels)



[x -> { (0,1) },
y -> { (0,2) }]

Environment transformers

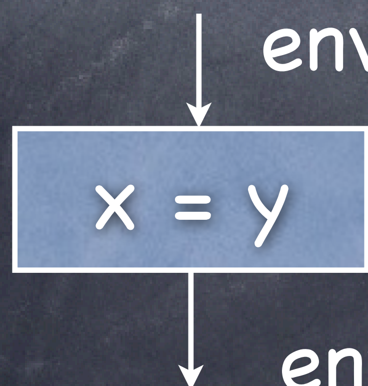
- Step 2: Define environment transformers (i.e. the transfer functions)
- They describe how the 'outgoing' environment is computed from the 'incoming' environment



Environment transformers

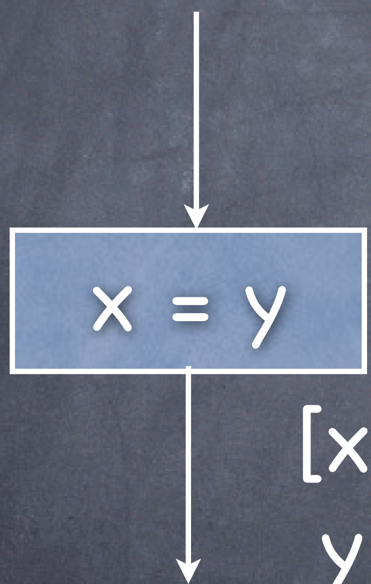
- Step 2: Define environment transformers (i.e. the transfer functions)
- They describe how the 'outgoing' environment is computed from the 'incoming' environment

$$env_{out} = t_{[x=y]}(env_{in})$$



env_{in}

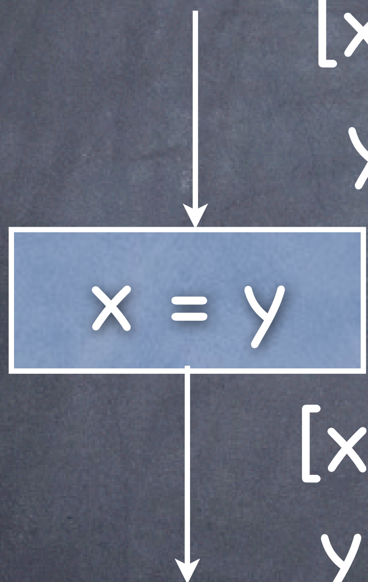
Environment transformers



[x -> { (0,1) },
 y -> { (0,2) }]

Environment transformers

[x -> ∅
 y -> { (0,2), (0,1) }]



[x -> { (0,1) },
 y -> { (0,2) }]

Environment transformers

[x -> ∅

y -> { (0,2), (0,1) }]

x = y

[x -> { (0,1) },

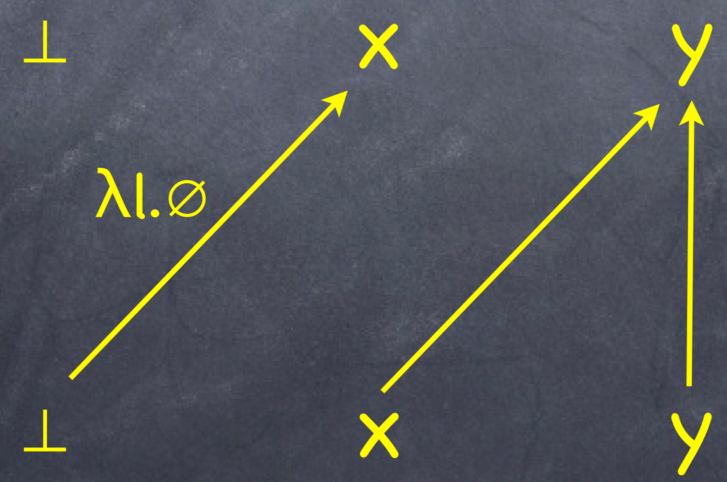
y -> { (0,2) }]

$t_{[x=y]} = \lambda e.e[y\text{->}e(y) \cup e(x)][x\text{->}\emptyset]$

Environment transformers (as in [Sagiv96])

- These transformers can be represented as graphs

$$t_{[x=y]} = \lambda e.e[y\text{->}e(y) \cup e(x)][x\text{->}\varnothing]$$



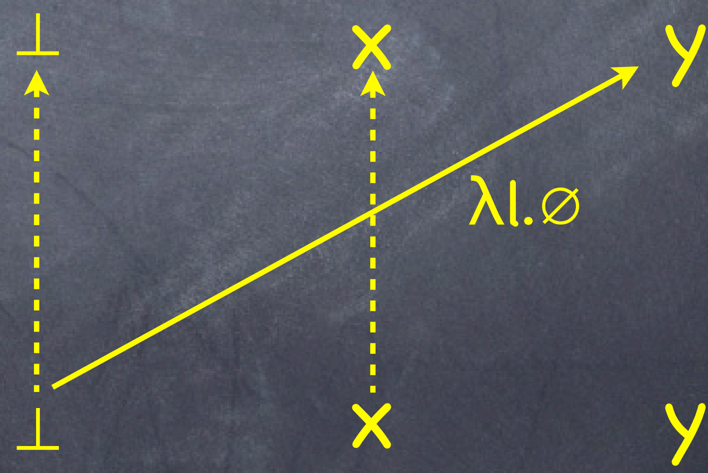
Environment transformers (as in [Sagiv96])

- Graphs are kept sparse by not explicitly representing obvious edges

[x -> { (0,1) }
 y -> ∅]

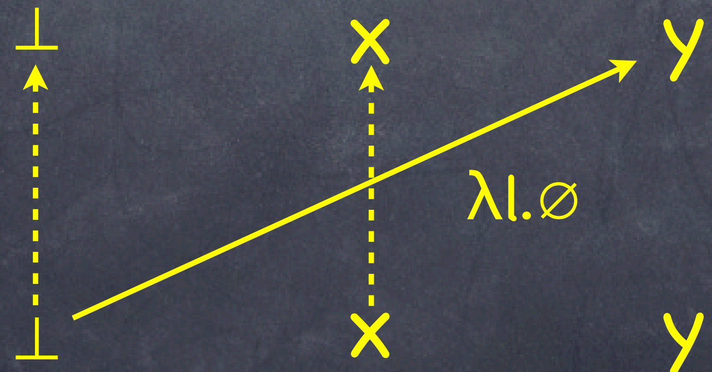
y = null

[x -> { (0,1) }
 y -> { (0,2) }]



Environment transformers (as in [Sagiv96])

- Transformer composition is simply the transitive closure
- Implicit edges should not have to be made explicit as that would be expensive
- But determining whether an implicit edge exists is costly in [Sagiv96] for our analysis



Environment transformers (Ours)

- We represent kills in transformers as:

$$x \longrightarrow \varnothing$$

$$\lambda e.e[x\text{->}\varnothing]$$

- Our lvalues analysis mostly rewrites lvalues, hence we change the meaning of transformer edges to pass on but also implicitly kill:

$$x \longrightarrow y$$

$$\lambda e.e[y\text{->}env(y) \cup env(x)][x\text{->}\varnothing]$$

- Result: implicit edge very easy to determine. This leads to fast transitive closure

Environment transformers (Ours)

- We represent kills in transformers as:

$$x \longrightarrow \varnothing$$

$$\lambda e.e[x\text{->}\varnothing]$$

Theirs

$$\perp \xrightarrow{\lambda l.\varnothing} x$$

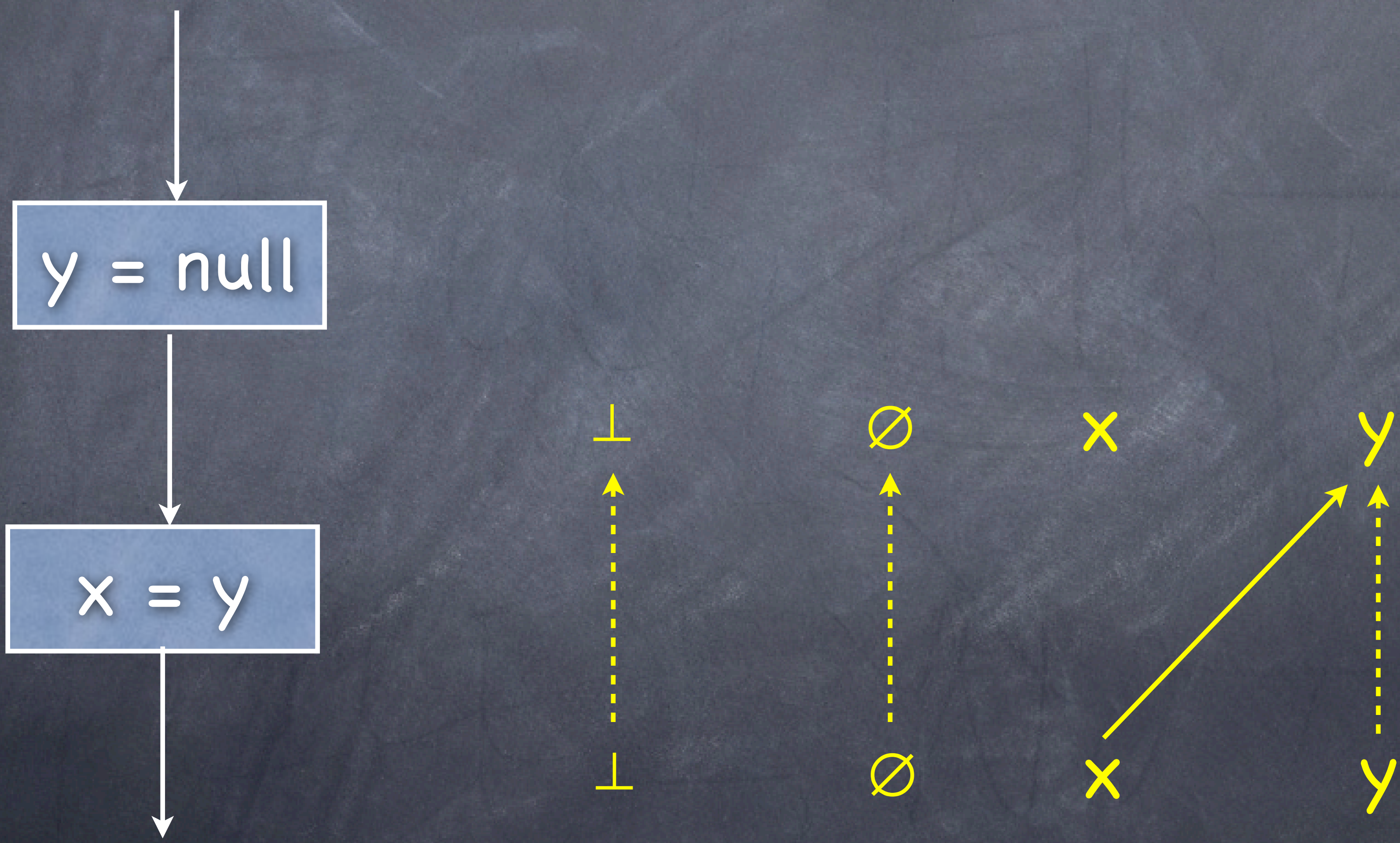
- Our lvalues analysis mostly rewrites lvalues, hence we change the meaning of transformer edges to pass on but also implicitly kill:

$$x \longrightarrow y$$

$$\lambda e.e[y\text{->}env(y) \cup env_{\perp}(x)][x\text{->}\varnothing]$$

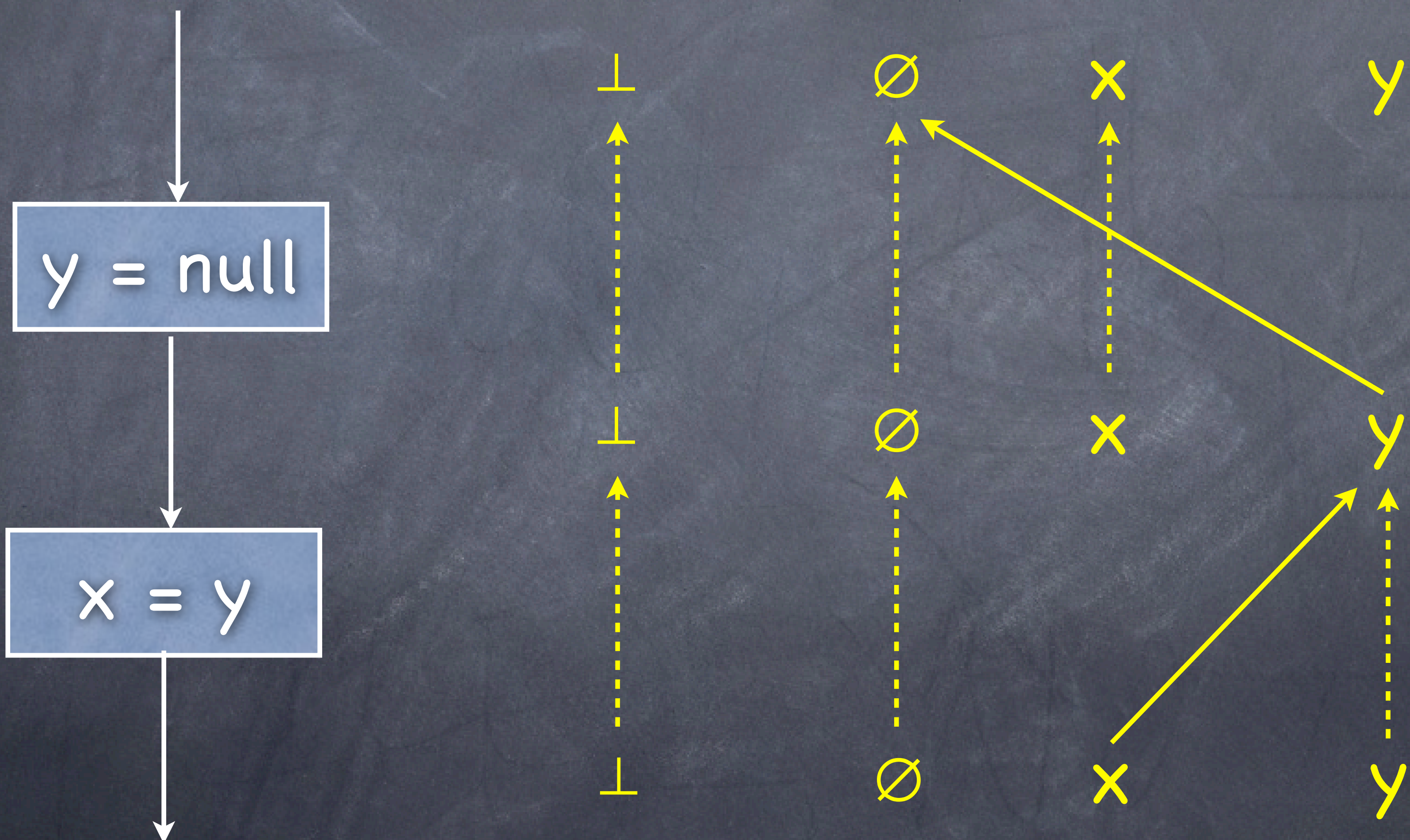
- Result: implicit edge very easy to determine. This leads to fast transitive closure

Environment transformers (Ours)

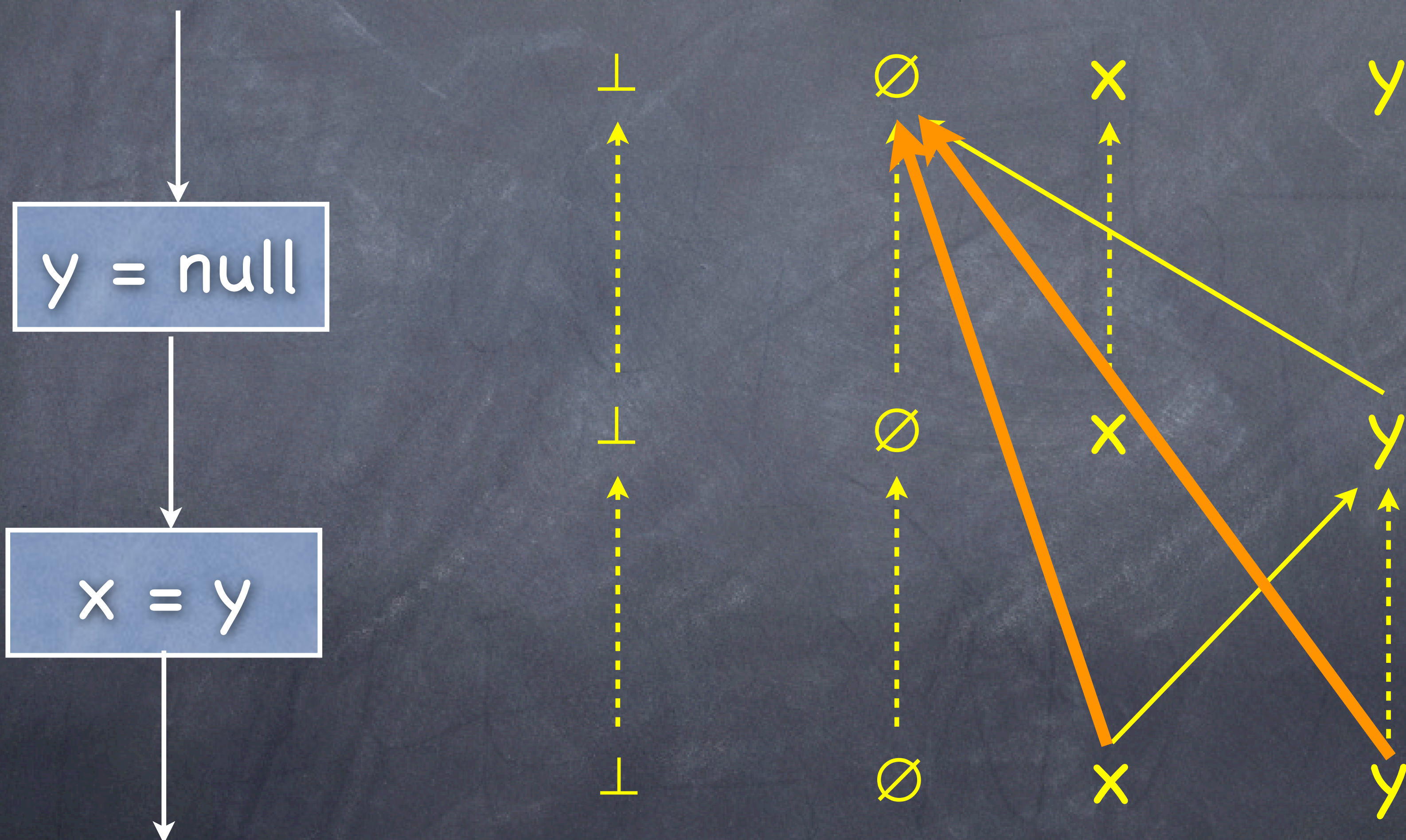


Tuesday, 16 June 2009

Environment transformers (Ours)



Environment transformers (Ours)



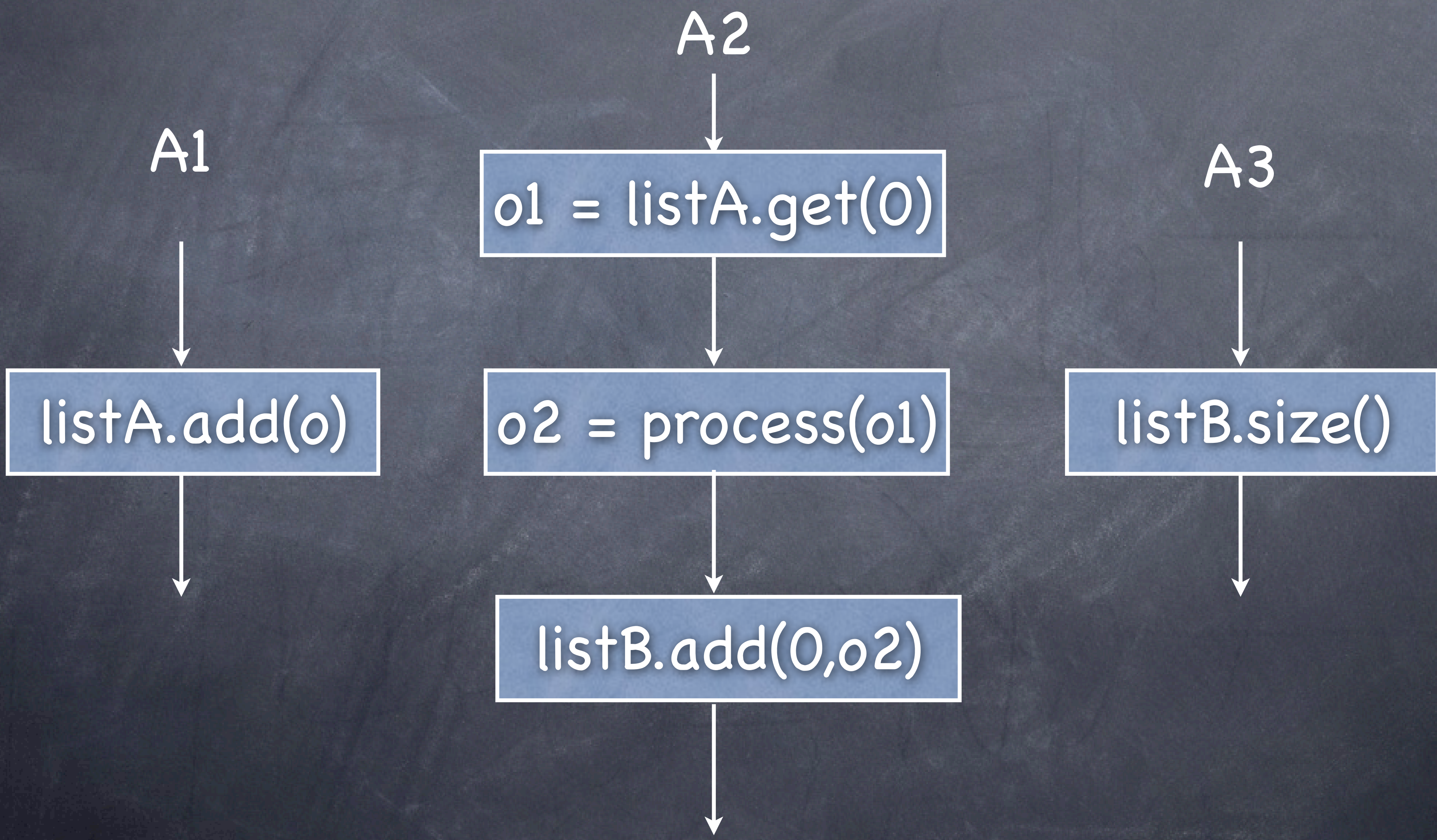
Implementation

- Implemented in the Soot bytecode analysis framework and am experimenting with small programs at present
- Implementation identifies strongly connected components (SCC) and propagates summaries up the SCC-DAG

