# … continuation

https://gpetri.github.io/
MPRI-2017/index.html

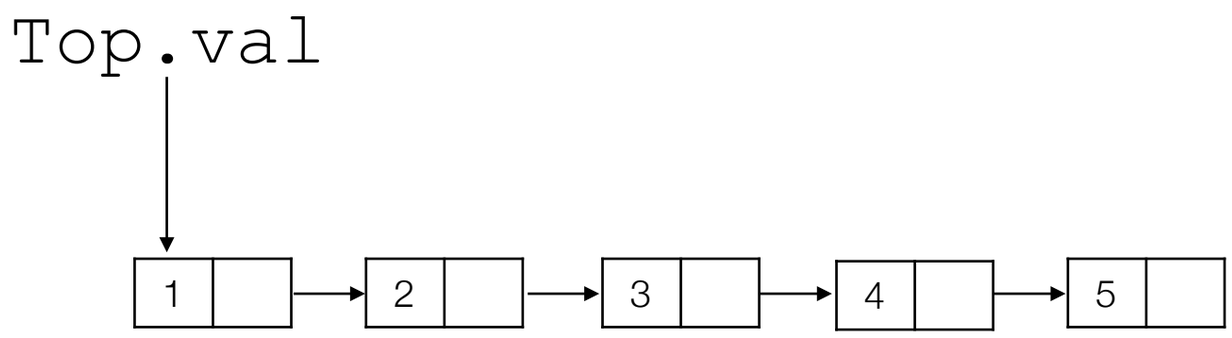
# Treiber Stack (ABA problem)

```
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;
}
```



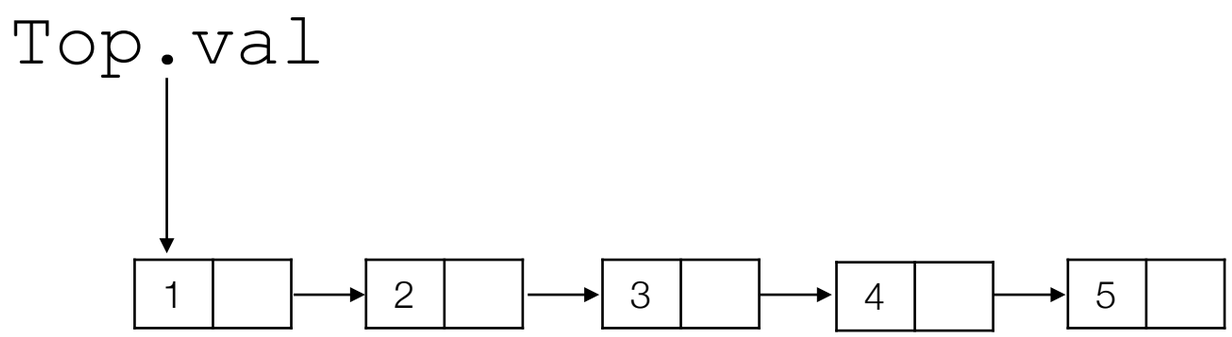
# Treiber Stack (ABA problem)

```
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;
}
```



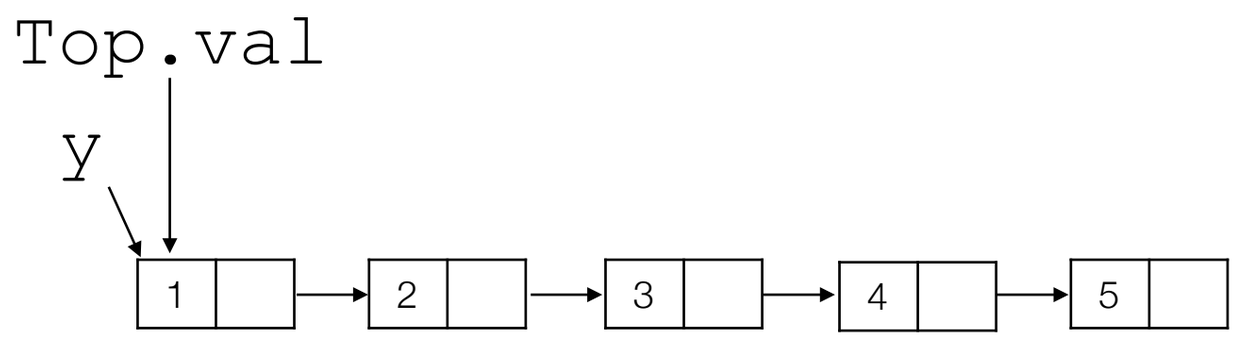
# Treiber Stack (ABA problem)

```
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;
}
```



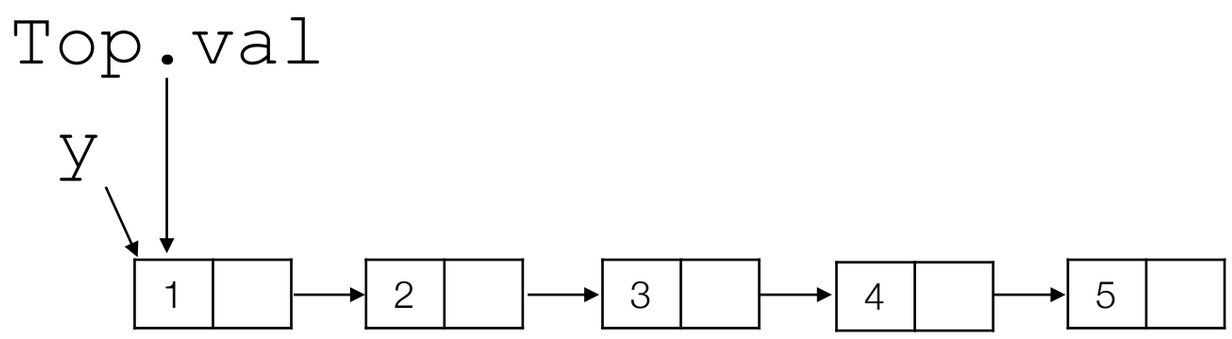
# Treiber Stack (ABA problem)

```
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;
}
```



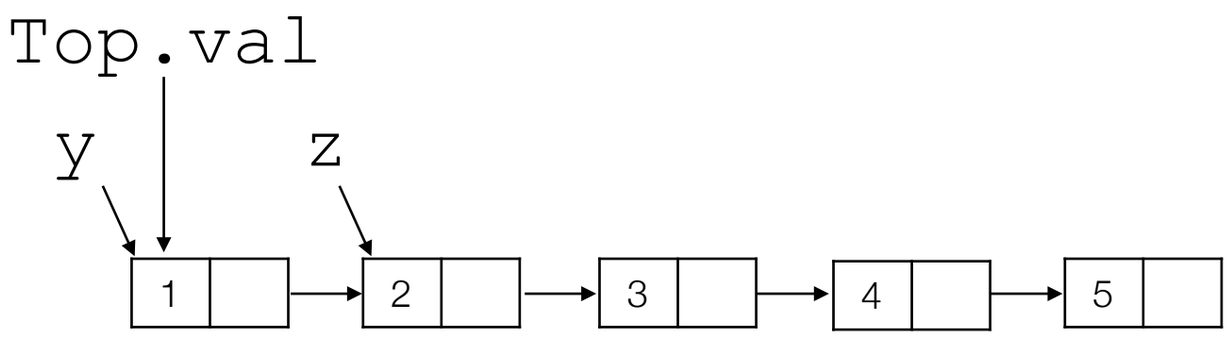
# Treiber Stack (ABA problem)

```
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)

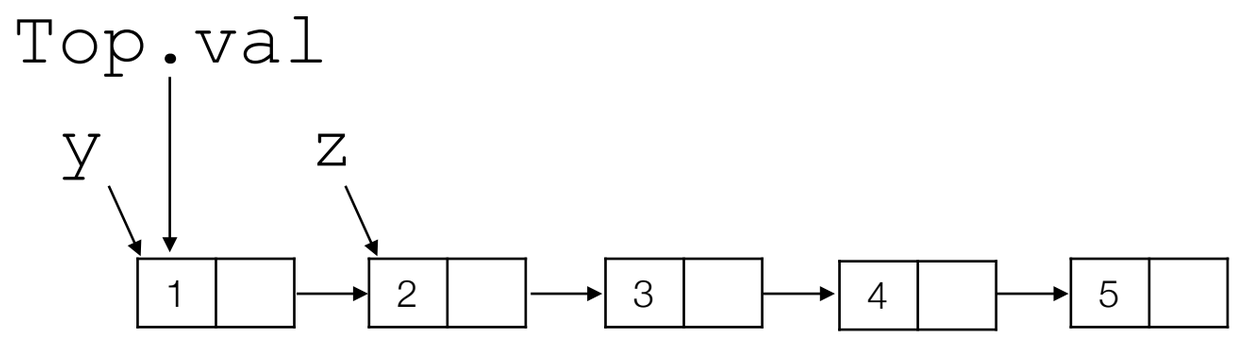
```
class Node {
  Node tl;
  int val;
}

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;
}
```

pop()



# Treiber Stack (ABA problem)

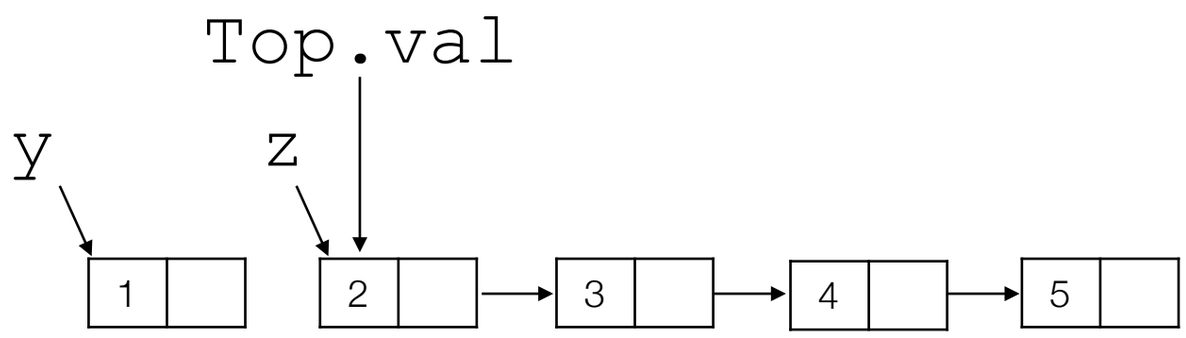
```
class Node {
  Node tl;
  int val;
}

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()



# Treiber Stack (ABA problem)

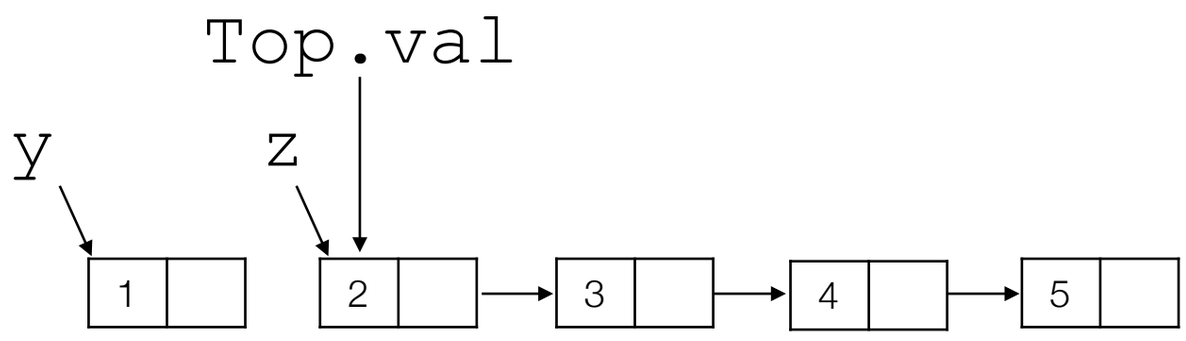
```
class Node {
  Node tl;
  int val;
}

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()



# Treiber Stack (ABA problem)

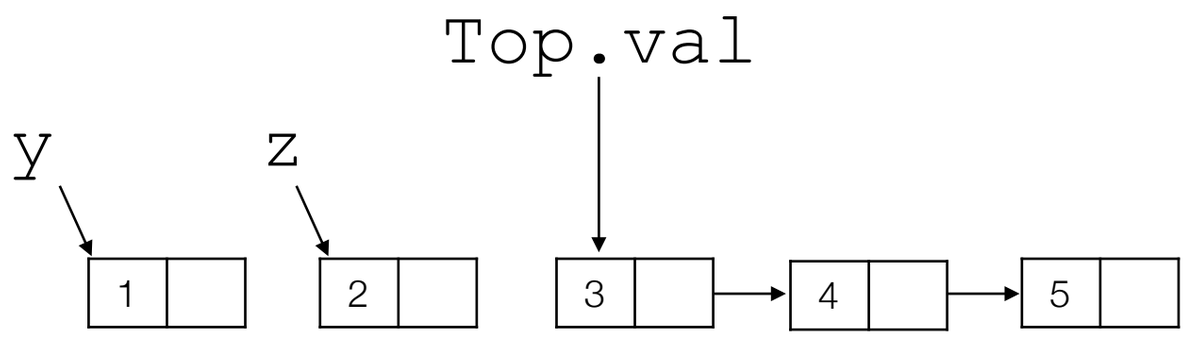
```
class Node {
  Node tl;
  int val;
}

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()



