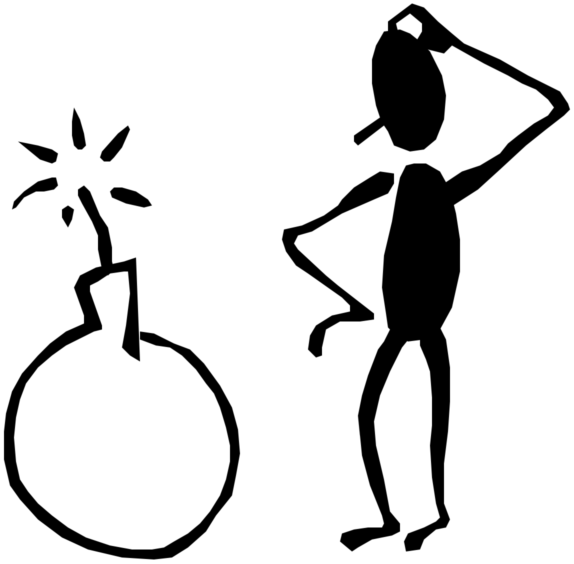
Fixing the Java Memory Model



William Pugh

Dept. Of Computer Science
Univ. of Maryland

http://www.cs.umd.edu/~pugh/java

Overview

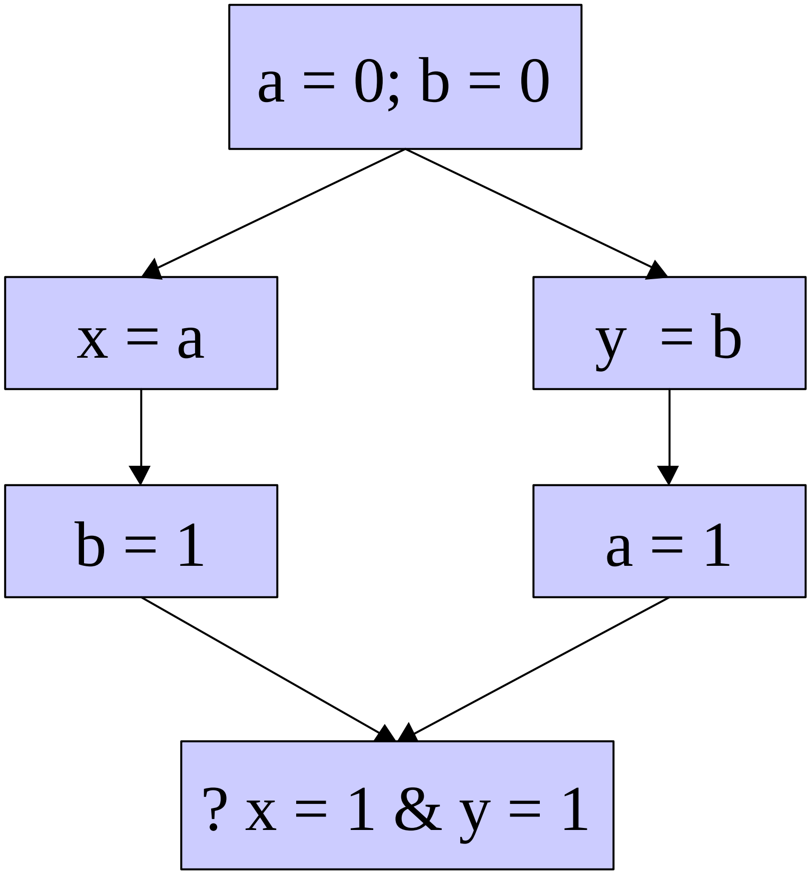
- Memory Models, and the JMM in particular
- The JMM is too strong
  - prohibits standard compiler optimizations
    - done by most existing JVM's,
      in violation of the spec
- The JMM is too weak
  - initialization safety issues
  - related type-safety issue in implementation

The Java Memory Model

- Chapter 17 of the Java Language Specification (and Chap 8 of the VM Spec)
- Describes how threads interact via locks and read/writes to memory
- Done in a style totally foreign to all other work on memory models
- Very hard to understand
  - At first I thought I was just dense
  - Eventually I figured out that no one understands it

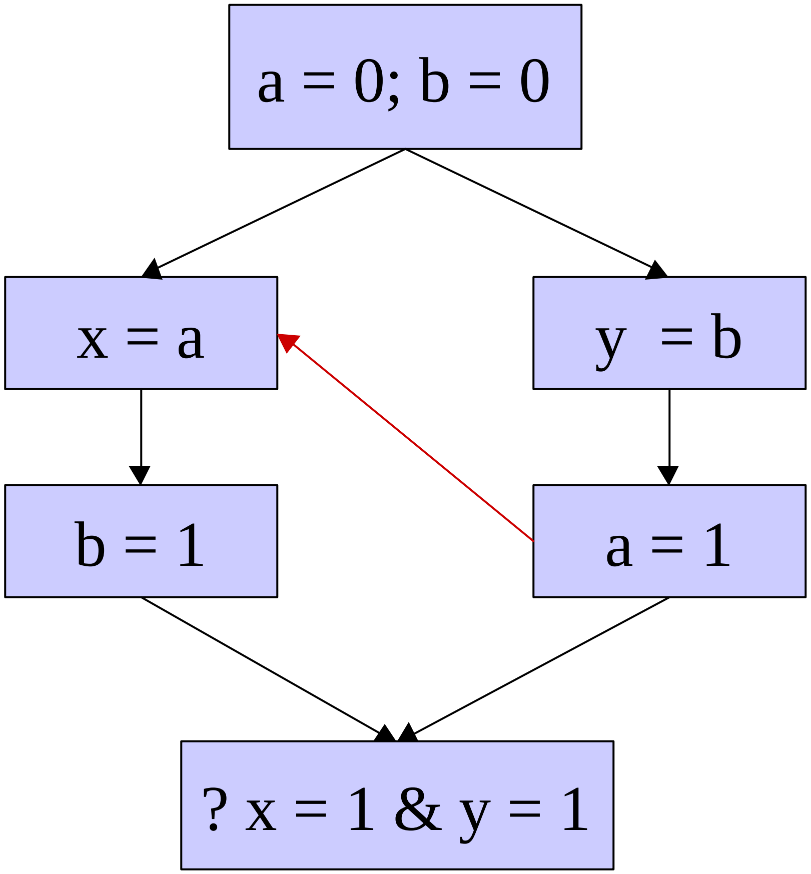
What is a memory model?

- If two threads have a data race, what behaviors are allowed?
- Sequential consistency
  - interleave memory operations consistent with original ordering in each thread



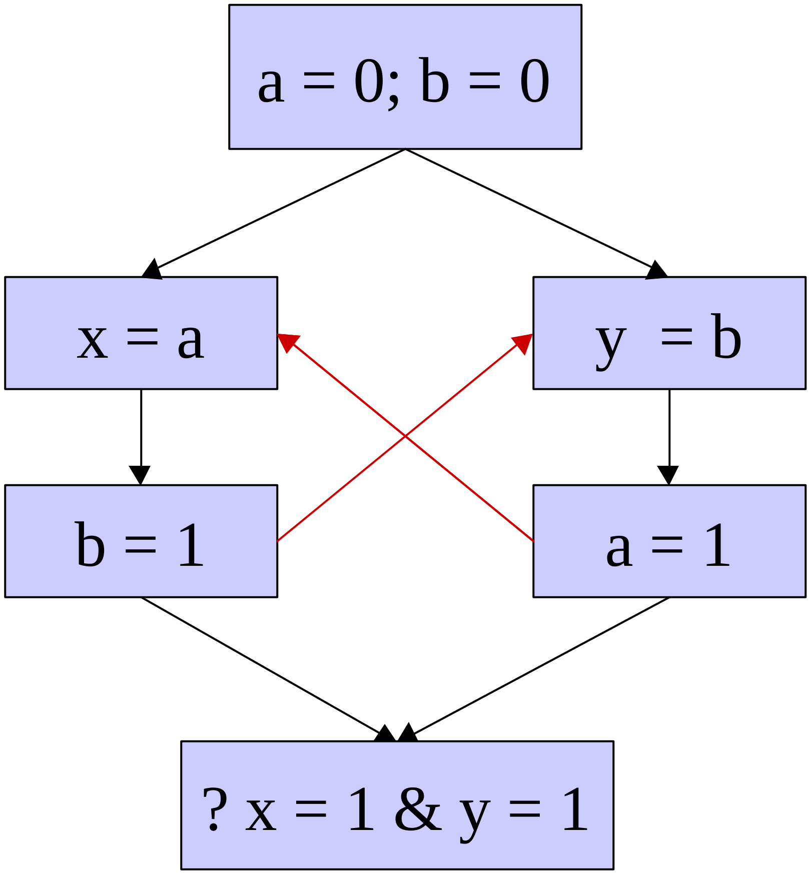
What is a memory model?

- If two threads have a data race, what behaviors are allowed?
- Sequential consistency
  - interleave memory operations consistent with original ordering in each thread



What is a memory model?

- If two threads have a data race, what behaviors are allowed?
- Sequential consistency
  - interleave memory operations consistent with original ordering in each thread



MM’s can interfere with optimization

- In each thread, no ordering constraint between actions in that thread
- Compiler could decide to reorder
- Processor architecture might perform out of order
- Sequential consistency prohibits almost all reordering of memory operations
  - unless you can prove accessed by single thread

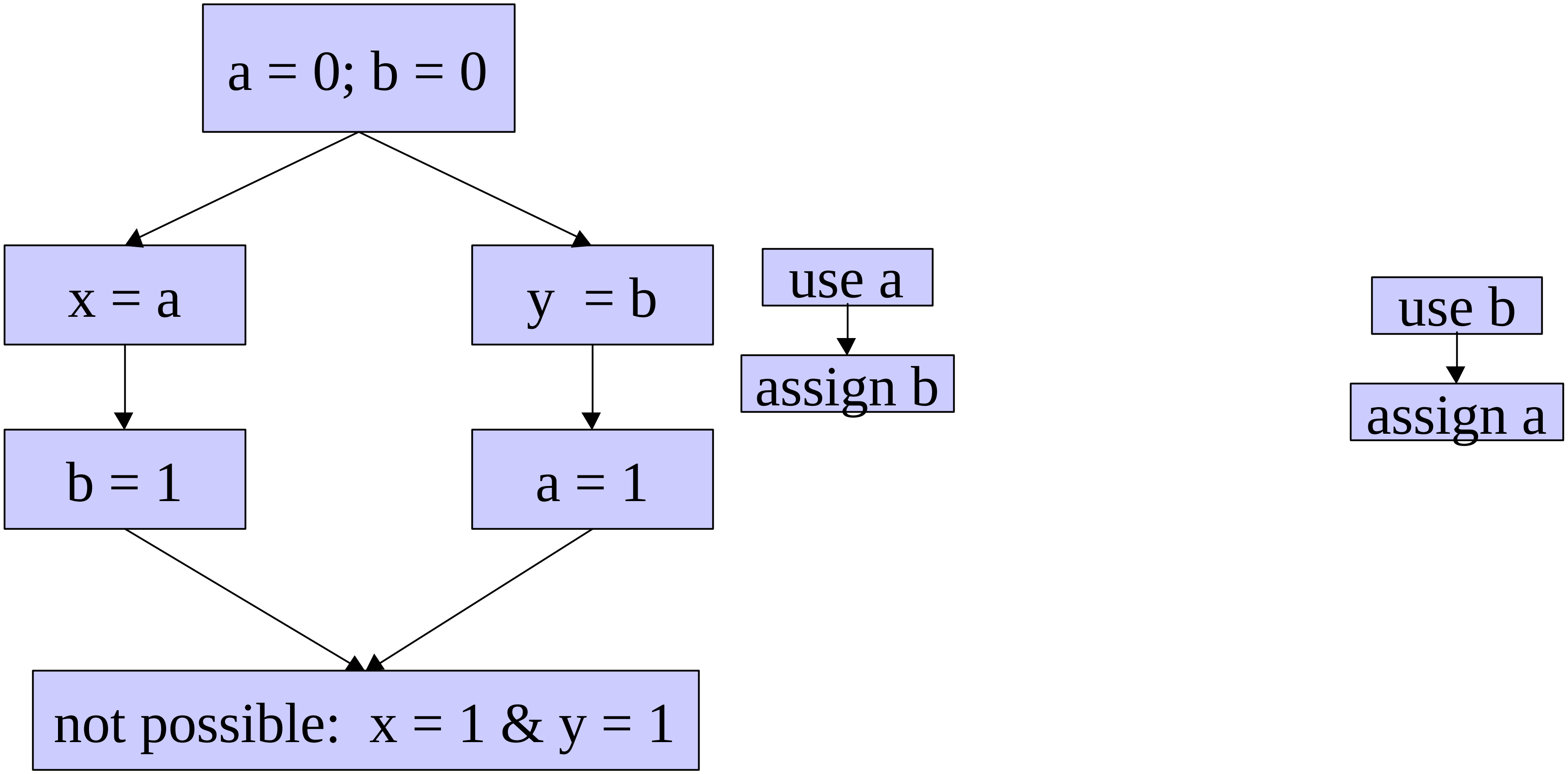
Do programmers care about the details of MM’s?

- If you are writing synchronization primitives
  - You care deeply about the memory model your processor supports
- But if you have synchronized everything properly
  - do you really care?

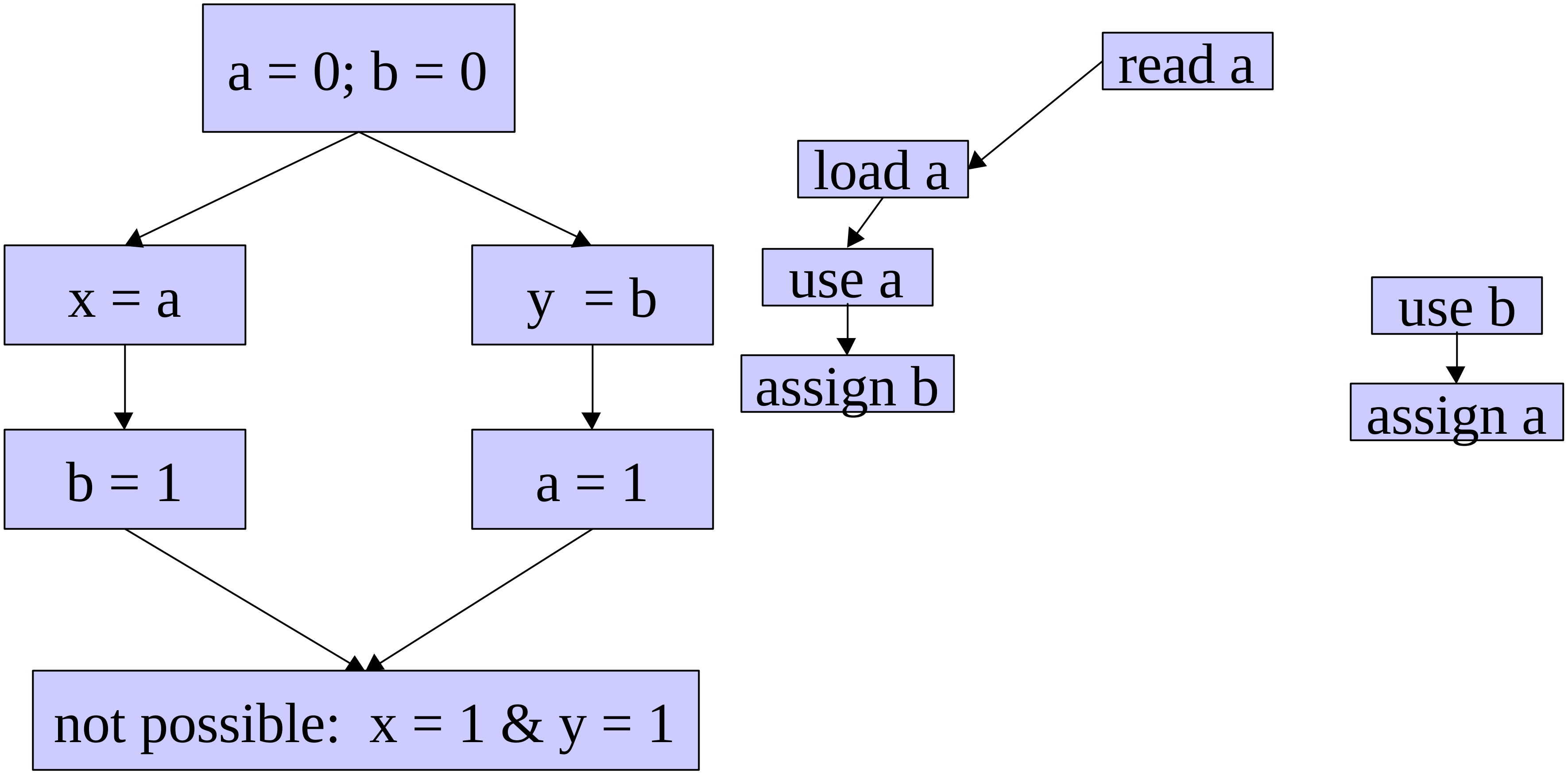
The Java Memory Model

- Idea (apparently):
  - threads have a local memory (cache, registers?)
  - Threads fill from/flush to global memory
- System modeled by constraints between actions
  - Use/assign actions correspond to thread computations
  - load/store actions correspond to thread fill/flush actions
  - read/write actions are main memory actions
- Not like any other memory model
  - don’t ask me why, or ask me to defend it