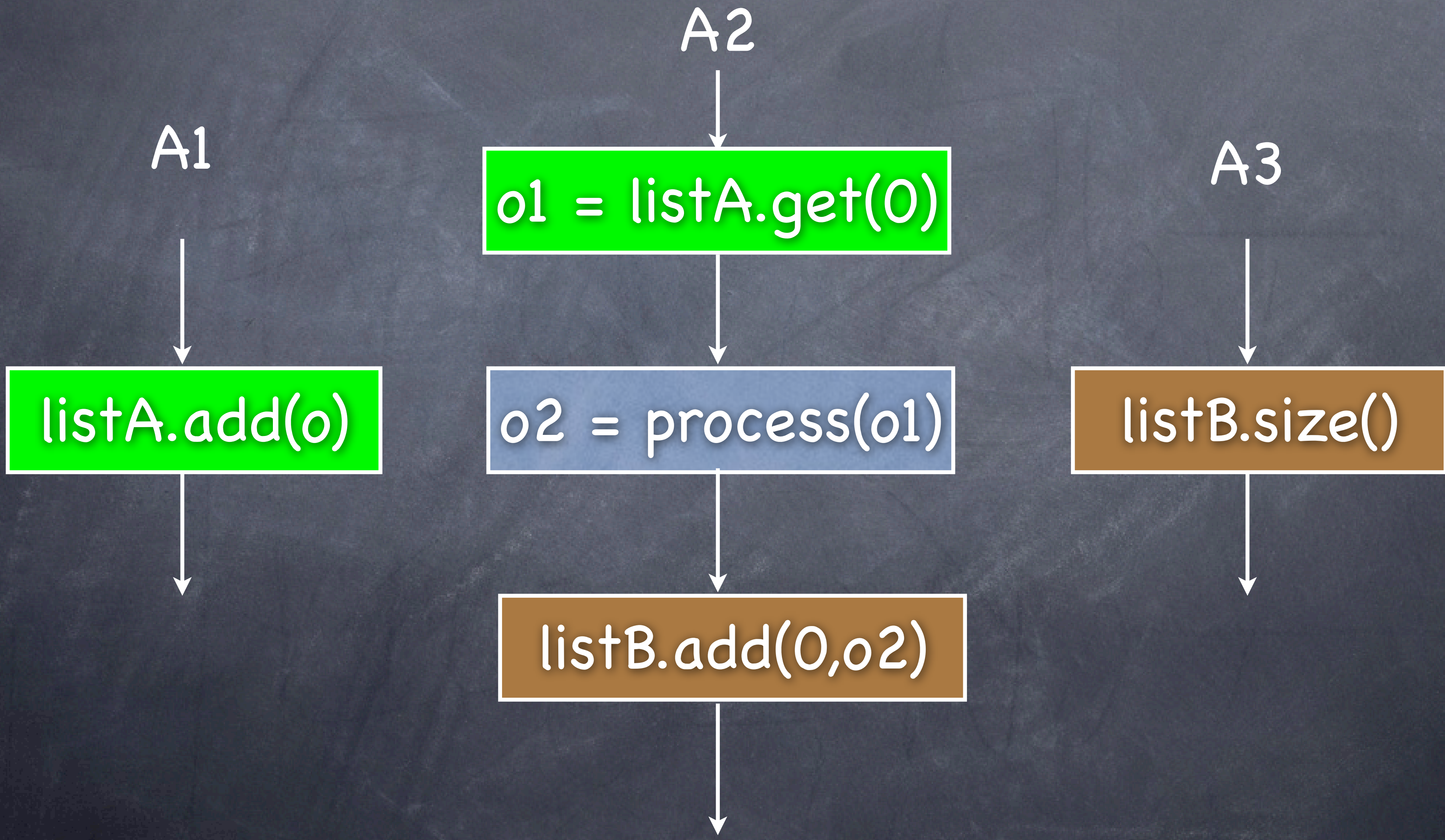
Future Work: Area 1

- Maximise concurrency between atomic sections that only partially conflict
- Existing work either:
  - Serialises whole atomics [Halpert07, Zhang07, Cherem08, Hicks06]
  - Serialises upto a conflict [Cunningham08]
  - Serialises after a conflict [McCloskey06, Emmi07]
- Two-phase locking can be too restrictive and thus hamper concurrency unnecessarily

Future Work: Area 1

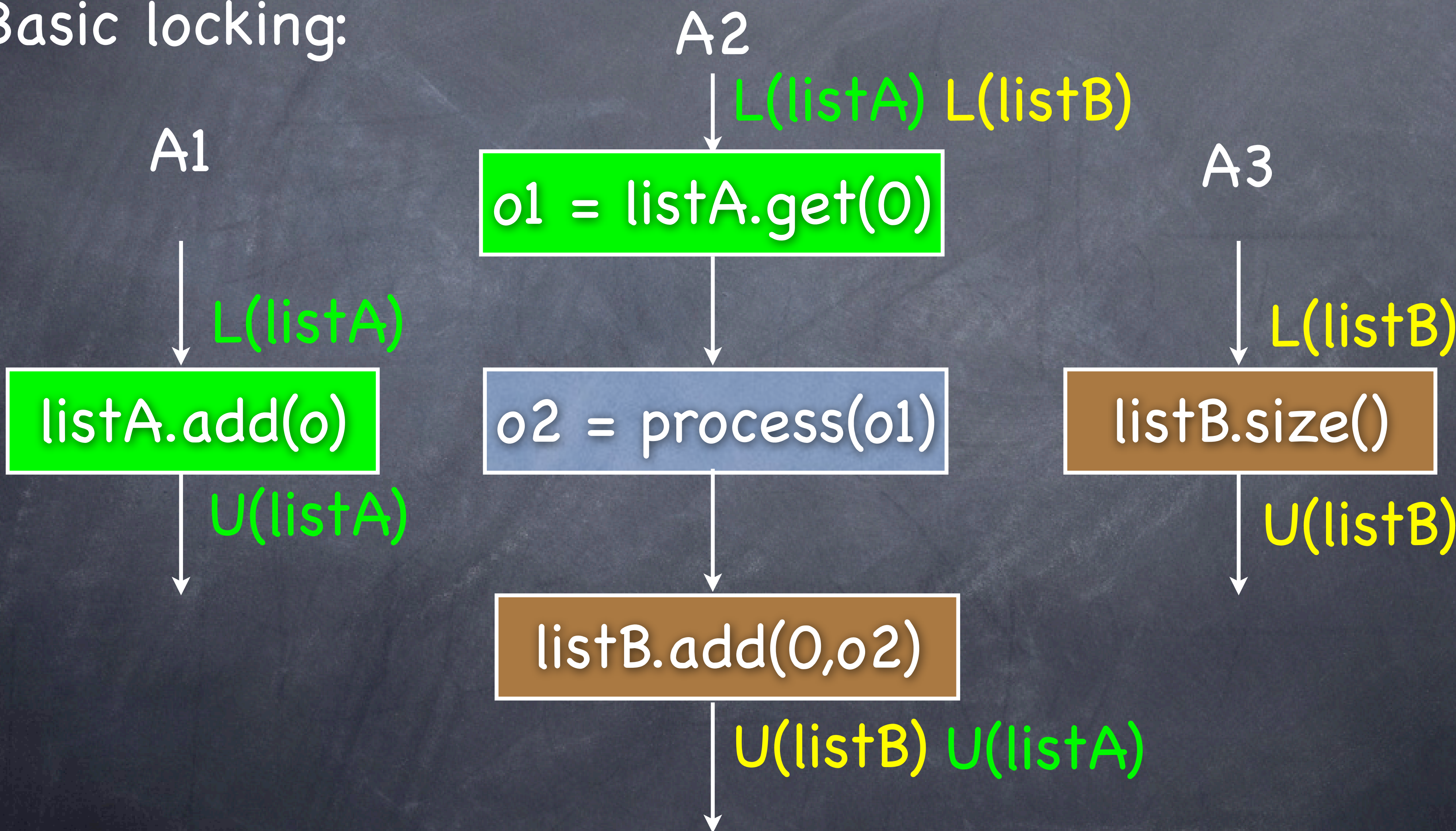


Future Work: Area 1



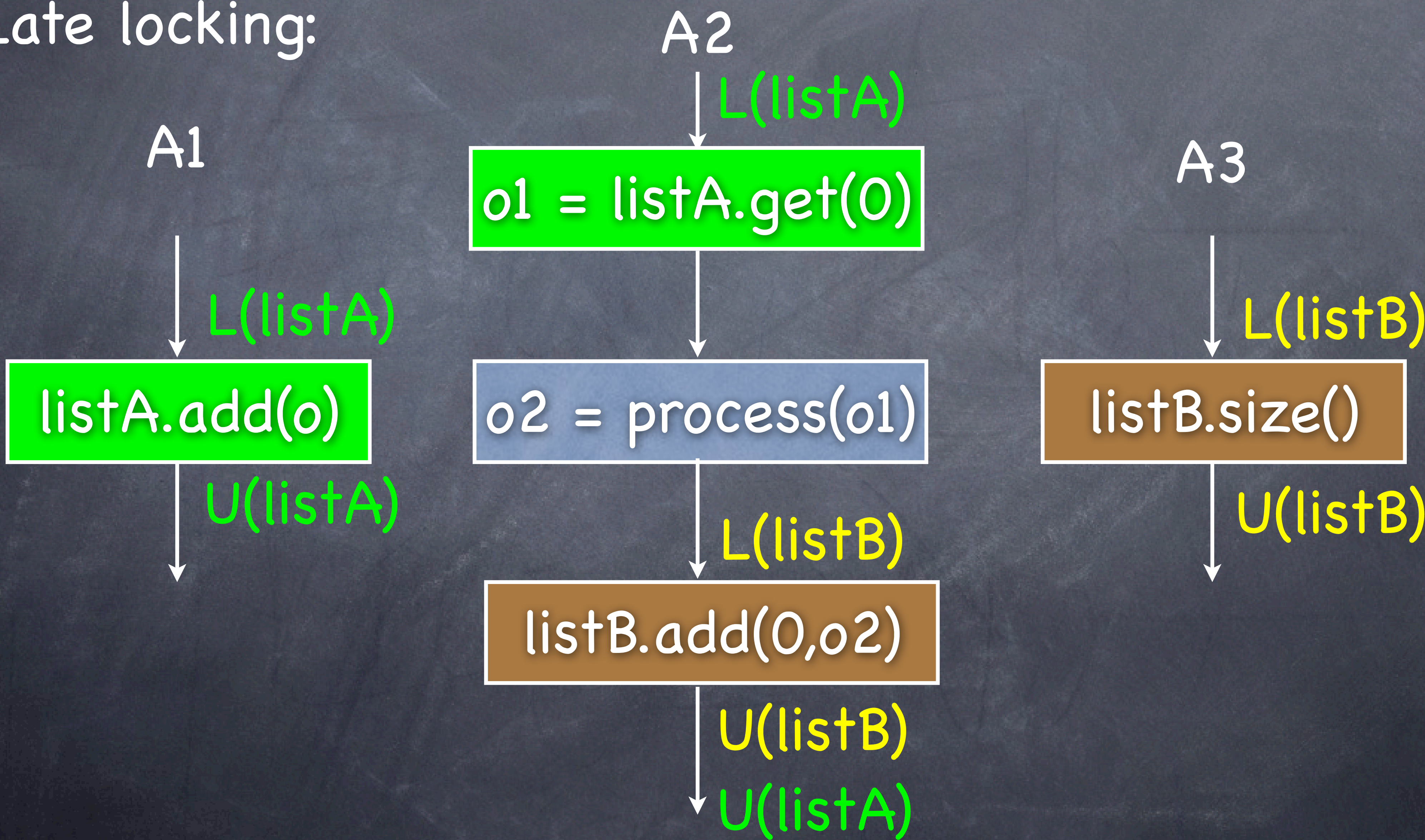
Future Work: Area 1

Basic locking:



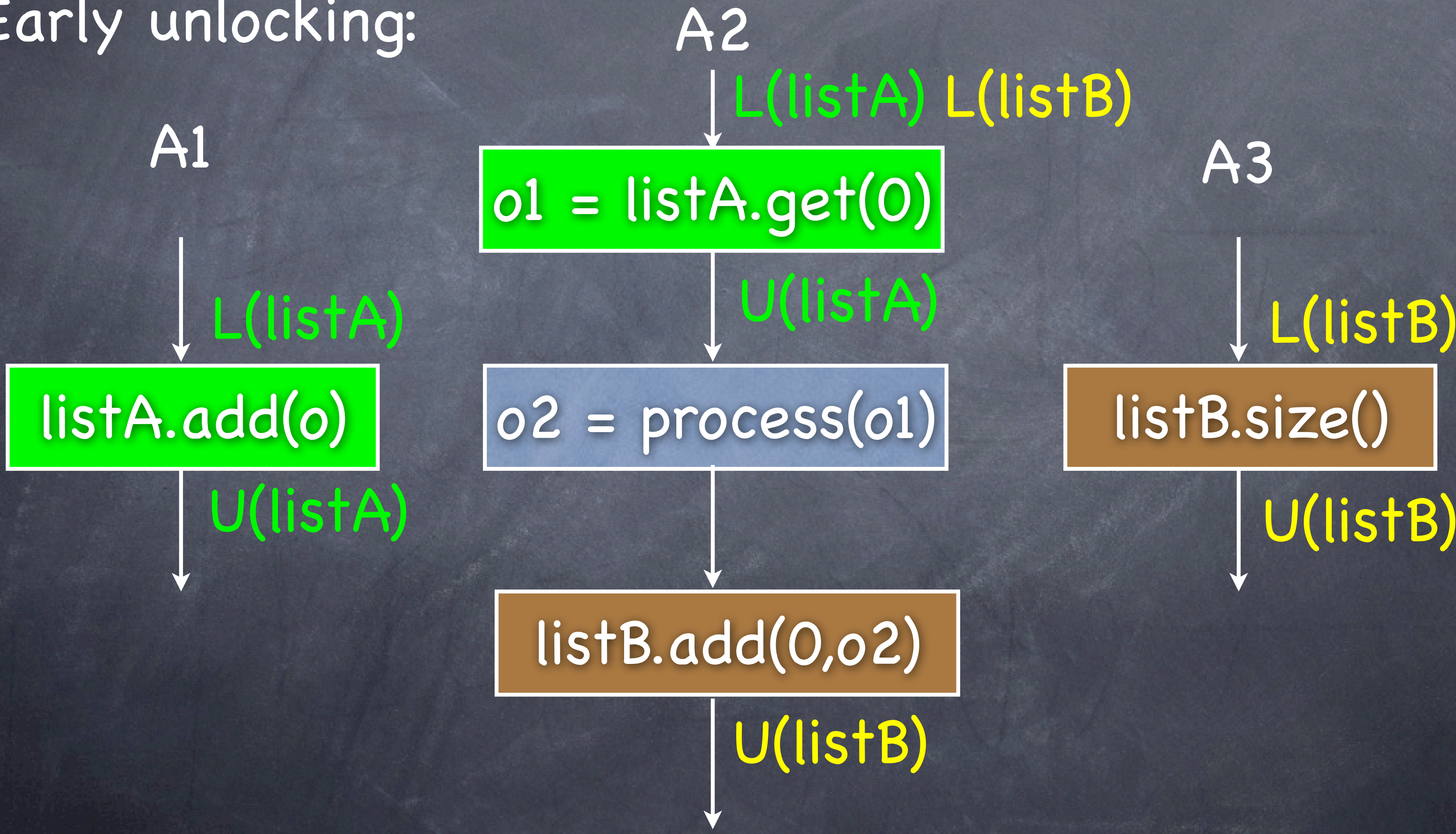
Future Work: Area 1

Late locking:

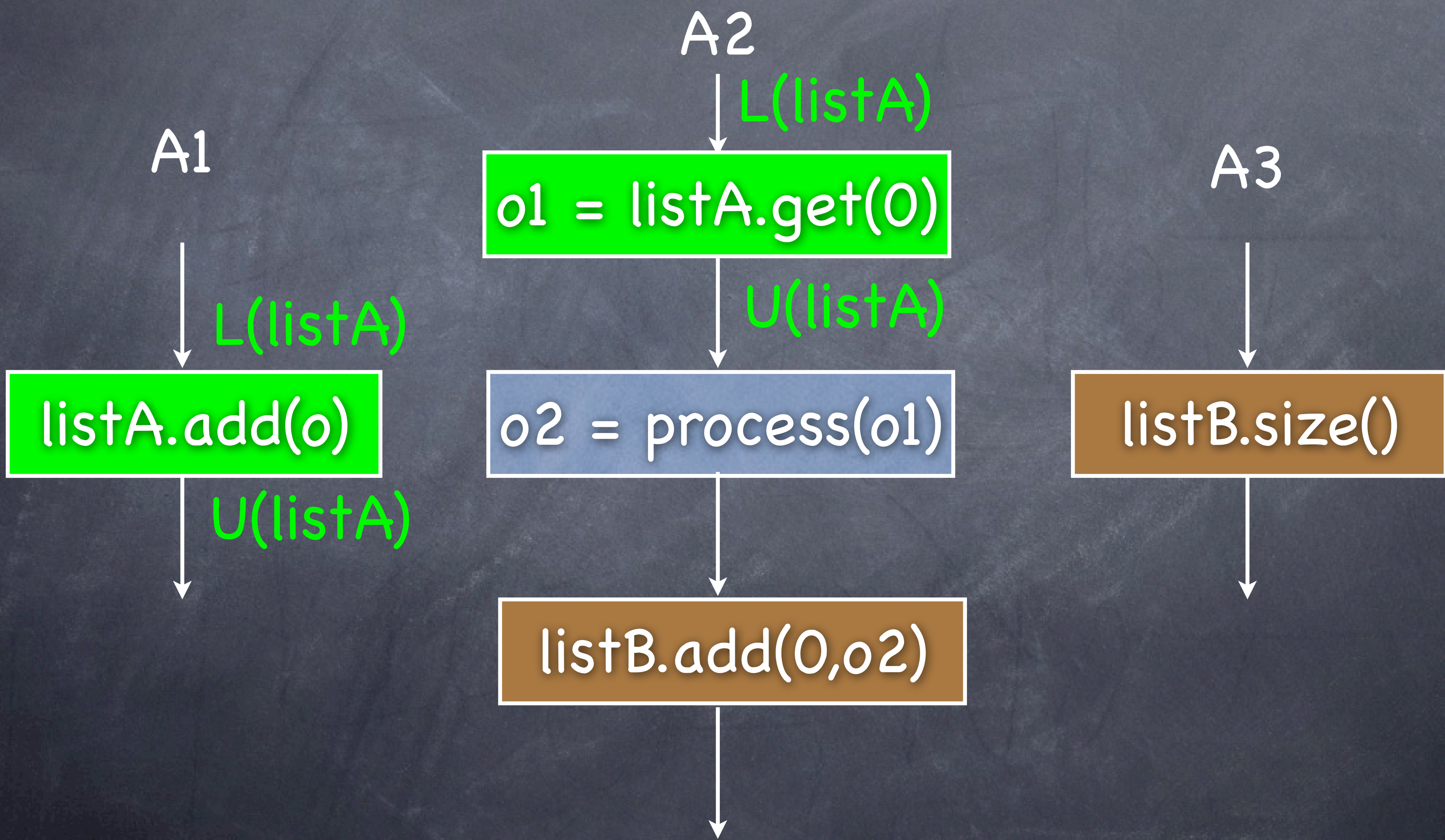


Future Work: Area 1

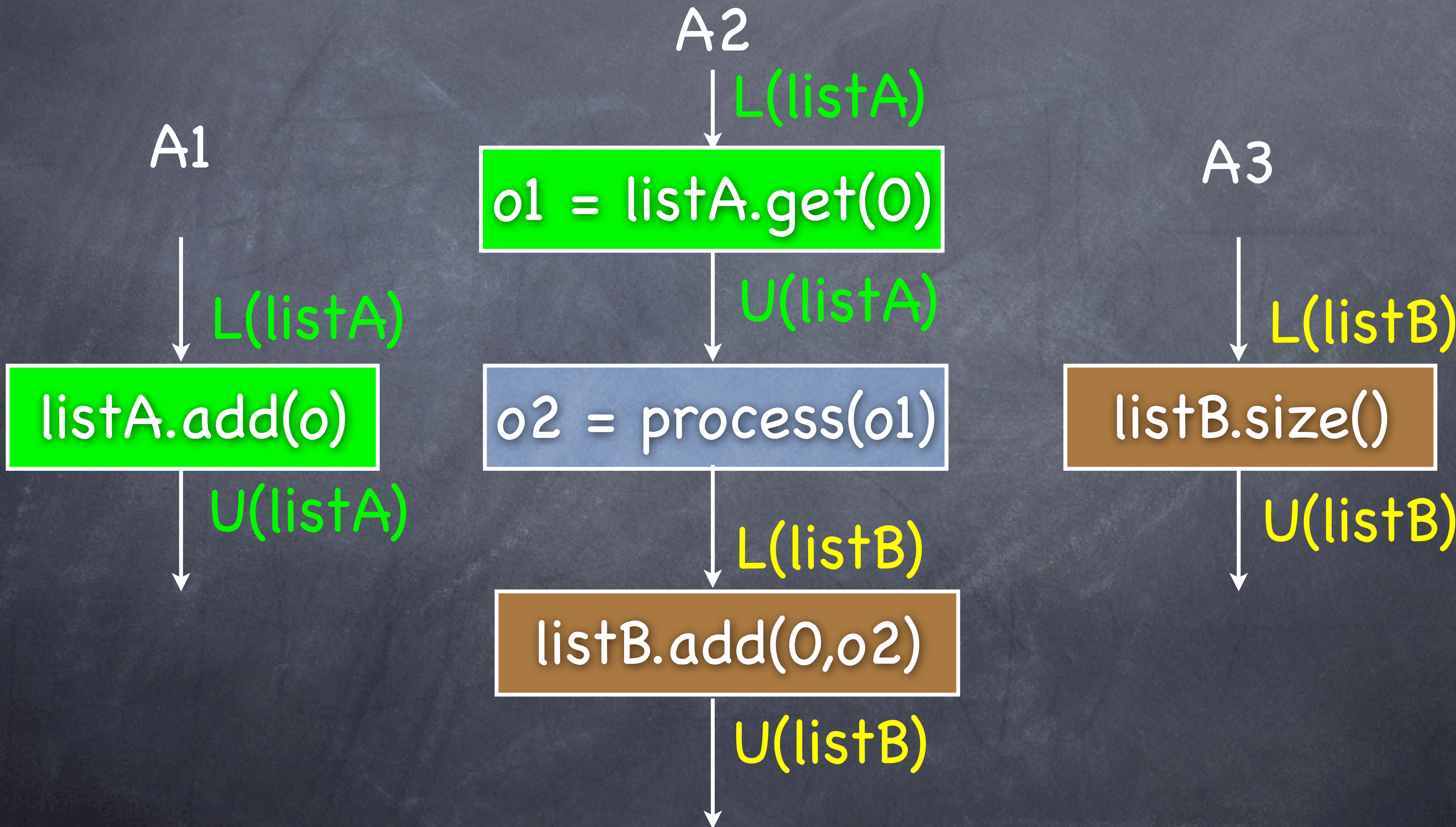
Early unlocking:



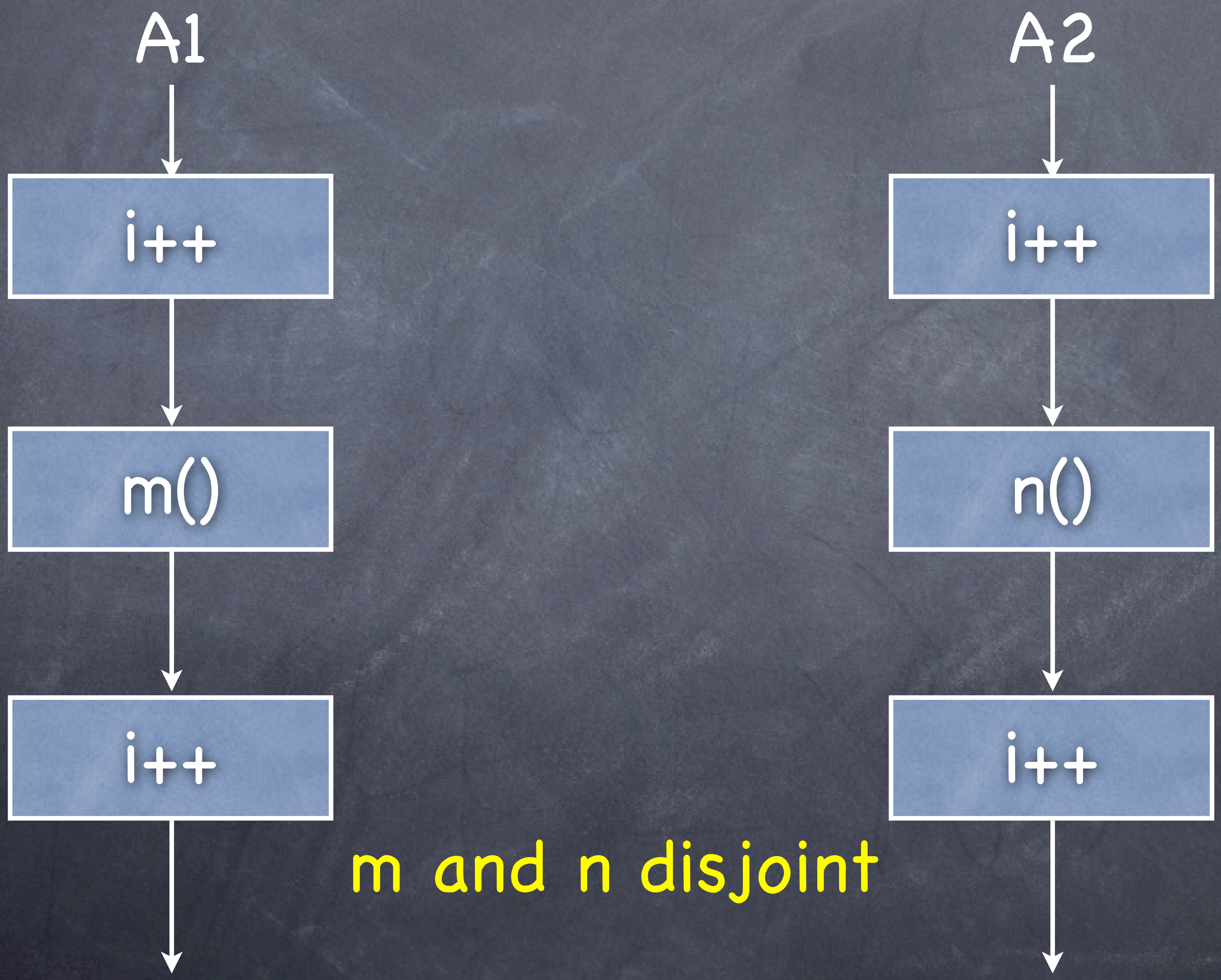
Future Work: Area 1



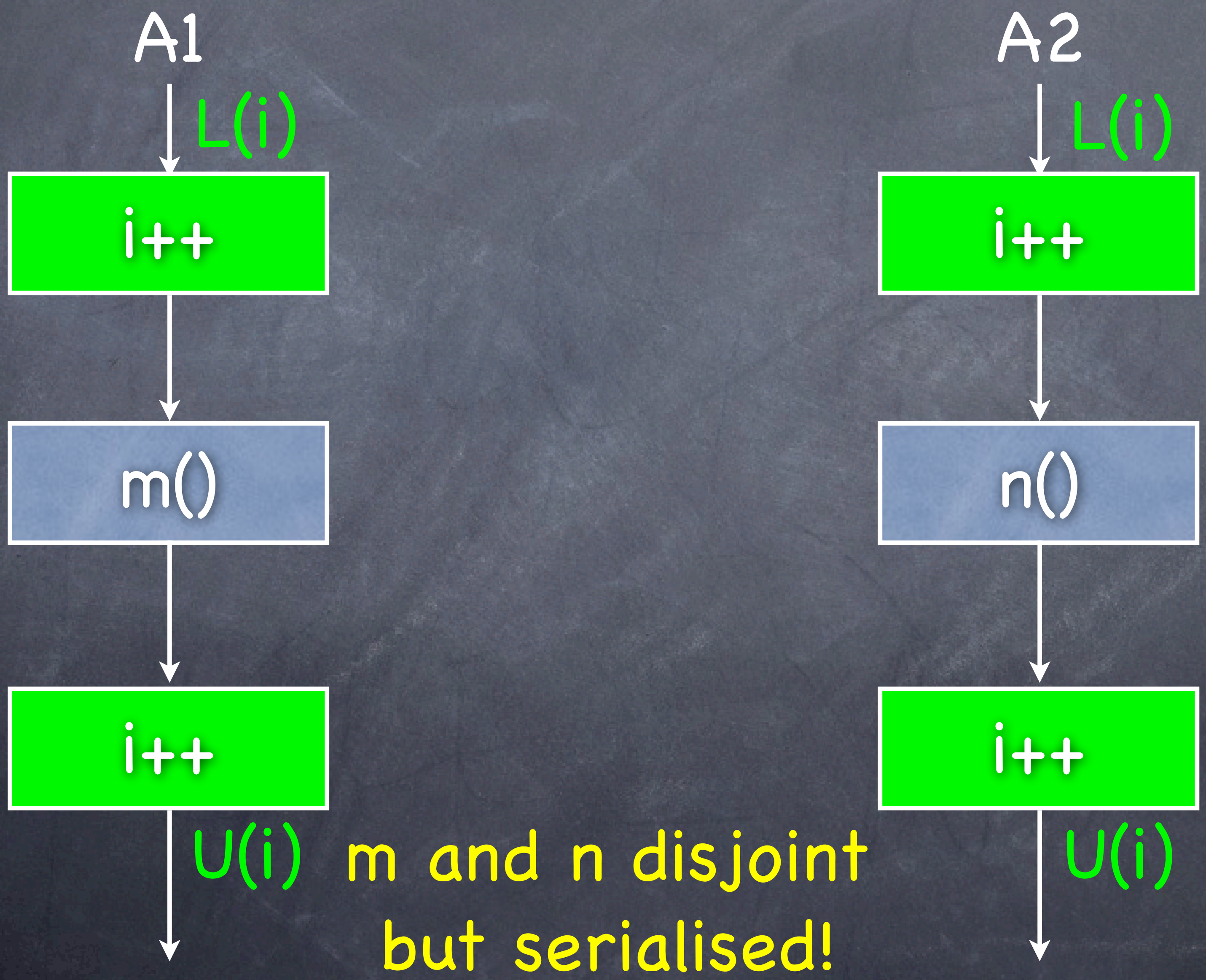
Future Work: Area 1



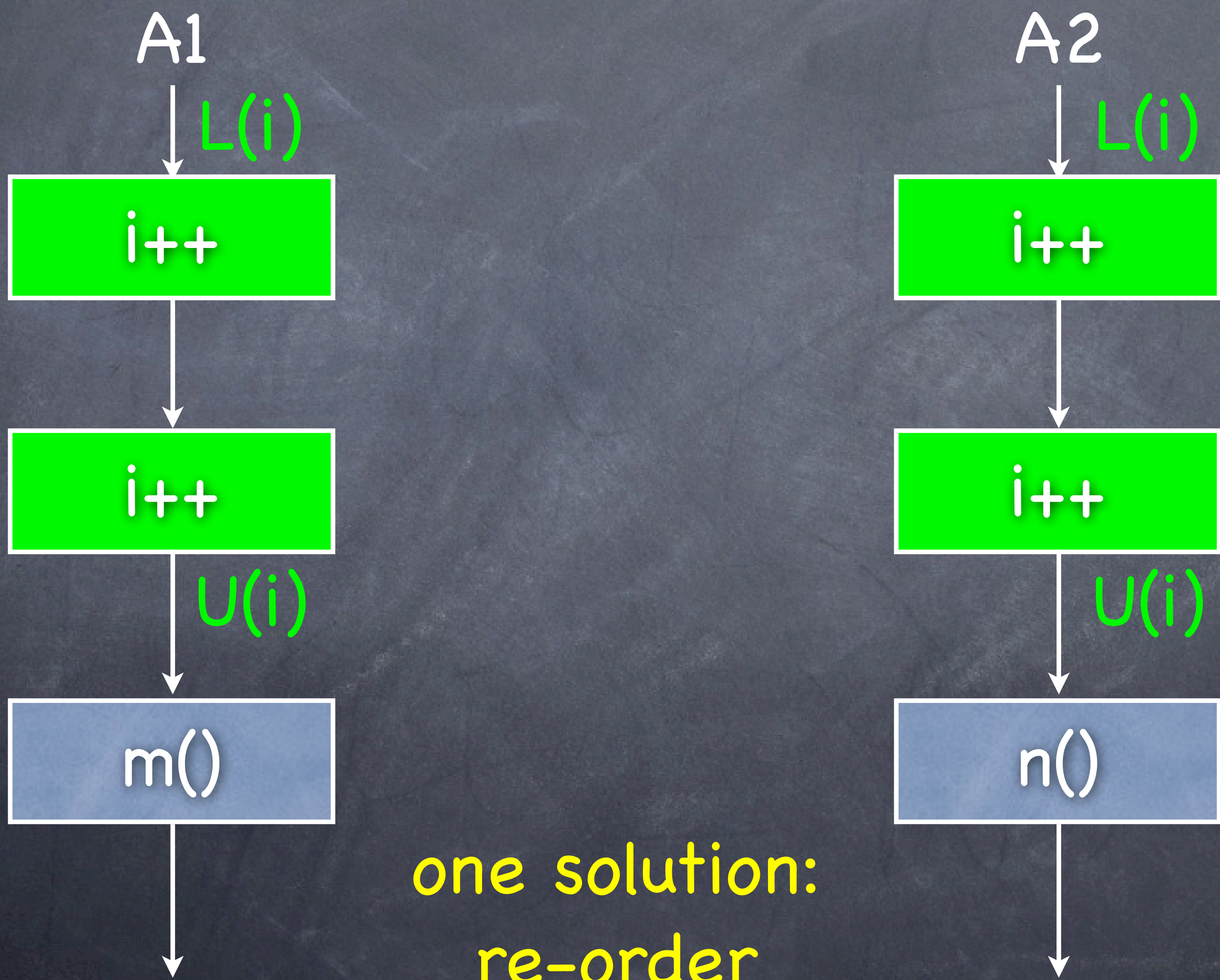
Future Work: Area 1



Future Work: Area 1



Future Work: Area 1



one solution:
re-order

Future Work: Area 2

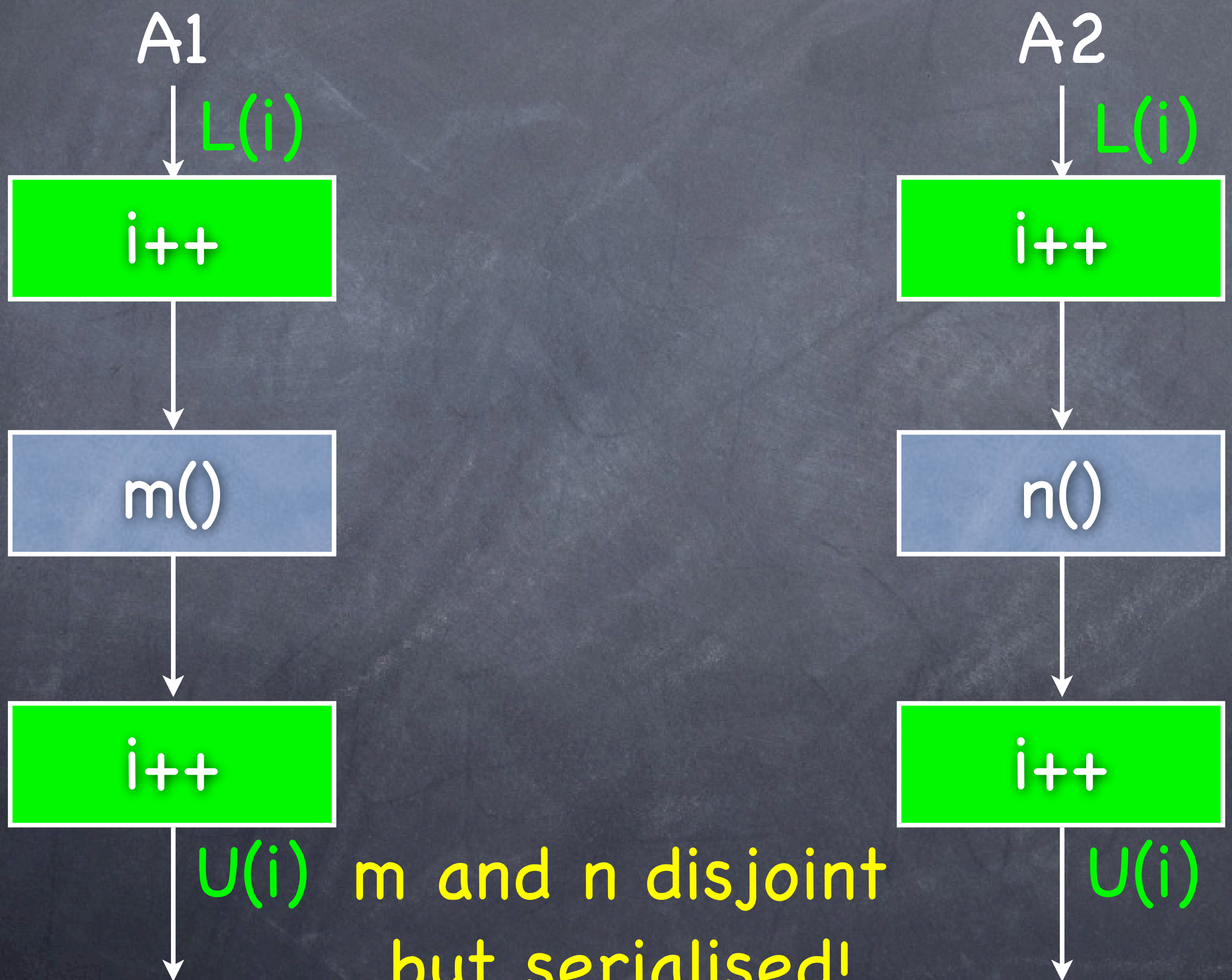
- Area 2: concurrent accesses to arrays: e.g. parallel map function:

```
for (int i=0; i<numChunks; i++) {
    spawn {
        int start = i*chunkSize;