# Treiber Stack (ABA problem)

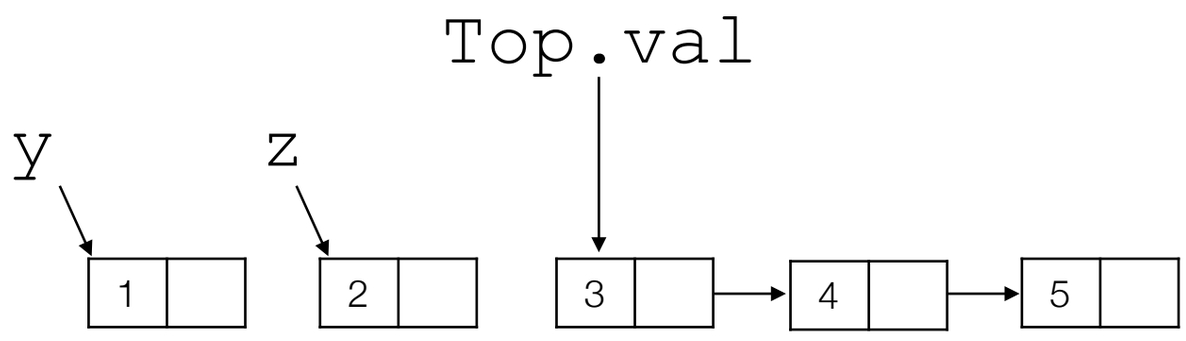
```
class Node {
  Node tl;
  int val;
}

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 Stack (ABA problem)

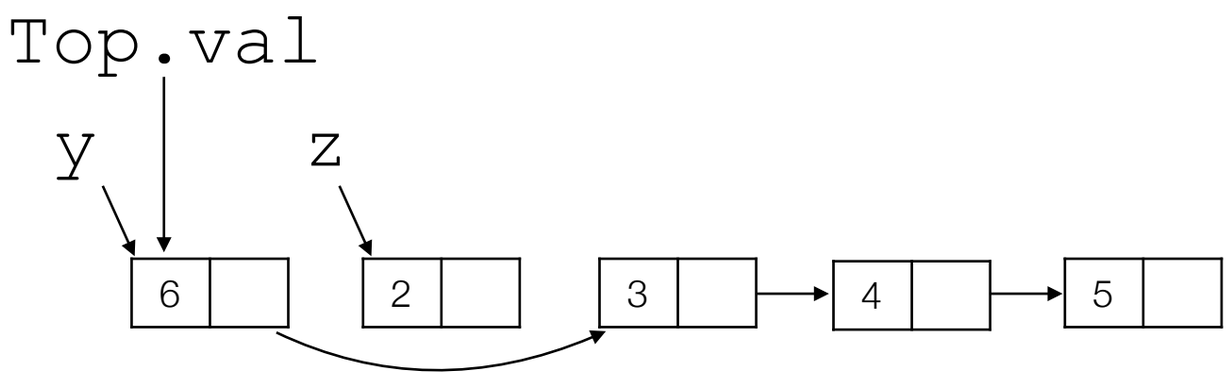
```
class Node {
  Node tl;
  int val;
}

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 Stack (ABA problem)

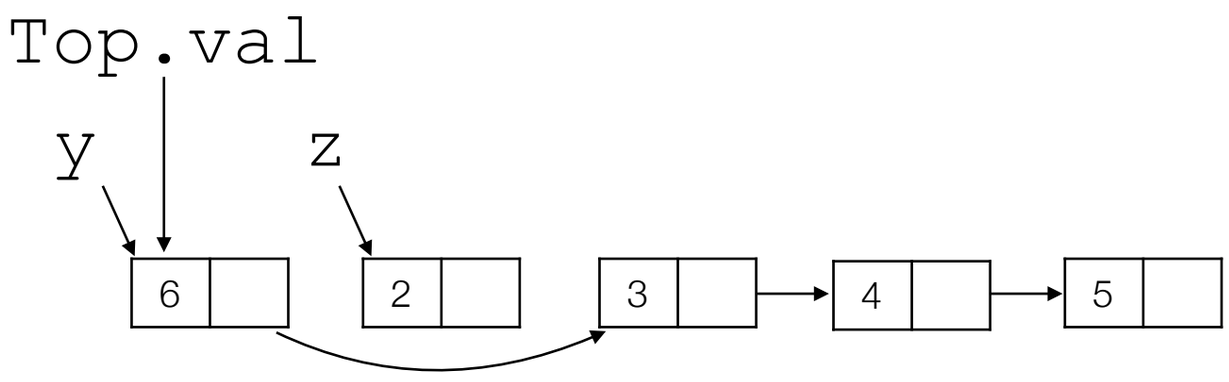
```
class Node {
  Node tl;
  int val;
}

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 Stack (ABA problem)

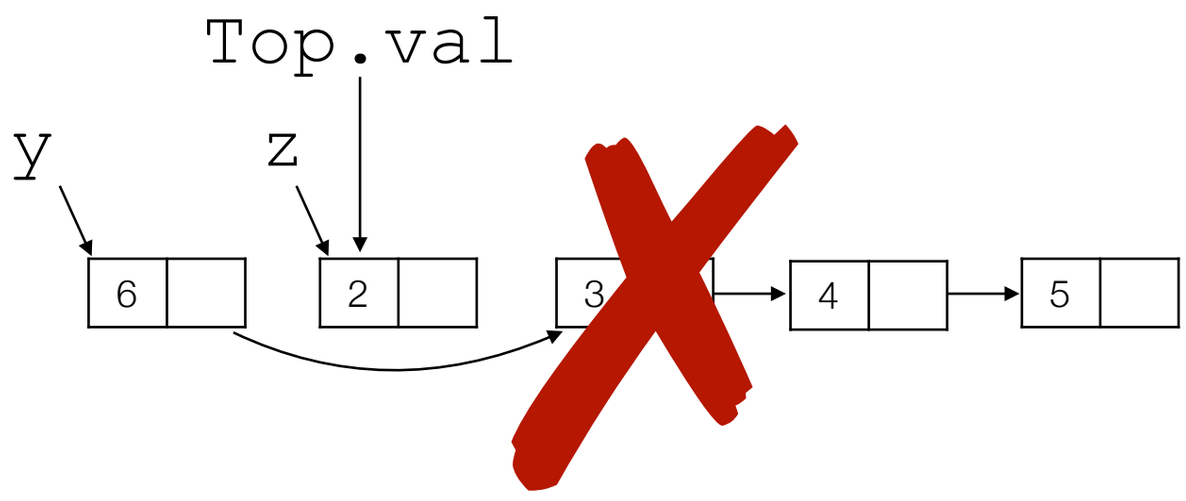
```
class Node {
  Node tl;
  int val;
}

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 {
  Node tl;
  int val;
}
class NodePtr{
  Node val;
  TID[] H;
} TOP;
```

```
void push(int e) {
  Node y, n;
  y = new();
  y->val = e;
  for(n=0; n<=THREADS; n++)
    if(H[n] == y)
      return false;
  while(true) {
    n = TOP->val;
    y->tl = n;
    if (cas(TOP->val, n, y))
      break;
  }
  return true;
}
```

```
int pop() {
  Node y,z;
  while(true) {
    y = TOP->val;
    if (y==0) return EMPTY;
    (TOP->H)[getTid()] = y;
    if (TOP->val != y)
      continue;
    z = y->tl;
    if (cas(TOP->val, y, z))
      break;
  }
  (TOP->H)[getTid()] = null
  return y->val;
}
```

Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects [M. Michael'04]

# 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;
  }
}
```

[Hendler et al.'04]

# VERIFYING LINEARIZABILITY

## RGSEP: Rely Guarantee + Separation Logic

# Resources, Concurrency and Local Reasoning*

Peter W. O'Hearn
Queen Mary, University of London

To John C. Reynolds for his 70th birthday.

**Abstract**

In this paper we show how a resource-oriented logic, separation logic, can be used to reason about the usage of resources in concurrent programs.

## 1 Introduction

Resource has always been a central concern in concurrent programming. Often, a number of processes share access to system resources such as memory, processor time, or network bandwidth, and correct resource usage is essential for the overall working of a system. In the 1960s and 1970s Dijkstra, Hoare and Brinch Hansen attacked the problem of resource control in their basic works on concurrent programming [17, 18, 23, 24, 8, 9]. In addition to the use of synchronization mechanisms to provide protection from inconsistent use, they stressed the importance of *resource separation* as a means of controlling the complexity of process interactions and reducing the possibility of time-dependent errors. This paper revisits their ideas using the formalism of separation logic [43].

Our initial motivation was actually rather simple-minded. Separation logic extends Hoare's logic to programs that manipulate data structures with embedded pointers. The main primitive of the logic is its separating conjunction, which allows local reasoning about the mutation of one portion of state, in a way that automatically guarantees that other portions of the system's state remain unaffected [34]. Thus far separation logic has been applied to sequential code but, because of the way it breaks state into chunks, it seemed as if the formalism might be well suited to shared-variable concurrency, where one would like to assign different portions of state to different processes.

Another motivation for this work comes from the perspective of general resource-oriented logics such as linear logic [19] and BI [35]. Given the development of these logics it might seem natural to try to apply them to the problem of reasoning about resources in concurrent programs. This paper is one attempt to do so – separation logic's assertion language is an instance of BI – but it

# Resources, Concurrency and Local Reasoning*

Peter W. O'Hearn
Queen Mary, University of London

To John C

In this paper we show how a
about the usage of resources i

## 1 Introduction

Resource has always been a cer
processes share access to system
and correct resource usage is ess
Dijkstra, Hoare and Brinch Hans
concurrent programming [17, 18,
to provide protection from inco
as a means of controlling the c
time-dependent errors. This pa

Our initial motivation was
logic to programs that manipu
of the logic is its separating co
portion of state, in a way that
remain unaffected [34]. Thus fa
of the way it breaks state int
shared-variable concurrency, w
processes.

Another motivation for thi
such as linear logic [19] and B
to try to apply them to the
paper is one attempt to do s

*Technical Report*

Number 726

UCAM-CL-TR-726
ISSN 1476-2986

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

# Modular fine-grained concurrency verification

Viktor Vafeiadis

# 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

Modular Fine-Grained Concurrency Verification
[Vafeiadis'08]

# 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

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

## Substitutions

$$
\begin{array}{rcl}
\{\exists X, x = X \wedge y = X + 1\}[x \leftarrow 8] & = & \{\exists X, 8 = X \wedge y = X + 1\} \\
 & \equiv & \{y = 9\}
\end{array}
$$

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

# The meaning of Triples

For any initial state satisfying P

The final state must satisfy Q

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

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

$\{x = 3\}\ x := x + 1\ \{x \leq 0\}$

$\{x = 3\}\ x := x + 1\ \{x \geq 0\}$

$\{x = 3\}\ x := x + 1; y := x\ \{y \geq 0\}$

$\{\exists X, x = X \wedge y = X + 1\}\ x := x + 1; y := x\ \{x = y\}$

$\{x = 0\}$ if $true$ then $x := x + 1$ else $x := x - 1$ fi $\{x = 1\}$

$\{x = 0\}$ if $x = 5$ then $x := x + 1$ else $x := x - 1$ fi $\{x = 1\}$

$\{x = 0\}$ while $true$ do $x := x + 1$ od $\{false\}$

# Hoare Logic Rules

# Hoare Logic Rules

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

# Hoare Logic Rules

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

# Hoare Logic Rules

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

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

# Hoare Logic Rules

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

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

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

# Hoare Logic Rules

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

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

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

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

# Hoare Logic Rules

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

$$\frac{\{P\}\ c_0\ \{R\} \qquad \{R\}\ c_1\ \{Q\}}{\{P\}\ c_0; c_1\ \{Q\}} \qquad \frac{\{P\}\ c\ \{Q\} \qquad P' \Rightarrow P \qquad Q \Rightarrow Q'}{\{P'\}\ c\ \{Q'\}}$$

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

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

# A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\mathtt{x} = \mathtt{y};$$
$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

# A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\{\mathtt{x} = X \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}$$
$$\mathtt{x} = \mathtt{y};$$

$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

# A simple Hoare Logic Proof

$$\{\mathtt{x} = X \wedge \mathtt{y} = Y\}$$
$$\mathtt{aux} = \mathtt{x};$$
$$\{\mathtt{x} = X \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}$$
$$\mathtt{x} = \mathtt{y};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = Y \ \wedge \mathtt{aux} = X\}$$
$$\mathtt{y} = \mathtt{aux};$$
$$\{\mathtt{x} = Y \wedge \mathtt{y} = X\}$$

# The Owicki/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)

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

# Quick intro to OG (R/G)

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

## 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)

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

## 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!

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

$$\begin{array}{l} \texttt{x} = 100 \\ \texttt{flag} = true \end{array} \quad \| \quad \begin{array}{l} \texttt{if(flag)} \\ \quad \texttt{x} = x - 50 \end{array}$$

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

# To the Board!

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

$$\begin{array}{l|\!|l} \texttt{x} = 100 & \texttt{if(flag)} \\ \texttt{flag} = true & \quad \texttt{x} = x - 50 \end{array}$$

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

$$\{\texttt{x} = 0\}$$

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

$$\{\texttt{x} = 2\}$$

# To the Board!

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

$$\begin{array}{l|\!|l} \texttt{x} = 100 & \texttt{if(flag)} \\ \texttt{flag} = true & \quad \texttt{x} = x - 50 \end{array}$$

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

Atomic $\longrightarrow$

$$\begin{array}{c} \{\texttt{x} = 0\} \\ \texttt{x} = \texttt{x} + 1 \quad \| \quad \texttt{x} = \texttt{x} + 1 \\ \{\texttt{x} = 2\} \end{array}$$

Ghost Variables

# Separation Logic (SL)

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v \mid \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P \\
 & | & \forall X, P \\
 & | & \texttt{emp} \\
 & | & \texttt{junk} \\
 & | & x \mapsto v \\
 & | & P * Q \\
 & | & P \mathrel{-\!\!\circledast} Q
\end{array}
$$

Spatial Assertions

# Separation Logic (SL)

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v \mid \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P \\
 & | & \forall X, P \\
 & | & \texttt{emp} \\
 & | & \texttt{junk} \\
 & | & x \mapsto v \\
 & | & P * Q \\
 & | & P -\!\!\circledast Q
\end{array}
$$

