

CONCURRENT DATA STRUCTURES LINKED IN TIME

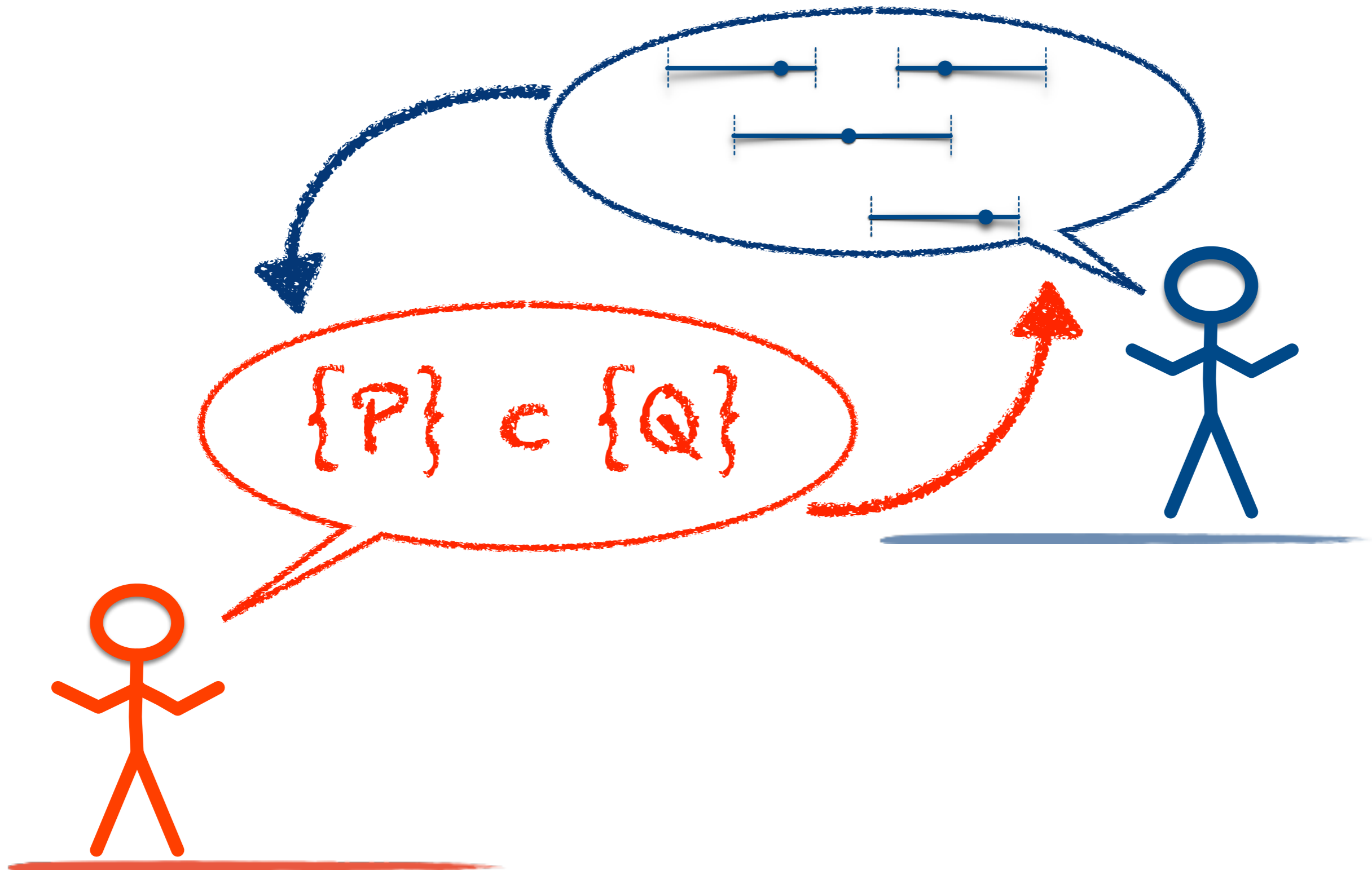


Germán Delbianco

Aleks Nanevski

Ilya Sergey

Anindya Banerjee



DATA STRUCTURES LINKED IN TIME

Linearization Points
= Pointers in Time

- Encode **linearization order** as auxiliary state in FCSL.
- Reasoning about (**non-fixed, non-local, non-regional**) LPs reduced to pointer-like manipulations on **auxiliary state**.
- Mechanised a complex snapshot algorithm by Jayanti.

Concurrent Data Structures Linked in Time

Germán Andrés Delbianco^{1,2}, Ilya Sergey², Aleksandar Nanevski¹, and Anindya Banerjee¹

- ¹ IMDEA Software Institute, Madrid, Spain
{german.delbianco, alexa.nanevski, anindya.banerjee}@imdea.org
- ² University College London, UK
i.sergey@ucl.ac.uk
- ³ Universidad Politécnica de Madrid, Spain

Abstract

Arguments about linearizability of a concurrent data structure are typically carried out by specifying the linearization points of the data structure's procedures. Proofs that use such specifications are often cumbersome as the linearization points' position in time can be dynamic, non-local and non-regional: it can depend on the interference, runtime values and events from the past, or even future, appear in procedures other than the one considered, and might be only determined after the considered procedure has terminated.

In this paper we propose a new method, based on a Hoare-style logic, for reasoning about concurrent objects with such linearization points. We embrace the dynamic nature of linearization points, and encode it as part of the data structure's auxiliary state, so that it can be dynamically modified in place by auxiliary code, as needed when some appropriate run-time event occurs.

We name the idea *linking-in-time* because it reduces temporal reasoning to spatial reasoning. For example, modifying a temporal position of a linearization point can be modeled similarly to a pointer update in a heap. We illustrate the method by verifying an intricate optimal snapshot algorithm due to Jayanti.

1 Introduction

Formal verification of concurrent objects commonly requires reasoning about linearizability [11]. This is a standard correctness criterion whereby a concurrent execution of an object's procedures is proved equivalent, via a simulation argument, to some sequential execution. The clients of the object can be verified under the sequentiality assumption, rather than by inlining the procedures and considering their interleavings. Linearizability is often established by describing the *linearization points* (LP) of the object, which are points in time where procedures take place, *logically*. In other words, even if the procedure physically executes across a time interval, exhibiting its linearization point enables one to pretend, for reasoning purposes, that it occurred instantaneously; hence, an interleaved execution of a number of procedures can be reduced to a sequence of instantaneous events.

However, reasoning with linearizability, and about linearization points, can be tricky. Many times, a linearization point of a procedure is not *local*, but may appear in another procedure or thread. Equally bad, a linearization points' place in time may not be determined statically, but may vary based on the past, and even future, *run-time* information. This complicates the simulation arguments, leading to unwieldy formal logical proofs.

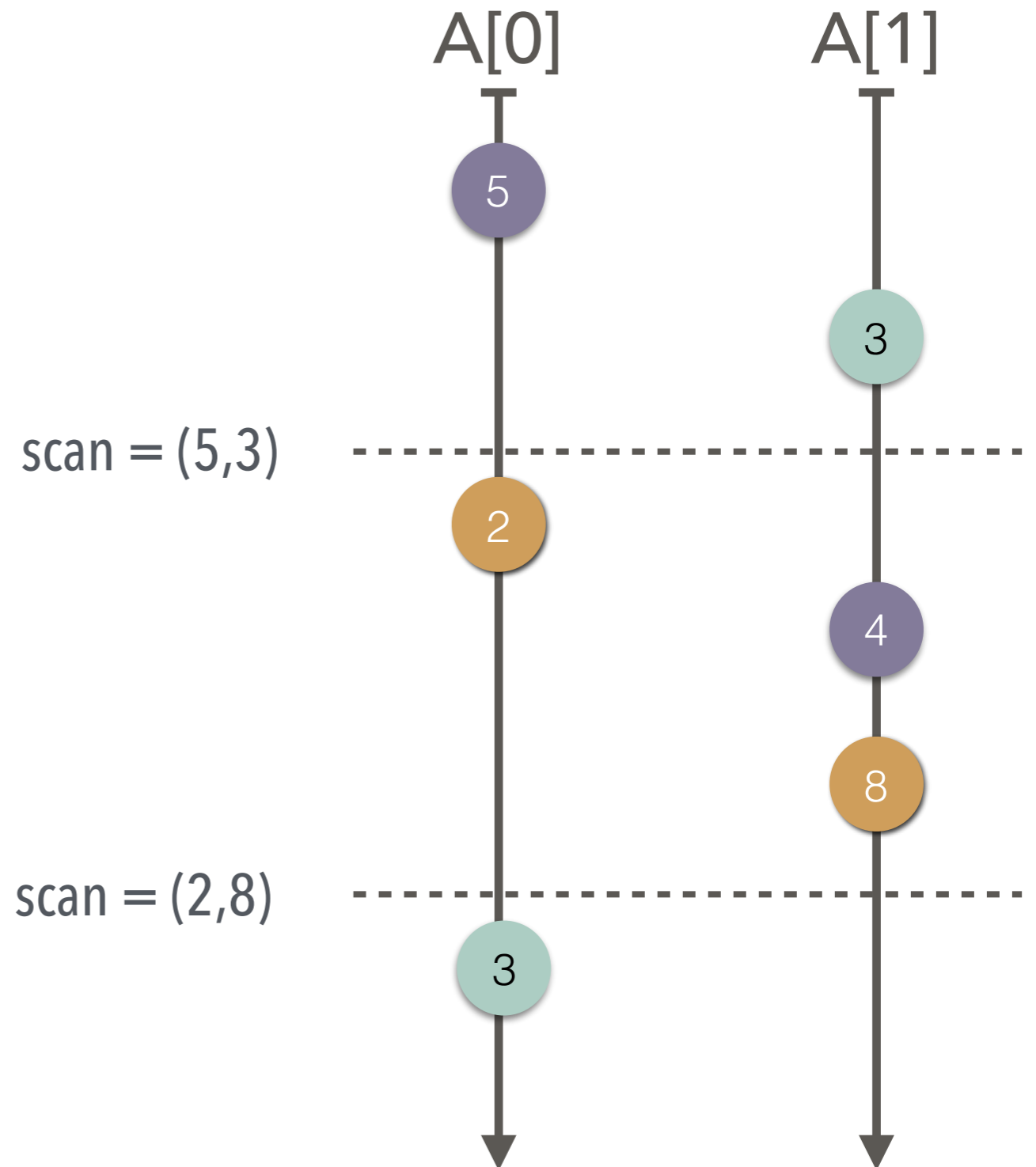
This paper presents a novel specification and verification method for concurrent objects, based on Hoare logic. It achieves the same goal as linearizability, that is, client proofs can reason out of Hoare triples of the object's procedures, rather than inline the procedures and consider all interleavings. The method improves on linearizability by offering natural abstractions for specifying dynamic and non-local linearization points, based on familiar concepts from Hoare logics for shared-memory concurrency. It also permits better information

© Germán Andrés Delbianco and Ilya Sergey and Aleksandar Nanevski and Anindya Banerjee;
licensed under Creative Commons License CC-BY
LIPDES: Linking-In-Time: Verifying Linearizability in Informal
EASIS Report - Technical Report 16, Technical Report, Digital Publishing, OpenM

(Delbianco, Sergey, Nanevski & Banerjee :ECOOP17)

CONCURRENT SNAPSHOTS

- Fine-grained **concurrent object** with a shared memory object e.g. array.
- **WAIT FREE** scan and write.
- **Scan** returns a memory **snapshot**: a collection of values that **co-existed** in memory.



