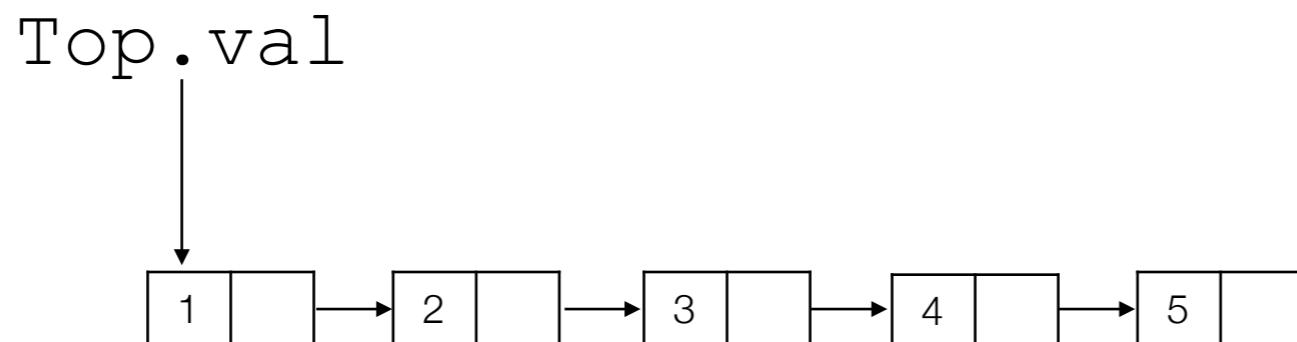


... continuation

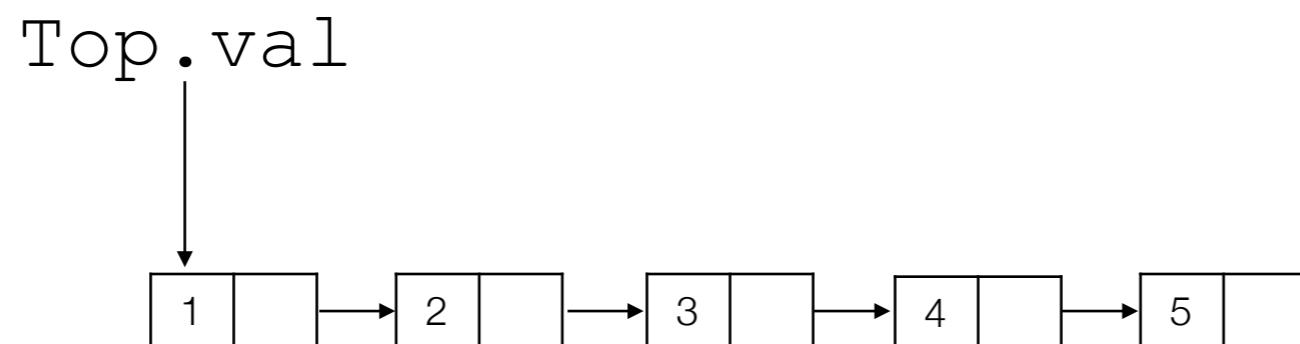
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------



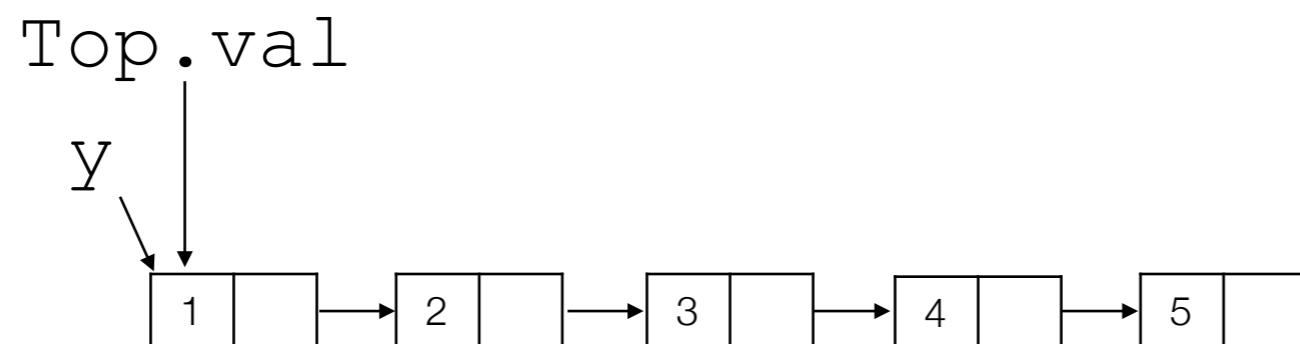
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { → y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------



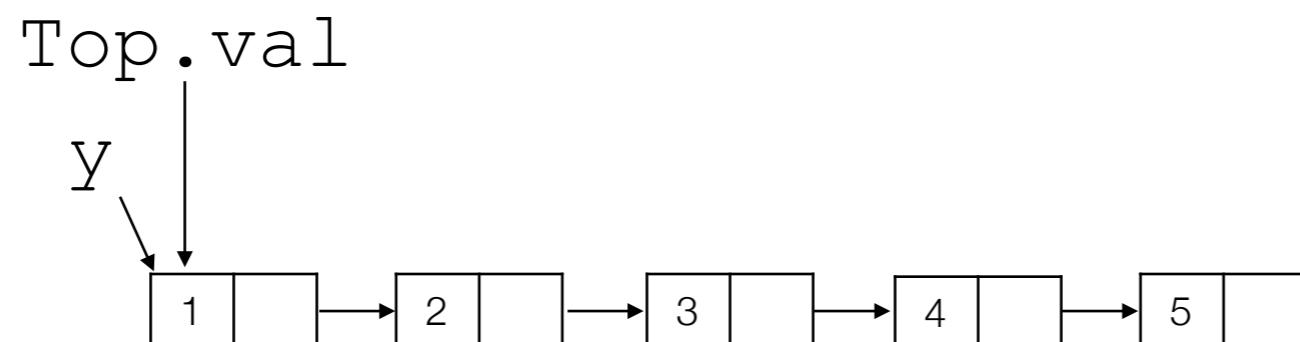
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { → y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------



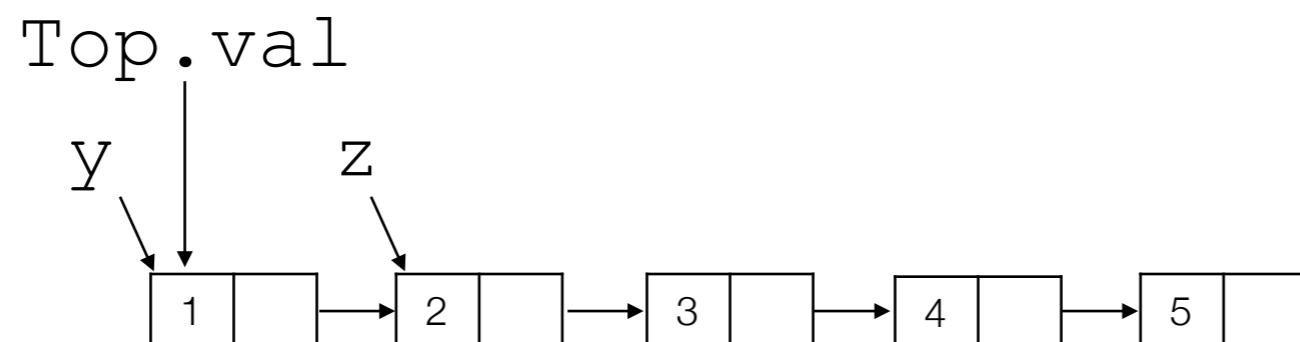
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; → z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------



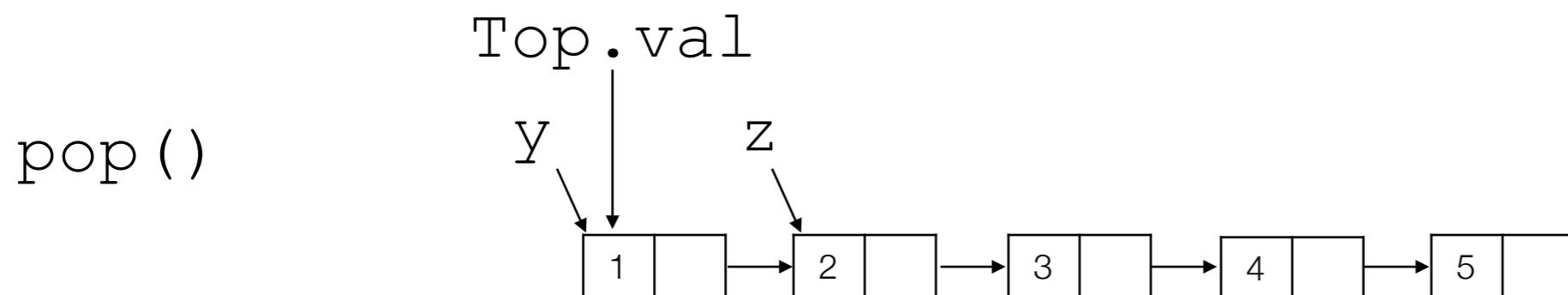
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; → z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------



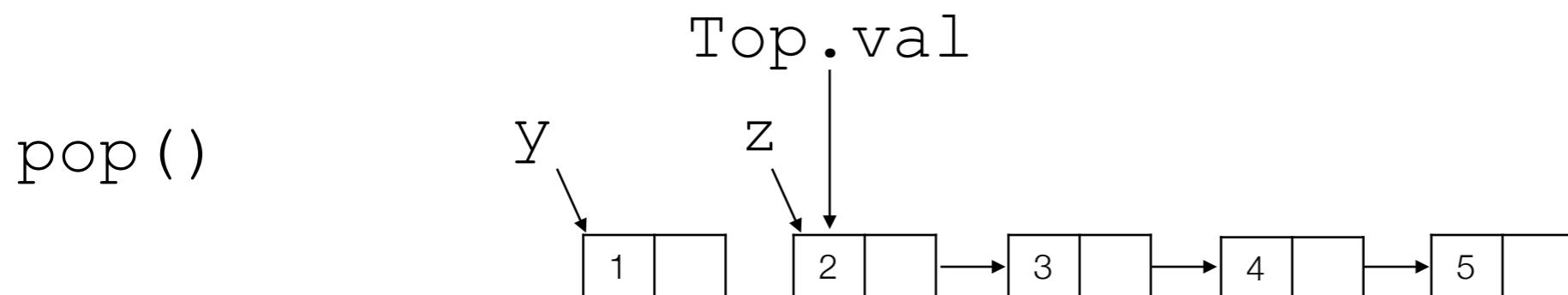
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------



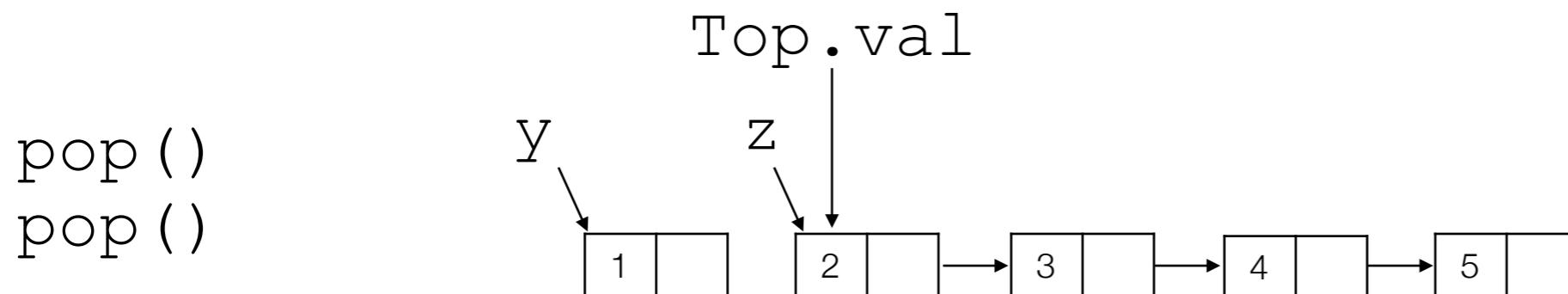
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------



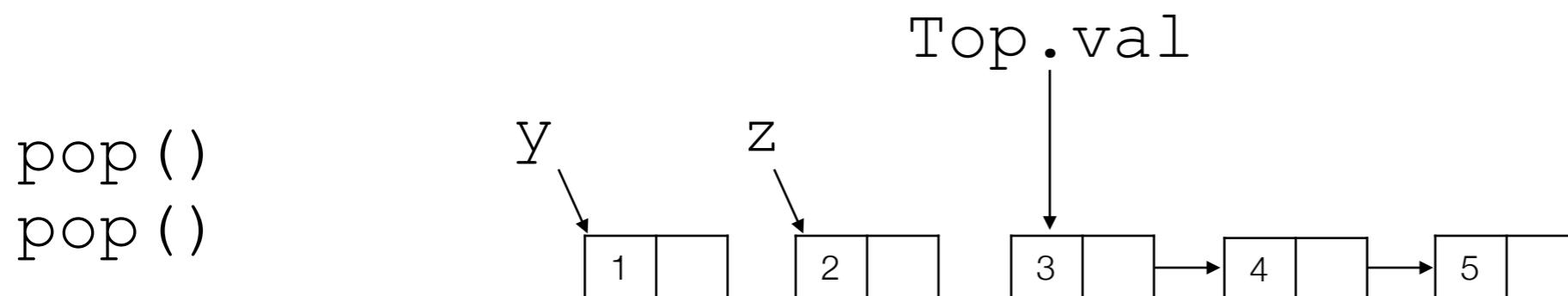
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------



Treiber Stack (ABA problem)

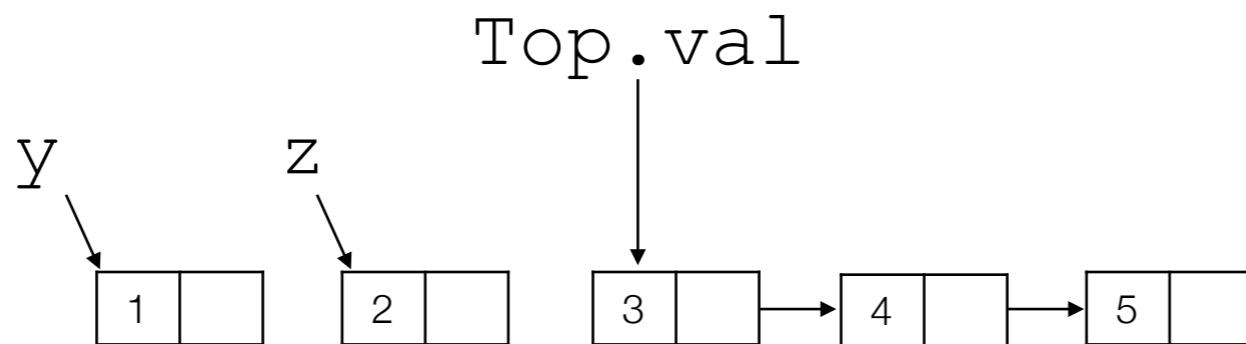
class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------



Treiber Stack (ABA problem)

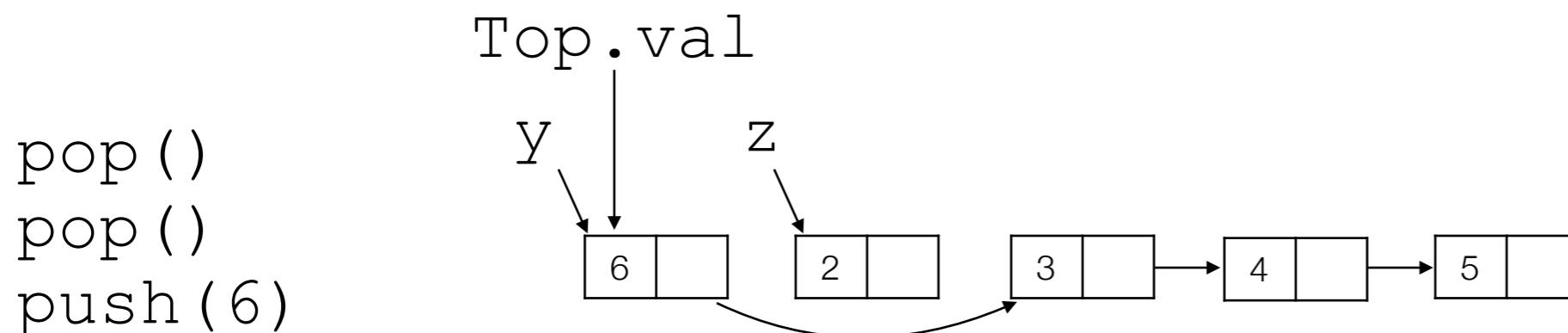
```
class Node { void push(int e) { int pop() { Node tl; Node y, n; Node y, z; int val; y = new(); while(true) { while(true) { n = TOP->val; if (cas(TOP->val, y->val, y)) z = y->tl; if (cas(TOP->val, n, y)) break; if (cas(TOP->val, y->val, z)) break; } return y->val; } } } } class NodePtr { Node val; } TOP;
```

pop()
pop()
push(6)