Spatial Assertions

$h, i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset$

$h, i \vDash_{SL} \texttt{junk} \;=\; h, i \vDash_{SL} \neg\texttt{emp}$

# Separation Logic (SL)

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v \mid \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P \\
 & | & \forall X, P \\
 & | & \texttt{emp} \\
 & | & \texttt{junk} \\
 & | & x \mapsto v \\
 & | & P * Q \\
 & | & P -\!\!\circledast Q
\end{array}
$$

Spatial Assertions

$h, i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset$

$h, i \vDash_{SL} \texttt{junk} \;=\; h, i \vDash_{SL} \neg\texttt{emp}$

$h, i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge i(x) = p \;\wedge\; h[i] = v$

$h$ $v$

# Separation Logic (SL)

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v \mid \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P \\
 & | & \forall X, P \\
 & | & \texttt{emp} \\
 & | & \texttt{junk} \\
 & | & x \mapsto v \\
 & | & P * Q \\
 & | & P -\!\!\circledast\, Q
\end{array}
$$

Spatial Assertions

$h, i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset$

$h, i \vDash_{SL} \texttt{junk} \;=\; h, i \vDash_{SL} \neg\texttt{emp}$

$h, i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge i(x) = p \;\wedge\; h[i] = v$

# Separation Logic (SL)

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v \mid \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P \\
 & | & \forall X, P \\
 & | & \texttt{emp} \\
 & | & \texttt{junk} \\
 & | & x \mapsto v \\
 & | & P * Q \\
 & | & P \mathrel{-\!\!\circledast} Q
\end{array}
$$

Spatial Assertions

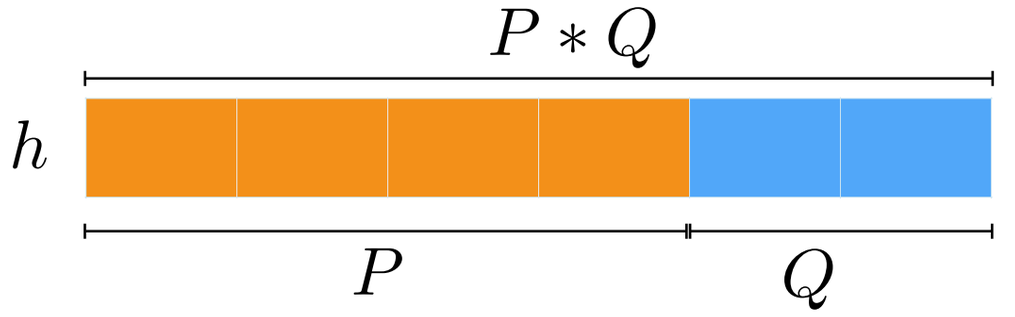
$h, i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset$

$h, i \vDash_{SL} \texttt{junk} \;=\; h, i \vDash_{SL} \neg\texttt{emp}$

$h, i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge i(x) = p \;\wedge\; h[i] = v$

$h, i \vDash_{SL} (P * Q) =$

$\exists\ h1\ h2, (h_1 \uplus h_2 = h) \wedge h_1, i \vDash_{SL} P \;\wedge\; h_2, i \vDash_{SL} Q$



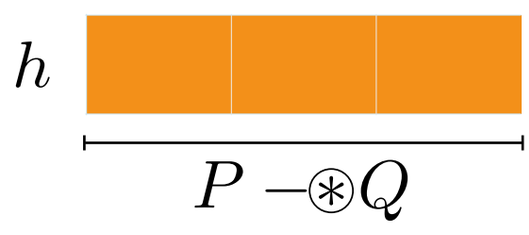
# Separation Logic (SL)

$$
\begin{array}{rcll}
P & ::= & True & h,i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset \\
 & | & \neg P & h,i \vDash_{SL} \texttt{junk} \;=\; h,i \vDash_{SL} \neg\texttt{emp} \\
 & | & P \wedge Q & \\
 & | & P \vee Q & h,i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge \\
 & | & P \Rightarrow Q & \qquad\qquad i(x) = p \;\wedge\; h[i] = v \\
 & | & x = v \mid \ldots & h,i \vDash_{SL} (P * Q) = \\
 & | & \exists X, P & \exists\, h1\; h2, (h_1 \uplus h_2 = h) \wedge \\
 & | & \forall X, P & \qquad h_1,i \vDash_{SL} P \;\wedge\; h_2,i \vDash_{SL} Q \\
 & | & \forall X, P & \\
 & | & \texttt{emp} & \\
 & | & \texttt{junk} & \\
 & | & x \mapsto v & \\
 & | & P * Q & \\
 & | & P -\!\circledast\; Q &
\end{array}
$$

Spatial Assertions

# Separation Logic (SL)

$$
\begin{array}{rcll}
P & ::= & True & h,i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset \\
 & | & \neg P & h,i \vDash_{SL} \texttt{junk} \;=\; h,i \vDash_{SL} \neg\texttt{emp} \\
 & | & P \wedge Q & \\
 & | & P \vee Q & h,i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge \\
 & | & P \Rightarrow Q & \qquad\qquad i(x) = p \;\wedge\; h[i] = v \\
 & | & x = v \mid \ldots & h,i \vDash_{SL} (P * Q) = \\
 & | & \exists X, P & \quad \exists\, h1\; h2, (h_1 \uplus h_2 = h) \wedge \\
 & | & \forall X, P & \qquad h_1,i \vDash_{SL} P \;\wedge\; h_2,i \vDash_{SL} Q \\
 & | & \forall X, P & \\
\text{Spatial} & | & \texttt{emp} & h,i \vDash_{SL} (P \mathrel{-\!\circledast} Q) = \\
\text{Assertions} & | & \texttt{junk} & \quad \exists\, h1\; h2, (h_1 \uplus h = h_2) \wedge \\
 & | & x \mapsto v & \qquad h_1,i \vDash_{SL} P \;\wedge\; h_2,i \vDash_{SL} Q \\
 & | & P * Q & \\
 & | & P \mathrel{-\!\circledast} Q &
\end{array}
$$



# Separation Logic (SL)

$$
\begin{array}{rcl}
P & ::= & True \\
 & | & \neg P \\
 & | & P \wedge Q \\
 & | & P \vee Q \\
 & | & P \Rightarrow Q \\
 & | & x = v \mid \ldots \\
 & | & \exists X, P \\
 & | & \forall X, P \\
 & | & \forall X, P \\
 & | & \texttt{emp} \\
 & | & \texttt{junk} \\
 & | & x \mapsto v \\
 & | & P * Q \\
 & | & P \mathrel{-\!\!\circledast} Q
\end{array}
$$

Spatial Assertions

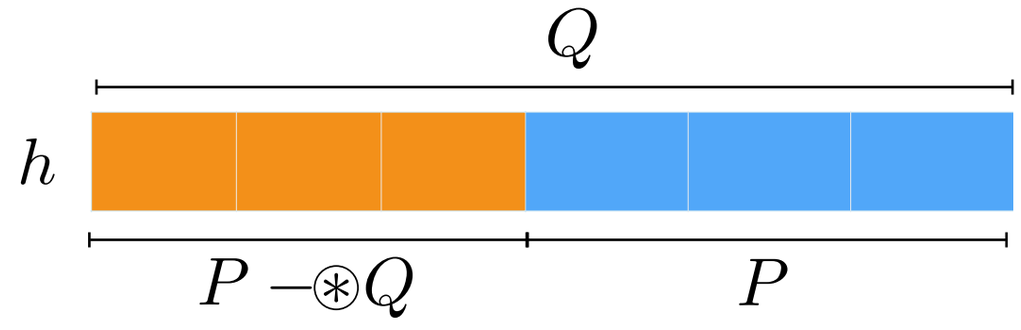
$h, i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset$

$h, i \vDash_{SL} \texttt{junk} \;=\; h, i \vDash_{SL} \neg\texttt{emp}$

$h, i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge i(x) = p \;\wedge\; h[i] = v$

$h, i \vDash_{SL} (P * Q) =$
$\exists\ h1\ h2, (h_1 \uplus h_2 = h) \wedge h_1, i \vDash_{SL} P \;\wedge\; h_2, i \vDash_{SL} Q$

$h, i \vDash_{SL} (P \mathrel{-\!\!\circledast} Q) =$
$\exists\ h1\ h2, (h_1 \uplus h = h_2) \wedge h_1, i \vDash_{SL} P \;\wedge\; h_2, i \vDash_{SL} Q$



# Separation Logic (SL)

$$
\begin{array}{rcll}
P & ::= & True & h,i \vDash_{SL} \texttt{emp} = \mathsf{dom}(h) = \emptyset \\
 & | & \neg P & h,i \vDash_{SL} \texttt{junk} \;=\; h,i \vDash_{SL} \neg\texttt{emp} \\
 & | & P \wedge Q & \\
 & | & P \vee Q & h,i \vDash_{SL} x \mapsto v = \exists p, \mathsf{dom}(h) = \{p\} \wedge \\
 & | & P \Rightarrow Q & \qquad\qquad i(x) = p \;\wedge\; h[i] = v \\
 & | & x = v \mid \ldots & h,i \vDash_{SL} (P * Q) = \\
 & | & \exists X, P & \quad \exists\, h1\ h2, (h_1 \uplus h_2 = h) \wedge \\
 & | & \forall X, P & \qquad h_1,i \vDash_{SL} P \;\wedge\; h_2,i \vDash_{SL} Q \\
 & | & \forall X, P & \\
 & | & \texttt{emp} & h,i \vDash_{SL} (P \mathrel{-\!\circledast} Q) = \\
 & | & \texttt{junk} & \quad \exists\, h1\ h2, (h_1 \uplus h = h_2) \wedge \\
 & | & x \mapsto v & \qquad h_1,i \vDash_{SL} P \;\wedge\; h_2,i \vDash_{SL} Q \\
 & | & P * Q & \\
 & | & P \mathrel{-\!\circledast} Q &
\end{array}
$$

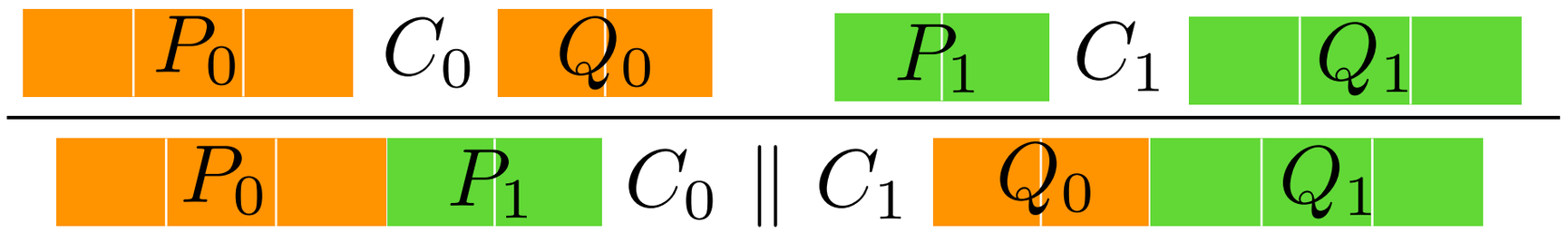
Spatial Assertions

# SL Frame Rule

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

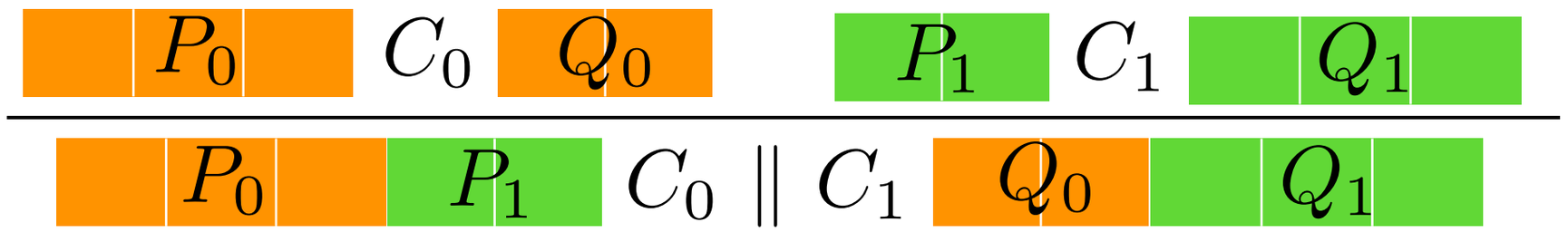
# Concurrent Separation Logic (CSL)

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



# Concurrent Separation Logic (CSL)

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




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

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

Local State

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

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

# 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

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

# RGSep Judgment

Pre Post

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

# RGSep Judgment

Pre Post

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

Set of Guarantee Actions:

Global actions allowed to this command

# RGSep Judgment

Pre Post

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

Set of Guarantee Actions:
Global actions allowed to this command

Set of Rely Actions:
Global actions of other concurrent commands

# RGSep Judgment

Pre Post

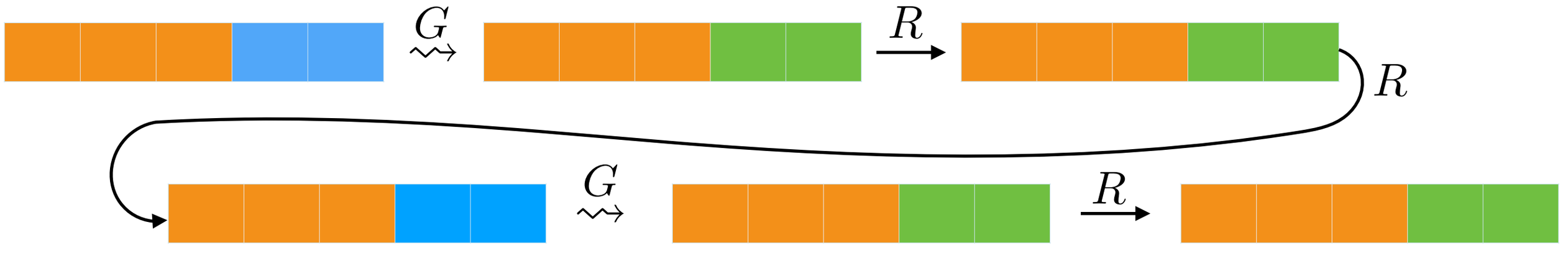
$$R, G \vDash \{P\}\ c\ \{Q\}$$

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

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

# RGSep Proof Rules

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

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

# RGSep Proof Rules

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

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

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

# RGSep Proof Rules

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

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

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

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

# RGSep Proof Rules

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

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

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

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

$$\frac{\begin{array}{ll}R \cup G_2, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} & P_1 \text{ stable for } R \cup G_2 \\ R \cup G_1, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\} & P_2 \text{ stable for } R \cup G_1\end{array}}{R, G_1 \cup G_2 \vdash \{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}}$$