Memory Model actions, without prescient stores



Memory Model actions, without prescient stores



a = 0; b = 0

x = a

y = b

b = 1

a = 1

not possible: x = 1 & y = 1

read a

load a

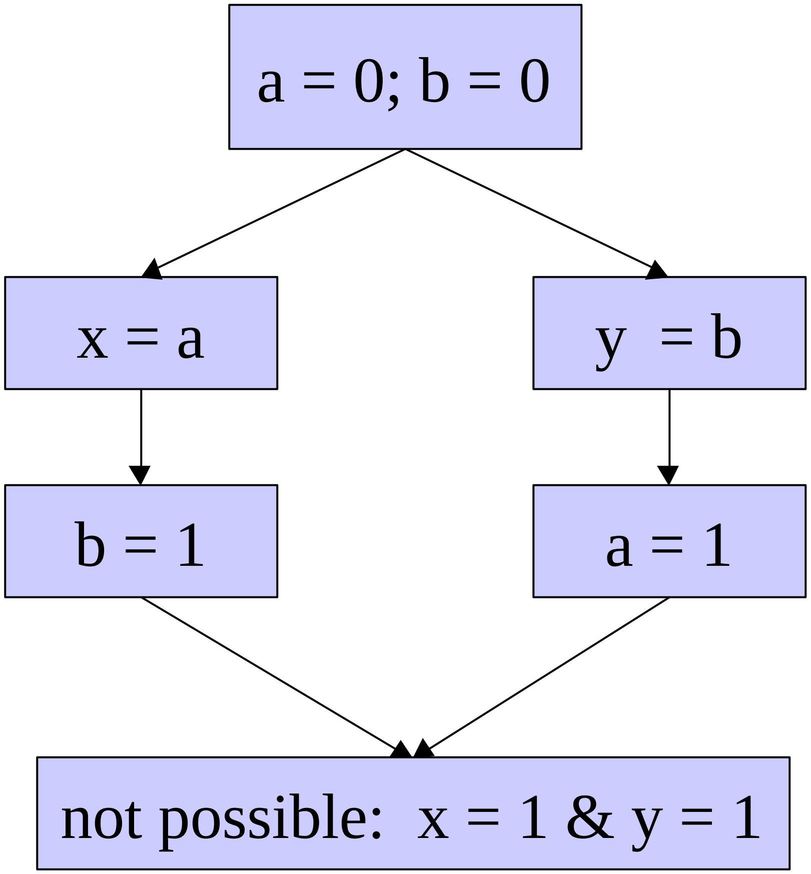
use a

assign b

use b

assign a

Memory Model actions, without prescient stores



a = 0; b = 0

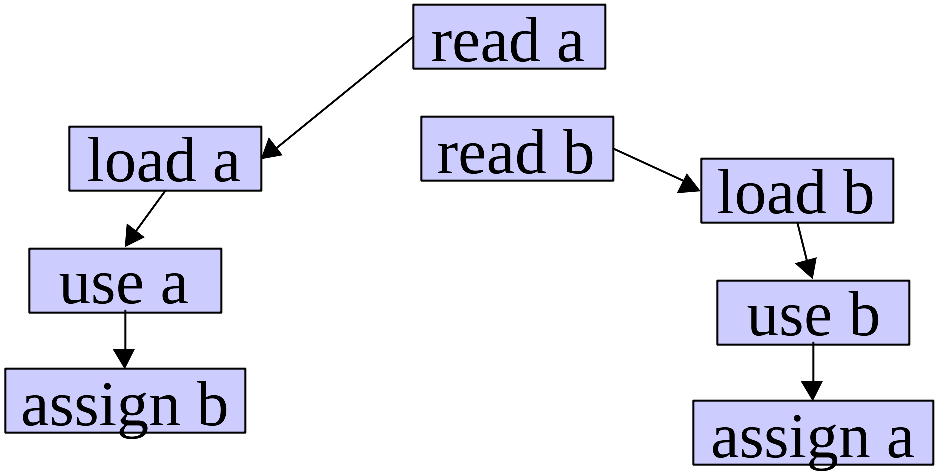
x = a

y = b

b = 1

a = 1

not possible: x = 1 & y = 1



read a

load a

use a

assign b

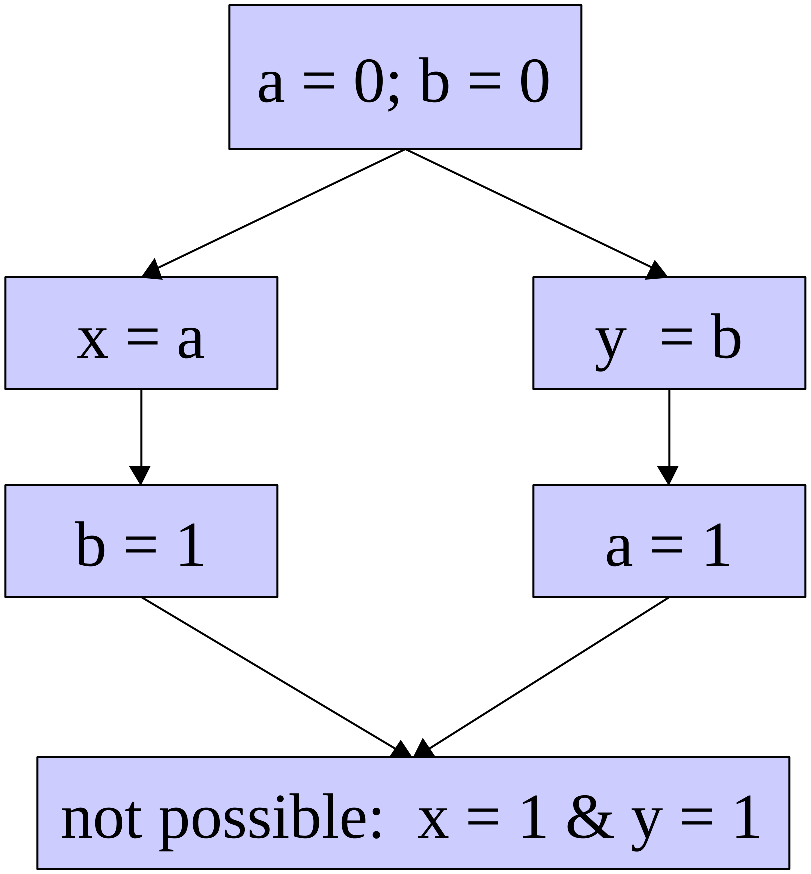
read b

load b

use b

assign a

Memory Model actions, without prescient stores



a = 0; b = 0

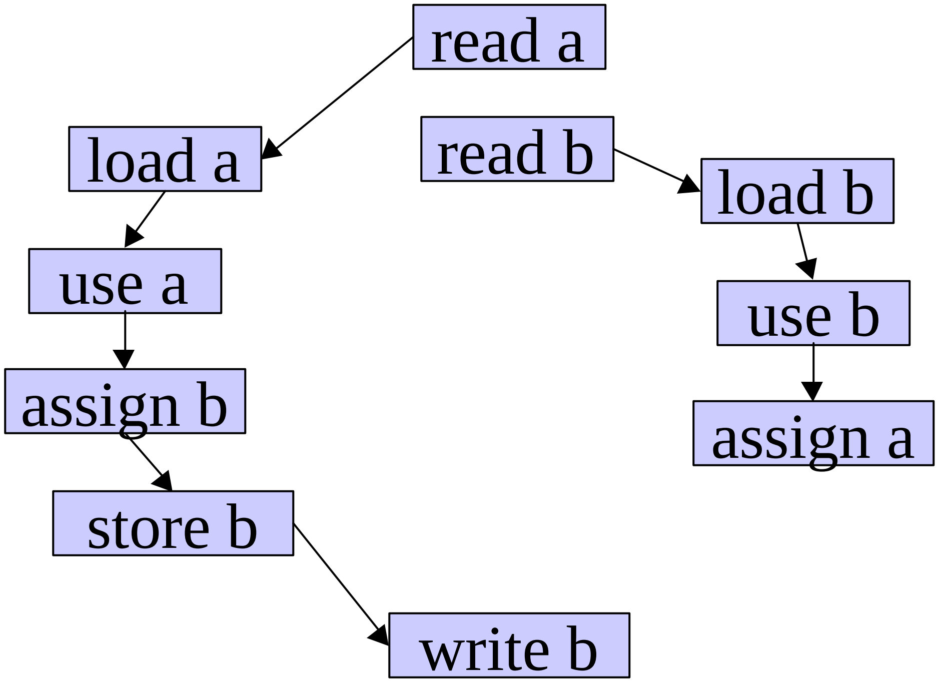
x = a

y = b

b = 1

a = 1

not possible: x = 1 & y = 1



read a

load a

use a

assign b

store b

write b

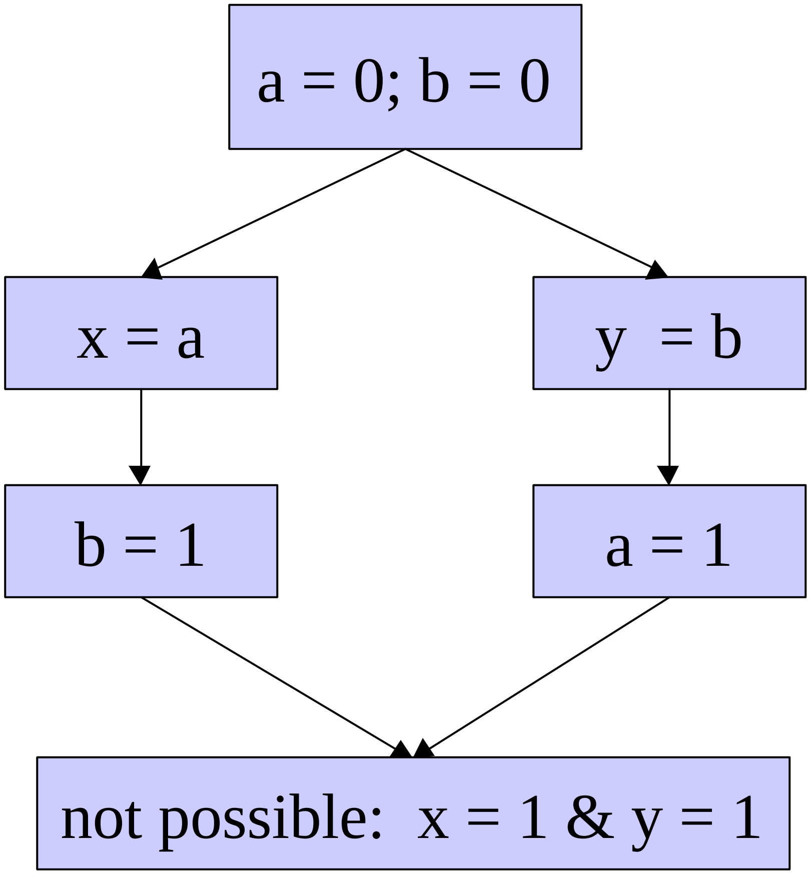
read b

load b

use b

assign a

Memory Model actions, without prescient stores



a = 0; b = 0

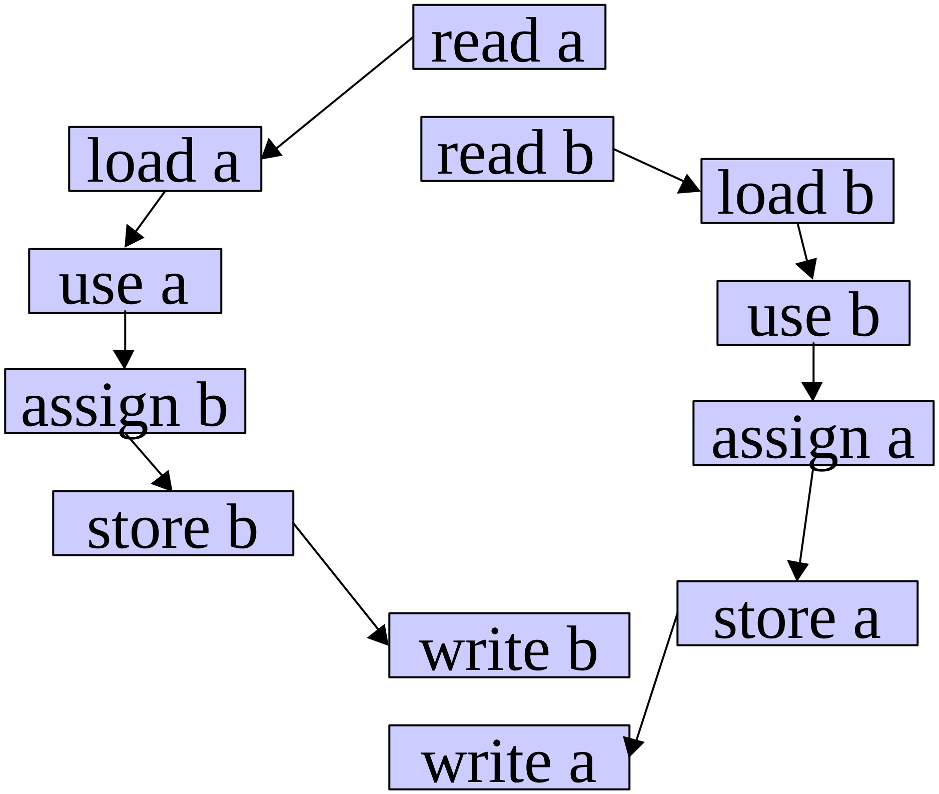
x = a

y = b

b = 1

a = 1

not possible: x = 1 & y = 1



read a

load a

read b

load b

use a

use b

assign b

assign a

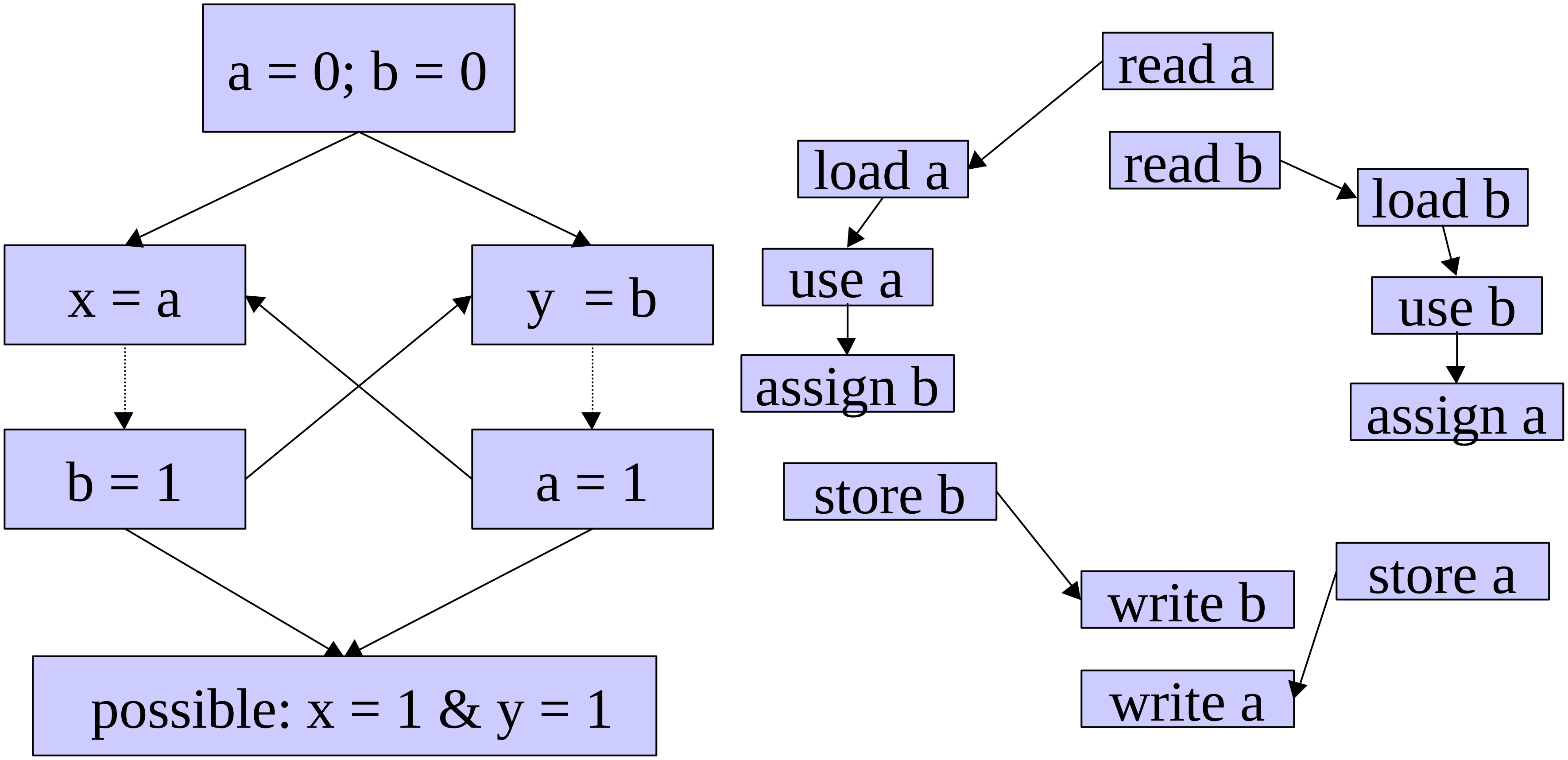
store b

store a

write b

write a

Memory Model actions, *with* prescient stores



a = 0; b = 0

x = a

y = b

b = 1

a = 1

possible: x = 1 & y = 1

read a

load a

use a

assign b

store b

write b

read b

load b

use b

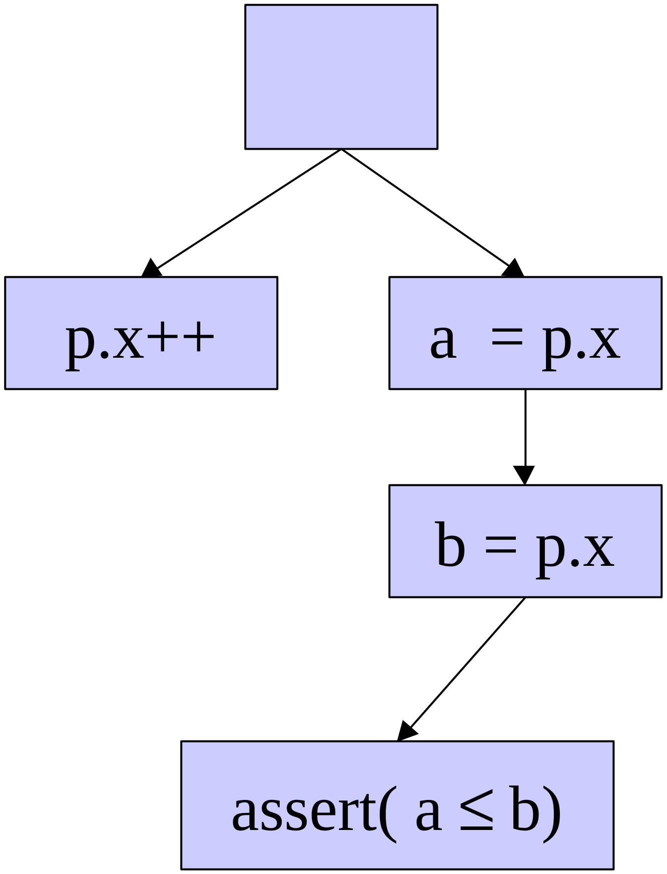
assign a

store a

write a

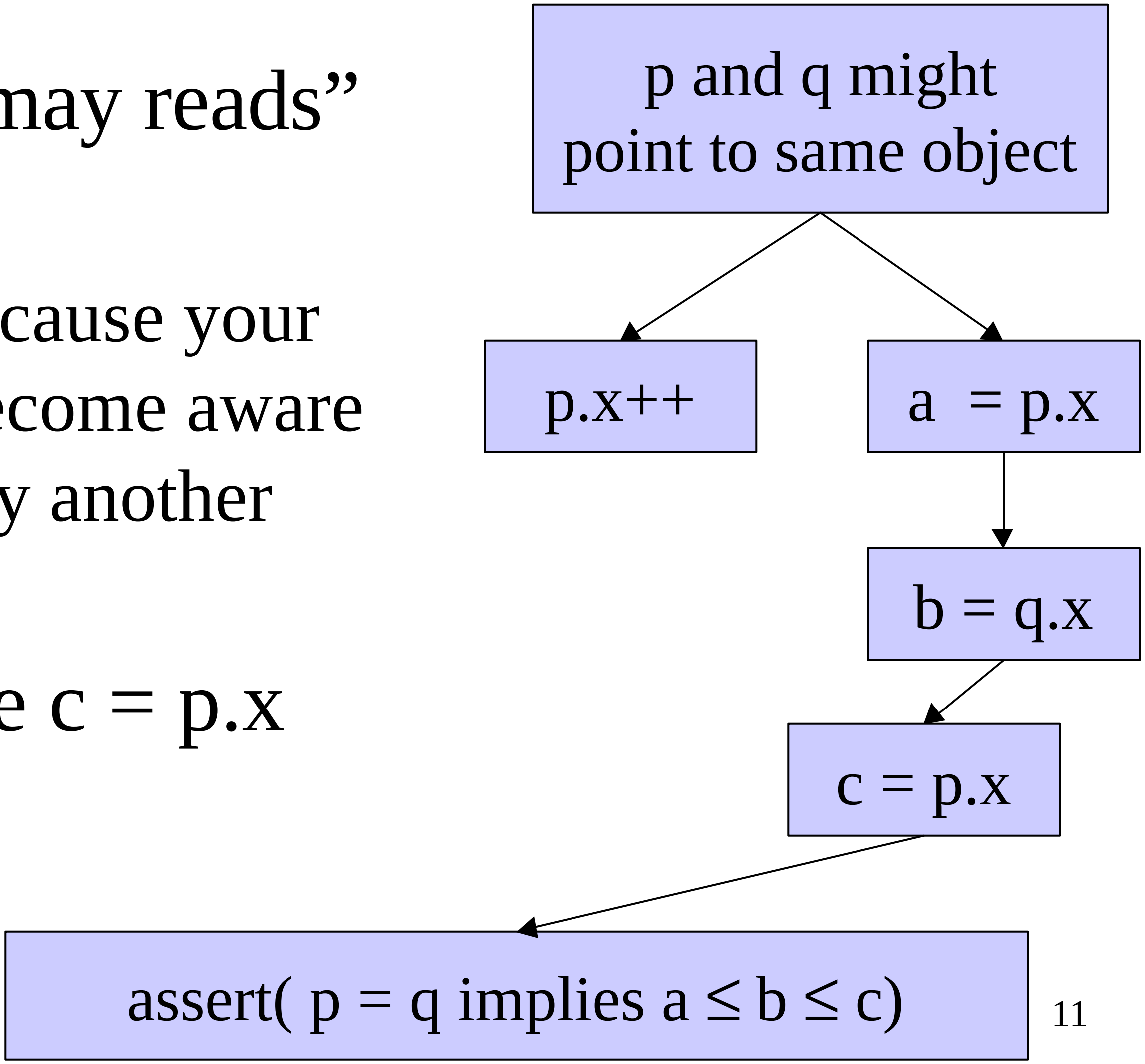
Coherent memory

- Once you see an update by another thread
  - can’t forget that you’ve seen the update
- Cannot reorder two reads of the same memory location



Reads kill reuse

- Must treat “may reads” as kills
  - a read may cause your thread to become aware of a write by another thread
- Can’t replace c = p.x with c = a



p and q might point to same object

p.x++

a = p.x

b = q.x

c = p.x

assert(p = q implies a ≤ b ≤ c)

Most JVM's violate Coherence

- Every JVM I've tested that eliminates redundant loads violates Coherence:
  - Sun's Classic Wintel JVM
  - Sun's Hotspot Wintel JVM
  - IBM's 1.1.7b Wintel JVM
  - Sun's production Sparc Solaris JVM
  - Microsoft's JVM
- Bug # 4242244 in Javasoft's bug parade
  - JVM's don't match spec

Impact on Compiler Optimizations?

