

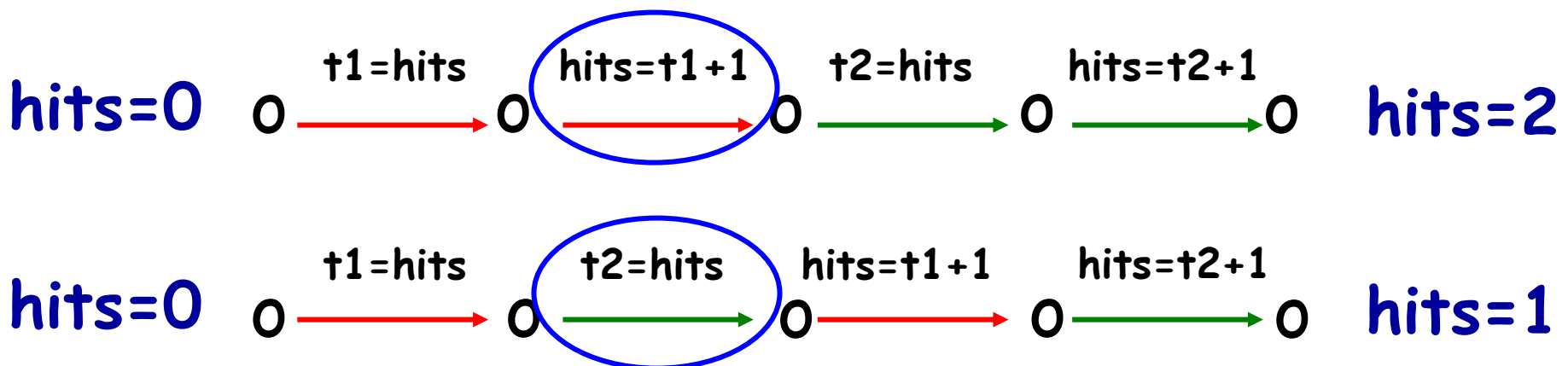
# Lecture 5a

## Checking Atomicity:

### A Comparison of Static and Dynamic Techniques

# Race Conditions

A *race condition* occurs if two threads access a shared variable at the same time, and at least one of the accesses is a write



# Bank Account Implementation

```
class Account {  
    private int balance = 0;
```

```
    public read() {  
        int r;  
        synchronized(this) {  
            r = balance;  
        }  
        return r;  
    }  
}
```

```
    public void deposit(int n) {  
        int r = read();  
        other threads can update balance  
        synchronized(this) {  
            balance = r + n;  
        }  
    }  
}
```

```
}
```

- Race-freedom is not sufficient!

# Fixed Bank Account

```
class Account {  
    private int balance = 0;
```

```
    public read() {  
        int r;  
        synchronized(this) {  
            r = balance;  
        }  
        return r;  
    }  
}
```

```
    public void deposit(int n) {  
        synchronized(this) {  
            int r = balance;  
            balance = r + n;  
        }  
    }  
}
```

# Optimized Bank Account

```
class Account {  
    private int balance = 0;  
  
    public read() {  
        return balance;  
    }  
  
    public void deposit(int n) {  
        synchronized(this) {  
            int r = balance;  
            balance = r + n;  
        }  
    }  
}
```

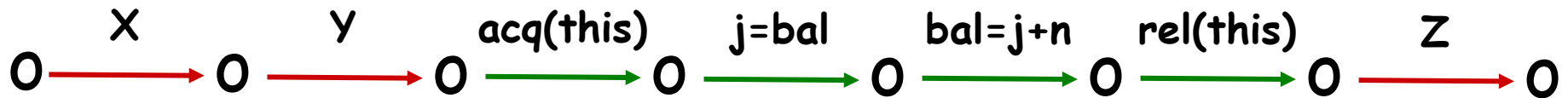
- Race-freedom is not necessary!