# RGSep Proof Rules

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

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

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

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

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

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

# To the Board (RGSep)

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

$$\begin{array}{l|\!|l} \texttt{x} = 100 & \texttt{if}(\texttt{flag}) \\ \texttt{flag} = true & \quad \texttt{x} = x - 50 \end{array}$$

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

$$\{\texttt{x} = 0\}$$

$$\texttt{x} = \texttt{x} + 1 \quad \| \quad \texttt{x} = \texttt{x} + 1$$

$$\{\texttt{x} = 2\}$$

# 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

$$\{emp\}\ \ \mathtt{x} := \mathtt{new}_{\mathtt{single}};\ \ \{\mathtt{x} \overset{s}{\mapsto} undef\}$$

$$\{y \overset{s}{\mapsto} z \wedge e = y\}\ \ \mathtt{x} := [e]_{\mathtt{single}};\ \ \{y \overset{s}{\mapsto} z \wedge \mathtt{x} = z\}$$

$$\{e_1 \overset{s}{\mapsto} undef\}\ \ [e_1]_{\mathtt{single}} := e_2;\ \ \{e_1 \overset{s}{\mapsto} e_2\}$$

$$\{e \overset{s}{\mapsto} \_\}\ \ \mathtt{dispose}_{\mathtt{single}}(e);\ \ \{emp\}$$

# Linearizability in RGSep:
## Some Additional Ingredients

### Single Assignment Variables

$$\{emp\}\ \ \mathtt{x} := \mathtt{new}_{\mathtt{single}};\ \ \{\mathtt{x} \overset{s}{\mapsto} undef\}$$

$$\{y \overset{s}{\mapsto} z \wedge e = y\}\ \ \mathtt{x} := [e]_{\mathtt{single}};\ \ \{y \overset{s}{\mapsto} z \wedge \mathtt{x} = z\}$$

$$\{e_1 \overset{s}{\mapsto} undef\}\ \ [e_1]_{\mathtt{single}} := e_2;\ \ \{e_1 \overset{s}{\mapsto} e_2\}$$

$$\{e \overset{s}{\mapsto} \_\}\ \ \mathtt{dispose}_{\mathtt{single}}(e);\ \ \{emp\}$$

### Simulation Argument in RGSep

$$\{p \wedge d.\mathtt{AbsResult} \overset{s}{\mapsto} undef\}\ \mathrm{ConcreteOp}(d)\ \{d.\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{Result}\}$$

# Linearizability in RGSep:
## Some Additional Ingredients

### Single Assignment Variables

$$\{emp\}\ \ \mathtt{x} := \mathtt{new}_{\mathtt{single}};\ \ \{\mathtt{x} \overset{s}{\mapsto} undef\}$$

$$\{y \overset{s}{\mapsto} z \wedge e = y\}\ \ \mathtt{x} := [e]_{\mathtt{single}};\ \ \{y \overset{s}{\mapsto} z \wedge \mathtt{x} = z\}$$

$$\{e_1 \overset{s}{\mapsto} undef\}\ \ [e_1]_{\mathtt{single}} := e_2;\ \ \{e_1 \overset{s}{\mapsto} e_2\}$$

$$\{e \overset{s}{\mapsto} \_\}\ \ \mathtt{dispose}_{\mathtt{single}}(e);\ \ \{emp\}$$

### Simulation Argument in RGSep

$$\{p \wedge d.\mathtt{AbsResult} \overset{s}{\mapsto} undef\}\ \mathrm{ConcreteOp}(d)\ \{d.\mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{Result}\}$$

### Shorthand Annotation for Linearization Points

$$\mathtt{Lin}_{e_1,\ldots,e_n} \overset{\text{def}}{=} e_1.\mathtt{AbsResult} := \mathtt{AbsOp}(e_1);\ \ldots;\ e_n.\mathtt{AbsResult} := \mathtt{AbsOp}(e_n);$$

# Specifications:

## from histories to states

Predefined Abstract State: **`Abs`**

Abstract specification of the operations

$$\texttt{Abs\_push}(e) \stackrel{\text{def}}{=} \langle\, \texttt{Abs} := e{\cdot}\texttt{Abs};\ \texttt{AbsResult} := e;\, \rangle$$

$$\begin{aligned}\texttt{Abs\_pop}() \stackrel{\text{def}}{=} \langle\, & \textbf{case}\ (\texttt{Abs}) \\ & \mid \epsilon \Longrightarrow \quad \texttt{AbsResult} := \texttt{EMPTY}; \\ & \mid v{\cdot}A \Longrightarrow \{\, \texttt{Abs} := A;\ \texttt{AbsResult} := v; \,\} \,\rangle\end{aligned}$$

We can generate the specification histories

δ = <Abs_push(v),_> · <Abs_pop(v),v> …

# 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 (¬CAS_this(&S, t, x));
}
```

```
value pop () {
  Cell t, x;
  do {
    ⟨t := S; Lin_this(t = null);⟩
    if(t = null)
      return EMPTY;
    x := t.next;
  } while(¬CAS_this(&S, t, x));
  return t.data;
}
```

Linearization points are marked

$$\texttt{Abs\_push}(e) \stackrel{\text{def}}{=} \langle\ \texttt{Abs} := e{\cdot}\texttt{Abs};\ \texttt{AbsResult} := e;\rangle$$

$$\begin{aligned}\texttt{Abs\_pop}() \stackrel{\text{def}}{=} \langle\ & \textbf{case}\ (\texttt{Abs}) \\ & \mid \epsilon \Longrightarrow \quad \texttt{AbsResult} := \texttt{EMPTY}; \\ & \mid v{\cdot}A \Longrightarrow \{\ \texttt{Abs} := A;\ \texttt{AbsResult} := v;\}\ \rangle\end{aligned}$$

# Treiber Stack in RGSep

## Actions

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

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

# Treiber Stack in RGSep

## Actions

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

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

## Inductive Representation of a List containing A

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

# Treiber Stack in RGSep

## Actions

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

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

## Inductive Representation of a List containing A

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

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

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

# Treiber Stack in RGSep

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

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

```
void push (value v) {
  Cell t, x;
  {AbsResult ↦^s undef * StackInv}
  x := new Cell();
  x.data := v;
  { AbsResult ↦^s undef
    * x↦Cell(v, _) * StackInv }
  do {
    { AbsResult ↦^s undef
      * x↦Cell(v, _) * StackInv }
    ⟨t := S;⟩
    x.next := t;
    { AbsResult ↦^s undef
      * x↦Cell(v, t) * StackInv }
  } while (¬CAS_this(&S, t, x));
  {AbsResult ↦^s v * StackInv}
}
```

```
value pop () {
  Cell t, x, temp;
  {AbsResult ↦^s undef * StackInv}
  do {
    ⟨t := S; Lin_this(t = null);⟩
    { (t=null ∧ AbsResult ↦^s EMPTY * StackInv)
    ∨ (∃x. AbsResult ↦^s undef * K(x)) }
    if(t = null) return EMPTY;
    {∃x. AbsResult ↦^s undef * K(x)}
    x := t.next;
    {AbsResult ↦^s undef * K(x)}
  } while(¬CAS_this(&S, t, x));
  { ∃v. AbsResult ↦^s v
  * ∃x A. &Abs ↦ A * &S ↦ x
    * lseg(x, null, A) * x↦Cell(v, _) * true }
  temp := t.data;
  {∃v. AbsResult ↦^s temp * StackInv}
  return temp;
}
```

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

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

# Treiber Stack in RGSep

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

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

```
void push (value v) {
  Cell t, x;
  {AbsResult ↦^s undef * StackInv}
  x := new Cell();
  x.data := v;
  { AbsResult ↦^s undef
    * x↦Cell(v, _) * StackInv }
  do {
    { AbsResult ↦^s undef
      * x↦Cell(v, _) * StackInv }
    ⟨t := S;⟩
    x.next := t;
    { AbsResult ↦^s undef
      * x↦Cell(v, t) * StackInv }
  } while (¬CAS_this(&S, t, x));
  {AbsResult ↦^s v * StackInv}
}
```