- Preliminary work by Dan Scales, DecWRL
- Made reads kill, have side effects
- Better is probably possible, but will require work
- Reads have side effects but can be done speculatively
  - – change intermediate representation

compress	1.18		mpegaudio	1.44
jess	1.03		richards	0.98
cst	1.01		mtrt	1.02
db	1.04		jack	1.06
si	1.03		tsgp	1.36
javac	0.99		tmix	1.11

Should the JMM require Coherence?

- Present in many processor memory models
  - They don’t have the aliasing problem
- Comes at a cost
  - performance
  - rethinking compiler design
- Violated by existing VM’s
  - programmers can’t depend on it

The really bad news

- The JMM is a mess
- I tried to prove that the JMM required exactly coherence
  - Was able to prove it requires coherence
- Tried to prove it requires nothing more
  - Got a *nasty* counter example
- Ordering constraints between memory operations on *different* memory locations

Counter Example

- Consider the following code, and a run in which
  - p and q happen to reference the same object
  - the read of q.x sees a different value than p.x

```
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

use r.y

use p.x

use p/q.x

assign p.x

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

§ 17.3, bullet 1: all use and assign actions must occur in their original order

use r.y

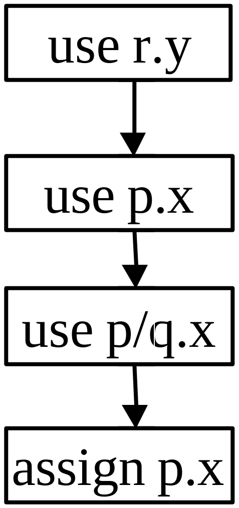
use p.x

use p/q.x

assign p.x

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

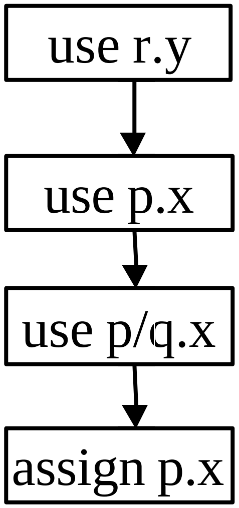
§ 17.3, bullet 1: all use and assign actions must occur in their original order



```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use


use r.y

use p.x

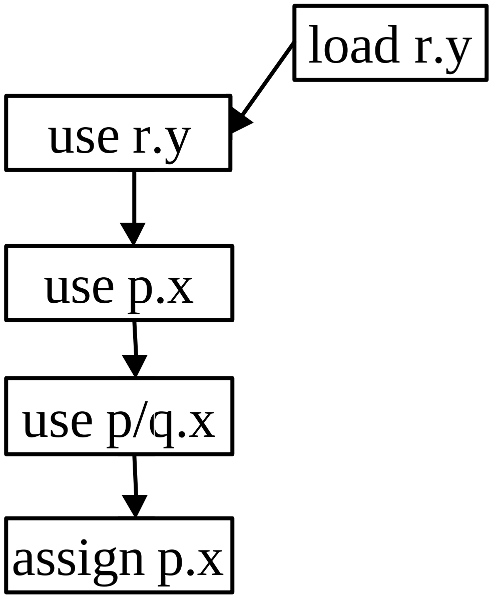
use p/q.x

assign p.x

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use



load r.y

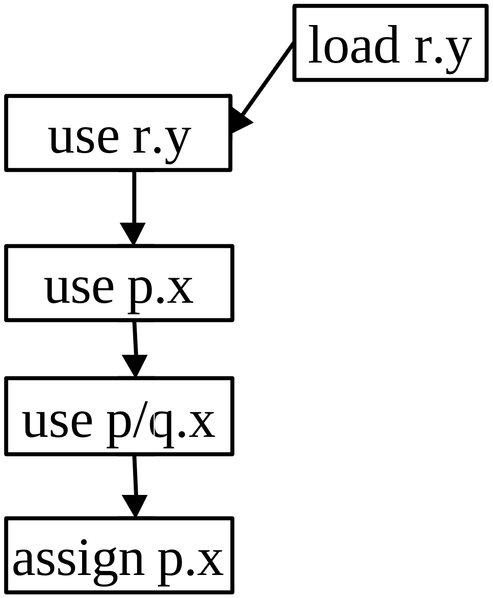
use r.y

use p.x

use p/q.x

assign p.x

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



load r.y

use r.y

use p.x

use p/q.x

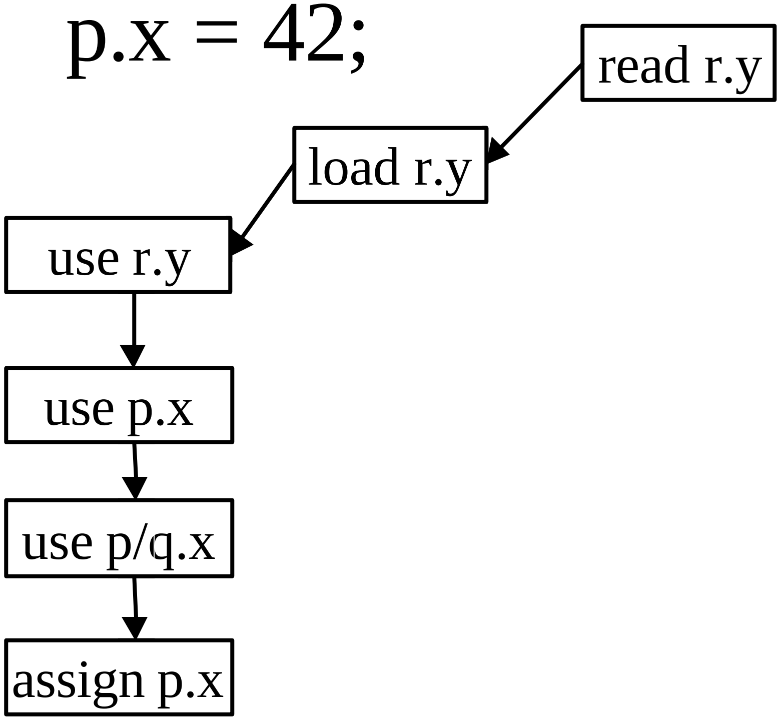
assign p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

use p.x

use p/q.x

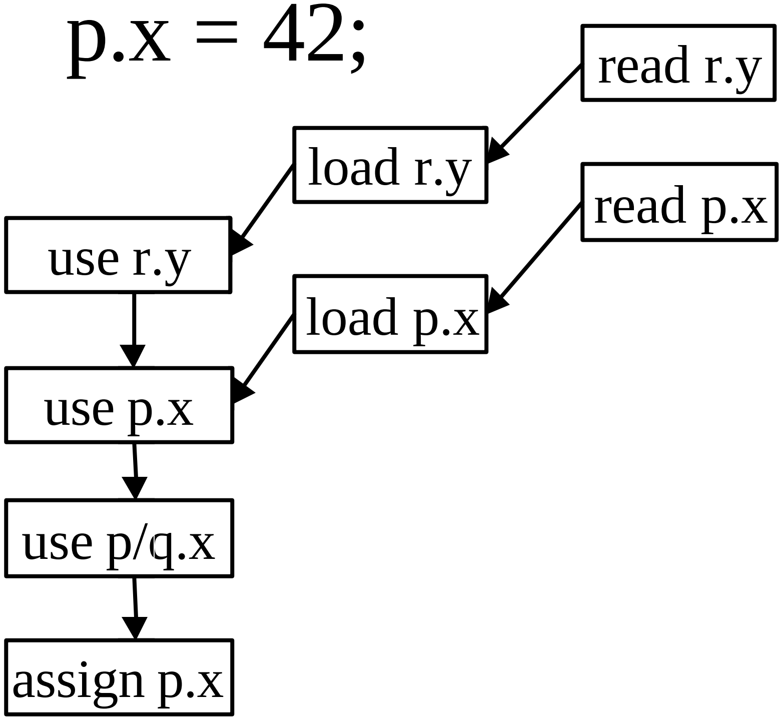
assign p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