# Atomicity

- A method is **atomic** if concurrent threads do not interfere with its behavior
- Maximal non-interference property
- Enables sequential reasoning
- Matches existing programming methodology

# Atomicity

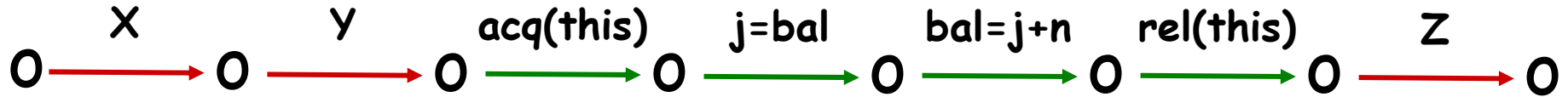
- Serialized execution of deposit



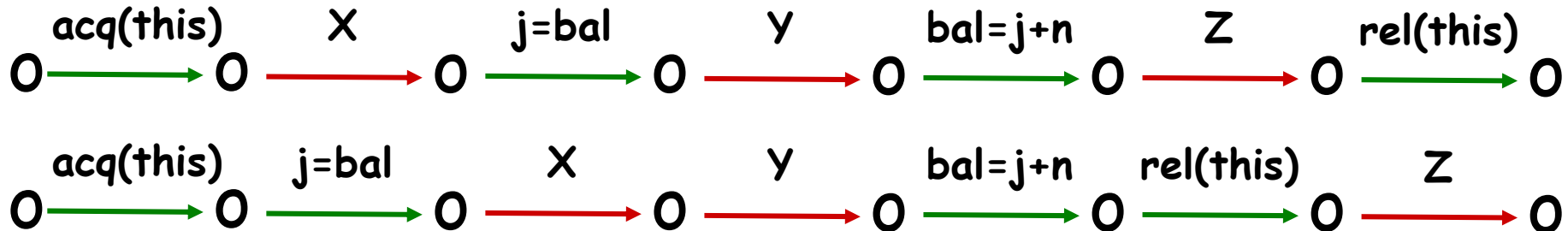
```
class Account {  
    int bal;  
    void deposit(int n) {  
        synchronized (this) {  
            int j = bal;  
            bal = j + n;  
        }  
    }  
}
```

# Atomicity

- Serialized execution of deposit



- Non-serialized executions of deposit



- deposit is *atomic* if, for every non-serialized execution, there is a serialized execution with the same overall behavior