        int end = start+chunkSize;
        for (int j=start; j<end; j++) {
            atomic {
                a[j] = f(a[j]);
            }
        }
    }
}
```

Future Work: Area 3

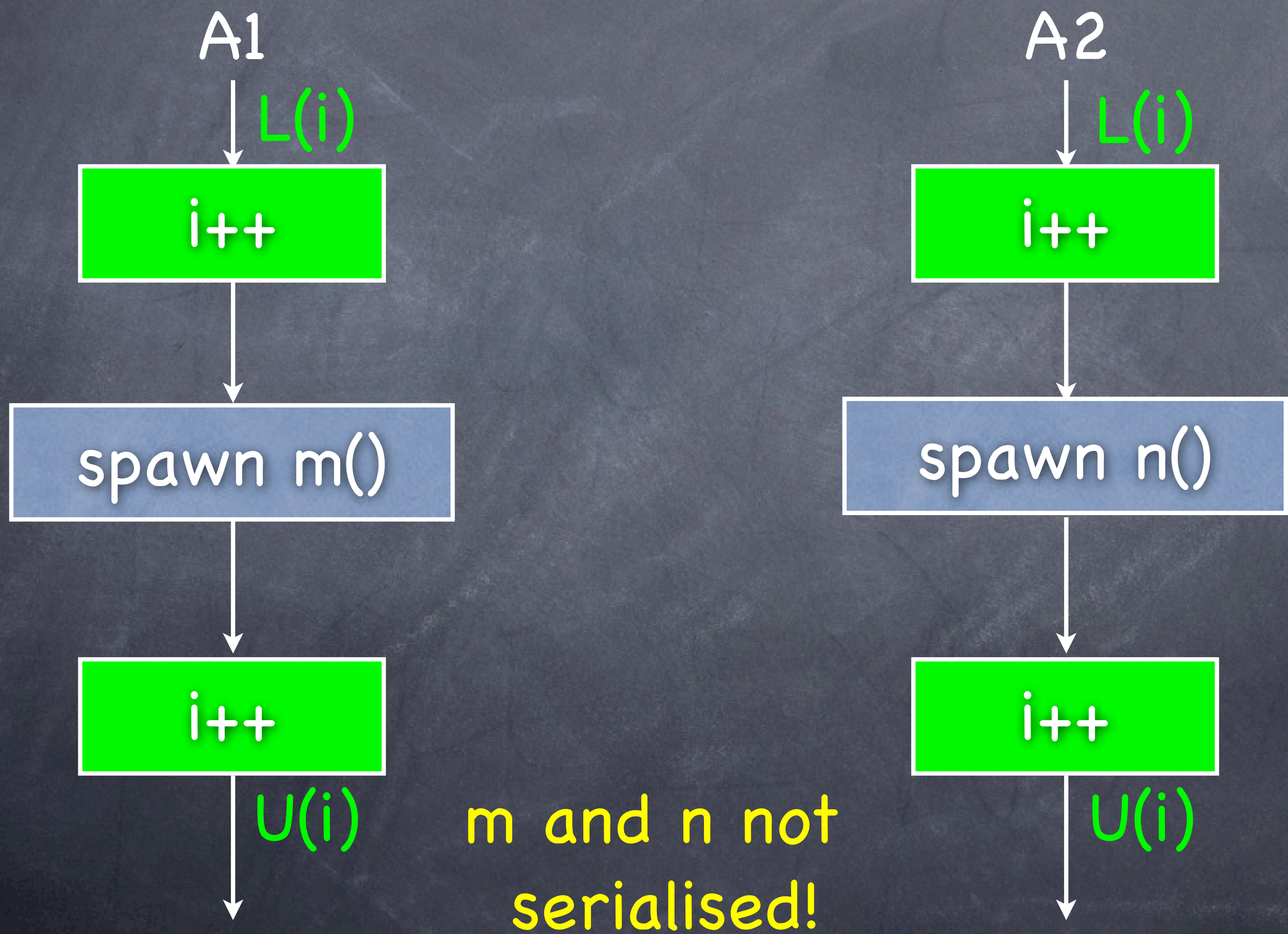
- Area 3: allow the use of multi-threaded code within atomic sections
- Amdahl's law, composability
- Support a spawn construct inside atomic { }
- Could also use to automatically improve the performance of atomic sections

Future Work: Area 3



m and n disjoint
but serialised!