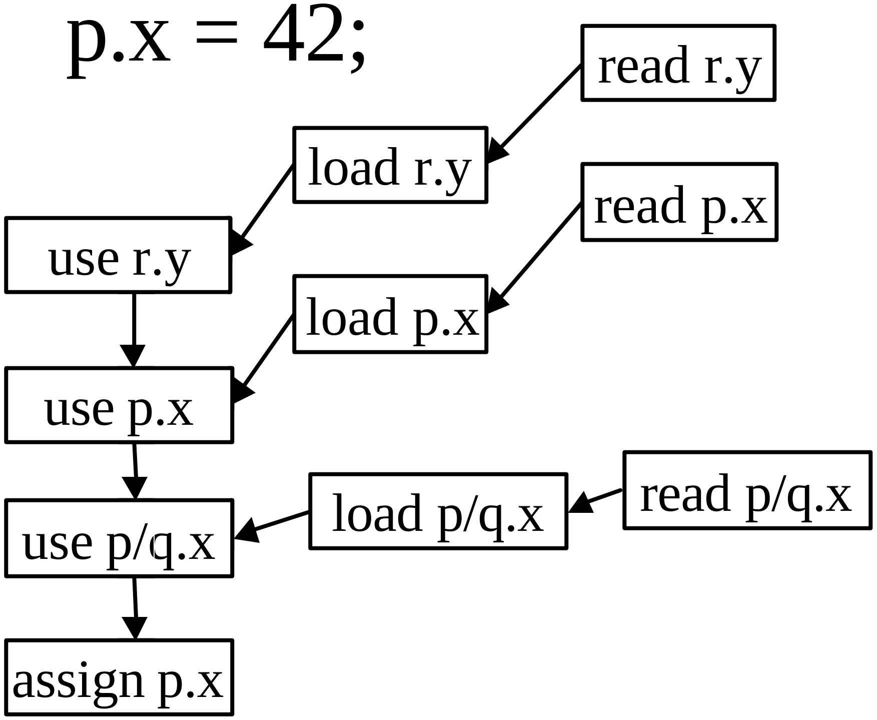
§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read