JAYANTI'S SNAPSHOT

- Optimal $O(m)$ *wait-free* scan with non-trivial correctness.

```
write (i, v) {  
1.   A[i] := v;  
2.   if S  
3.   then B[i] := v  
}
```

```
scan : array nat {  
4.   S := true;  
5.   for i = 0..n do B[i] :=  $\perp$ ;  
6.   for i = 0..n do V[i] := A[i];  
7.   S := false;  
8.   for i = 0..n do  
9.     v = B[i];  
10.    if (v  $\neq$   $\perp$ ) then V[i] := v;  
11.  return V  
}
```

JAYANTI'S SNAPSHOT

- Optimal $O(m)$ *wait-free* scan with non-trivial correctness.
- Shared arrays **A** and **B**, shared bit **S**.

```
write (i, v) {  
1.   A[i] := v;  
2.   if S  
3.   then B[i] := v  
}
```

```
scan : array nat {  
4.   S := true;  
5.   for i = 0..n do B[i] :=  $\perp$ ;  
6.   for i = 0..n do V[i] := A[i];  
7.   S := false;  
8.   for i = 0..n do  
9.     v = B[i];  
10.    if (v  $\neq$   $\perp$ ) then V[i] := v;  
11.  return V  
}
```

JAYANTI'S SNAPSHOT

- Optimal $O(m)$ *wait-free* scan with non-trivial correctness.
- Shared arrays **A** and **B**, shared bit **S**.
- Single scanner/writer.

```
write (i, v) {  
1.   A[i] := v;  
2.   if S  
3.   then B[i] := v  
}
```

```
scan : array nat {  
4.   S := true;  
5.   for i = 0..n do B[i] :=  $\perp$ ;  
6.   for i = 0..n do V[i] := A[i];  
7.   S := false;  
8.   for i = 0..n do  
9.     v = B[i];  
10.    if (v  $\neq$   $\perp$ ) then V[i] := v;  
11.  return V  
}
```

JAYANTI'S SNAPSHOT

- Optimal $O(m)$ *wait-free* scan with non-trivial correctness.
- Shared arrays A and B , shared bit S .
- Single scanner/writer.
- **write** might forward the written value. Or not.

```
write (i, v) {  
1.   A[i] := v;  
2.   if S  
3.   then B[i] := v  
}
```

```
scan : array nat {  
4.   S := true;  
5.   for i = 0..n do B[i] :=  $\perp$ ;  
6.   for i = 0..n do V[i] := A[i];  
7.   S := false;  
8.   for i = 0..n do  
9.     v = B[i];  
10.    if (v  $\neq$   $\perp$ ) then V[i] := v;  
11.  return V  
}
```


JAYANTI'S SNAPSHOT

- Optimal $O(m)$ *wait-free* scan with non-trivial correctness.
- Shared arrays A and B , shared bit S .
- Single scanner/writer.
- **write** might forward the written value. Or not.
- Hows does scan compute a snapshot? Is it a valid snapshot?

```
write (i, v) {  
1.   A[i] := v;  
2.   if S  
3.   then B[i] := v  
}
```

```
scan : array nat {  
4.   S := true;  
5.   for i = 0..n do B[i] :=  $\perp$ ;  
6.   for i = 0..n do V[i] := A[i];  
7.   S := false;  
8.   for i = 0..n do  
9.     v = B[i];  
10.    if (v  $\neq$   $\perp$ ) then V[i] := v;  
11.  return V  
}
```

JAYANTI'S SNAPSHOTS

```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0 .. n$  do B[i] :=  $\perp$ ;  
6.   for  $i = 0 .. n$  do V[i] := A[i];  
7.   S := false;  
8.   for  $i = 0 .. n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then V[i] := v;  
11.  return V  
}
```

JAYANTI'S SNAPSHOTS

{


```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0..n$  do  $B[i] := \perp$ ;  
6.   for  $i = 0..n$  do  $V[i] := A[i]$ ;  
7.   S := false;  
8.   for  $i = 0..n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then  $V[i] := v$ ;  
11.      return  $V$   
}
```

PRELUDE : COLLECT
ORIGINAL VALUES FROM A

JAYANTI'S SNAPSHOTS

```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0 .. n$  do  $B[i] := \perp$ ;  
6.   for  $i = 0 .. n$  do  $V[i] := A[i]$ ;  
7.   S := false;  
8.   for  $i = 0 .. n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then  $V[i] := v$ ;  
11.      return  $V$   
}
```

JAYANTI'S SNAPSHOTS

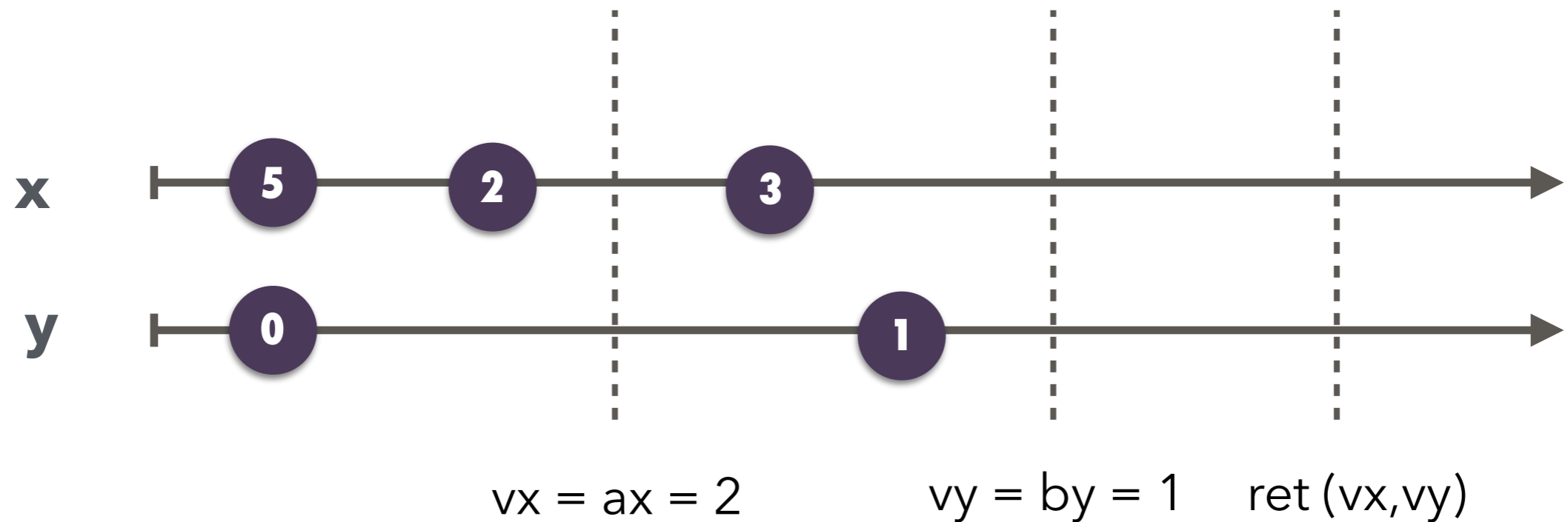


```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0..n$  do  $B[i] := \perp$ ;  
6.   for  $i = 0..n$  do  $V[i] := A[i]$ ;  
7.   S := false;  
8.   for  $i = 0..n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then  $V[i] := v$ ;  
11.      return  $V$   
}
```

EPILOGUE : UPDATE WITH
FORWARDED VALUES FROM B

JAYANTI'S SNAPSHOTS

```
l: write(x,2); || c: scan() || r: write(x,3)
   write(y,1)
```



SCAN RETURNS (2,1) !

An Optimal Multi-Writer Snapshot Algorithm *

[Extended Abstract]

Prasad Jayanti
Dartmouth College
6211 Sudikoff Lab for Computer Science
Hanover, NH 03755
USA
prasad.jayanti@dartmouth.edu

ABSTRACT

An m -component, n -process snapshot object is an abstraction of shared memory that consists of m words and allows up to n processes to concurrently execute the following two types of operations: $write(i, v)$, which writes v into the i th word, and $scan()$, which returns the current values of all m locations [1, 3]. The snapshot problem is to design algorithms for the write and scan operations that meet two challenging requirements: (1) operations appear to be atomic, and (2) operations are wait-free.

For any (m -component, n -process) snapshot algorithm, which runs on hardware that supports only word-sized objects, $\Omega(1)$ and $\Omega(m)$ are trivial lower bounds on the time complexity of $write(i, v)$ and $scan()$, respectively. But, are these bounds tight?

For a restricted version of the snapshot problem, known in the literature as the single-writer snapshot problem, Riany, Shavit and Touitou [18] showed that the answer is yes: they designed an algorithm with $O(1)$ and $O(m)$ running times for the $write(i, v)$ and $scan()$ operations, respectively. (The single-writer snapshot problem assumes that (i) the number m of words of the snapshot object is equal to the number n of processes, and (ii) only the i th process may write into the i th snapshot word.)

This paper shows that the same (optimal) running times of $O(1)$ for $write(i, v)$ and $O(m)$ for $scan()$ are achievable for the general problem, known in the literature as the multi-writer snapshot problem. Our algorithm requires hardware support for the CAS (compare&swap) operation (in comparison, Riany, Shavit and Touitou's algorithm requires hardware support for CAS, $fetch\&inc$, and $fetch\&dec$ operations).

*We gratefully acknowledge the equipment and system support from the NSF grant EIA-9802068.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

STOC'05, May 22-24, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-58113-960-8/05/0005 ...\$5.00.

Categories and Subject Descriptors

F.2 [Analysis of Algorithms and Problem Complexity]: Miscellaneous

General Terms

Algorithms, Reliability

Keywords

asynchronous, concurrent algorithm, fault-tolerant, lock-free, snapshot, wait-free

1. INTRODUCTION

In a multiprocess system, the hardware allows concurrent processes to execute atomic read and write operations on individual words of memory. But what if processes require the capability to atomically scan a large chunk of memory that consists of many words, even as other processes concurrently write into some of these words? Clearly, such a capability can only be provided in software through the design of suitable algorithms. This observation motivates the snapshot problem, formulated and first solved by Afek et al. [1] and Anderson [3], which we describe below.

An m -component, n -process snapshot object is an abstraction of a shared memory that consists of m words and allows up to n processes to concurrently execute any of the following two types of operations: $write(i, v)$, which writes v into the i th word, and $scan()$, which returns the current values of all m locations. The problem is to implement this abstraction, i.e., design algorithms for the $write(i, v)$ and $scan()$ operations, such that two challenging requirements are met: (1) the write and scan operations must be atomic: they must appear to act instantaneously, or more technically, they must be linearizable [12], and (2) the write and scan operations must be wait-free: any process should be able to execute either operation in a bounded number of its own steps, regardless of whether other processes (which may be concurrently executing write and scan operations) slow down, speed up or even crash.

The above version, which is the general version of the snapshot problem, is known in the literature as the multi-writer snapshot problem. A simpler version, known as the single-writer snapshot problem, assumes that (i) $m = n$ (the number of words of the snapshot object is equal to the number of processes), and (ii) the i th snapshot word may be

(Jayanti:STOC'05)

An Optimal Multi-Writer Snapshot Algorithm *

[Extended Abstract]

Prasad Jayanti
Dartmouth College
6211 Sudikoff Lab for Computer Science
Hanover, NH 03755
USA
orasad.jayanti@dartmouth.edu

THEOREM 1. *If Scan operations are executed one after another, without overlap, and Write operations to the same component are executed one after another, without overlap, the algorithm in Figure 1 correctly implements a linearizable m -component snapshot for any number of processes. The time complexity of Scan and Write operations are $O(m)$ and $O(1)$, respectively, and the space complexity is $O(m)$.*

ware support for the CAS (compare&swap) operation (in comparison, Riany, Shavit and Touitou's algorithm requires hardware support for CAS, *fetch&inc*, and *fetch&dec* operations).

*We gratefully acknowledge the equipment and system support from the NSF grant EIA-9802068.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

STOC'05, May 22-24, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-58113-960-8/05/0005 ...\$5.00.

v into the i th word, and *scan()*, which returns the current values of all m locations. The problem is to implement this abstraction, i.e., design algorithms for the *write(i, v)* and *scan()* operations, such that two challenging requirements are met: (1) the *write* and *scan* operations must be atomic: they must appear to act instantaneously, or more technically, they must be linearizable [12], and (2) the *write* and *scan* operations must be wait-free: any process should be able to execute either operation in a bounded number of its own steps, regardless of whether other processes (which may be concurrently executing *write* and *scan* operations) slow down, speed up or even crash.

The above version, which is the general version of the snapshot problem, is known in the literature as the multi-writer snapshot problem. A simpler version, known as the single-writer snapshot problem, assumes that (i) $m = n$ (the number of words of the snapshot object is equal to the number of processes), and (ii) the i th snapshot word may be

(Jayanti:STOC'05)

Linearizability: A Correctness Condition for Concurrent Objects

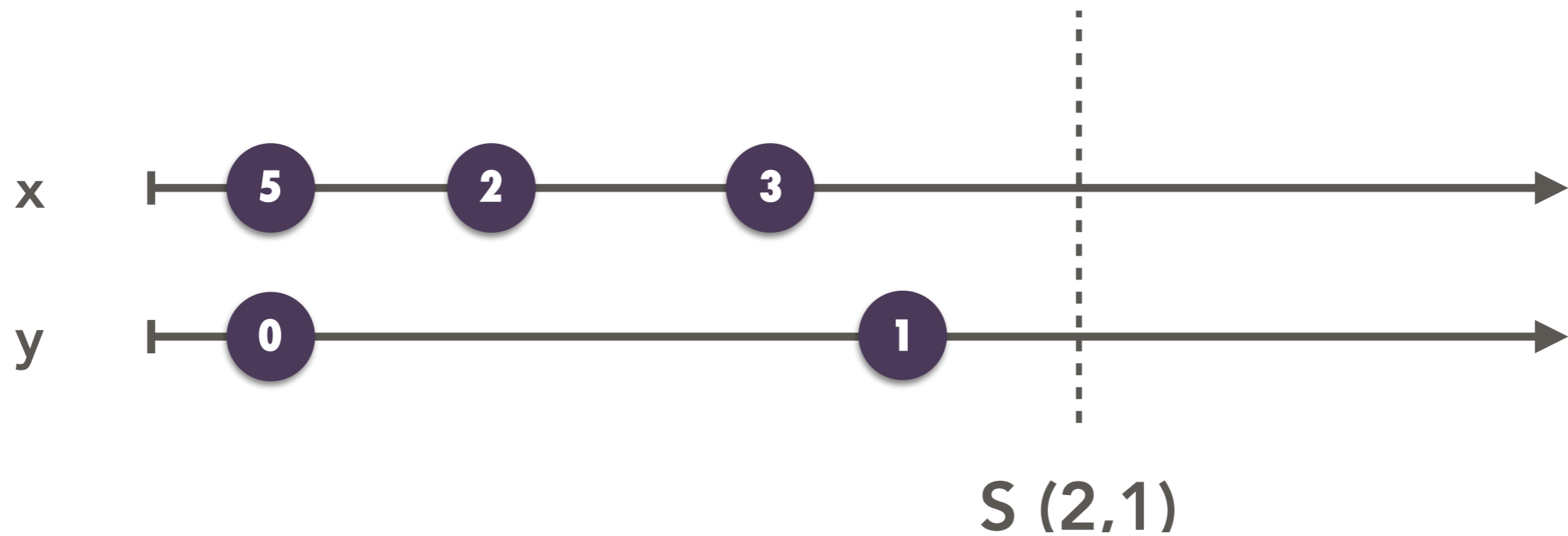
MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

