

# Concurrency in the XXI century

It's WEAK

université  
**PARIS**  
PARIS 7 **DIDEROT**



Gustavo Petri  
<https://gpetri.github.io>

# Introduction to Weak Consistency

# Concurrency: Critical Sections



# Concurrency: Critical Sections



Many Solutions:

- Locks
- Semaphores
- Monitors
- Message Passing
- Actors ...

# Concurrency: Critical Sections



Many Solutions:

- Locks
- Semaphores
- Monitors
- Message Passing
- Actors ...

Many Implementations:

- Dekker
- Patterson
- Lamport (Bakery)
- Dijkstra (P/V) ...

# Concurrency: Critical Sections



Many Solutions:

- Locks
- Semaphores
- Monitors
- Message Passing
- Actors ...

Many Implementations:

- Dekker
- Patterson
- Lamport (Bakery)
- Dijkstra (P/V) ...

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}

t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}
t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}
t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

What went wrong?

```
public class Dekker {
    public static boolean t1 = false;
    public static boolean t2 = false;
    public static int nonNeg = 1;

    public static void main(String[] args) {

        for (;;) {
            Dekker.t1 = false;
            Dekker.t2 = false;
            Dekker.nonNeg = 1;

            Thread t1 = new Thread() {
                public void run() {
```

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}
t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

What went wrong?

- t1 and t2 aren't sync

```
public class Dekker {
    public static boolean t1 = false;
    public static boolean t2 = false;
    public static int nonNeg = 1;

    public static void main(String[] args) {

        for (;;) {
            Dekker.t1 = false;
            Dekker.t2 = false;
            Dekker.nonNeg = 1;

            Thread t1 = new Thread() {
                public void run() {
```

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}
t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

What went wrong?

- t1 and t2 aren't sync
- nonNeg is not sync

```
public class Dekker {
    public static boolean t1 = false;
    public static boolean t2 = false;
    public static int nonNeg = 1;

    public static void main(String[] args) {

        for (;;) {
            Dekker.t1 = false;
            Dekker.t2 = false;
            Dekker.nonNeg = 1;

            Thread t1 = new Thread() {
                public void run() {
```

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}
t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

What went wrong?

- t1 and t2 aren't sync
- nonNeg is not sync
- can we fix it?

```
public class Dekker {
    public static boolean t1 = false;
    public static boolean t2 = false;
    public static int nonNeg = 1;

    public static void main(String[] args) {

        for (;;) {
            Dekker.t1 = false;
            Dekker.t2 = false;
            Dekker.nonNeg = 1;

            Thread t1 = new Thread() {
                public void run() {
```

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}

t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

What went wrong?

- t1 and t2 aren't sync
- nonNeg is not sync
- can we fix it?
- nonNeg--?

```
public class Dekker {
    public static boolean t1 = false;
    public static boolean t2 = false;
    public static int nonNeg = 1;

    public static void main(String[] args) {

        for (;;) {
            Dekker.t1 = false;
            Dekker.t2 = false;
            Dekker.nonNeg = 1;

            Thread t1 = new Thread() {
                public void run() {
```

# Dekker's Mutual Exclusion

```
t1 = false; t2 = false;
t1 = true;
if (!t2) {
    // I'm alone in the
    // critical section
}
t2 = true;
if (!t1) {
    // I'm alone in the
    // critical section
}
```

**Dekker.java**

What went wrong?

- t1 and t2 aren't sync
- nonNeg is not sync
- can we fix it?
- nonNeg--?
- How do we know we fixed it?

```
public class Dekker {
    public static boolean t1 = false;
    public static boolean t2 = false;
    public static int nonNeg = 1;

    public static void main(String[] args) {

        for (;;) {
            Dekker.t1 = false;
            Dekker.t2 = false;
            Dekker.nonNeg = 1;

            Thread t1 = new Thread() {
                public void run() {
```

# Where to from here?

# Where to from here?

What is a memory model?

- What are the possible results of a *memory read* operation

# Where to from here?

What is a memory model?

- What are the possible results of a *memory read* operation

## Understanding Memory Models

- Testing
- Formalization
- Validation

# Where to from here?

What is a memory model?

- What are the possible results of a *memory read* operation

## Understanding Memory Models

- Testing
- Formalization
- Validation

## Using Memory Models

- Programming
- Optimization
- Verification

# How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

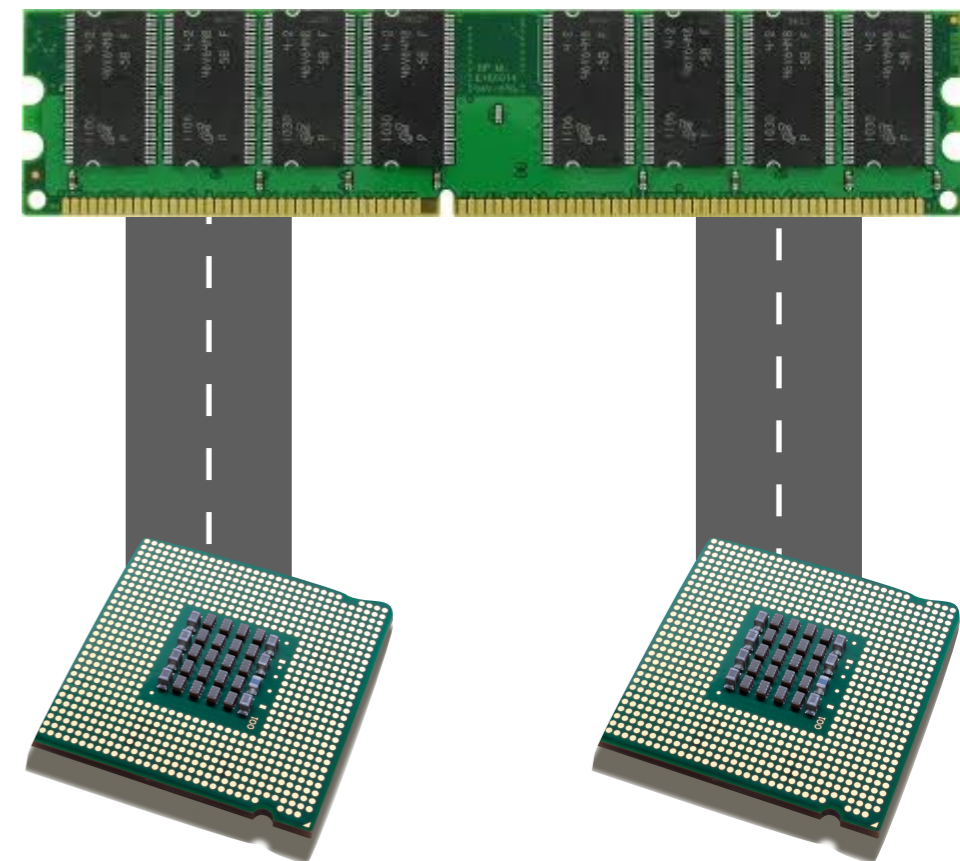
**Abstract**—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

**Index Terms**—Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program. A multiprocessor satisfying this condition is called *sequentially consistent*. The sequentiality

# Sequential Consistency (SC)

- R1: Each processor issues *memory requests in the order specified by its program*
- R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Entering a memory request consists of entering the request on this queue



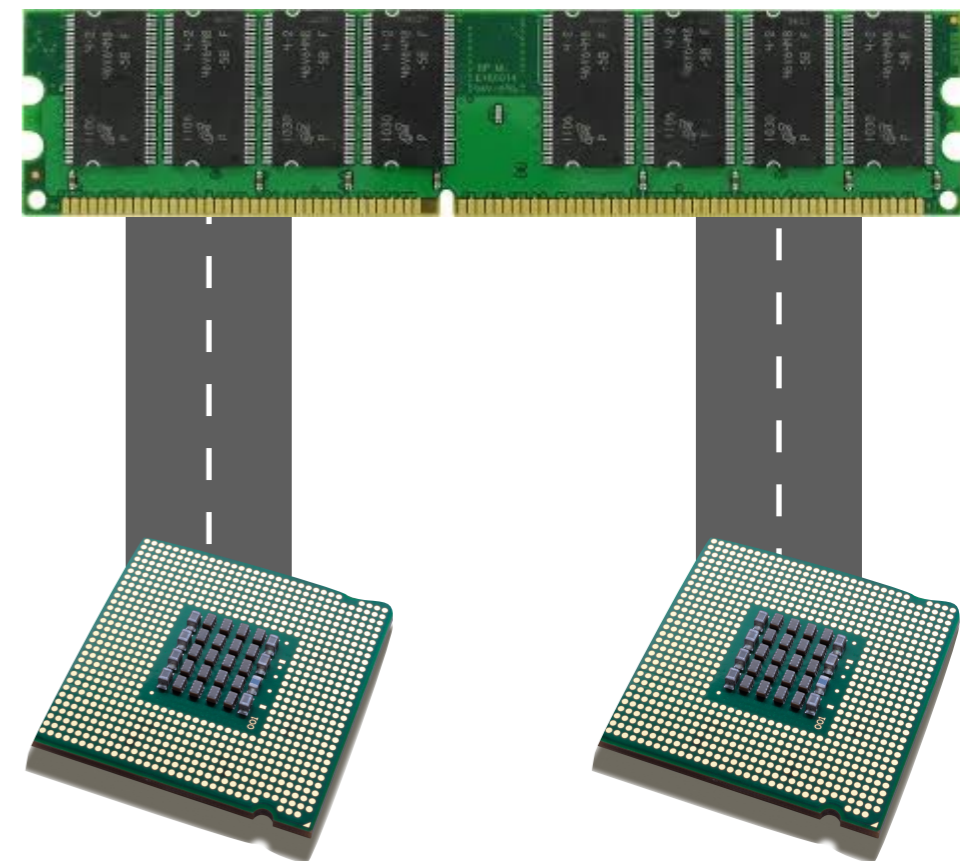
# Sequential Consistency (SC)

- R1: Each processor issues *memory requests in the order specified by its program*
- R2: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Entering a memory request consists of entering the request on this queue

Dekker is safe

```
t1 = false; t2 = false;  
t1 = true;  
if (!t2) {  
    // I'm alone in the  
    // critical section  
}
```

```
t2 = true;  
if (!t1) {  
    // I'm alone in the  
    // critical section  
}
```