read r.y

load r.y

read p.x

use r.y

load p.x

use p.x

load p/q.x

read p/q.x

use p/q.x

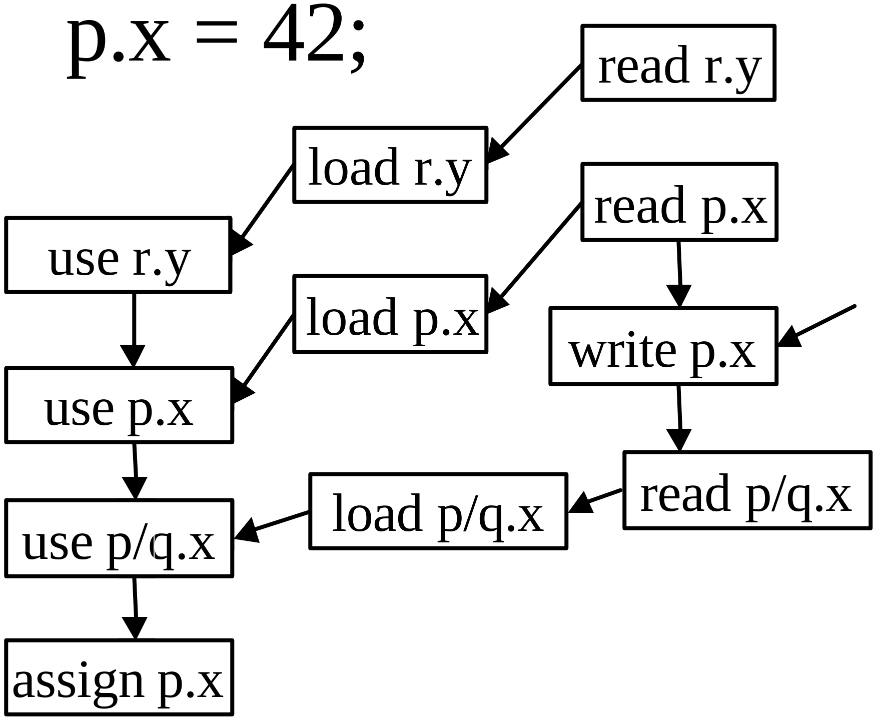
assign p.x

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

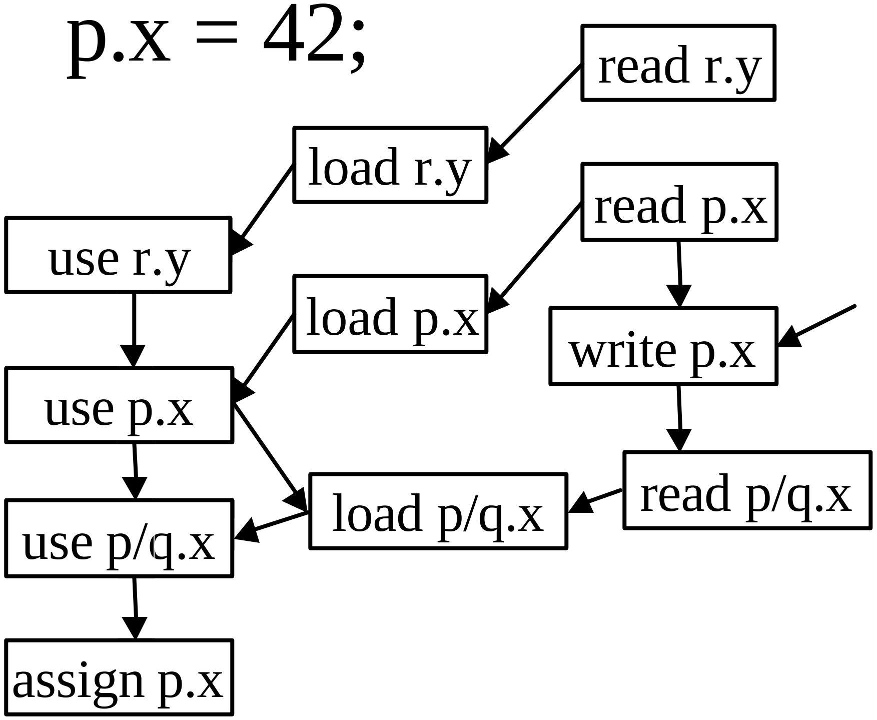


```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

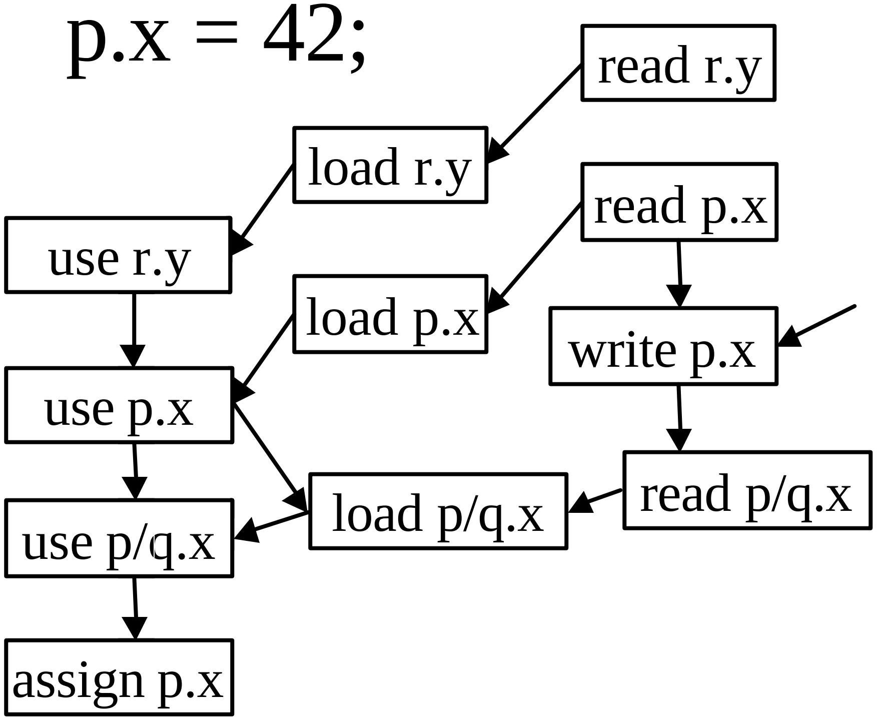
load p/q.x

read p/q.x

use p/q.x

assign p.x

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



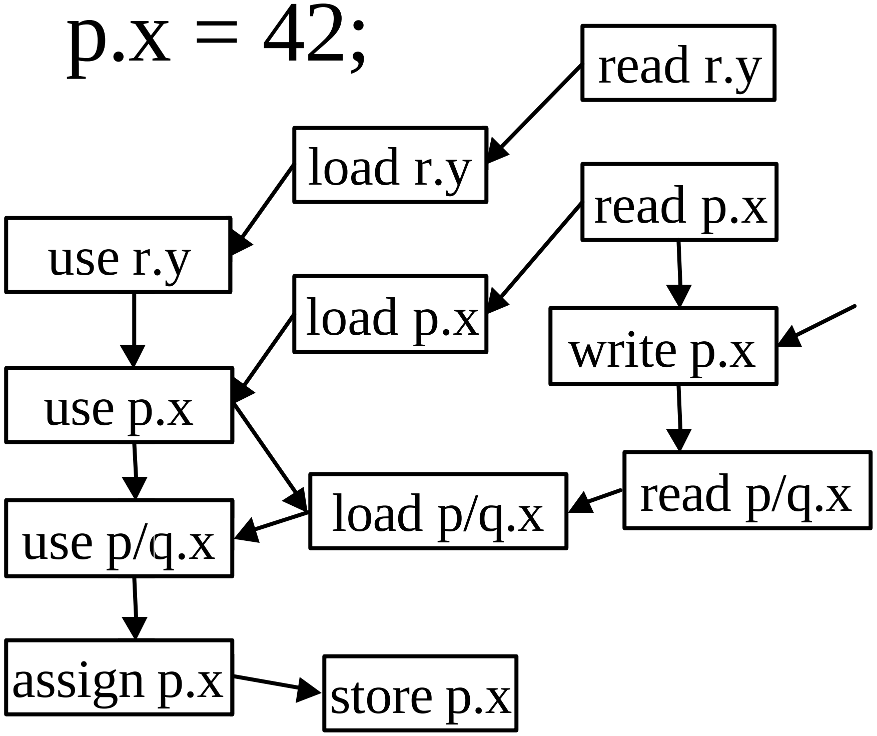
§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination*!), a store must intervene

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

read p/q.x

load p/q.x

use p/q.x

assign p.x

store p.x

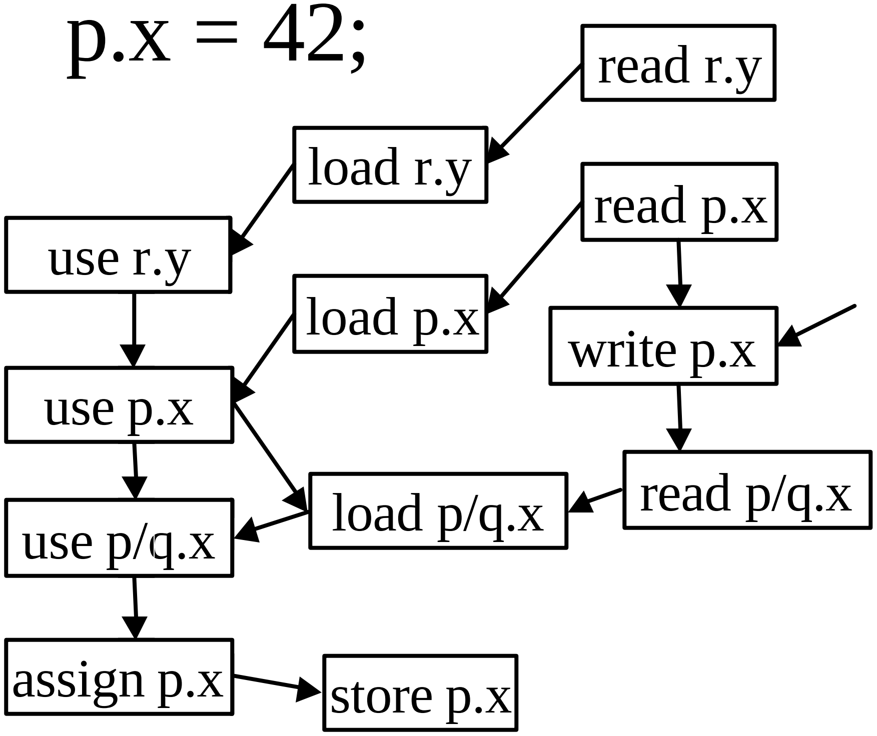
§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination*!), a store must intervene

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

read p/q.x

load p/q.x

use p/q.x

assign p.x

store p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

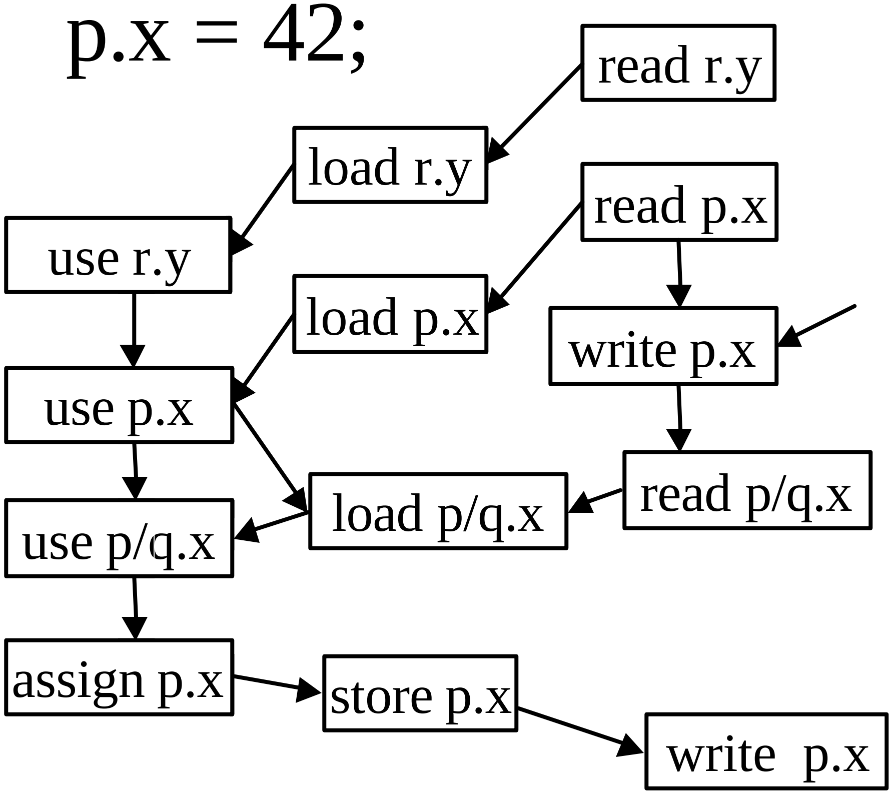
§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination!*), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

read p/q.x

load p/q.x

use p/q.x

assign p.x

store p.x

write p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

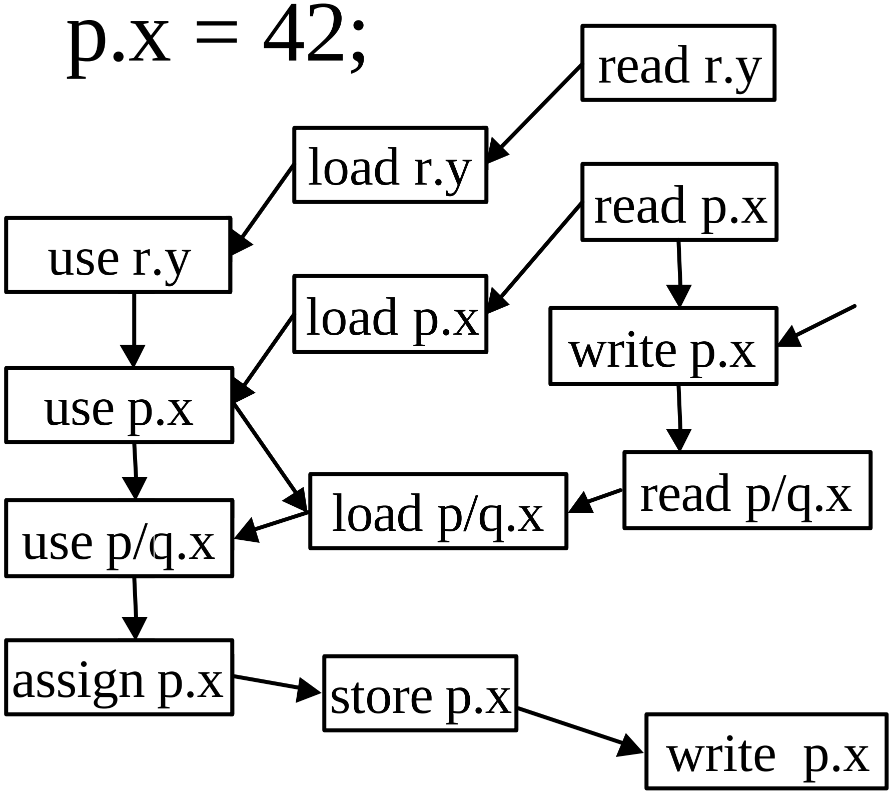
§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination*!), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

use p/q.x

load p/q.x

read p/q.x

assign p.x

store p.x

write p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

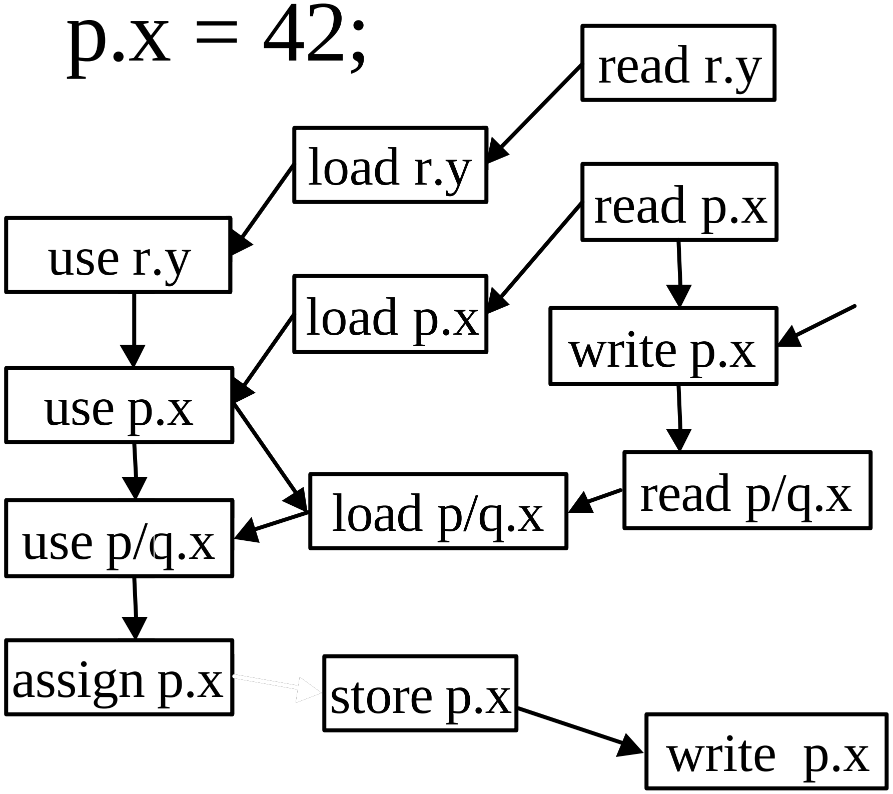
§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination*!), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

§ 17.8: A *prescient* store can occur before the corresponding assign

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```


read r.y
load r.y
use r.y
read p.x
load p.x
write p.x
use p.x
read p/q.x
load p/q.x
use p/q.x
assign p.x
store p.x
write p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

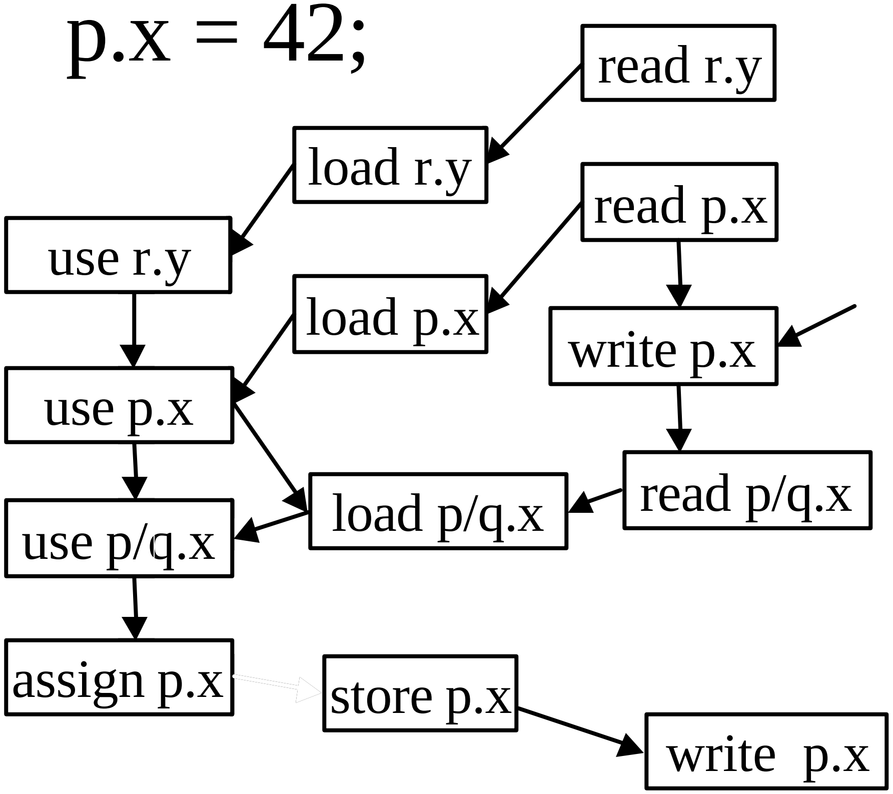
§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination*!), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

§ 17.8: A *prescient* store can occur before the corresponding assign

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

load p/q.x

read p/q.x

use p/q.x

assign p.x

store p.x

write p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

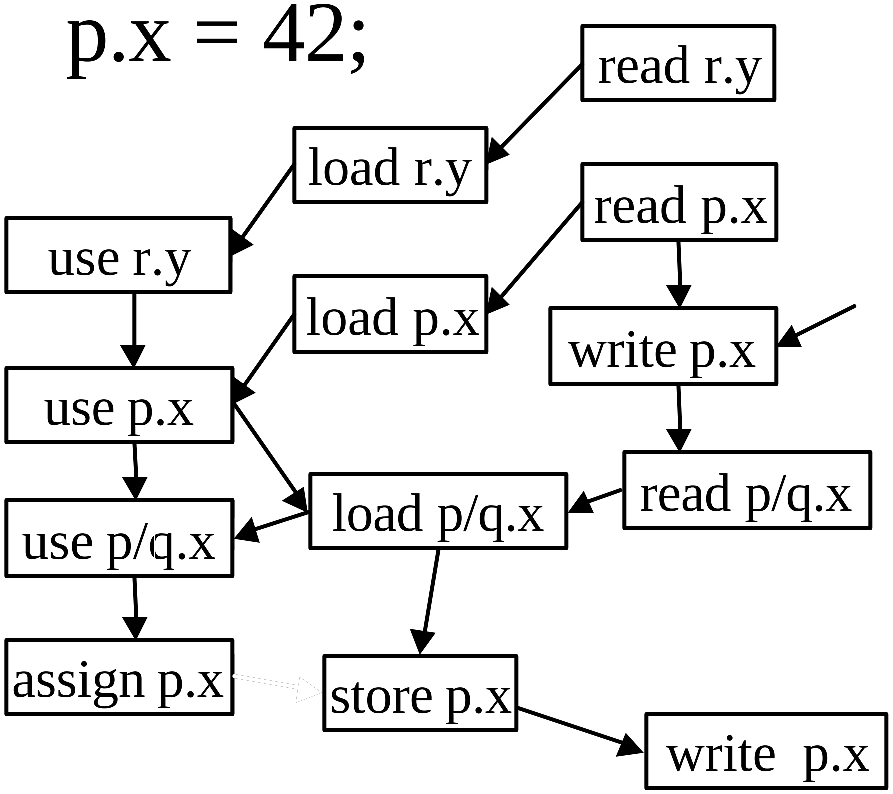
§ 17.6, bullet 1: between an assign and an unlock (*or thread termination!*), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

§ 17.8: A *prescient* store can occur before the corresponding assign

§ 17.8, bullet 3: No load intervenes between the assign and the corresponding prescient store

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

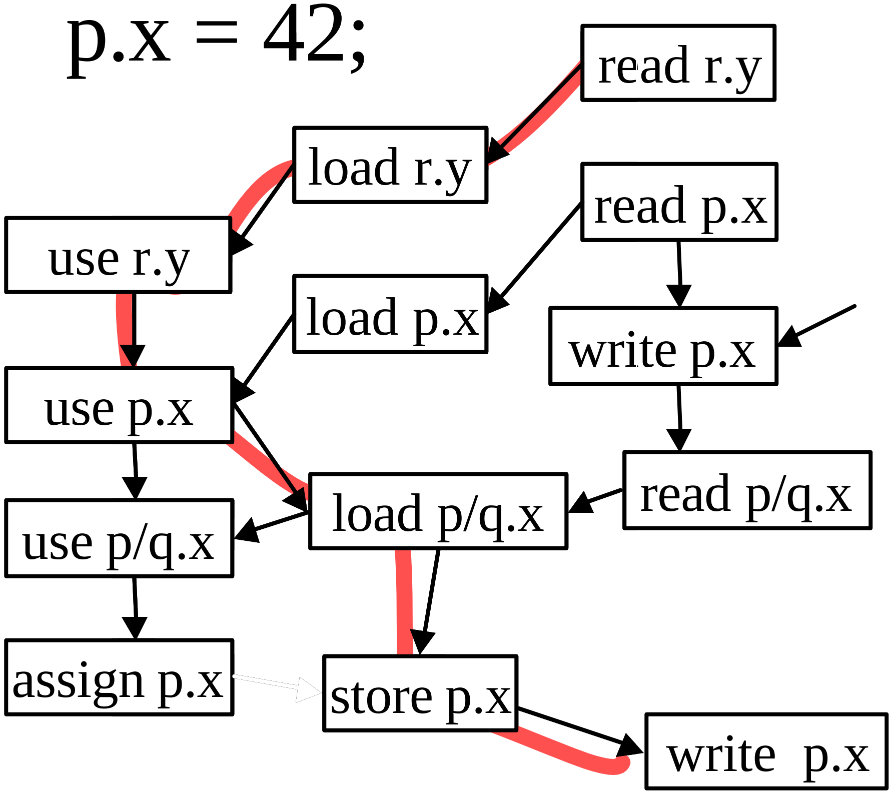
§ 17.6, bullet 1: between an assign and an unlock (*or thread termination*!), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

§ 17.8: A *prescient* store can occur before the corresponding assign

§ 17.8, bullet 3: No load intervenes between the assign and the corresponding prescient store

```
// p & q are aliased
i = r.y;
j = p.x;
// concurrent write to p.x
k = q.x;
p.x = 42;
```



read r.y

load r.y

use r.y

read p.x

load p.x

write p.x

use p.x

read p/q.x

load p/q.x

use p/q.x

assign p.x

store p.x

write p.x

§ 17.3, bullet 1: all use and assign actions must occur in their original order

§ 17.3, bullet 4: a load or assign before a use

§ 17.3, 2nd list of bullets, bullet 1: for each load, a corresponding preceding read

§ 17.6, bullet 1: between an assign and an unlock (*or thread termination!*), a store must intervene

§ 17.3, 2nd list of bullets, bullet 2: for each store, a corresponding following write

§ 17.8: A *prescient* store can occur before the corresponding assign

§ 17.8, bullet 3: No load intervenes between the assign and the corresponding prescient store

Memory barriers required

- Many processors have a relaxed memory model

- This constraint (read of r.y before write of p.x) not supported on most relaxed memory models

- Memory barrier required

Reordering memory references

- Read of r.y must occur before write to p.x
  - but read of r.y could occur after read of p.x

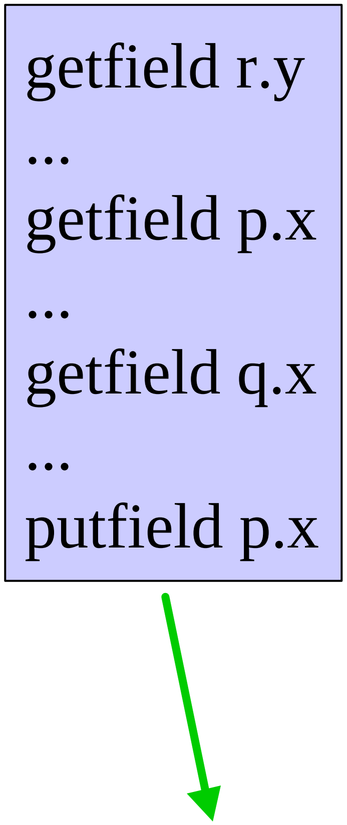
- Can we reorder read of r.y and p.x as a compiler optimization
  - and use a simple internal representation
    - e.g., bytecode

Reordering memory refs in IR

Reordering memory refs in IR