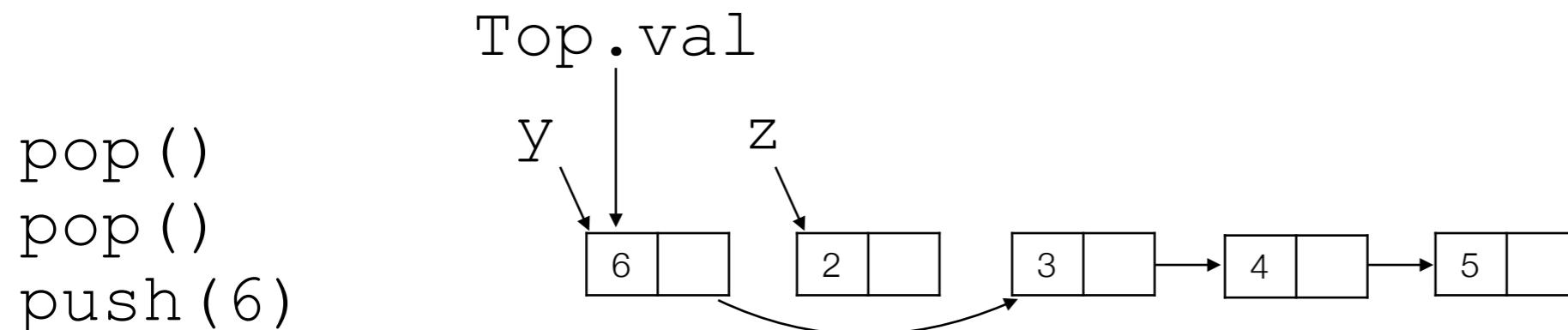
Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------



Treiber Stack (ABA problem)

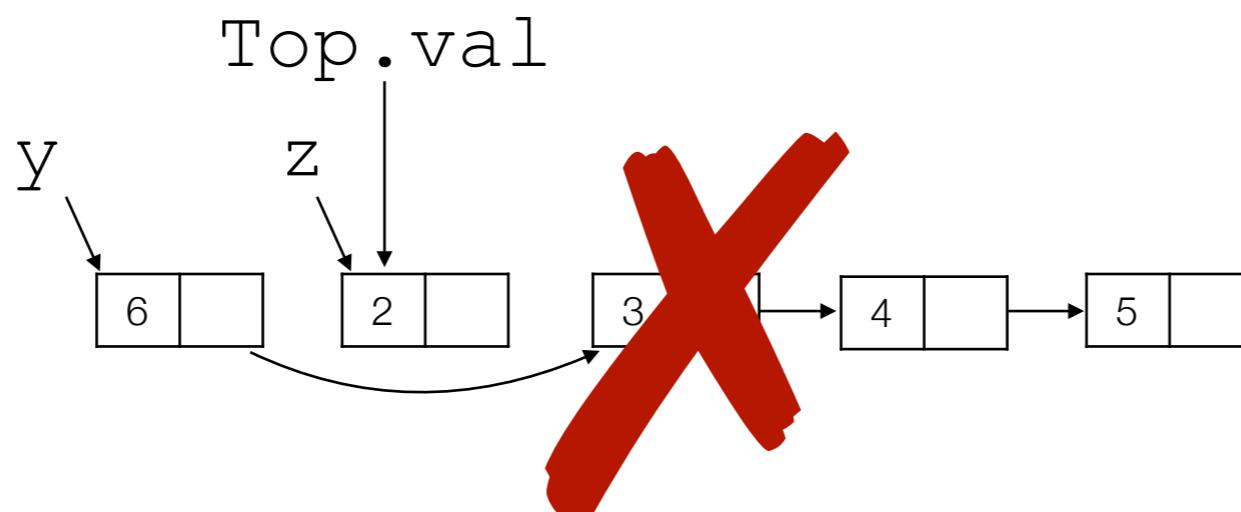
class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; → if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------



Treiber Stack (ABA problem)

class Node { Node tl; int val; } class NodePtr { Node val; } TOP;	void push(int e) { Node y, n; y = new(); y->val = e; while(true) { n = TOP->val; y->tl = n; if (cas(TOP->val, n, y)) break; } }	int pop() { Node y, z; while(true) { y = TOP->val; if (y==0) return EMPTY; z = y->tl; → if (cas(TOP->val, y, z)) break; } return y->val; }
-----------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------

pop()
pop()
push(6)



Treiber + Hazard Pointers

```
class Node {           void push(int e) {           int pop() {  
    Node tl;          Node y, n;           Node y, z;  
    int val;          y = new();           while(true) {  
}           y->val = e;           y = TOP->val;  
class NodePtr{       for(n=0; n<=THREADS; n++)           if (y==0) return EMPTY;  
    Node val;         if (H[n] == y)           (TOP->H)[getTid()] = y;  
    TID[] H;           return false;           if (TOP->val != y)  
} TOP;             while(true)           continue;  
                     n = TOP->val;           z = y->tl;  
                     y->tl = n;           if (cas(TOP->val, y, z))  
                     if (cas(TOP->val, n, y))           break;  
                     break;           }  
                     }           (TOP->H)[getTid()] = null  
                     return true;           return y->val;  
} }
```

HSY Elimination Stack

Extremely simplified version: 1 collision

```
class Node {  
    Node tl;  
    int val;  
}  
  
class NodePtr {  
    Node val;  
} TOP;  
  
class TidPtr {  
    int val;  
} clash;  
  
void push(int e) {  
    Node y, n;  
    TID hisId;  
    y = new();  
    y->val = e;  
  
    while (true) {  
        n = TOP->val;  
        y->tl = n;  
        if (cas(TOP->val, n, y))  
            return;  
        //elimination scheme  
        TidPtr t = new TidPrt();  
        t->val = e;  
        if (cas(clash,null,t)){  
            wait(DELAY);  
            //not eliminated  
            if (cas(clash,t,null))  
                continue;  
            else break; //eliminated  
        }  
    }  
}  
  
int pop() {  
    Node y,z;  
    int t;  
    TID hisId;  
    while (true) {  
        y = TOP->val;  
        if (y == 0)  
            return EMPTY;  
        z = y->tl;  
        t = y->val;  
        if (cas(TOP->val, y, z))  
            return t;  
        //elimination scheme  
        pusher = clash;  
        while (pusher!=null){  
            if (cas(clash, pusher, null))  
                //eliminated push  
                return pusher->val;  
        }  
    }  
}
```

VERIFYING LINEARIZABILITY

RGSEP: Rely Guarantee + Separation Logic

Verifying Linearizability with RGSEP

- ▶ Today: Program Logic approach (manual)
 - ▶ Later: automate part of the process
- ▶ Separation Logic (SL)
- ▶ Rely Guarantee (RG)
- ▶ Linearizability Proofs

Verifying Linearizability

- ▶ Simulation based approach:

1. For each method locate the *linearization point* in the code (conceptually the atomic execution)
2. Embed an “*abstract atomic operation*” at the linearization point
3. Provide an *abstraction map* between concrete and abstract states
4. Prove the invariance of the concrete and abstract state relation
5. Show that abstract and concrete operations have the same return values for similar inputs

Linearizability with RGSep

A Primer on Hoare-Logics

- ▶ Also known as (Floyd-)Hoare Logic
- ▶ We need a language of assertions describing property of the state at different program points
- ▶ Program variables can be used in assertions to relate the program term and the state
- ▶ Invariants: Assertions that are true of all the states in a piece of code (e.g. loop invariant, global invariant, etc.)

Assertions

$P ::= True$	Any possible state
$\neg P$	
$P \wedge Q$	All states that satisfy P and Q
$P \vee Q$	
$P \Rightarrow Q$	
$x = v \mid \dots$	Logical variables relate values in
$\exists X, P$	one state $\{\exists X, x = X \wedge y = X + 1\}$
$\forall X, P$	

Substitutions

$$\{\exists X, x = X \wedge y = X + 1\}[x \leftarrow 8] = \{\exists X, 8 = X \wedge y = X + 1\} \\ \equiv \{y = 9\}$$

Notation: $P[x \leftarrow 8] \approx [x/8]P$

The meaning of Triples

For any initial state
satisfying P

The final state must
satisfy Q

$$\{P\} \ c \ \{Q\} \iff \forall \sigma \models P, \text{ when } (\sigma, c) \xrightarrow{*} \sigma', \text{ we have } \sigma' \models Q$$

If we execute c reaching a final state

Triples can be composed to
prove complex programs

$$\{Pre\} \ c_0; \{P\} \ c; \{Q\} \ c_1; \{Post\}$$

Check these examples

$$\{x = 3\} \quad x := x + 1 \quad \{x \leq 0\}$$
$$\{x = 3\} \quad x := x + 1 \quad \{x \geq 0\}$$
$$\{x = 3\} \quad x := x + 1; y := x \quad \{y \geq 0\}$$
$$\{\exists X, x = X \wedge y = X + 1\} \quad x := x + 1; y := x \quad \{x = y\}$$
$$\{x = 0\} \text{ if } \textit{true} \text{ then } x := x + 1 \text{ else } x := x - 1 \text{ fi } \{x = 1\}$$
$$\{x = 0\} \text{ if } x = 5 \text{ then } x := x + 1 \text{ else } x := x - 1 \text{ fi } \{x = 1\}$$
$$\{x = 0\} \text{ while } \textit{true} \text{ do } x := x + 1 \text{ od } \{\textit{false}\}$$

Hoare Logic Rules

Hoare Logic Rules

$\{P\} \ skip \ {P}$

Hoare Logic Rules

$$\{P\} \text{ skip } \{P\}$$
$$\{P[x \leftarrow e]\} \ x := e \ \{P\}$$

Hoare Logic Rules

$$\{P\} \text{ skip } \{P\}$$

$$\{P[x \leftarrow e]\} \ x := e \ \{P\}$$

$$\frac{\{P\} \ c_0 \ \{R\} \quad \{R\} \ c_1 \ \{Q\}}{\{P\} \ c_0; c_1 \ \{Q\}}$$

Hoare Logic Rules

$$\{P\} \text{ skip } \{P\}$$

$$\{P[x \leftarrow e]\} \ x := e \ \{P\}$$

$$\frac{\{P\} \ c_0 \ \{R\} \quad \{R\} \ c_1 \ \{Q\}}{\{P\} \ c_0; c_1 \ \{Q\}}$$

$$\frac{\{P \wedge b\} \ c_0 \ \{Q_0\} \quad \{P \wedge \neg b\} \ c_1 \ \{Q_1\}}{\{P\} \text{ if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$

Hoare Logic Rules

$$\{P\} \text{ skip } \{P\}$$

$$\{P[x \leftarrow e]\} \ x := e \ \{P\}$$

$$\frac{\{P\} \ c_0 \ \{R\} \quad \{R\} \ c_1 \ \{Q\}}{\{P\} \ c_0; c_1 \ \{Q\}}$$

$$\frac{\{P \wedge b\} \ c_0 \ \{Q_0\} \quad \{P \wedge \neg b\} \ c_1 \ \{Q_1\}}{\{P\} \text{ if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$

$$\frac{\{P \wedge b\} \ c \ \{P\}}{\{P\} \text{ while } b \text{ do } c \text{ od } \{\neg b \wedge P\}}$$

Hoare Logic Rules

$$\{P\} \text{ skip } \{P\}$$

$$\{P[x \leftarrow e]\} \ x := e \ \{P\}$$

$$\frac{\{P\} \ c_0 \ \{R\} \quad \{R\} \ c_1 \ \{Q\}}{\{P\} \ c_0; c_1 \ \{Q\}}$$

$$\frac{\{P\} \ c \ \{Q\} \quad P' \Rightarrow P \quad Q \Rightarrow Q'}{\{P'\} \ c \ \{Q'\}}$$

$$\frac{\{P \wedge b\} \ c_0 \ \{Q_0\} \quad \{P \wedge \neg b\} \ c_1 \ \{Q_1\}}{\{P\} \text{ if } b \text{ then } c_0 \text{ else } c_1 \text{ fi } \{b \Rightarrow Q_0 \wedge \neg b \Rightarrow Q_1\}}$$

$$\frac{\{P \wedge b\} \ c \ \{P\}}{\{P\} \text{ while } b \text{ do } c \text{ od } \{\neg b \wedge P\}}$$

A simple Hoare Logic Proof

$\{x = X \wedge y = Y\}$

aux = **x**;

x = **y**;

y = **aux**;

$\{x = Y \wedge y = X\}$

A simple Hoare Logic Proof

$\{x = X \wedge y = Y\}$

aux = **x**;

$\{x = X \wedge y = Y \wedge \text{aux} = X\}$

x = **y**;

y = **aux**;

$\{x = Y \wedge y = X\}$

A simple Hoare Logic Proof

$\{x = X \wedge y = Y\}$

aux = **x**;

$\{x = X \wedge y = Y \wedge \text{aux} = X\}$

x = **y**;

$\{x = Y \wedge y = Y \wedge \text{aux} = X\}$

y = **aux**;

$\{x = Y \wedge y = X\}$

The Owicky/Gries Method

- ▶ Hoare logics is insufficient to prove concurrent programs
- ▶ In 1976 Owicky and Gries provide a method to verify concurrent programs (using the cobegin/ coend parallel construct)
- ▶ First method to be able to prove properties of multiprgrams

Quick intro to OG (R/G)

$$\frac{\{P_0\} \ C_0 \ \{Q_0\} \quad \{P_1\} \ C_1 \ \{Q_1\} \quad \text{non-interference}}{\{P_0 \wedge P_1\} \ C_0 \| C_1 \ \{Q_0 \wedge Q_1\}}$$

Quick intro to OG (R/G)

$$\frac{\{P_0\} \ C_0 \ \{Q_0\} \quad \{P_1\} \ C_1 \ \{Q_1\} \quad \text{non-interference}}{\{P_0 \wedge P_1\} \ C_0 \| C_1 \ \{Q_0 \wedge Q_1\}}$$

Non-Interference

- ▶ For each triple $\{p_i\} \ c_i \ \{q_i\}$ occurring in $\{P_1\} \ C_1 \ \{Q_1\}$
- ▶ For each pair $\{p_j\} \ c_j$ occurring in $\{P_0\} \ C_0 \ \{Q_0\}$, where c_j is an atomic command (write, read, ...)
- ▶ Show that: p_i and q_i are stable w.r.t. $\{p_i \wedge p_j\} \ c_j \ \{p_i\}$

Quick intro to OG (R/G)

$$\frac{\{P_0\} C_0 \{Q_0\} \quad \{P_1\} C_1 \{Q_1\} \quad \text{non-interference}}{\{P_0 \wedge P_1\} C_0 \| C_1 \{Q_0 \wedge Q_1\}}$$

Non-Interference

- ▶ For each triple $\{p_i\} c_i \{q_i\}$ occurring in $\{P_1\} C_1 \{Q_1\}$
- ▶ For each pair $\{p_j\} c_j$ occurring in $\{P_0\} C_0 \{Q_0\}$, where c_j is an atomic command (write, read, ...)
- ▶ Show that: p_i and q_i are stable w.r.t. $\{p_i \wedge p_j\} c_j \{p_i\}$

Exponential number of proof obligations on number of threads

To the Board!

$$\{x = 0 \wedge \text{flag} = \text{false}\}$$

$x = 100$ || $\text{if}(\text{flag})$
 $\text{flag} = \text{true}$ || $x = x - 50$

$$\{x \geq 50 \wedge \text{flag} = \text{true}\}$$

To the Board!