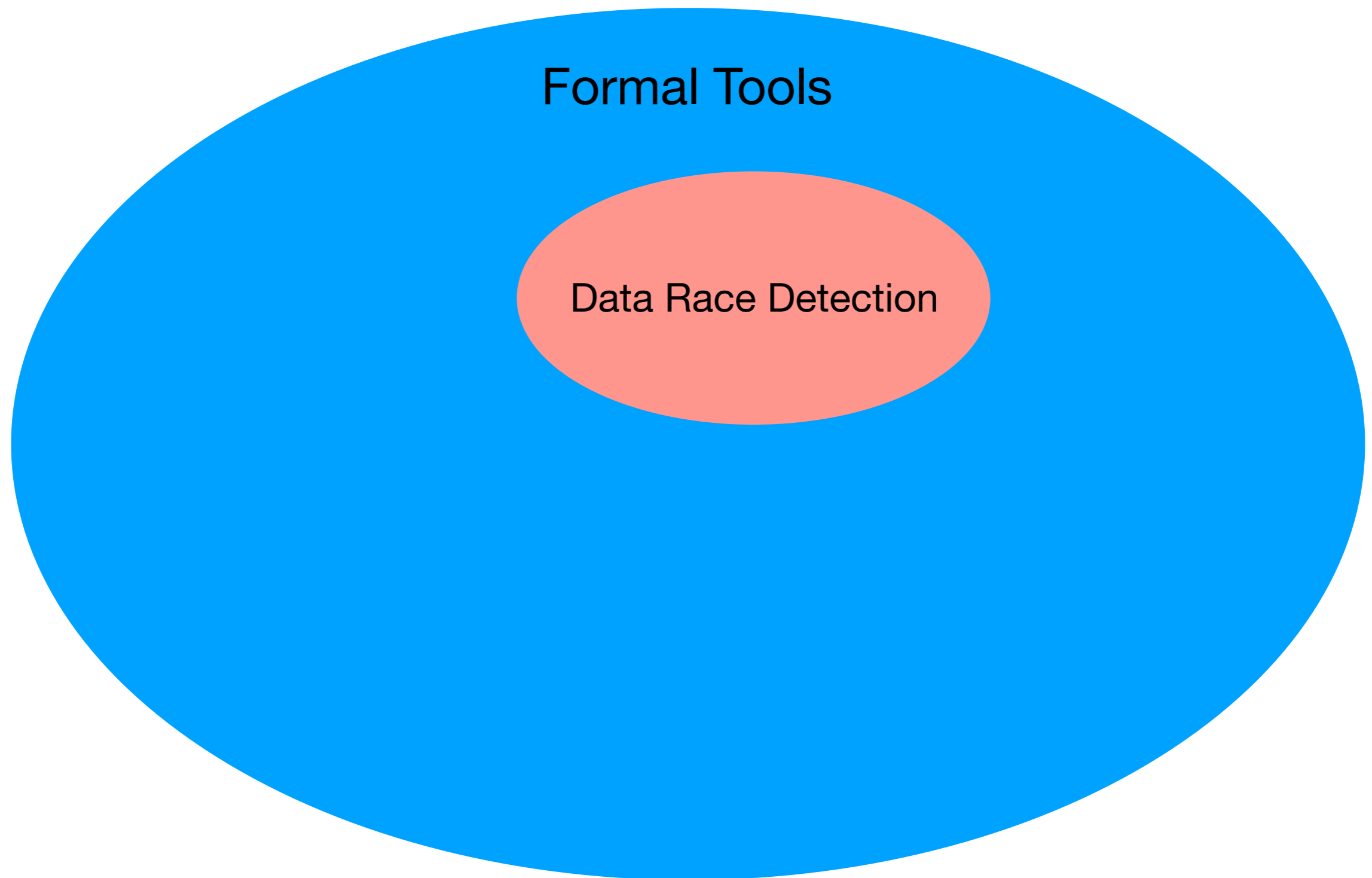
# Sequential Consistency

# Sequential Consistency

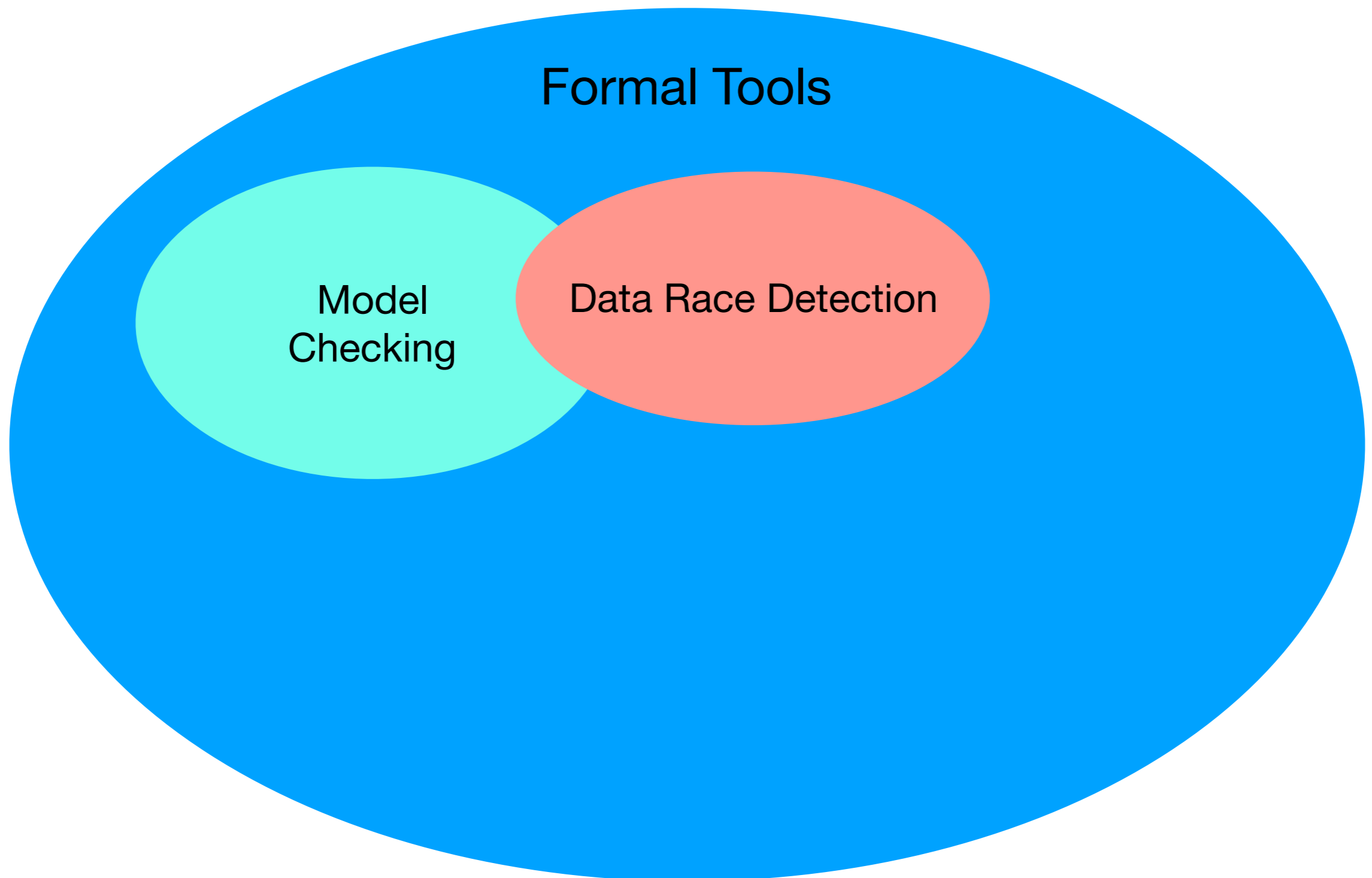


Formal Tools

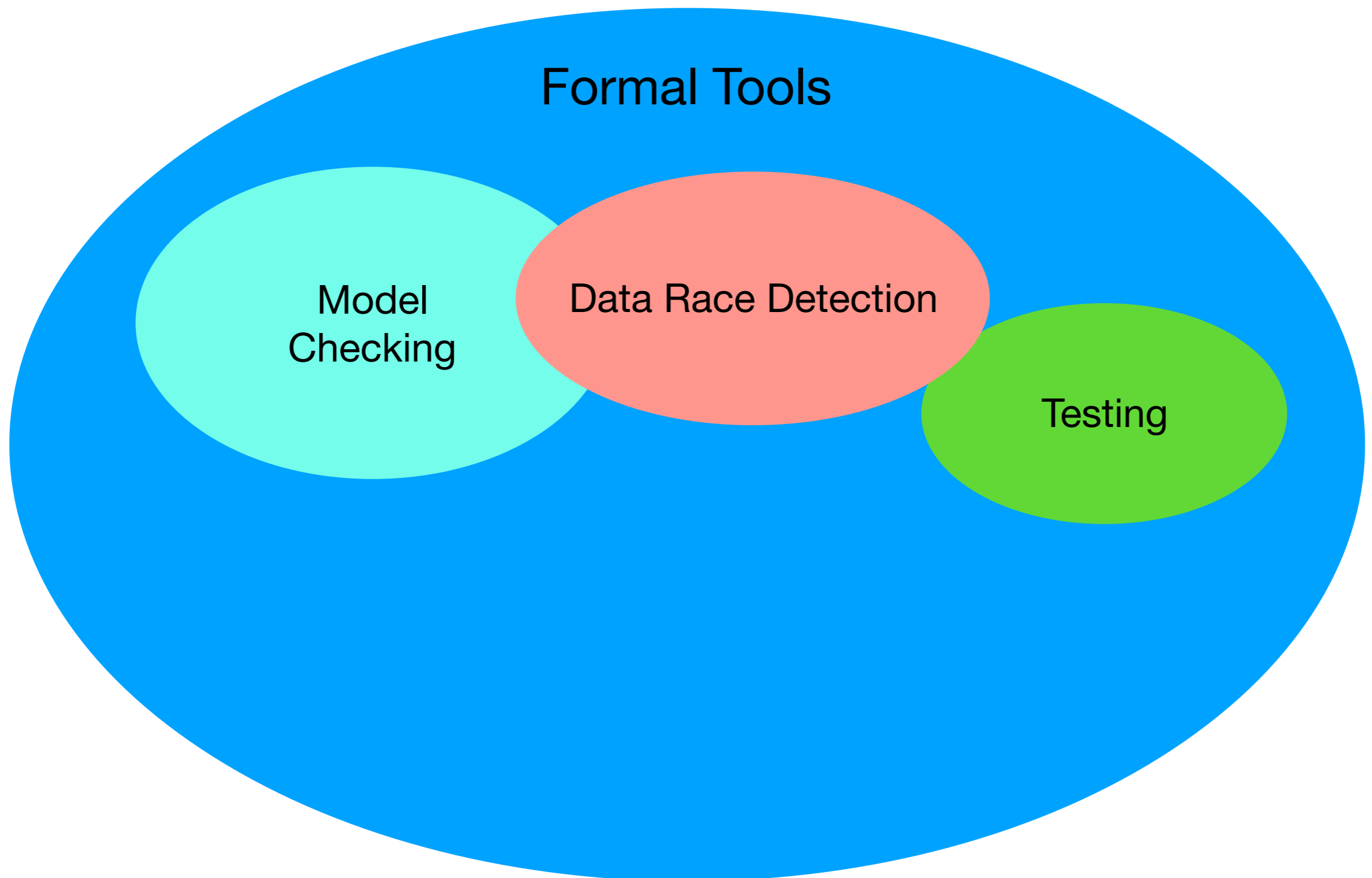
# Sequential Consistency



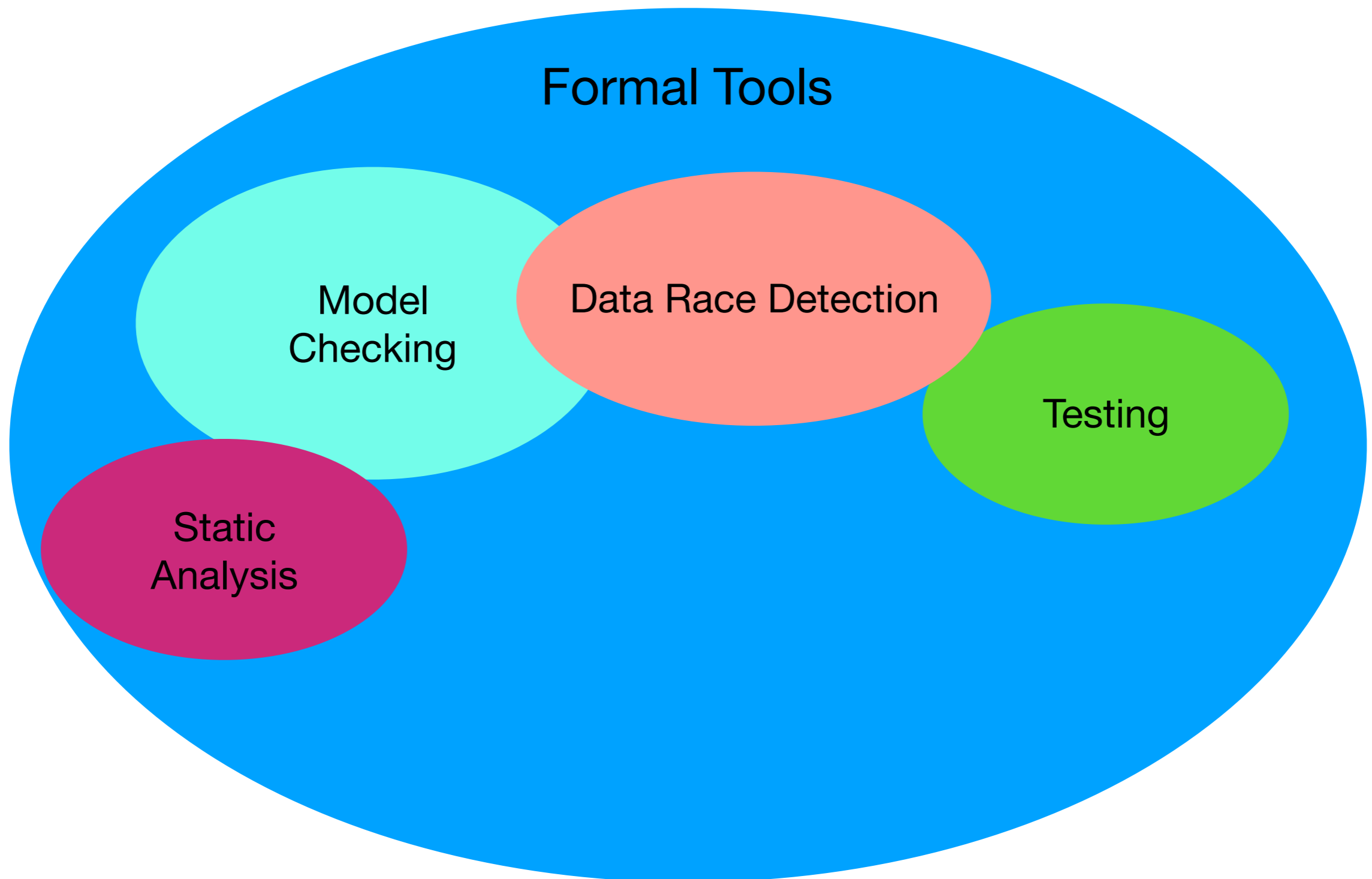
# Sequential Consistency



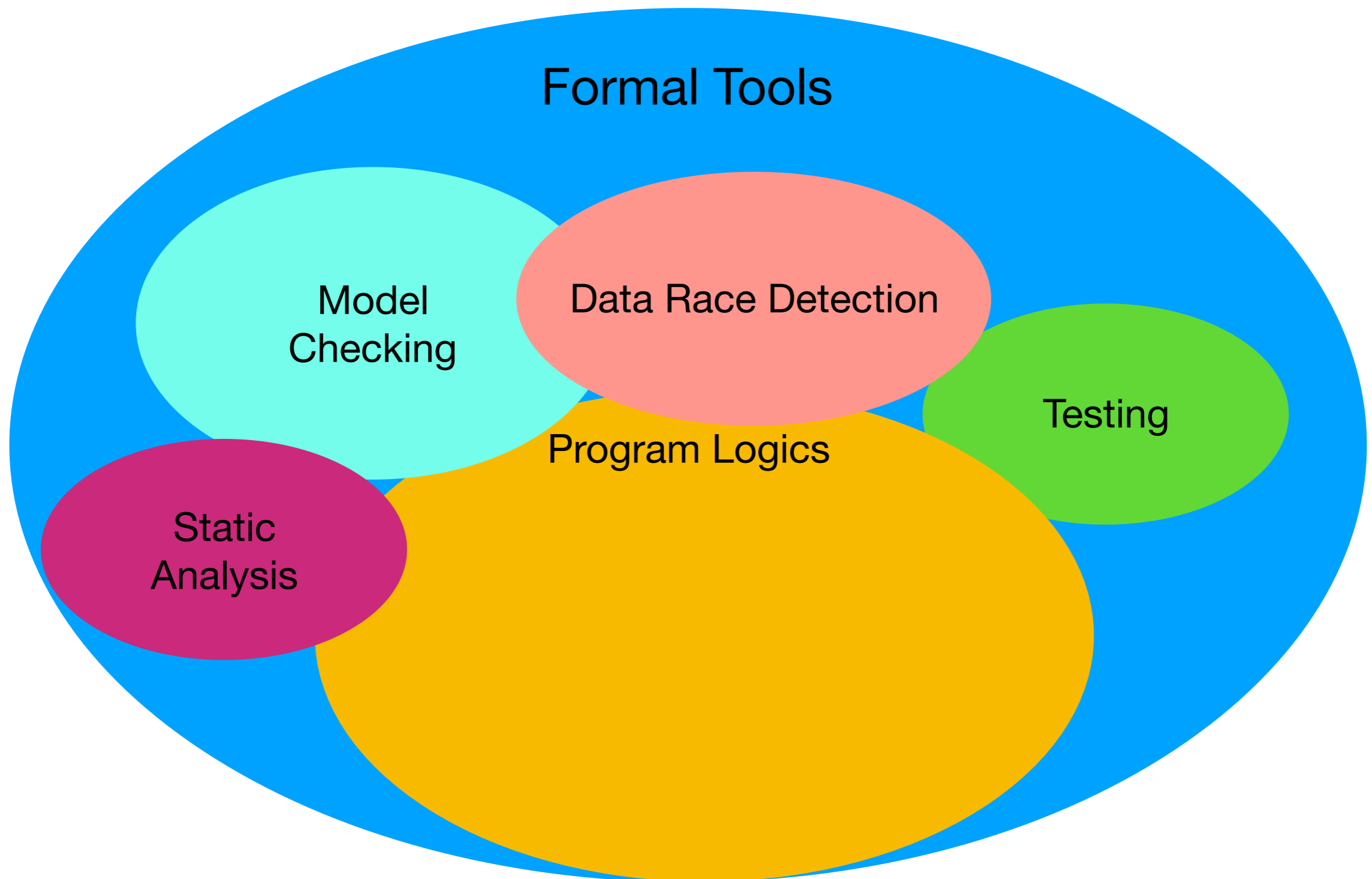
# Sequential Consistency



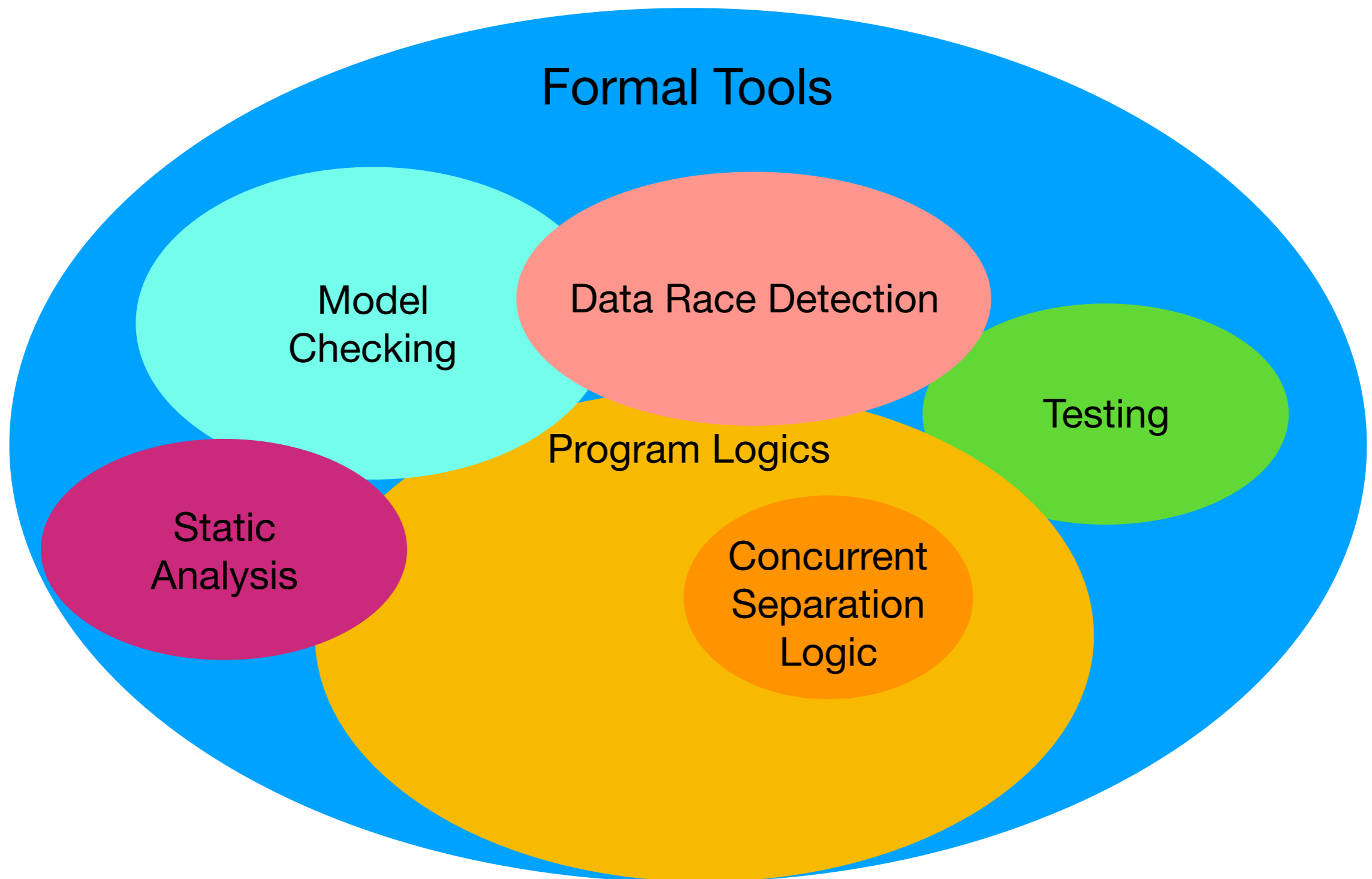
# Sequential Consistency



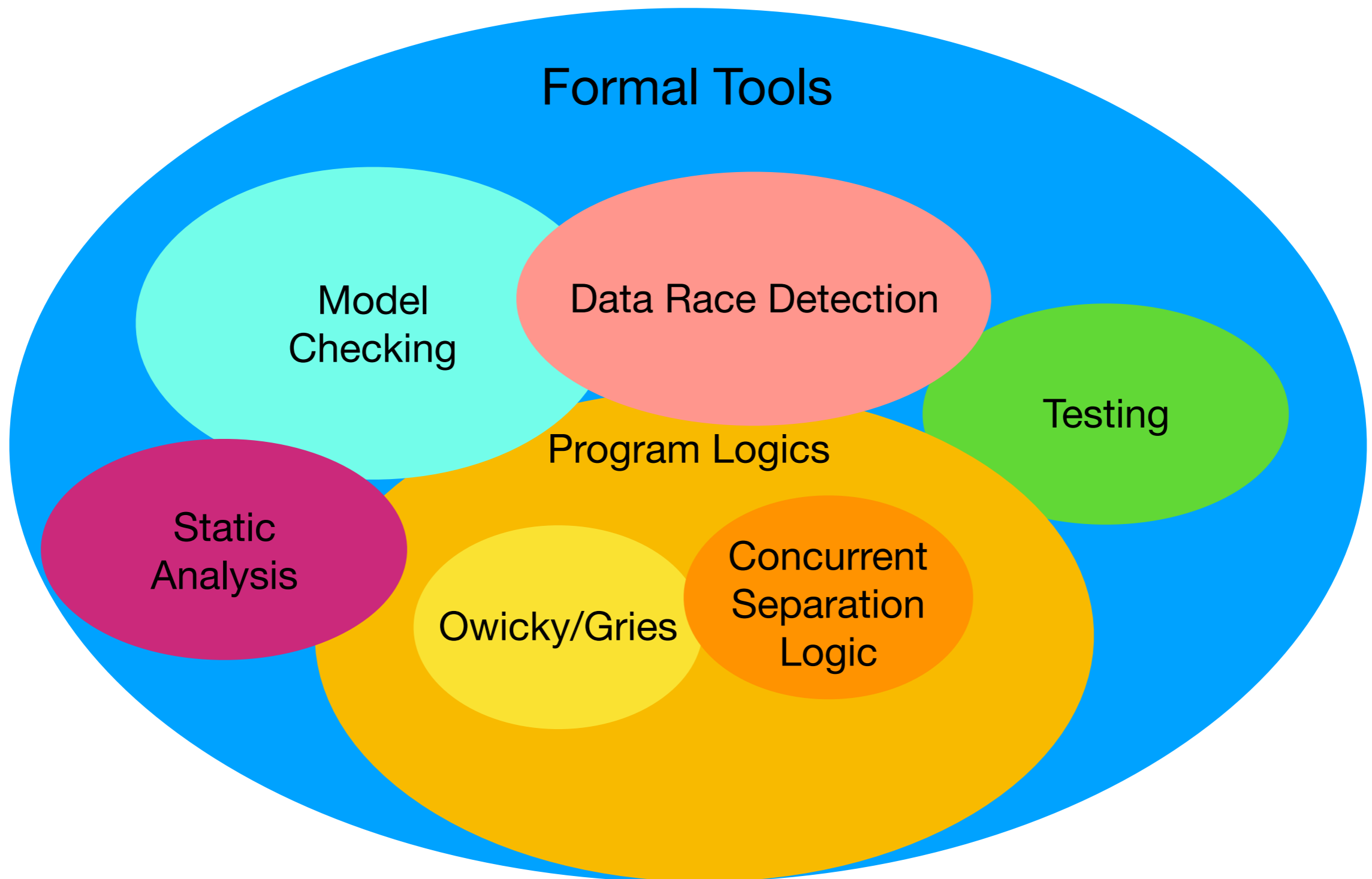
# Sequential Consistency



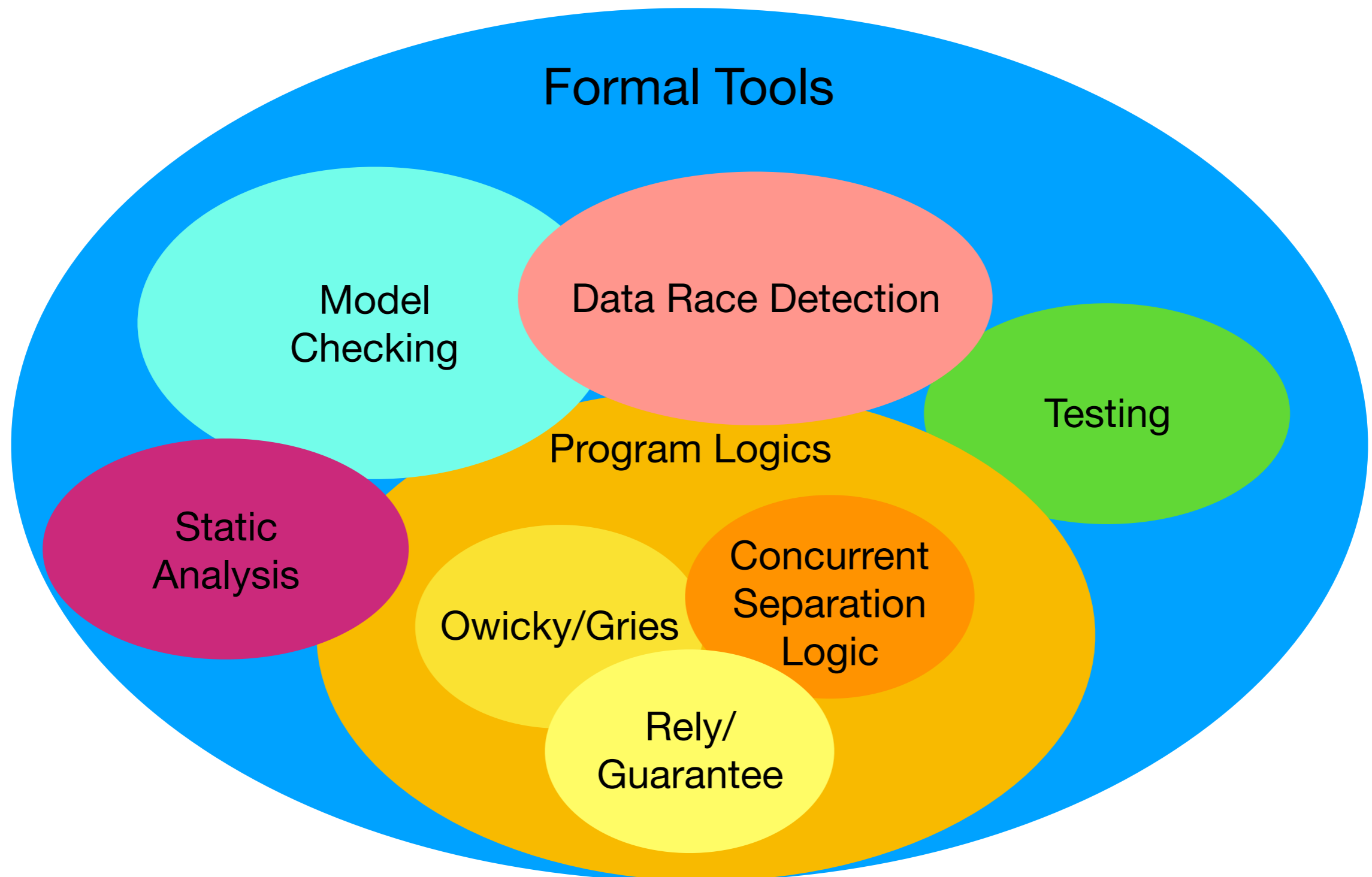
# Sequential Consistency



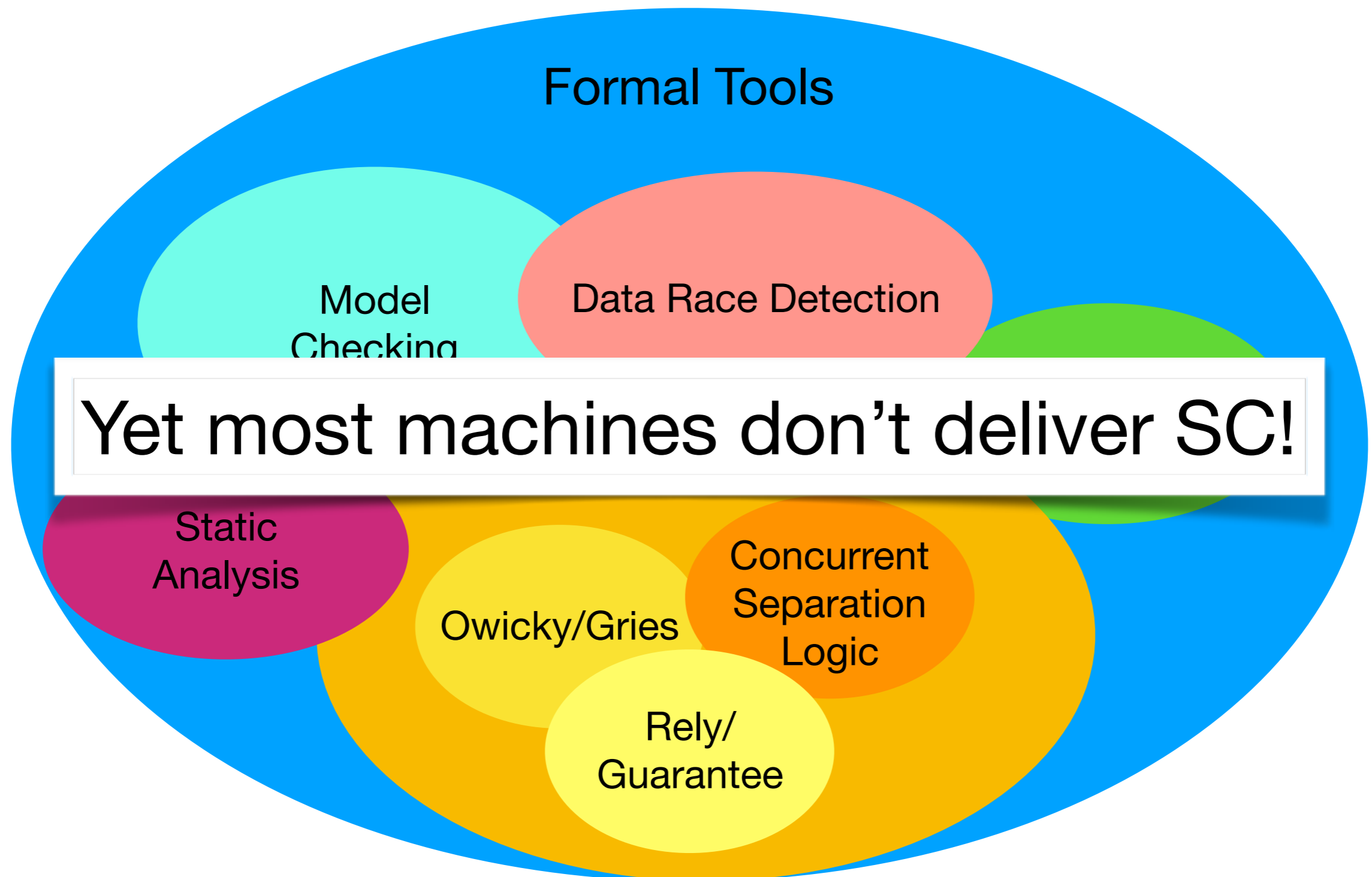
# Sequential Consistency



# Sequential Consistency



# Sequential Consistency



# How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

**Abstract**—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

**Index Terms**—Computer design, concurrent computing, hardware correctness, multiprocessing, parallel processing.

A high-speed processor may execute operations in a different order than is specified by the program. The correctness of the execution is guaranteed if the processor satisfies the following condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called *sequential*. Consider a computer composed of several such processors accessing a common memory. The customary approach to designing and proving the correctness of multiprocess algorithms [1]–[3] for such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program. A multiprocessor satisfying this condition is called *sequentially consistent*. The sequentiality

# How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs

LESLIE LAMPORT

**Abstract**—Many large sequential computers execute operations in a different order than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor does not guarantee the correct execution of the entire program. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

**Index Terms**—Computer design, concurrent computing, correctness, multiprocessing, parallel processing.

A high-speed processor may execute operations in a different order than is specified by the program. The correct execution is guaranteed if the processor satisfies the condition: the result of an execution is the same as if the operations had been executed in the order specified by the program. A processor satisfying this condition will be called sequentially consistent. Consider a computer composed of several such processors sharing a common memory. The customary approach to proving the correctness of multiprocess algorithms is to show that such a computer assumes that the following condition is satisfied: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the order of each individual processor appears in this sequence in the order in which they executed. A multiprocessor satisfying this condition is called sequentially consistent. The sequentiality

# Shared Memory Consistency Models: A Tutorial \*

Sarita V. Adve<sup>†</sup> and Kourosh Gharachorloo<sup>‡</sup>

<sup>†</sup>Department of Electrical and Computer Engineering  
Rice University  
Houston, Texas 77251-1892

<sup>‡</sup>Western Research Laboratory  
Digital Equipment Corporation  
Palo Alto, California 94301-1616

Rice University ECE Technical Report 9512  
Western Research Laboratory Research Report 95/7  
September 1995

## Abstract

Parallel systems that support the shared memory abstraction are becoming widely accepted in many areas of computing. Writing correct and efficient programs for such systems requires a formal specification of memory semantics, called a *memory consistency model*. The most intuitive model—*sequential consistency*—greatly restricts the use of many performance optimizations commonly used by uniprocessor hardware and compiler designers, thereby reducing the benefit of using a multiprocessor. To alleviate this problem, many current multiprocessors support more *relaxed consistency models*. Unfortunately, the models supported by various systems differ from each other in subtle yet important ways. Furthermore, precisely defining the semantics of each model often leads to complex specifications that are difficult to understand for typical users and builders of computer systems.

The purpose of this tutorial paper is to describe issues related to memory consistency models in a way that would be understandable to most computer professionals. We focus on consistency models proposed for hardware-based shared-memory systems. Many of these models are originally specified with an emphasis on the

# Hardware Models

# The case for Relaxed

## Sequential Program Optimizations

- Memory store takes 10 cycles\*
- Memory load takes 1 cycle\*
- Memory stores stall memory loads

```
x = 1;
```

```
r1 = y;
```

```
r2 = z;
```

\* numbers made up for the example

# The case for Relaxed

## Sequential Program Optimizations

- Memory store takes 10 cycles\*
- Memory load takes 1 cycle\*
- Memory stores stall memory loads

```
x = 1;  
r1 = y;  
r2 = z;
```

12 Cycles

\* numbers made up for the example

# The case for Relaxed

## Sequential Program Optimizations

- Memory store takes 10 cycles\*
- Memory load takes 1 cycle\*
- Memory stores stall memory loads

```
x = 1;  
r1 = y;  
r2 = z;
```

12 Cycles

Optimization

```
r1 = y;  
r2 = z;  
x = 1;
```

\* numbers made up for the example

# The case for Relaxed

## Sequential Program Optimizations

- Memory store takes 10 cycles\*
- Memory load takes 1 cycle\*
- Memory stores stall memory loads

```
x = 1;  
r1 = y;  
r2 = z;
```

12 Cycles

Optimization

```
r1 = y;  
r2 = z;  
x = 1;
```

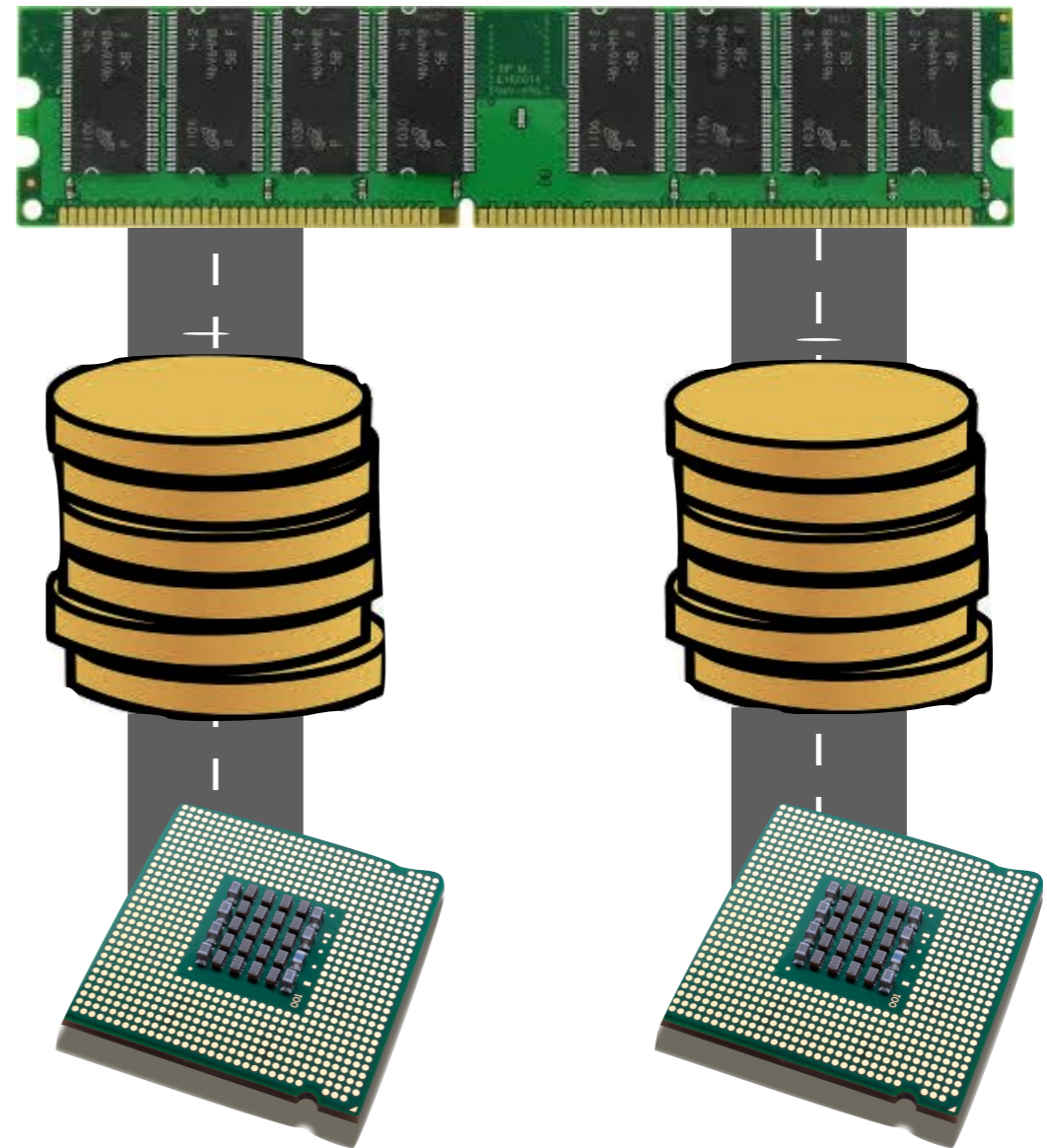
10 Cycles

\* numbers made up for the example

# The Case for Relaxed

## Store Buffers:

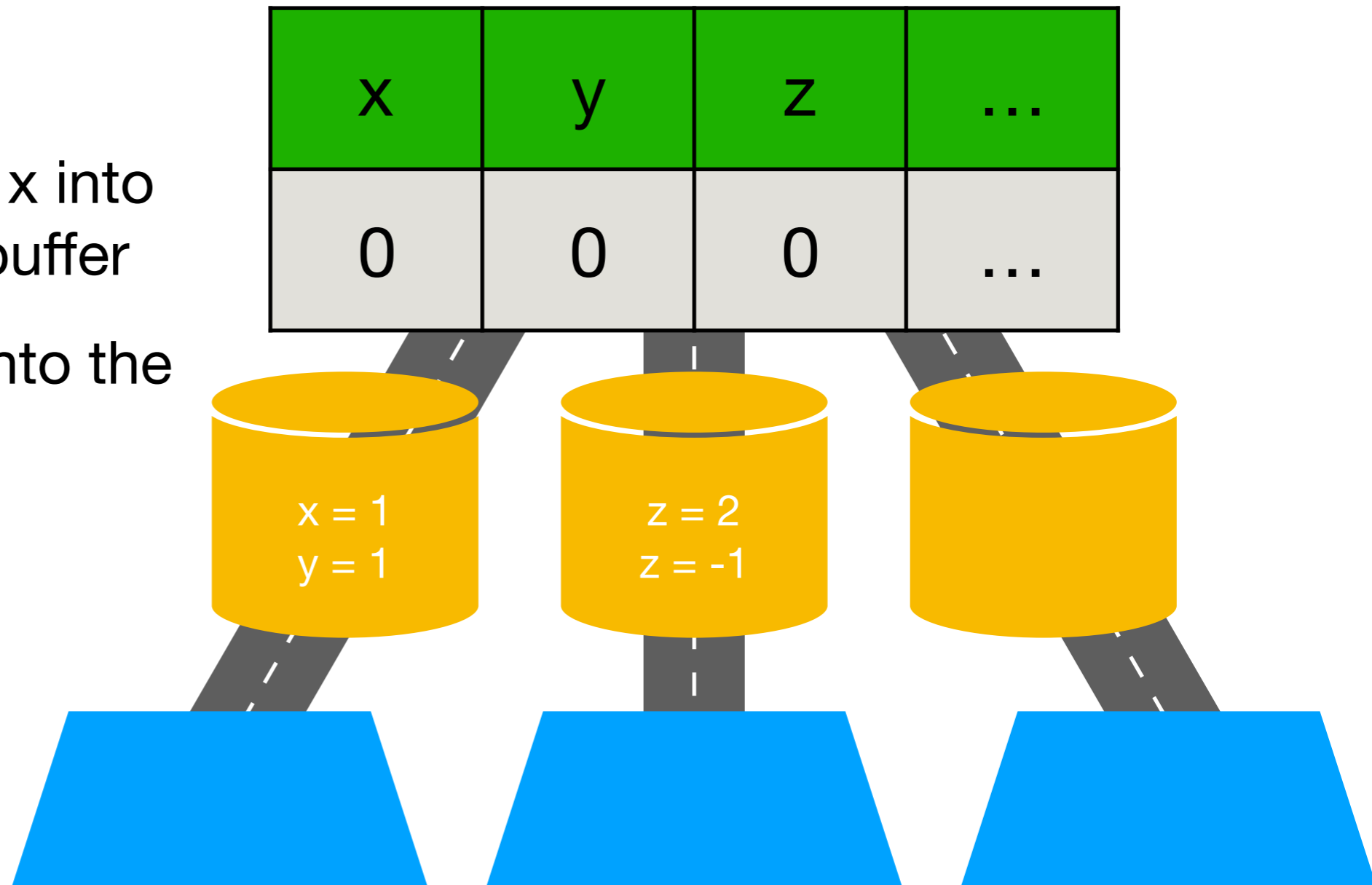
- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory



# The Case for Relaxed

## Store Buffers:

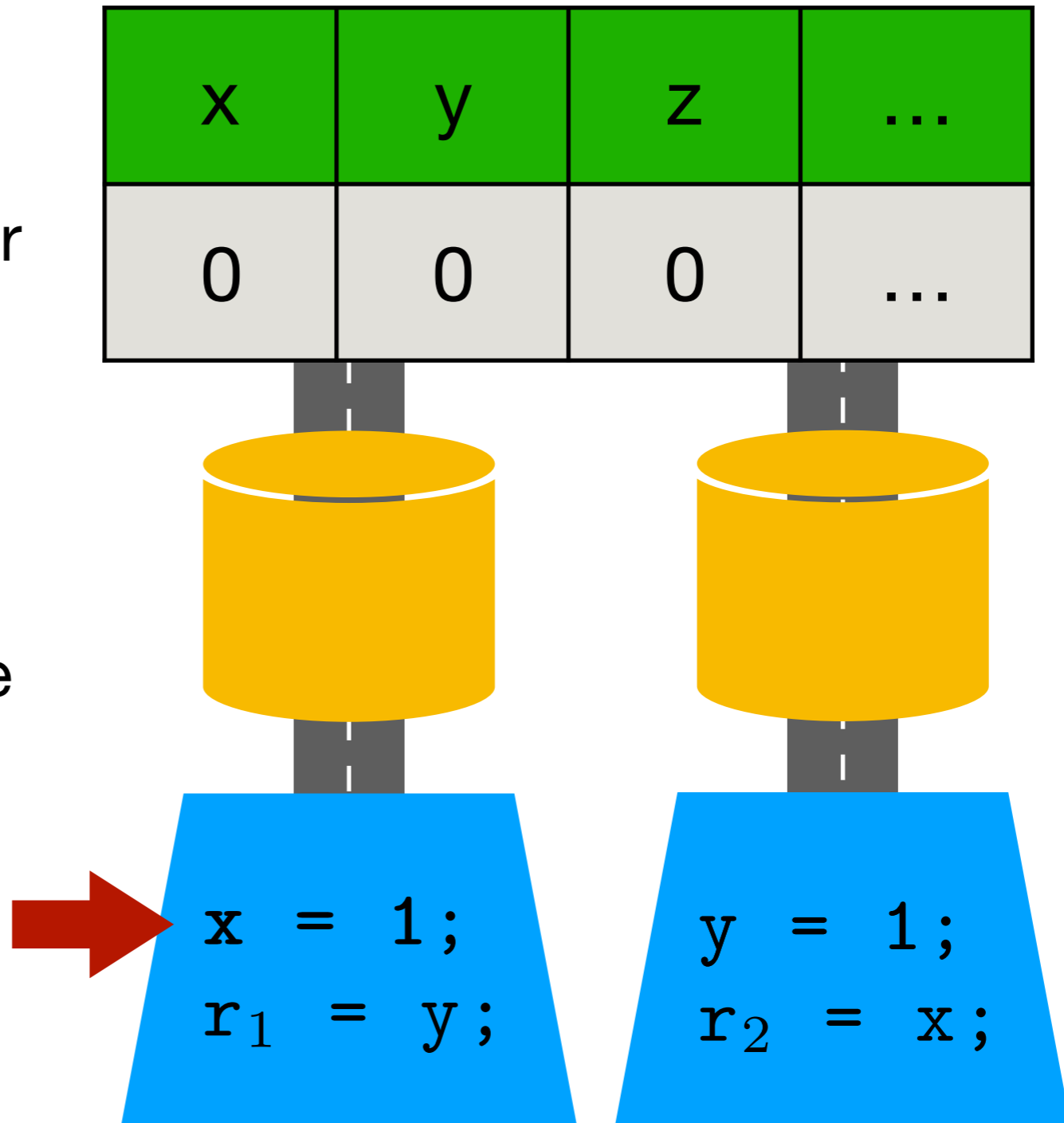
- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory



# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

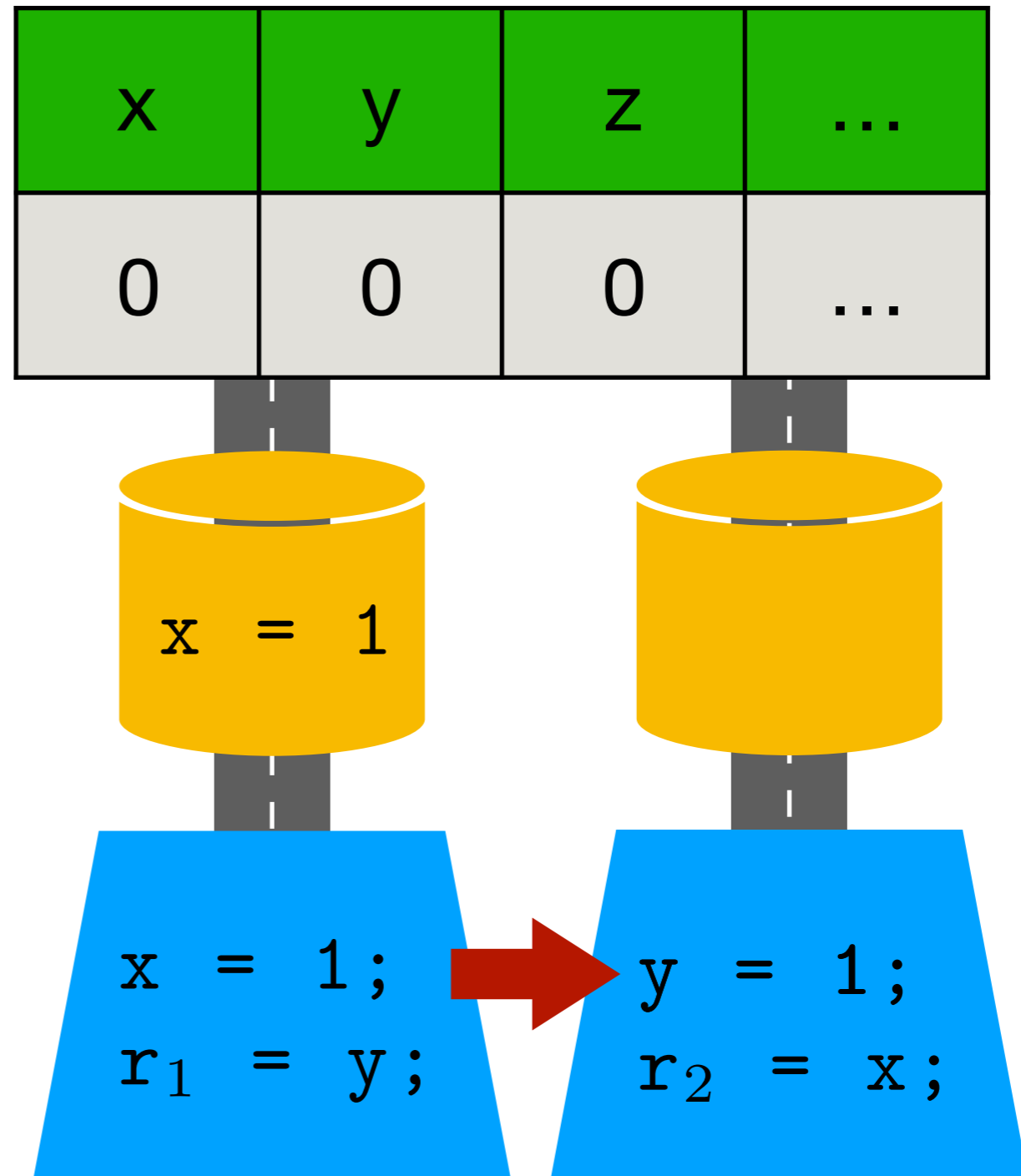
$$\begin{array}{c} x = 0 \ \& \ y = 0 \\ \hline x = 1; \quad \parallel \quad y = 1; \\ r_1 = y; \quad \parallel \quad r_2 = x; \\ \hline r_1 = 0 \ \& \ r_2 = 0 \end{array}$$


# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

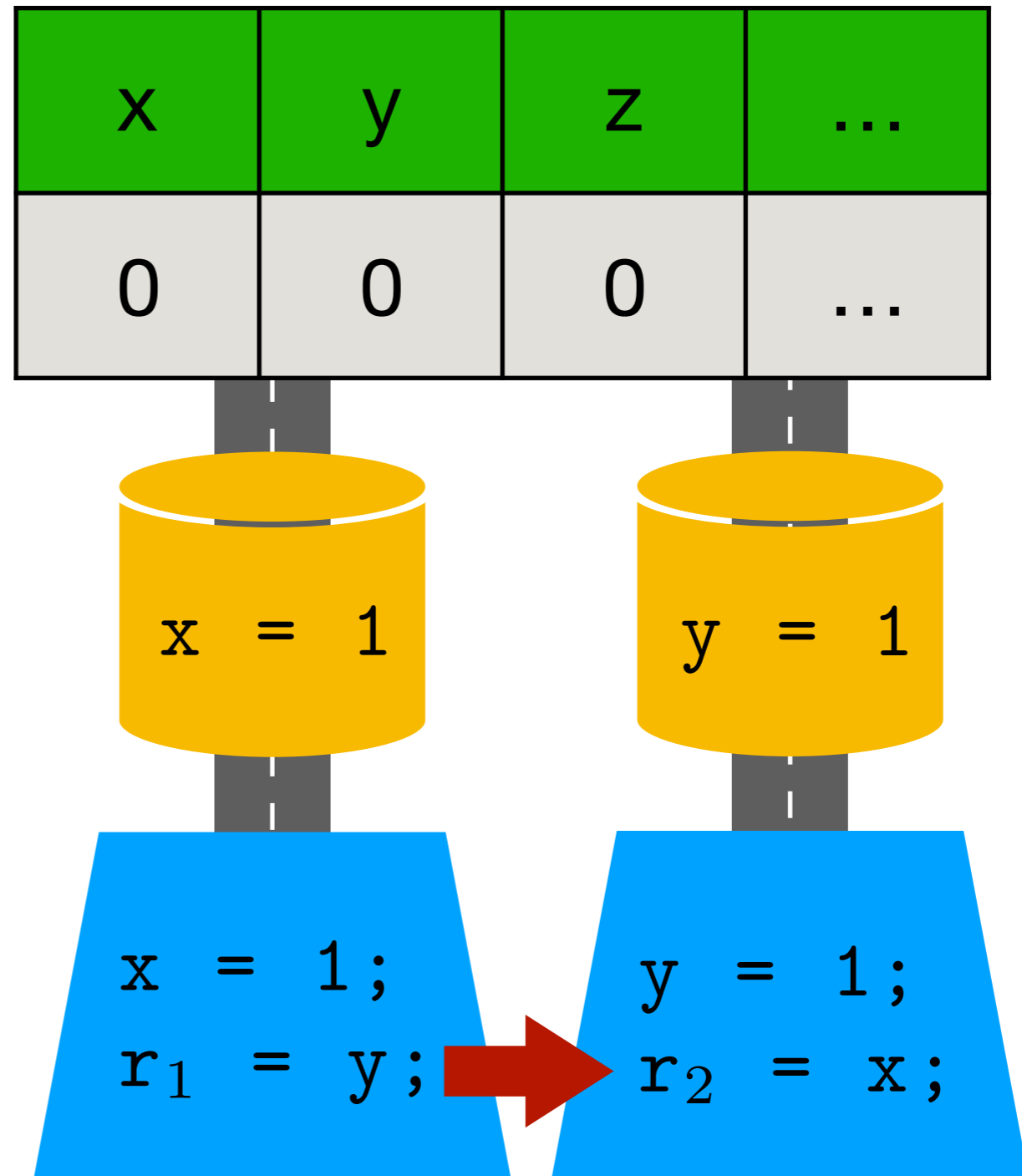
$x = 0 \ \& \ y = 0$	
$x = 1;$	$y = 1;$
$r_1 = y;$	$r_2 = x;$
$r_1 = 0 \ \& \ r_2 = 0$	



# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

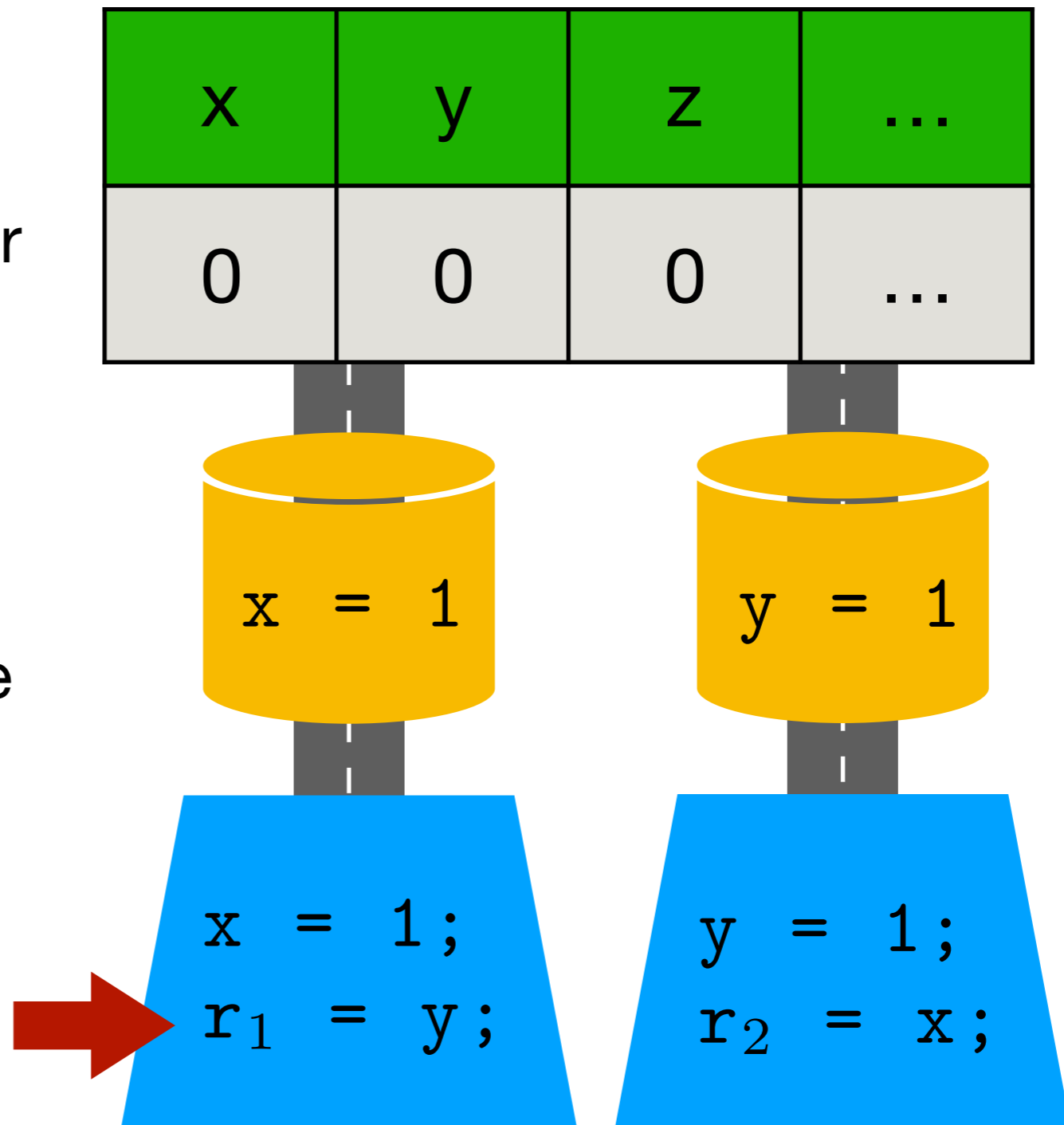
$$\begin{array}{c} x = 0 \ \& \ y = 0 \\ \hline x = 1; \quad \parallel \quad y = 1; \\ r_1 = y; \quad \parallel \quad r_2 = x; \\ \hline r_1 = 0 \ \& \ r_2 = 0 \end{array}$$


# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

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

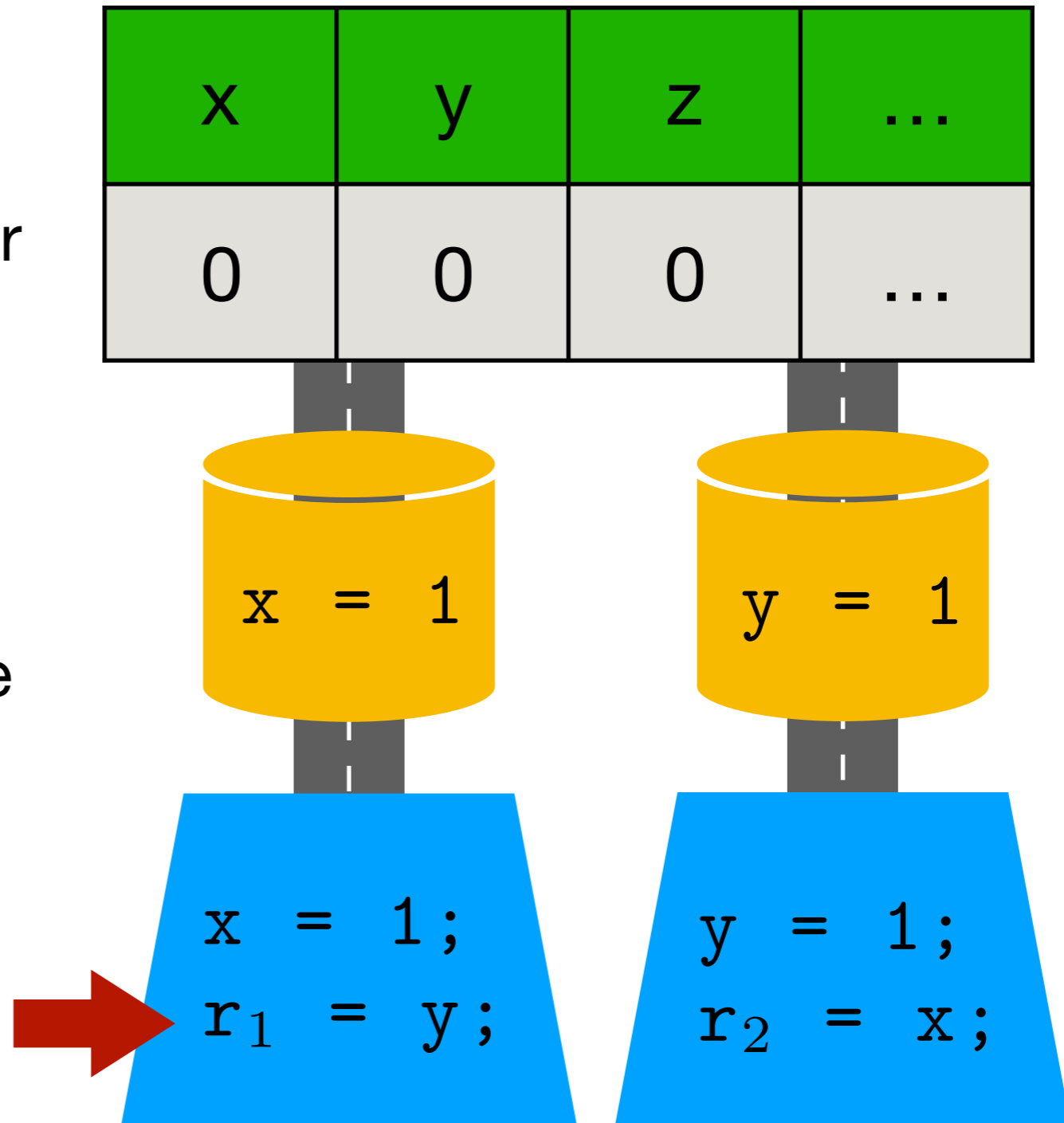


# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

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

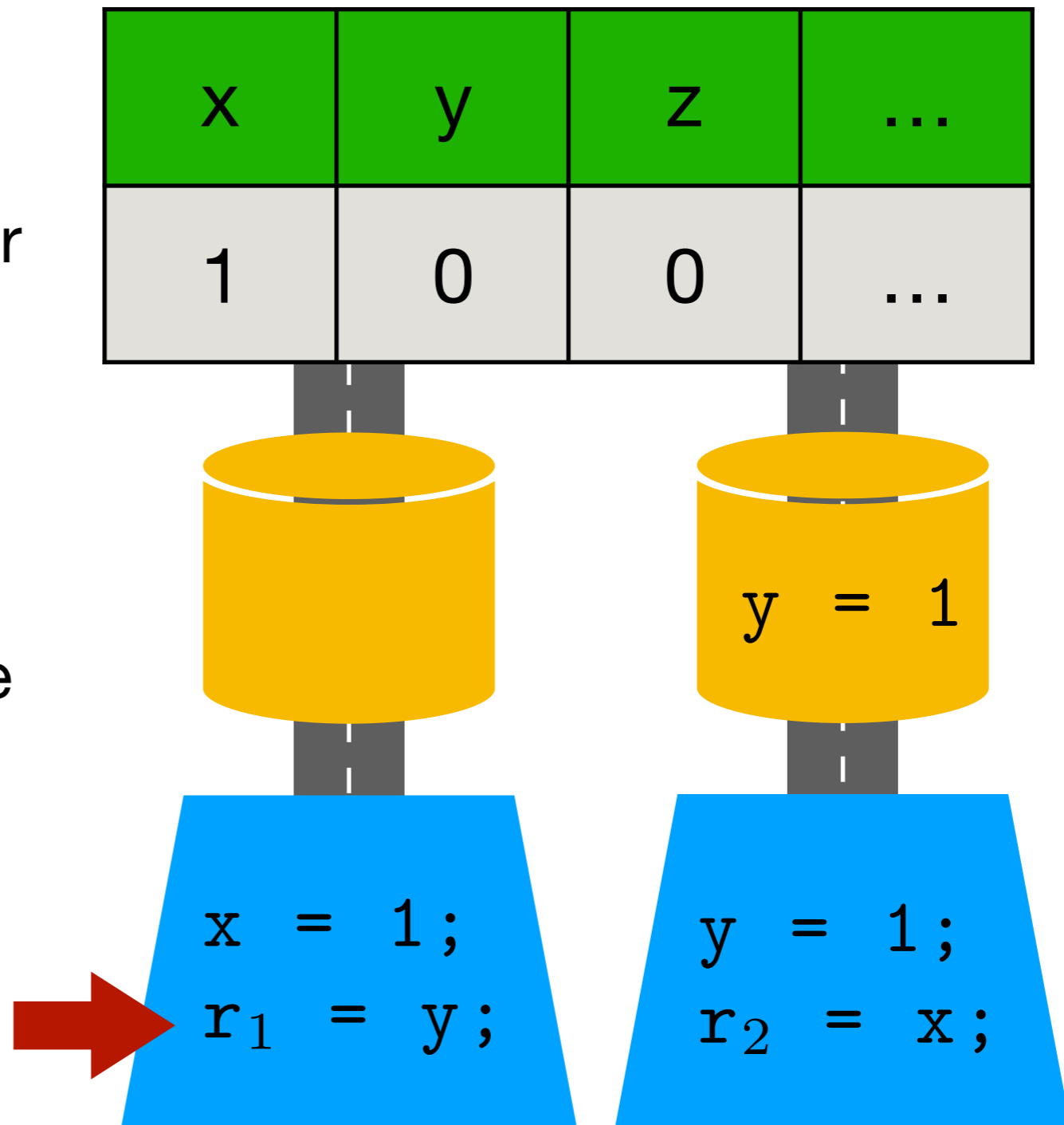


# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

$x = 0 \ \& \ y = 0$	
<div style="display: flex; justify-content: space-between;"> <span><math>x = 1;</math></span> <span><math>y = 1;</math></span> </div>	
$r_1 = y;$	$r_2 = x;$
$r_1 = 0 \ \& \ r_2 = 0$	

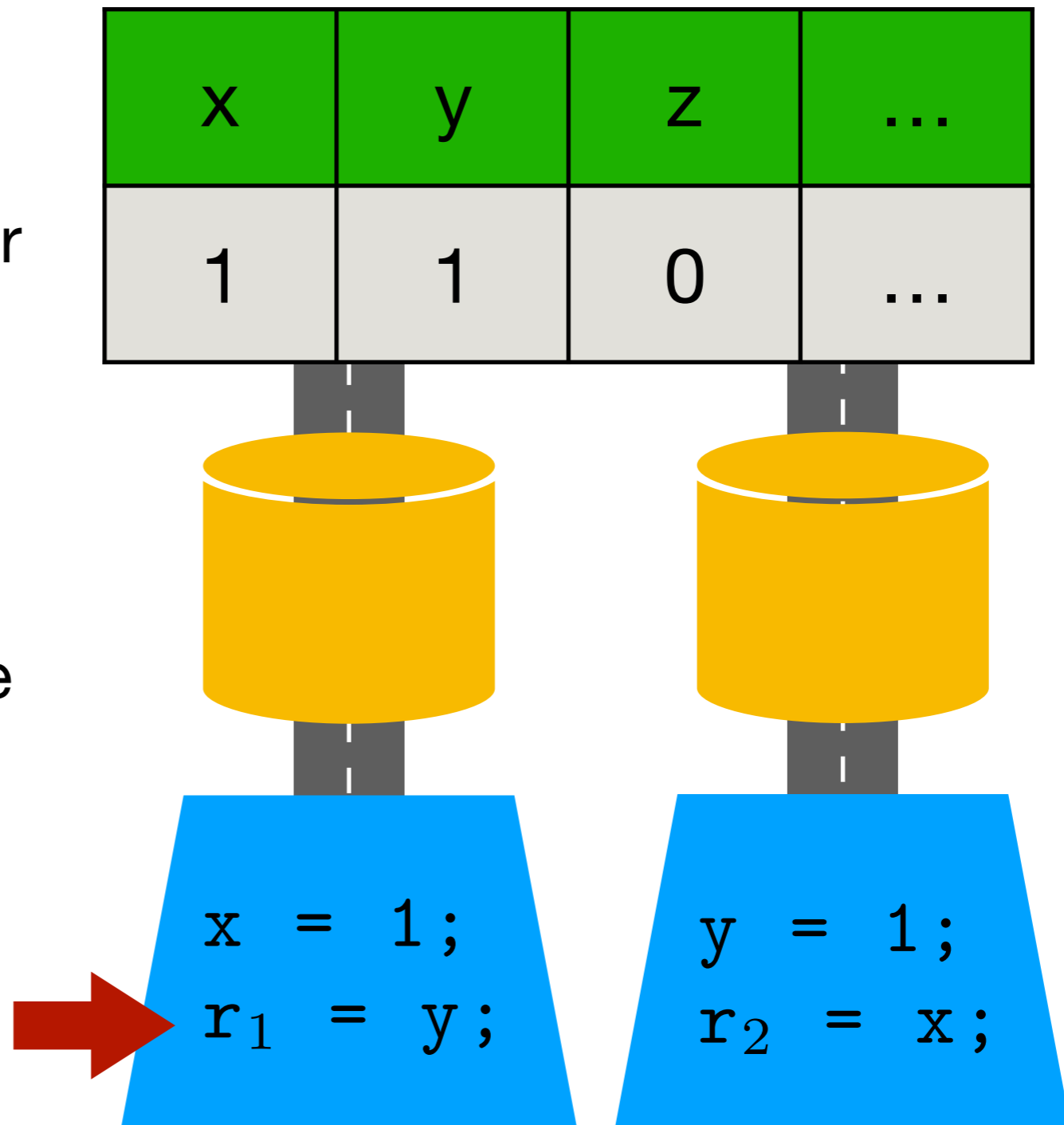


# The Case for Relaxed

## Store Buffers:

- Store x: enqueue into a FIFO buffer per processor
- Load x:
  - find last store to x into the processors buffer
  - if none, lookup into the memory

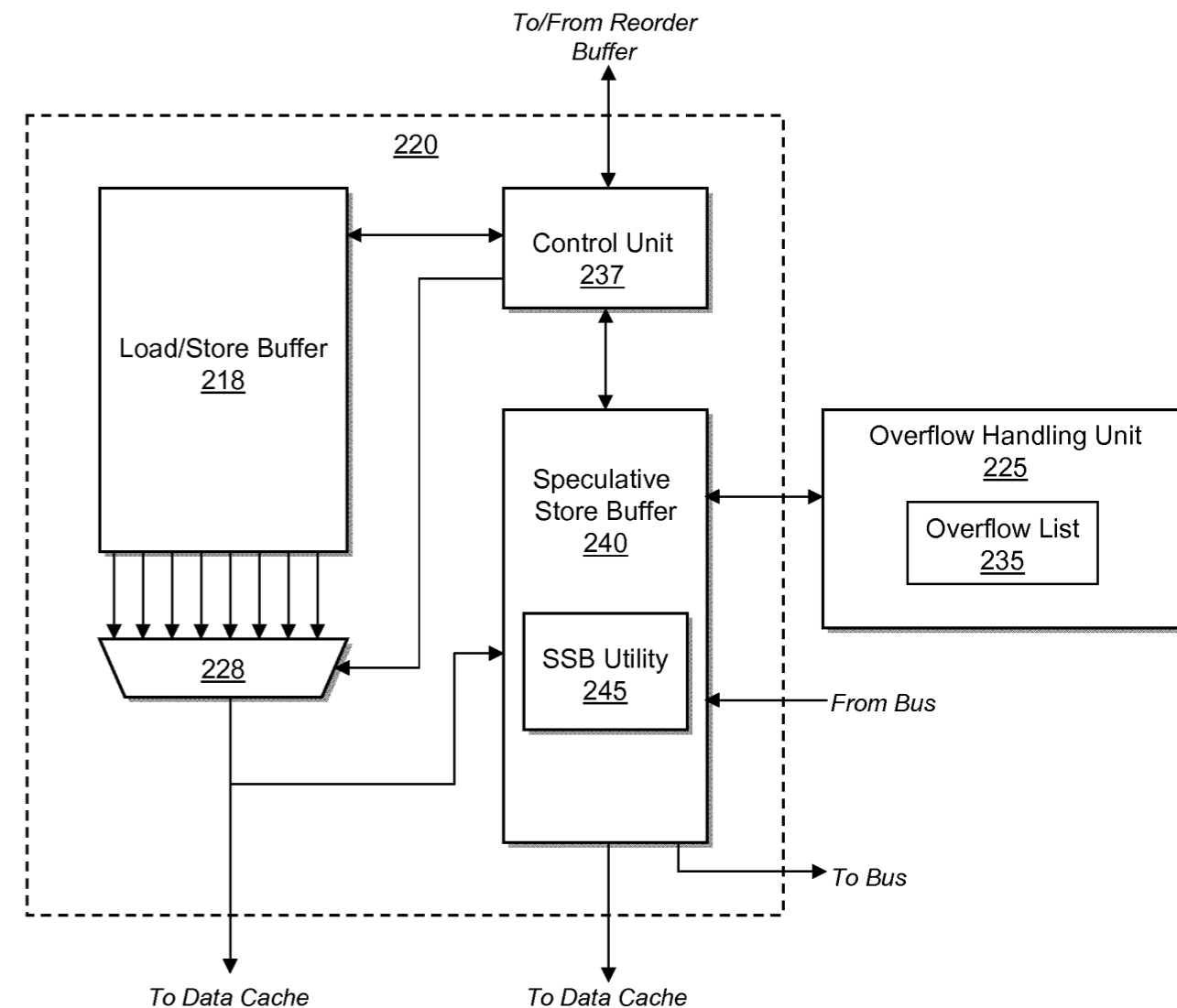
$x = 0 \ \& \ y = 0$	
$x = 1;$	$y = 1;$
$r_1 = y;$	$r_2 = x;$
$r_1 = 0 \ \& \ r_2 = 0$	



# The Case for Relaxed

## Architectural Mechanisms

- Store Buffers
- Caches at different levels (L1, L2)
- Instruction Level Parallelism (ILP)
- Pipelines
- Branch Prediction
- Parallelization
- NUMA
- GPU
- etc.



# The Case for Relaxed

## Architectural Mechanisms

- Store Buffers
- Caches at different levels (L1, L2)
- Instruction Level Parallelism (ILP)
- Pipelines
- Branch Prediction
- Parallelization
- NUMA
- GPU
- etc.

