PURDUE UNIVERSITY

Principles of Concurrency

Lecture 13
Memory Models: Java

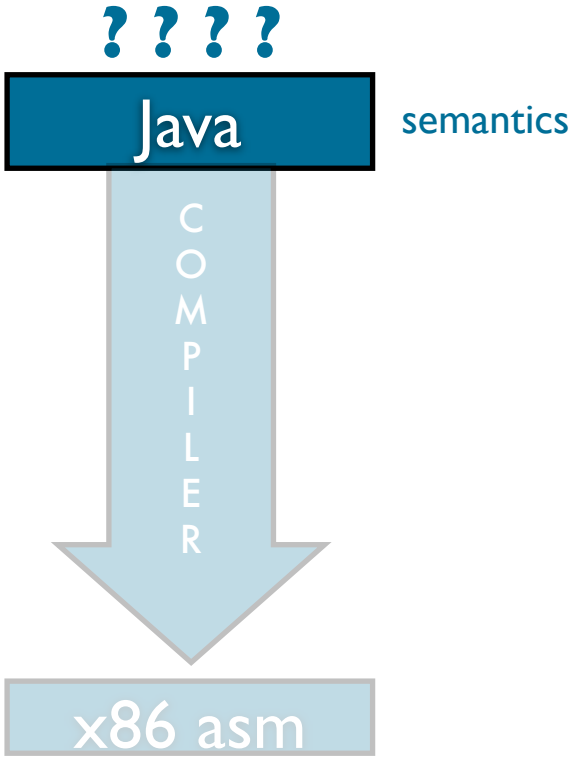
Based on:

- On Validity of Program Transformations in the Java Memory Model, (Sevcik and Aspinall, ECOOP'08)

Java Semantics

- The sequential semantics of Java is well-specified and well-understood.

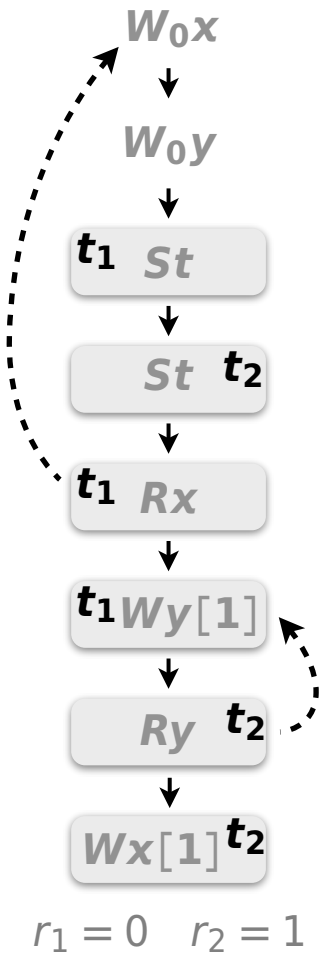
- About Java concurrency:
  - ★ Do we have a formal definition?
  - ★ Do people understand it?



First Try: Sequential Consistency

- During execution, memory actions should appear to execute one at a time in an imaginary interleaving of actions on a shared memory.
- In particular, reads of a shared variable should return the value written most recently to the memory.
- Example:

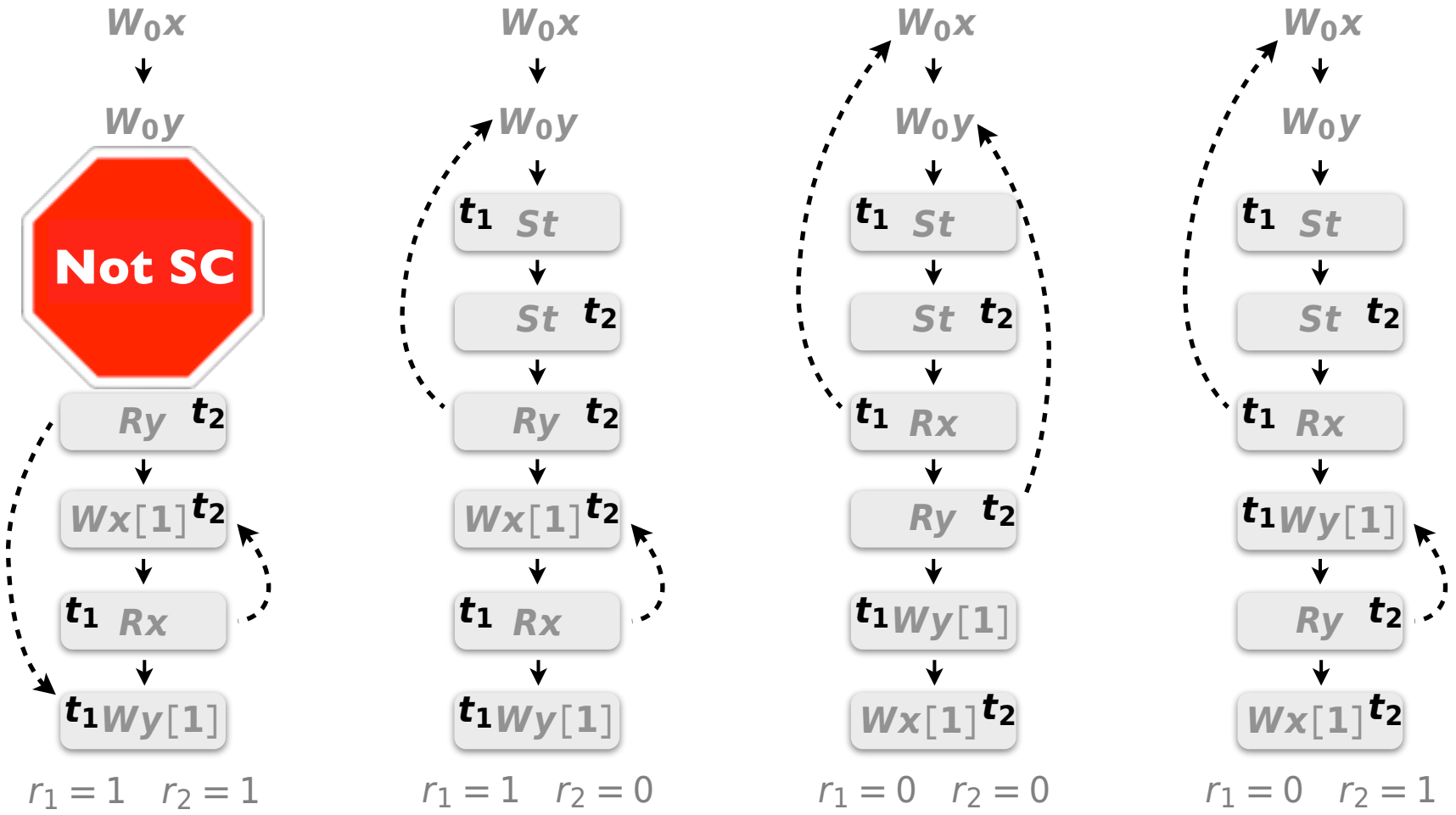
$x \leftarrow 0; y \leftarrow 0$	
$r_1 \leftarrow x$ $y \leftarrow 1$	$r_2 \leftarrow y$ $x \leftarrow 1$