```
getfield r.y
...
getfield p.x
...
getfield q.x
...
putfield p.x
```

Reordering memory refs in IR



```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

Reordering memory refs in IR

getfield r.y

…

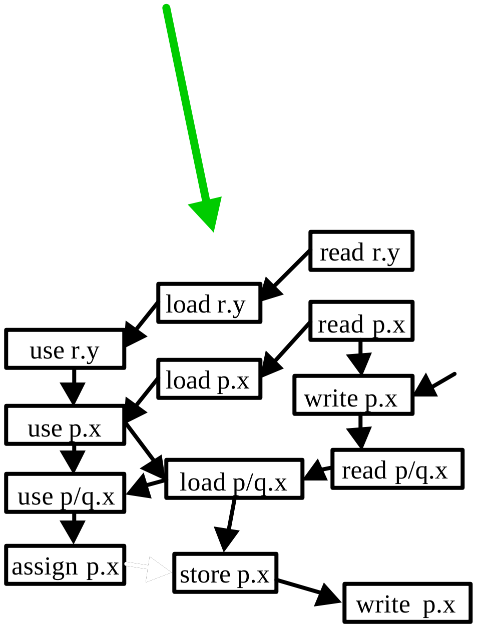
getfield p.x

…

getfield q.x

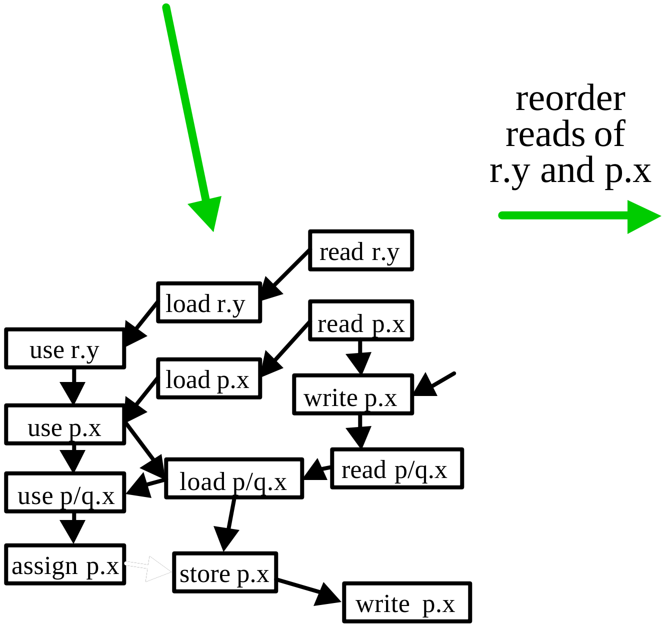
…

putfield p.x



Reordering memory refs in IR

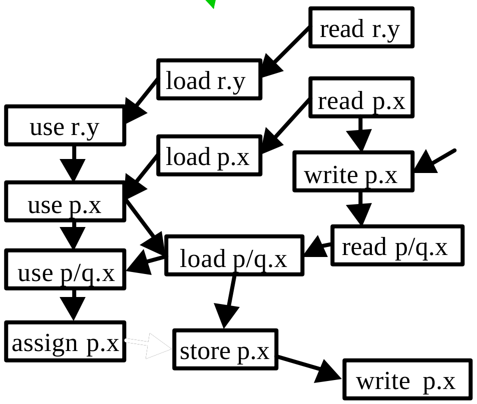
```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```



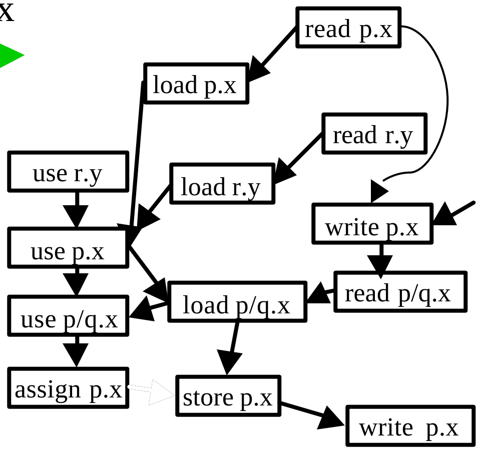
reorder
reads of
r.y and p.x

Reordering memory refs in IR

getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x

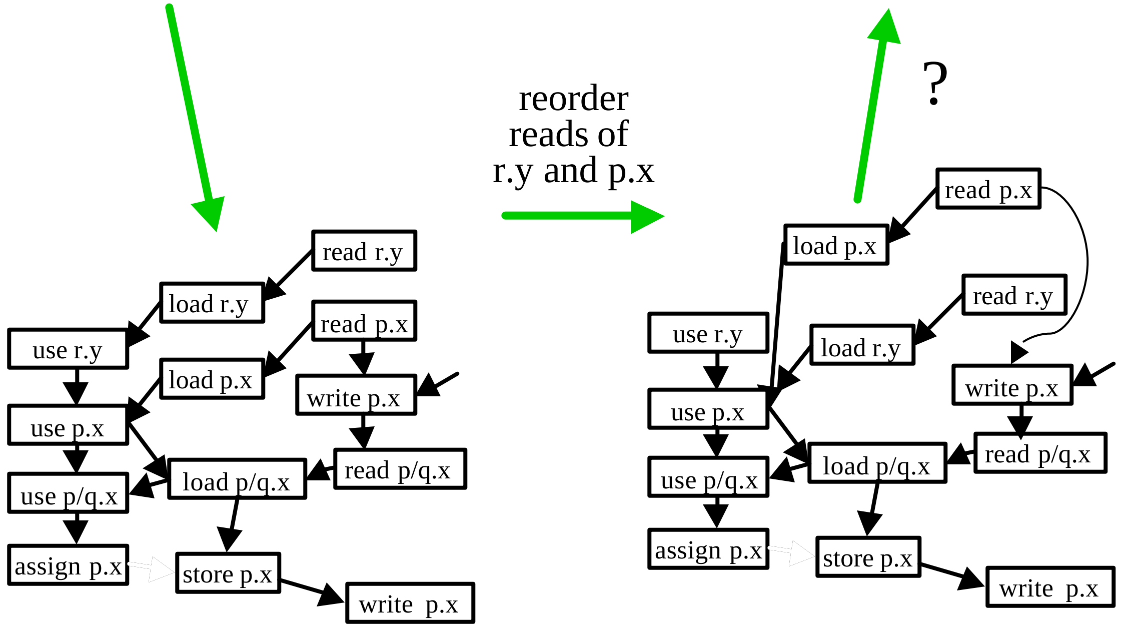


reorder
reads of
r.y and p.x



Reordering memory refs in IR

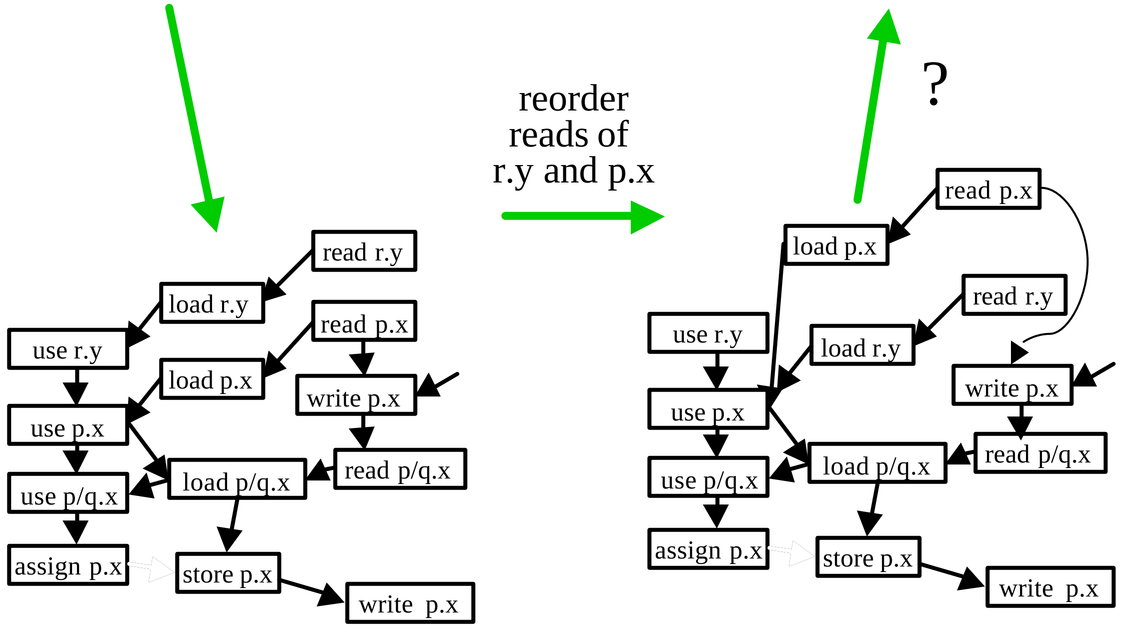
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x



Reordering memory refs in IR

```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

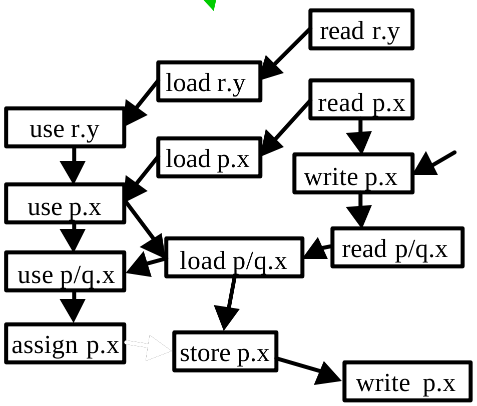


reorder reads of r.y and p.x

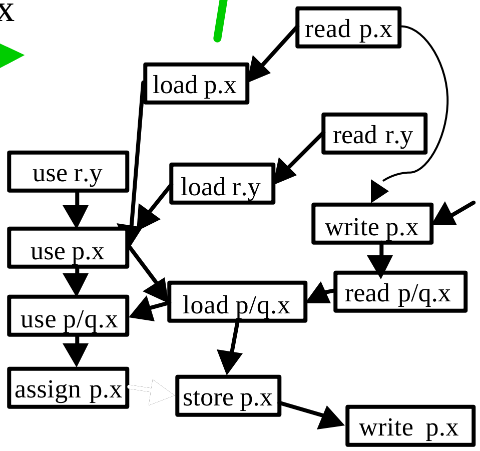
?

Reordering memory refs in IR

```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```



reorder reads of r.y and p.x

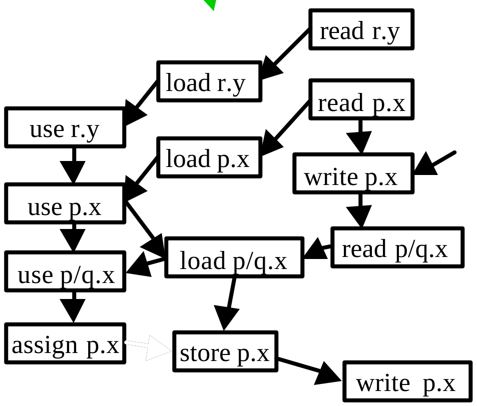


?

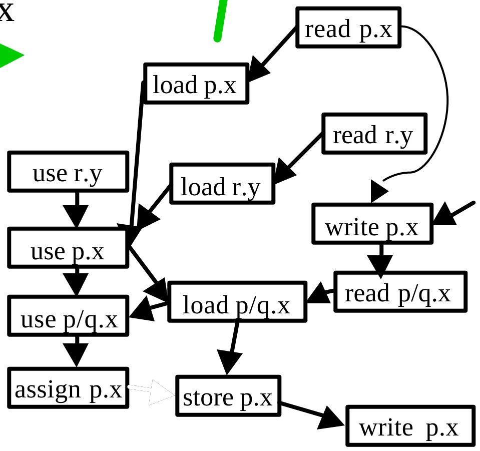
```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

Reordering memory refs in IR

```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```



reorder reads of r.y and p.x



?

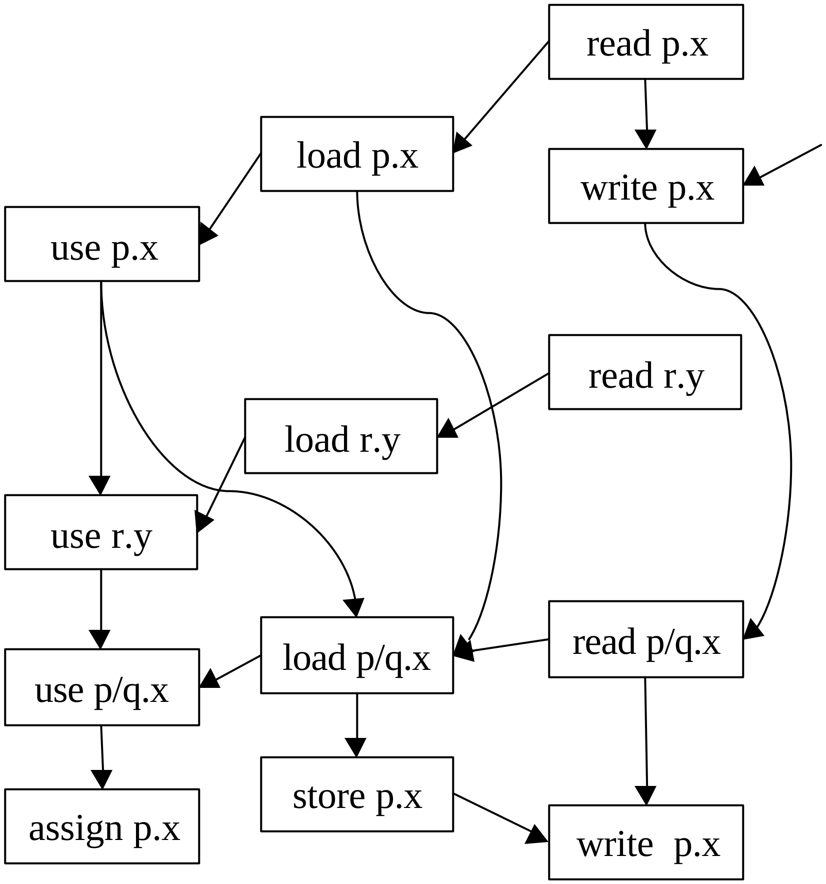
```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

?