Future Work: Area 3



Future Work: Area 4

- Area 4: consider a hybrid implementation with transactional memory
- Benefit of transactional memory's high concurrency
- Reduce run-time overhead and allow irreversible operations using locks

Related work

- Philosophy of approach
  Top down [Zhang07, Halpert07]
  Bottom up
  [McCloskey06, Hicks06, Emmi07, Cunningham08, Cherem08]

- Compile-time representation of objects:
  Abstract objects [Hicks06, Halpert07]
  Lvalues
  [McCloskey06, Hicks06, Emmi07, Cunningham08, Cherem08]

- Granularity of locks:
  Fine [McCloskey06, Emmi07, Halpert07]
  Coarse [Hicks06, Halpert07, Zhang07]

Related work

- The specific two-phase locking policy:
  Basic [Hicks06, Zhang07, Halpert07, Cherem08]
  Late locking [McCloskey06, Emmi07]
  Early unlocking [Cunningham08]

- Deadlock avoidance:
  Static [McCloskey06, Hicks06, Emmi07, Zhang07, Halpert07]
  Dynamic [Cunningham08, Cherem08]

Conclusion

- My thesis:
  - Implement atomic using locks
  - Maximise concurrency between atomics
  - Be able to handle a real language

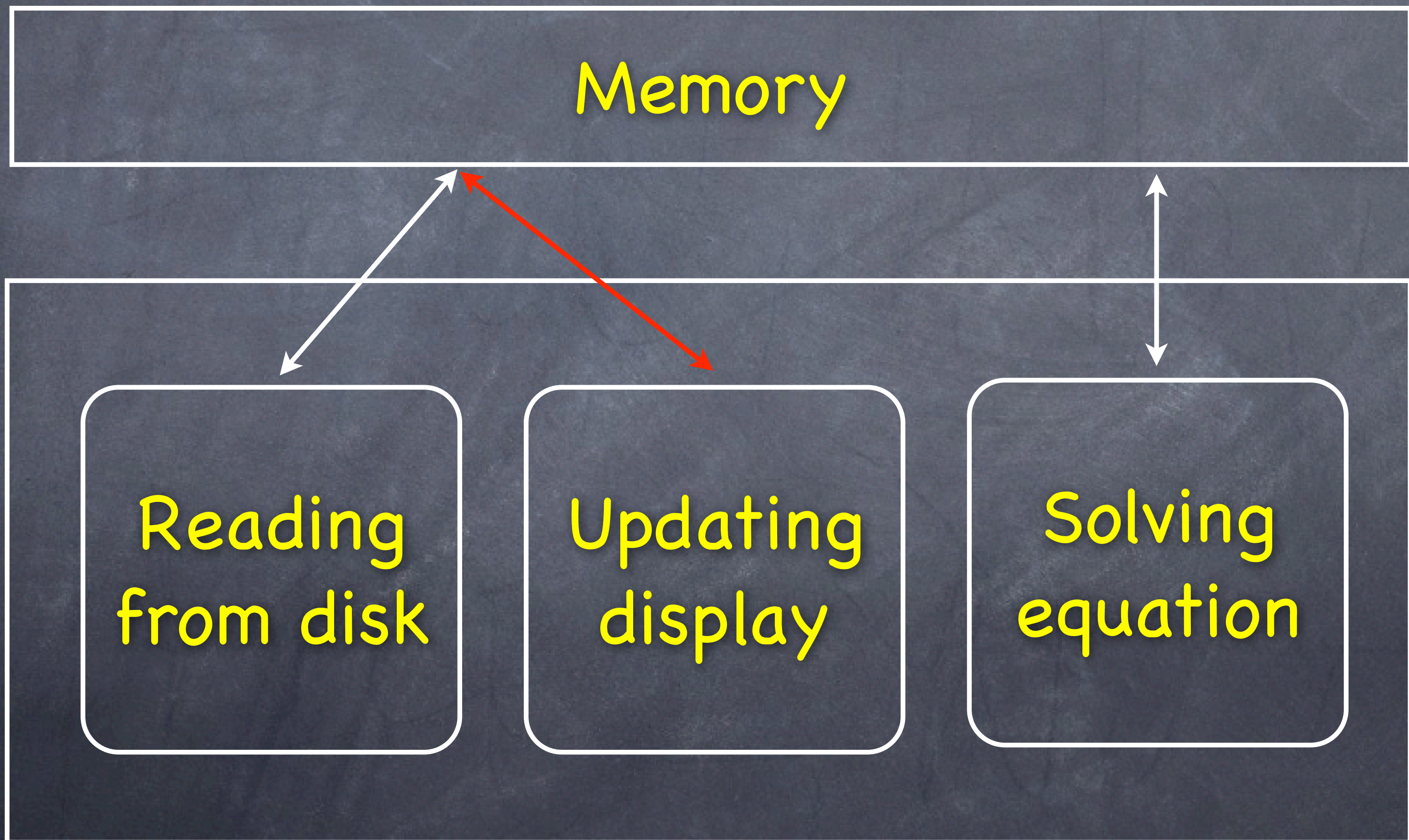
Questions?

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in; we're computer professionals. We cause accidents."

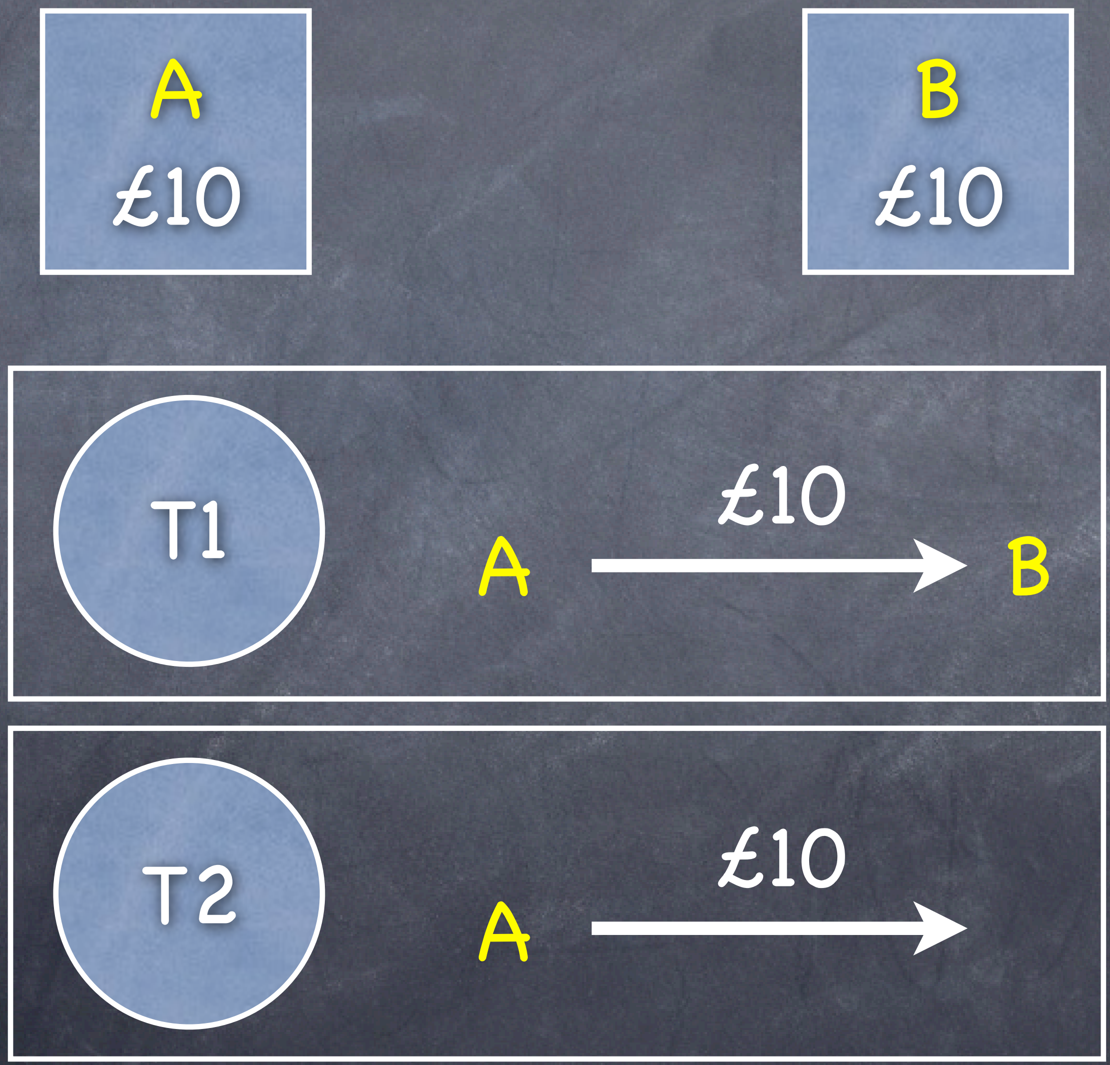
Nathaniel Borenstein (co-creator of MIME)

We need better abstractions!

The problem: shared memory



Bank account example



Bank account (locks)

- Method that transfers money between accounts, if sufficient funds are available:

```
void transfer(Acct A, Acct B, int amt) {
    int bal = A.getBalance();
    if (amt <= bal) {
        A.withdraw(amt);
        B.deposit(amt);
    }
}
```

Bank account (locks)

transfer(A, B, 10) || a.withdraw(10)

Time	T1	T2	A	B
1	Check A`s balance		£10	£10
2		Withdraw £10 from A	£0	£10
3	Withdraw £10 from A		-£10	£10
4	Deposit £10 into B		-£10	£20

Bank account (locks)

transfer(A, B, 10) || a.withdraw(10)

Time	T1	T2	A	B
1	Check A`s balance		£10	£10
2		Withdraw £10 from A	£0	£10
3	Withdraw £10 from A		-£10	£10
4	Deposit £10 into B		-£10	£20

Bank account (locks)

transfer(A, B, 10) || a.withdraw(10)

Time	T1	T2	A	B
1	Check A`s balance		£10	£10
2		Withdraw £10 from A	£0	£10
3	Withdraw £10 from A		-£10	£10
4	Deposit £10 into B		-£10	£20

Bank account (locks)

transfer(A, B, 10) || a.withdraw(10)

Time	T1	T2	A	B
1	Check A`s balance		£10	£10
2		Withdraw £10 from A	£0	£10
3	Withdraw £10 from A		-£10	£10
4	Deposit £10 into B		-£10	£20

Bank account (locks)
Second attempt:

```
void transfer(Acct A, Acct B, int amt) {
    synchronized(A) {
        synchronized(B) {
            int bal = A.getBalance();
            if (amt <= bal) {
                A.withdraw(amt);
                B.deposit(amt);
            }
        }
    }
}
```

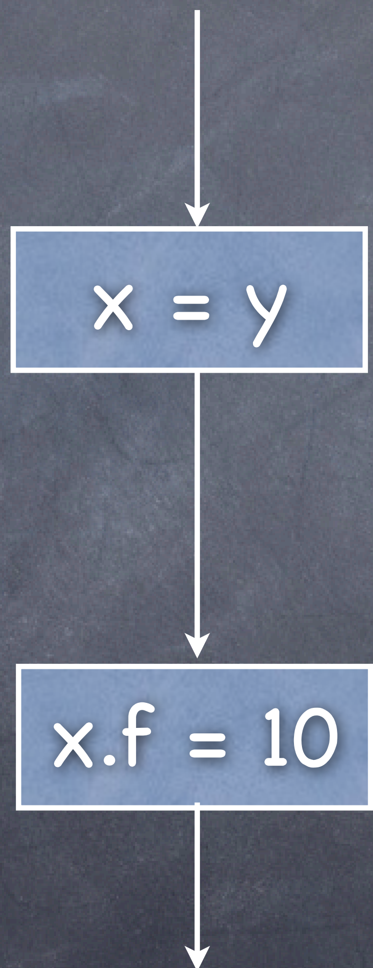
Bank account (locks)

- The new implementation has introduced the possibility of deadlock:
- transfer(A, B, 10) || transfer(B, A, 20)

Time	T1	T2
1	lock A	
2		lock B
3	lock B	
4	waiting	lock A
5	waiting	waiting

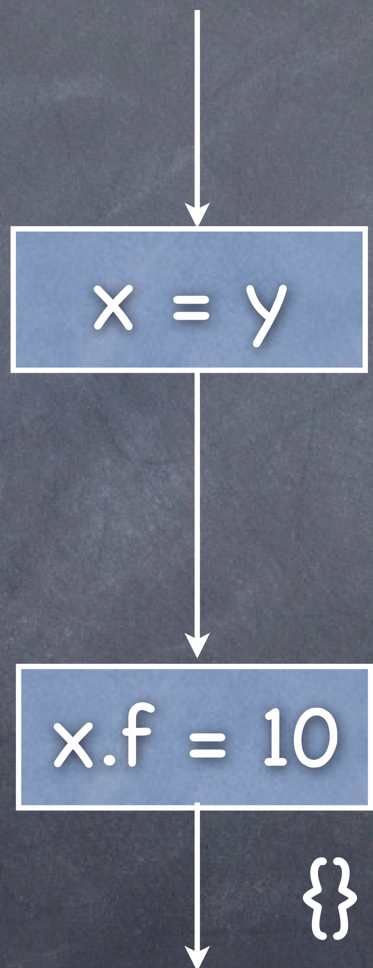
Inferring lvalues

```
atomic {
    x = y;
    x.f = 10;
}
```