$$\{x = 0 \wedge \text{flag} = \text{false}\}$$

$$\begin{array}{lcl} x = 100 & & \text{if}(\text{flag}) \\ \text{flag} = \text{true} & \parallel & x = x - 50 \end{array}$$

$$\{x \geq 50 \wedge \text{flag} = \text{true}\}$$

$$\{x = 0\}$$

$$\text{Atomic} \longrightarrow x = x + 1 \parallel x = x + 1$$

$$\{x = 2\}$$

To the Board!

$$\{x = 0 \wedge \text{flag} = \text{false}\}$$

$$\begin{array}{ll} x = 100 & \text{if(flag)} \\ \text{flag} = \text{true} & \| \\ & x = x - 50 \end{array}$$

$$\{x \geq 50 \wedge \text{flag} = \text{true}\}$$

$$\{x = 0\}$$

$$\text{Atomic} \longrightarrow x = x + 1 \quad \| \quad x = x + 1$$

$$\{x = 2\}$$

Ghost Variables

Separation Logic (SL)

P	$::=$	$True$
		$\neg P$
		$P \wedge Q$
		$P \vee Q$
		$P \Rightarrow Q$
		$x = v \mid \dots$
		$\exists X, P$
		$\forall X, P$
		$\forall X, P$
Spatial Assertions		emp
		junk
		$x \mapsto v$
		$P * Q$
		$P -\circledast Q$

Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	
	$P \Rightarrow Q$	
	$x = v \mid \dots$	
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
Spatial Assertions	emp	
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -@ Q$	

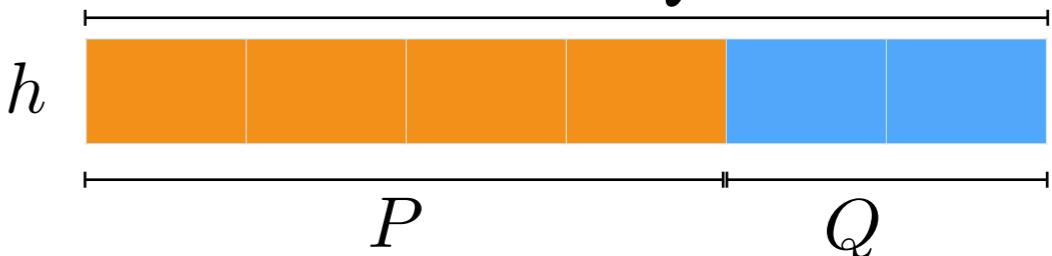
Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge$
	$P \Rightarrow Q$	$i(x) = p \wedge h[i] = v$
	$x = v \mid \dots$	
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
Spatial Assertions	emp	
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -@ Q$	

Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge$
	$P \Rightarrow Q$	$i(x) = p \wedge h[i] = v$
	$x = v \mid \dots$	
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
Spatial Assertions	emp	
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -@ Q$	

Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge i(x) = p \wedge h[i] = v$
	$P \Rightarrow Q$	
	$x = v \mid \dots$	$h, i \models_{SL} (P * Q) = \exists h_1 h_2, (h_1 \uplus h_2 = h) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
	emp	
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -\circledast Q$	
Spatial Assertions		 <p style="text-align: center;">$P * Q$</p>

Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge$
	$P \Rightarrow Q$	$i(x) = p \wedge h[i] = v$
	$x = v \mid \dots$	$h, i \models_{SL} (P * Q) =$
	$\exists X, P$	$\exists h_1 h_2, (h_1 \uplus h_2 = h) \wedge$
	$\forall X, P$	$h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	$\forall X, P$	
Spatial Assertions	emp	
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -@ Q$	

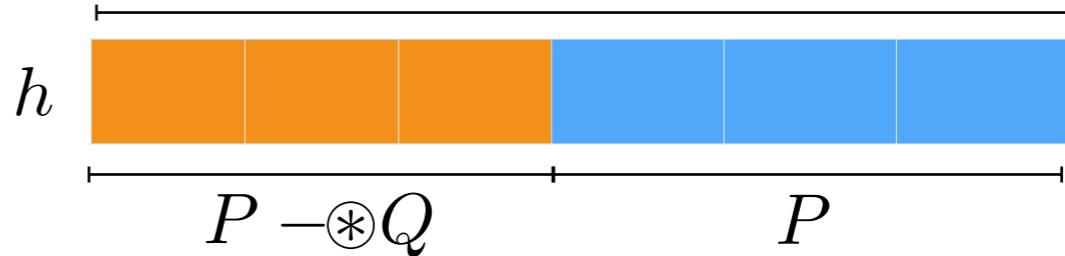
Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge i(x) = p \wedge h[i] = v$
	$P \Rightarrow Q$	
	$x = v \mid \dots$	$h, i \models_{SL} (P * Q) = \exists h_1 h_2, (h_1 \uplus h_2 = h) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
Spatial Assertions	emp	$h, i \models_{SL} (P -\circledast Q) = \exists h_1 h_2, (h_1 \uplus h = h_2) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -\circledast Q$	

$$h \quad \begin{array}{|c|c|c|}\hline & | & | & | \\ \hline & & & \end{array} \quad P -\circledast Q$$

Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge i(x) = p \wedge h[i] = v$
	$P \Rightarrow Q$	
	$x = v \mid \dots$	$h, i \models_{SL} (P * Q) = \exists h_1 h_2, (h_1 \uplus h_2 = h) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
Spatial Assertions		$h, i \models_{SL} (P -\circledast Q) = \exists h_1 h_2, (h_1 \uplus h = h_2) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	emp	
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -\circledast Q$	

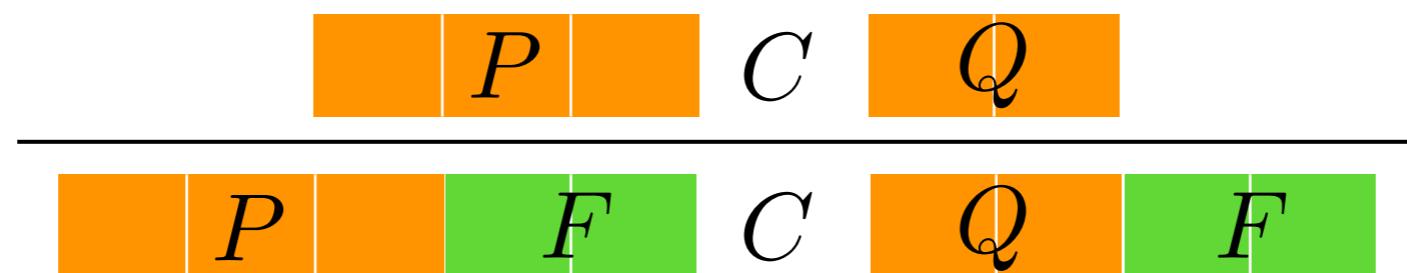


Separation Logic (SL)

$P ::=$	$True$	$h, i \models_{SL} \text{emp} = \text{dom}(h) = \emptyset$
	$\neg P$	$h, i \models_{SL} \text{junk} = h, i \models_{SL} \neg \text{emp}$
	$P \wedge Q$	
	$P \vee Q$	$h, i \models_{SL} x \mapsto v = \exists p, \text{dom}(h) = \{p\} \wedge i(x) = p \wedge h[i] = v$
	$P \Rightarrow Q$	
	$x = v \mid \dots$	$h, i \models_{SL} (P * Q) = \exists h1\ h2, (h_1 \uplus h_2 = h) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	$\exists X, P$	
	$\forall X, P$	
	$\forall X, P$	
Spatial Assertions	emp	$h, i \models_{SL} (P -\circledast Q) = \exists h1\ h2, (h_1 \uplus h = h_2) \wedge h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q$
	junk	
	$x \mapsto v$	
	$P * Q$	
	$P -\circledast Q$	

SL Frame Rule

$$\frac{\{P\} \ C \ \{Q\}}{\{P * F\} \ C \ \{Q * F\}}$$



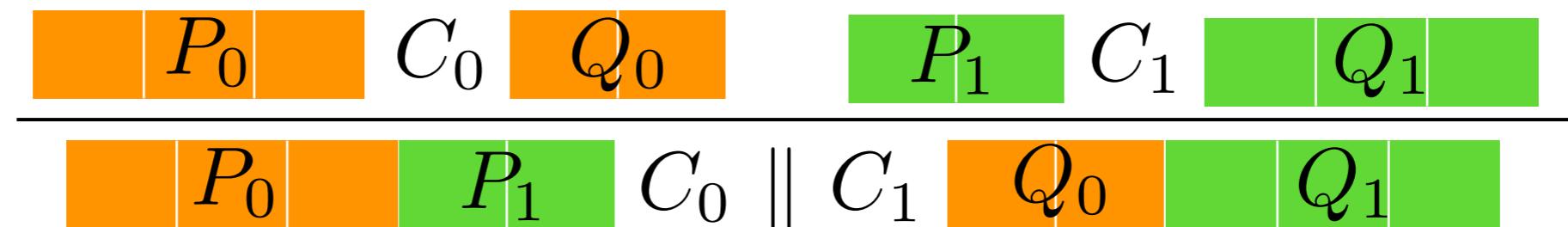
Concurrent Separation Logic (CSL)

$$\frac{\{P_0\} \ C_0 \ \{Q_0\} \quad \{P_1\} \ C_1 \ \{Q_1\}}{\{P_0 * P_1\} \ C_0 \parallel C_1 \ \{Q_0 * Q_1\}}$$

$$\frac{\begin{array}{c|c|c|c|c} & P_0 & & C_0 & Q_0 \\ \hline & & & & \end{array} \quad \begin{array}{c|c|c|c|c} P_1 & & C_1 & & Q_1 \\ \hline & & & & \end{array}}{\begin{array}{c|c|c|c|c|c|c} & P_0 & & P_1 & & C_0 & \parallel & C_1 & Q_0 & & Q_1 \\ \hline & & & & & & & & & & \end{array}}$$

Concurrent Separation Logic (CSL)

$$\frac{\{P_0\} \ C_0 \ \{Q_0\} \quad \{P_1\} \ C_1 \ \{Q_1\}}{\{P_0 * P_1\} \ C_0 \parallel C_1 \ \{Q_0 * Q_1\}}$$



We will ignore resources for now

SL + Rely Guarantee (RGSep)

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x. \, p \mid \forall x. \, p$$

SL + Rely Guarantee (RGSep)

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x. \, p \mid \forall x. \, p$$

The expression \boxed{P} is highlighted with a box. Two arrows point to this box from two separate rectangular boxes below it. The left box is labeled "Local State" and the right box is labeled "Global State".

SL + Rely Guarantee (RGSep)

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x. \, p \mid \forall x. \, p$$

SL + Rely Guarantee (RGSep)

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x. \, p \mid \forall x. \, p$$

Interference

$$\llbracket P \rightsquigarrow Q \rrbracket = \{(h_1 \uplus h_0, h_2 \uplus h_0) \mid h_1, i \vDash_{SL} P \wedge h_2, i \vDash_{SL} Q\}$$

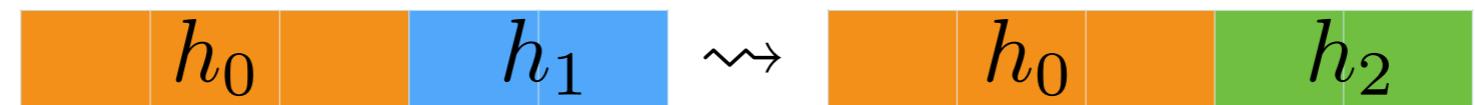
SL + Rely Guarantee (RGSep)

Locality

$$p, q, r ::= P \mid \boxed{P} \mid p * q \mid p \wedge q \mid p \vee q \mid \exists x. \, p \mid \forall x. \, p$$

Interference

$$\llbracket P \rightsquigarrow Q \rrbracket = \{(h_1 \uplus h_0, h_2 \uplus h_0) \mid h_1, i \models_{SL} P \wedge h_2, i \models_{SL} Q\}$$



RGSep Judgment

Pre Post

$$R,G \models \{P\} \ c \ \{Q\}$$

RGSep Judgment

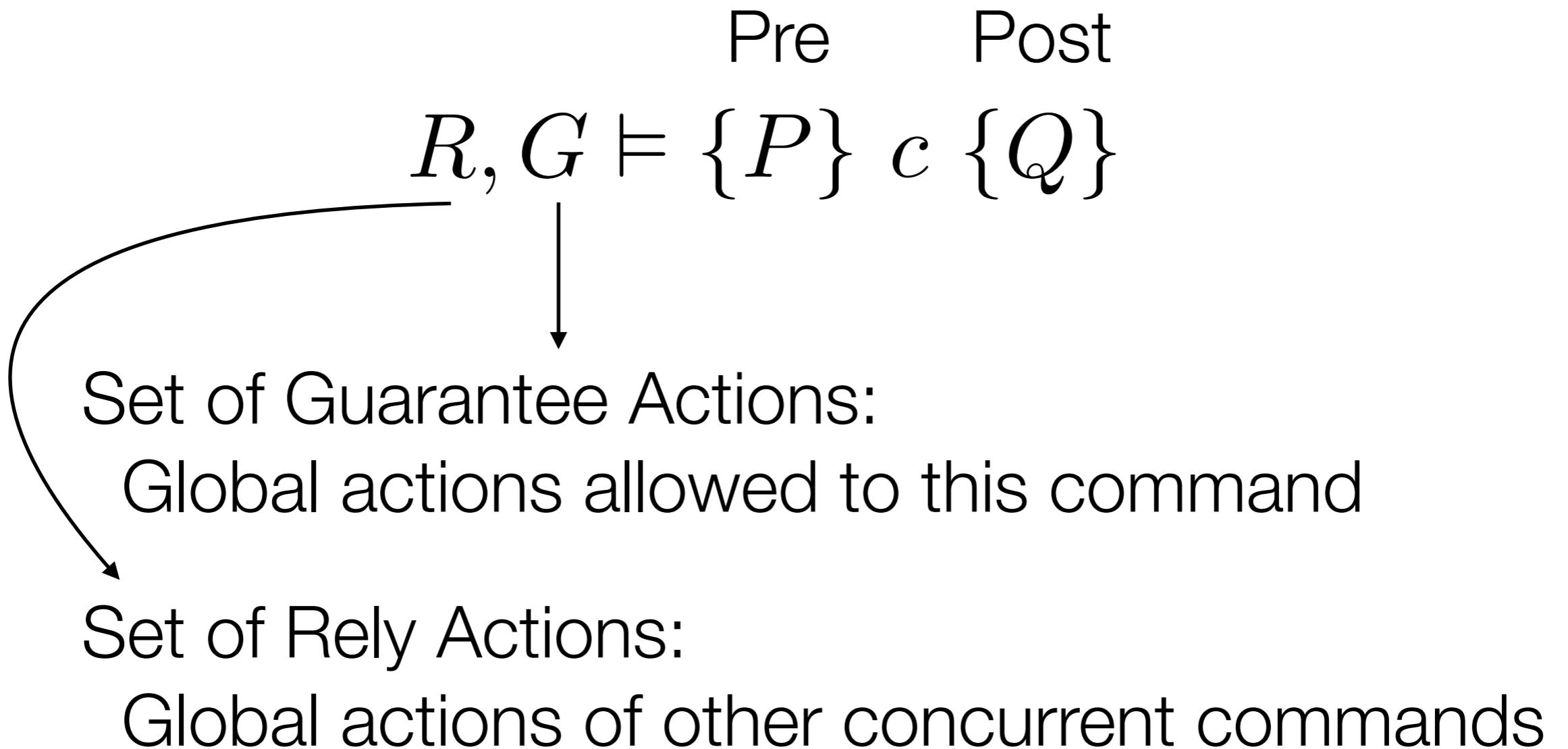
$$R, G \models \{P\} \; c \; \{Q\}$$

↓

Pre Post

Set of Guarantee Actions:
Global actions allowed to this command

RGSep Judgment



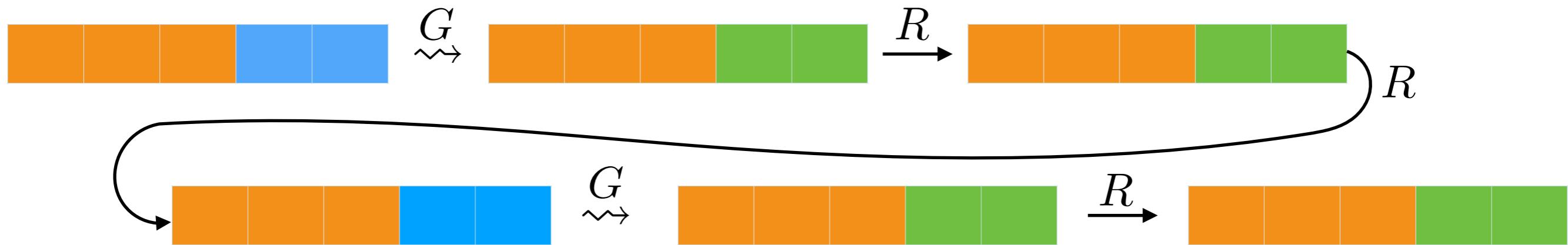
RGSep Judgment

$$R, G \models \{P\} \; c \; \{Q\}$$

Pre Post

Set of Guarantee Actions:
Global actions allowed to this command

Set of Rely Actions:
Global actions of other concurrent commands



RGSep Proof Rules

RGSep Proof Rules

$$\frac{R, G \vdash \{P\} \ C \ \{Q\} \quad F \text{ stable for } (R \cup G) \text{ or } C \text{ has no atomic}}{R, G \vdash \{P * F\} \ C \ \{Q * F\}}$$