(Herlihy-Wing:TOPLAS'90)

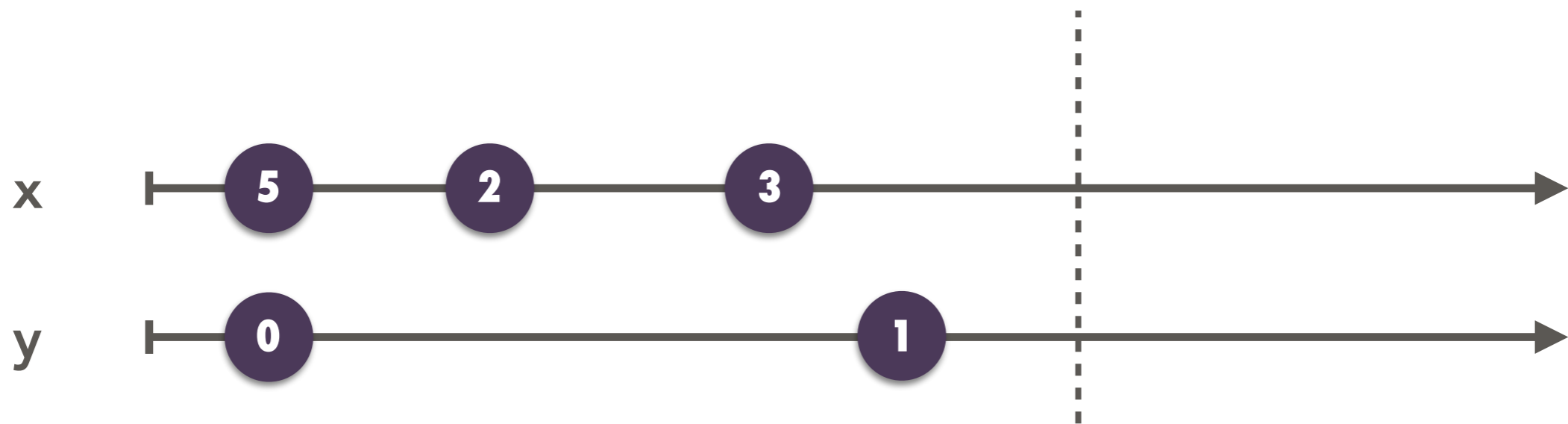
LINEARIZABILITY TO THE RESCUE

```
write (x,2); || scan () || write (x,3)  
write (y,1)
```



LINEARIZABILITY TO THE RESCUE

```
write (x,2); || scan () || write (x,3)
write (y,1)
```

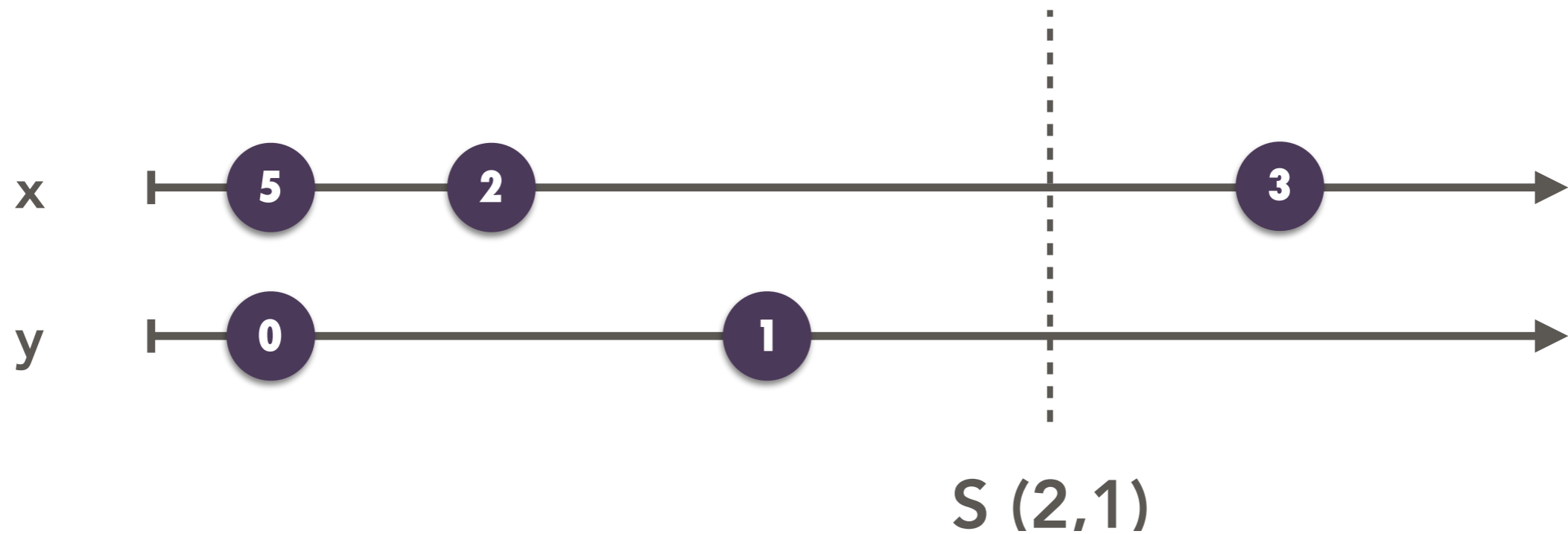


(2,1) was NOT a snapshot, but it COULD have been!

S (2,1)

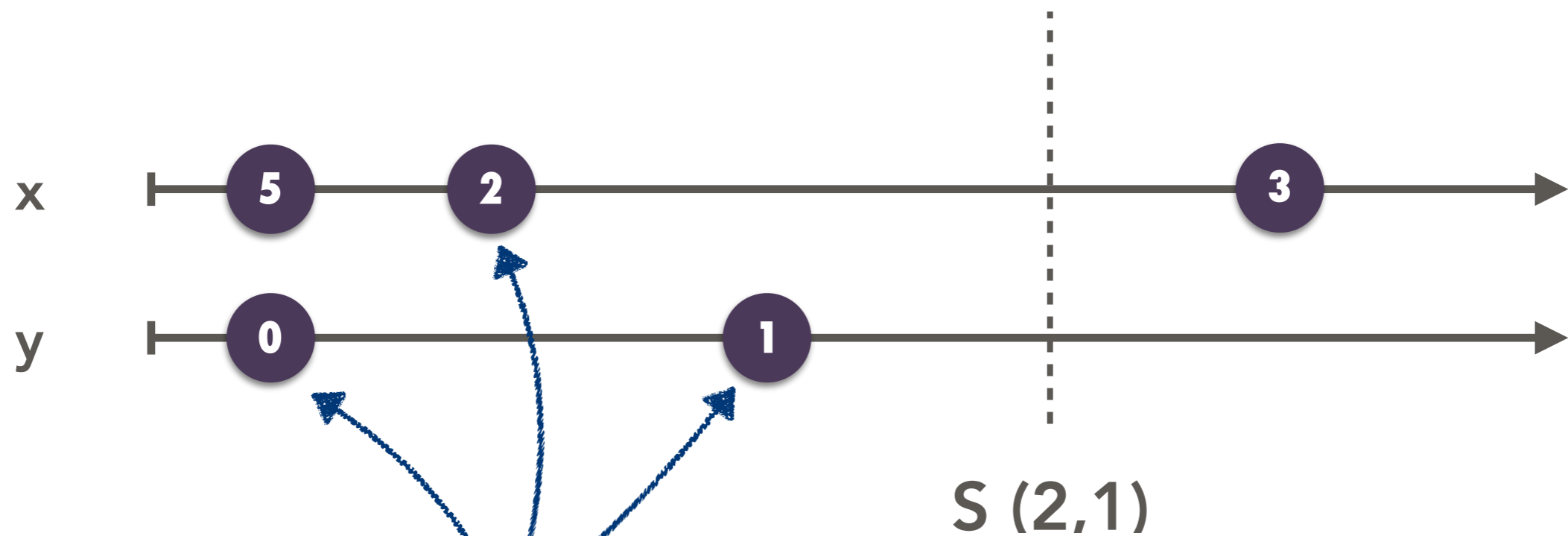
LINEARIZABILITY TO THE RESCUE

```
write(x, 2); write(y, 1); scan(); write(x, 3)
```



LINEARIZABILITY TO THE RESCUE

```
write(x, 2); write(y, 1); scan(); write(x, 3)
```



LP = logical, atomic, moment where action happens

- [12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.
- [13] P. Jayanti. *f*-arrays: implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270 – 279, 2002.
- [14] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.
- [15] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword ll/sc variables. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, June 2005.
- [16] L. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pages 229–241, 1991.
- [17] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.
- [18] Y. Riaz, N. Shavit, and D. Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1-2):163–201, 2001.

APPENDIX

A. PROOF OF CORRECTNESS OF THE BASIC ALGORITHM

In this appendix, we sketch the proof that the basic algorithm in Figure 1 correctly implements a snapshot object if no two Scan operations are concurrent and no two Write operations to the same component are concurrent. Because of space constraints, we state the lemmas without proofs.

DEFINITION 1. If S is any `Scan()` operation and W is any `Write(i, *)` operation, we say that S returns W for component i if either of the following two conditions is met: (1) S reads \perp from $B[i]$ (at Line 9) and W is the last operation to write $A[i]$ before S reads $A[i]$ at Line 6, or (2) S reads a non- \perp value from $B[i]$ (at Line 9) and W is the last operation to write $B[i]$ before S reads $B[i]$.

Notation: If S is any `Scan()` operation and W is any `Write(i, *)` operation, $S[k]$ (respectively, $W[k]$) denotes the point in time at which S (respectively, W) executes Line k of its algorithm. Since Lines 5 and 6 are not atomic actions, we refine the notation further for these two lines: $S[5, i]$ denotes the time at which S performs the i th iteration of the for-loop on Line 5. $S[6, i]$ is similarly defined. $S[9, i]$ denotes the time at which S performs Line 9 for the i th iteration of the for-loop that begins at Line 8.

The first lemma states that if a `Write(i, v)` operation W completes before a `Scan` operation S performs its Line 7, then W takes effect before S .

LEMMA 1. Let S be any `Scan` operation and W be any `Write` operation to component i such that W completes before $S[7]$. Then, S does not return an older `Write` than W for component i : i.e., if S returns W' for component i , then W' does not precede W .

The next lemma states that if a `Write(i, v)` operation W starts before a `Scan` operation S starts, then W takes effect before S .

LEMMA 2. Let S be any `Scan` operation and W be any `Write` operation to component i such that $W[1] < S[4]$. Then, S does not return an older `Write` than W for component i : i.e., if S returns W' for component i , then W' does not precede W .

The next lemma states that if a `Write(i, v)` operation W starts after a `Scan` operation S performs its Line 7, then W takes effect after S (in other words, W is too late to be noticed by S).

LEMMA 3. Let S be any `Scan` operation and W be any `Write` operation to component i such that $S[7] < W[1]$. Then, S returns an older `Write` than W for component i : i.e., if S returns W' for component i , then W precedes W' .

Next we define the linearization points for `Scan` and `Write` operations. Lemmas 1 and 3 suggest $S[7]$ as the natural choice for where a `Scan` operation S should be linearized (because the two lemmas state that a `Write` operation W is taken into account by S if W completes before $S[7]$ but not if it starts after $S[7]$). A `Write` operation W , on the other hand, is linearized at $W[1]$ except in one case: W overlaps with a `Scan` S in such a way that S does not notice W . In this case, we linearize W immediately after S . The precise definitions are as follows. Henceforth, $LP(OP)$ denotes the linearization point of an operation OP .

DEFINITION 2 (Linearization Points). For any `Scan()` operation S , we define $LP(S)$ as $S[7]$.

For any `Write(i, *)` operation W , $LP(W)$ is defined in two cases:

Case 1: There exists a `Scan()` operation S such that $S[4] < W[1] < S[7]$ and, for component i , S returns an older `Write` than W .

In this case, Lemma 1 guarantees that W completes only after $S[7]$. We define $LP(W)$ to be immediately after $LP(S) = S[7]$. More precisely, $LP(W)$ is defined as any point that lies after $S[7]$ and within the intervals of both W and S .

Case 2: Case 1 does not hold.

In this case, we define $LP(W)$ as $W[1]$. ■

To prove the algorithm correct, we need to argue two points: (1) LP containment: the linearization point of each operation is within the interval of that operation, and (2) Correctness of `Scan`: if a `Scan` operation S returns W for component i , then W is the latest `Write` operation to component i to be linearized before S . These two points are proved in the next two lemmas.

LEMMA 4 (LP CONTAINMENT). For any operation OP , which may be a `Scan` or a `Write`, $LP(OP)$ lies within the interval of OP .

LEMMA 5 (CORRECTNESS OF SCAN). If a `Scan` operation S returns W for component i , then (1) $LP(W) < LP(S)$, and (2) if W' is also a `Write` to component i and is executed after W , then $LP(W') > LP(S)$.

Theorem 1 of Section 2, is immediate from Lemmas 4, 5.