# Part I: A Type and Effect System for Atomicity

# Reduction [Lipton 75]



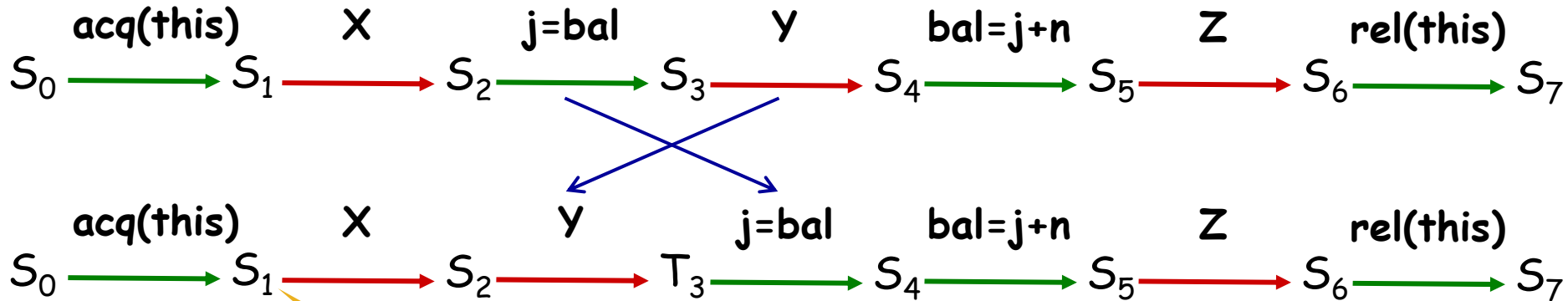
green thread holds lock

⇒ red thread does not hold lock

⇒ operation y does not access balance

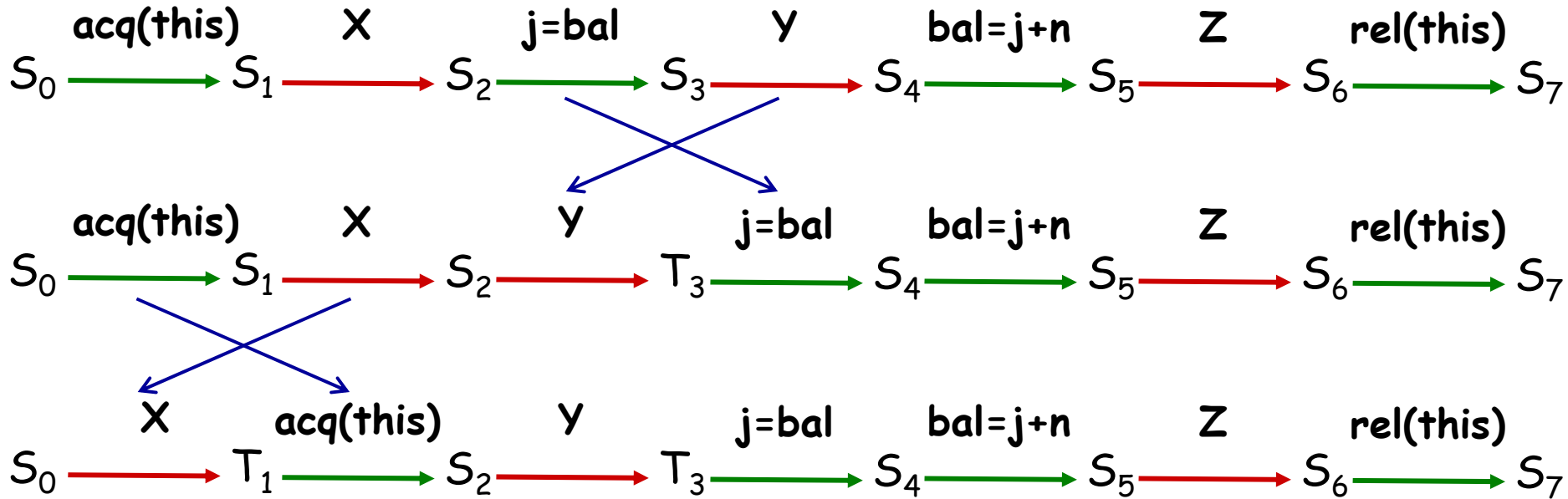
⇒ operations commute