Some More SC Executions

$x \leftarrow 0; y \leftarrow 0$

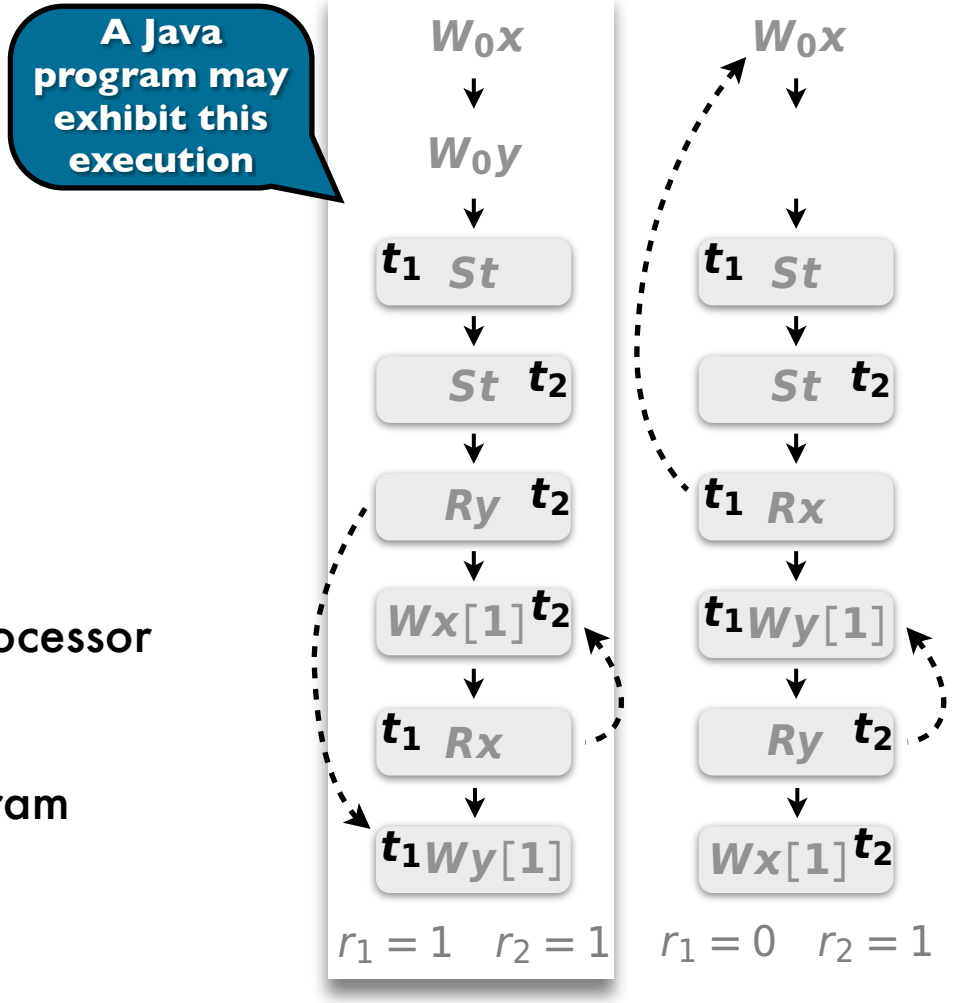
$r_1 \leftarrow x$	$r_2 \leftarrow y$
$y \leftarrow 1$	$x \leftarrow 1$



Sequential Consistency and Java

SC memory model

- Do we have a formal definition? yes
- Do people understand it? yes
- Is it faithful to Java semantics and processor behaviour? **no**
  - compilers will often reorder program statements
  - and hardware will do the same



Sequential Consistency and Java

Let's try...

$x \leftarrow 0; y \leftarrow 0$	
$y \leftarrow 1$	$x \leftarrow 1$
$r_1 \leftarrow x$	$r_2 \leftarrow y$
$r_1 = 0$	$r_2 = 0$

is not visible in SC

```
public class Relaxed1 extends Thread {
        public static volatile long iter = 0;
        public static volatile int fence = 0;
        public static int res0 = 1;
        public static int res1 = 1;
        private static int x = 0;
        private static int y = 0;
        private int tid;
        private long wait;

        public Relaxed1(int tid, long wait){
                this.tid = tid;
                this.wait = wait;
        }

        public void run(){
                if (this.tid == 0) {
                        while(iter == this.wait);
                        x = 1;
```

```
Administrator — bash — 80×24
bash-3.2$ javac Relaxed1.java
bash-3.2$ java Relaxed1
Relaxed after 5055 iterations!
bash-3.2$
```

```
                while (res0 != 0 || res1 != 0){
                        Relaxed1 th1 = new Relaxed1(0,iter);
                        Relaxed1 th2 = new Relaxed1(1,iter);
                        th1.start();
                        th2.start();
                        iter++;
                        try {
                                th1.join();
                                th2.join();
                        } catch (InterruptedException e) {}
                        x = 0;
                        y = 0;
                }
                if (res0 == 0 && res1 == 0)
                    System.out.println("Relaxed after "+iter+" iterations!");
        }
}
```

The Java Memory Model

- Proposed in 1995, but broken.
- New version in 2004 (JSR-133). Published POPL'05 (Manson, Pugh, Adve)
  - Also broken :-)