- [12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463-492, 1990.
- [13] P. Jayanti. *f*-arrays: implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270 - 279, 2002.
- [14] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.
- [15] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword ll/sc variables. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*, June 2005.
- [16] L. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, pages 229-241, 1991.
- [17] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219-228, August 1997.
- [18] Y. Riaz, N. Shavit, and D. Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, 269(1-2):163-201, 2001.

APPENDIX

A. PROOF OF CORRECTNESS OF THE BASIC ALGORITHM

In this appendix, we sketch the proof that the basic algorithm in Figure 1 correctly implements a snapshot object if no two Scan operations are concurrent and no two Write operations to the same component are concurrent. Because of space constraints, we state the lemmas without proofs.

DEFINITION 1. *If S is any Scan() operation and W is any Write($i, *$) operation, We say that S returns W for component i if either of the following two conditions is met: (1) S reads \perp from $B[i]$ (at Line 9) and W is the last operation to write $A[i]$ before S reads $A[i]$ at Line 6, or (2) S reads a non- \perp value from $B[i]$ (at Line 9) and W is the last operation to write $B[i]$ before S reads $B[i]$.*

Notation: If S is any Scan() operation and W is any Write($i, *$) operation, $S[k]$ (respectively, $W[k]$) denotes the point in time at which S (respectively, W) executes Line k of its algorithm. Since Lines 5 and 6 are not atomic actions, we refine the notation further for these two lines: $S[5, i]$ denotes the time at which S performs the i th iteration of the for-loop on Line 5. $S[6, i]$ is similarly defined. $S[9, i]$ denotes the time at which S performs Line 9 for the i th iteration of the for-loop that begins at Line 8.

The first lemma states that if a Write(i, v) operation W completes before a Scan operation S performs its Line 7, then W takes effect before S .

LEMMA 1. *Let S be any Scan operation and W be any Write operation to component i such that W completes before $S[7]$. Then, S does not return an older Write than W for component i : i.e., if S returns W' for component i , then W' does not precede W .*

The next lemma states that if a Write(i, v) operation W starts before a Scan operation S starts, then W takes effect before S .

LEMMA 2. *Let S be any Scan operation and W be any Write operation to component i such that $W[1] < S[4]$.*

Then, S does not return i : i.e., if S returns W' for component i , then W' does not precede W .

The next lemma states that if a Write(i, v) operation W starts after a Scan operation S starts, then W takes effect after S .

LEMMA 3. *Let S be any Scan operation and W be any Write operation to component i such that $W[1] > S[7]$.*

Then, S returns W for component i : i.e., if S returns W' for component i , then W' does not precede W .

DEFINITION 2 (Linearization Points). *For any Scan() operation S , we define $LP(S)$ as $S[7]$.*

*For any Write($i, *$) operation W , $LP(W)$ is defined in two cases:*

Case 1: *There exists a Scan() operation S such that $S[4] < W[1] < S[7]$ and, for component i , S returns an older Write than W .*

Case 2: *Case 1 does not hold. In this case, we define $LP(W)$ as $W[1]$.*

LEMMA 4 (LP CONTAINMENT). *For any operation OP, which may be a Scan or a Write, $LP(OP)$ lies within the interval of OP.*

LEMMA 5 (CORRECTNESS OF SCAN). *If a Scan operation S returns W for component i , then (1) $LP(W) < LP(S)$, and (2) if W' is also a Write to component i and is executed after W , then $LP(W') > LP(S)$.*

Theorem 1 of Section 2, is immediate from Lemmas 4, 5.

DEFINITION 2 (Linearization Points). *For any Scan() operation S , we define $LP(S)$ as $S[7]$.*

*For any Write($i, *$) operation W , $LP(W)$ is defined in two cases:*

Case 1: *There exists a Scan() operation S such that $S[4] < W[1] < S[7]$ and, for component i , S returns an older Write than W .*

In this case, Lemma 1 guarantees that W completes only after $S[7]$. We define $LP(W)$ to be immediately after $LP(S) = S[7]$. More precisely, $LP(W)$ is defined as any point that lies after $S[7]$ and within the intervals of both W and S .

Case 2: *Case 1 does not hold.*

In this case, we define $LP(W)$ as $W[1]$. ■



```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0..n$  do B[i] :=  $\perp$ ;  
6.   for  $i = 0..n$  do V[i] := A[i];  
7.   S := false;  
8.   for  $i = 0..n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then V[i] :=  $v$ ;  
11.  return V  
}
```

```
write ( $i, v$ ) {  
1.   A[i] :=  $v$ ;  
2.   if S  
3.   then B[i] :=  $v$   
}
```



```

    scan : array nat {
4.     S := true;
5.     for  $i = 0..n$  do B[i] :=  $\perp$ ;
6.     for  $i = 0..n$  do V[i] := A[i];
7.     S := false;
8.     for  $i = 0..n$  do
9.          $v = B[i]$ ;
10.        if ( $v \neq \perp$ ) then V[i] :=  $v$ ;
11.    return V
    }

```

Scan's LP is fixed!

```

    write ( $i, v$ ) {
1.     A[i] :=  $v$ ;
2.     if S
3.     then B[i] :=  $v$ 
    }

```

```
scan : array nat {
4.   S := true;
5.   for i = 0..n do B[i] := ⊥;
6.   for i = 0..n do V[i] := A[i];
7.   S := false;
8.   for i = 0..n do
9.     v = B[i];
10.    if (v ≠ ⊥) then V[i] := v;
11.  return V
}
```

```
write (i, v) {
1.   A[i] := v;
2.   if S
3.     then B[i] := v
}
```

Write's LP is here...
sometimes

```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0..n$  do B[i] :=  $\perp$ ;  
6.   for  $i = 0..n$  do V[i] := A[i];  
7.   S := false;  
8.   for  $i = 0..n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then V[i] :=  $v$ ;  
11.  return V  
}
```

```
write ( $i, v$ ) {  
1.   A[i] :=  $v$ ;  
2.   if S  
3.   then B[i] :=  $v$   
}
```

Or here!

```
scan : array nat {  
4.   S := true;  
5.   for  $i = 0..n$  do B[i] :=  $\perp$ ;  
6.   for  $i = 0..n$  do V[i] := A[i];  
7.   S := false;  
8.   for  $i = 0..n$  do  
9.     v = B[i];  
10.    if ( $v \neq \perp$ ) then V[i] := v;  
11.  return V  
}
```

```
write ( $i, v$ ) {  
1.   A[i] := v;  
2.   if S  
3.   then B[i] := v  
}
```

Even anywhere here!

LINEARIZATION POINTS FOR WRITE

LINEARIZATION POINTS FOR WRITE

- **Non-fixed:** LP determined by dynamic, run-time information.

LINEARIZATION POINTS FOR WRITE

- **Non-fixed:** LP determined by dynamic, run-time information.
- **Non-local:** LP might be in write's code. Or not.

LINEARIZATION POINTS FOR WRITE

- **Non-fixed:** LP determined by dynamic, run-time information.
- **Non-local:** LP might be in `write's` code. Or not.
- **Future-dependent:** the position of the LP depends on future events.

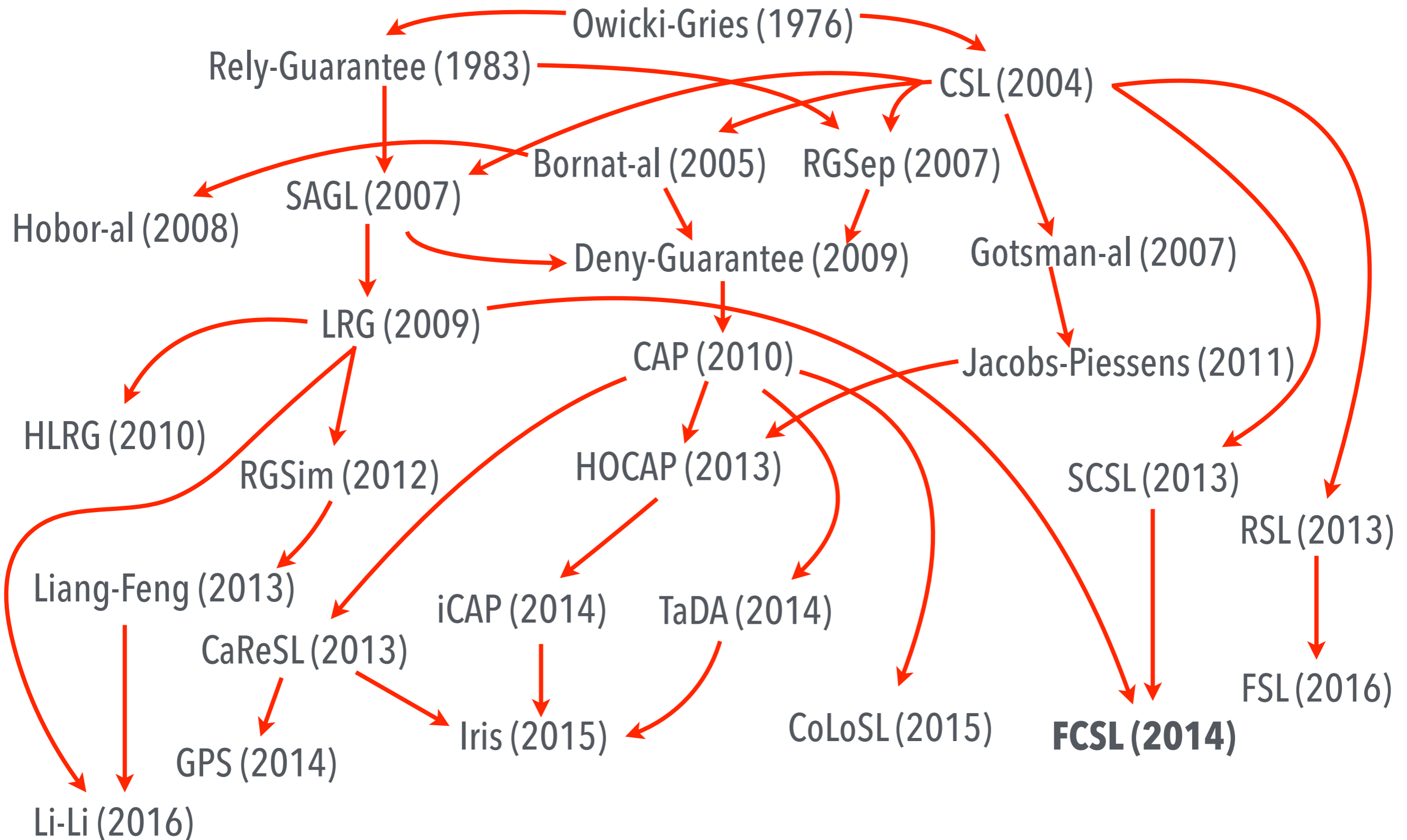
LINEARIZATION POINTS FOR WRITE

- **Non-fixed:** LP determined by dynamic, run-time information.
- **Non-local:** LP might be in `write`'s code. Or not.
- **Future-dependent:** the position of the LP depends on future events.
- **Non-regional (Far-Future):** which (*potentially*) take place after `write` has finished.



Linearization Points = Pointers in Time

LOGICS FOR CONCURRENCY



FCSL: FINE-GRAINED CONCURRENT SEPARATION LOGIC

- **Subjective Auxiliary State:** PCMs identify thread-specific contributions
- **State-Transition Systems:** user-defined concurrent protocols.
- **Types:** mechanization and compositionality.

Communicating State Transition Systems for Fine-Grained Concurrent Resources

Aleksandar Nanevski¹, Ruy Ley-Wild², Ilya Sergey¹, and Germán Andrés Delbianco¹

¹ IMDEA Software Institute, Spain
{aleks.nanevski, ilya.sergey, german.delbianco}@imdea.org

² LogicBlox, USA
ruy.leywild@logicblox.com

Abstract. We present a novel model of concurrent computations with shared memory and provide a simple, yet powerful, logical framework for uniform Hoare-style reasoning about partial correctness of coarse- and fine-grained concurrent programs. The key idea is to specify arbitrary resource protocols as communicating *state transition systems* (STS) that describe valid states of a resource and the transitions the resource is allowed to make, including transfer of heap ownership.

We demonstrate how reasoning in terms of communicating STS makes it easy to crystallize behavioral invariants of a resource. We also provide *entanglement* operators to build large systems from an arbitrary number of STS components, by interconnecting their lines of communication. Furthermore, we show how the classical rules from the Concurrent Separation Logic (CSL), such as scoped resource allocation, can be generalized to fine-grained resource management. This allows us to give specifications as powerful as Rely-Guarantee, in a concise, scoped way, and yet regain the compositionality of CSL-style resource management. We proved the soundness of our logic with respect to the denotational semantics of action trees (variation on Brookes' action traces). We formalized the logic as a shallow embedding in Coq and implemented a number of examples, including a construction of coarse-grained CSL resources as a modular composition of various logical and semantic components.

1 Introduction

There are two main styles of program logics for shared-memory concurrency, customarily divided according to the supported kind of granularity of program interference. Logics for coarse-grained concurrency such as Concurrent Separation Logic (CSL) [12, 14] restrict the interference to critical sections only, but generally lead to more modular specifications and simpler proofs of program correctness. Logics for fine-grained concurrency, such as Rely-Guarantee (RG) [8] admit arbitrary interference, but their specifications have traditionally been more monolithic, as we shall illustrate. In this paper, we identify the essential ingredients required for compositional specification of concurrent programs, and combine them in a novel way to reconcile the two approaches. We present a semantic model and a logic that enables specification and reasoning about fine-grained programs, but in the style of CSL. To describe our contribution more precisely, we first compare the relevant properties of CSL and RG.