RGSep Proof Rules

$$R, G \vdash \{P\} \ C \ \{Q\}$$

F stable for $(R \cup G)$ or C has no atomic

$$R, G \vdash \{P * F\} \ C \ \{Q * F\}$$

$$Q \equiv (P * X \mapsto Y) \quad x \notin fv(P)$$

$$\frac{}{R, G \vdash \{\boxed{Q} \wedge e = X\} \ x := [e] \ \{\boxed{Q} * x = Y\}}$$

RGSep Proof Rules

$$R, G \vdash \{P\} \ C \ \{Q\}$$

F stable for $(R \cup G)$ or C has no atomic

$$R, G \vdash \{P * F\} \ C \ \{Q * F\}$$

$$Q \equiv (P * X \mapsto Y) \quad x \notin fv(P)$$

$$\frac{}{R, G \vdash \{\boxed{Q} \wedge e = X\} \ x := [e] \ \{\boxed{Q} * x = Y\}}$$

$$R, G \vdash \{P\} \ C_1 \ \{R\} \quad R, G \vdash \{R\} \ C_2 \ \{Q\}$$

$$R, G \vdash \{P\} \ C_1; C_2 \ \{Q\}$$

RGSep Proof Rules

$$R, G \vdash \{P\} \ C \ \{Q\}$$

F stable for $(R \cup G)$ or C has no atomic

$$R, G \vdash \{P * F\} \ C \ \{Q * F\}$$

$$\frac{}{Q \equiv (P * X \mapsto Y) \quad x \notin fv(P)}$$

$$\frac{}{R, G \vdash \{\boxed{Q} \wedge e = X\} \ x := [e] \ \{\boxed{Q} * x = Y\}}$$

$$R, G \vdash \{P\} \ C_1 \ \{R\} \quad R, G \vdash \{R\} \ C_2 \ \{Q\}$$

$$R, G \vdash \{P\} \ C_1; C_2 \ \{Q\}$$

$$\frac{\vdash \{P_1 * P_2\} \ C \ \{Q_1 * Q_2\} \quad \boxed{Q} \text{ stable for } R}{\bar{y} \cap fv(P_2) = \emptyset \quad P \Rightarrow P_1 * F \quad Q_1 * F \Rightarrow Q \quad (P_1 \rightsquigarrow Q_1) \subseteq G}$$

$$\frac{}{\vdash \{\exists \bar{y}. \boxed{P} * P_2\} \text{ atomic } C \ \{\exists \bar{y}. \boxed{Q} * Q_2\}}$$

RGSep Proof Rules

$$R, G \vdash \{P\} \ C \ \{Q\}$$

F stable for $(R \cup G)$ or C has no atomic

$$R, G \vdash \{P * F\} \ C \ \{Q * F\}$$

$$\frac{Q \equiv (P * X \mapsto Y) \quad x \notin fv(P)}{R, G \vdash \{\boxed{Q} \wedge e = X\} \ x := [e] \ \{\boxed{Q} * x = Y\}}$$

$$R, G \vdash \{P\} \ C_1 \ \{R\} \quad R, G \vdash \{R\} \ C_2 \ \{Q\}$$

$$R, G \vdash \{P\} \ C_1; C_2 \ \{Q\}$$

$$\frac{\begin{array}{c} \vdash \{P_1 * P_2\} \ C \ \{Q_1 * Q_2\} \quad \boxed{Q} \text{ stable for } R \\ \bar{y} \cap fv(P_2) = \emptyset \quad P \Rightarrow P_1 * F \quad Q_1 * F \Rightarrow Q \quad (P_1 \rightsquigarrow Q_1) \subseteq G \end{array}}{\vdash \{\exists \bar{y}. \boxed{P} * P_2\} \text{ atomic } C \ \{\exists \bar{y}. \boxed{Q} * Q_2\}}$$

$$R \cup G_2, G_1 \vdash \{P_1\} \ C_1 \ \{Q_1\} \quad P_1 \text{ stable for } R \cup G_2$$

$$R \cup G_1, G_2 \vdash \{P_2\} \ C_2 \ \{Q_2\} \quad P_2 \text{ stable for } R \cup G_1$$

$$R, G_1 \cup G_2 \vdash \{P_1 * P_2\} \ C_1 \| C_2 \ \{Q_1 * Q_2\}$$

RGSep Proof Rules

$$\frac{}{x \mapsto y \rightsquigarrow x \mapsto y \subseteq G} \text{G-EXACT}$$

$$\frac{P_1 \rightsquigarrow S * Q_1 \subseteq G \quad P_2 * S \rightsquigarrow Q_2 \subseteq G}{P_1 * P_2 \rightsquigarrow Q_1 * Q_2 \subseteq G} \text{G-SEQ}$$

$$\frac{\models_{\text{SL}} P' \Rightarrow P \quad P \rightsquigarrow Q \subseteq G \quad \models_{\text{SL}} Q' \Rightarrow Q}{P' \rightsquigarrow Q' \subseteq G} \text{G-CONS}$$

$$\frac{P \rightsquigarrow Q \in G}{P \rightsquigarrow Q \subseteq G} \text{G-AX}$$

$$\frac{P \rightsquigarrow Q \subseteq G}{P[e/x] \rightsquigarrow Q[e/x] \subseteq G} \text{G-SUB}$$

$$\frac{(P * F) \rightsquigarrow (Q * F) \subseteq G}{P \rightsquigarrow Q \subseteq G} \text{G-COFRM}$$

To the Board (RGSep)

$$\{x = 0 \wedge \text{flag} = \text{false}\}$$

$$\begin{array}{ll} x = 100 & \text{if(flag)} \\ \text{flag} = \text{true} & \| \\ & x = x - 50 \end{array}$$

$$\{x \geq 50 \wedge \text{flag} = \text{true}\}$$

$$\{x = 0\}$$

$$x = x + 1 \quad \| \quad x = x + 1$$

$$\{x = 2\}$$

Linearizability: Proof Technique

- ▶ For each implementation method of a library:
 - ▶ Identify a *syntactic linearization point*
 - ▶ Check that for each successful execution of a method there is *exactly one linearization point*
 - ▶ Check that the *input/output corresponds to the sequential spec.* of the object

Linearizability in RGSep: Some Additional Ingredients

Single Assignment Variables

$$\begin{array}{lll} \{emp\} & x := \text{new}_{\text{single}}; & \{x \xrightarrow{s} \text{undef}\} \\ \{y \xrightarrow{s} z \wedge e = y\} & x := [e]_{\text{single}}; & \{y \xrightarrow{s} z \wedge x = z\} \\ \{e_1 \xrightarrow{s} \text{undef}\} & [e_1]_{\text{single}} := e_2; & \{e_1 \xrightarrow{s} e_2\} \\ \{e \xrightarrow{s} __ \} & \text{dispose}_{\text{single}}(e); & \{emp\} \end{array}$$

Linearizability in RGSep: Some Additional Ingredients

Single Assignment Variables

$$\begin{array}{lll} \{emp\} & x := \mathbf{new}_{\mathbf{single}}; & \{x \xrightarrow{s} undef\} \\ \{y \xrightarrow{s} z \wedge e = y\} & x := [e]_{\mathbf{single}}; & \{y \xrightarrow{s} z \wedge x = z\} \\ \{e_1 \xrightarrow{s} undef\} & [e_1]_{\mathbf{single}} := e_2; & \{e_1 \xrightarrow{s} e_2\} \\ \{e \xrightarrow{s} __ \} & \mathbf{dispose}_{\mathbf{single}}(e); & \{emp\} \end{array}$$

Simulation Argument in RGSep

$$\{p \wedge d.\mathbf{AbsResult} \xrightarrow{s} undef\} \text{ConcreteOp}(d) \{d.\mathbf{AbsResult} \xrightarrow{s} \mathbf{Result}\}$$

Linearizability in RGSep: Some Additional Ingredients

Single Assignment Variables

$$\begin{array}{l} \{emp\} \quad x := \mathbf{new}_{\mathbf{single}}; \quad \{x \xrightarrow{s} \text{undef}\} \\ \{y \xrightarrow{s} z \wedge e = y\} \quad x := [e]_{\mathbf{single}}; \quad \{y \xrightarrow{s} z \wedge x = z\} \\ \{e_1 \xrightarrow{s} \text{undef}\} \quad [e_1]_{\mathbf{single}} := e_2; \quad \{e_1 \xrightarrow{s} e_2\} \\ \{e \xrightarrow{s} __ \} \quad \mathbf{dispose}_{\mathbf{single}}(e); \quad \{emp\} \end{array}$$

Simulation Argument in RGSep

$$\{p \wedge d.\mathbf{AbsResult} \xrightarrow{s} \text{undef}\} \text{ ConcreteOp}(d) \quad \{d.\mathbf{AbsResult} \xrightarrow{s} \mathbf{Result}\}$$

Shorthand Annotation for Linearization Points

$$\mathbf{Lin}_{e_1, \dots, e_n} \stackrel{\text{def}}{=} e_1.\mathbf{AbsResult} := \mathbf{AbsOp}(e_1); \dots; e_n.\mathbf{AbsResult} := \mathbf{AbsOp}(e_n);$$

Specifications: from histories to states

Predefined Abstract State: **Abs**

Abstract specification of the operations

$$\begin{aligned}\text{Abs_push}(e) &\stackrel{\text{def}}{=} \langle \text{Abs} := e \cdot \text{Abs}; \text{AbsResult} := e; \rangle \\ \text{Abs_pop}() &\stackrel{\text{def}}{=} \langle \text{case } (\text{Abs}) \\ &\quad | \epsilon \implies \text{AbsResult} := \text{EMPTY}; \\ &\quad | v \cdot A \implies \{ \text{Abs} := A; \text{AbsResult} := v; \} \rangle\end{aligned}$$

We can generate the specification histories

$$\delta = \langle \text{Abs_push}(v), _ \rangle \cdot \langle \text{Abs_pop}(v), v \rangle \dots$$

Treiber Stack

Treiber Stack in RGSep

```
class Cell {Cell next;  value_t data; }  
Cell S, Abs;  
  
void push (value v) {  
    Cell t, x;  
    x := new Cell();  
    x.data := v;  
    do {  
        <t := S; >  
        x.next := t;  
    } while ( $\neg$ CASthis(&S, t, x));  
}  
  
value pop () {  
    Cell t, x;  
    do {  
        <t := S; Linthis(t = null);>  
        if(t = null)  
            return EMPTY;  
        x := t.next;  
    } while ( $\neg$ CASthis(&S, t, x));  
    return t.data;  
}
```

Linearization points
are marked

$$\begin{aligned} \text{Abs_push}(e) &\stackrel{\text{def}}{=} \langle \text{Abs} := e \cdot \text{Abs}; \text{AbsResult} := e; \rangle \\ \text{Abs_pop}() &\stackrel{\text{def}}{=} \langle \text{case } (\text{Abs}) \\ &\quad | \epsilon \implies \text{AbsResult} := \text{EMPTY}; \\ &\quad | v \cdot A \implies \{ \text{Abs} := A; \text{AbsResult} := v; \} \rangle \end{aligned}$$

Treiber Stack in RGSep

Actions

$$\&S \mapsto y * \&\text{Abs} \mapsto A \rightsquigarrow \&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \quad (\text{Push})$$

$$\&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \rightsquigarrow \&S \mapsto y * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto A \quad (\text{Pop})$$

Treiber Stack in RGSep

Actions

$$\&S \mapsto y * \&\text{Abs} \mapsto A \rightsquigarrow \&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \quad (\text{Push})$$

$$\&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \rightsquigarrow \&S \mapsto y * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto A \quad (\text{Pop})$$

Inductive Representation of a List containing A

$$\begin{aligned} \text{lseg}(x, y, A) &\stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge \text{emp}) \\ &\vee (x \neq y \wedge \exists v z B. x \mapsto \text{Cell}(v, z) * \text{lseg}(z, y, B) \wedge A = v \cdot B) \end{aligned}$$

Treiber Stack in RGSep

Actions

$$\&S \mapsto y * \&\text{Abs} \mapsto A \rightsquigarrow \&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \quad (\text{Push})$$

$$\&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \rightsquigarrow \&S \mapsto y * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto A \quad (\text{Pop})$$

Inductive Representation of a List containing A

$$\begin{aligned} \text{lseg}(x, y, A) &\stackrel{\text{def}}{=} (x = y \wedge A = \epsilon \wedge \text{emp}) \\ &\vee (x \neq y \wedge \exists v z B. x \mapsto \text{Cell}(v, z) * \text{lseg}(z, y, B) \wedge A = v \cdot B) \end{aligned}$$

Abstraction Invariant (a.k.a. Simulation Relation)

$$StackInv \stackrel{\text{def}}{=} \boxed{\exists x A. \&S \mapsto x * \&\text{Abs} \mapsto A * \text{lseg}(x, \text{null}, A) * \text{true}}$$

Treiber Stack in RGSep

```

void push (value v) {
    Cell t, x;
    {AbsResult  $\xrightarrow{s}$  undef * StackInv}
    x := new Cell();
    x.data := v;
    {AbsResult  $\xrightarrow{s}$  undef
     { * x $\mapsto$ Cell(v,_) * StackInv}}
    do {
        {AbsResult  $\xrightarrow{s}$  undef
         { * x $\mapsto$ Cell(v,_) * StackInv}}
        {t := S; Linthis(t = null);}
        { (t=null  $\wedge$  AbsResult  $\xrightarrow{s}$  EMPTY * StackInv)
          {  $\vee$  ( $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)) }
          if(t = null) return EMPTY;
          { $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)}}
        x := t.next;
        {AbsResult  $\xrightarrow{s}$  undef * K(x)}
    } while( $\neg$ CASthis(&S, t, x));
    {AbsResult  $\xrightarrow{s}$  v * StackInv}
}

```

&S \mapsto y * &Abs \mapsto A \rightsquigarrow &S \mapsto x * x \mapsto Cell(v, y) * &Abs \mapsto v·A (Push)

&S \mapsto x * x \mapsto Cell(v, y) * &Abs \mapsto v·A \rightsquigarrow &S \mapsto y * x \mapsto Cell(v, y) * &Abs \mapsto A (Pop)

```

value pop () {
    Cell t, x, temp;
    {AbsResult  $\xrightarrow{s}$  undef * StackInv}
    do {
        {t := S; Linthis(t = null);}
        { (t=null  $\wedge$  AbsResult  $\xrightarrow{s}$  EMPTY * StackInv)
          {  $\vee$  ( $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)) }
          if(t = null) return EMPTY;
          { $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)}}
        x := t.next;
        {AbsResult  $\xrightarrow{s}$  undef * K(x)}
    } while( $\neg$ CASthis(&S, t, x));
    { $\exists v$ . AbsResult  $\xrightarrow{s}$  v
     {  $\exists x A$ . &Abs $\mapsto$ A * &S $\mapsto$ x
       { * lseg(x, null, A) * x $\mapsto$ Cell(v,_) * true}}
     temp := t.data;
     { $\exists v$ . AbsResult  $\xrightarrow{s}$  temp * StackInv}
     return temp;
    }
}

```

$$StackInv \stackrel{\text{def}}{=} \exists x A. \&S \mapsto x * \&Abs \mapsto A * lseg(x, \text{null}, A) * true$$

$$K(y) \stackrel{\text{def}}{=} \left(\begin{array}{l} \exists x v A B. \&Abs \mapsto A \cdot v \cdot B * \&S \mapsto x * lseg(x, t, A) \\ * t \mapsto Cell(v, y) * lseg(y, \text{null}, B) * true \end{array} \right) \\ \vee (\exists x A. \&Abs \mapsto A * \&S \mapsto x * lseg(x, \text{null}, A) * t \mapsto Cell(., .) * true)$$