```
value pop () {
  Cell t, x, temp;
  {AbsResult ↦^s undef * StackInv}
  do {
    ⟨t := S; Lin_this(t = null);⟩
    { (t=null ∧ AbsResult ↦^s EMPTY * StackInv)
    ∨ (∃x. AbsResult ↦^s undef * K(x)) }
    if(t = null) return EMPTY;
    {∃x. AbsResult ↦^s undef * K(x)}
    x := t.next;
    {AbsResult ↦^s undef * K(x)}
  } while(¬CAS_this(&S, t, x));
  { ∃v. AbsResult ↦^s v
  * ∃x A. &Abs ↦ A * &S ↦ x
    * lseg(x, null, A) * x↦Cell(v, _) * true }
  temp := t.data;
  {∃v. AbsResult ↦^s temp * StackInv}
  return temp;
}
```

Check Stability: Exercise :)

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

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

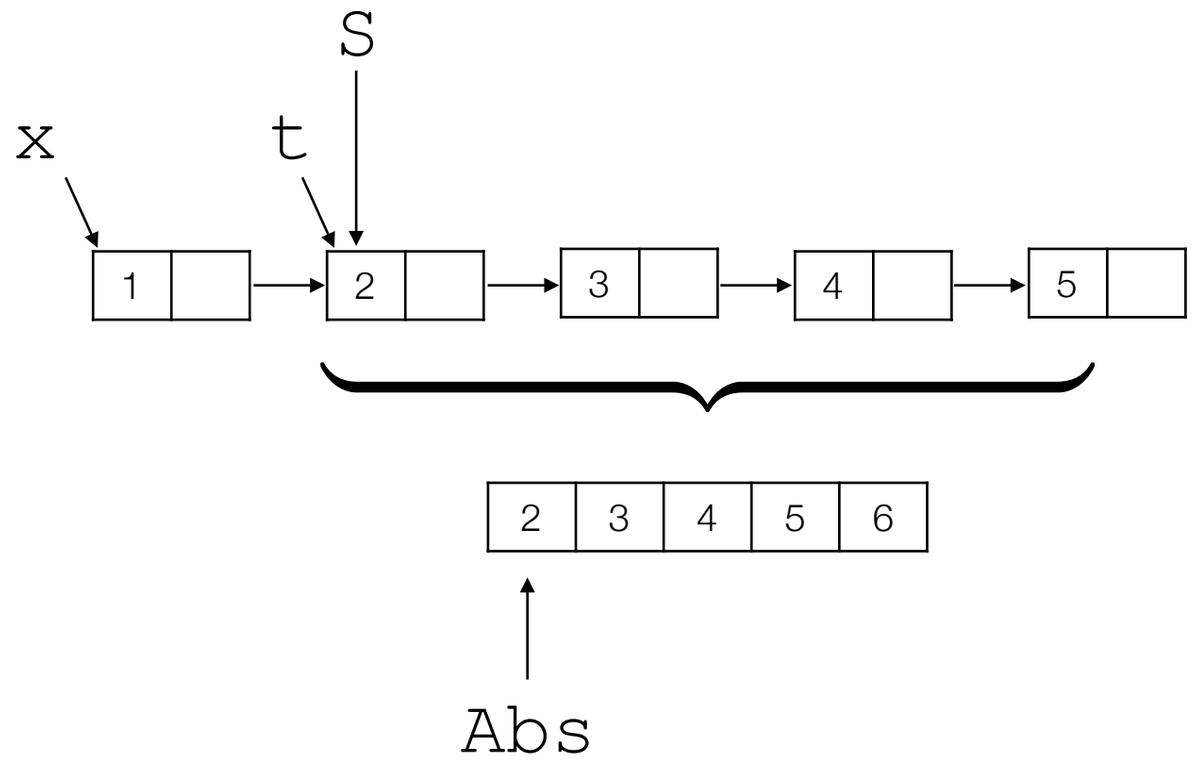
# Push

## Linearization point of Push

$$\left\{\begin{array}{l}\texttt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \texttt{x} \mapsto \mathsf{Cell}(\texttt{v}, \texttt{t}) * \mathit{StackInv}\end{array}\right\}$$

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

$$\left\{\begin{array}{l}(\texttt{b} \wedge \texttt{AbsResult} \overset{s}{\mapsto} \texttt{v} * \mathit{StackInv}) \\ \vee \begin{pmatrix}\neg\texttt{b} \wedge \texttt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \texttt{x} \mapsto \mathsf{Cell}(\texttt{v}, \texttt{t}) * \mathit{StackInv}\end{pmatrix}\end{array}\right\}$$



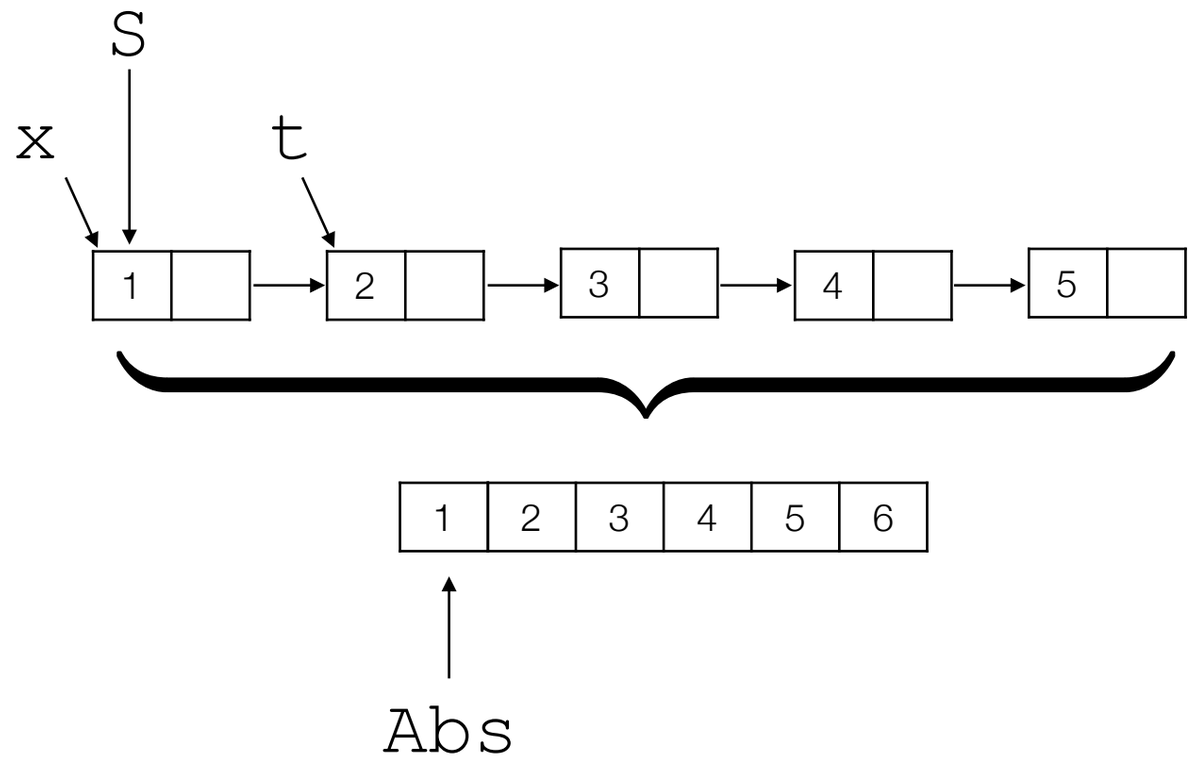
# Push

## Linearization point of Push

$$\left\{\begin{array}{l}\texttt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \texttt{x} \mapsto \mathsf{Cell}(\texttt{v}, \texttt{t}) * \mathit{StackInv}\end{array}\right\}$$

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

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



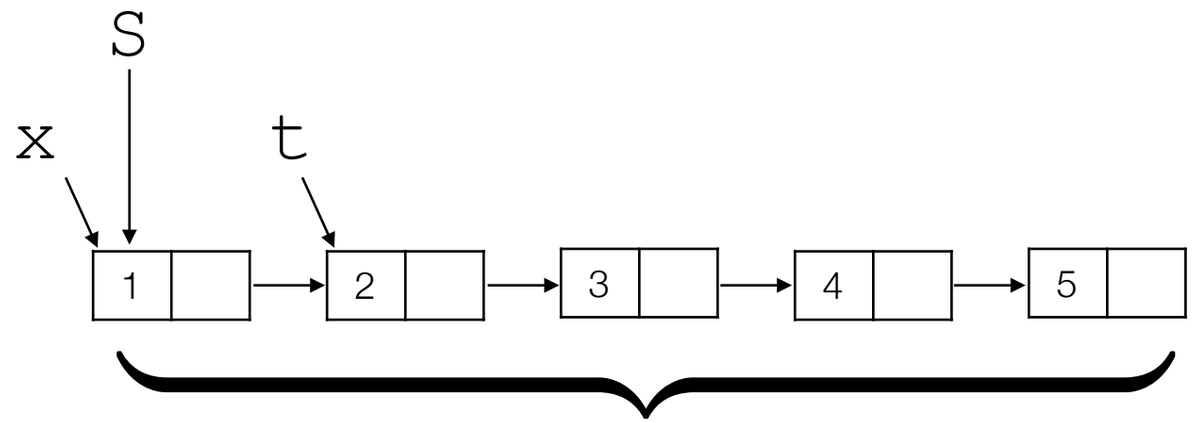
# Push

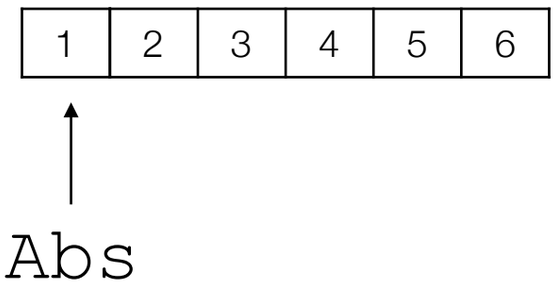
## Linearization point of Push

$$\left\{\begin{array}{l}\mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \, \mathtt{x} \mapsto \mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \mathit{StackInv}\end{array}\right\}$$

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

$$\left\{\begin{array}{l}(\mathtt{b} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathtt{v} * \mathit{StackInv}) \\ \vee \begin{pmatrix}\neg\mathtt{b} \wedge \mathtt{AbsResult} \overset{s}{\mapsto} \mathit{undef} \\ * \, \mathtt{x} \mapsto \mathsf{Cell}(\mathtt{v}, \mathtt{t}) * \mathit{StackInv}\end{pmatrix}\end{array}\right\}$$




## Succesful CAS

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

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

$\langle [\&\mathtt{S}] := \mathtt{x}; \mathtt{Abs\_push}(\mathtt{v}) \rangle$

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

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

# Pop

## CAS Linearization point of Pop

$$\left\{\texttt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(\texttt{x})\right\}$$

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

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

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

# Pop

## CAS Linearization point of Pop

$$\left\{\texttt{AbsResult} \overset{s}{\mapsto} \mathit{undef} * K(\texttt{x})\right\}$$

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

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

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

## Read Linearization point of Pop

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

# 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

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

# Lock Coupling Set

Some predicates

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

`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

$$N_s(x, v, t) = x \mapsto \{\texttt{.lock} = s, \texttt{.value} = v, \texttt{.next} = y\}$$
$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$

`x` points to a node locked by a real thread `t`

# Lock Coupling Set

Some predicates

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

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

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

$\mathrm{x}$ points to an unlocked node

# Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{\texttt{.lock} = s, \texttt{.value} = v, \texttt{.next} = y\}$$
$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$
$$U(x, v, y) = N_0(x, v, y)$$

# Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{\texttt{.lock} = s, \texttt{.value} = v, \texttt{.next} = y\}$$
$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$
$$U(x, v, y) = N_0(x, v, y)$$

## Lock and Unlock Actions

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

# Lock Coupling Set

Some predicates

$$N_s(x, v, t) = x \mapsto \{.\texttt{lock} = s, .\texttt{value} = v, .\texttt{next} = y\}$$
$$L_t(x, v, y) = N_t(x, v, y) \wedge t > 0$$
$$U(x, v, y) = N_0(x, v, y)$$

## Lock and Unlock Actions

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

## Rules

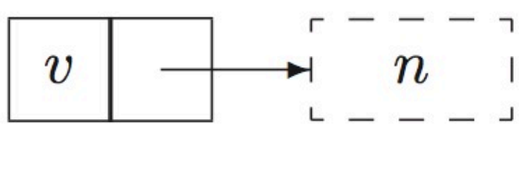
$$\frac{P \text{ stable for } R \qquad Q \text{ stable for } R \qquad P \Rightarrow N\_(p, n, n) * F \qquad L_{tid}(p, v, n) * F \Rightarrow Q}{R, G \vdash \{\boxed{\text{P}}\}\ \texttt{lock}(p)\ \{\boxed{\text{Q}}\}}$$

$$\frac{P \text{ stable for } R \qquad Q \text{ stable for } R \qquad P \Rightarrow L_{tid}(p, n, n) * F \qquad U(p, v, n) * F \Rightarrow Q}{R, G \vdash \{\boxed{\text{P}}\}\ \texttt{unlock}(p)\ \{\boxed{\text{Q}}\}}$$

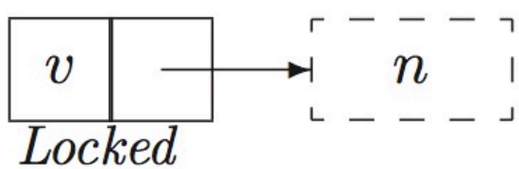
# Lock Coupling Set

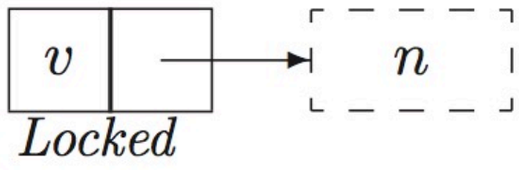
$$(t \in T) \wedge (u < v < w) \wedge (L_t(x, u, n) * N_s(n, w, y)) \rightsquigarrow L_t(x, u, m) * U(m, v, n) * N_s(n, w, y) \quad \text{(Insert)}$$

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

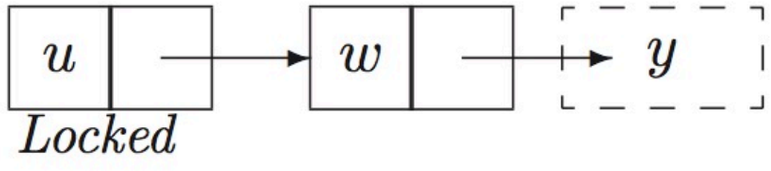


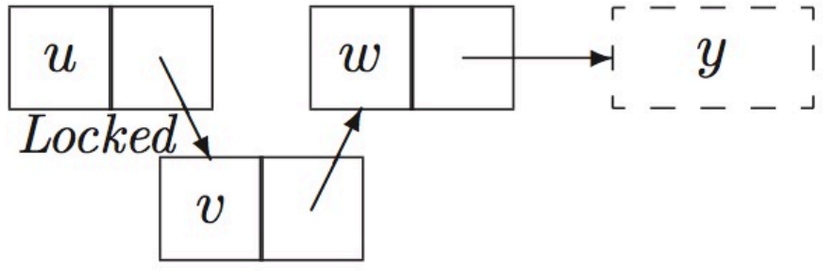

(Lock)



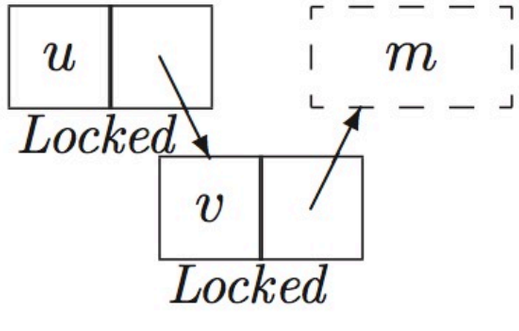



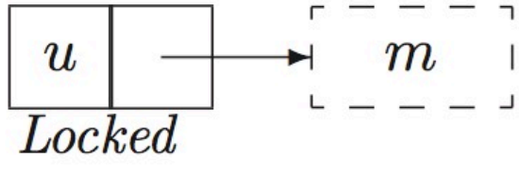

(Unlock)




$u < v < w$ (Insert)




$v < +\infty$ (Remove)

# Locate