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 290–310, 2014.
© Springer Verlag Berlin Heidelberg 2014

(Nanevski, Ley-Wild, Sergey & **Delbianco**:ESOP14)

1

2

5

3

7

1

2

Atomic write events

5

3

7

Colors encode
runtime "visibility" of
events

1

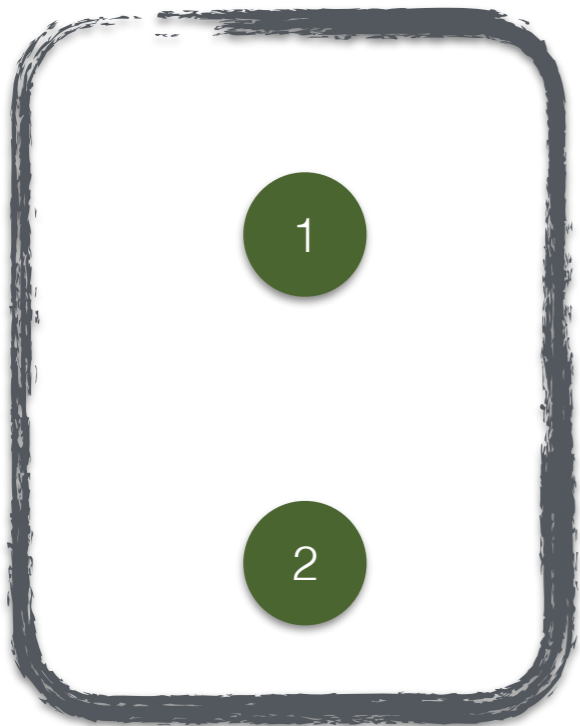
2

5

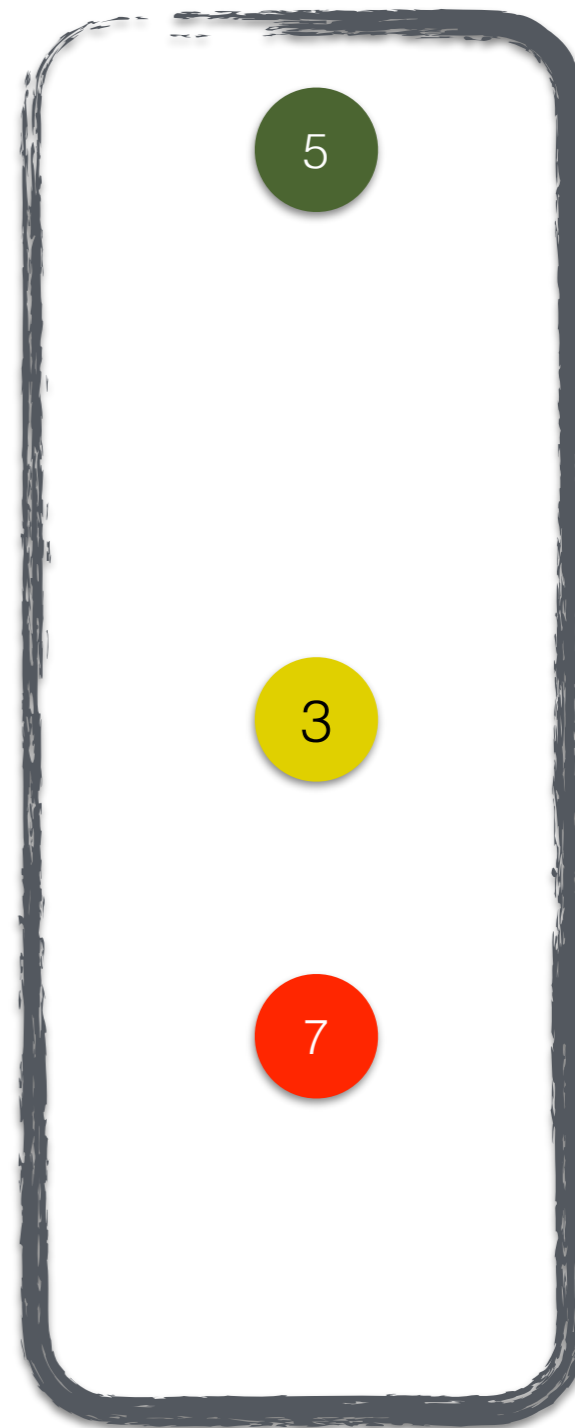
3

7

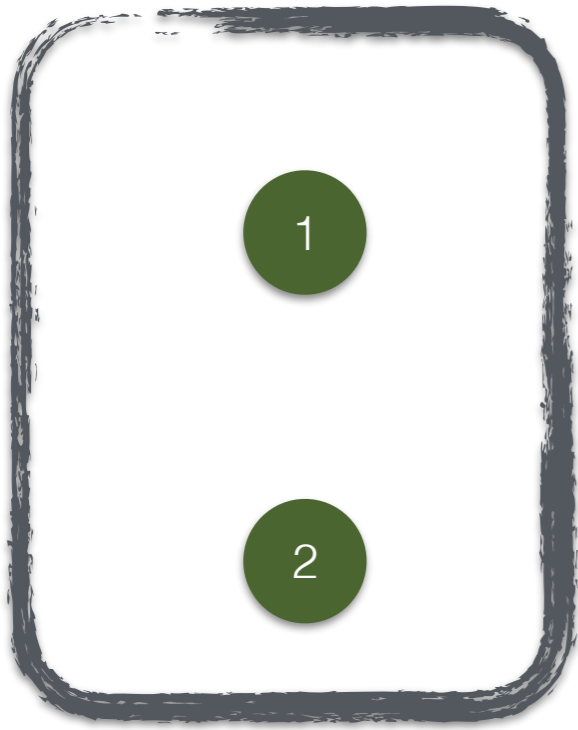