# Reduction [Lipton 75]

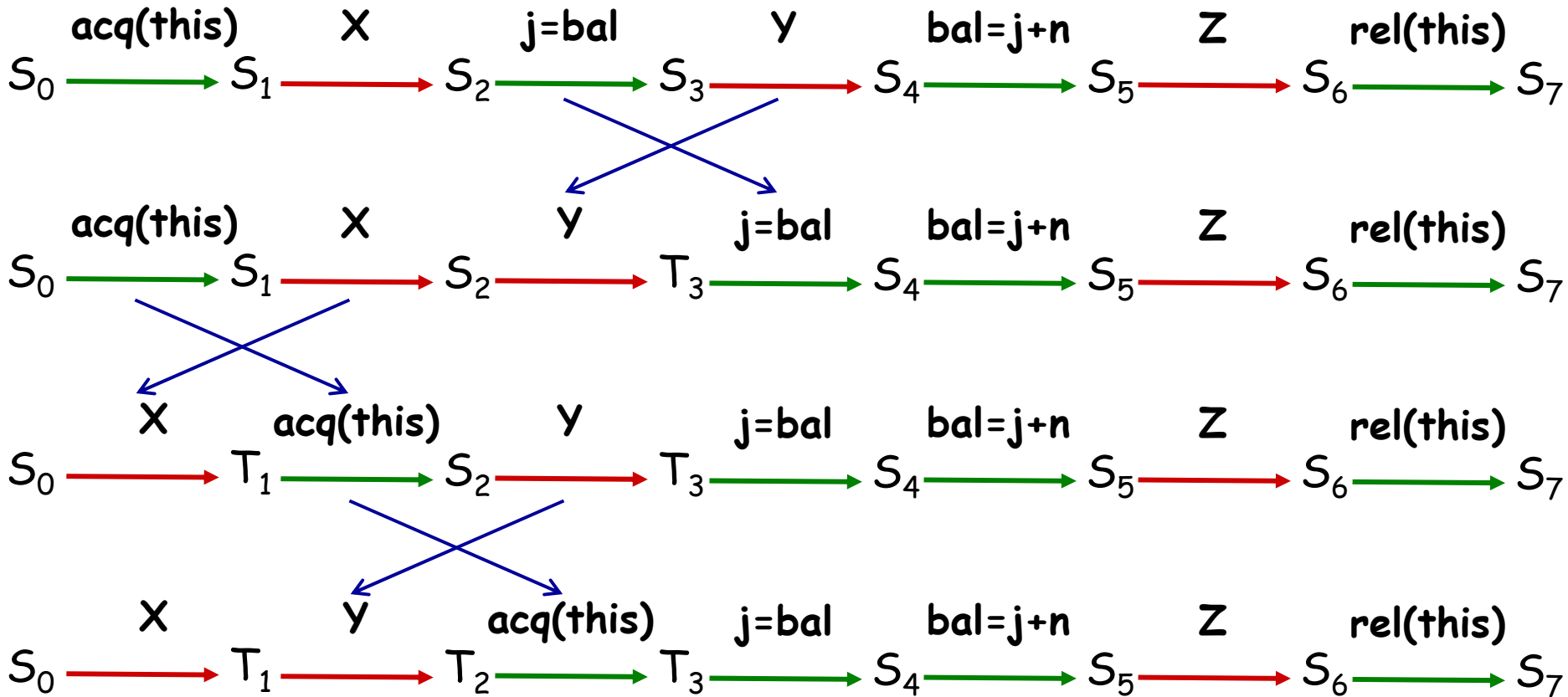


green thread holds lock after acquire  
 $\Rightarrow$  operation  $x$  does not modify lock  
 $\Rightarrow$  operations commute

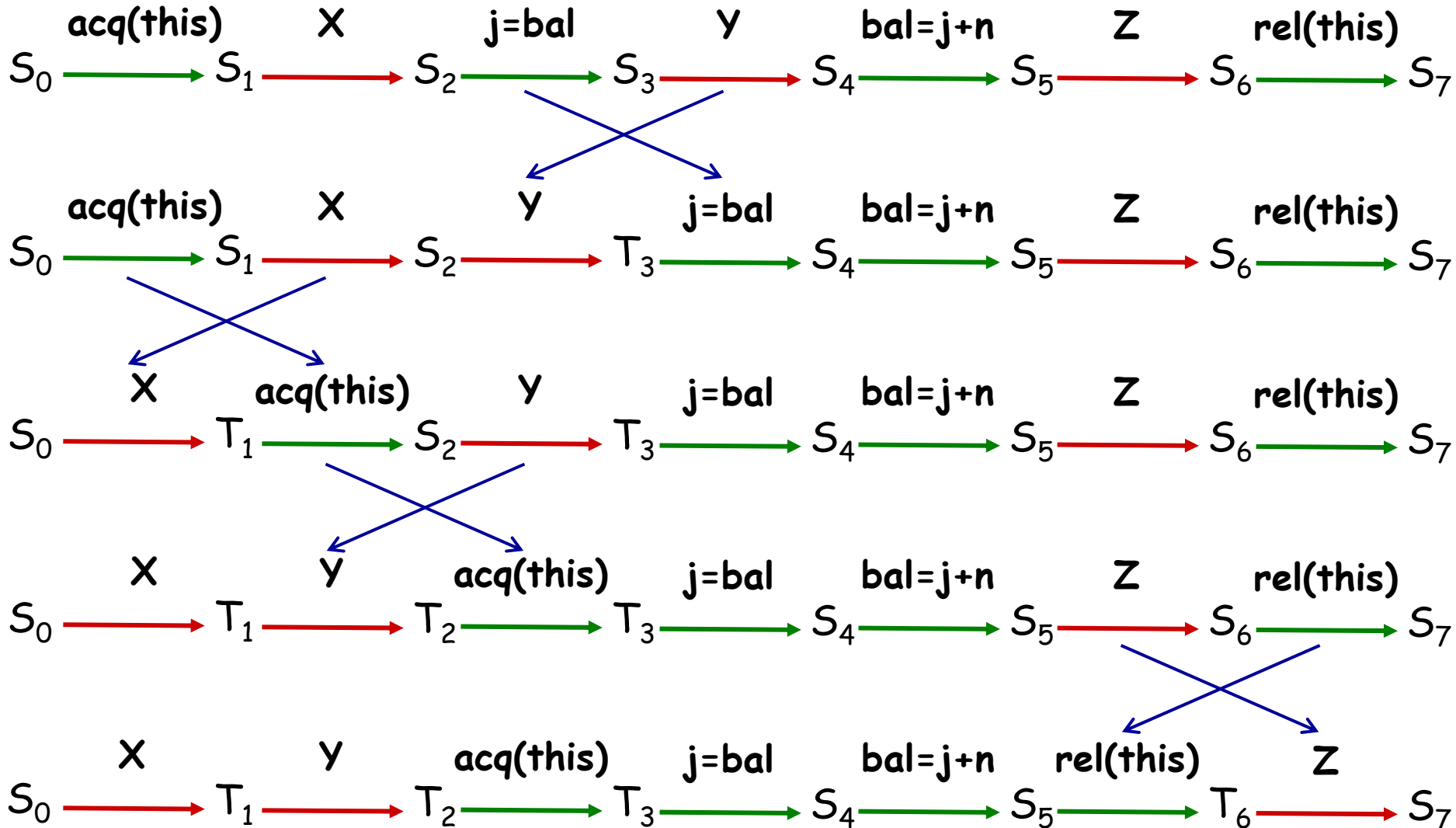
# Reduction [Lipton 75]