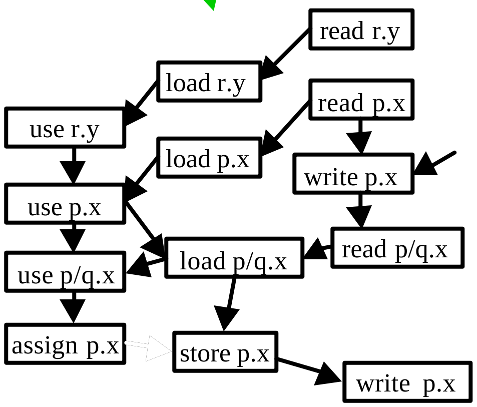
Reorder uses of p.x and r.y

- Reorder the bytecodes for reading p.x and r.y
  - reorder uses

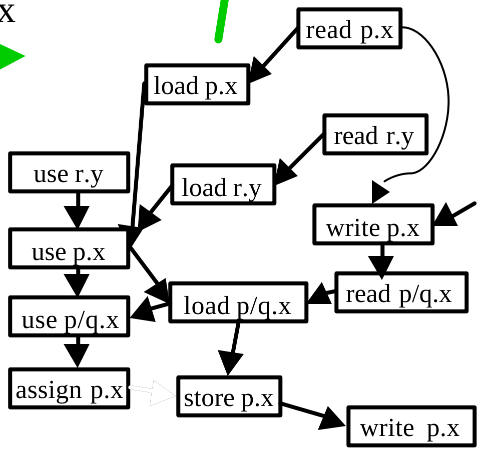
- Don’t need to perform the read of r.y before the write of p.x



getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x

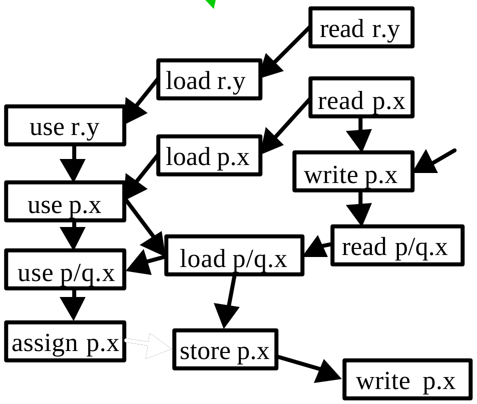


reorder
reads of
r.y and p.x



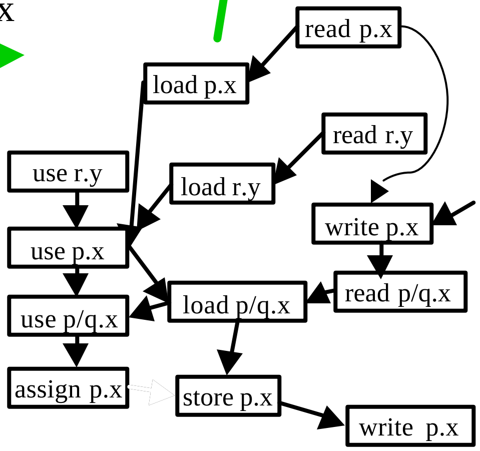
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x

getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x



reorder
reads of
r.y and p.x

getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x



getfield r.y

…

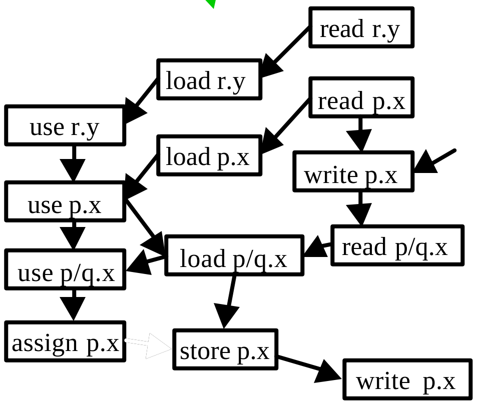
getfield p.x

…

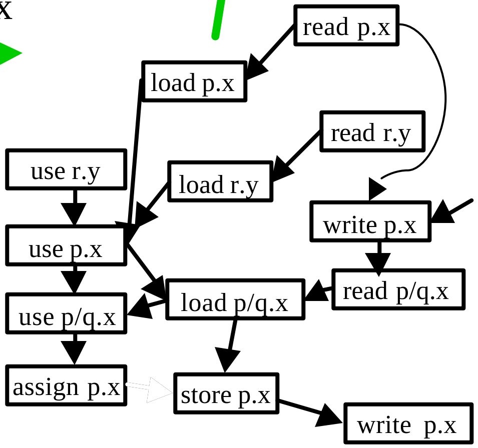
getfield q.x

…

putfield p.x



reorder
reads of
r.y and p.x



getfield p.x

…

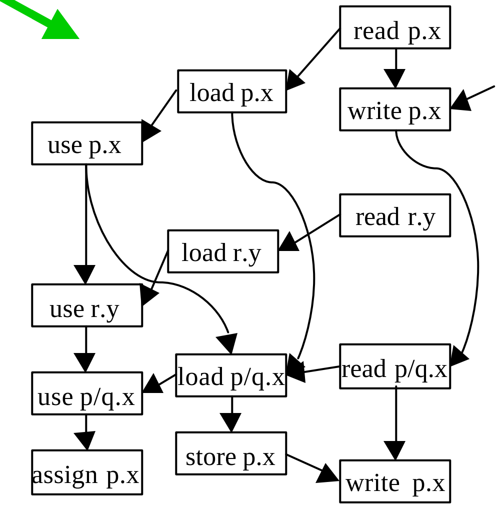
getfield r.y

…

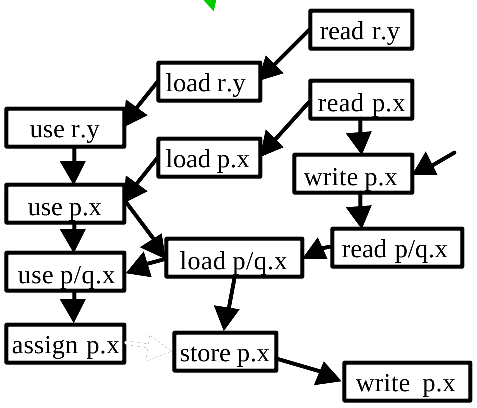
getfield q.x

…

putfield p.x

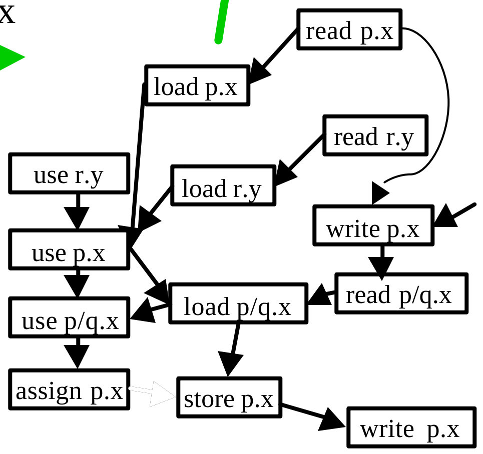


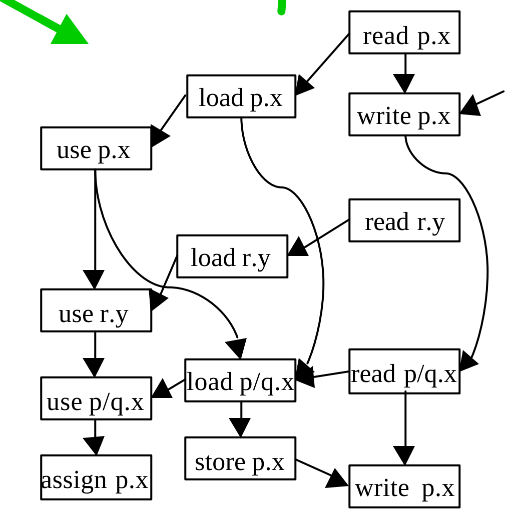
```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```



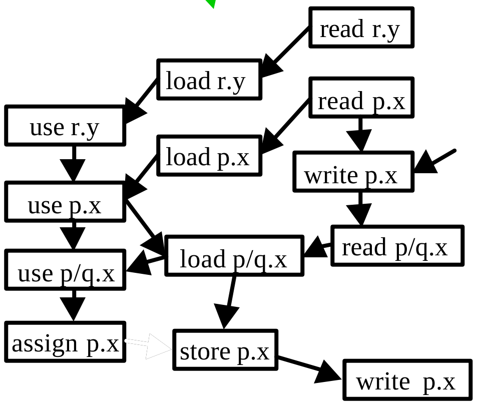
reorder
reads of
r.y and p.x

```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

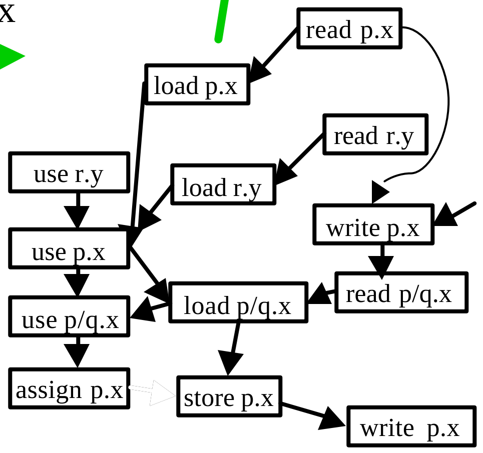




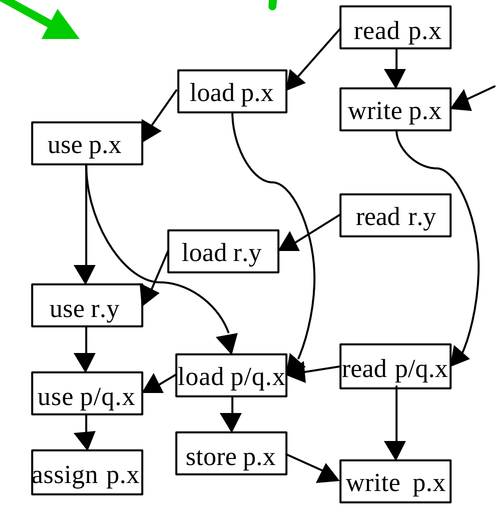
```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

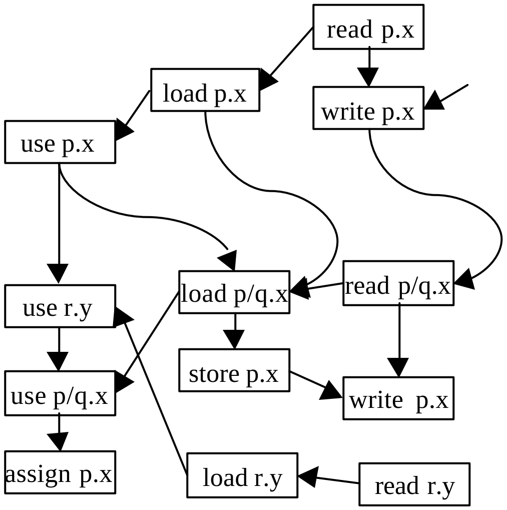


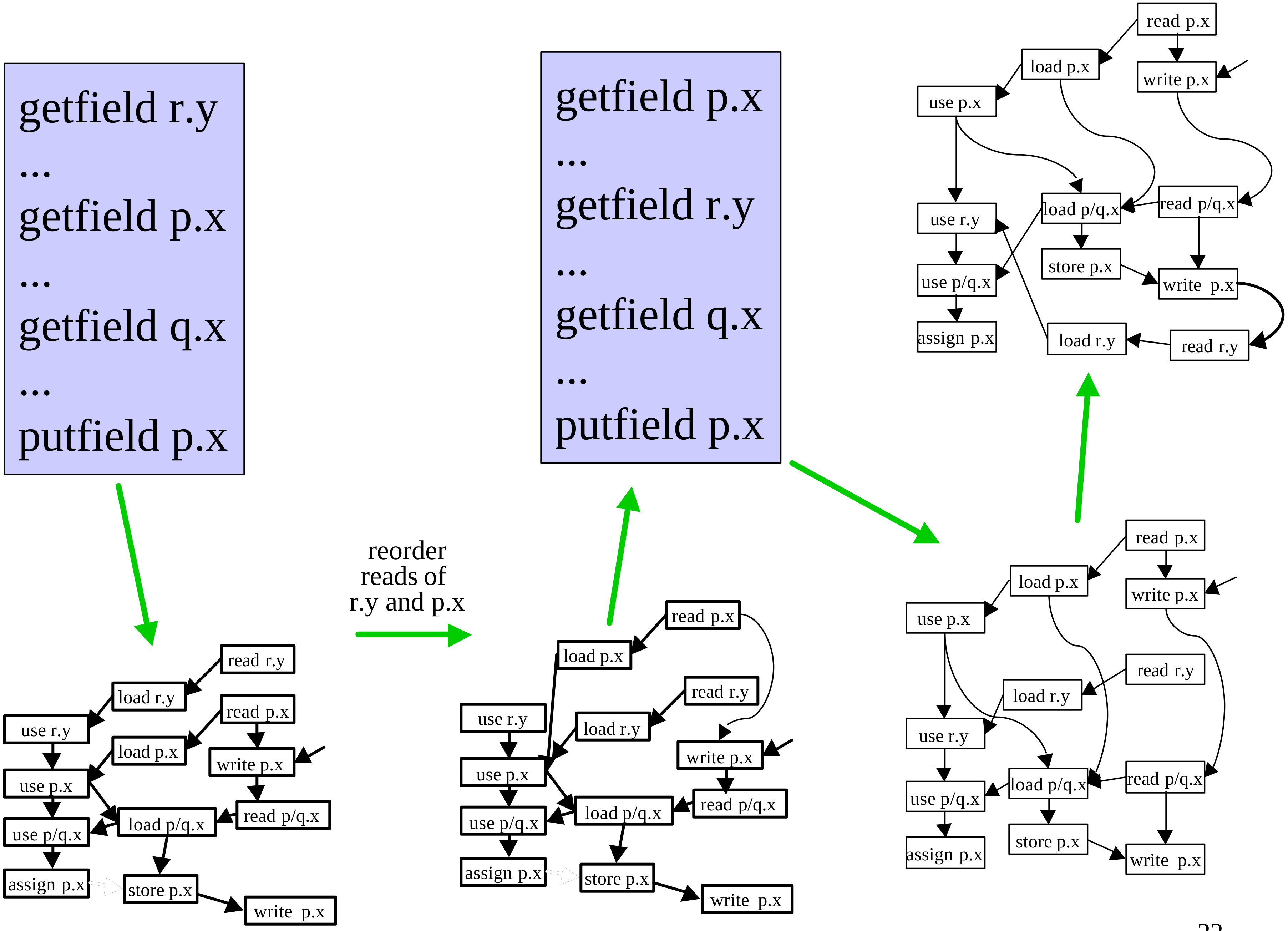
reorder
reads of
r.y and p.x



```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```



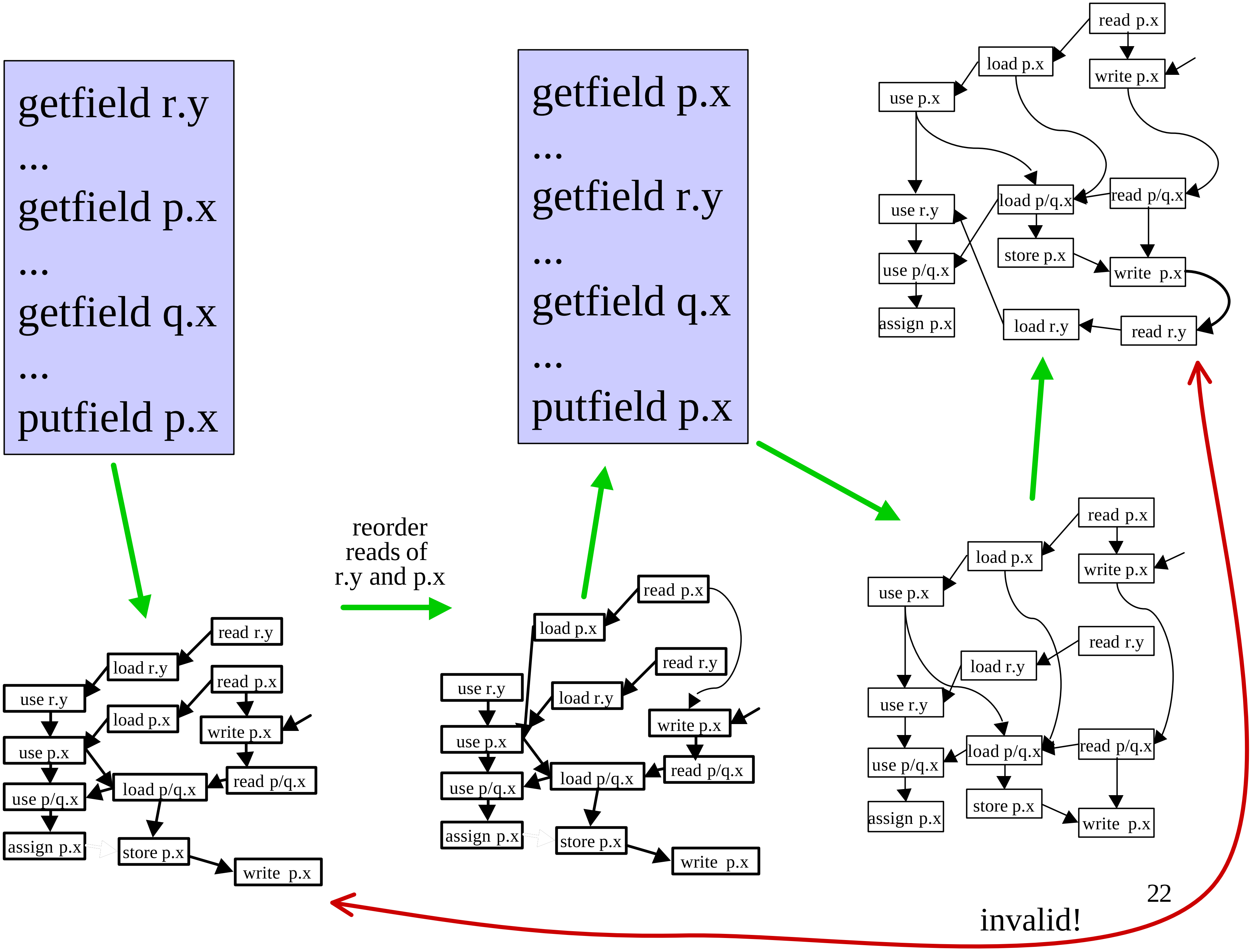




```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

reorder reads of r.y and p.x