Xs



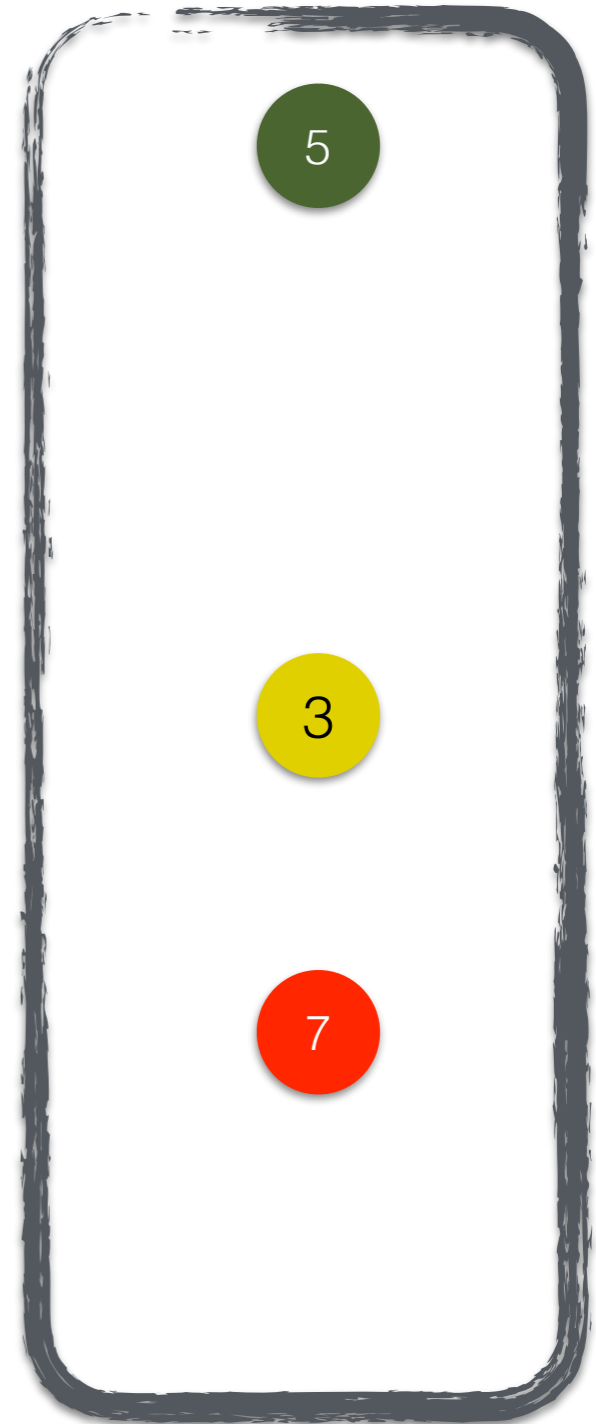
Xo



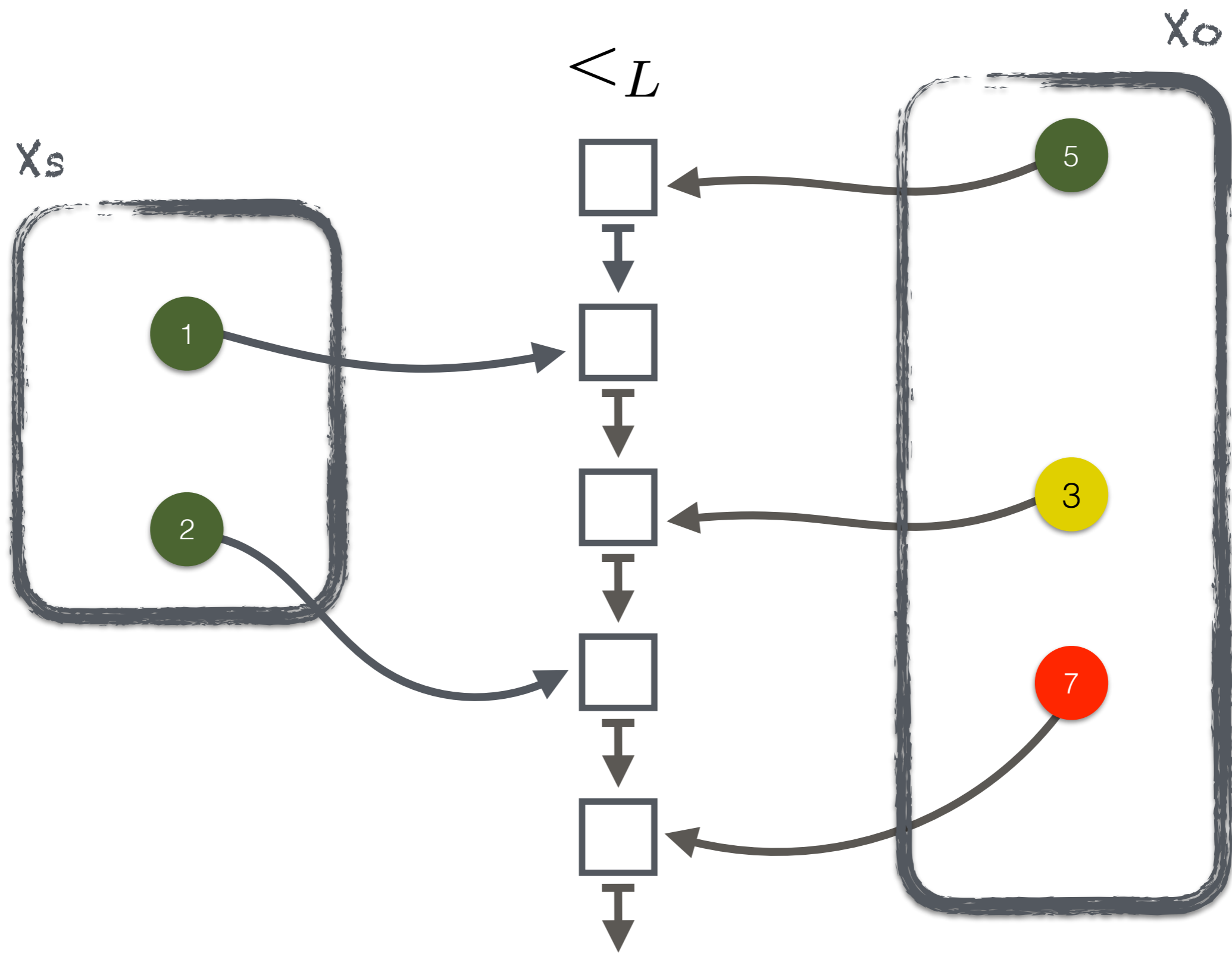
X_s



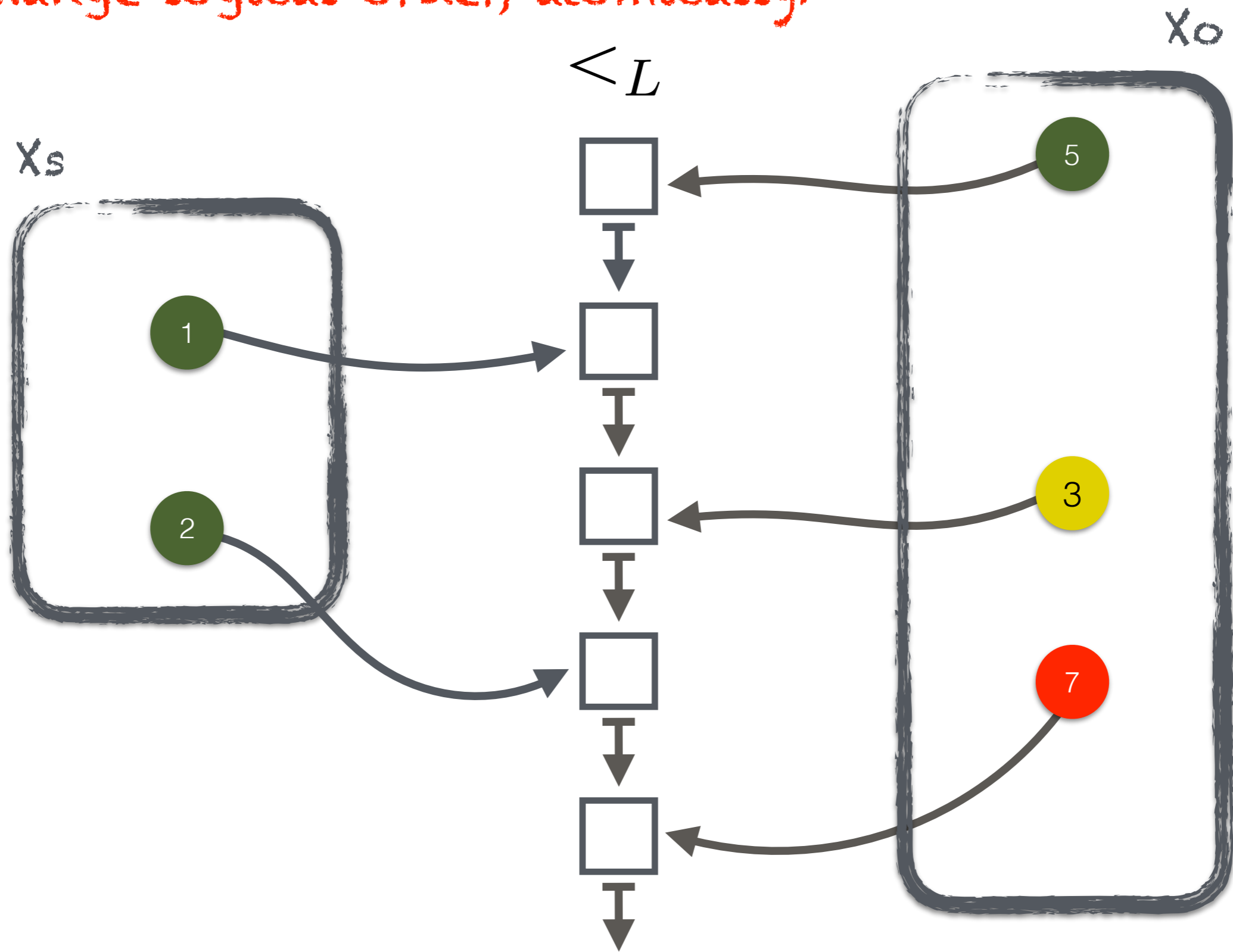
X_o



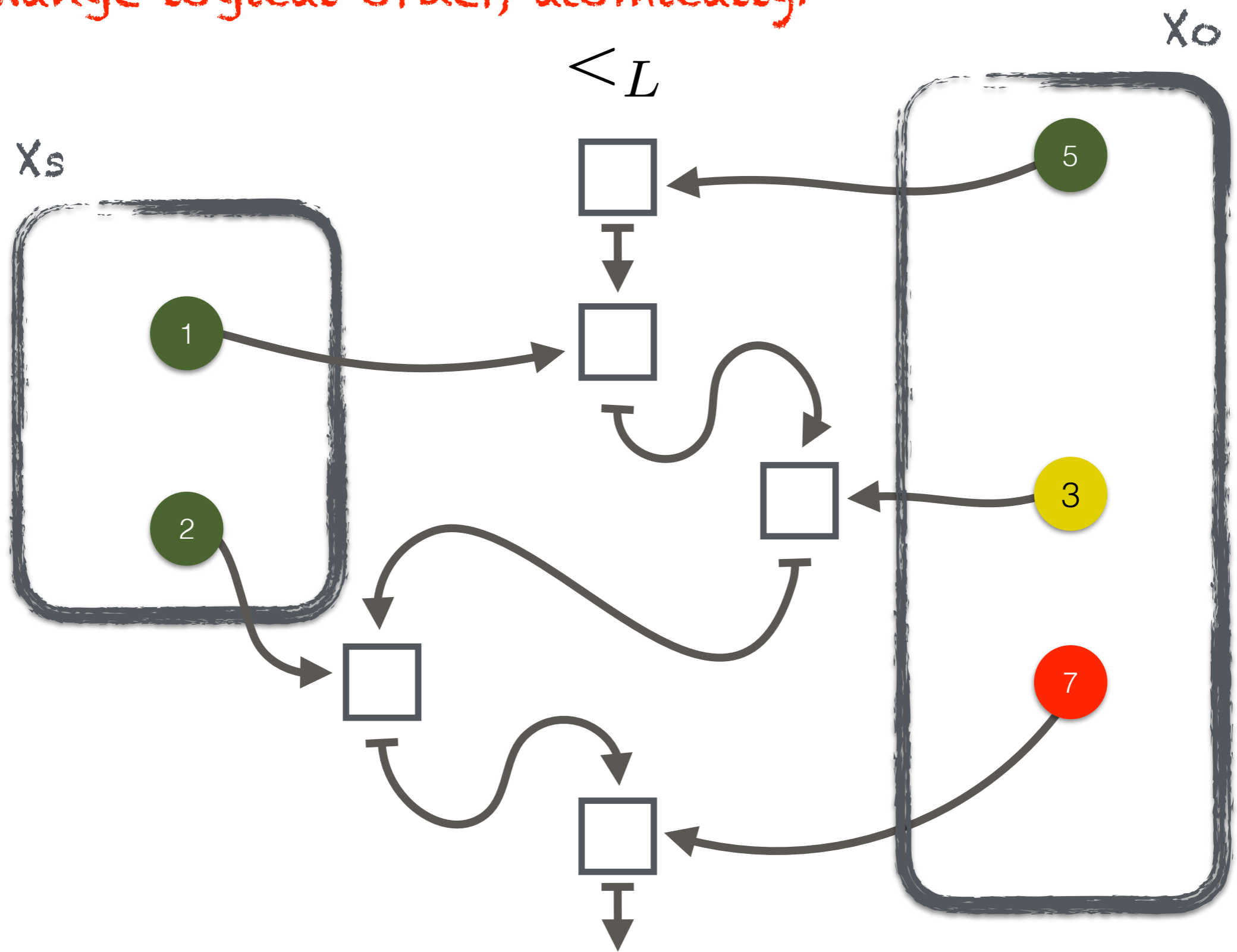
$X = X_s + X_o$
Subjective History!



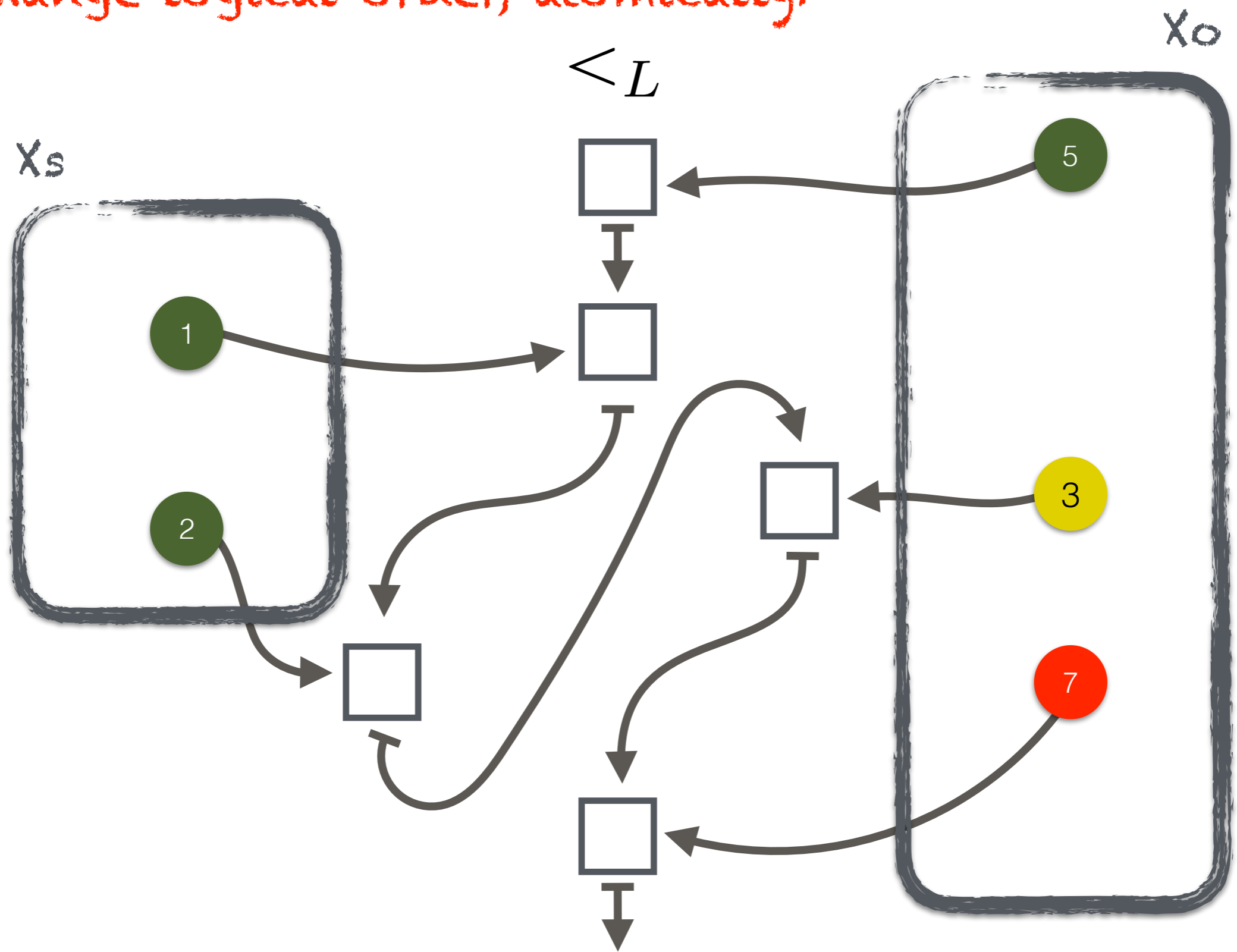
Change logical order, atomically!



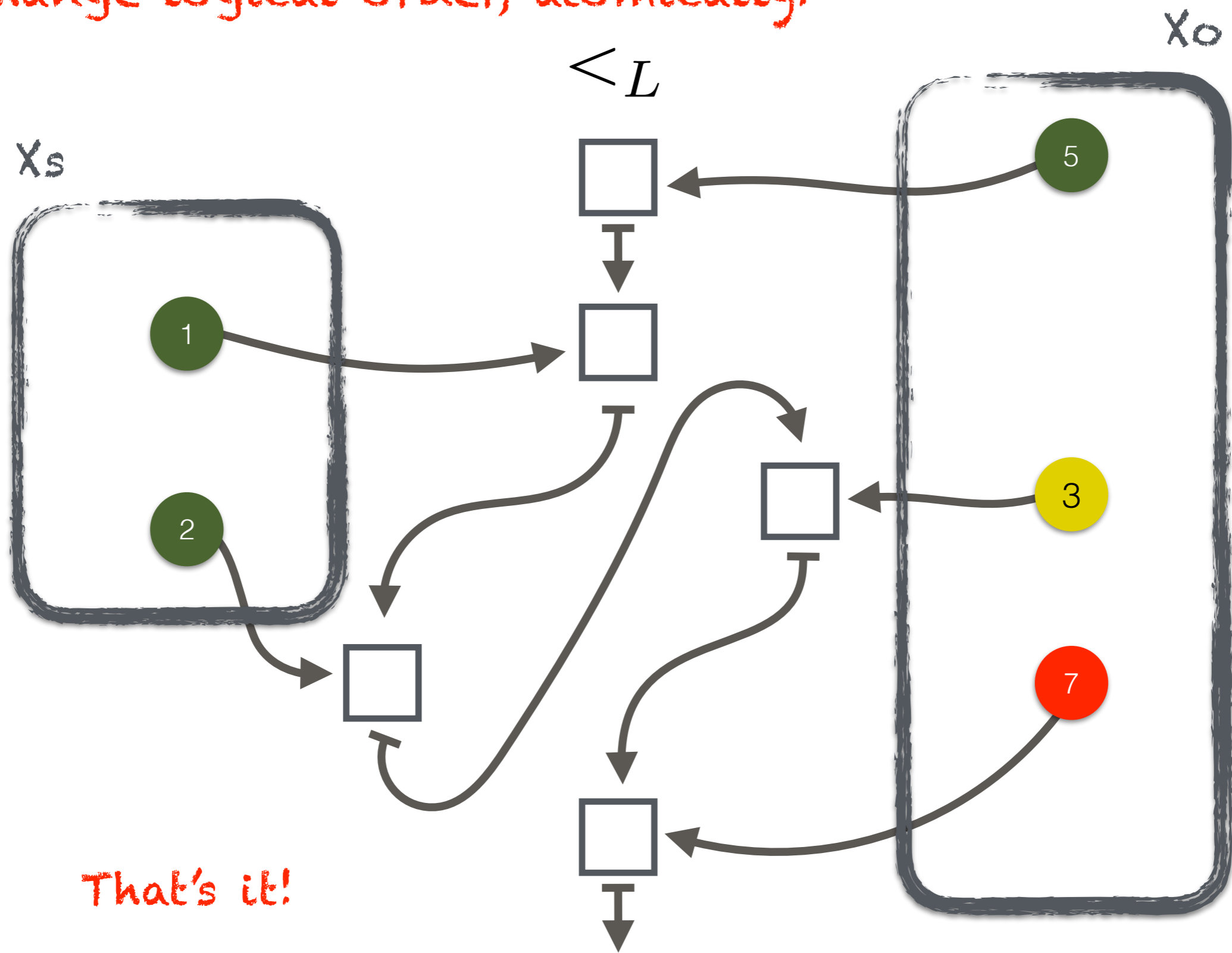
Change logical order, atomically!