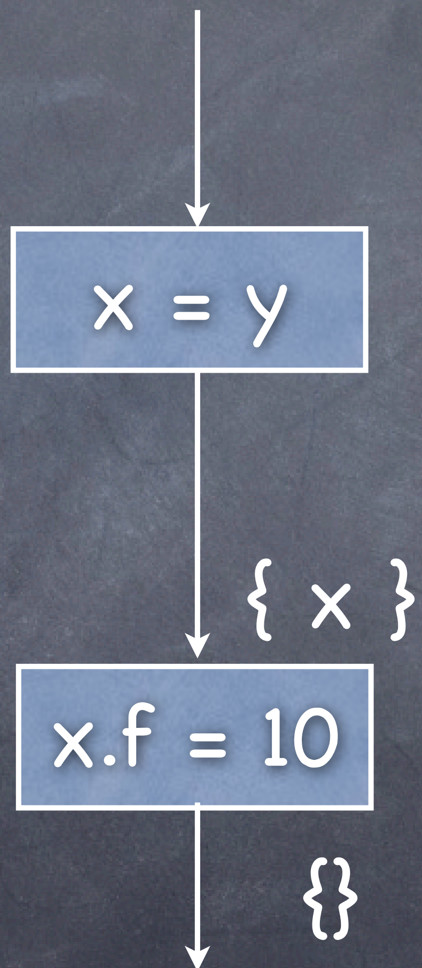
Inferring lvalues

```
atomic {
    x = y;
    x.f = 10;
}
```



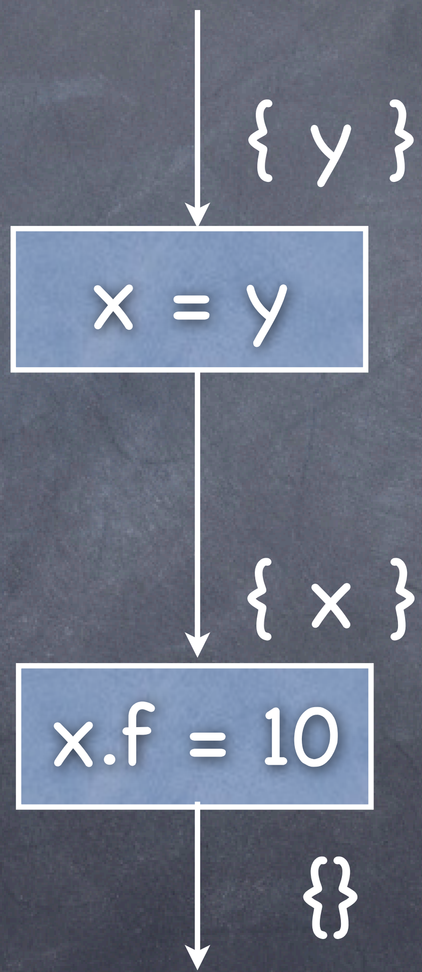
Inferring lvalues

```
atomic {
    x = y;
    x.f = 10;
}
```



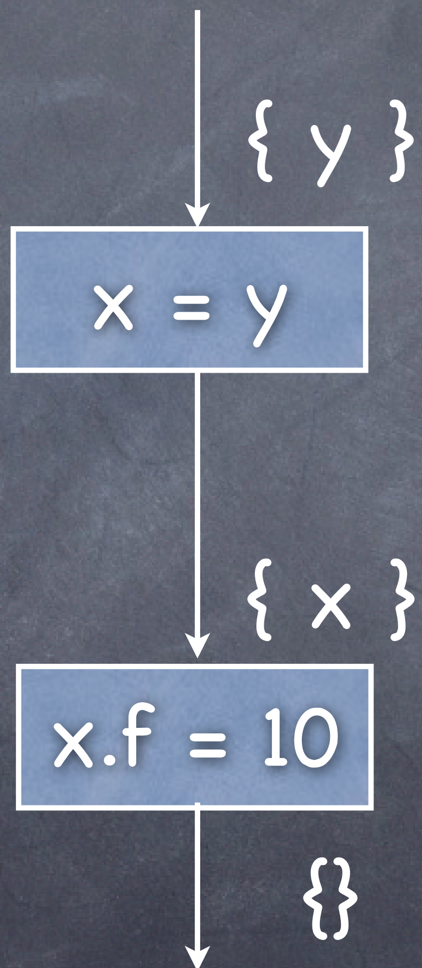
Inferring lvalues

```
atomic {
    x = y;
    x.f = 10;
}
```



Inferring lvalues

```
atomic {
    x = y;
    x.f = 10;
}
```

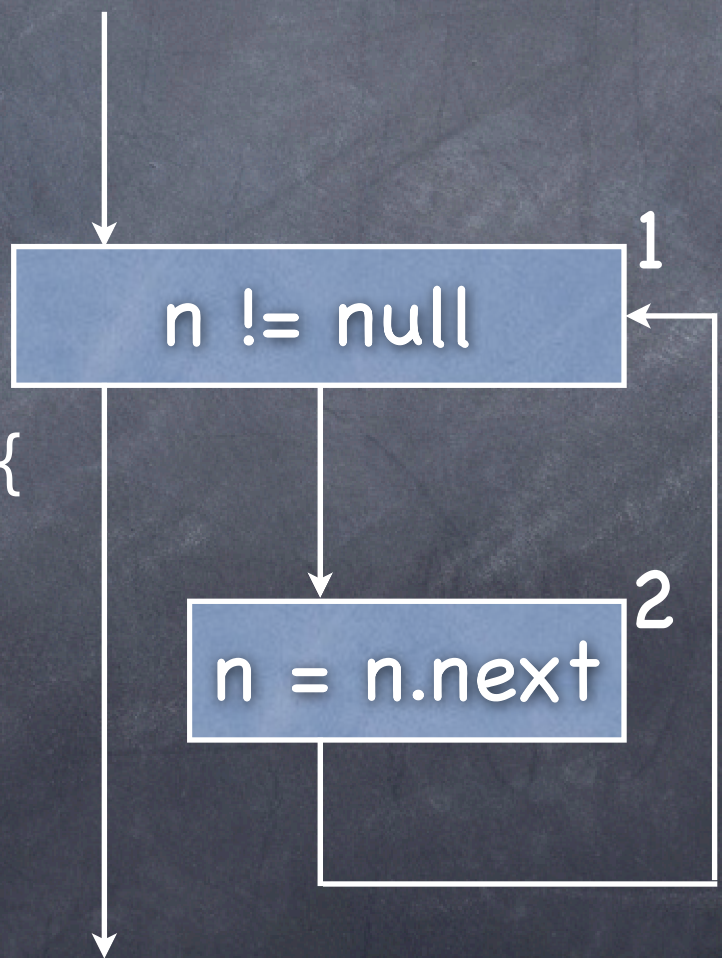



```
synchronized(y) {
    x = y;
    x.f = 10;
}
```

Problems with iteration

- How many objects accessed?

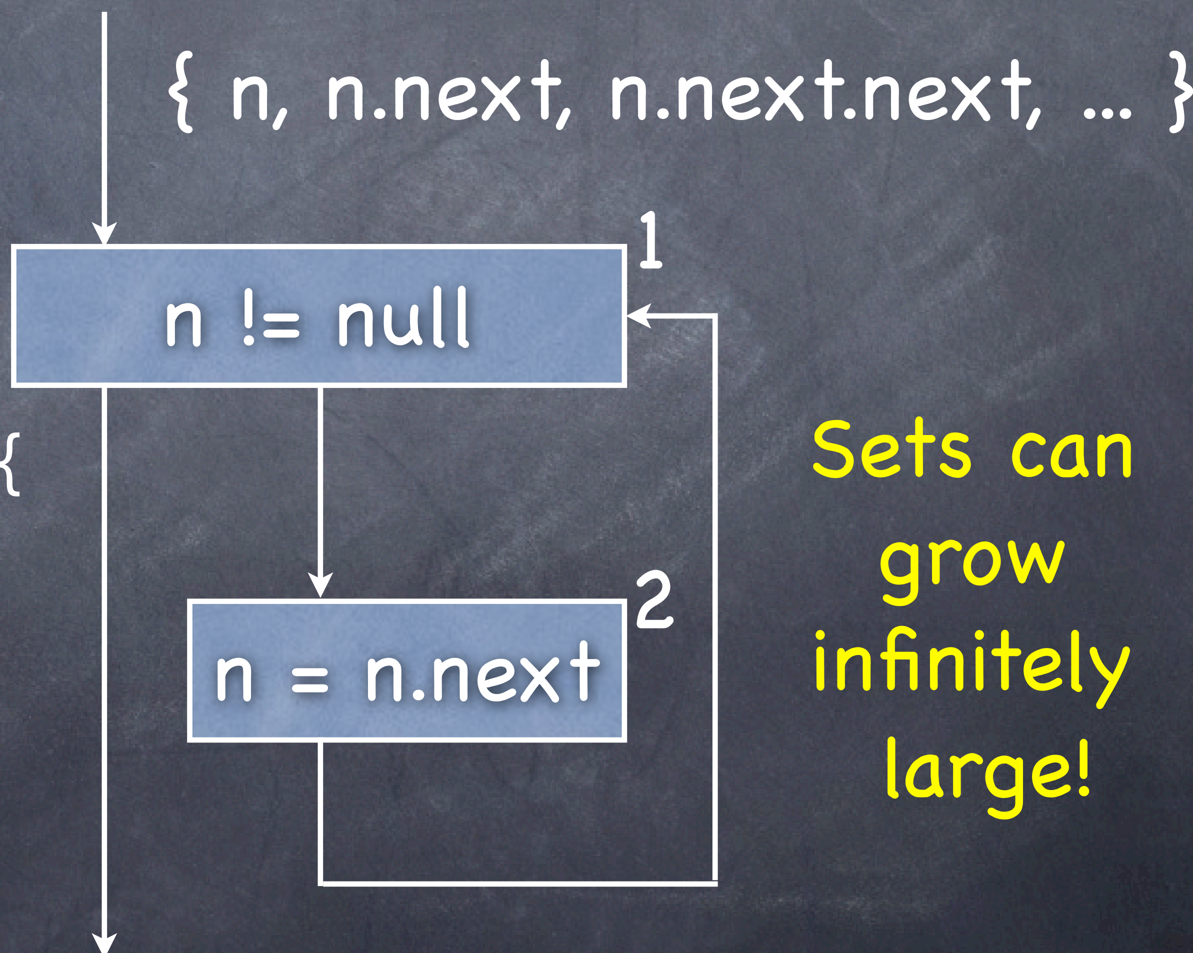
```
atomic {
   while (n != null) {
       n = n.next;
   }
}
```



Problems with iteration

- How many objects accessed?

```
atomic {
   while (n != null) {
       n = n.next;
   }
}
```



Sets can grow infinitely large!