# Reduction [Lipton 75]



# Reduction [Lipton 75]

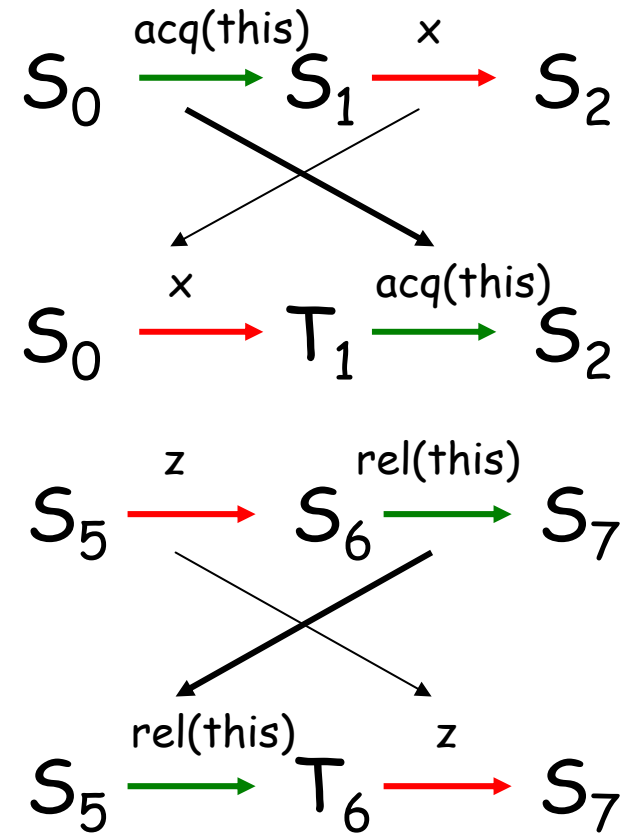


# Type System for Atomicity

- Assign to each statement an *atomicity* that characterizes the statement's behavior
  - five atomicities: **R, L, B, A, C**
  - does the statement left or right commute with steps of other threads?
  - is the statement atomic?
- Leverage Race Condition Checker to check that protecting lock is held when variables accessed