Change logical order, atomically!



Change logical order, atomically!



That's it!

AUXILIARY STATE INVARIANTS

AUXILIARY STATE INVARIANTS



Color invariant: At all times, many greens in the head,
at most one yellow, any reds in the tail.

$G+.Y?.R^*$

AUXILIARY STATE INVARIANTS



Color invariant: At all times, many greens in the head, at most one yellow, any reds in the tail.

*G+.Y?.R**

$E(t)$ **End times:** preserve ordering of non-overlapping events.

$$E(t_1) < t_2 \implies t_1 <_L t_2$$

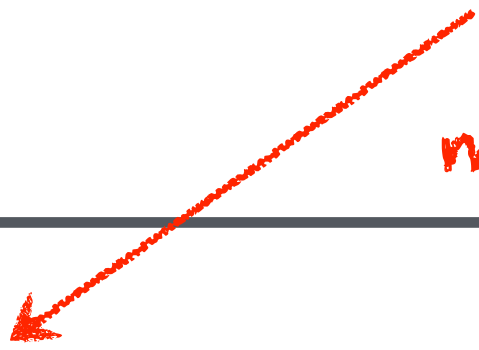
as in Linearizability!

INSTRUMENTED SCAN

```
scan : array nat {  
4.   ⟨S := true; scanPrelude()⟩;  
5.   for  $i = 0 .. n$  do ⟨B[i] :=  $\perp$ ; clear( $i$ )⟩  
6.   for  $i = 0 .. n$  do V[i] := A[i];  
7.   ⟨S := false; scanEpilogue()⟩;  
8.   for  $i = 0 .. n$  do  
9.        $v = B[i]$ ;  
10.      if ( $v \neq \perp$ ) then V[i] := v;  
11.  ⟨relink(); return V⟩  
}
```

INSTRUMENTED SCAN

Erasable
auxiliary
code
methods



```
scan : array nat {  
4.   ⟨S := true; scanPrelude()⟩;  
5.   for i = 0 .. n do ⟨B[i] := ⊥; clear(i)⟩  
6.   for i = 0 .. n do V[i] := A[i];  
7.   ⟨S := false; scanEpilogue()⟩;  
8.   for i = 0 .. n do  
9.     v = B[i];  
10.    if (v ≠ ⊥) then V[i] := v;  
11.  ⟨relink(); return V⟩  
}
```

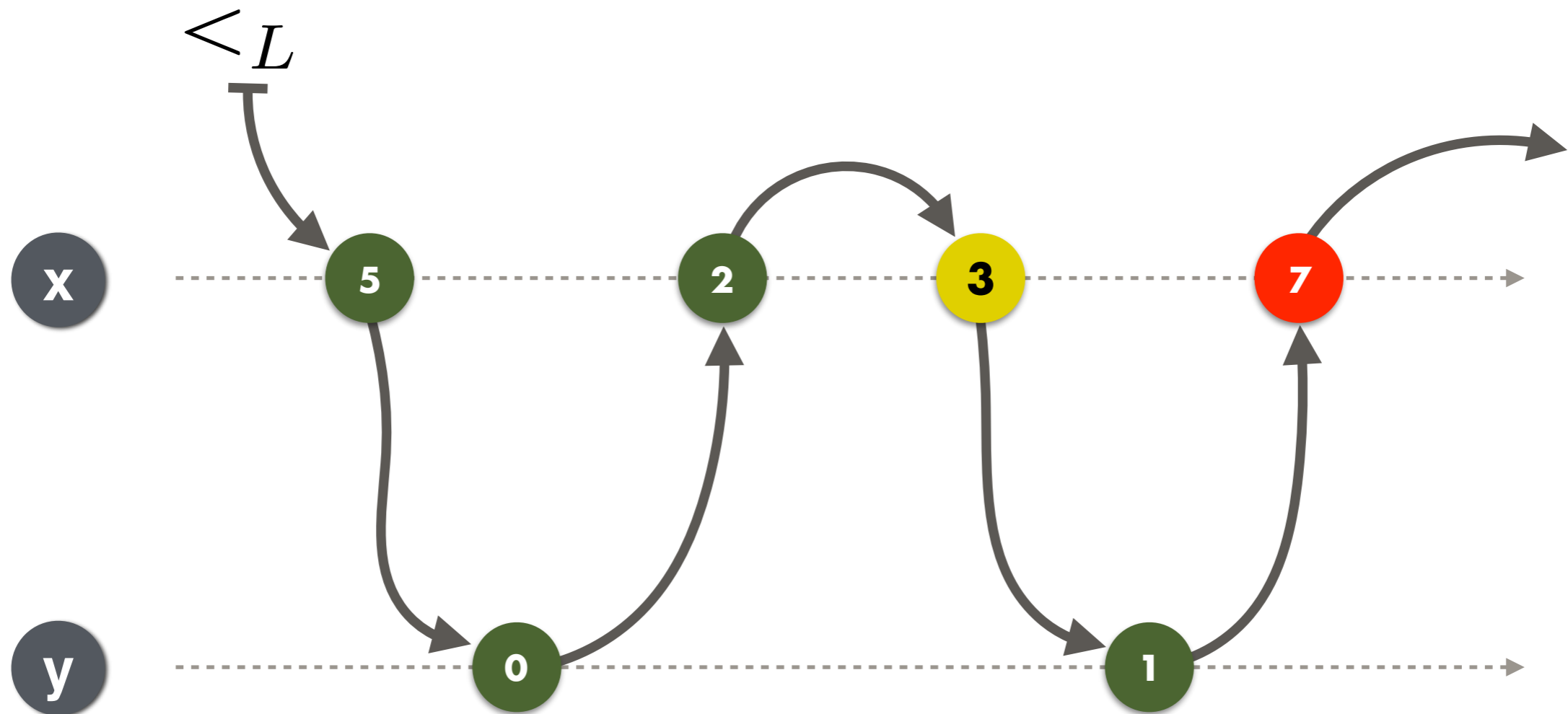
INSTRUMENTED SCAN

Erasable
auxiliary
code
methods

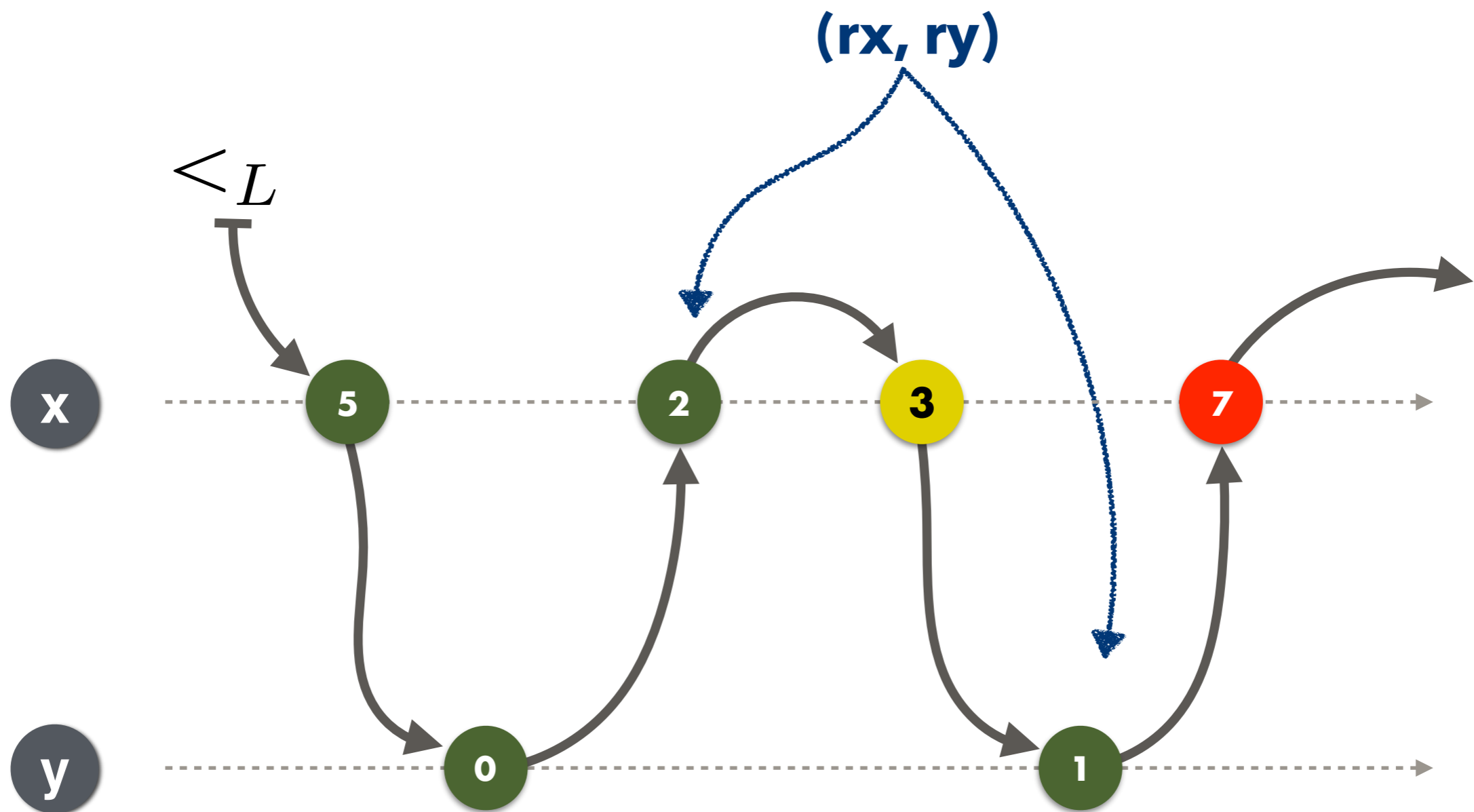
```
scan : array nat {  
4.   ⟨S := true; scanPrelude()⟩;  
5.   for  $i = 0 .. n$  do ⟨B[i] :=  $\perp$ ; clear( $i$ )⟩  
6.   for  $i = 0 .. n$  do V[i] := A[i];  
7.   ⟨S := false; scanEpilogue()⟩;  
8.   for  $i = 0 .. n$  do  
9.     v = B[i];  
10.    if ( $v \neq \perp$ ) then V[i] := v;  
11.  ⟨relink(); return V⟩  
}
```

Fix Order before return!