## Architectural Choices

- Manuals are explicitly obscure about the actual mechanisms
- Eg: x86 behaves as if it had store buffers
- Eg: Power behaves as if it had predictive caches
- The Manuals tend to be informal (at best)

# Total Store Ordering (TSO)

# A simple *relaxed* model TSO

The Anomalies / Litmus test

# A simple *relaxed* model TSO

The Anomalies / Litmus test

$x = 0 \ \& \ y = 0$	
<hr/>	
$x = 1;$	$y = 1;$
$r_1 = y;$	$r_2 = x;$
<hr/>	
$r_1 = 0 \ \& \ r_2 = 0$	

# A simple *relaxed* model TSO

## The Anomalies / Litmus test

$$\frac{x = 0 \ \& \ y = 0}{\begin{array}{l} x = 1; \quad || \quad y = 1; \\ r_1 = y; \quad || \quad r_2 = x; \end{array}} \\ r_1 = 0 \ \& \ r_2 = 0$$

$$\frac{x = 0 \ \& \ y = 0}{\begin{array}{l} x = 1; \quad \quad \quad y = 1; \\ r_1 = x; \quad \quad || \quad r_3 = y; \\ r_2 = y; \quad \quad \quad r_4 = x; \end{array}} \\ r_1 = r_3 = 1 \ \& \ r_2 = r_4 = 0$$

# A simple *relaxed* model TSO

## The Anomalies / Litmus test

$$\begin{array}{c}
 \frac{x = 0 \ \& \ y = 0}{x = 1; \parallel y = 1; \\ r_1 = y; \parallel r_2 = x;} \\
 \hline
 r_1 = 0 \ \& \ r_2 = 0
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{x = 0 \ \& \ y = 0}{x = 1; \parallel y = 1; \\ r_1 = x; \parallel r_3 = y; \\ r_2 = y; \parallel r_4 = x;} \\
 \hline
 r_1 = r_3 = 1 \ \& \ r_2 = r_4 = 0
 \end{array}$$

$$\begin{array}{c}
 \frac{x = 0 \ \& \ y = 0}{x = 1; \parallel y = 1; \parallel y = 2; \\ r_1 = x; \parallel r_3 = y; \parallel x = 2; \\ r_2 = y; \parallel r_4 = x;} \\
 \hline
 r_1 = r_3 = 2 \Rightarrow r_2 \neq 0 \vee r_4 \neq 0
 \end{array}$$

# Litmus



## Part I Running tests with litmus7

- [A tour of litmus7](#)
  - [A simple run](#)
  - [Cross compilation](#)
  - [Running several tests at once](#)
- [Controlling test parameters](#)
  - [Architecture of tests](#)
  - [Affinity](#)
  - [Controlling executable files](#)
- [Advanced control of test parameters](#)
  - [Timebase synchronisation mode](#)
  - [Advanced prefetch control](#)
- [Usage of litmus7](#)
  - [Arguments](#)
  - [Options](#)
  - [Configuration files](#)

Traditionally, a *litmus test* is a small parallel program designed to exercise the memory model of a parallel, shared-memory, computer. Given a litmus test in assembler (X86, Power or ARM) litmus7 runs the test.

Using litmus7 thus requires a parallel machine, which must additionally feature `gcc` and the `pthread`s library. Our tool litmus7 has some limitations especially as regards recognised instructions. Nevertheless, litmus7 should accept all tests produced by the companion test generators (see Part [II](#)) and has been successfully used on Linux, MacOS, AIX and Android.

### 1 A tour of litmus7

Litmus

# Litmus

- SB
- SB+rfi-pos
- SBB
- MP
- IRIW

# Operational Formalizations

# A simple language

$v \in \mathcal{Val}$	$::=$	$x \mid \lambda x e \mid tt \mid ff \mid ()$	<i>values</i>
$e \in \mathcal{Expr}$	$::=$	$v \mid (ve) \mid (\text{ref } v)$ $\mid$ $(!v) \mid (v_0 := v_1)$ $\mid$ $(\text{cas } v) \mid \langle \text{wr} \mid \text{rd} \rangle \mid \langle \text{wr} \mid \text{wr} \rangle$	<i>expressions</i>
		$e_0; e_1 \equiv \lambda x e_1 e_0$	

# A simple language

$v \in \mathcal{Val}$	$::=$	$x \mid \lambda x e \mid tt \mid ff \mid ()$	<i>values</i>
$e \in \mathcal{Expr}$	$::=$	$v \mid (ve) \mid (\text{ref } v)$ $\mid (!v) \mid (v_0 := v_1)$ $\mid (\text{cas } v) \mid \langle \text{wr} \mid \text{rd} \rangle \mid \langle \text{wr} \mid \text{wr} \rangle$	<i>expressions</i>
		$e_0; e_1 \equiv \lambda x e_1 e_0$	

$$\begin{array}{l} \text{x} = 1; \parallel \text{y} = 1; \\ \text{r}_1 = \text{y}; \parallel \text{r}_2 = \text{x}; \end{array}$$

# A simple language

$$\begin{array}{lll}
 v \in \mathcal{Val} & ::= & x \mid \lambda x e \mid tt \mid ff \mid () \qquad \text{values} \\
 e \in \mathcal{Expr} & ::= & v \mid (ve) \mid (\text{ref } v) \qquad \text{expressions} \\
 & & \mid (!v) \mid (v_0 := v_1) \\
 & & \mid (\text{cas } v) \mid \langle \text{wr} \mid \text{rd} \rangle \mid \langle \text{wr} \mid \text{wr} \rangle \\
 & & e_0; e_1 \equiv \lambda x e_1 e_0
 \end{array}$$

$$\begin{array}{ll}
 \text{x} = 1; \parallel y = 1; & (x := 1); (!y) \parallel (y := 1); (!x) \\
 \text{r}_1 = y; \parallel \text{r}_2 = \text{x}; &
 \end{array}$$

# A simple language

$v \in \mathcal{Val} \quad ::= \quad x \mid \lambda x e \mid tt \mid ff \mid () \quad \text{values}$

$e \in \mathcal{Expr} \quad ::= \quad v \mid (ve) \mid (\text{ref } v) \quad \text{expressions}$   
 $\quad \mid \quad (!v) \mid (v_0 := v_1)$   
 $\quad \mid \quad (\text{cas } v) \mid \langle \text{wr} \mid \text{rd} \rangle \mid \langle \text{wr} \mid \text{wr} \rangle$

$e_0; e_1 \equiv \lambda x e_1 e_0$

$\mathbf{x} = 1; \parallel \mathbf{y} = 1; \quad (x := 1); (!y) \parallel (y := 1); (!x)$

$\mathbf{r}_1 = \mathbf{y}; \parallel \mathbf{r}_2 = \mathbf{x}; \quad (\lambda z !y)(x := 1) \parallel (\lambda z !x)(y := 1)$

# A simple language

$v \in \mathcal{Val} \quad ::= \quad x \mid \lambda x e \mid tt \mid ff \mid () \quad \text{values}$

$e \in \mathcal{Expr} \quad ::= \quad v \mid (ve) \mid (\text{ref } v)$   
 $\quad \quad \quad \mid \quad (!v) \mid (v_0 := v_1)$   
 $\quad \quad \quad \mid \quad (\text{cas } v) \mid \langle \text{wr} \mid \text{rd} \rangle$

$e_0; e_1 \equiv \lambda x e_1 e_0$

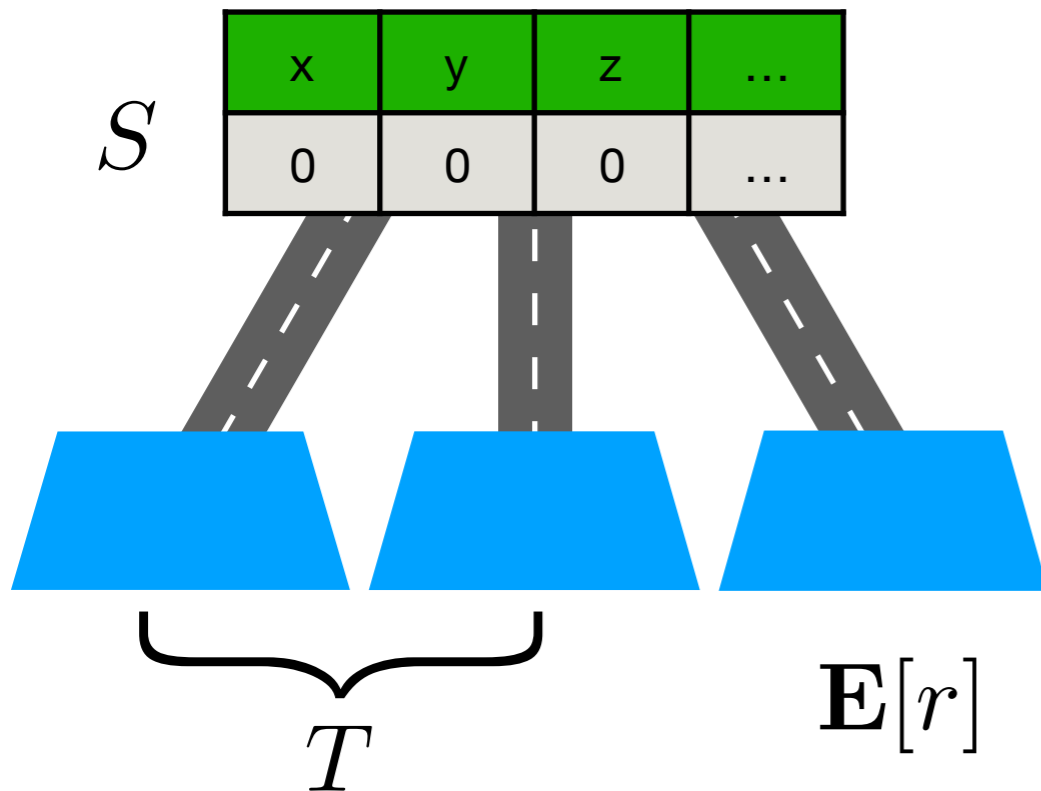
$r \quad ::= \quad (\lambda x ev) \mid (\text{ref } v) \quad \text{redexes}$   
 $\quad \quad \mid \quad (!p) \mid (p := v)$   
 $\quad \quad \mid \quad (\text{cas } p) \mid \langle \text{wr} \mid \text{rd} \rangle \mid \langle \text{wr} \mid \text{wr} \rangle$

$\mathbf{E} \quad ::= \quad [] \mid (v\mathbf{E}) \quad \text{evaluation contexts}$

$p := 1; (!q) \equiv (\lambda x (!q) (p := 1)) \equiv \underbrace{(\lambda x (!q))}_{\mathbf{E}} \underbrace{(p := 1)}_{[r]}$

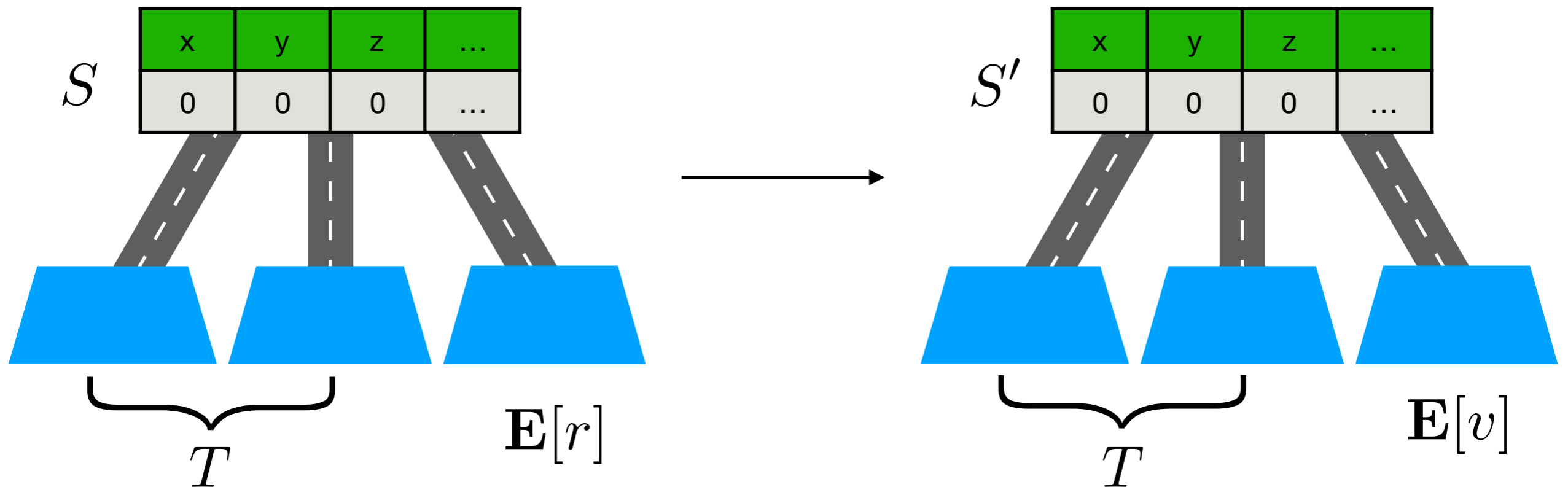
# Sequential Consistency

$$(S, T \parallel \mathbf{E}[(\lambda x \text{ } ev)])$$



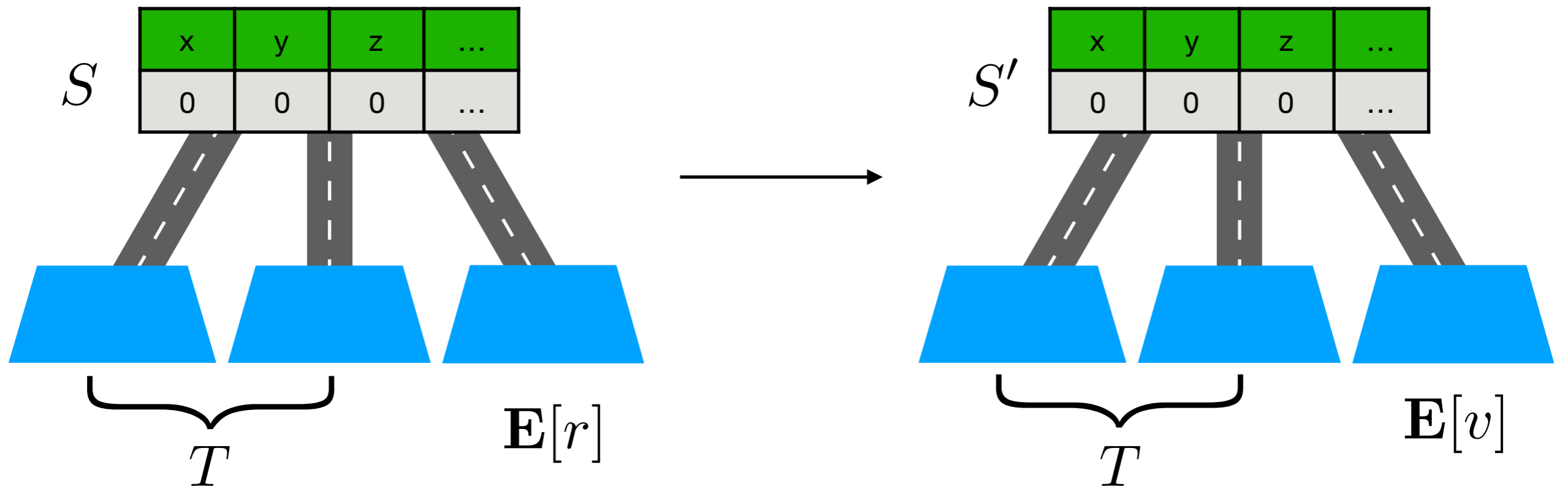
# Sequential Consistency

$$(S, T \parallel \mathbf{E}[(\lambda x \text{ } ev)]) \xrightarrow{\beta} (S, T \parallel \mathbf{E}[\{x/v\}e])$$



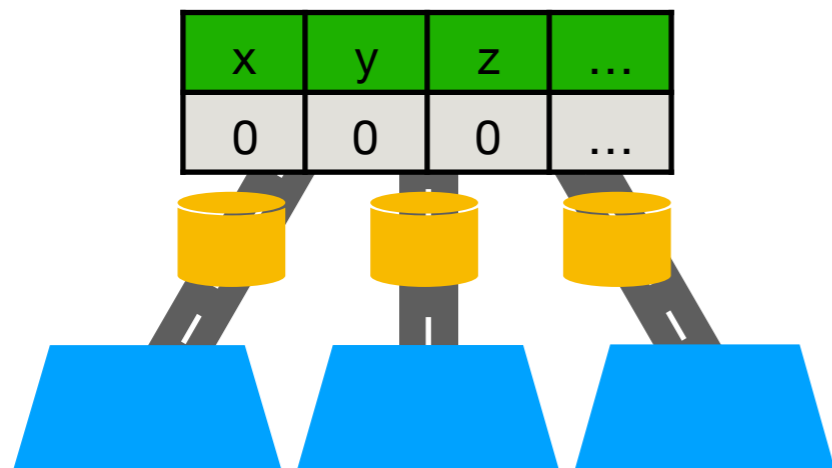
# Sequential Consistency

$$\begin{array}{ll}
 (S, T \parallel \mathbf{E}[(\lambda x \text{ } ev)]) & \xrightarrow{\beta} (S, T \parallel \mathbf{E}[\{x/v\}e]) \\
 (S, T \parallel \mathbf{E}[(\text{ref } v)]) & \xrightarrow{\nu_{p,v}} (S \cup \{p \mapsto v\}, T \parallel \mathbf{E}[p]) \quad p \notin \text{dom}(S) \\
 (S, T \parallel \mathbf{E}[(p := v)]) & \xrightarrow{\text{wr}_{p,v}} (S[p \mapsto v], T \parallel \mathbf{E}[\emptyset]) \\
 (S, T \parallel \mathbf{E}[(!p)]) & \xrightarrow{\text{rd}_{p,v}} (S, T \parallel \mathbf{E}[v]) \quad S(p) = v \\
 (S, T \parallel \mathbf{E}[\langle \text{wr} | \text{rd} \rangle]) & \xrightarrow{\text{wr}} (S, T \parallel \mathbf{E}[\emptyset])
 \end{array}$$



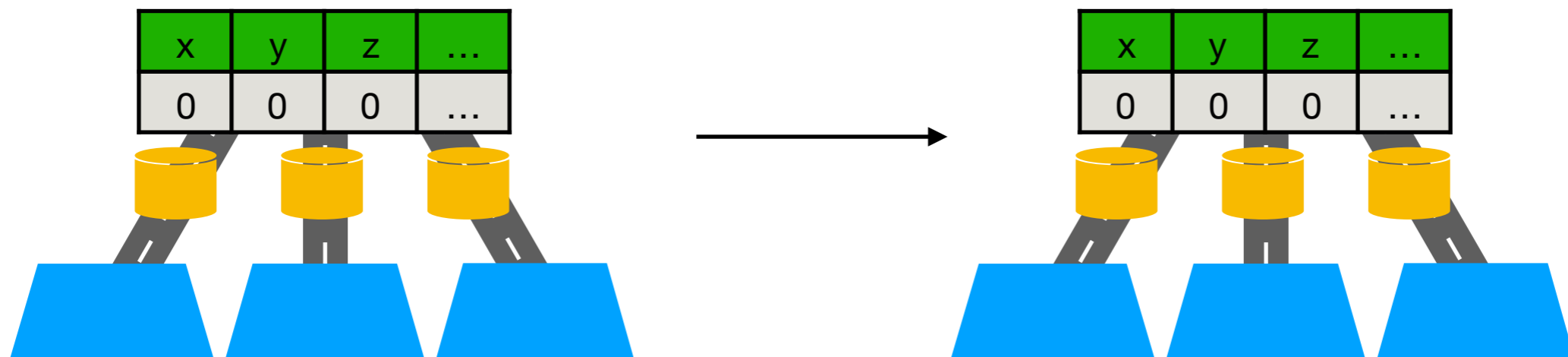
# Write buffer semantics

$(S, T \parallel (B, \mathbf{E}[(\lambda x \text{ } ev)]))$



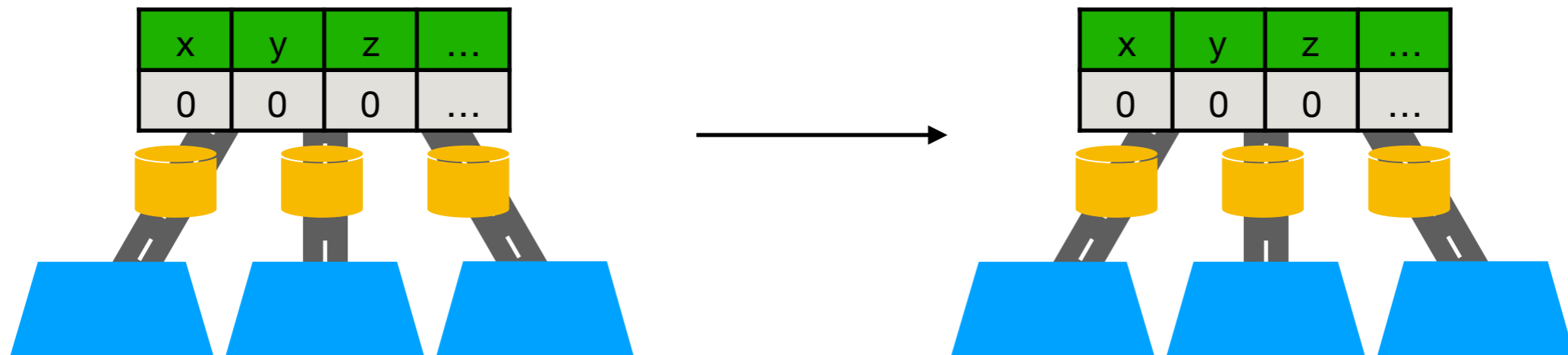
# Write buffer semantics

$$\begin{array}{lll}
 (S, T \parallel (B, \mathbf{E}[(\lambda x \text{ } ev)])) & \xrightarrow{\beta} & (S, T \parallel (B, \mathbf{E}[\{x/v\}e])) \\
 (S, T \parallel (B, \mathbf{E}[(\text{ref } v)])) & \xrightarrow{\nu_{p,v}} & (S \cup \{p \mapsto v\}, T \parallel (B, \mathbf{E}[p])) \quad p \notin \text{dom}(S) \\
 (S, T \parallel (B, \mathbf{E}[(p := v)])) & \xrightarrow{\text{wr}_{p,v}} & (S, T \parallel (B \triangleleft [p \mapsto v], \mathbf{E}[\emptyset])) \\
 (S, T \parallel (B, \mathbf{E}[(! p)])) & \xrightarrow{\text{rd}_{p,v}} & (S, T \parallel (B, \mathbf{E}[v])) \quad B(p) = \epsilon \ \& \ S(p) = v \\
 (S, T \parallel (B, \mathbf{E}[(! p)])) & \xrightarrow{\text{rd}_{p,v}} & (S, T \parallel (B, \mathbf{E}[v])) \quad B(p) = ls :: v \\
 (S, T \parallel (B, \mathbf{E}[\langle \text{wr} | \text{rd} \rangle])) & \xrightarrow{\text{wr}} & (S, T \parallel (B, \mathbf{E}[\emptyset])) \quad \forall p, B(p) = \epsilon
 \end{array}$$



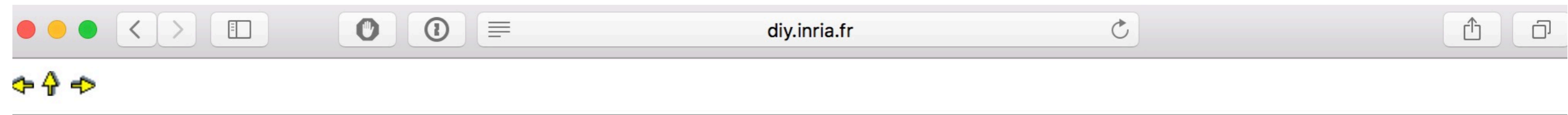
# Write buffer semantics

$$\begin{array}{ll}
 (S, T \parallel (B, \mathbf{E}[(\lambda x \text{ } ev)])) & \xrightarrow{\beta} (S, T \parallel (B, \mathbf{E}[\{x/v\}e])) \\
 (S, T \parallel (B, \mathbf{E}[(\text{ref } v)])) & \xrightarrow{\nu_{p,v}} (S \cup \{p \mapsto v\}, T \parallel (B, \mathbf{E}[p])) \quad p \notin \text{dom}(S) \\
 (S, T \parallel (B, \mathbf{E}[(p := v)])) & \xrightarrow{\text{wr}_{p,v}} (S, T \parallel (B \triangleleft [p \mapsto v], \mathbf{E}[\emptyset])) \\
 (S, T \parallel (B, \mathbf{E}[(! p)])) & \xrightarrow{\text{rd}_{p,v}} (S, T \parallel (B, \mathbf{E}[v])) \quad B(p) = \epsilon \ \& \ S(p) = v \\
 (S, T \parallel (B, \mathbf{E}[(! p)])) & \xrightarrow{\text{rd}_{p,v}} (S, T \parallel (B, \mathbf{E}[v])) \quad B(p) = ls :: v \\
 (S, T \parallel (B, \mathbf{E}[\langle \text{wr} | \text{rd} \rangle])) & \xrightarrow{\text{wr}} (S, T \parallel (B, \mathbf{E}[\emptyset])) \quad \forall p, B(p) = \epsilon \\
 (S, T \parallel ([p \mapsto v] \triangleright B, e)) & \xrightarrow{\text{bu}_{p,v}} (S[p \mapsto v], T \parallel (B, e)) \quad \textcolor{red}{TSO}
 \end{array}$$



# Axiomatic Formalizations

# Herd



## Part III

### Simulating memory models with herd7

- Writing simple models
  - Sequential consistency
  - Total Store Order (TSO)
  - Sequential consistency, total order definition
  - Computing coherence orders
- Producing pictures of executions
  - Graph modes
  - Showing forbidden executions
- Model definitions
  - Overview
  - Identifiers
  - Expressions
  - Instructions
  - Bell extensions
  - Models
  - Primitives
  - Library
- Usage of herd7
  - Arguments
  - Options
  - Configuration files

# Partial Orders

- Strict Partial Orders
  - irreflexive, transitive
  - $po$  or  $\xrightarrow{po}$  for program order
- Operations
  - inverse:  $po^{-1}$
  - transitive closure:  $po^{+}$
  - composition:  $po; rf$
  - set operations:  $po \cup rf \quad po \cap rf$
- Conditions on Orders
  - Acyclicity
  - Irreflexivity
  - Transitive
  - Consistency
$$(po \cup rf)^{+} \cap id = \emptyset$$

# Burckhardt's cheatsheet

Property	Element-wise Definition $\forall x, y, z \in A :$	Algebraic Definition
symmetric	$x \xrightarrow{\text{rel}} y \Rightarrow y \xrightarrow{\text{rel}} x$	$\text{rel} = \text{rel}^{-1}$
reflexive	$x \xrightarrow{\text{rel}} x$	$\text{id}_A \subseteq \text{rel}$
irreflexive	$x \not\xrightarrow{\text{rel}} x$	$\text{id}_A \cap \text{rel} = \emptyset$
transitive	$(x \xrightarrow{\text{rel}} y \xrightarrow{\text{rel}} z) \Rightarrow (x \xrightarrow{\text{rel}} z)$	$(\text{rel} ; \text{rel}) \subseteq \text{rel}$
acyclic	$\neg(x \xrightarrow{\text{rel}} \dots \xrightarrow{\text{rel}} x)$	$\text{id}_A \cap \text{rel}^+ = \emptyset$
total	$x \neq y \Rightarrow (x \xrightarrow{\text{rel}} y \vee y \xrightarrow{\text{rel}} x)$	$\text{rel} \cup \text{rel}^{-1} \cup \text{id}_A = A \times A$

Property	Definition
natural	$\forall x \in A :  \text{rel}^{-1}(x)  < \infty$
partialorder	irreflexive $\wedge$ transitive
totalorder	partialorder $\wedge$ total
enumeration	totalorder $\wedge$ natural
equivalencerelation	reflexive $\wedge$ transitive $\wedge$ symmetric

**Figure 2.1:** Definitions of common properties of a binary relation  $\text{rel} \subseteq A \times A$ .

# TSO — The Model

SC

$x = 0 \ \& \ y = 0$

$x = 1;$

$y = 1;$

$r_1 = y;$

$r_2 = x;$

$r_1 = 1 \ \& \ r_2 = 0$

# TSO — The Model

SC

$x = 0 \ \& \ y = 0$

$\text{po} \downarrow$   
 $x = 1;$   
 $r_1 = y;$

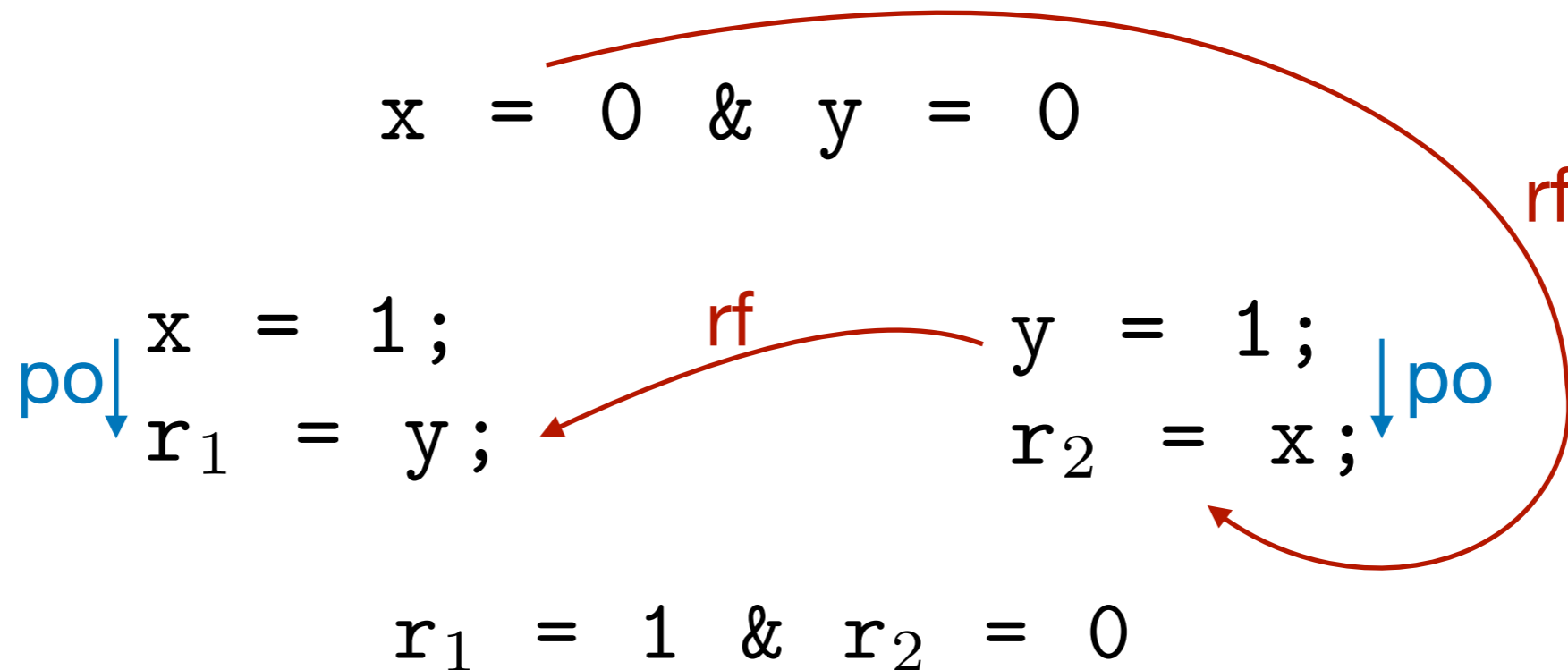
$y = 1;$   
 $r_2 = x;$   
 $\downarrow \text{po}$