- JMM tries *squaring the circle*
  - ★Guarantees for programmers
    - ✦Data-race free programs execute like in a SC model
    - ✦A (safe) formal semantics for all Java programs (incl. those with races)
  - ★Guarantees for optimizers
    - ✦Allows all various hardware reordering semantics
    - ✦Allows aggressive compiler optimizations

Happens-Before Model

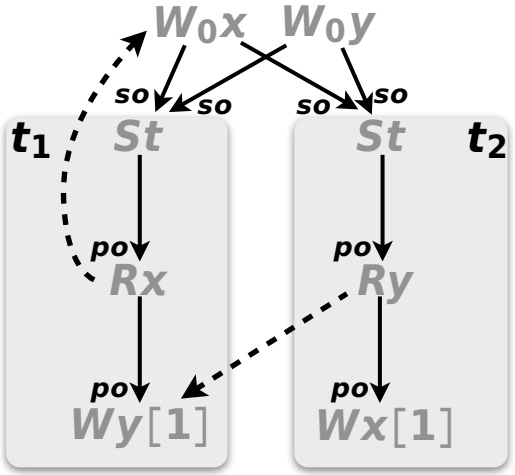
An interleaving of actions is better described without global time.

It is in fact a consistent extension of a partial order called **happens before**.

The **hb** relation is the smallest transitive relation that respects

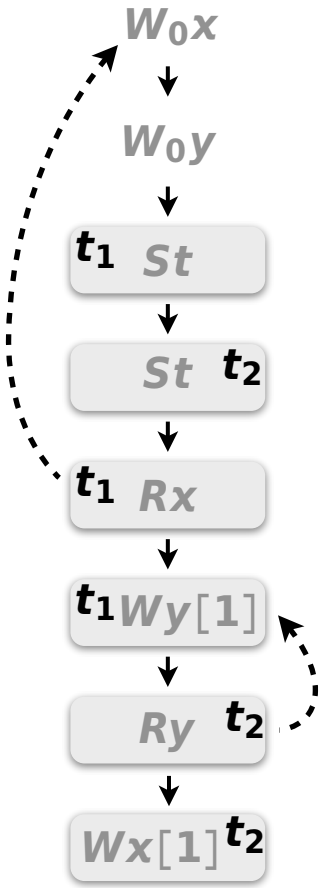
- program order between actions of same thread
- synchronization order between actions of threads

This relation should constrain the write actions that a read action can legally see.



$$x \leftarrow 0; y \leftarrow 0$$

$r_1 \leftarrow x$	$r_2 \leftarrow y$
$y \leftarrow 1$	$x \leftarrow 1$



Happens-Before Model

An axiomatic execution is described by a tuple

$$\left\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W \right\rangle$$

Among all executions, the happens-before model selects the so-called **well-formed executions**

Ex: a read action must not see a write that happens after it.