```
locate(e) {
  local p, c, t;
  {∃A. ls(Head, A, nil) * s(A)} ∧ −∞ < e}
  p := Head;
  {∃ZB. ls(Head, ϵ, p) * N(p, −∞, Z) * ls(Z, B, nil) * s(−∞·B)} ∧ −∞ < e}
  lock(p);
  {∃Z. ∃B. ls(Head, ϵ, p) * L(p, −∞, Z) * ls(Z, B, nil) * s(−∞·B)} ∧ −∞ < e}
  ⟨c := p.next;⟩
  {∃B. ls(Head, ϵ, p) * L(p, −∞, c) * ls(c, B, nil) * s(−∞·B)} ∧ −∞ < e}
  ⟨t := c.value;⟩
  {∃u. ∃ABZ. ls(Head, A, p) * L(p, u, c) * N(c, t, Z) * ls(c, B, nil) * s(A·u·t·B)} ∧ u < e}
  {∃u. ∃ABZ. ls(Head, A, p) * L(p, u, c) * N(c, t, Z) * ls(c, B, nil) * s(A·u·t·B)} ∧ u < e}
  while (t < e) {
    {∃u. ∃ABZ. ls(Head, A, p) * L(p, u, c) * N(c, t, Z) * ls(c, B, nil) * s(A·u·t·B)} ∧ u < e ∧ t < e}
    lock(c);
    {∃uZ. ∃AB. ls(Head, A, p) * L(p, u, c) * L(c, t, Z) * ls(Z, B, nil) * s(A·u·t·B)} ∧ t < e}
    unlock(p);
    {∃Z. ∃AB. ls(Head, A, c) * L(c, t, Z) * ls(Z, B, nil) * s(A·t·B)} ∧ t < e}
    p := c;
    {∃uZ. ∃AB. ls(Head, A, p) * L(p, u, Z) * ls(Z, B, nil) * s(A·u·B)} ∧ u < e}
    ⟨c := p.next;⟩
    {∃u. ∃AB. ls(Head, A, p) * L(p, u, c) * ls(c, B, nil) * s(A·u·B)} ∧ u < e}
    ⟨t := c.value;⟩
    {∃u. ∃ABZ. ls(Head, A, p) * L(p, u, c) * N(c, t, Z) * ls(Z, B, nil) * s(A·u·t·B)} ∧ u < e}
  }
  {∃uv. ∃ABZ. ls(Head, A, p) * L(p, u, c) * N(c, v, Z) * ls(Z, B, nil) * s(A·u·v·B)} ∧ u < e ∧ e ≤ v}
  return (p, c);
}
```

# Add

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

# Remove

$\texttt{remove(e)}\ \{\ \texttt{local x, y, z, t;}$

$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \mathrm{nil}) * s(A)} \wedge -\infty < \texttt{e} \wedge \texttt{e} < +\infty\}$

$(\texttt{x}, \texttt{y}) = \texttt{locate(e)};$

$\left\{\exists uv.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * N(\texttt{y}, v, Z) \\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}v{\cdot}B)\end{array}} \wedge u < \texttt{e} \wedge \texttt{e} \leq v \wedge \texttt{e} < +\infty\right\}$

$\langle \texttt{t} = \texttt{y.value;} \rangle\ \ \texttt{if(t = e)}\ \{$

$\left\{\exists u.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * N(\texttt{y}, \texttt{e}, Z) \\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)\end{array}} \wedge \texttt{e} < +\infty\right\}$

$\texttt{lock(y)};$

$\left\{\exists u.\ \boxed{\begin{array}{l}\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * L(\texttt{y}, \texttt{e}, Z) \\ * ls(Z, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)\end{array}} \wedge \texttt{e} < +\infty\right\}$

$\langle \texttt{z} := \texttt{y.next;} \rangle$

$\left\{\exists u.\ \boxed{\begin{array}{l}\exists AB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * L(\texttt{y}, \texttt{e}, \texttt{z}) \\ * ls(\texttt{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}\texttt{e}{\cdot}B)\end{array}} \wedge \texttt{e} < +\infty\right\}$

$\langle \texttt{x.next} := \texttt{z;} \rangle$

$\{\exists u.\ \boxed{\exists AB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{z}) * ls(\texttt{z}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}B)} * L(\texttt{y}, \texttt{e}, \texttt{z})\}$

$\texttt{unlock(x)};$

$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \mathrm{nil}) * s(A)} * L(\texttt{y}, \texttt{e}, \texttt{z})\}$

$\texttt{dispose(y)};$

$\}\ \texttt{else}\ \{\ \ \{\exists u.\ \boxed{\exists ZAB.\ ls(\texttt{Head}, A, \texttt{x}) * L(\texttt{x}, u, \texttt{y}) * ls(\texttt{y}, B, \mathrm{nil}) * s(A{\cdot}u{\cdot}B)}\}$

$\texttt{unlock(x)};\}$

$\{\boxed{\exists A.\ ls(\texttt{Head}, A, \mathrm{nil}) * s(A)}\}$

$\}$

# … to be continued …

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