```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

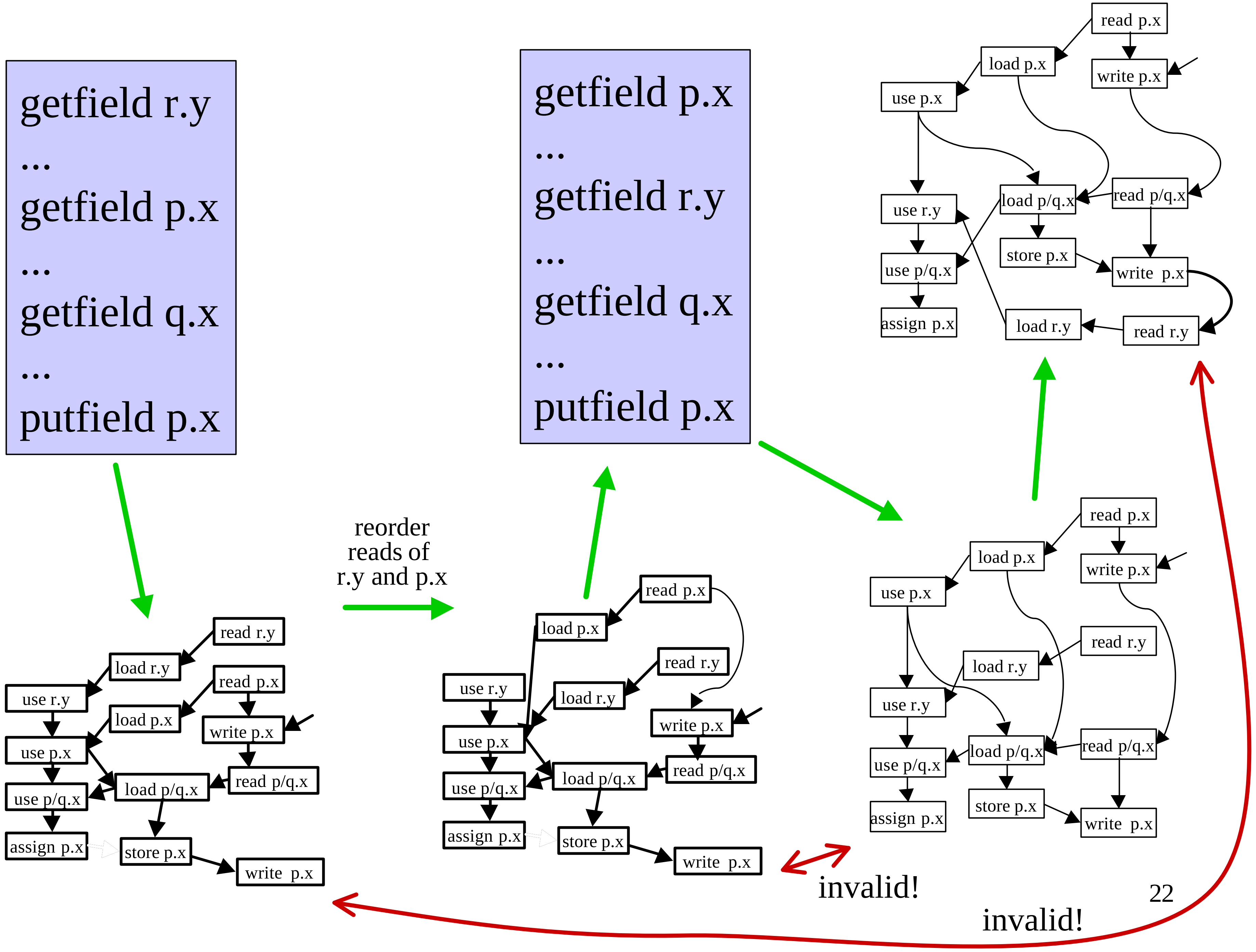


```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

reorder
reads of
r.y and p.x

```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

invalid!



```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

reorder reads of r.y and p.x

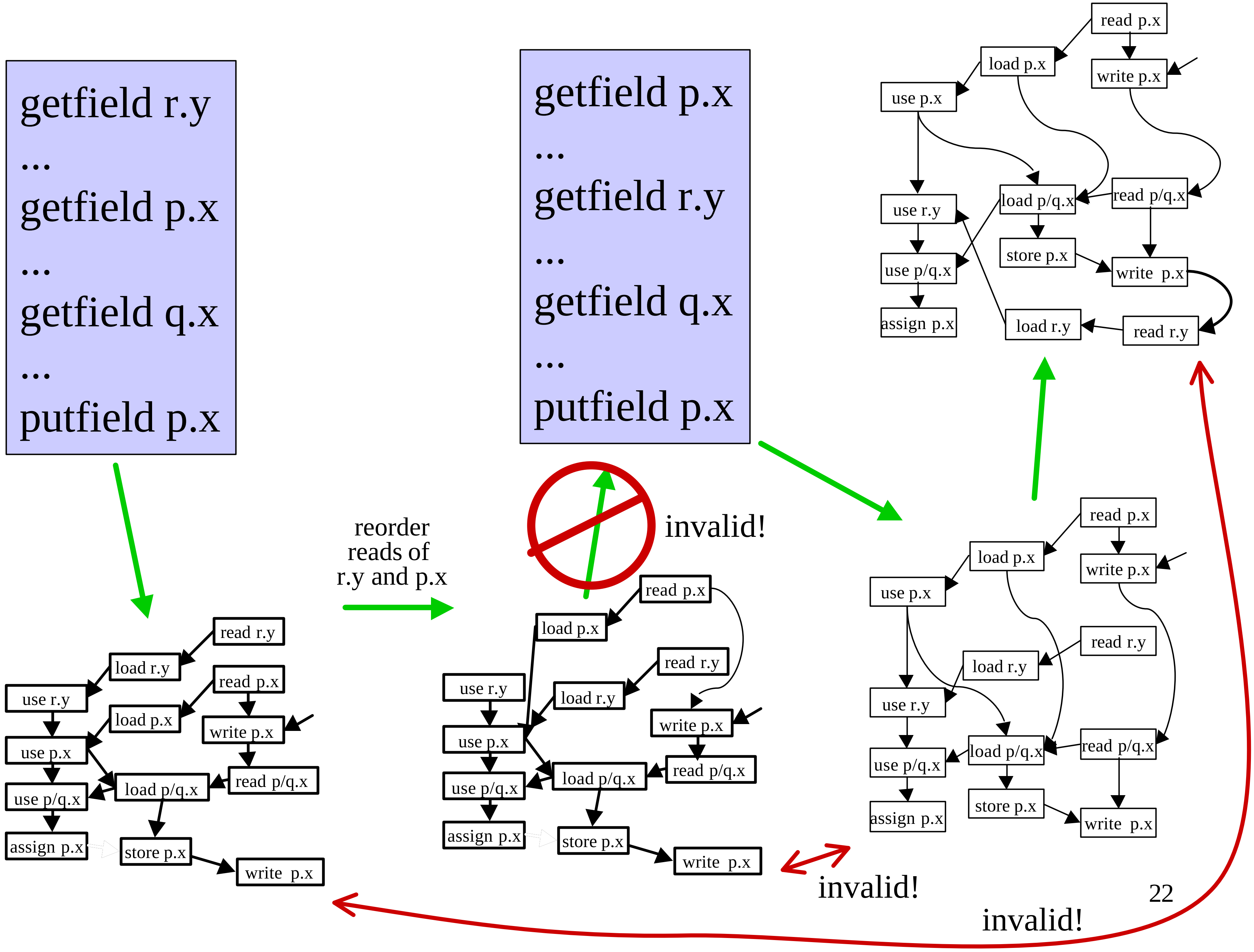
```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

invalid!

invalid!

```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

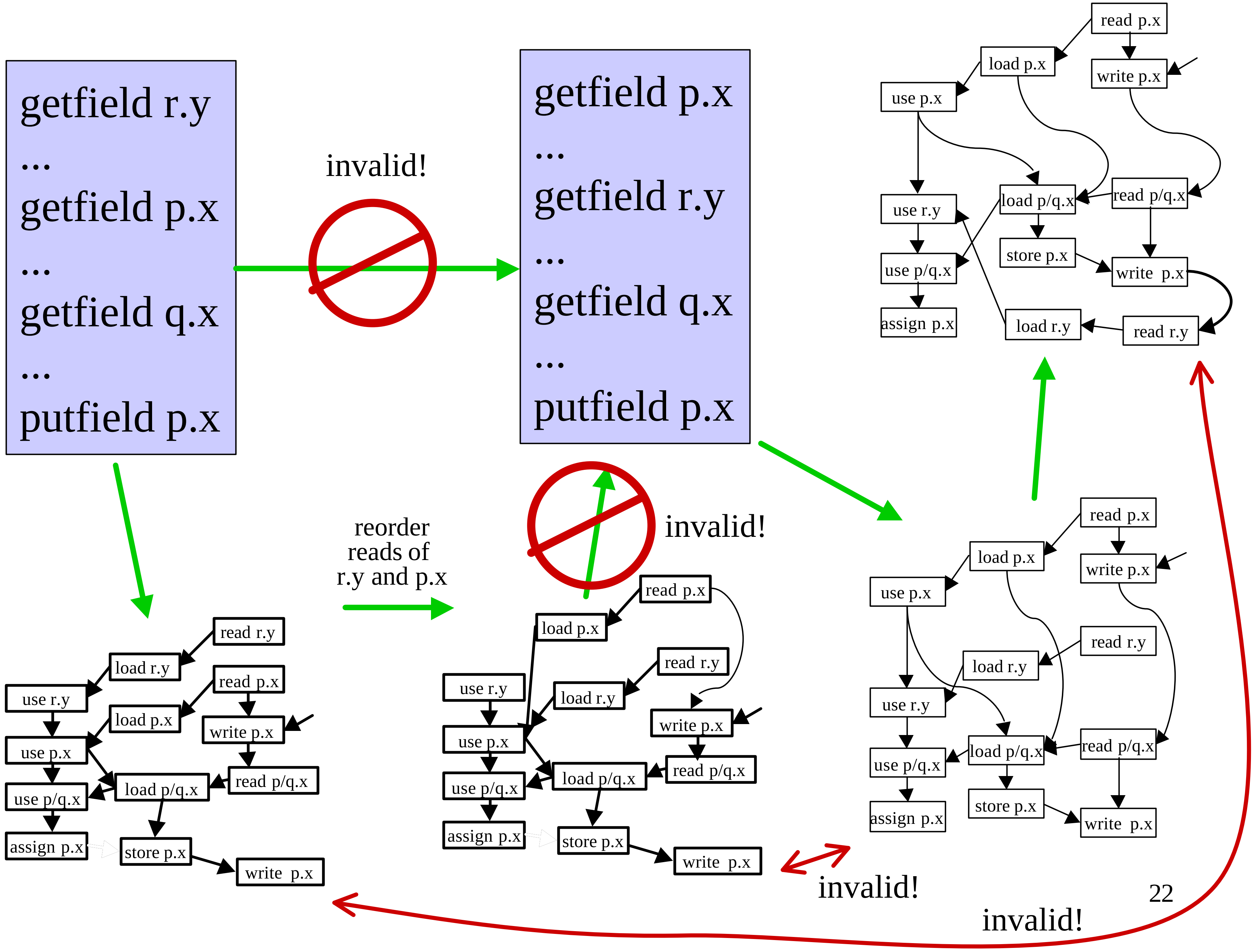


reorder reads of r.y and p.x

invalid!

invalid!

invalid!



```
getfield r.y
…
getfield p.x
…
getfield q.x
…
putfield p.x
```

```
getfield p.x
…
getfield r.y
…
getfield q.x
…
putfield p.x
```

invalid!

reorder
reads of
r.y and p.x

invalid!

invalid!

invalid!

If you respect the current JMM

- If you want to do *any* reordering of memory references
  - *Must* use JMM’s double indirection in your compiler’s intermediate representation
  - *Must* reason about whether there exists any downstream component that might do a reordering that, when composed with your reordering, is illegal
    - e.g., the processor

This is *ugly*

- I went over it with Guy Steele
  - agrees (tentatively) that with constraints I derived
- *Definitely* not intended
- Argument for scrapping the current JMM
  - patches to fix it will just make it harder to understand
    - just hide the bugs, not fix them

What do we need/want?

- Need to be able to reason about whether compiler transformations are legal with respect to memory model
  - The clearer, the better
- Want to avoid changing/discarding compiler techniques for non-synchronized code

My proposed Java Memory Model

- Actions within a thread can be reordered in any way that respects the data dependencies within that thread
- Global behavior some interleaving of reordered actions
- Also need to allow for scalar replacement (equivalently write buffers) and dead store elimination
- Plus stuff for locks and volatile variables

The Java memory model is too weak

- Idioms used by many programmers aren’t thread safe, but
  - are widely used (including in Sun’s JDK)
  - are thought to be safe by many
  - create initialization-safety issues
  - create type-safety issues in implementation
- Joshua Bloch of Javasoft first highlighted this issue
- (Only issue for multiprocessor systems)

Could see reference to object before fields of object

- This isn’t guaranteed to work

```
class A {
 Point p;
 synchronized void setPos(int x, int y) {
    p = new Point(x, y);
    }
 double distanceToOrigin() {
    // not synchronized
    Point q = p;
    return Math.sqrt(q.x*q.x + q.y*q.y);
    }
}
```

Reading Garbage

- An implementation issue, not a spec issue
- What if another thread sees an object before the writes that initialize the objects fields to default values (e.g., null)?
  - major type safety breach
  - not a problem if objects allocated out of pre-zeroed memory
- What if another thread sees an object before the writes that initialize the object header?

Are constructors special?

- writes in constructor:

```
 synchronized void setPos(int x, int y) {
      p = new Point(x, y);
      }
```

- writes outside of constructor:

```
 synchronized void setPos(int x, int y) {
      Point tmp = new Point();
      tmp.x = x;
      tmp.y = y;}
      p = tmp;
      }
```

My suggestion for fixing this

- The system must not reorder:
  - A write during the construction of object X
  - A store of a reference to X
    - after the constructor for X has terminated

- Read side handled already by requiring data dependence be respected

Commentary

- Doesn’t handle the general case
  - outside of constructors
- Doesn’t use barriers
  - lightweight object construction becomes heavyweight
    - e.g. unboxed complex numbers
  - memory barriers harder to eliminate through analysis
- Leverages off of barriers that might be required for garbage collection

Summary

- No one understands the current model
- Java memory model is too strong
  - Coherence
  - Hairball
- Java memory model is too weak
  - initialization safety issues
  - type safety implementation issue
- Consensus is that it needs to be replaced
  - but with what?

Questions?