$r_1 = 1 \ \& \ r_2 = 0$

po → program order    Total order on the actions of each process

# TSO — The Model

SC



$\text{po} \rightarrow$   
 $\text{rf} \rightarrow$

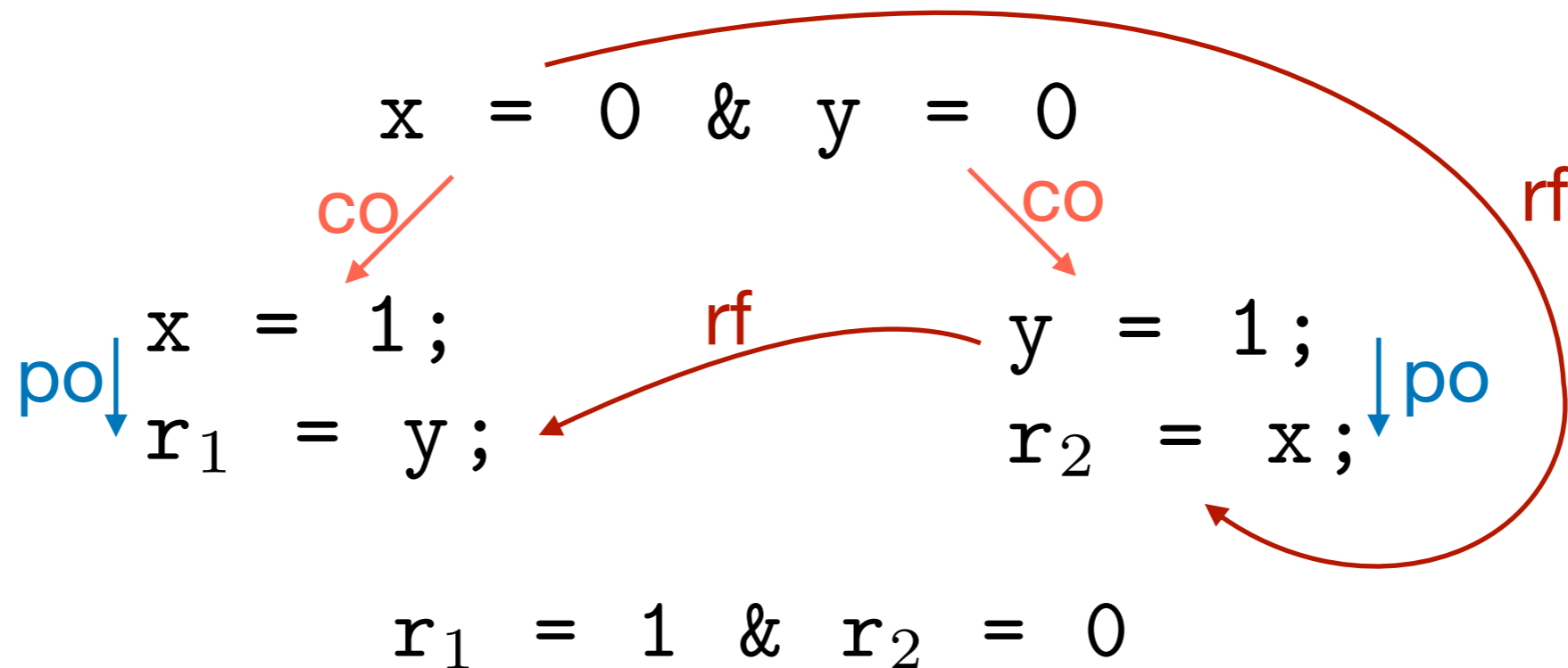
program order  
reads from

Total order on the actions of each process

Relates a read to the write that stores its value

# TSO — The Model

SC



po →

program order      Total order on the actions of each process

rf →

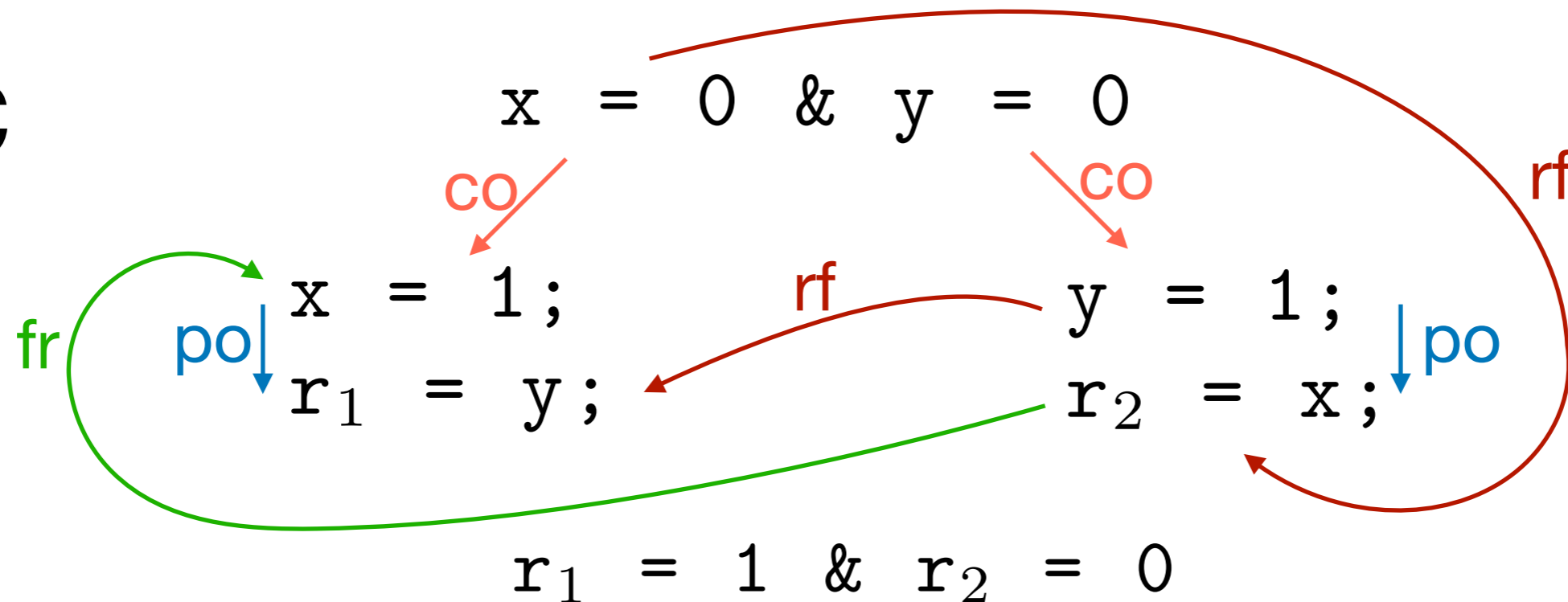
reads from      Relates a read to the write that stores its value

co →

commit order      Relates writes to the same address

# TSO — The Model

SC



po →

program order      Total order on the actions of each process

rf →

reads from      Relates a read to the write that stores its value

co →

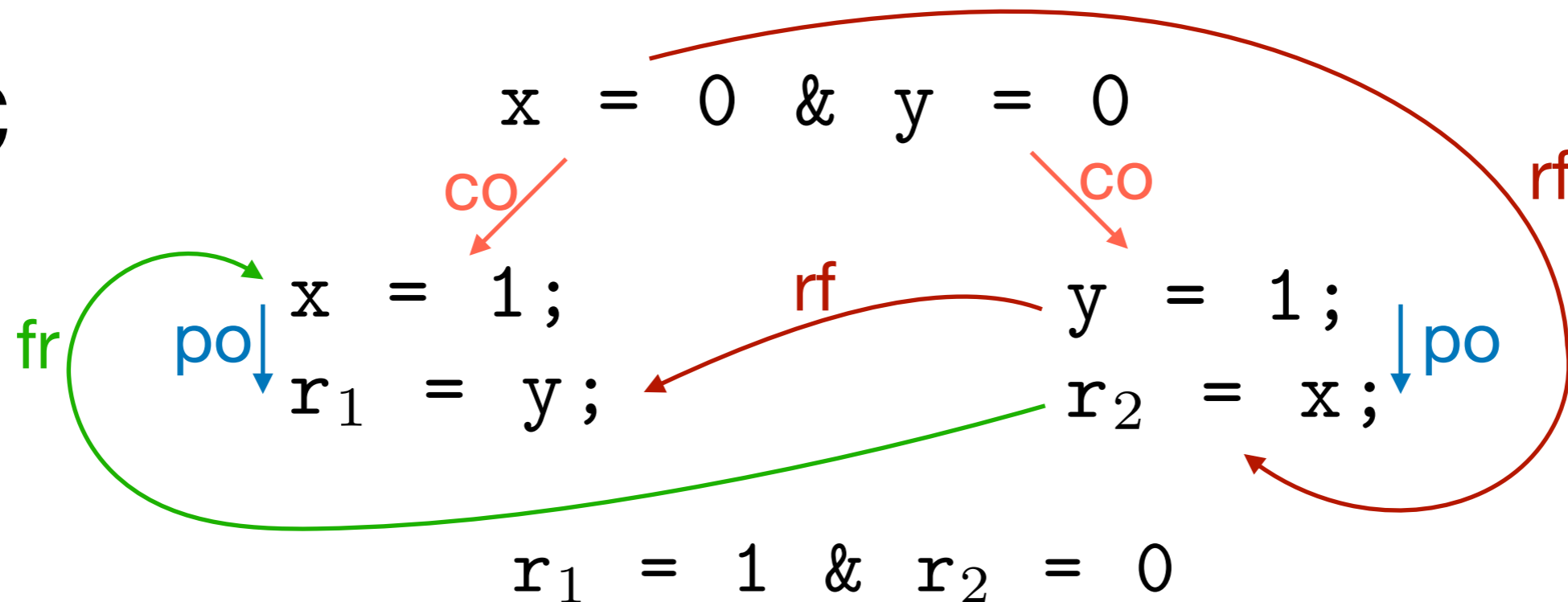
commit order      Relates writes to the same address

fr →

from reads      Read to write order derived from rf and co

# TSO — The Model

SC



po →

program order      Total order on the actions of each process

rf →

reads from      Relates a read to the write that stores its value

co →

commit order      Relates writes to the same address

fr →

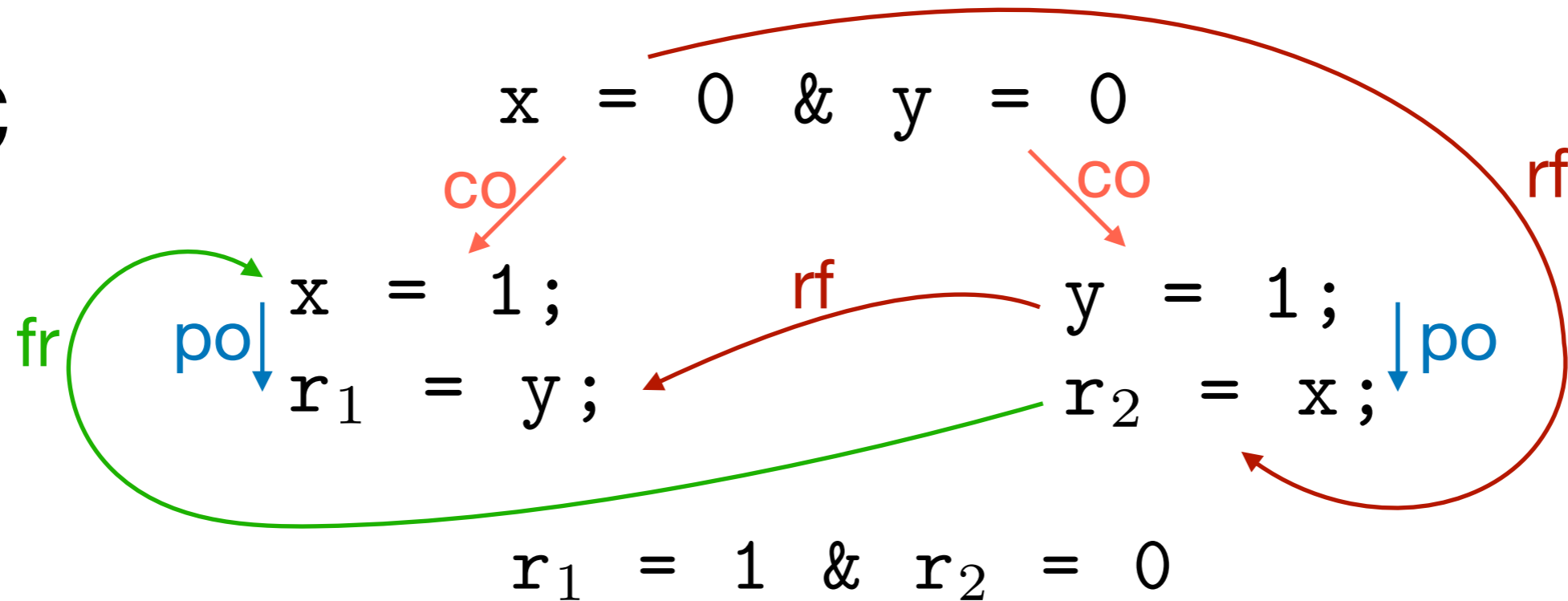
from reads      Read to write order derived from rf and co

hb →

happens before\*  $(po \cup rf)^+$

# TSO — The Model

SC



SC

Initial state:  $x = 0 \ \& \ y = 0$

Processor 1 (P1) execution:

- $x = 1;$
- $r_1 = y;$

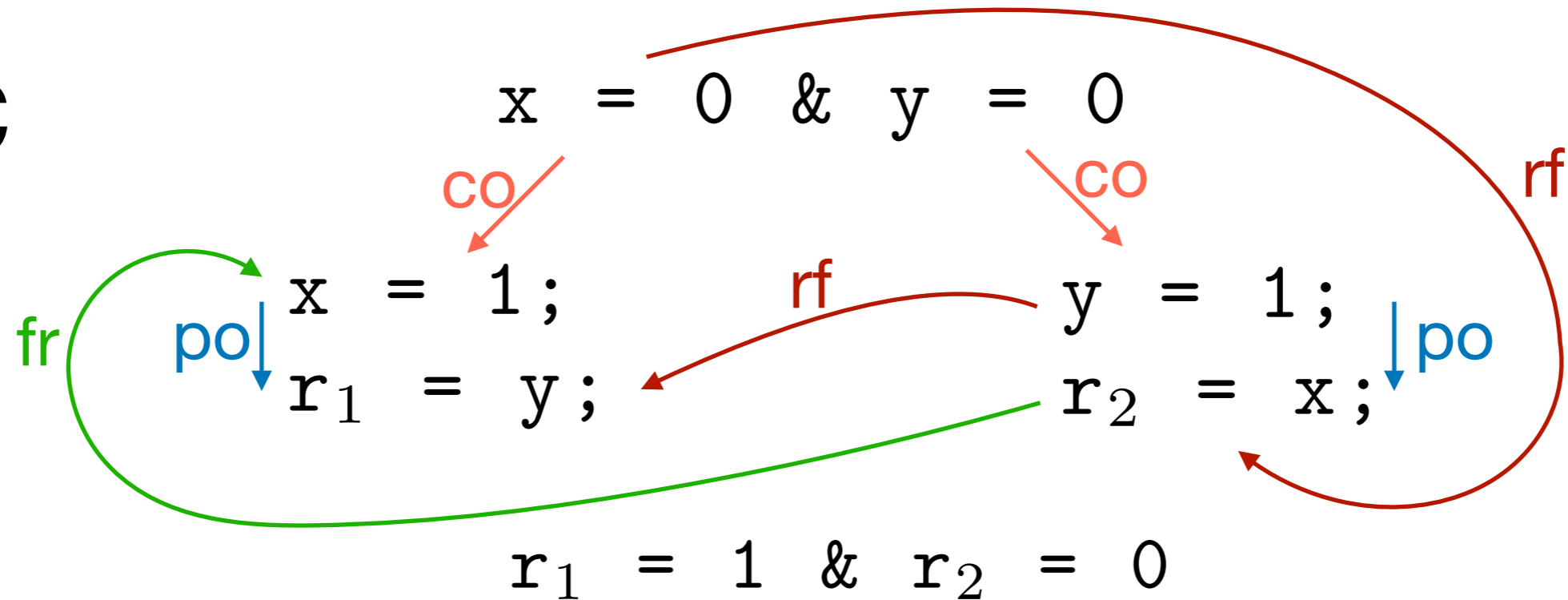
Processor 2 (P2) execution:

- $y = 1;$
- $r_2 = x;$

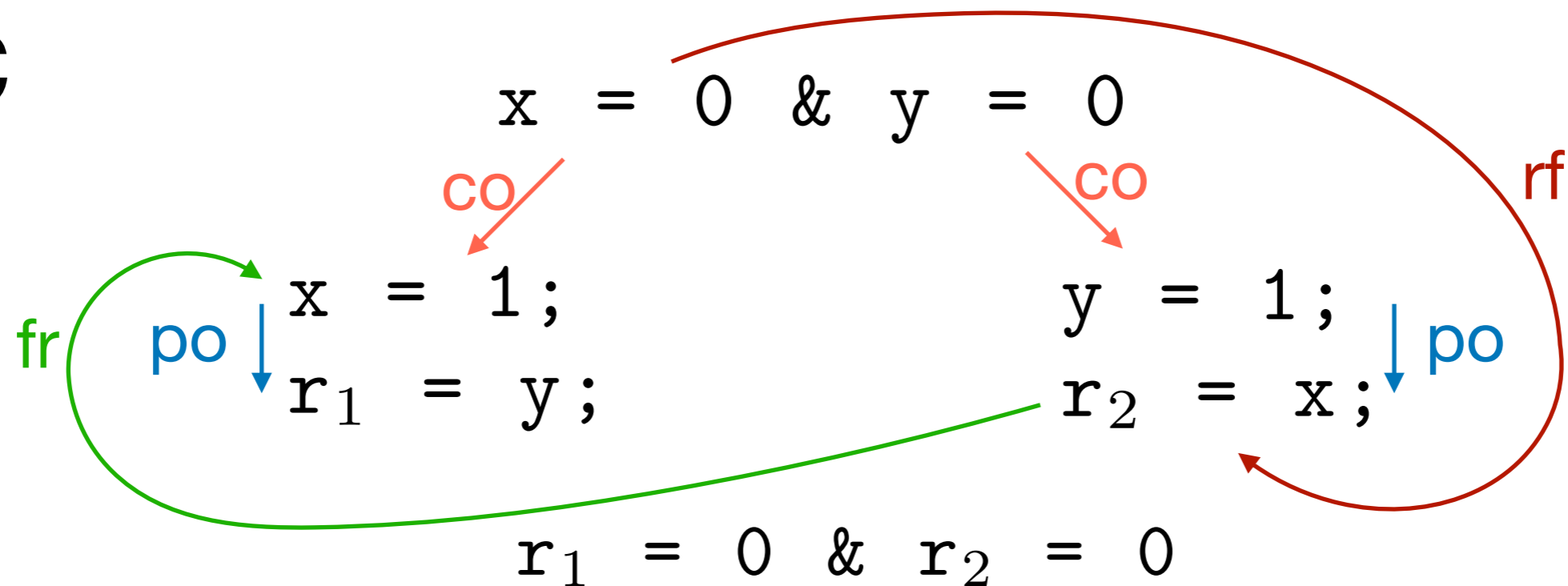
Final state:  $r_1 = 0 \ \& \ r_2 = 0$

# TSO — The Model

SC

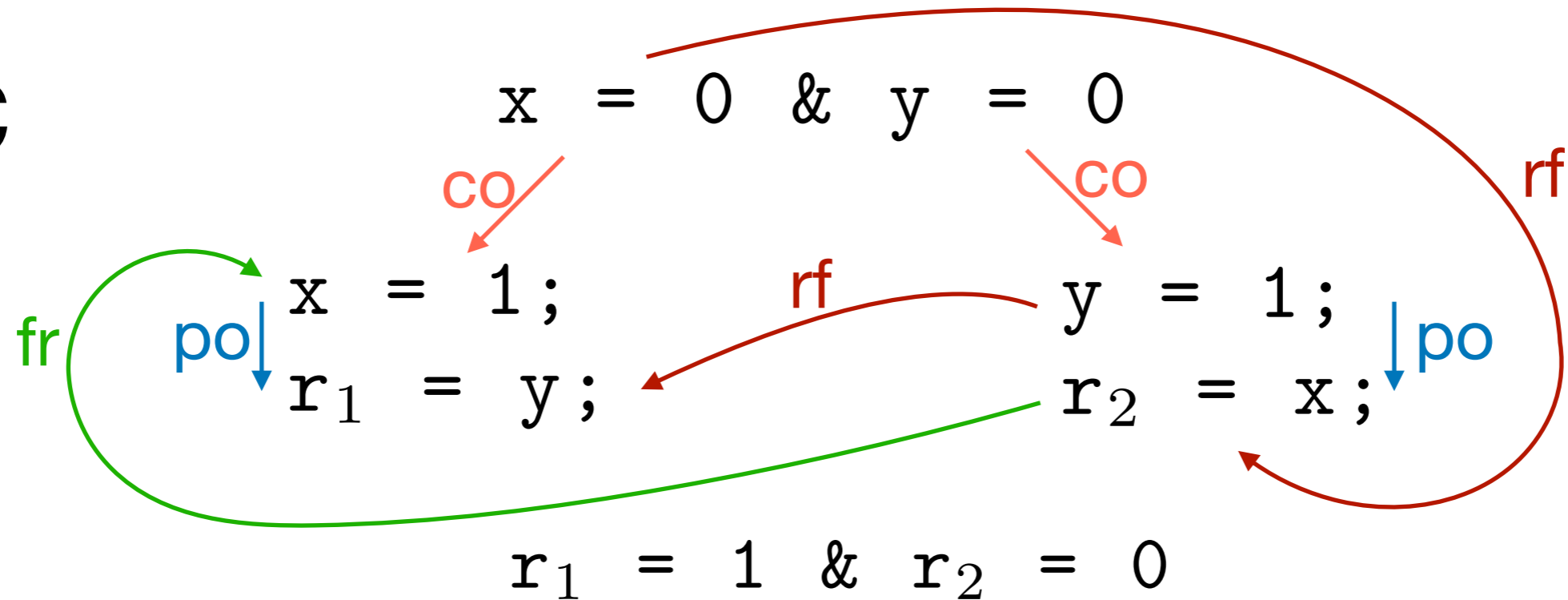


SC

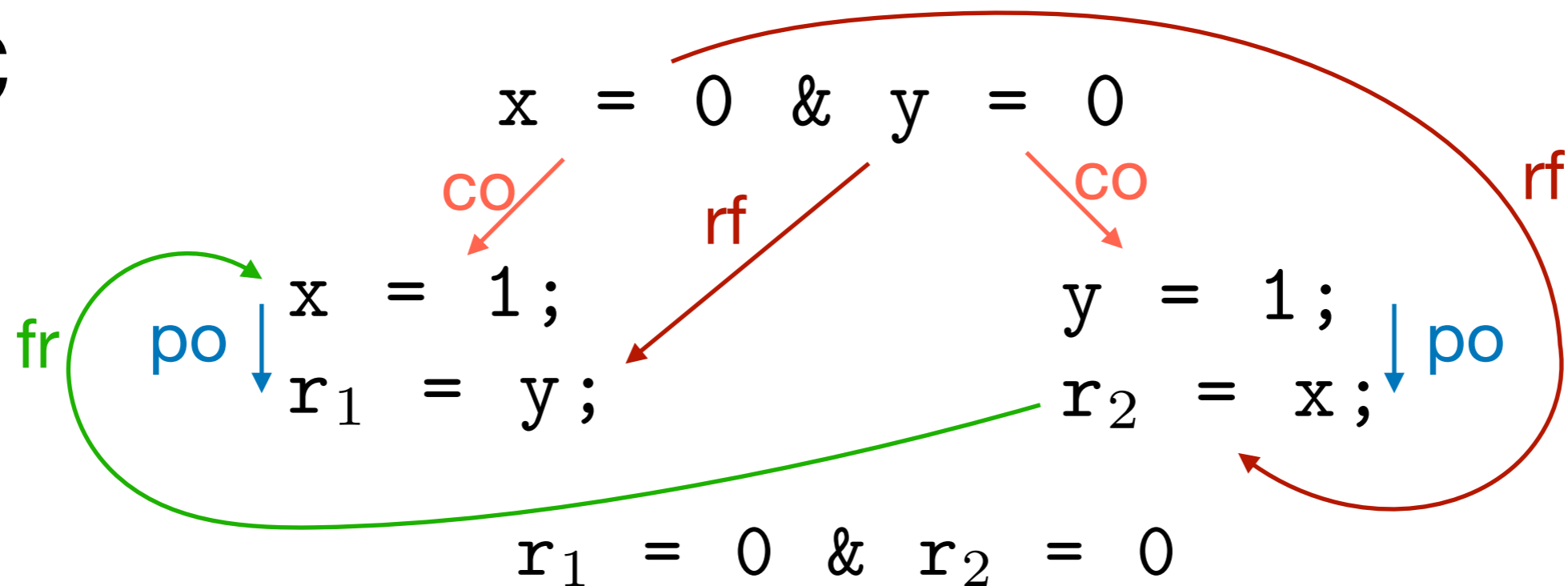


# TSO — The Model

SC

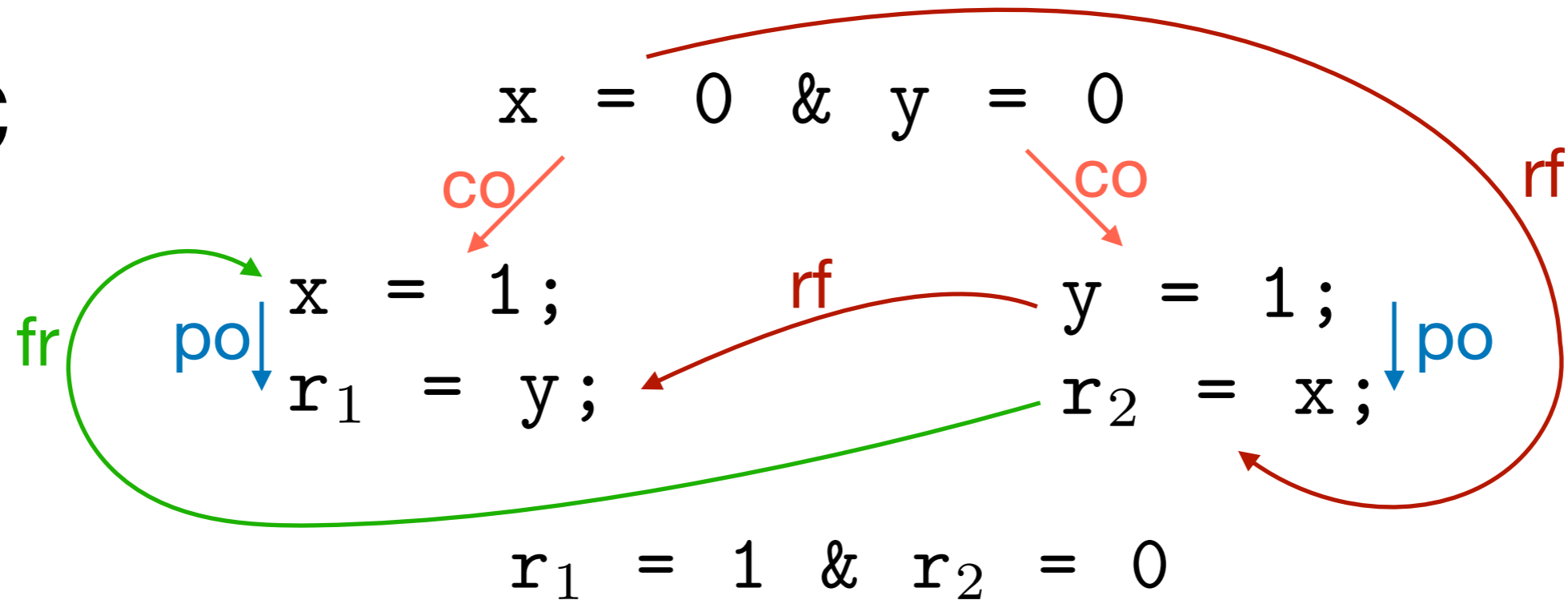


SC

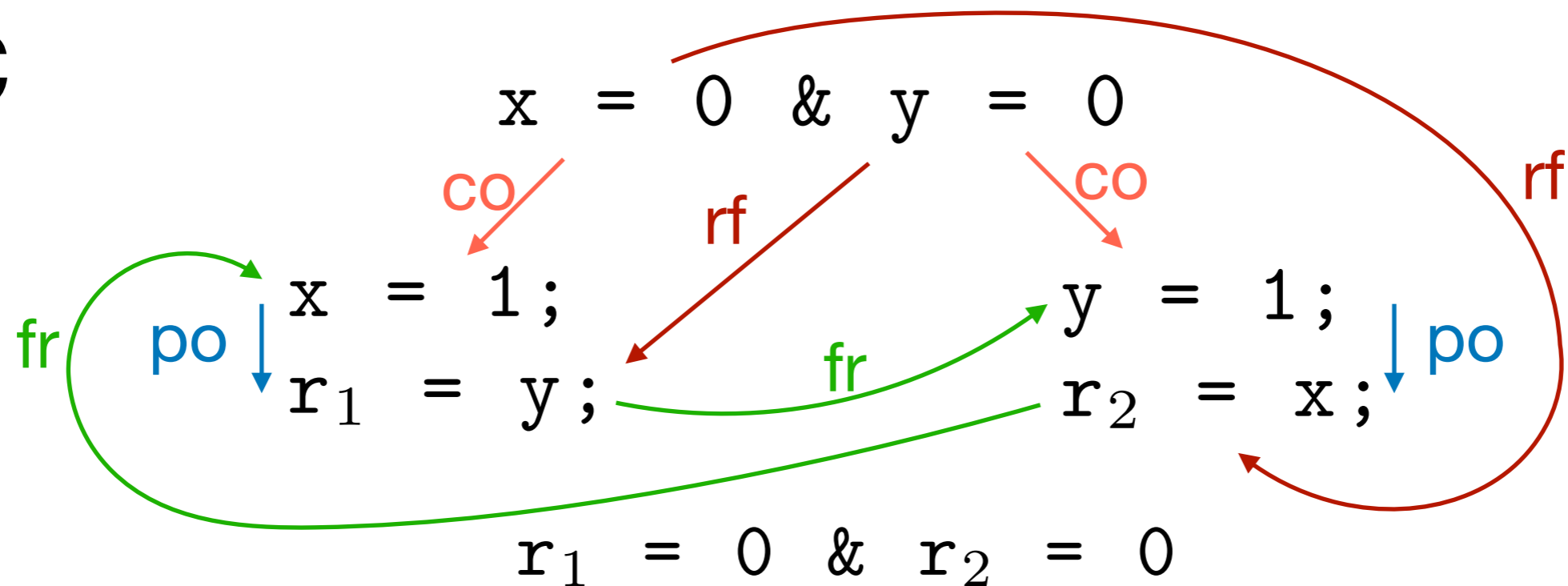


# TSO — The Model

SC

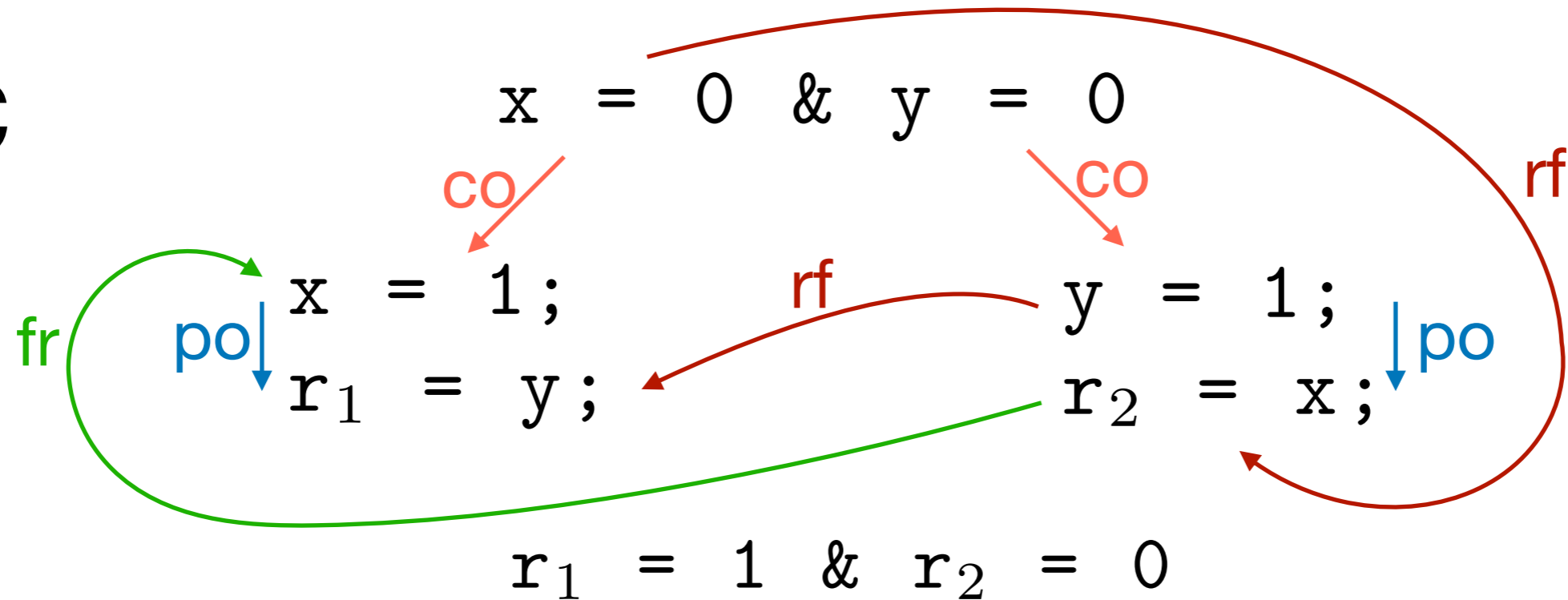


SC

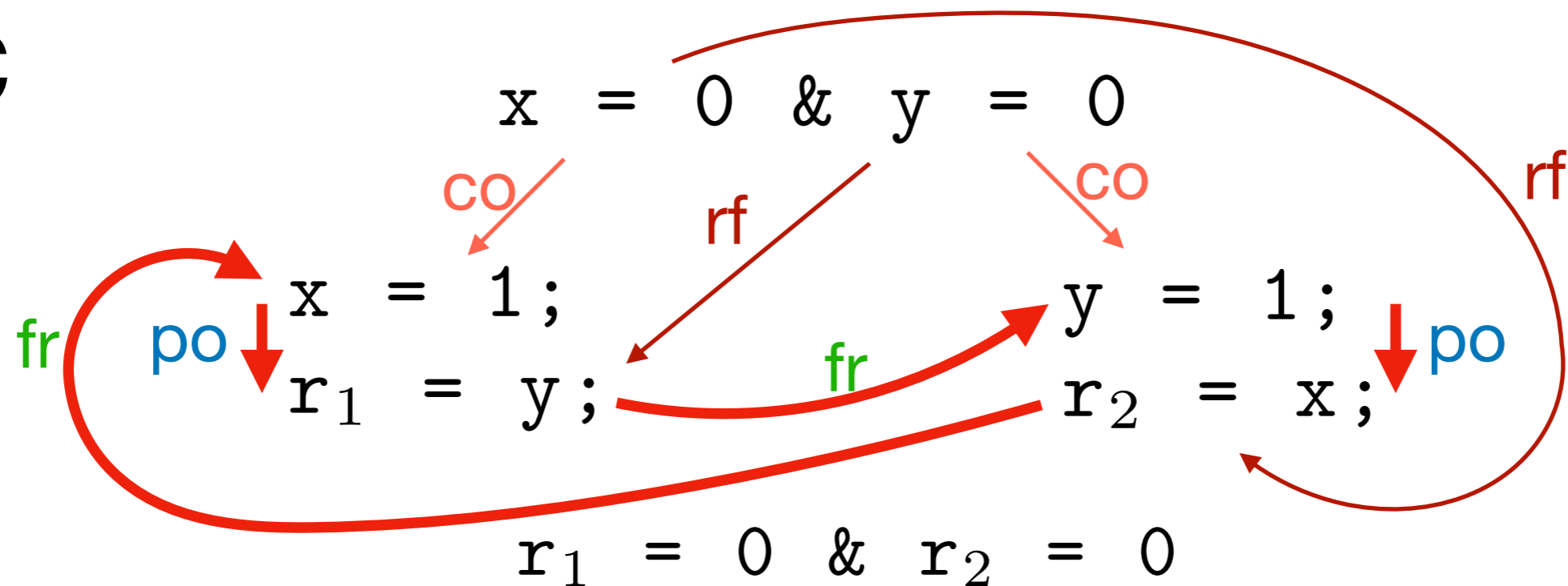


# TSO — The Model

SC

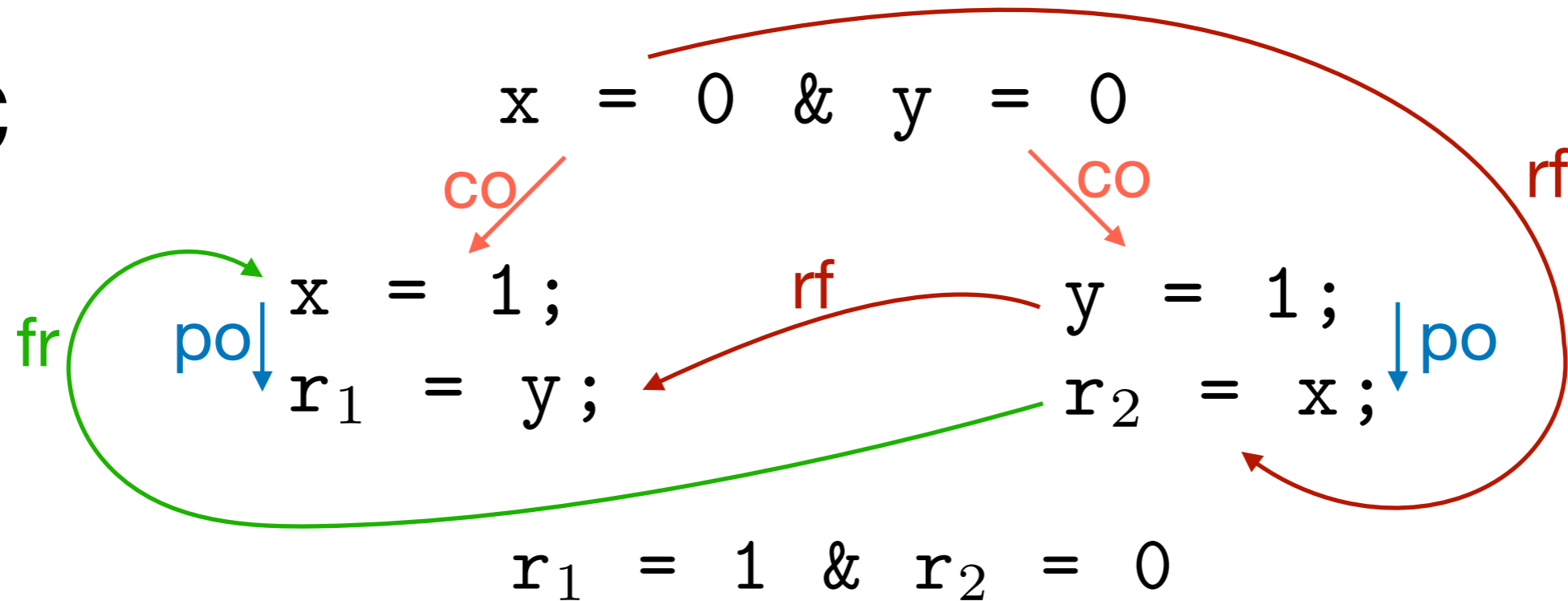


SC

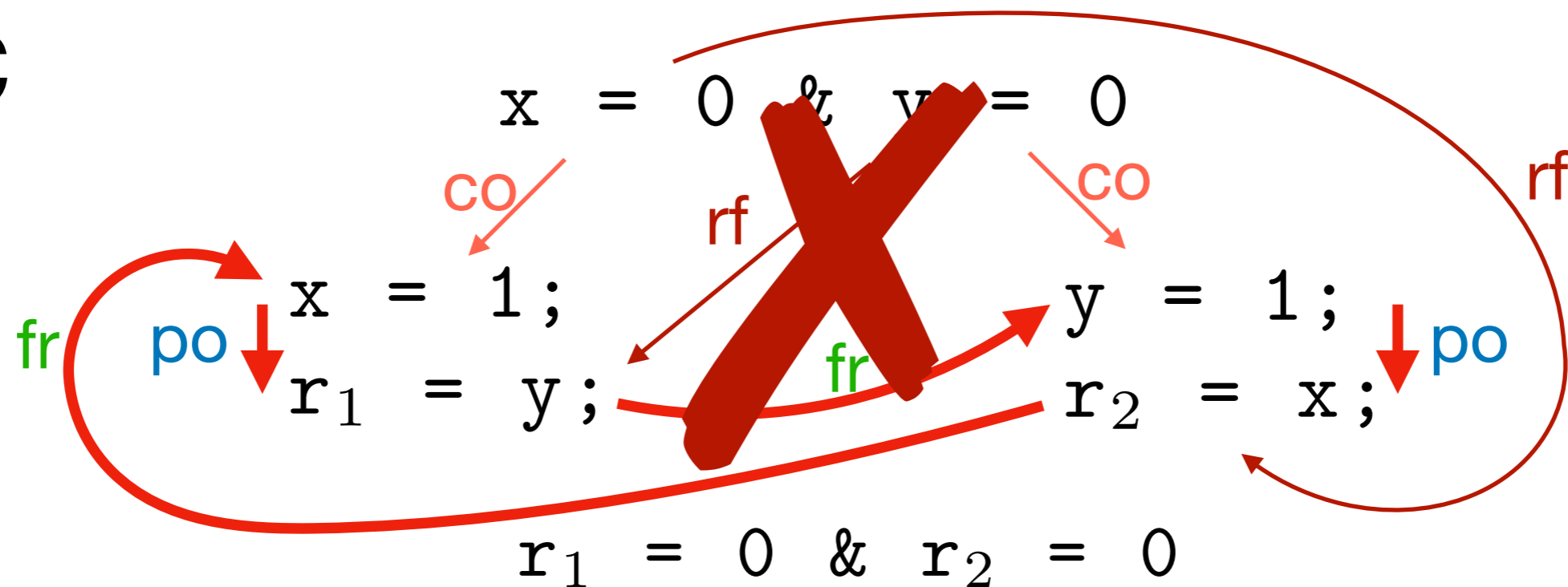


# TSO — The Model

SC

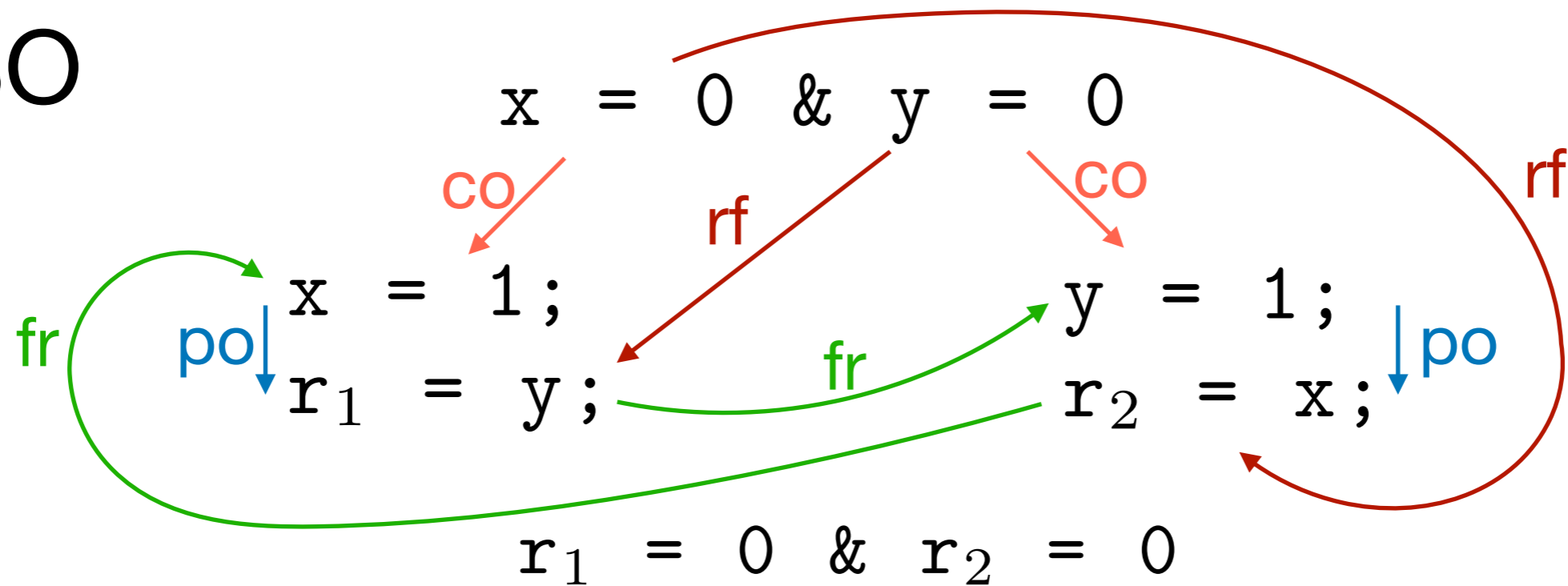


SC



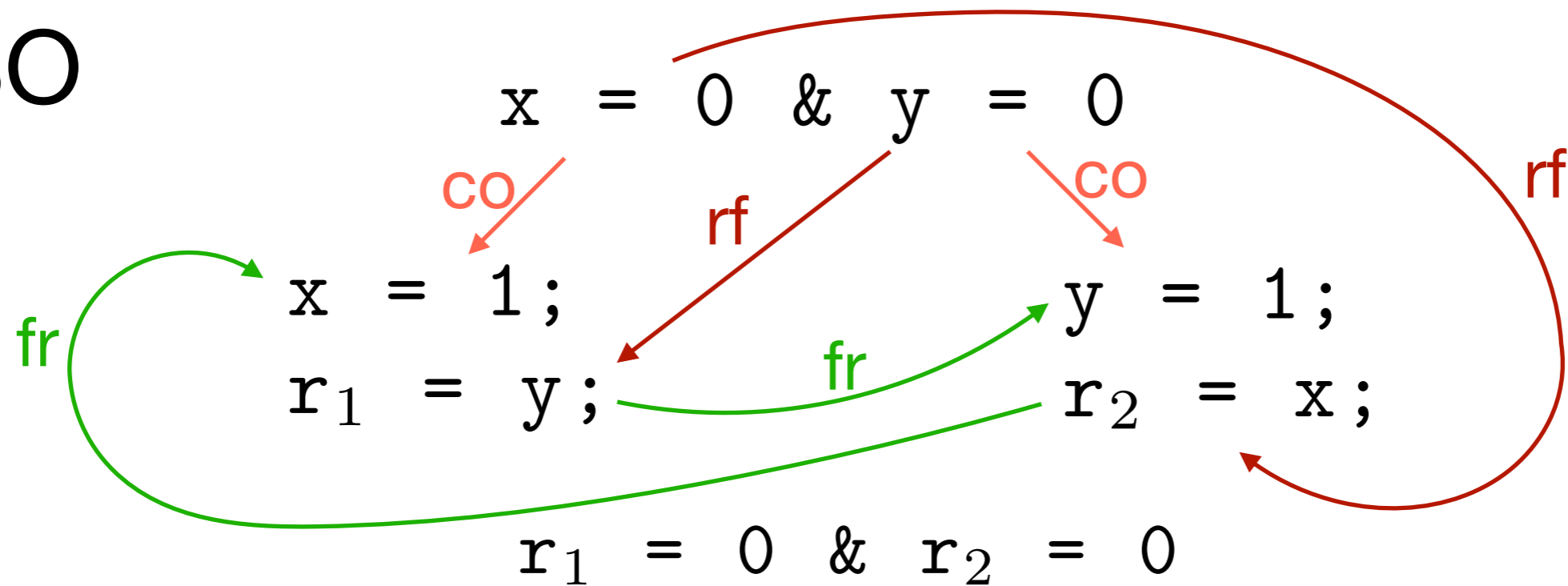
# TSO — The Model

TSO



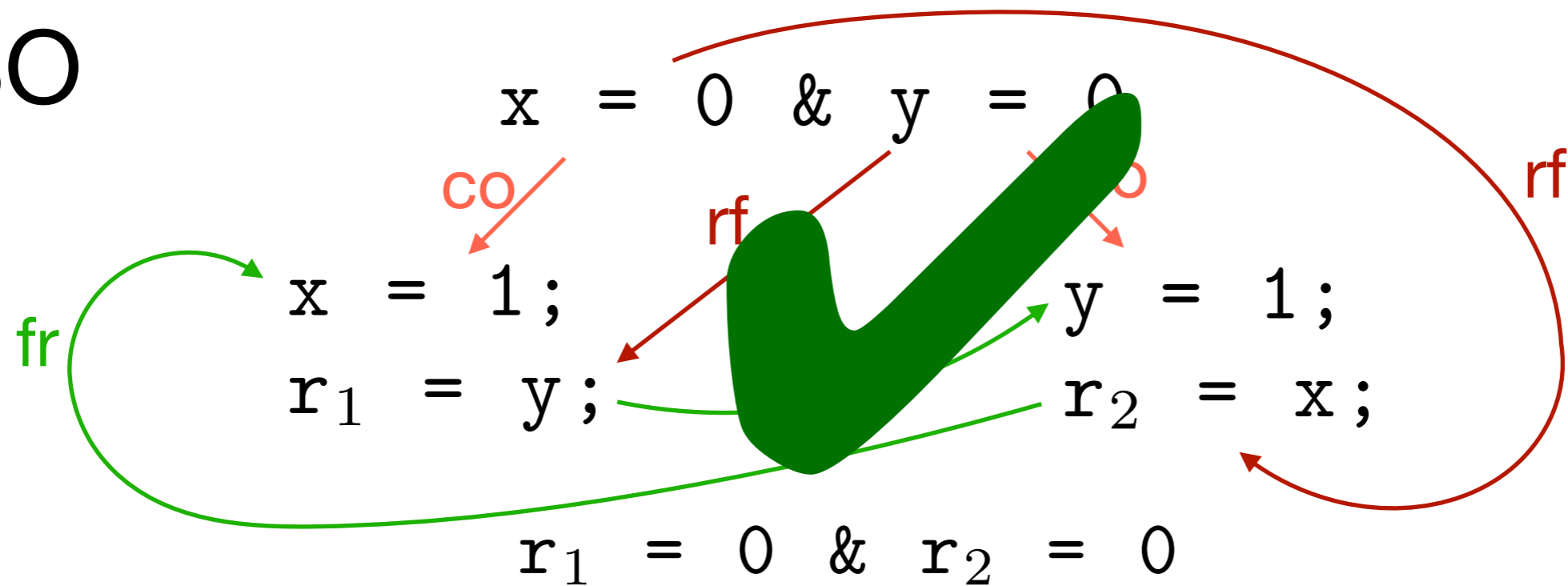
# TSO — The Model

TSO



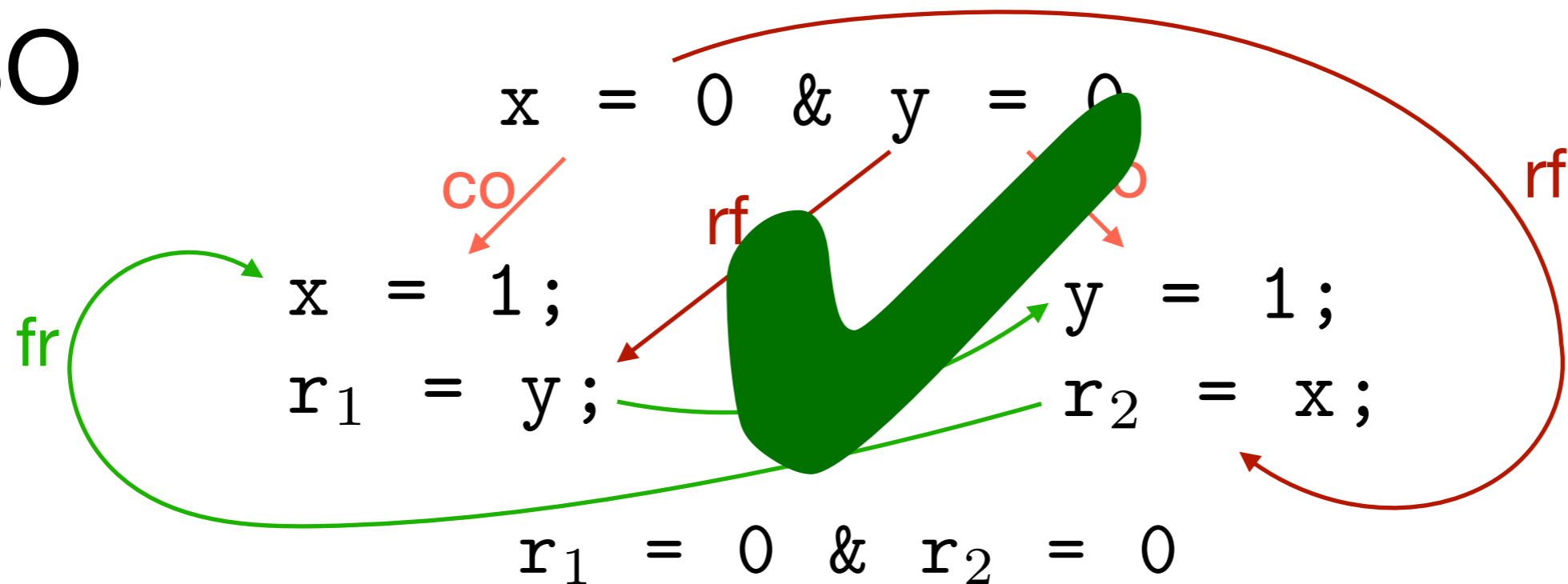
# TSO — The Model

TSO

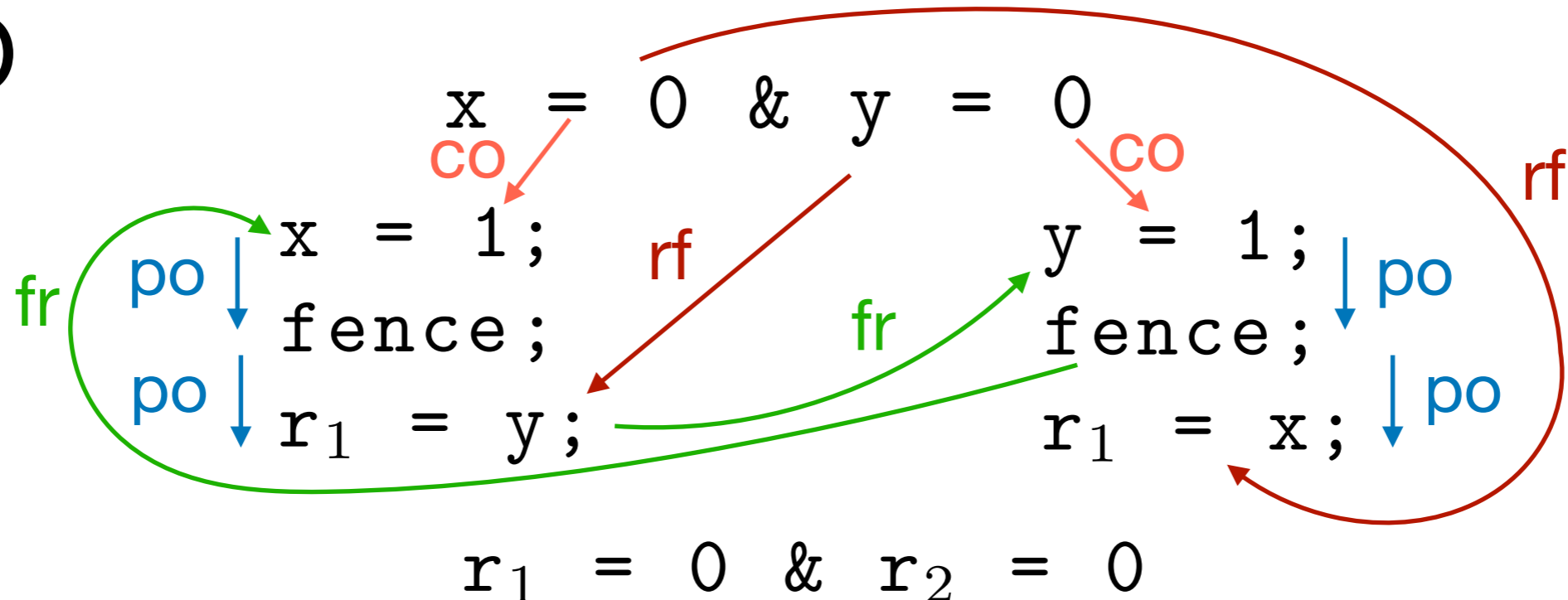


# TSO — The Model

TSO

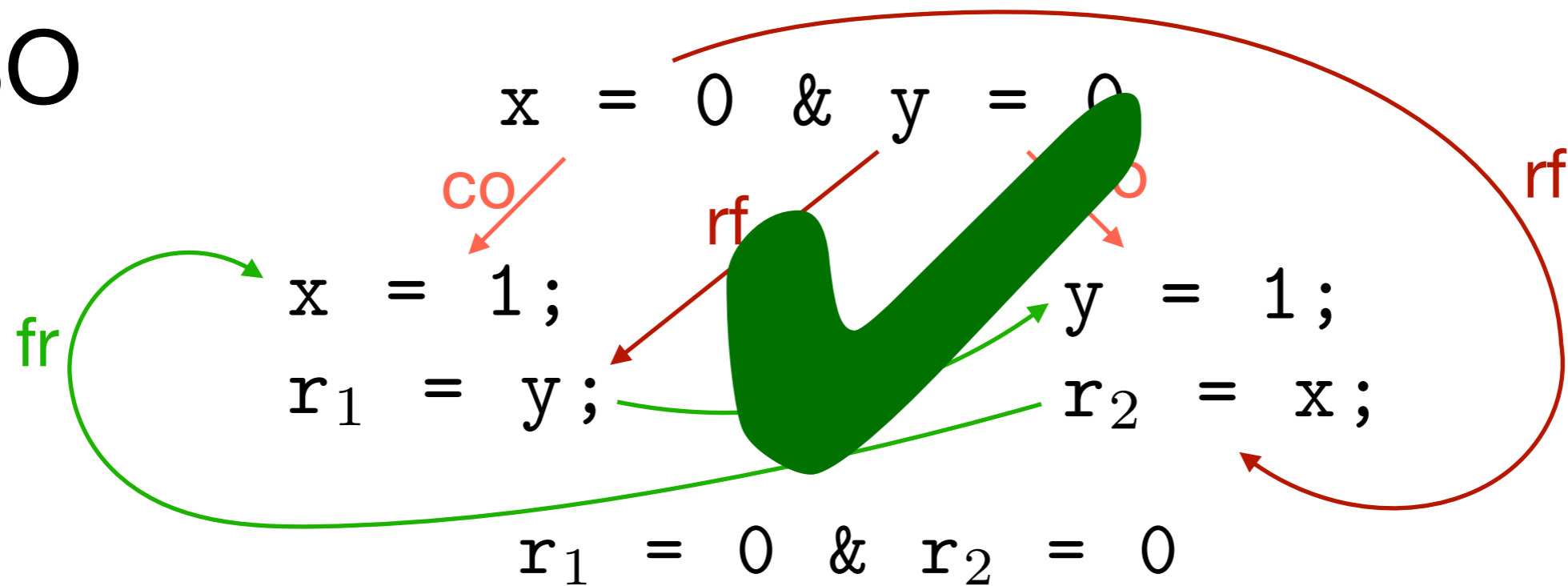


TSO

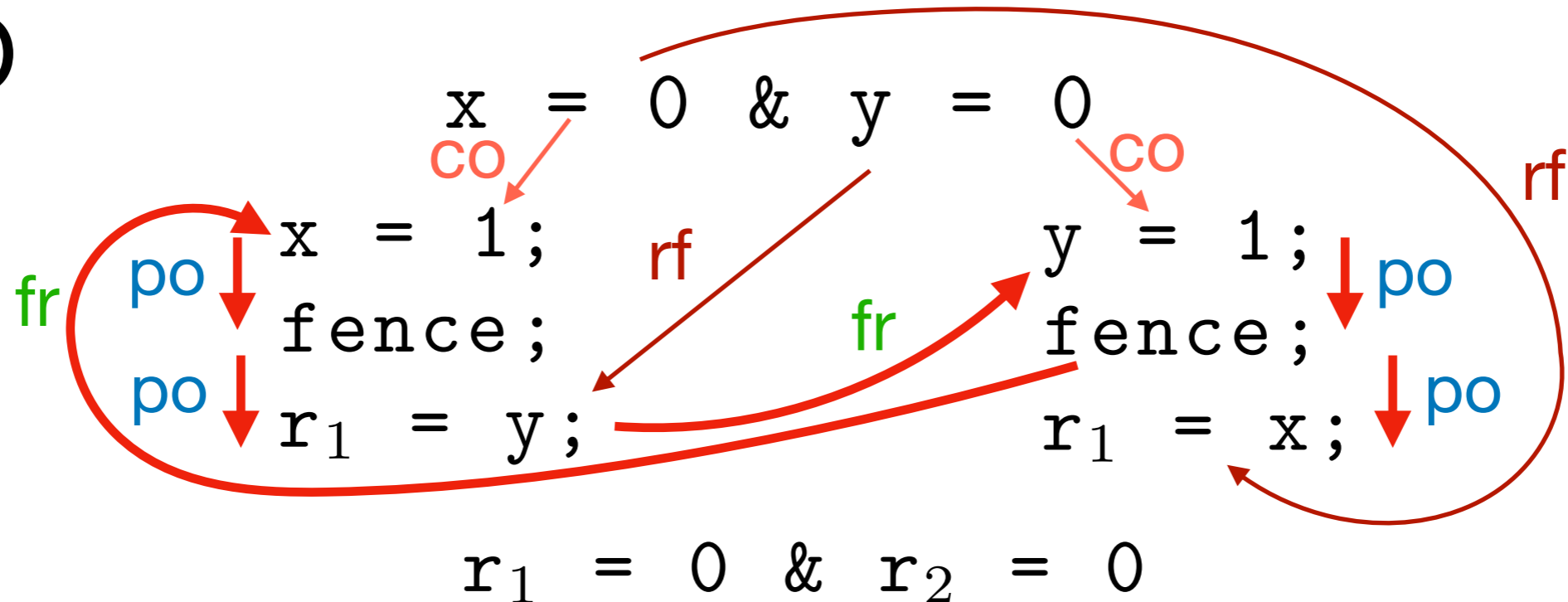


# TSO — The Model

TSO

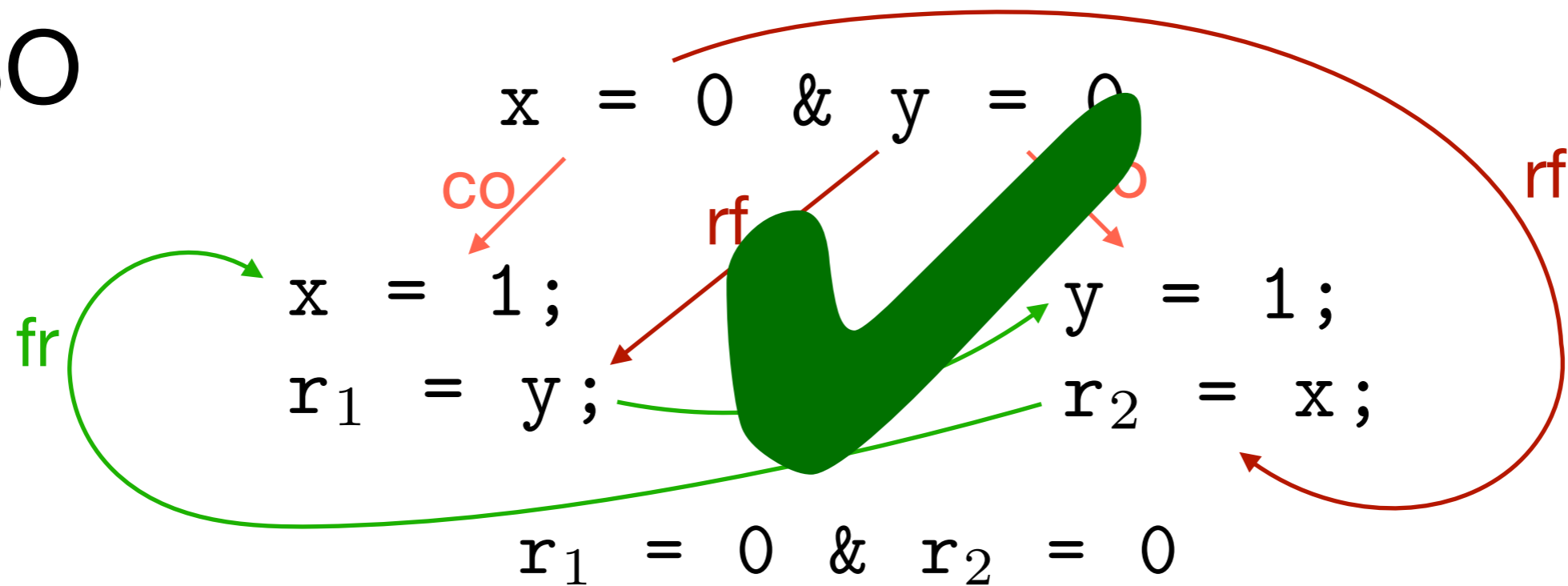


TSO

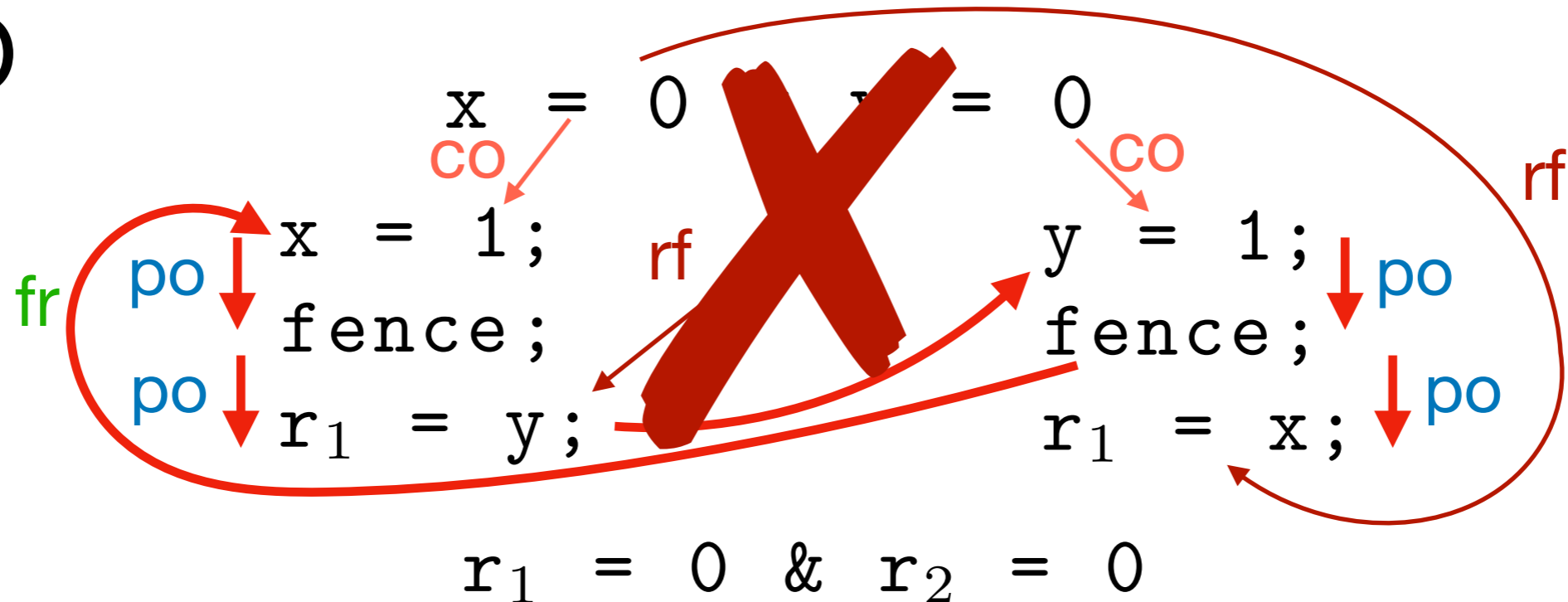


# TSO — The Model

TSO



TSO



# Formalizing MMs

Events :  $e = p:R[x]=1 \mid p:W[x]=1 \mid p:Fence$

Execution:  $E = \langle P, Ev, po \rangle$

Candidate Execution:  $C = \langle E, rf, co \rangle$

Memory Access Dec:  $D = W \mid R \mid M$

Derived Relations:  $R = DD$   
 $\mid ext \mid int \mid fr$

Constraints:  $acyclic \mid irreflexive$

# Causality (Happens-Before)

In the simplest case (SC):

# Causality (Happens-Before)

In the simplest case (SC):

- events from the same process happen in the order of their program:  $po \subseteq hb$

# Causality (Happens-Before)

In the simplest case (SC):

- events from the same process happen in the order of their program:  $po \subseteq hb$
- If a read sees a value, the write storing that value happens before that read:  $rf \subseteq hb$

# Causality (Happens-Before)

In the simplest case (SC):

- events from the same process happen in the order of their program:  $po \subseteq hb$
- If a read sees a value, the write storing that value happens before that read:  $rf \subseteq hb$
- happens before is a transitive relation:  $hb^* \subseteq hb$

# Causality (Happens-Before)

In the simplest case (SC):

- events from the same process happen in the order of their program:  $po \subseteq hb$
- If a read sees a value, the write storing that value happens before that read:  $rf \subseteq hb$
- happens before is a transitive relation:  $hb^* \subseteq hb$
- happens before is *acyclic*

# Causality (Happens-Before)

In the simplest case (SC):

- events from the same process happen in the order of their program:  $po \subseteq hb$
- If a read sees a value, the write storing that value happens before that read:  $rf \subseteq hb$
- happens before is a transitive relation:  $hb^* \subseteq hb$
- happens before is *acyclic*
- we can add fr and co to hb ( $rf \subseteq hb$  and  $cp \subseteq hb$ ) but it doesn't change anything

# Causality (Happens-Before)

TSO\*:

# Causality (Happens-Before)

TSO\*:

- Reads can bypass writes on the same processor

# Causality (Happens-Before)

TSO\*:

- Reads can bypass writes on the same processor
  - Define the *preserved* program order:  $ppo = po / WR$

# Causality (Happens-Before)

TSO\*:

- Reads can bypass writes on the same processor
  - Define the *preserved* program order:  $\text{ppo} = \text{po} / \text{WR}$
- events from the same process happen in their preserved program:  
 $\text{ppo} \subseteq \text{hb}$

\* not quite enough (we'll see why)

# Causality (Happens-Before)

TSO\*:

- Reads can bypass writes on the same processor
  - Define the *preserved* program order:  $ppo = po / WR$
- events from the same process happen in their preserved program:  $ppo \subseteq hb$
- If a read sees a value, the write storing that value happens before that read:  $rf \subseteq hb$
- fr and co are included in hb ( $rf \subseteq hb$  and  $cp \subseteq hb$ )
- happens before is a transitive relation:  $hb^* \subseteq hb$
- happens before is *acyclic*

\* not quite enough (we'll see why)

tso.cat

# tso.cat

- SB
- SB+rfi-pos
- SBB
- MP
- IRIW

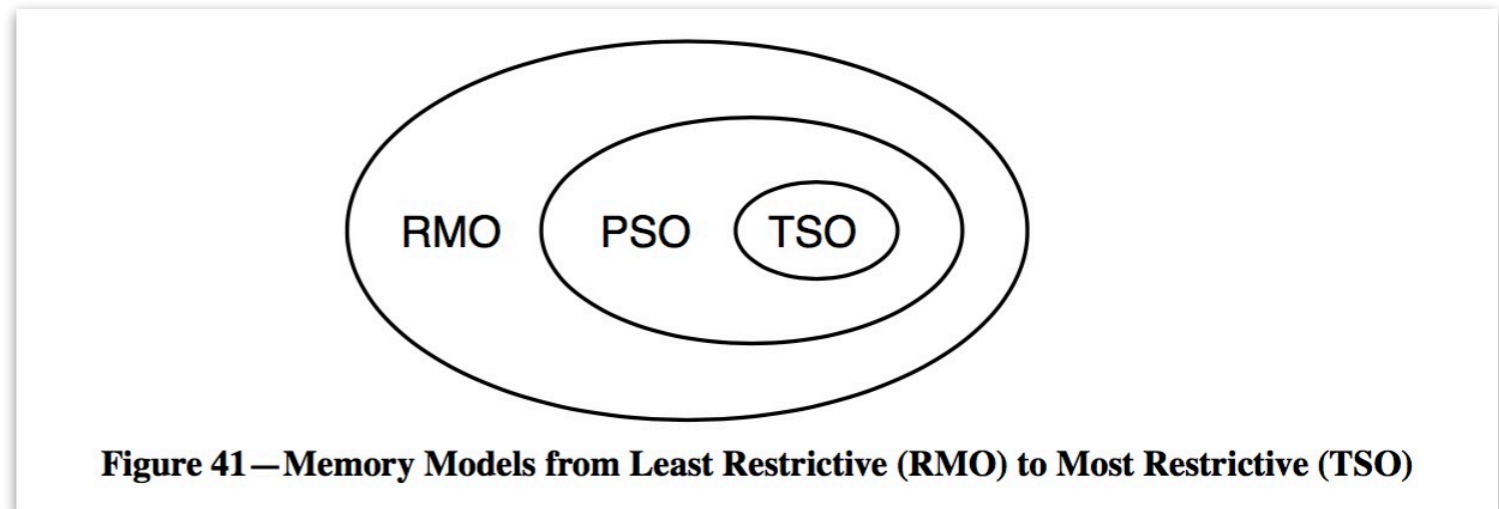
notation	name	nature	dirns	reference	description
po	program order	execution	any, any	§Relations over events	instruction order lifted to events
rf	read-from	execution	WR	§Relations over events	links a write $w$ to a read $r$ taking its value from $w$
co	coherence	execution	WW	§Relations over events	total order over writes to the same memory location
ppo	preserved program order	architecture	any, any	§Architectures	program order maintained by the architecture
ffence, ff	full fence	architecture	any, any	§Architectures	e.g. sync on Power, dmb and dsb on ARM
lwfence, lwf	lightweight fence	architecture	any, any	§Architectures	e.g. lwsync on Power
cfence	control fence	architecture	any, any	§Architectures	e.g. isync on Power, isb on ARM
fences	fences	architecture	any, any	§Architectures	architecture-dependent subset of the fence relations, e.g. ffence, lwfence, cfence
prop	propagation	architecture	WW	§Architectures	order in which writes propagate, typically enforced by fences