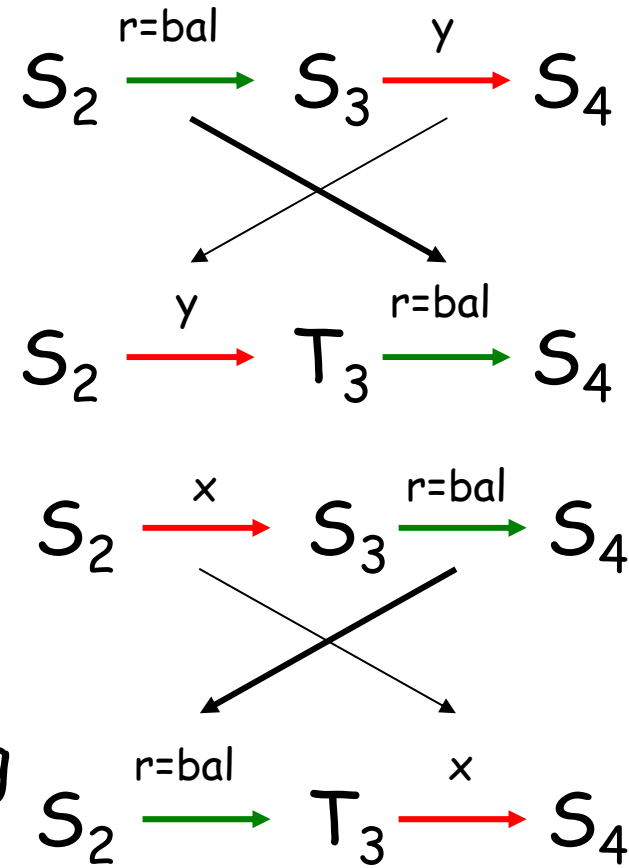
# Five Atomicities

- **R**: right commutes
  - lock acquire
- **L**: left commutes
  - lock release



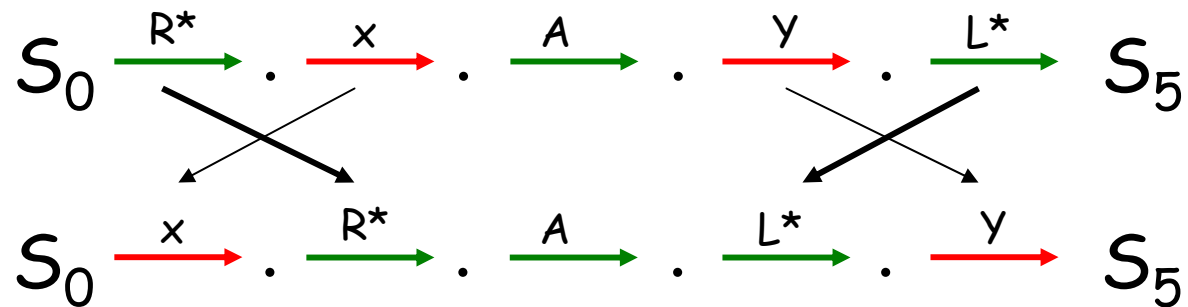
# Five Atomicities

- **R**: right commutes
  - lock acquire
- **L**: left commutes
  - lock release
- **B**: both right + left commutes
  - variable access holding lock
- **A**: atomic action, non-commuting
  - access unprotected variable
- **C**: compound, non-atomic statement
- atomicities for constants, conditional atomicities



# Sequentially Composing Atomicities

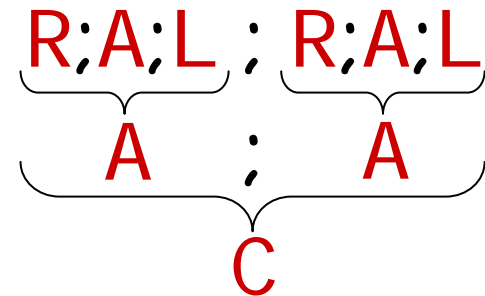
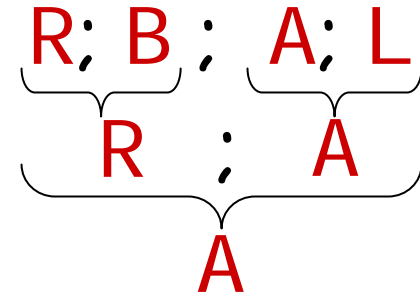
- Use atomicities to perform reduction
- Lipton: any sequence  $R^*;A;L^*$  is atomic



# Sequentially Composing Atomicities

- Use atomicities to perform reduction
- Lipton: any sequence  $R^*;A;L^*$  is atomic