Treiber Stack in RGSep

```

void push (value v) {
    Cell t, x;
    {AbsResult  $\xrightarrow{s}$  undef * StackInv}
    x := new Cell();
    x.data := v;
    {AbsResult  $\xrightarrow{s}$  undef
     { * x $\mapsto$ Cell(v,_) * StackInv}}
    do {
        {AbsResult  $\xrightarrow{s}$  undef
         { * x $\mapsto$ Cell(v,_) * StackInv}}
        {t := S; Linthis(t = null);}
        { (t=null  $\wedge$  AbsResult  $\xrightarrow{s}$  EMPTY * StackInv)
          {  $\vee$  ( $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)) }
          if(t = null) return EMPTY;
          { $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)}}
        x := t.next;
        {AbsResult  $\xrightarrow{s}$  undef * K(x)}
    } while( $\neg$ CASthis(&S, t, x));
    {AbsResult  $\xrightarrow{s}$  v * StackInv}
}

```

(Push)

$$\&S \mapsto y * \&\text{Abs} \mapsto A \rightsquigarrow \&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \quad (\text{Push})$$

$$\&S \mapsto x * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto v \cdot A \rightsquigarrow \&S \mapsto y * x \mapsto \text{Cell}(v, y) * \&\text{Abs} \mapsto A \quad (\text{Pop})$$

```

value pop () {
    Cell t, x, temp;
    {AbsResult  $\xrightarrow{s}$  undef * StackInv}
    do {
        {t := S; Linthis(t = null);}
        { (t=null  $\wedge$  AbsResult  $\xrightarrow{s}$  EMPTY * StackInv)
          {  $\vee$  ( $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)) }
          if(t = null) return EMPTY;
          { $\exists x$ . AbsResult  $\xrightarrow{s}$  undef * K(x)}}
        x := t.next;
        {AbsResult  $\xrightarrow{s}$  undef * K(x)}
    } while( $\neg$ CASthis(&S, t, x));
    { $\exists v$ . AbsResult  $\xrightarrow{s}$  v
     {  $\exists x A$ . &Abs  $\mapsto A * \&S \mapsto x$ 
       { * lseg(x, null, A) * x  $\mapsto$  Cell(v,_) * true } }
      temp := t.data;
      { $\exists v$ . AbsResult  $\xrightarrow{s}$  temp * StackInv}
      return temp;
    }
}

```

Check Stability: Exercise :)

$$StackInv \stackrel{\text{def}}{=} \exists x A. \&S \mapsto x * \&\text{Abs} \mapsto A * \text{lseg}(x, \text{null}, A) * \text{true}$$

$$K(y) \stackrel{\text{def}}{=} \left(\begin{array}{l} \exists x v A B. \&\text{Abs} \mapsto A \cdot v \cdot B * \&S \mapsto x * \text{lseg}(x, t, A) \\ * t \mapsto \text{Cell}(v, y) * \text{lseg}(y, \text{null}, B) * \text{true} \end{array} \right) \\ \vee (\exists x A. \&Abs \mapsto A * \&S \mapsto x * \text{lseg}(x, \text{null}, A) * t \mapsto \text{Cell}(_, _) * \text{true})$$

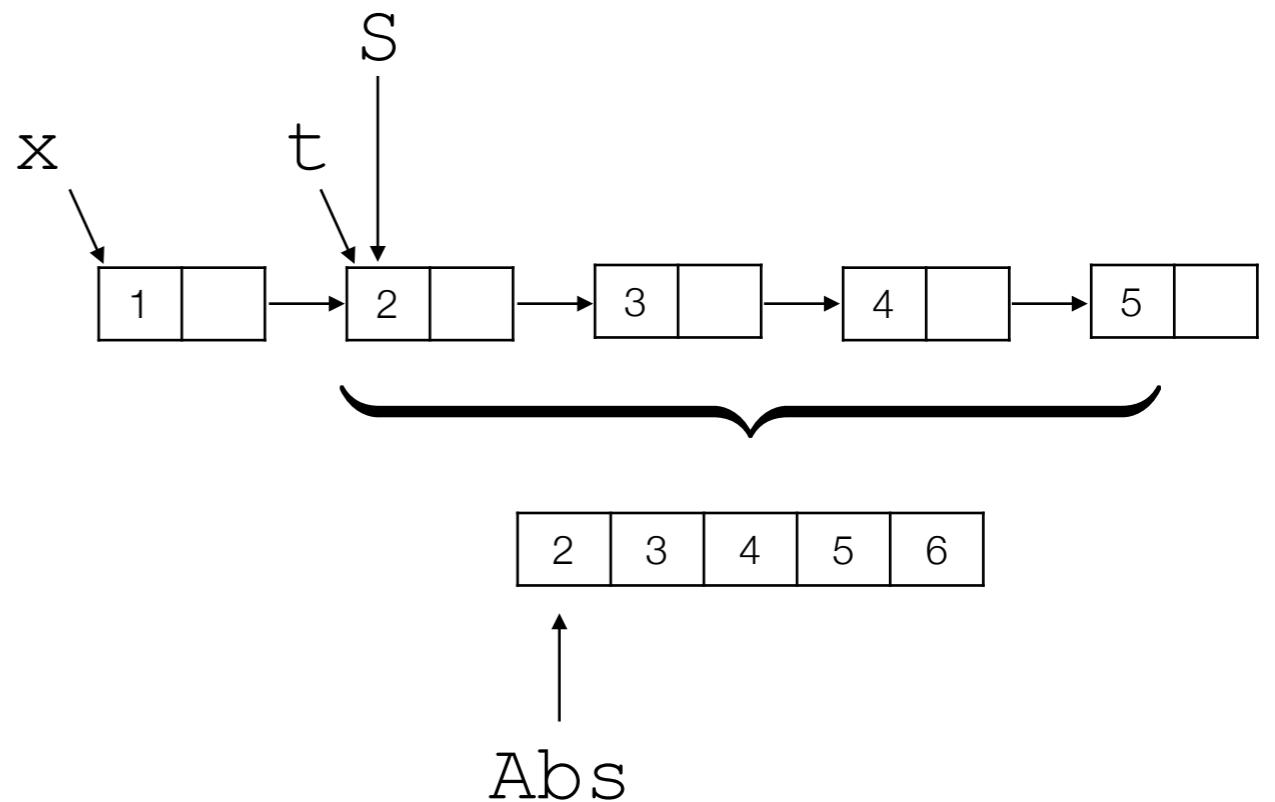
Push

Linearization point of Push

$$\left\{ \begin{array}{l} \text{AbsResult} \xrightarrow{s} \text{undef} \\ * \mathbf{x} \mapsto \text{Cell}(\mathbf{v}, \mathbf{t}) * \text{StackInv} \end{array} \right\}$$

$\mathbf{b} := \text{CAS}_{\text{this}}(\&\mathbf{S}, \mathbf{t}, \mathbf{x});$

$$\left\{ \begin{array}{l} (\mathbf{b} \wedge \text{AbsResult} \xrightarrow{s} \mathbf{v} * \text{StackInv}) \\ \vee \left(\neg \mathbf{b} \wedge \text{AbsResult} \xrightarrow{s} \text{undef} \right) \\ * \mathbf{x} \mapsto \text{Cell}(\mathbf{v}, \mathbf{t}) * \text{StackInv} \end{array} \right\}$$



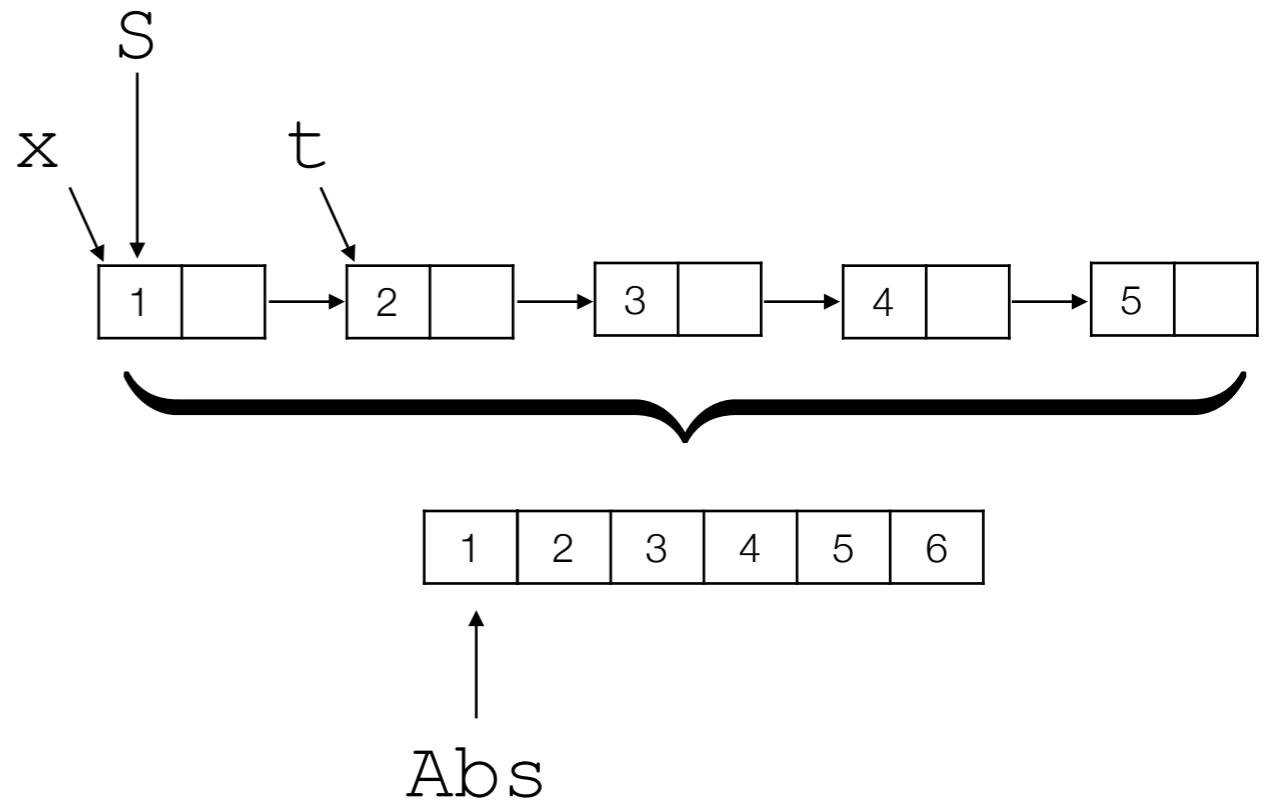
Push

Linearization point of Push

$$\left\{ \begin{array}{l} \text{AbsResult} \xrightarrow{s} \text{undef} \\ * \mathbf{x} \mapsto \text{Cell}(\mathbf{v}, \mathbf{t}) * \text{StackInv} \end{array} \right\}$$

$\mathbf{b} := \text{CAS}_{\text{this}}(\&\mathbf{S}, \mathbf{t}, \mathbf{x});$

$$\left\{ \begin{array}{l} (\mathbf{b} \wedge \text{AbsResult} \xrightarrow{s} \mathbf{v} * \text{StackInv}) \\ \vee \left(\neg \mathbf{b} \wedge \text{AbsResult} \xrightarrow{s} \text{undef} \right) \\ * \mathbf{x} \mapsto \text{Cell}(\mathbf{v}, \mathbf{t}) * \text{StackInv} \end{array} \right\}$$



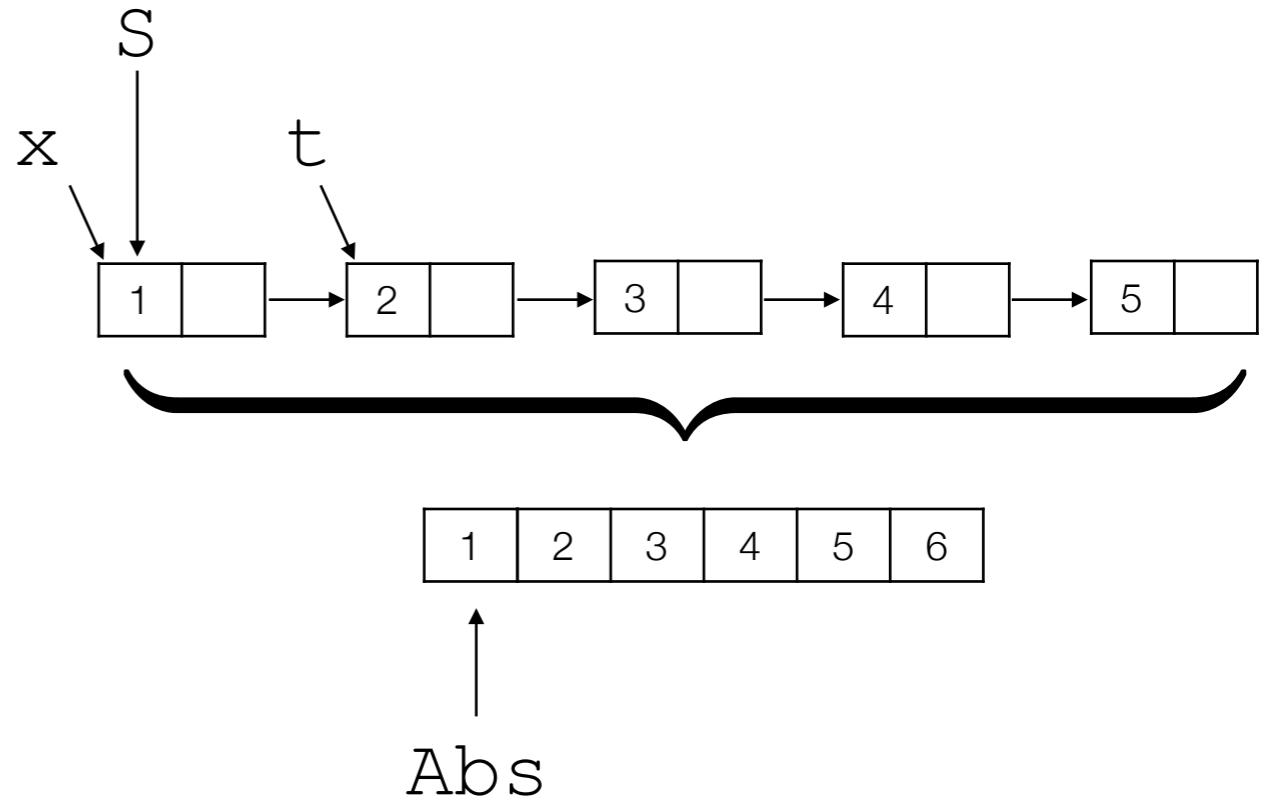
Push

Linearization point of Push

$$\left\{ \begin{array}{l} \text{AbsResult} \xrightarrow{s} \text{undef} \\ * \text{x} \mapsto \text{Cell}(\text{v}, \text{t}) * \text{StackInv} \end{array} \right\}$$

$\text{b} := \text{CAS}_{\text{this}}(\&\text{S}, \text{t}, \text{x});$

$$\left\{ \begin{array}{l} (\text{b} \wedge \text{AbsResult} \xrightarrow{s} \text{v} * \text{StackInv}) \\ \vee \left(\neg \text{b} \wedge \text{AbsResult} \xrightarrow{s} \text{undef} \right) \\ * \text{x} \mapsto \text{Cell}(\text{v}, \text{t}) * \text{StackInv} \end{array} \right\}$$



Successful CAS

$$\left\{ \text{AbsResult} \xrightarrow{s} \text{undef} * \text{x} \mapsto \text{Cell}(\text{v}, \text{t}) * \boxed{\exists A. \&S \mapsto \text{t} * \&\text{Abs} \mapsto A * \text{lseg}(\text{t}, \text{null}, A) * \text{true}} \right\}$$

$$\left\{ \text{AbsResult} \xrightarrow{s} \text{undef} * \text{x} \mapsto \text{Cell}(\text{v}, \text{t}) * \boxed{\&S \mapsto \text{t} * \&\text{Abs} \mapsto A * \text{lseg}(\text{t}, \text{null}, A) * \text{true}} \right\}$$