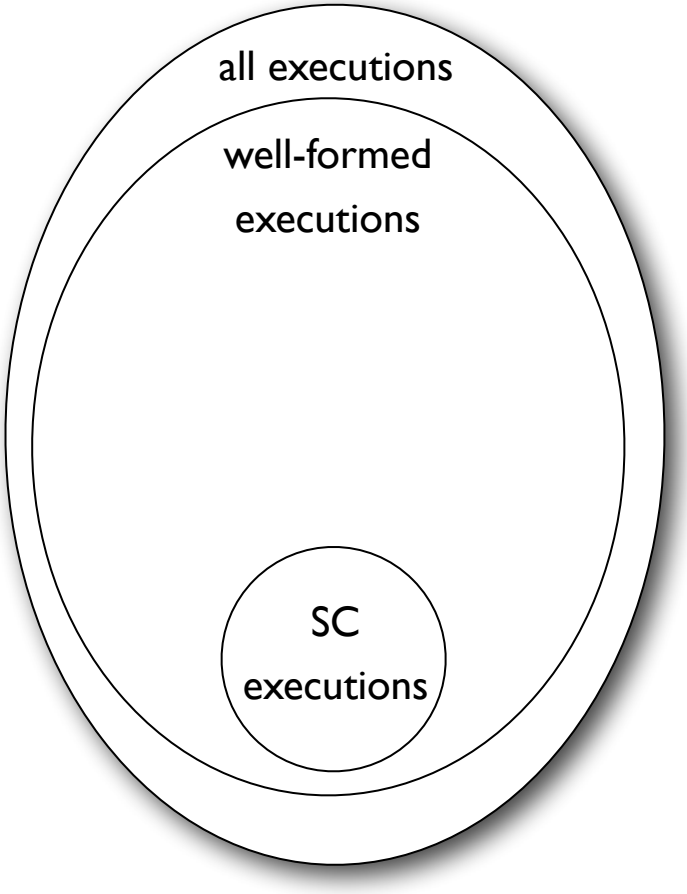
Such a model

is quite easy to understand,

contains all SC executions,

but is too relaxed: it does not respect the DRF property

allows cyclic dependencies

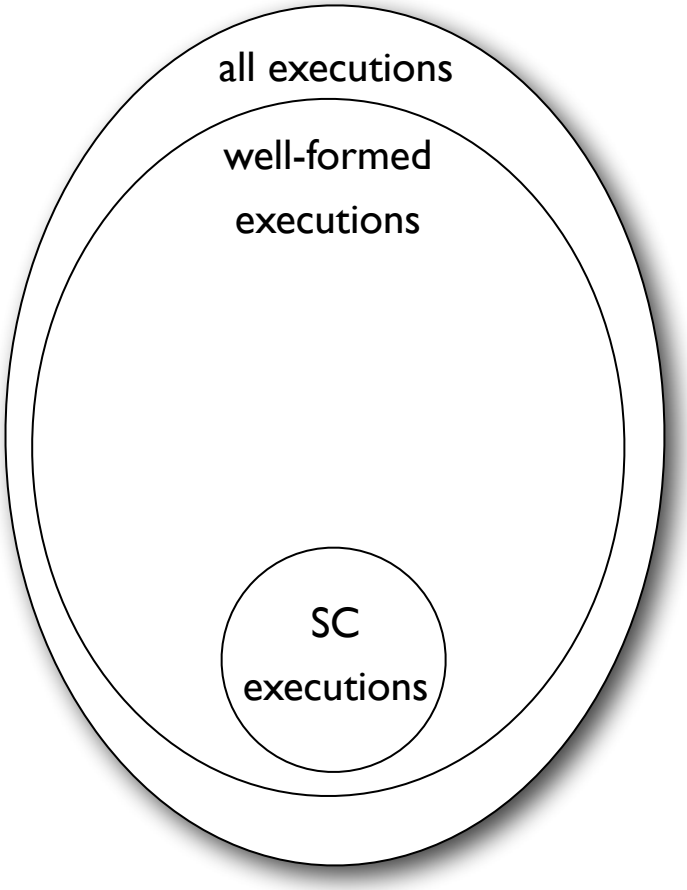


Data Race Freedom

Data race free program:

- Intuitively: a program where two threads don't access at the same time a non-volatile shared location (at least one of the access is a write)
- Formally: in all executions of the model, conflicting actions must be in the **hb** relation

Data race free model: data-race free programs only have SC executions.



Race Committing Sequence

$x \leftarrow 0; y \leftarrow 0$

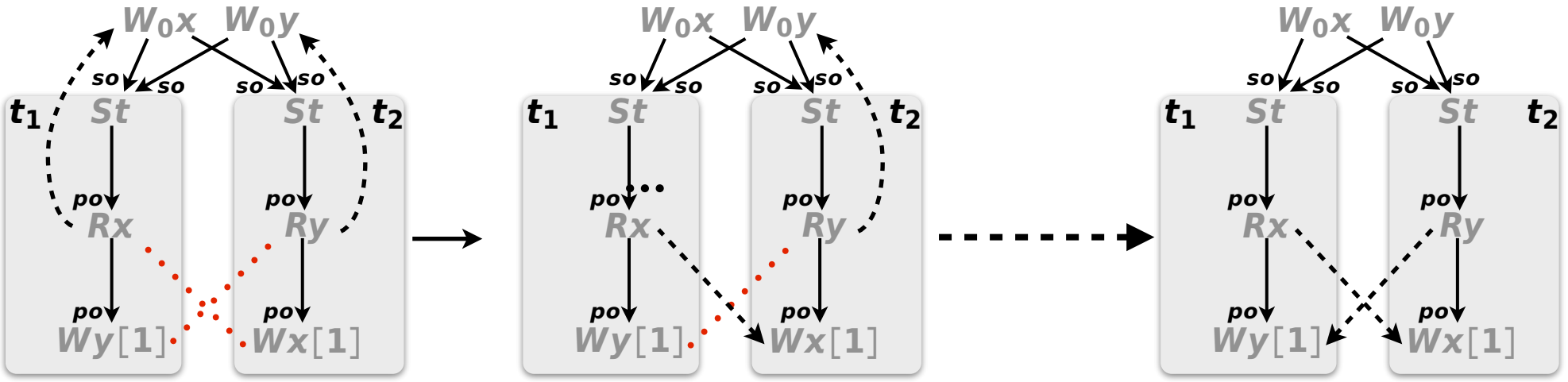
$r_1 \leftarrow x$	$r_2 \leftarrow y$
$y \leftarrow 1$	$x \leftarrow 1$