po-loc	program order restricted to the same memory location	derived	any, any	§SC PER LOCATION	$\{(x, y) \mid (x, y) \in \text{po} \wedge \text{addr}(x) = \text{addr}(y)\}$
com	communications	derived	any, any	§Relations over events	$\text{co} \cup \text{rf} \cup \text{fr}$
fr	from-read	derived	RW	§Relations over events	links a read $r$ to a write $w'$ co-after the write $w$ from which $r$ takes its value
hb	happens before	derived	any, any	§NO THIN AIR	$\text{ppo} \cup \text{fences} \cup \text{rfe}$
rdw	read different writes	derived	RR	Fig. 27	two threads; first thread holds a write, second thread holds two reads

notation	name	nature	dirns	reference	description
po	program order	execution	any, any	§Relations over events	instruction order lifted to events
rf	read-from	execution	WR	§Relations over events	links a write $w$ to a read $r$ taking its value from $w$
co	coherence	execution	WW	§Relations over events	total order over writes to the same memory location
ppo	preserved program order	architecture	any, any	§Architectures	program order maintained by the architecture
ffence, ff	full fence	architecture	any, any	§Architectures	e.g. sync on Power, dmb and dsb on ARM
lwfence, lwf	lightweight fence	architecture	any, any	§Architectures	e.g. lwsync on Power
cfence	control fence	architecture	any, any	§Architectures	e.g. isync on Power, isb on ARM
fences	fences	architecture	any, any	§Architectures	architecture-dependent
prop	propagation	architecture			propagate, typically enforced by fences
po-loc	program order restricted to the same memory location	derived	any, any	§SC PER LOCATION	$\{(x, y) \mid (x, y) \in \text{po} \wedge \text{addr}(x) = \text{addr}(y)\}$
com	communications	derived	any, any	§Relations over events	$\text{co} \cup \text{rf} \cup \text{fr}$
fr	from-read	derived	RW	§Relations over events	links a read $r$ to a write $w'$ co-after the write $w$ from which $r$ takes its value
hb	happens before	derived	any, any	§NO THIN AIR	$\text{ppo} \cup \text{fences} \cup \text{rfe}$
rdw	read different writes	derived	RR	Fig. 27	two threads; first thread holds a write, second thread holds two reads

Relax: we won't study all of these!

# PSO - RMO

- Herd
- Other semantical styles



- TSO:
  - Denotational semantics based on sequences
  - Denotational semantics based on POSETs
- PSO & RMO:
  - Axiomatic and operational models are relatively simple
  - Denotational? Not so much

# A Better x86 Memory Model: x86-TSO

Scott Owens    Susmit Sarkar    Peter Sewell

University of Cambridge  
<http://www.cl.cam.ac.uk/users/pes20/weakmemory>

**Abstract.** Real multiprocessors do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, typically described in ambiguous prose, which lead to widespread confusion. These are prime targets for mechanized formalization. In previous work we produced a rigorous *x86-CC* model, formalizing the Intel and AMD architecture specifications of the time, but those turned out to be unsound with respect to actual hardware, as well as arguably too weak to program above. We discuss these issues and present a new *x86-TSO* model that suffers from neither problem, formalized in HOL4. We believe it is sound with respect to real processors, reflects better the vendor's intentions, and is also better suited for programming. We give two equivalent definitions of *x86-TSO*: an intuitive operational model based on local write buffers, and an axiomatic total store ordering model, similar to that of the SPARCv8. Both are adapted to handle x86-specific features. We have implemented the axiomatic model in our `memevents` tool, which calculates the set of all valid executions of test programs, and, for greater confidence, verify the witnesses of such executions directly, with code extracted from a third, more algorithmic, equivalent version of the definition.

## 1 Introduction

Most previous research on the semantics and verification of concurrent programs assumes sequential consistency: that accesses by multiple threads to a shared resource occur in a global-time linear order. Real multiprocessors, however, incur optimisations. These are typically unobservable by programmers, but have observable consequences for the behaviour of programs. AMD x86 proces-

# x86 is TSO

- Documentations are really imprecise
- So you say x86 is TSO ..., how do you know?
  - Litmus Test
  - No conclusive proof
- Errors in both the specification and implementations have been found (mostly ARM/Power)

PowerPC / ARM

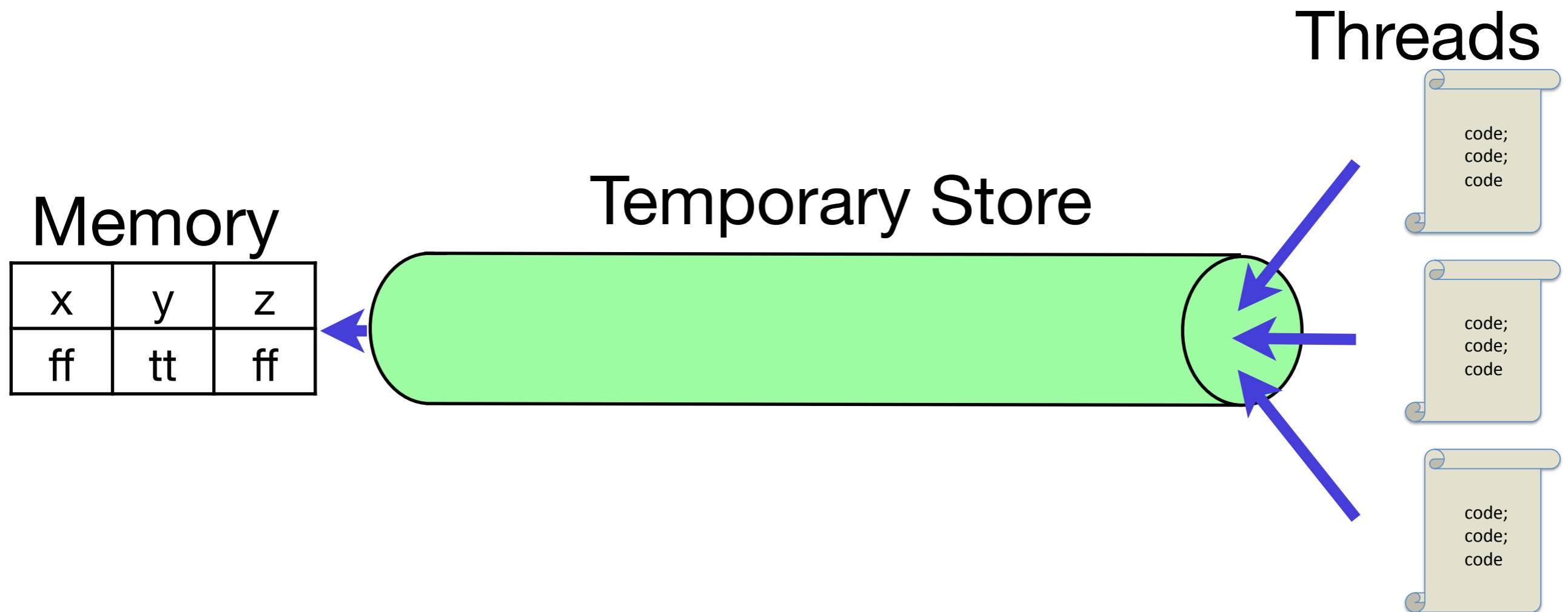
# Power? ARM

- The story is more complicated
- Operationally: Store Atomicity relaxations (co)
- Axiomatically: Many more axioms
- How do we restore assurance?
  - Herd/CAT
  - Operational Simulators
  - Still ..., no guarantees

PCCMEM

# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store



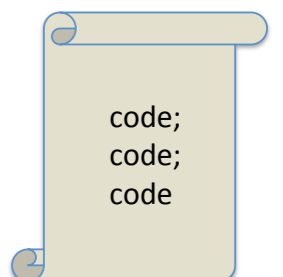
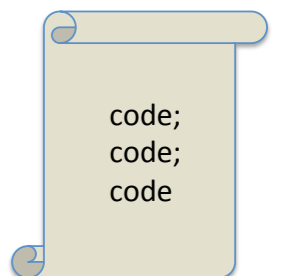
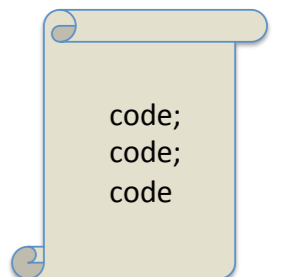
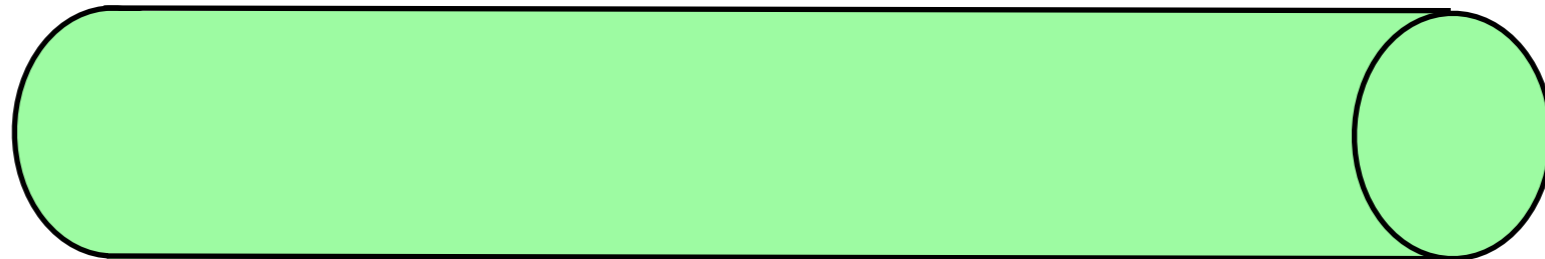
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

Temporary Store

x	y	z
ff	tt	ff



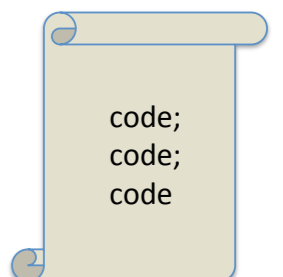
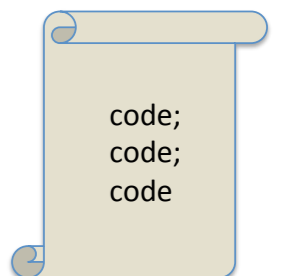
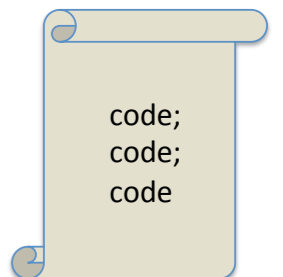
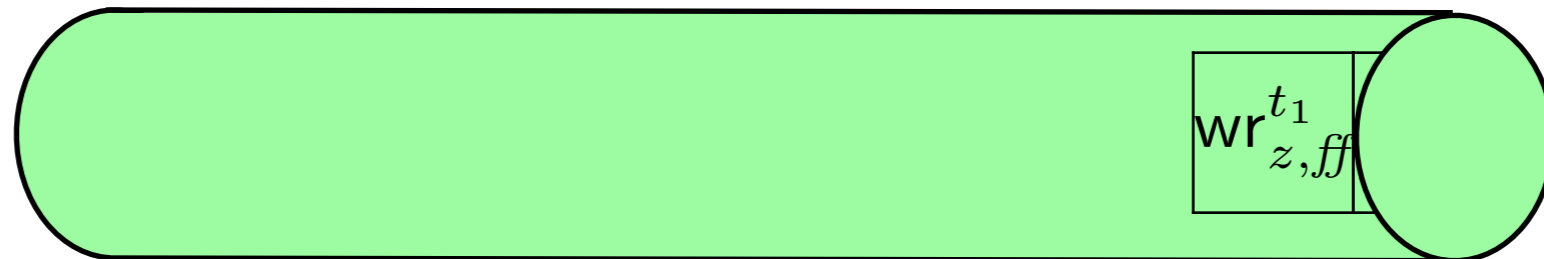
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