$\langle [\&\text{S}] := \text{x}; \text{Abs_push}(\text{v}) \rangle$

$$\left\{ \text{AbsResult} \xrightarrow{s} \text{v} * \boxed{\&S \mapsto \text{x} * \&\text{Abs} \mapsto \text{v} \cdot A * \text{x} \mapsto \text{Cell}(\text{v}, \text{t}) * \text{lseg}(\text{t}, \text{null}, A) * \text{true}} \right\}$$

$$\left\{ \text{AbsResult} \xrightarrow{s} \text{v} * \boxed{\exists A. \&S \mapsto \text{x} * \&\text{Abs} \mapsto A * \text{lseg}(\text{x}, \text{null}, A) * \text{true}} \right\}$$

Pop

CAS Linearization point of Pop

$$\left\{ \text{AbsResult} \xrightarrow{s} \text{undef} * K(\mathbf{x}) \right\}$$

$b := \text{CAS}_{\text{this}}(\&S, t, x);$

$$\left\{ \begin{array}{l} \left(b \wedge \exists v. \text{AbsResult} \xrightarrow{s} v * \boxed{\begin{array}{l} \exists x A. \&\text{Abs} \mapsto A * \&S \mapsto x * \text{lseg}(x, \text{null}, A) \\ * \text{phead} \mapsto \text{Cell}(v, _) * \text{true} \end{array}} \right) \\ \vee (\neg b \wedge \text{AbsResult} \xrightarrow{s} \text{undef} * K(\mathbf{x})) \end{array} \right\}$$

$$K(y) \stackrel{\text{def}}{=} \left(\begin{array}{l} \left(\exists x v A B. \&\text{Abs} \mapsto A \cdot v \cdot B * \&S \mapsto x * \text{lseg}(x, \text{phead}, A) \right) \\ * \text{phead} \mapsto \text{Cell}(v, y) * \text{lseg}(y, \text{null}, B) * \text{true} \end{array} \right) \\ \vee (\exists x A. \&\text{Abs} \mapsto A * \&S \mapsto x * \text{lseg}(x, \text{null}, A) * \text{phead} \mapsto \text{Cell}(_, _) * \text{true})$$

Pop

CAS Linearization point of Pop

$$\left\{ \text{AbsResult} \xrightarrow{s} \text{undef} * K(x) \right\}$$

$b := \text{CAS}_{\text{this}}(\&S, t, x);$

$$\left\{ \begin{array}{l} \left(b \wedge \exists v. \text{AbsResult} \xrightarrow{s} v * \boxed{\begin{array}{l} \exists x A. \&\text{Abs} \mapsto A * \&S \mapsto x * \text{lseg}(x, \text{null}, A) \\ * \text{phead} \mapsto \text{Cell}(v, _) * \text{true} \end{array}} \right) \\ \vee (\neg b \wedge \text{AbsResult} \xrightarrow{s} \text{undef} * K(x)) \end{array} \right\}$$

$$K(y) \stackrel{\text{def}}{=} \left(\begin{array}{l} \exists x v A B. \&\text{Abs} \mapsto A \cdot v \cdot B * \&S \mapsto x * \text{lseg}(x, \text{phead}, A) \\ * \text{phead} \mapsto \text{Cell}(v, y) * \text{lseg}(y, \text{null}, B) * \text{true} \end{array} \right) \\ \vee (\exists x A. \&\text{Abs} \mapsto A * \&S \mapsto x * \text{lseg}(x, \text{null}, A) * \text{phead} \mapsto \text{Cell}(_, _) * \text{true})$$

Read Linearization point of Pop

$$\left\{ \begin{array}{l} \text{AbsResult} \xrightarrow{s} \text{undef} \\ * \text{StackInv} \end{array} \right\} - \left\{ \begin{array}{l} (t = \text{null} \wedge \text{AbsResult} \xrightarrow{s} \text{EMPTY} * \text{StackInv}) \\ \vee (\exists x. \text{AbsResult} \xrightarrow{s} \text{undef} * K(x)) \end{array} \right\}$$

Lock Coupling List

Lock Coupling Set

```
locate(e) {  
    local p, c;  
    p := Head;  
    lock(p);  
    c := p.next;  
    while (c.value < e) {  
        lock(c);  
        unlock(p);  
        p := c;  
        c := p.next;  
        lock(c);  
    }  
    return(p, c);  
}
```

```
remove(e) {  
    local x, y, z;  
    (x, y) := locate(e);  
    if (y.value = e) {  
        lock(y);  
        z := y.next;  
        x.next := z;  
        unlock(x);  
        dispose(y);  
    } else {  
        unlock(x);  
    }  
}
```

```
add(e) {  
    local x, y, z;  
    (x, z) := locate(e);  
    if (z.value ≠ e) {  
        y := new Node();  
        y.lock := 0;  
        y.value := e;  
        y.next := z;  
        x.next := y;  
    }  
    unlock(x);  
}
```

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

x points to a node locked by s or s is 0,
with value v, and followed by Node y

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$

x points to a node locked by a real thread t

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$

$$U(x, v, y) = N_0(x, v, y)$$

x points to an unlocked node

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$

$$U(x, v, y) = N_0(x, v, y)$$

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$

$$U(x, v, y) = N_0(x, v, y)$$

Lock and Unlock Actions

$$t \in T \wedge U(x, v, b) \rightsquigarrow L_t(x, v, n) \quad t \in T \wedge L_t(x, v, b) \rightsquigarrow U(x, v, n)$$

Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\text{lock} = s, .\text{value} = v, .\text{next} = y\}$$

$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$

$$U(x, v, y) = N_0(x, v, y)$$

Lock and Unlock Actions

$$t \in T \wedge U(x, v, b) \rightsquigarrow L_t(x, v, n) \quad t \in T \wedge L_t(x, v, b) \rightsquigarrow U(x, v, n)$$

Rules

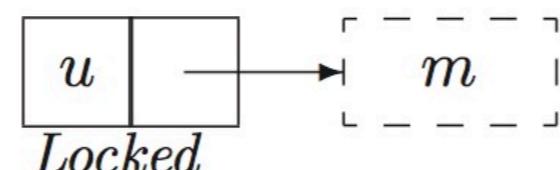
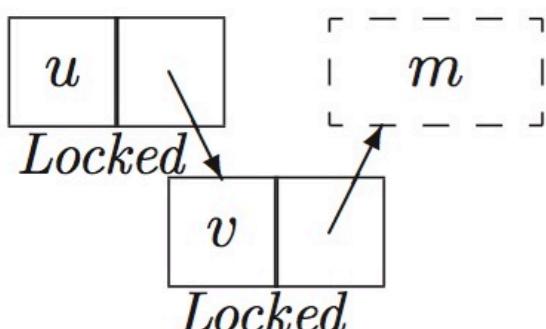
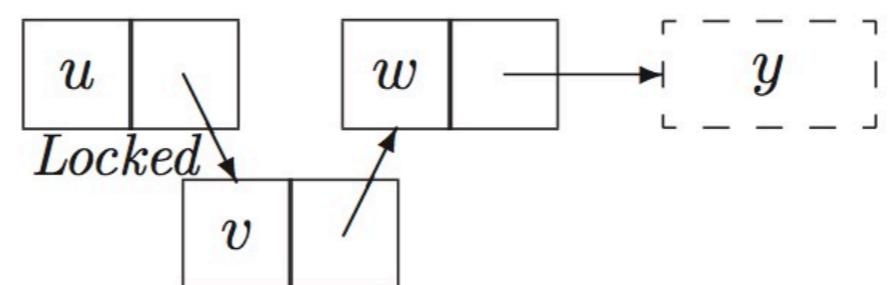
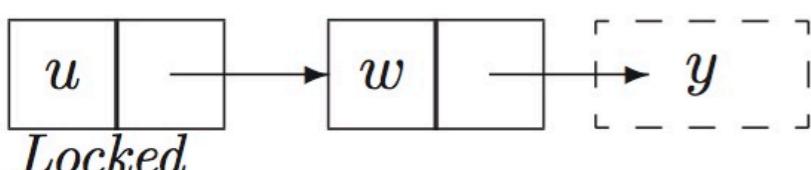
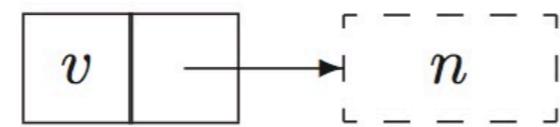
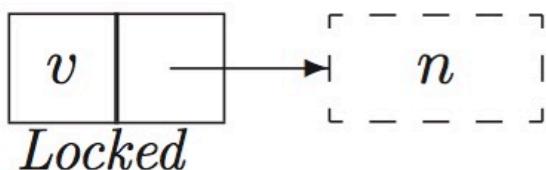
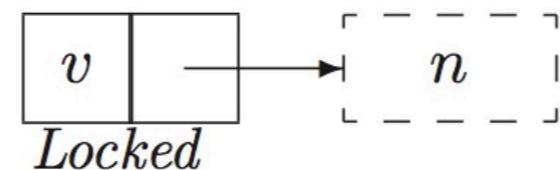
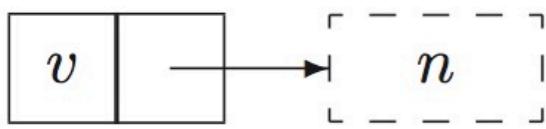
$$\frac{\begin{array}{c} P \text{ stable for } R \quad Q \text{ stable for } R \\ P \Rightarrow N_-(p, n, n) * F \\ L_{tid}(p, v, n) * F \Rightarrow Q \end{array}}{R, G \vdash \{\boxed{P}\} \text{ lock}(p) \{\boxed{Q}\}}$$

$$\frac{\begin{array}{c} P \text{ stable for } R \quad Q \text{ stable for } R \\ P \Rightarrow L_{tid}(p, n, n) * F \\ U(p, v, n) * F \Rightarrow Q \end{array}}{R, G \vdash \{\boxed{P}\} \text{ unlock}(p) \{\boxed{Q}\}}$$

Lock Coupling Set

$$(t \in T) \wedge (u < v < w) \wedge (L_t(x, u, n) * N_s(n, w, y)) \rightsquigarrow L_t(x, u, m) * U(m, v, n) * N_s(n, w, y) \quad (\text{Insert})$$

$$(t \in T) \wedge (v < \infty) \wedge (L_t(x, u, n) * L_t(n, v, m)) \rightsquigarrow L_t(x, u, m) \quad (\text{Remove})$$



Locate

```

locate(e) {
    local p, c, t;
    { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$   $\wedge -\infty < e$ } 
    p := Head;
    { $\exists ZB. ls(\text{Head}, \epsilon, p) * N(p, -\infty, Z)$   $\wedge -\infty < e$ } 
    { $* ls(Z, B, \text{nil}) * s(-\infty \cdot B)$ } 
    lock(p);
    { $\exists Z. \exists B. ls(\text{Head}, \epsilon, p) * L(p, -\infty, Z)$   $\wedge -\infty < e$ } 
    { $* ls(Z, B, \text{nil}) * s(-\infty \cdot B)$ } 
    {c := p.next;}
    { $\exists B. ls(\text{Head}, \epsilon, p) * L(p, -\infty, c)$   $\wedge -\infty < e$ } 
    { $* ls(c, B, \text{nil}) * s(-\infty \cdot B)$ } 
    {t := c.value;}
    { $\exists u. \exists ABZ. ls(\text{Head}, A, p) * L(p, u, c)$   $\wedge u < e$ } 
    { $* N(c, t, Z) * ls(c, B, \text{nil}) * s(A \cdot u \cdot t \cdot B)$ } 
    while (t < e) {
        { $\exists u. \exists ABZ. ls(\text{Head}, A, p) * L(p, u, c)$   $\wedge u < e \wedge t < e$ } 
        { $* N(c, t, Z) * ls(c, B, \text{nil}) * s(A \cdot u \cdot t \cdot B)$ } 
        lock(c);
        { $\exists uZ. \exists AB. ls(\text{Head}, A, p) * L(p, u, c)$   $\wedge t < e$ } 
        { $* L(c, t, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot t \cdot B)$ } 
        unlock(p);
        { $\exists Z. \exists AB. ls(\text{Head}, A, c) * L(c, t, Z)$   $\wedge t < e$ } 
        { $* ls(Z, B, \text{nil}) * s(A \cdot t \cdot B)$ } 
        p := c;
        { $\exists uZ. \exists AB. ls(\text{Head}, A, p) * L(p, u, Z)$   $\wedge u < e$ } 
        { $* ls(Z, B, \text{nil}) * s(A \cdot u \cdot B)$ } 
        {c := p.next;}
        { $\exists u. \exists AB. ls(\text{Head}, A, p) * L(p, u, c)$   $\wedge u < e$ } 
        { $* ls(c, B, \text{nil}) * s(A \cdot u \cdot B)$ } 
        {t := c.value;}
        { $\exists u. \exists ABZ. ls(\text{Head}, A, p) * L(p, u, c)$   $\wedge u < e$ } 
        { $* N(c, t, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot t \cdot B)$ } 
    }
    { $\exists uv. \exists ABZ. ls(\text{Head}, A, p) * L(p, u, c)$   $\wedge u < e \wedge e \leq v$ } 
    { $* N(c, v, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B)$ } 
    return (p, c);
}

```

Add

```

add(e) { local x,y,z,t;
{ $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }  $\wedge -\infty < e$ }
(x,z) := locate(e);
{ $\exists uv. \left\{ \begin{array}{l} \exists ZAB. ls(\text{Head}, A, x) * L(x, u, z) * N(z, v, Z) \\ * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \end{array} \right\} \wedge u < e \wedge e \leq v$ }
<t = z.value;> if(t  $\neq$  e) {
{ $\exists uv. \left\{ \begin{array}{l} \exists ZAB. ls(\text{Head}, A, x) * L(x, u, z) * N(z, v, Z) \\ * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \end{array} \right\} \wedge u < e \wedge e < v$ }
y = cons(0, e, z);
{ $\exists uv. \left\{ \begin{array}{l} \exists ZAB. ls(\text{Head}, A, x) * L(x, u, z) * N(z, v, Z) \\ * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \end{array} \right\} * U(y, e, z) \wedge u < e \wedge e < v$ }
<x.next = y;>
{ $\exists uv. \left\{ \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * N(y, e, Z) * ls(Z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B) \right\}$ }
}
unlock(x);
{ $\exists v. \left\{ \exists A. ls(\text{Head}, A, \text{nil}) * s(A) \right\}$ }
}

```

Remove

```

remove(e) { local x, y, z, t;
  { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }  $\wedge -\infty < e \wedge e < +\infty$ 
  (x, y) = locate(e);
  { $\exists uv. \begin{cases} \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * N(y, v, Z) \\ * ls(Z, B, \text{nil}) * s(A \cdot u \cdot v \cdot B) \end{cases}$ }  $\wedge u < e \wedge e \leq v \wedge e < +\infty$ 
  {t = y.value; } if(t = e) {
    { $\exists u. \begin{cases} \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * N(y, e, Z) \\ * ls(Z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B) \end{cases}$ }  $\wedge e < +\infty$ 
    lock(y);
    { $\exists u. \begin{cases} \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * L(y, e, Z) \\ * ls(Z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B) \end{cases}$ }  $\wedge e < +\infty$ 
    {z := y.next; }
    { $\exists u. \begin{cases} \exists AB. ls(\text{Head}, A, x) * L(x, u, y) * L(y, e, z) \\ * ls(z, B, \text{nil}) * s(A \cdot u \cdot e \cdot B) \end{cases}$ }  $\wedge e < +\infty$ 
    {x.next := z; }
    { $\exists u. \exists AB. ls(\text{Head}, A, x) * L(x, u, z) * ls(z, B, \text{nil}) * s(A \cdot u \cdot B) * L(y, e, z)$ }
    unlock(x);
    { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ } * L(y, e, z)
    dispose(y);
  } else { { $\exists u. \exists ZAB. ls(\text{Head}, A, x) * L(x, u, y) * ls(y, B, \text{nil}) * s(A \cdot u \cdot B)$ } }
    unlock(x);
    { $\exists A. ls(\text{Head}, A, \text{nil}) * s(A)$ }
}

```

.... to be continued

- ▶ Automatization
 - ▶ Symbolic Execution of SL
 - ▶ Shape Analysis on SL
 - ▶ Shape-Value Abstraction for Linearizability
 - ▶ RGSep Action Inference
 - ▶ Automatically Proving Linearizability



Monitoring Serializability

Conflict Serializability (recap)