;	B	L	R	A	C
B	B	L	R	A	C
R	R	A	R	A	C
L	L	L	C	C	C
A	A	A	C	C	C
C	C	C	C	C	C



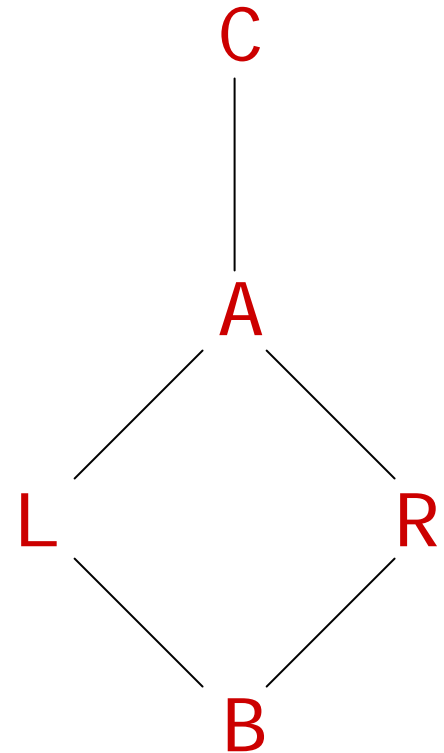
# Atomicities for Conditionals

If  $E$  has atomicity  $ae$   
and  $S1$  has atomicity  $a1$   
and  $S2$  has atomicity  $a2$  then

$\text{if } (E) \text{ } S1 \text{ else } S2$

has atomicity

$ae ; (a1 \sqcup a2)$



# Checking the Alternative Bank Account

```
class Account {  
    private int balance = 0 guarded_by this;
```

```
    public read() {
```

```
        B int r;  
        R synchronized(this) {  
            B r = balance;  
            L }  
        B return r;  
    }
```

```
    public void deposit(int n) {
```

```
        A int r = read();  
        R synchronized(this) {  
            B balance = r + n;  
            L }  
    }
```

- read is atomic
- deposit is compound, *not* atomic

# Checking the Fixed Bank Account

```
class Account {  
    private int balance = 0 guarded_by this;
```

```
    public read() {  
        A {  
            B int r;  
            R synchronized(this) {  
                B r = balance;  
                L }  
            B return r;  
        }  
    }  
}
```

```
    public void deposit(int n) {  
        A {  
            R synchronized(this) {  
                B int r = balance;  
                B balance = r + n;  
                L }  
        }  
    }
```

- fixed deposit *is* atomic

# Experience with Atomicity Checker

Class	Size (lines)	Annotations per KLOC					
		total	guard	req.	atomic	array	esc.
Inflater	296	20	17	0	3	0	0
Deflater	364	25	20	0	5	0	0
PrintWriter	557	36	5	0	25	0	5
Vector	1029	14	3	1	4	3	3
URL	1269	33	10	1	10	0	13
StringBuffer	1272	19	2	4	5	7	1
String	2399	22	0	0	1	19	1
<b>Total/average</b>	<b>7366</b>	<b>24</b>	<b>8</b>	<b>1</b>	<b>8</b>	<b>4</b>	<b>3</b>

# java.lang.StringBuffer

```
/**
```

```
... used by the compiler to implement the binary  
string concatenation operator ...
```

```
String buffers are safe for use by multiple  
threads. The methods are synchronized so that  
all the operations on any particular instance  
behave as if they occur in some serial order  
that is consistent with the order of the method  
calls made by each of the individual threads  
involved.
```

```
*/
```

```
public atomic class StringBuffer { ... }
```

# java.lang.StringBuffer is *not* Atomic!

```
public atomic StringBuffer {  
    private int count guarded_by this;  
A public synchronized int length() { return count; }  
A public synchronized void getChars(...) { ... }
```

```
    public synchronized void append(StringBuffer sb) {
```

```
        A int len = sb.length();  
        ...  
        ...  
        A sb.getChars(..., len, ...);  
        ...  
    }  
}
```

sb.length() acquires the lock on sb, gets the length, and releases lock

other threads can change sb

use of stale len may yield StringIndexOutOfBoundsException inside getChars(...)

• `append(...)` is *not* atomic