# Monitoring Atomicity in Concurrent Programs

Azadeh Farzan[1] and P. Madhusudan[2]

[1] Carnegie Mellon University (afarzan@cs.cmu.edu)
[2] Univ. of Illinois at Urbana-Champaign (madhu@cs.uiuc.edu)

**Abstract.** We study the problem of monitoring concurrent program runs for atomicity violations. Unearthing fundamental results behind scheduling algorithms in database control, we build space-efficient monitoring algorithms for checking atomicity that use space polynomial in the number of active threads and entities, and independent of the length of the run monitored. Second, by interpreting the monitoring algorithm as a *finite automaton*, we solve the model checking problem for atomicity of finite-state concurrent models. This establishes (for the first time) that model checking finite-state concurrent models for atomicity is decidable, and remedies incorrect proofs published in the literature. Finally, we exhibit experimental evidence that our atomicity monitoring algorithm ... and space benefits on benchmark applications.

# 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 : $\texttt{wr}_{\texttt{p,v}}$
  - Read : $\texttt{rd}_{\texttt{p,v}}$
  - RMW : $\texttt{cas}_{\texttt{p,v,w}}$
- Sometimes we need to mention the thread: (t, $\texttt{wr}_{\texttt{p,v}}$)

# 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 $\backsim$, such that

    $$\delta_0 \cdot o_1 \cdot o_2 \cdot \delta_1 \backsim \delta_0 \cdot o_2 \cdot o_1 \cdot \delta_1$$

    whenever $o_1$ # $o_2$
  - In a nutshell, reordering non-conflicting events renders equivalent histories

A Theory of Database Concurrency Control
[Papadimitriou 1986]

# 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 $o \in \Sigma$: $push(v)$, $ret_{push}$, $pop()$, $ret_v$, $wr_{p,v}$, $rd_{p,v}$, $cas_{p,v,w}$
- Events: $\Sigma_t = \{(t,a) | a \in \Sigma, t \in T\}$
- 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 $\delta$ is (Conflict) Serializable if it has a *conflict equivalent serial history*

# Conflict Serializability Monitoring

- Definition: (Conflict Graph)
  If operations $o_1, o_2, \ldots, o_n$ happen in $\delta$. The conflict graph CG($\delta$) of $\delta$ is a tuple CG($\delta$) = (V, E, S) where:
  - V = ($o_1, o_2, \ldots, o_n$),
  - S : V $\longrightarrow 2^{\Sigma}$ where S($o_i$) is the set of events of $o_i$ in $\delta$
  - $o_i \longrightarrow o_j \in$ E if there are events $(t, a) \in$ S($o_i$) and $(t', a') \in$ S($o_i$) such that $a \# a'$, and $a$ appears in $\delta$ before $a'$
- Lemma:
  There exists a cycle in $\delta$ iff $\delta$ is not conflict serializable

# Conflict Serializability Monitoring

- So, we could record the Conflict Graph, and check for cycles at every operation:

# Conflict Serializability Monitoring

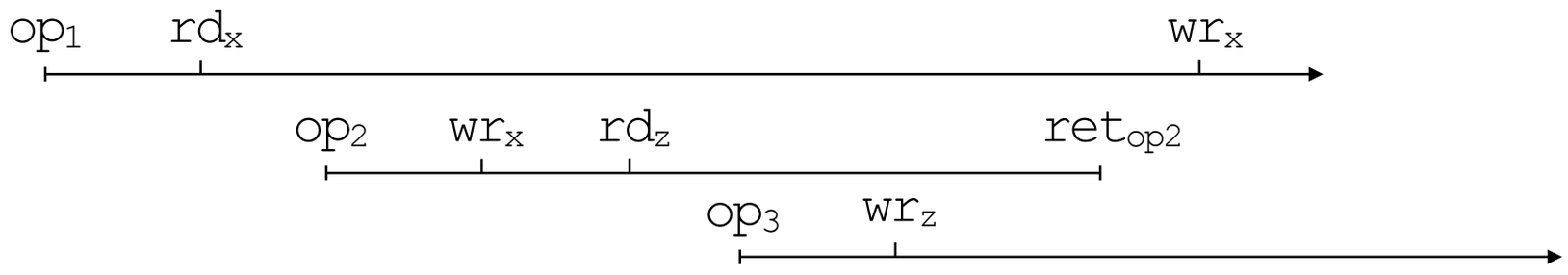
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(

# 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?

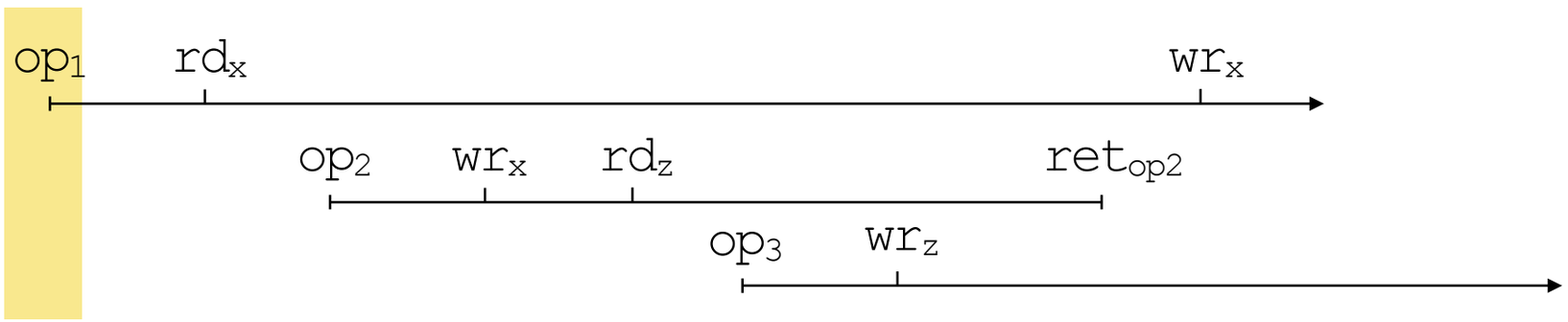
# Conflict Serializability Monitoring

- 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?



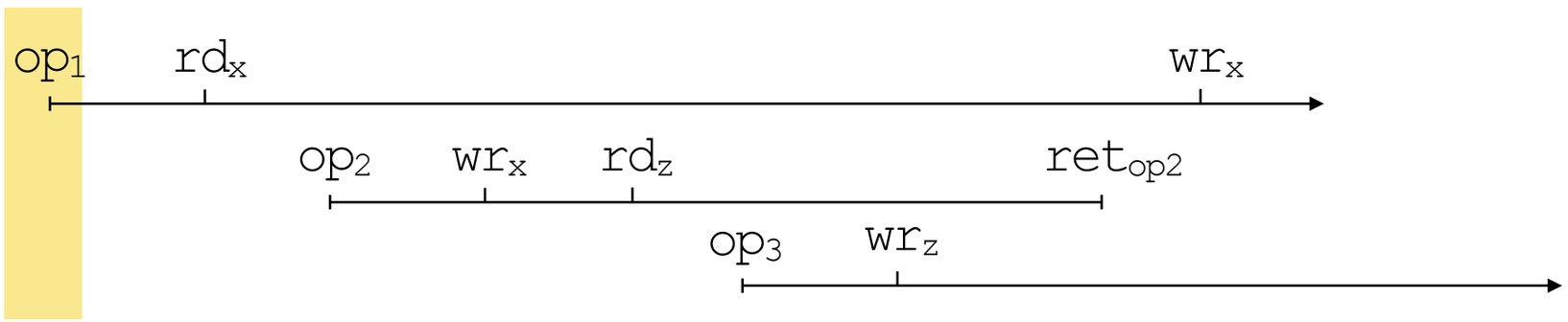
# Conflict Serializability Monitoring

- 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

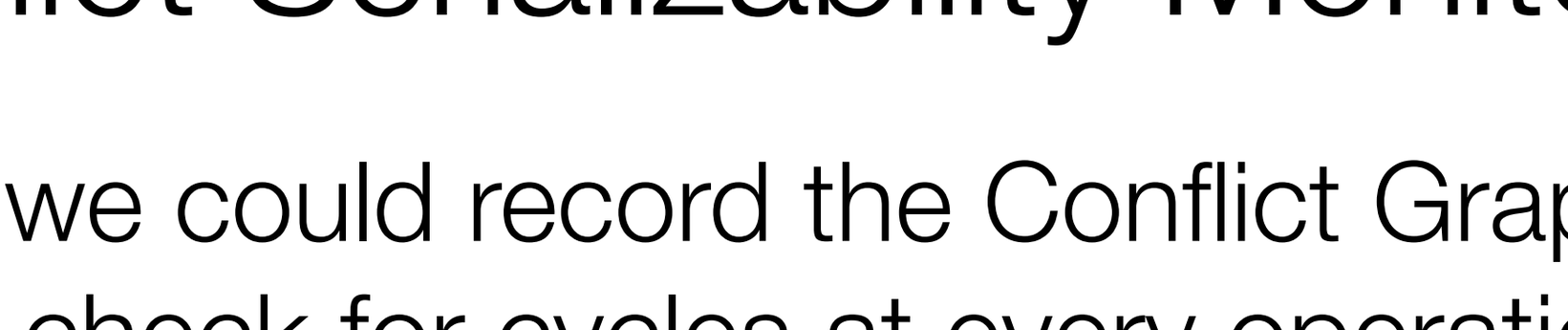
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?




Serialization $op_1$

# Conflict Serializability Monitoring

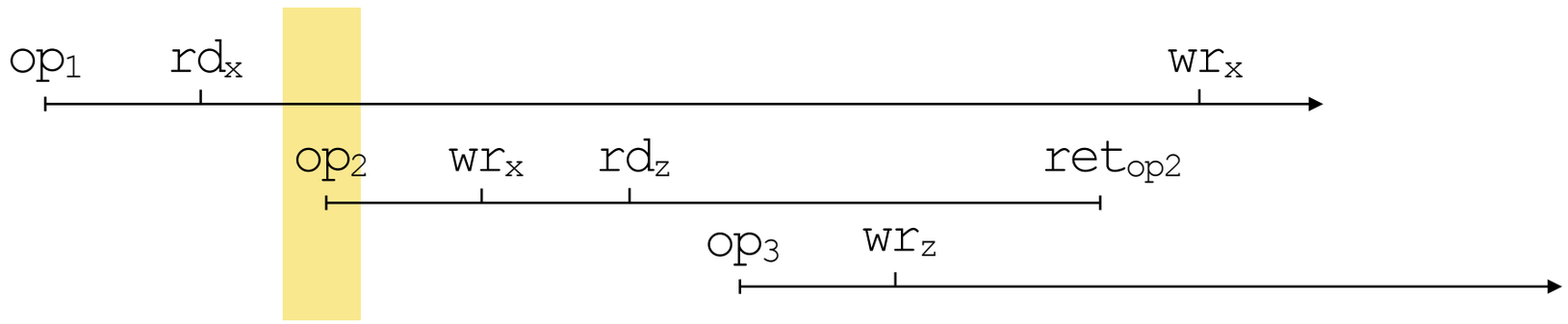
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?




Serialization op1