- ▶ We need to inspect the implementation of the library
 - ▶ In a transaction these are writes and reads to different registers
- ▶ Specification Histories:
 - ▶ Call : beginTx
 - ▶ Return : commitTx
- ▶ Implementation Histories:
 - ▶ Call : beginTx
 - ▶ Return : commitTx
 - ▶ Write : $wr_{p,v}$
 - ▶ Read : $rd_{p,v}$
 - ▶ RMW : $cas_{p,v,w}$
- ▶ Sometimes we need to mention the thread: $(t, wr_{p,v})$

Conflict Serializability (recap)

- ▶ We define a conflict relation $\#$ between operations:
 - ▶ $wr_{p,v} \# rd_{p,w}$
 - ▶ $wr_{p,v} \# wr_{p,w}$
 - ▶ $rd_{p,w} \# wr_{p,v}$
- ▶ Conflict Equivalence:
 - ▶ Minimal equivalence on histories \sim , such that
$$\delta_0 \cdot o_1 \cdot o_2 \cdot \delta_1 \sim \delta_0 \cdot o_2 \cdot o_1 \cdot \delta_1$$
whenever $o_1 \# o_2$
 - ▶ In a nutshell, reordering non-conflicting events renders equivalent histories

Conflict Serializability Monitoring

- ▶ Let's instrument the program to detect Serializability violations at runtime
- ▶ We can record the history of the execution and check when it becomes un-serializable
- ▶ Problem: How much do we need to remember?
 - ▶ Let's try to minimize it

Conflict Serializability Monitoring

- ▶ Actions $\sigma \in \Sigma$: $\text{push}(v)$, ret_{push} , $\text{pop}()$, ret_v ,
 $\text{wr}_{p,v}$, $\text{rd}_{p,v}$, $\text{cas}_{p,v,w}$
- ▶ Events: $\Sigma_t = \{a_t \mid a \in \Sigma, t \in T\}$
- ▶ History: $\delta \in \Sigma_{T^*}^*$ + well formedness conditions
- ▶ Conflict, Dependency and Equivalence of histories:
as before
- ▶ Definition: (Atomicity – a.k.a. Conflict Serializability)
A history δ is (Conflict) Serializable if it has a *conflict equivalent serial history*

Conflict Serializability Monitoring

► Definition: (Conflict Graph)

If operations o_1, o_2, \dots, o_n happen in δ . The conflict graph $CG(\delta)$ of δ is a tuple $CG(\delta) = (V, E, S)$ where:

► $V = (o_1, o_2, \dots, o_n)$,

► $S : V \rightarrow 2^\Sigma$ where $S(o_i)$ is the set of events of o_i in δ

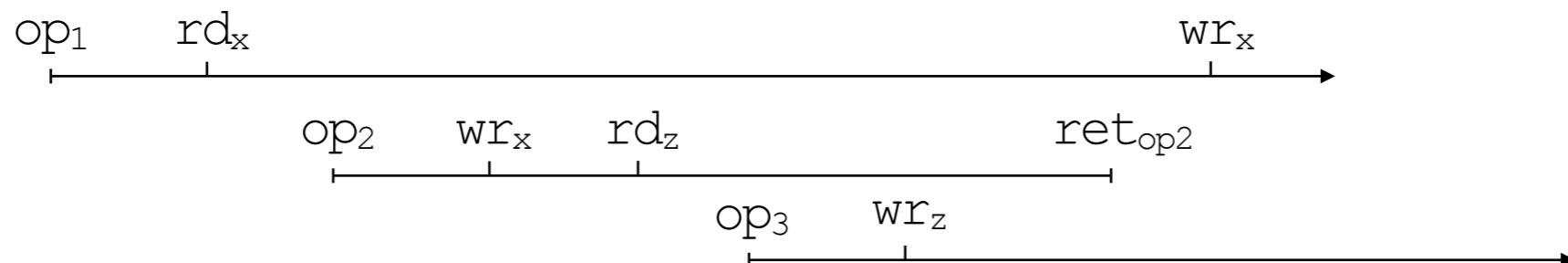
► $o_i \rightarrow o_j$ appears in E if there exists events $a_t \in S(o_i)$ and $a'_{t'} \in S(o_j)$ such that $a \# a'$,
and a appears in δ before a'

► Lemma:

There exists a cycle in δ iff δ is not conflict serializable

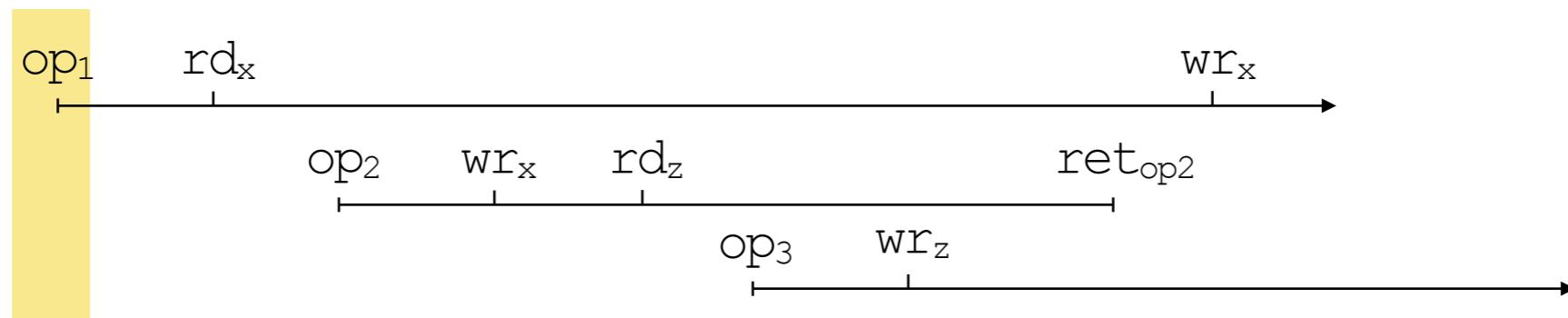
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



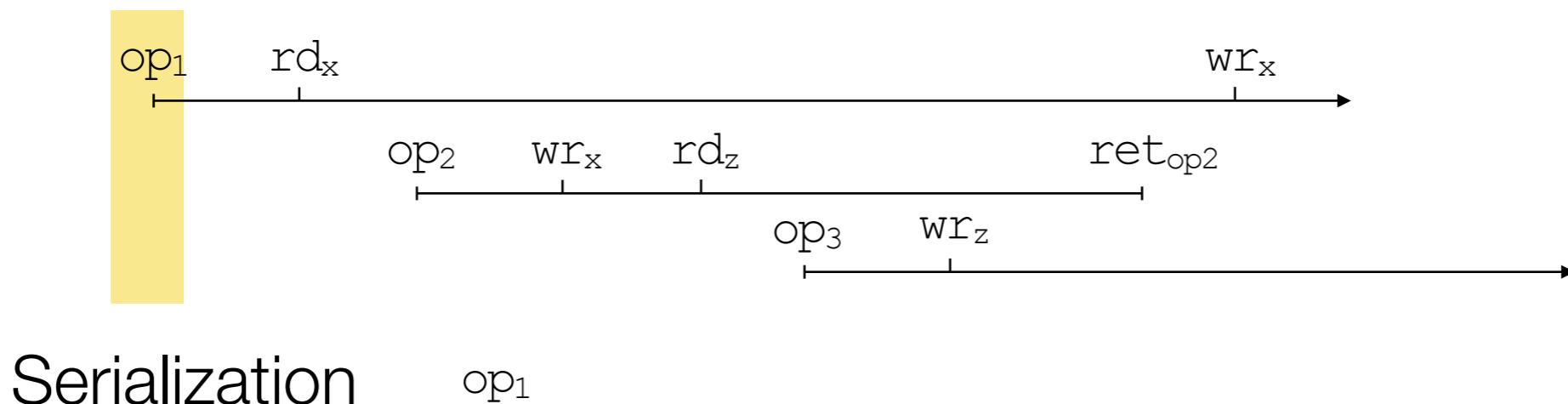
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



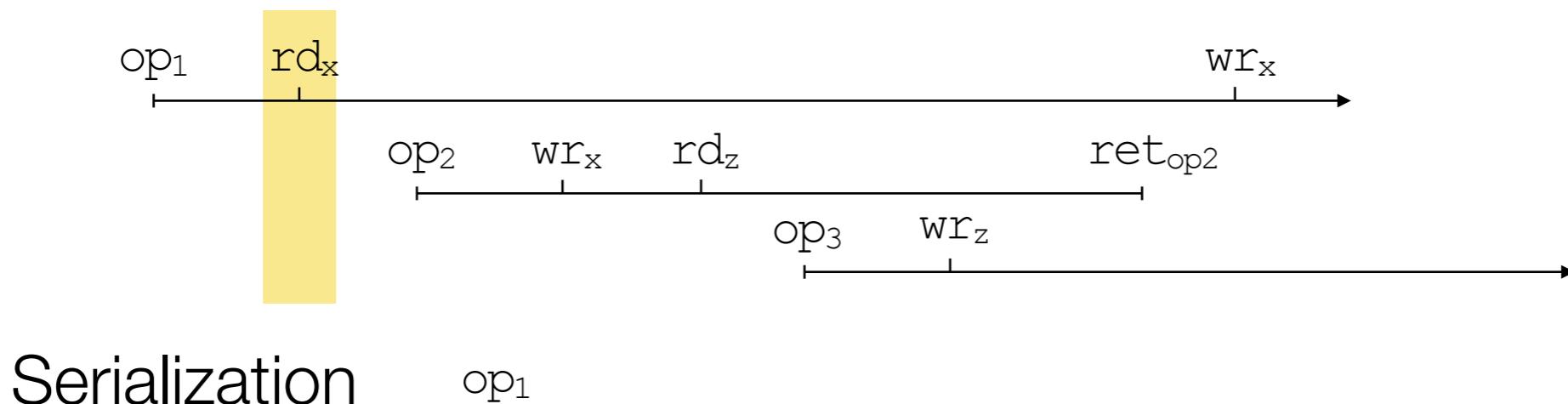
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



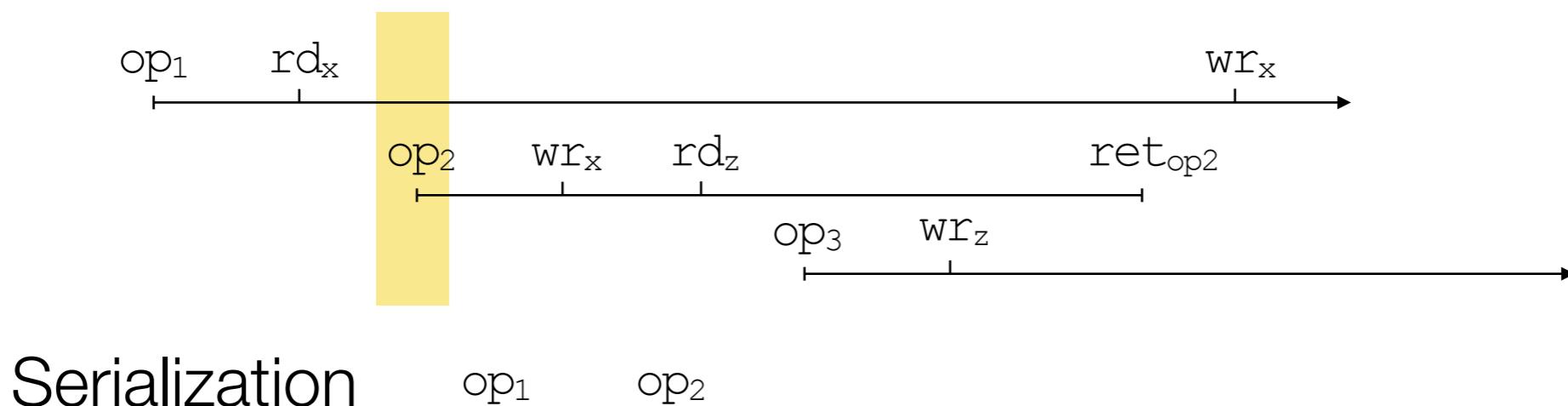
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



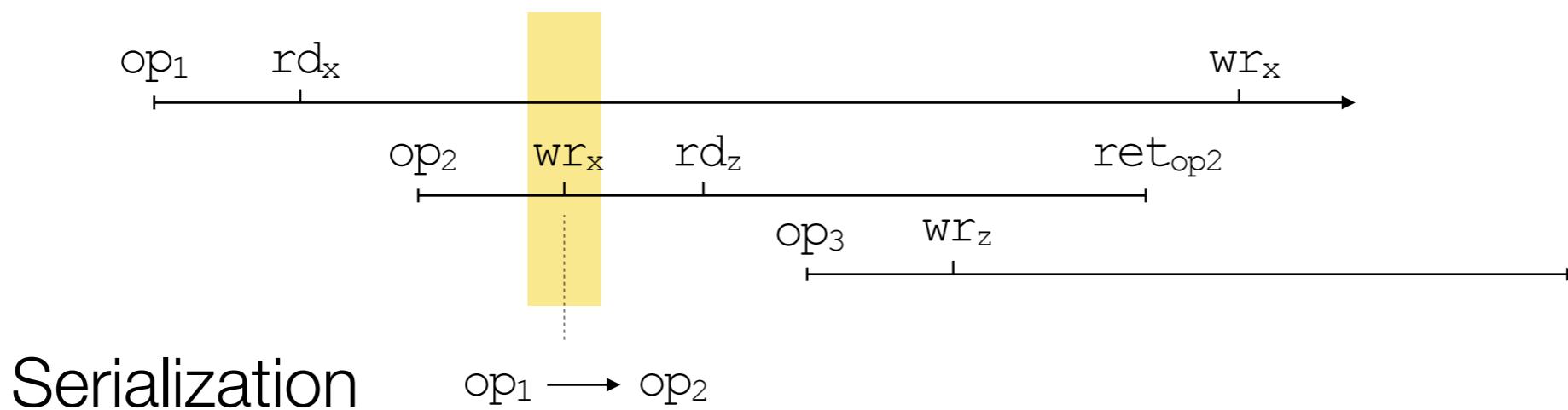
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



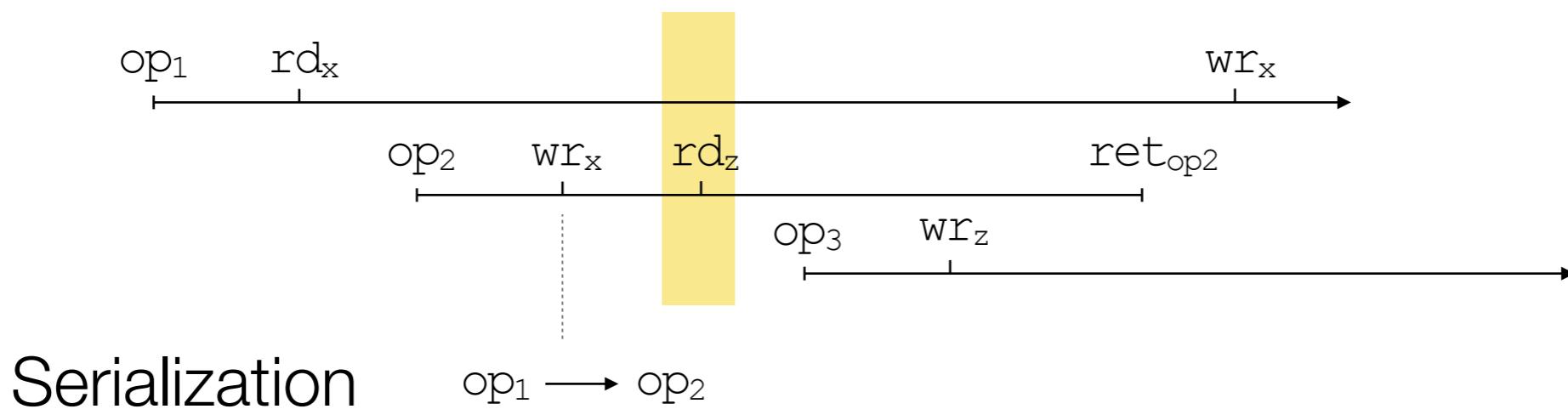
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