Temporary Store

x	y	z
ff	tt	ff



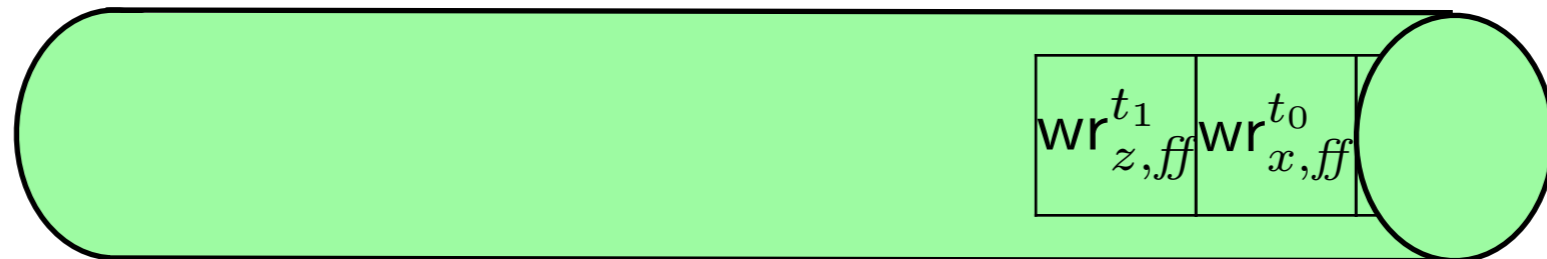
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



code;  
code;  
code

code;  
code;  
code

code;  
code;  
code

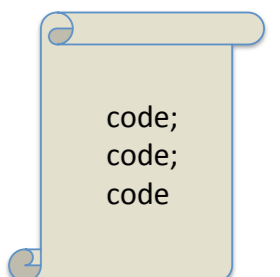
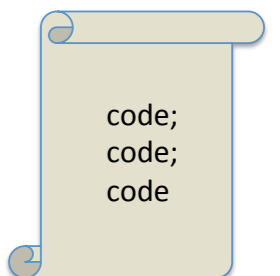
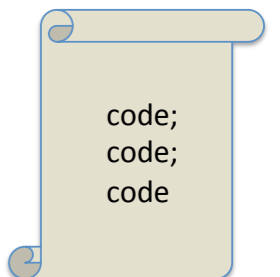
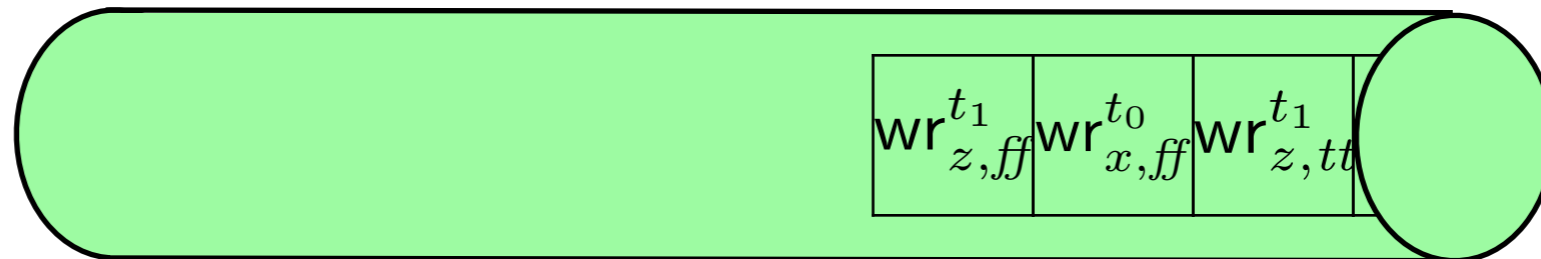
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



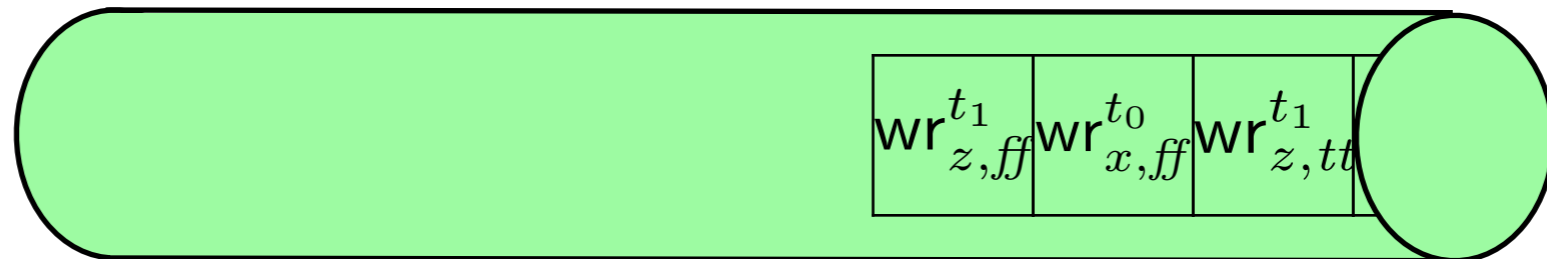
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

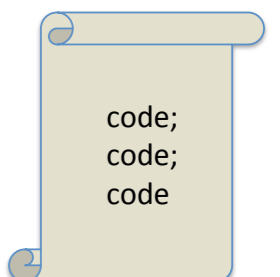
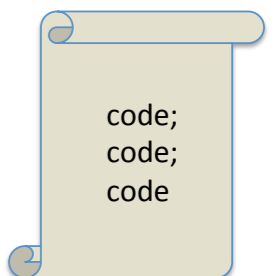
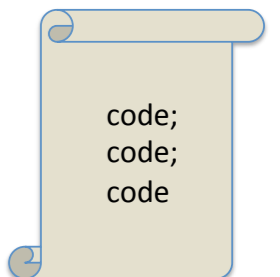
$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

Temporary Store

x	y	z
ff	tt	ff



Placeholder values for reads



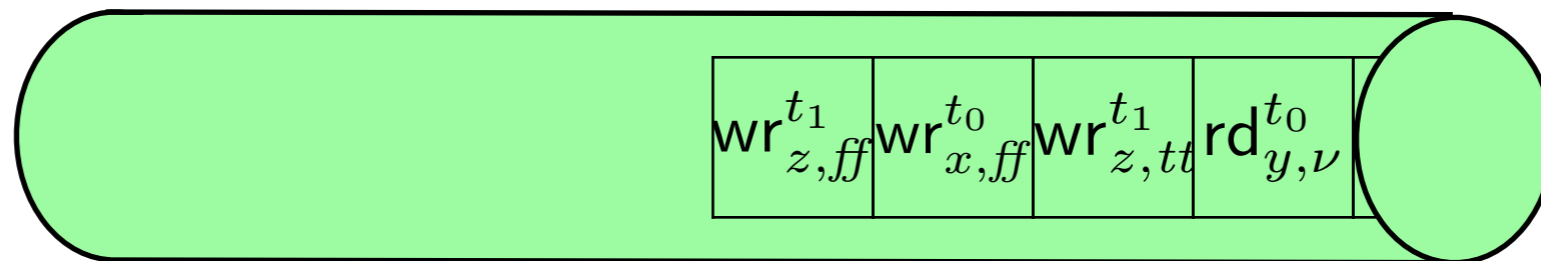
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

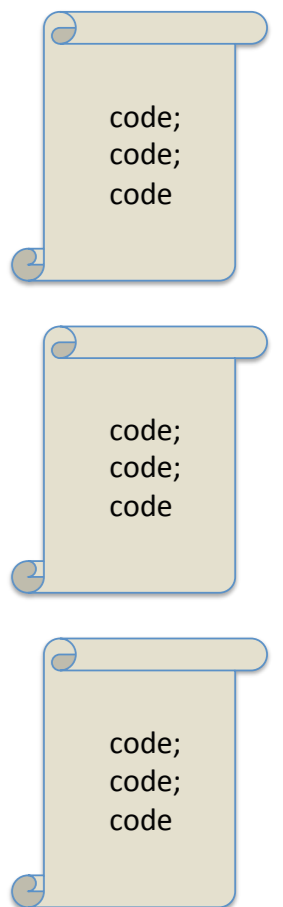
$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



Placeholder values for reads



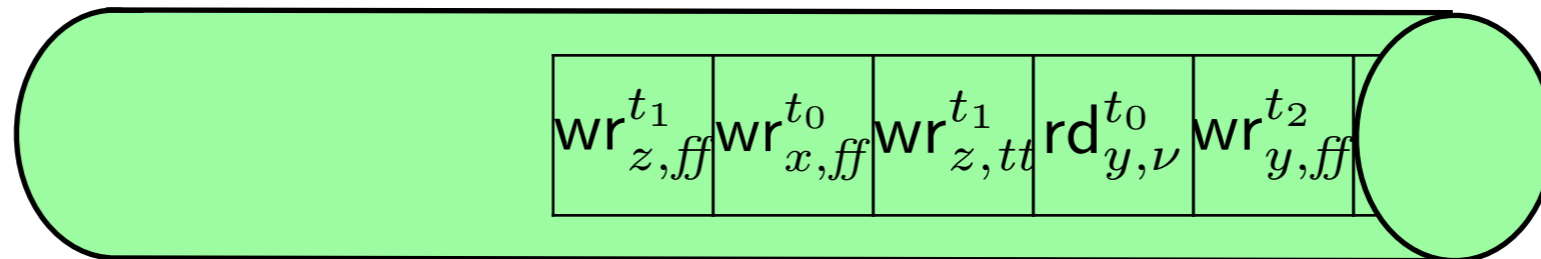
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



Placeholder values for reads

code;  
code;  
code

code;  
code;  
code

code;  
code;  
code

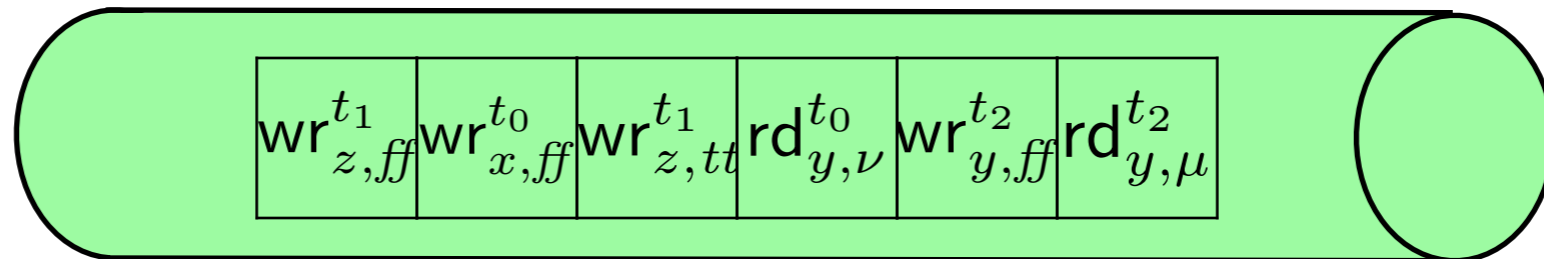
# The Semantic Framework in a Nutshell

Threads contribute operations to a pipeline-like temporary store

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



Placeholder values for reads

code;  
code;  
code

code;  
code;  
code

code;  
code;  
code

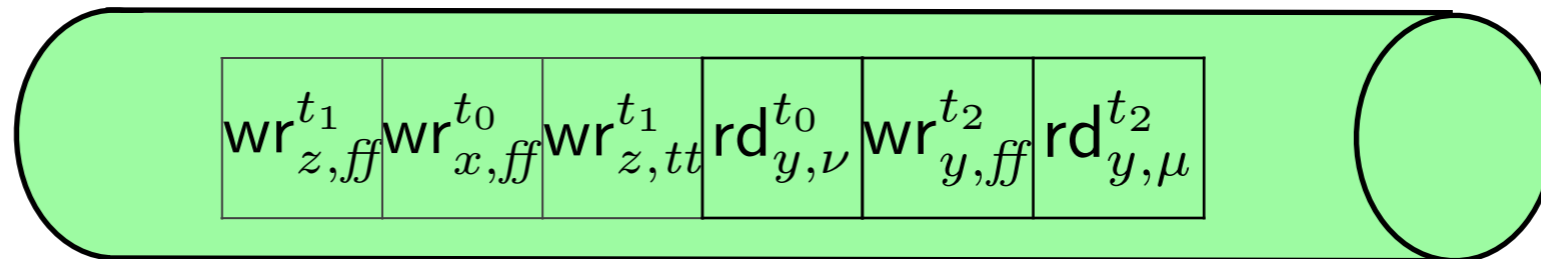
# The Semantic Framework in a Nutshell

The Memory Executes Operations  
Reordering as Allowed by the Memory Model

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



code;  
code;  
code

code;  
code;  
code

code;  
code;  
code

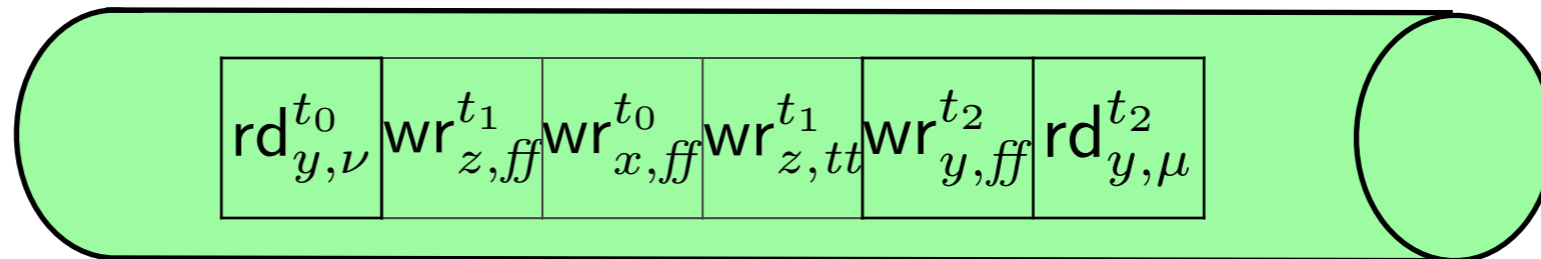
# The Semantic Framework in a Nutshell

The Memory Executes Operations  
Reordering as Allowed by the Memory Model

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



code;  
code;  
code

code;  
code;  
code

code;  
code;  
code

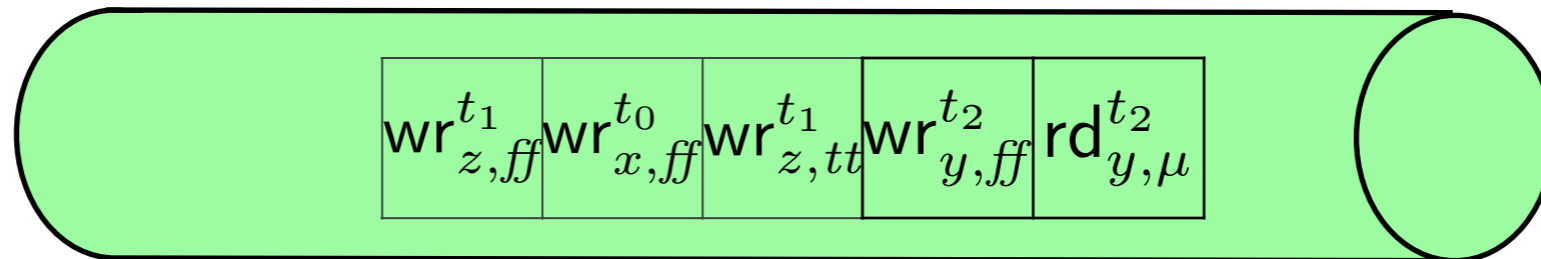
# The Semantic Framework in a Nutshell

The Memory Executes Operations  
Reordering as Allowed by the Memory Model

$$\begin{array}{l} x := ff; \\ r_0 := (!y) \end{array} \parallel \begin{array}{l} z := ff; \\ z := tt \end{array} \parallel \begin{array}{l} y := ff; \\ r_1 := (!y) \end{array}$$

## Temporary Store

x	y	z
ff	tt	ff



code;  
code;  
code

code;  
code;  
code

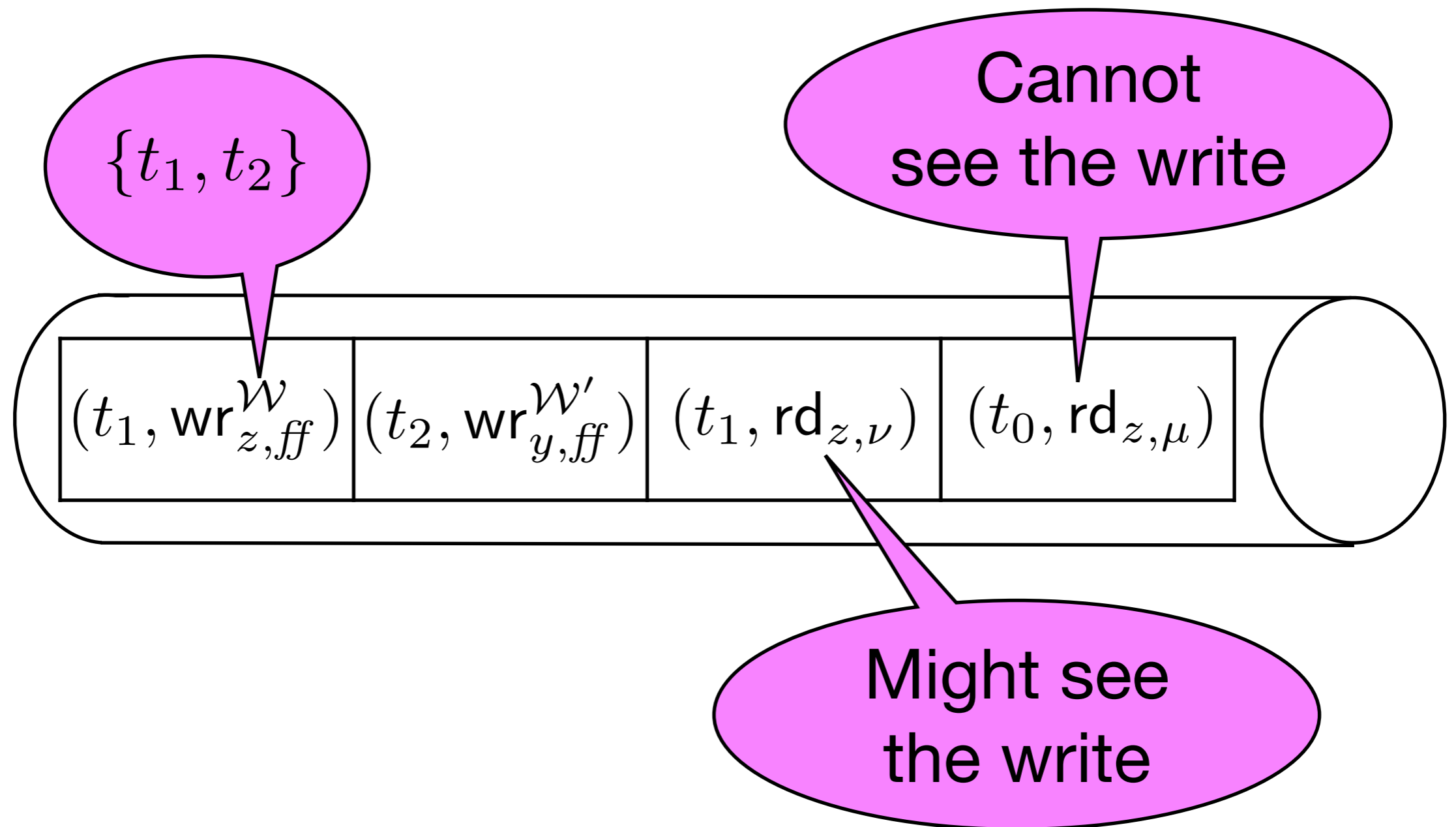
code;  
code;  
code

# Examples

- We use a commutability relation constraining the permissible reorderings

TSO	$(t, wr_{p,v}^W) \hookrightarrow (t, rd_{q,w})$
PSO	$(t, wr_{p,v}^W) \hookrightarrow (t, rd_{q,w}) \ \& \ (t, wr_{p,v}^W) \hookrightarrow (t, wr_{q,w})$
RMO	$(t, wr_{p,v}^W) \hookrightarrow (t, rd_{q,w}) \ \& \ (t, wr_{p,v}^W) \hookrightarrow (t, wr_{q,w})$ $(t, rd_{p,v}) \hookrightarrow (t, rd_{q,w})$

# Store-Atomicity Relaxation



# Memory Write Rules

## Normal Write

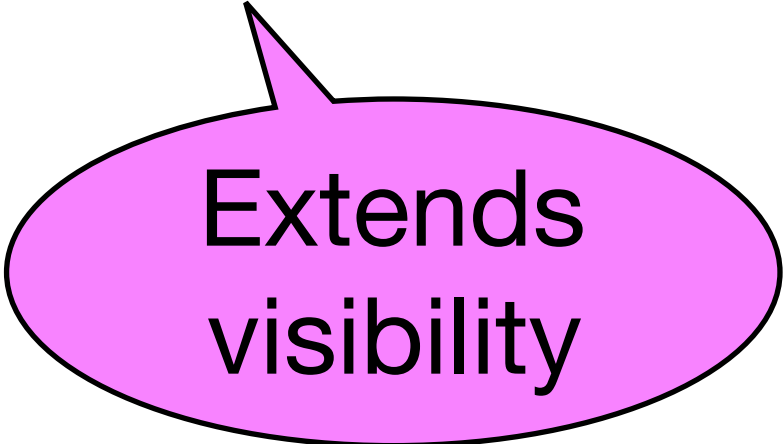
$$(S, \sigma_0 \cdot (t, \text{wr}_{p,v}^{W,I}) \cdot \sigma_1, T) \xrightarrow{\hookrightarrow, \mathcal{W}} (S[p := v], \sigma_0 \cdot \sigma_1, T)$$

if  $\sigma_0 \hookrightarrow (t, \text{wr}_{p,v}^{W,I}) \ \& \ v \in \mathcal{Val}$

## Early Write

$$(S, \sigma_0 \cdot (t, \text{wr}_{\varrho,v}^{W,I}) \cdot \sigma_1, T) \xrightarrow{\hookrightarrow, \mathcal{W}} (S, \sigma_0 \cdot (t, \text{wr}_{\varrho,v}^{W',I}) \cdot \sigma_1, T)$$

if  $t \in W' \ \& \ W \subset W' \in \mathcal{W}$



Extends  
visibility

# Memory Read Rules

## Normal Read

$$\begin{aligned} (S, \sigma_0 \cdot (t, \text{rd}_{p,\iota}) \cdot \sigma_1, T) &\xrightarrow{\hookrightarrow, \mathcal{W}} (S, \{\iota \mapsto v\}(\sigma_0 \cdot \sigma_1, T)) \\ &\text{if } \sigma_0 \hookrightarrow (t, \text{rd}_{p,\iota}) \ \& \ S(p) = v \end{aligned}$$

## Early Read

$$\begin{aligned} (S, \sigma_0 \cdot (t', \text{wr}_{p,v}^{W,I}) \cdot \sigma_1 \cdot (t, \text{rd}_{p,\iota}) \cdot \sigma_2, T) \\ &\xrightarrow{\hookrightarrow, \mathcal{W}} (S, \{\iota \mapsto v\}(\sigma_0 \cdot (t', \text{wr}_{p,v}^{W, I \cup \{\iota\}}) \cdot \sigma_1 \cdot \sigma_2, T)) \\ &\text{if } t \in W \ \& \ \sigma_1 \hookrightarrow (t, \text{rd}_{p,\iota}) \end{aligned}$$

# IRIW Example

IRIW  $p := tt \parallel q := tt \parallel \begin{matrix} r_0 := !p; \\ r_1 := !q \end{matrix} \parallel \begin{matrix} r_2 := !q; \\ r_3 := !p \end{matrix}$

$r_0 = r_2 = tt \ \& \ r_1 = r_3 = ff$

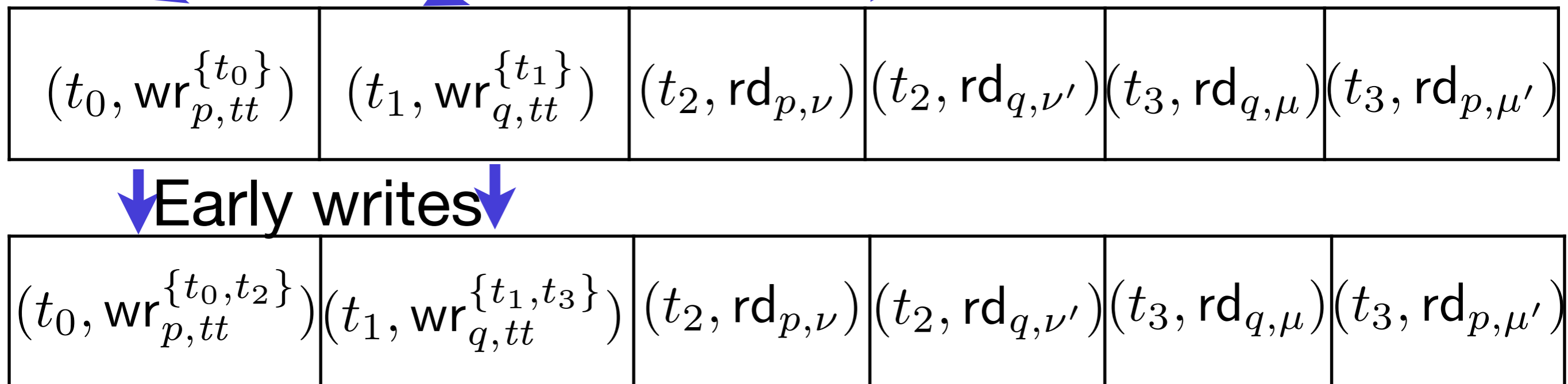


$(t_0, wr_{p,tt}^{\{t_0\}})$	$(t_1, wr_{q,tt}^{\{t_1\}})$	$(t_2, rd_{p,\nu})$	$(t_2, rd_{q,\nu'})$	$(t_3, rd_{q,\mu})$	$(t_3, rd_{p,\mu'})$
------------------------------	------------------------------	---------------------	----------------------	---------------------	----------------------

# IRIW Example

IRIW  $p := tt \parallel q := tt \parallel \begin{matrix} r_0 := !p; \\ r_1 := !q \end{matrix} \parallel \begin{matrix} r_2 := !q; \\ r_3 := !p \end{matrix}$

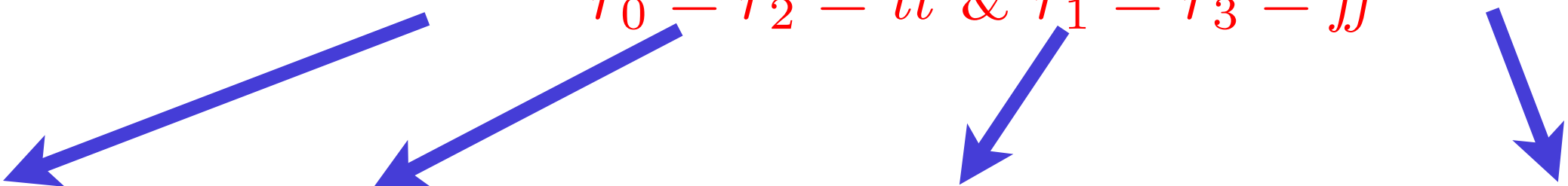
$r_0 = r_2 = tt \ \& \ r_1 = r_3 = ff$



# IRIW Example

IRIW  $p := tt \parallel q := tt \parallel \begin{matrix} r_0 := !p; \\ r_1 := !q \end{matrix} \parallel \begin{matrix} r_2 := !q; \\ r_3 := !p \end{matrix}$

$r_0 = r_2 = tt \ \& \ r_1 = r_3 = ff$



$(t_0, wr_{p,tt}^{\{t_0\}})$	$(t_1, wr_{q,tt}^{\{t_1\}})$	$(t_2, rd_{p,\nu})$	$(t_2, rd_{q,\nu'})$	$(t_3, rd_{q,\mu})$	$(t_3, rd_{p,\mu'})$
------------------------------	------------------------------	---------------------	----------------------	---------------------	----------------------

↓ Early writes ↓

$(t_0, wr_{p,tt}^{\{t_0,t_2\}})$	$(t_1, wr_{q,tt}^{\{t_1,t_3\}})$	$(t_2, rd_{p,\nu})$	$(t_2, rd_{q,\nu'})$	$(t_3, rd_{q,\mu})$	$(t_3, rd_{p,\mu'})$
----------------------------------	----------------------------------	---------------------	----------------------	---------------------	----------------------

↓ Early reads ↓

$(t_0, wr_{p,tt}^{\{t_0,t_2\}})$	$(t_1, wr_{q,tt}^{\{t_1,t_3\}})$		$(t_2, rd_{q,\nu'})$		$(t_3, rd_{p,\mu'})$
----------------------------------	----------------------------------	--	----------------------	--	----------------------

PCC.cat

# PCC.cat

- SB-PPC
- SB-PPC-lwsync
- SB-PPC-sync
- WRC
- WRC+realdata
- WRC-lwsync
- IRIW
- IRIW-lwsync
- IRIW-sync

# POWER and ARM Litmus Tests

<http://www.cl.cam.ac.uk/~pes20/ppc-supplemental>

Coherence tests			
<b>CoRR1: rf,po,fr</b> forbidden  Test CoRR1	<b>CoRW: rf,po,co</b> forbidden  Test CoRW	<b>CoWR: co,po,rf<sup>-1</sup></b> forbidden  Test CoWR	<b>CoWW: po,co</b> forbidden  Test CoWW
4-edge 2-thread tests		5-edge extensions along one rf edge	
<b>One rf</b> <b>MP: rf,fr</b> needs lwsync+RRdep  Test MP	<b>Two rf</b> <b>WRC: rf,rf,fr</b> needs lwsync+RRdep  Test WRC	<b>Preserved read-read program order</b> <b>PPO000-019: barrier,rf,intra-thread*,fr</b>  Test PPO000	
<b>S: rf,co</b> needs lwsync+RWdep  Test S	<b>WWC: rf,rf,co</b> needs lwsync+RWdep  Test WWC		
No rf	One rf	6-edge extensions along two rf edges	
<b>SB: fr,fr</b> needs sync+sync  Test SB	<b>RWC: rf,fr,fr</b> needs sync+sync  Test RWC	<b>IRIW: rf,fr,rf,fr</b> needs sync+sync  Test IRIW	
<b>R: co,fr</b> needs sync+sync	<b>WRW+WR: rf,co,fr</b> needs sync+sync	<b>IRRWIW: rf,fr,rf,co</b> needs sync+sync	

# Power Barriers

- sync: heavyweight barrier
  - cumulative
- lwsync: lightweight barrier
  - similar to sync
  - does not prevent WR reordering
- Examples
  - Herd

# Simple Spin-lock

```
lock(l);      ||      lock(l);  
    r = x;      r = x;  
    x = r+1;    x = r+1;  
unlock(l);    unlock(l);
```

```
lock (l) {  
    while (!cas(lock, 0, 1)) {  
        while (lock == 0);  
    }  
}  
  
unlock (l) {  
    l = 0  
}
```

To fence or not to fence?

# Other Models

- And yet these are not the most complicated models
  - NVIDIA
  - Alpha (obsolete)
  - We'll see Programming Languages models in the next lecture