Is $r_1 = 1 \quad r_2 = 1$ allowed ?

each read sees a write that happens before it

1. Start from a well-behaved execution
2. Commit races: modifies the write-seen
3. Restart the execution taking the race into

+ constraints on the sequence to rule out cyclic causality



a well-behaved execution — one write-seen is updated — final JMM execution

Examples

initially x = y = 0

r1 = x	r2 = y
y = 1	x = 1

A. (allowed)

initially x = y = 0

lock m1	lock m2
r1=x	r2=y
unlock m1	unlock m2
lock m2	lock m1
y=1	x=1
unlock m2	unlock m1

B. (prohibited)

initially x = y = 0

r1 = x	r2 = y
y = r1	x = r2

C. (prohibited)

Is it possible to get r1 = r2 = 1 at the end of an execution?

Compiler Transformations

The meaning of a Java program is given by its set of traces:

- A transformation that does not change this set is trivially valid

```
if (r1==1)                    x=1
     {x=1;y=1}      ⇄         y=1
else {x=1;y=1}
```

Independent writes can be reordered

```
x=1          y=1
y=1    ⟶    x=1
```

Elimination of redundant reads

```
r1 = x          r1 = x          x = r1          x = r1
r2 = x    ⟶    r2 = r1    ⟶    r2 = x    ⟶    r2 = r1
if (r1==r2)     if (r1==r2)     if (r1==r2)     if (r1==r2)
  y = 1           y = 1           y = 1           y = 1
```

(read after read) (read after write)

Is this valid?

Compiler Transformations

Irrelevant Read Introduction (Speculation)

```
if (r1==1) {          if (r1==1) {
  r2=x                  r2=x
  y=r2          →       y=r2          ⇄   r2=x
}                     } else r2=x         if (r1==1)
                                            y=r2
```

Is this valid?

Roach Motel Semantics

```
x=1                       lock m
lock m                      x=1
  y=1            ⟶         y=1
unlock m                  unlock m
```

Interestingly, this transformation is invalid in general

Compiler Transformations

Redundant Write-After-Read Elimination

initially x = 0

lock m1	lock m2	lock m1
x=2	x=1	lock m2
unlock m1	unlock m2	r1=x
		x=r1
		r2=x
		unlock m2
		unlock m1

Is this valid?

- No well-behaved execution contains a datarace
- The read "r2 = x" must always see the write "x=r1"
- Removing the redundant write allows r1 and r2 to see different values

Compiler Transformations

x = y = 0	
r1=x	r2=y
y=r1	if (r2==1) {
	r3=y
	x=r3
	} else x=1

Suppose we rewrite to r3=r2.
Can we now observe r1=r2=1?

- Initially only one well-behaved execution, r1=r2=0
- Two dataraces:
  - between y=r1 and r2=y:
    - r2=0 is guaranteed
  - between r1=x and x=1 with value 1
    - now, commit datarace between y=r1 and r2=y with value 1
    - if r1=r2=1, then r3=y must read a value that happens-before it
      - there's no datarace between r3=y and y=r1
      - thus, the only value that it can read is y=0
      - then x=r3 must write 0: contradiction
- Transforming r3=y to r3=r2 allows r1=r2=1
  - commit the datarace betwen r1=x and x=1
  - then, commit the datarace between y=r1 and r2=y
  - can keep commitment to write x=1 since r1=r2=1