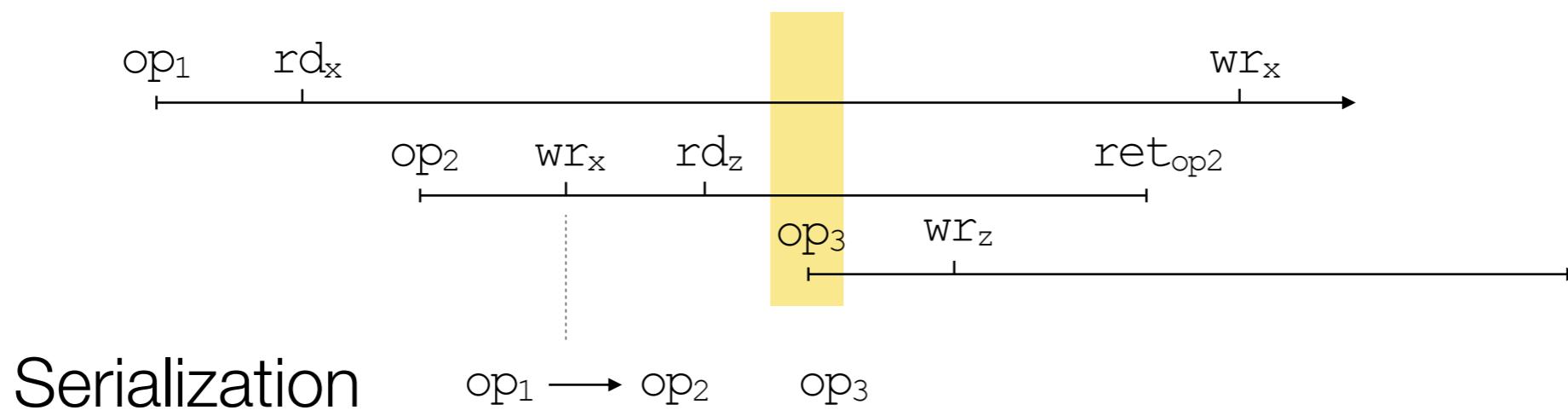
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



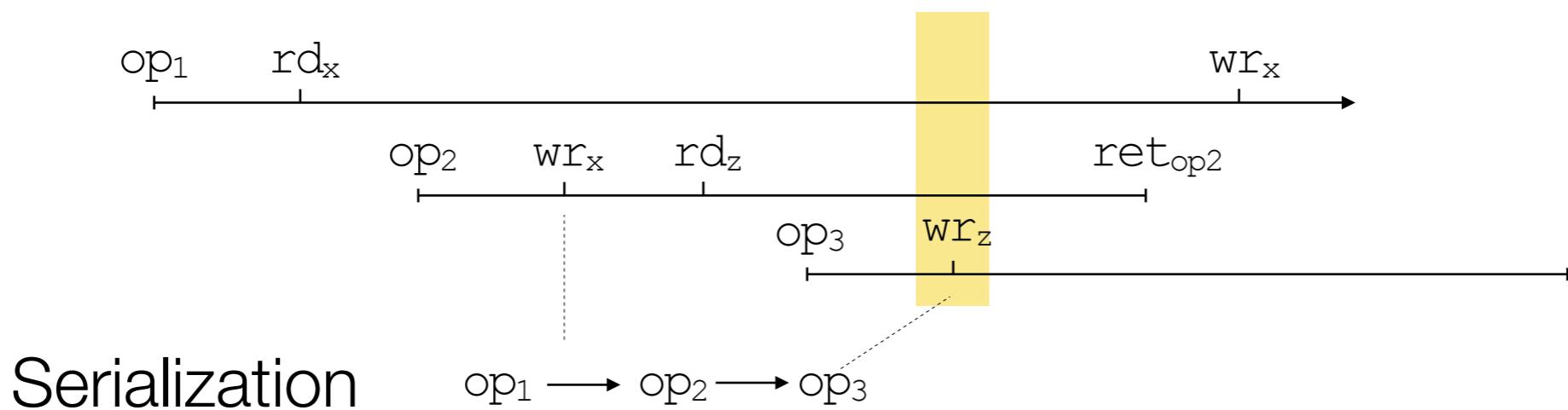
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



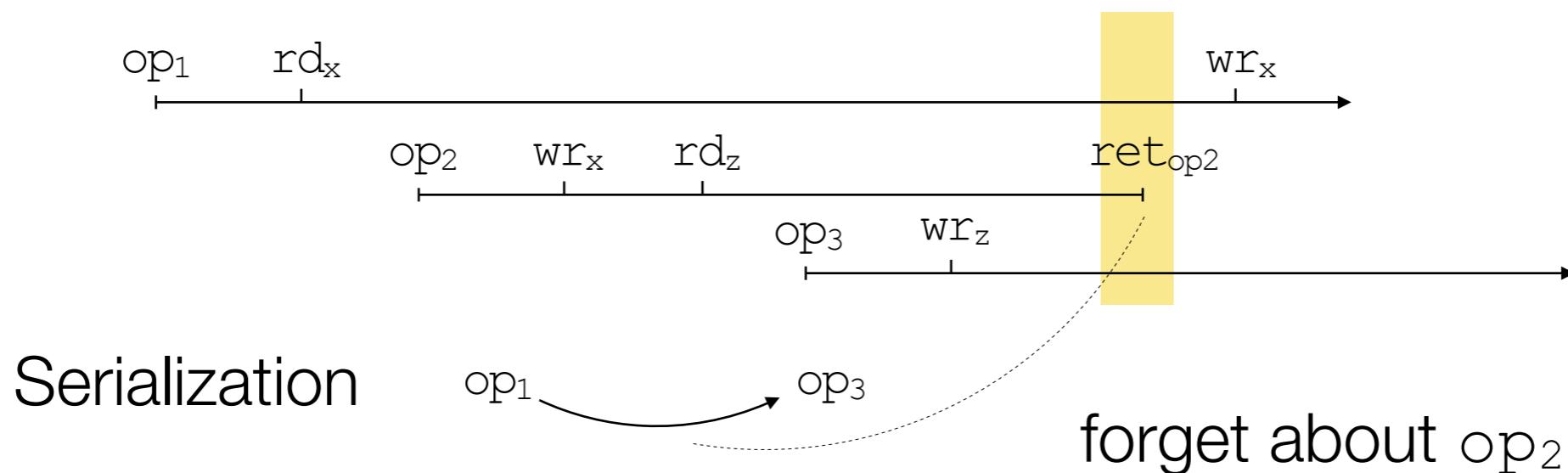
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



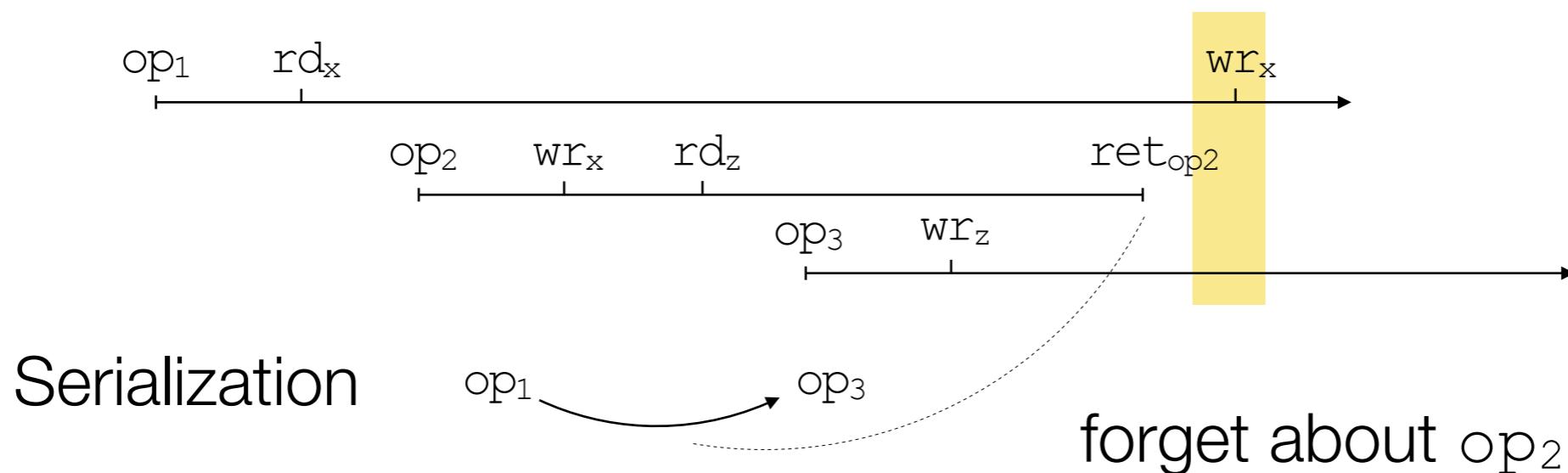
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



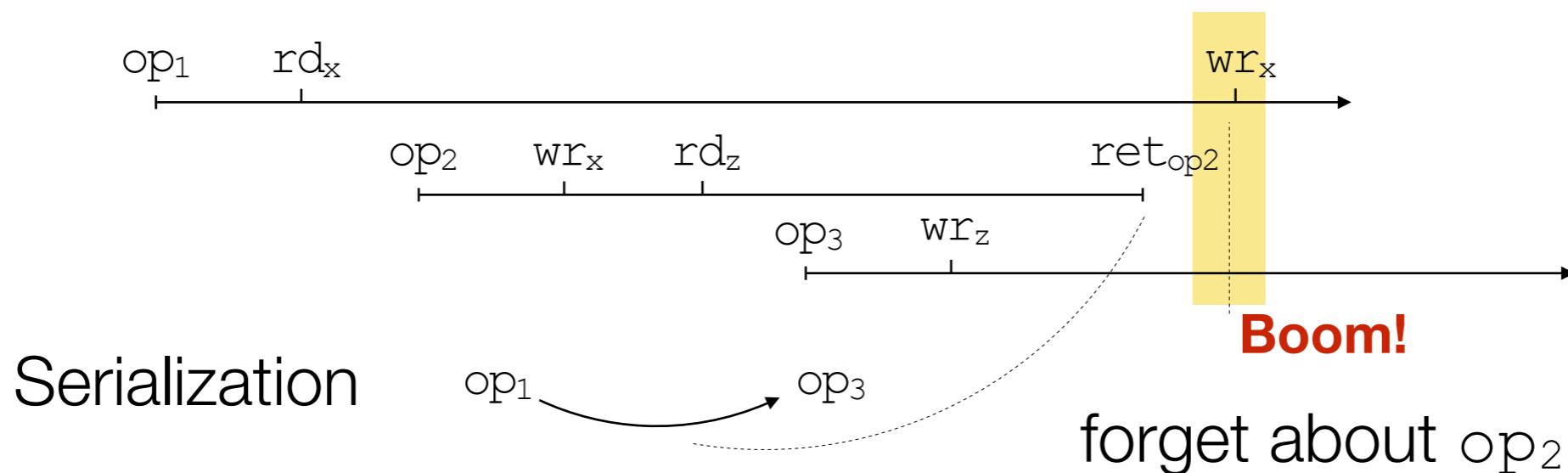
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about erasing operations that have already finished?



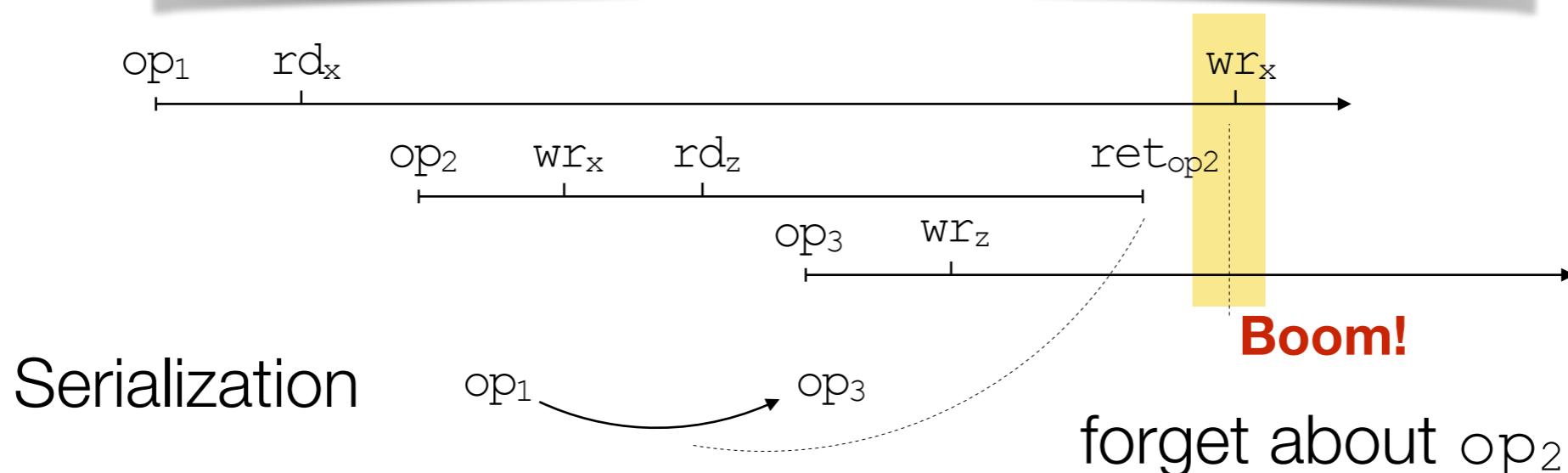
Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-)
 - ▶ What about erasing operations that have already finished?



Conflict Serializability Monitoring

- ▶ So, we could record the Conflict Graph, and check for cycles at every operation:
 - ▶ The graph is unbounded :-(
 - ▶ What about crossing operations that have:
Forget the operations but not the conflicts they induced



Conflict Serializability Monitoring

- ▶ But recording every event of every transaction is too expensive in memory
- ▶ Definition: (Summarized Conflict Graph)
Let $CF(\delta)$ be the conflict graph of δ . We define $SCF(\delta) = (V, E, S, C)$ where:
 1. V contains a node $t o_i$ for each active thread t (we assume executing the operation o_i)
 2. $t o_i \rightarrow t' o_i$ appears in E if there exists $o_i \rightarrow o_i'$ a path in $CG(\delta)$ which contains no other operation in V
 3. $S(t o_i) = S(o_i) \in CG(\delta)$
 4. a) $C(t o_i)$ contains every $t' a$ belonging to $S(o_j)$ of a completed operation $t' o_j$ of an active thread in δ such that there exists $o_i \rightarrow o_i'$ a path in $CG(\delta)$, and
 4. b) $C(t o_i)$ contains every a belonging to $S(o_j)$ of a completed operation $t' o_j$ of an inactive thread in σ such that there exists $o_i \rightarrow o_i'$ a path in $CG(\delta)$
- ▶ Lemma:
 $CG(\delta)$ contains a cycle iff $SCG(\delta')$ contains a cycle for δ' a prefix of δ

Conflict Serializability Monitoring

- ▶ Algorithm: we consider the next event in σ
 - ▶ $t_{\text{call}_{\text{op}}}$: create a vertex t_{op} and set $S(t_{\text{op}}) = \emptyset$ and $C(t_{\text{op}}) = \emptyset$
 - ▶ $t_{\text{return}_{\text{op}}}$: for each $t_1 \text{op}_1$ and $t_2 \text{op}_2$ such that $t_1 \text{op}_1 \rightarrow t_{\text{op}}$, and $t_{\text{op}} \rightarrow t_2 \text{op}_2$ appear in E , remove the edges from E and add $t_1 \text{op}_1 \rightarrow t_2 \text{op}_2$. Set $C(t_1 \text{op}_1) = C(t_1 \text{op}_1) \cup S(t_{\text{op}}) \cup C(t_{\text{op}})$
 - ▶ t_a : Set $S(t_{\text{op}}) = S(t_{\text{op}}) \cup \{t_a\}$. For each $t_1 \text{op}_1$ in V such that $t_2 a_2$ appears in $C(t_1 \text{op}_1) \cup S(t_1 \text{op}_1)$ and $t_2 a_2 \# t_a$, add $t_1 \text{op}_1 \rightarrow t_2 \text{op}_2$ to E . For any action $t_1 a_1$ in $C(t_1 \text{op}_1)$ with $t_1 a_1 \# t_a$ add $t_1 \text{op}_1 \rightarrow t_{\text{op}}$ to E
 - ▶ If a thread t terminates, erase all the labels t from the conflict sets
 - ▶ If at any point there is a *self loop* report a *serializability violation*

Conflict Serializability Monitoring

- ▶ Lemma: (Complexity)
If a) k is the maximum number of active threads, and b) n is the number of variables, then the size of the SCG is $O(k^2 + k * n)$
- ▶ Theorem: (Decidability)
Checking if a Boolean Program is Serializable is PSPACE-complete