# Conflict Serializability Monitoring

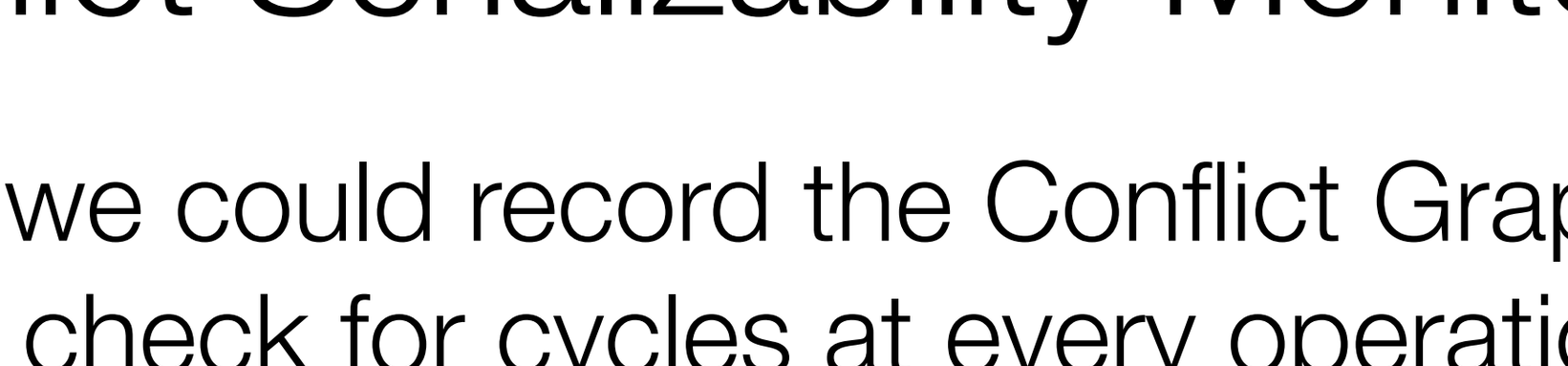
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?




Serialization op1 op2

# Conflict Serializability Monitoring

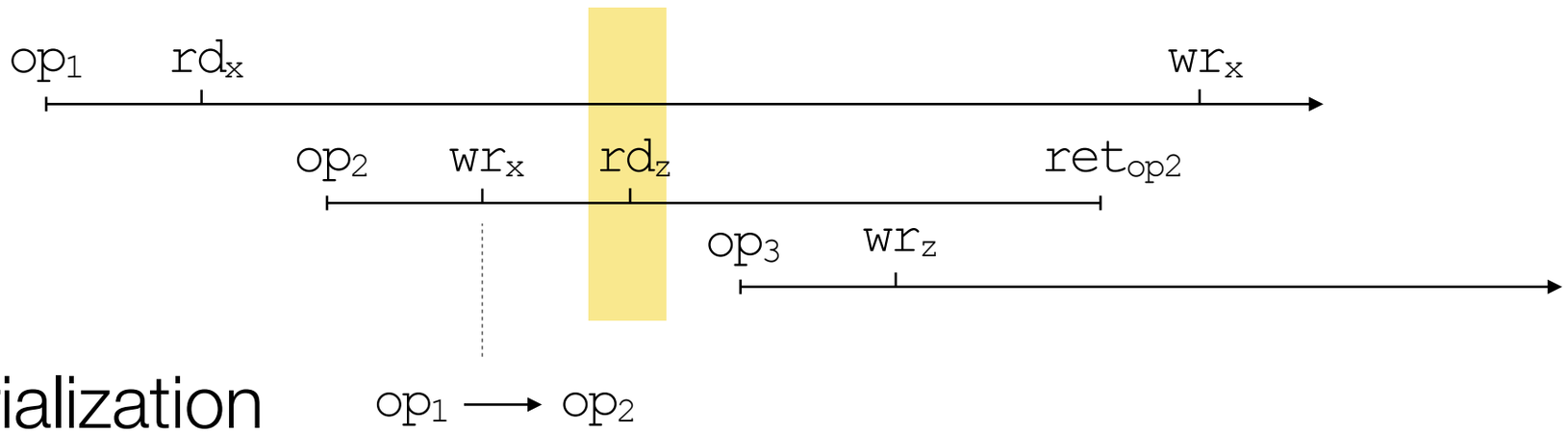
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?




Serialization $op_1 \longrightarrow op_2$

# Conflict Serializability Monitoring

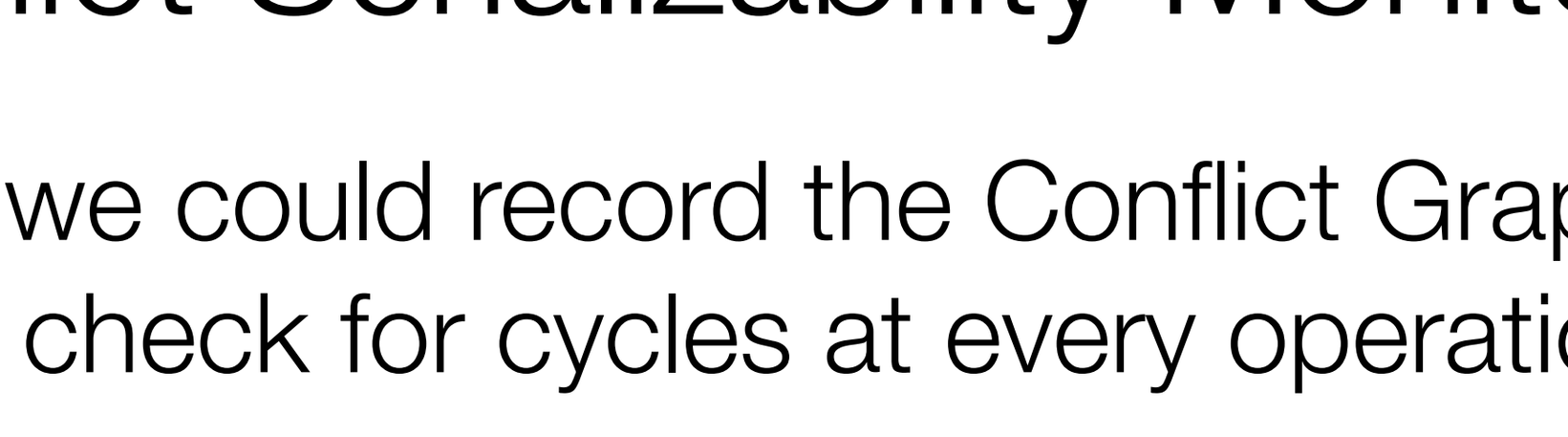
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?




Serialization $op_1 \longrightarrow op_2$

# Conflict Serializability Monitoring

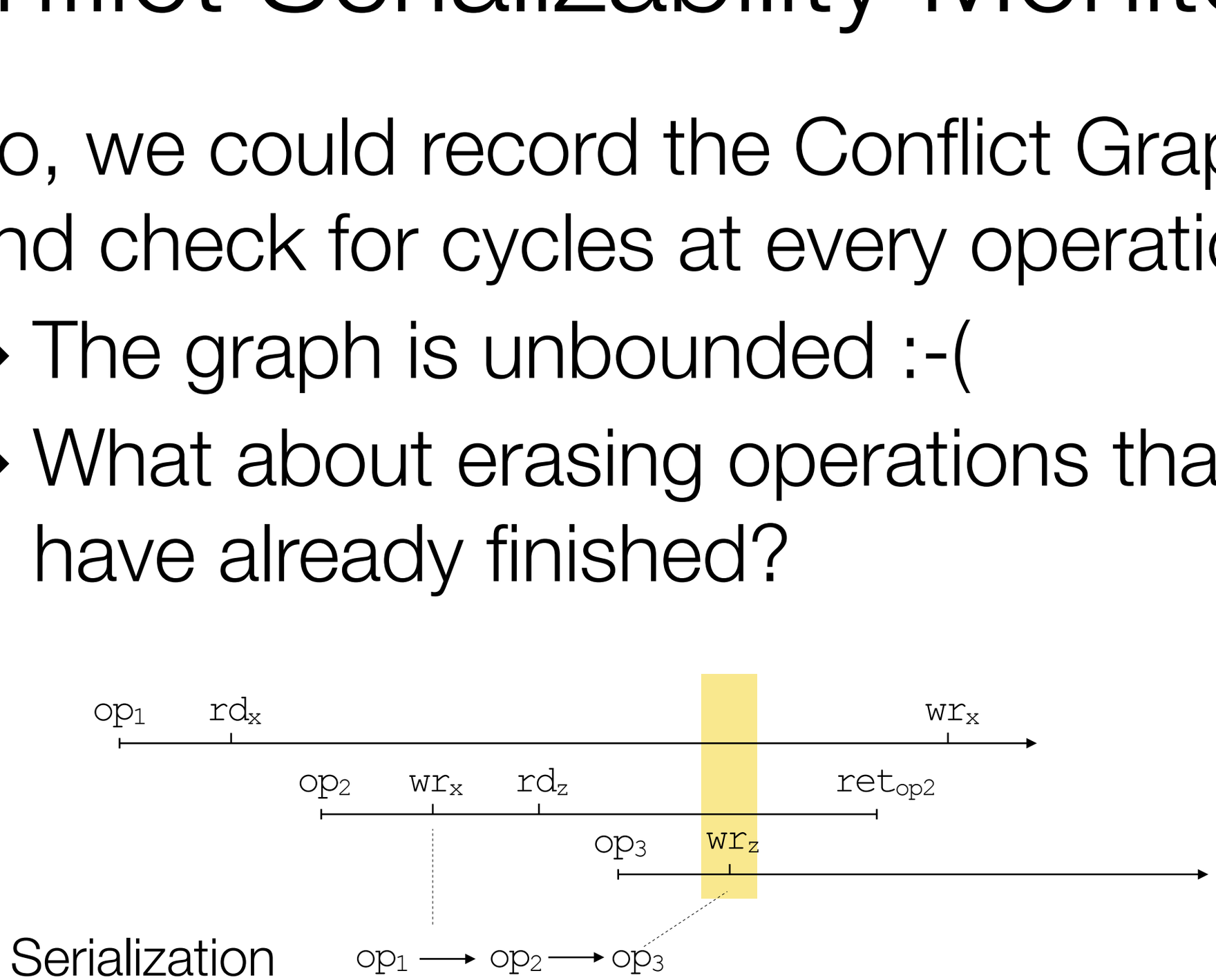
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?




Serialization $op_1 \longrightarrow op_2$ $op_3$

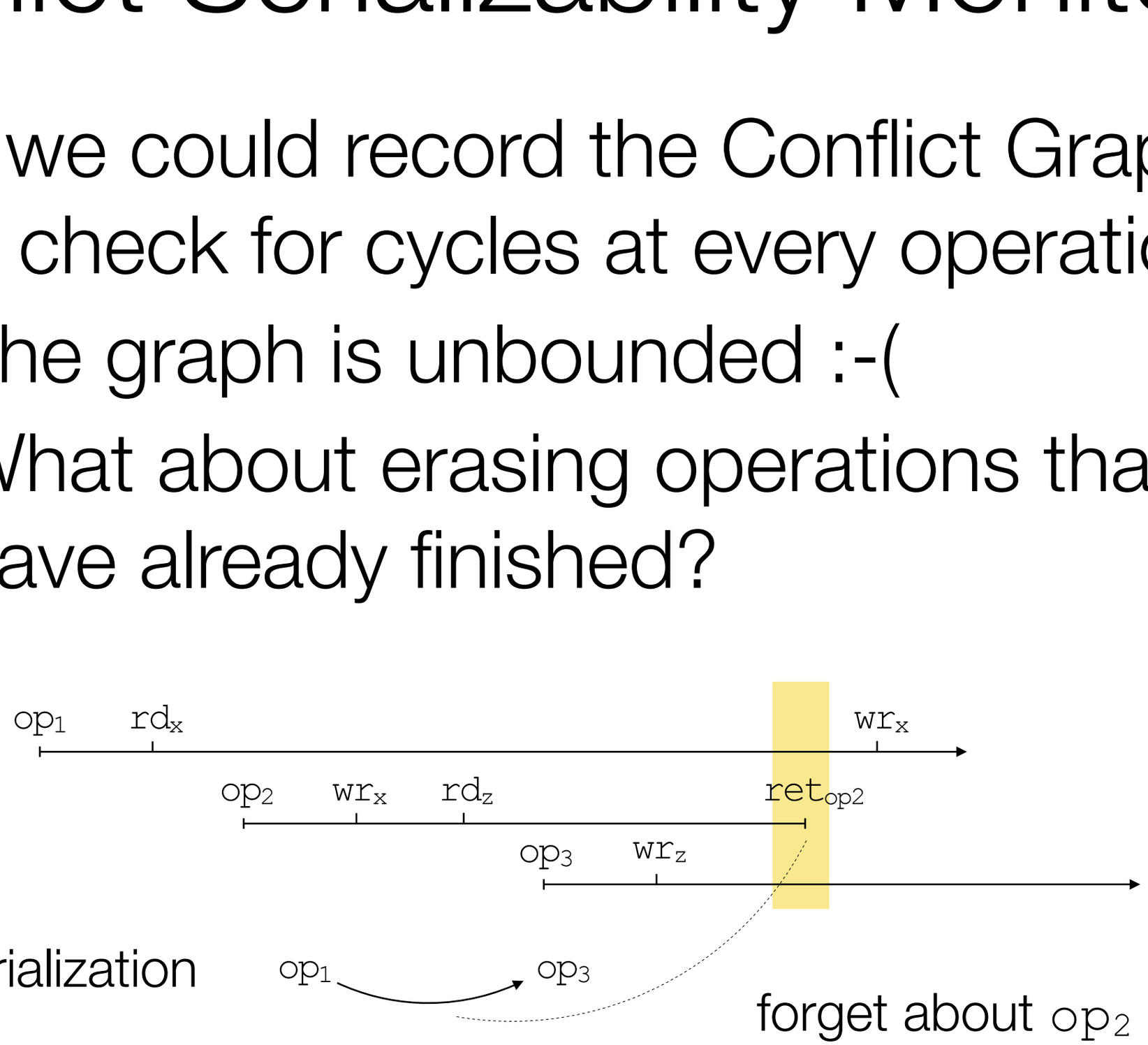
# Conflict Serializability Monitoring

- 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?



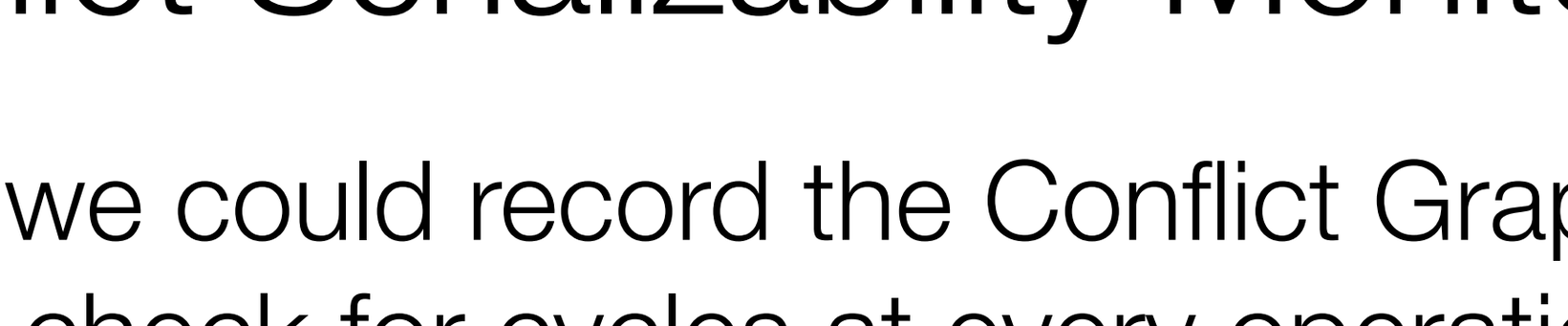
# Conflict Serializability Monitoring

- 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?

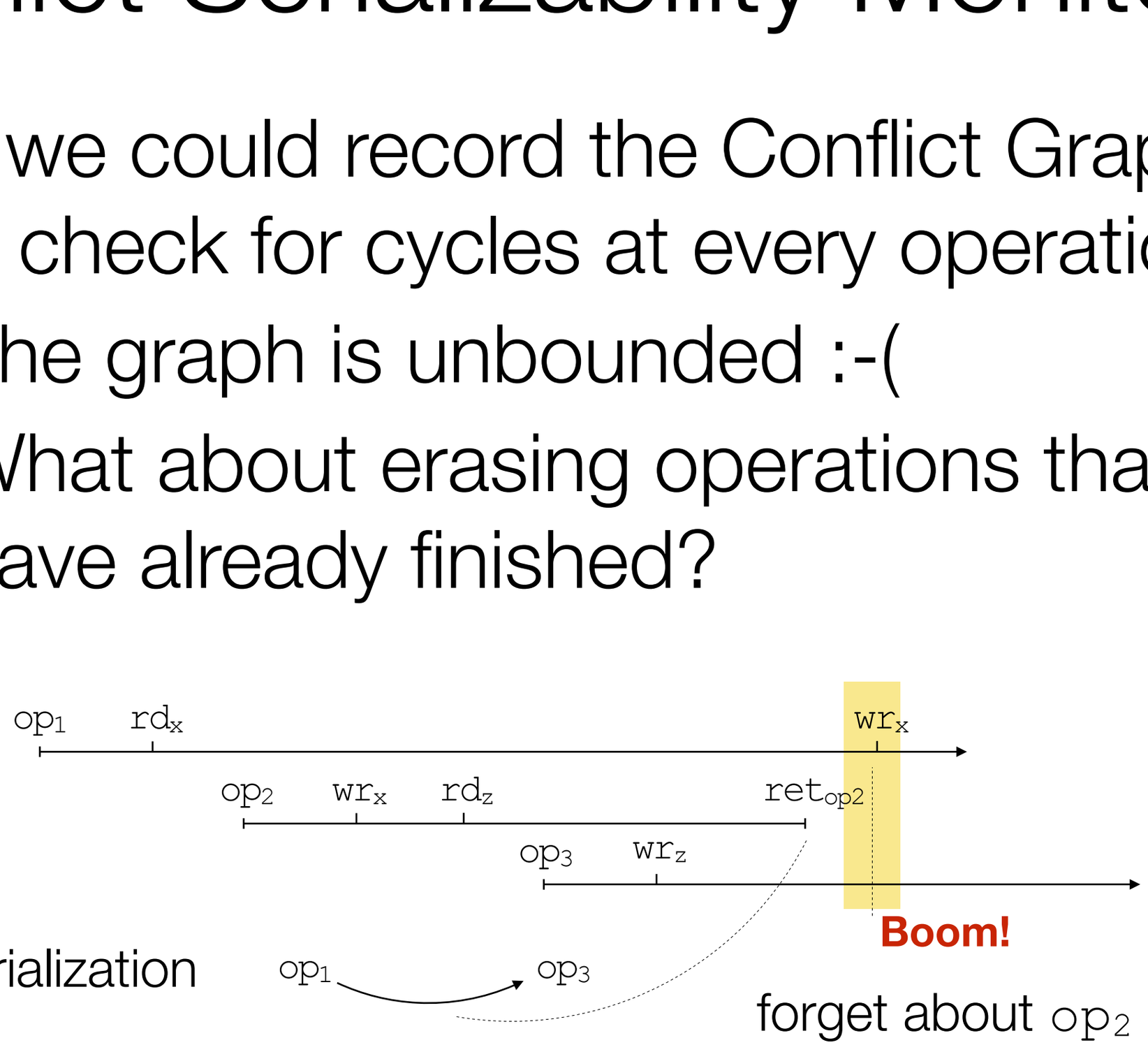



Serialization


forget about op2

# Conflict Serializability Monitoring

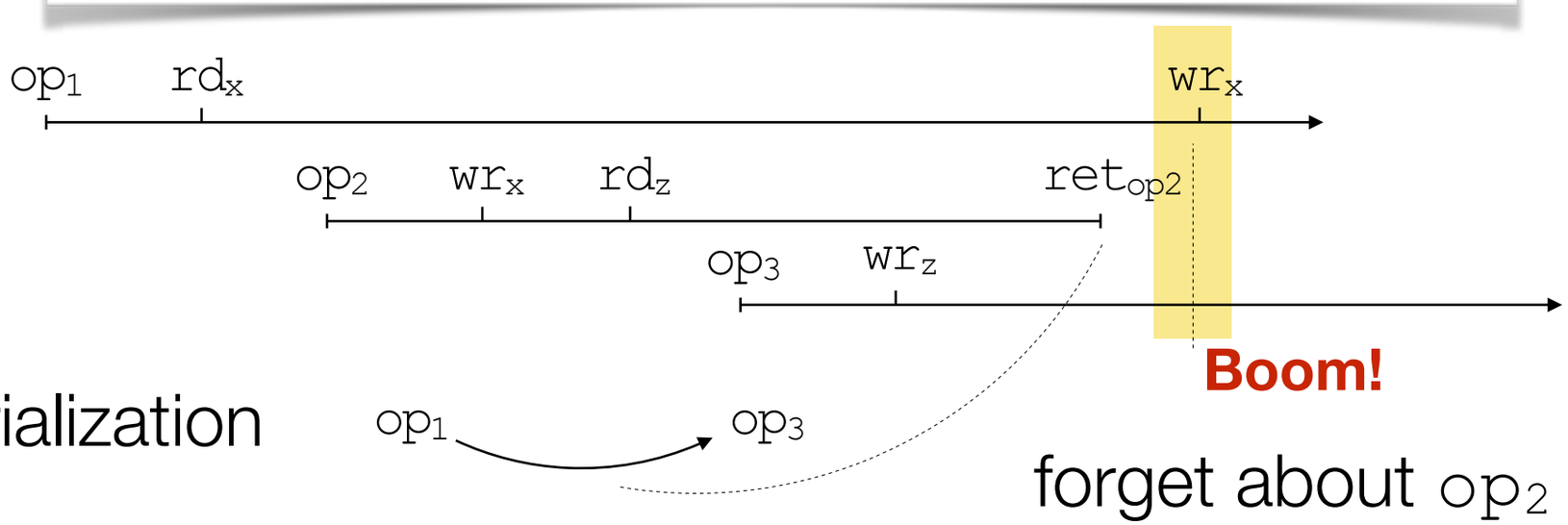
- So, we could record the Conflict Graph, and check for cycles at every operation:
  - The graph is unbounded :-(
  - What about erasing operations that have already finished?



# 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 ha

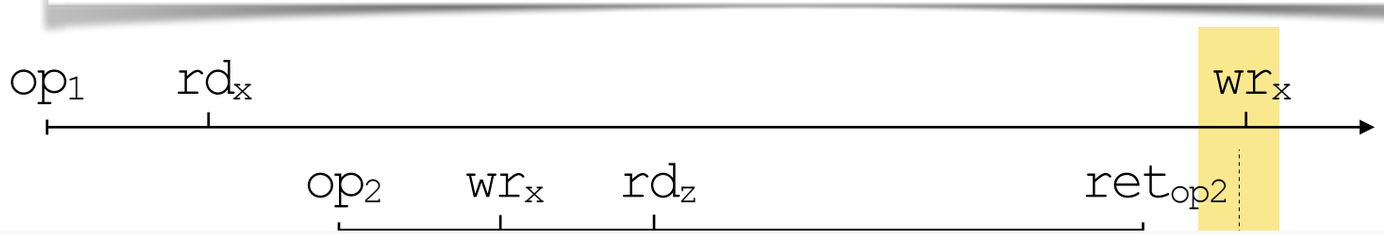
Forget the operations but not the conflicts they induced



# 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 ha

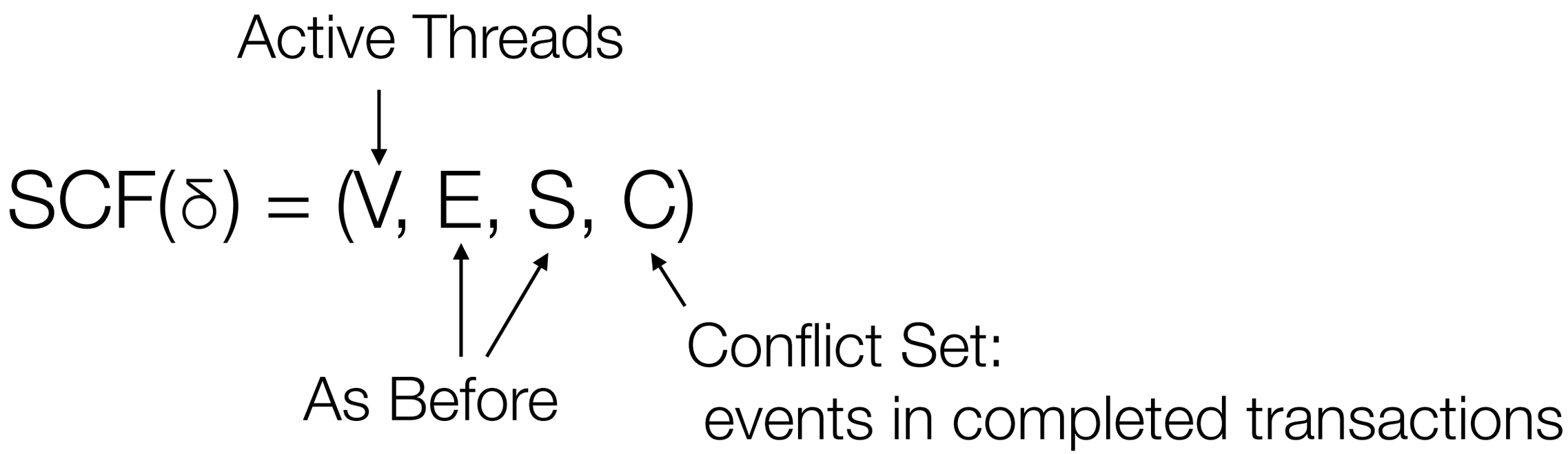
Forget the operations but not the conflicts they induced




But recording every event of every transaction is too expensive in memory

# Summarized CG

- Keep track of active threads only
- Paths between active transactions are summarized as edges
- Keep set of conflicts C, events in transactions that must be scheduled after the current one



# Conflict Serializability Monitoring

- Definition: [SCG] Given CF(δ), we define SCF(δ) = (V,E,S,C) where:

  1. V contains a node $v_i$ for each *active* thread $t$ (executing $(t,o_i)$)

  2. $v \longrightarrow v'$ (executing $(t,o_i)$ and $(t',o_i)$ resp.) appears in E if there is $o_i \dashrightarrow o_i$ a *path* in CG(δ) which contains no other node $v^*$ (in V)

  3. if $v \in V$ (corresponding to $(t,o_i)$ ) then S($(t,o_i)$) = S($o_i$) ∈ CG(δ) and C($v$) contains:

     a) $(t',a)$ such that $t'$ is an active thread, and there is some completed $o_j$ with $o_i \dashrightarrow o_j$ a *path* in CG(δ), where $(t',a) \in S(o_j)$

     b) $a$ such that there is a completed $o_j$ in CG(δ) with $o_i \dashrightarrow o_j$ and $(t',a) \in S(o_j)$ where $t'$ has terminated.

- Lemma:

  CG(δ) contains a cycle iff SCG(δ') contains a cycle for δ' a prefix of δ

# Conflict Serializability Monitoring

- Algorithm: we consider the next event $v$ in $\delta$
  - $v=(t,call_{op})$: create a vertex $v$ and set S($v$) = ∅ and C($v$) = ∅
  - $v=(t,ret_{op})$: for each $v'$ and $v^*$ such that $v' \longrightarrow v \longrightarrow v^*$ appears in E, remove $v$ and add edge $v' \longrightarrow v^*$ to E.
    Set C($v'$)=C($v'$)∪S($v$)∪C($v$).
  - $v=(t,a)$: Set S($v$) = S($v$) ∪ {$(t,a)$}.
    For each $v'$ in V such that $(t^*,a^*)$ appears in C($v'$) ∪ S($v'$) and $(t^*,a^*)\#(t,a)$, add $v' \longrightarrow v$ to E. Also, for any action $b$ in C($v'$) with $b\#a$ $v' \longrightarrow v$ 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

Happy Holidays!