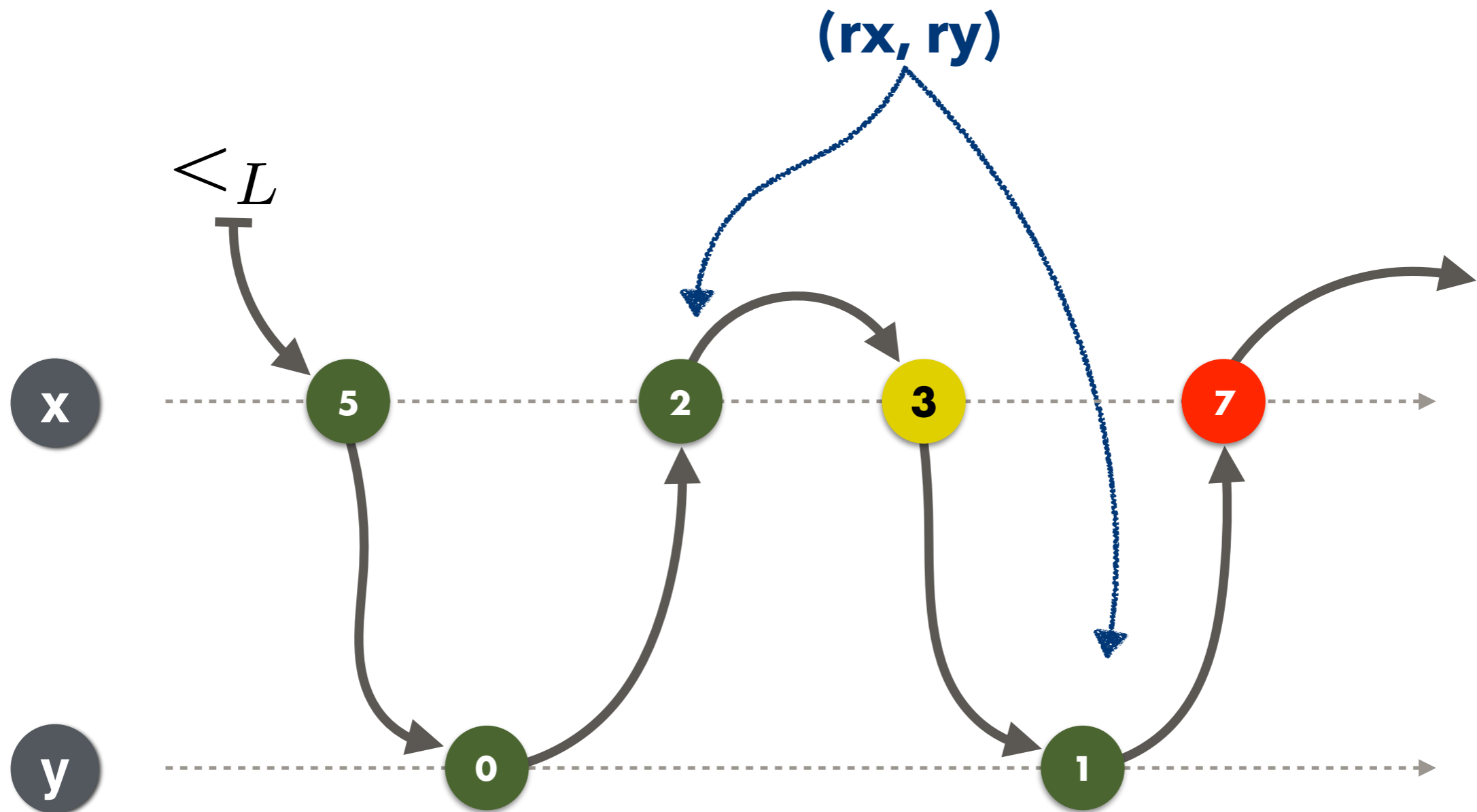
BEFORE RELINK



BEFORE RELINK

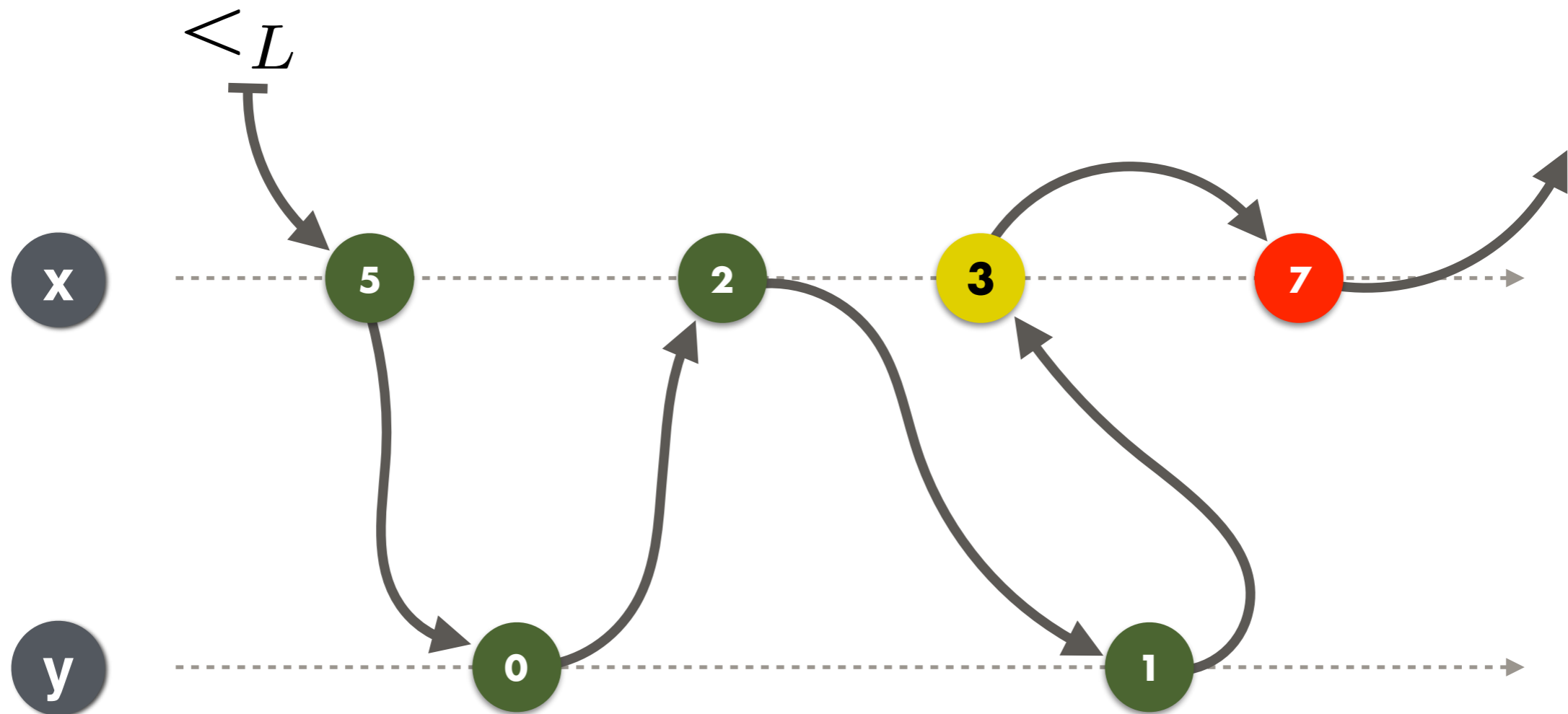


BEFORE RELINK

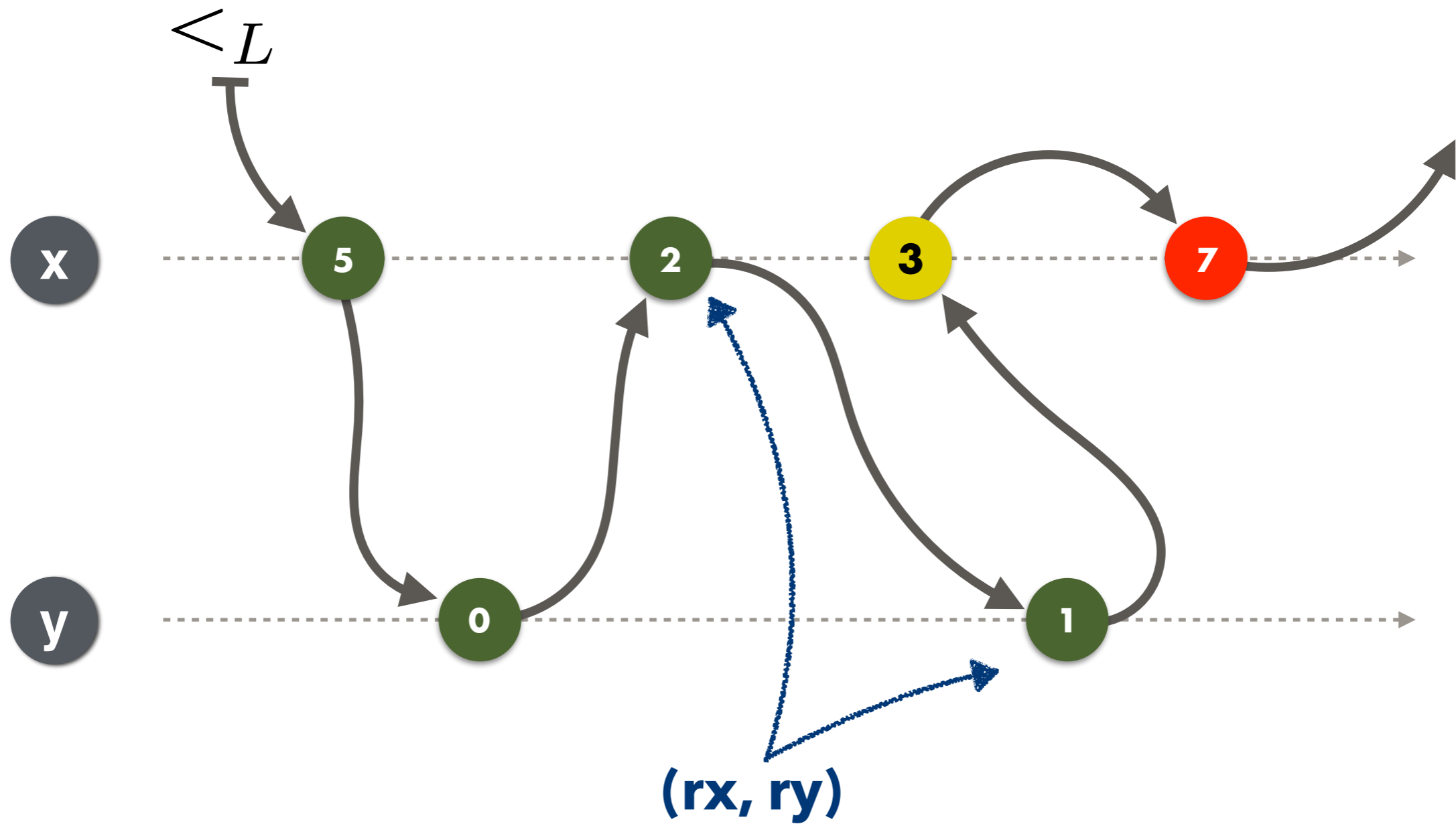


NOT A VALID
SNAPSHOT!

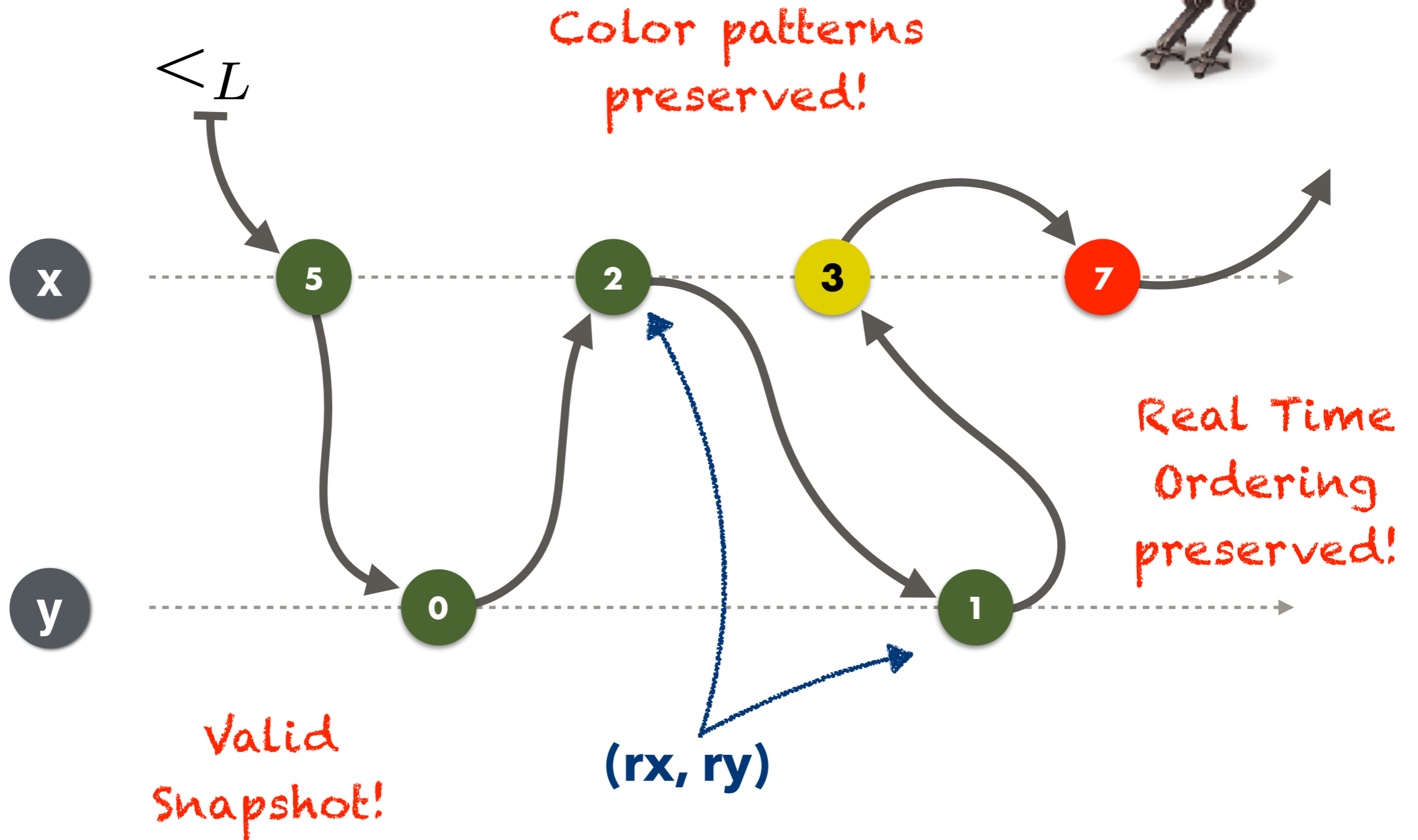
AFTER RELINK



AFTER RELINK



AFTER RELINK



SPEC FOR WRITE

`write (p : ptr, n : int)`

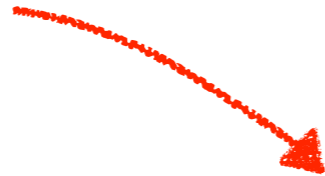
SPEC FOR WRITE

$$\{\chi_s = \emptyset\}$$

`write (p : ptr, n : int)`

SPEC FOR WRITE

Self-history χ_s is
empty



$$\{\chi_s = \emptyset\}$$

`write (p : ptr, n : int)`

SPEC FOR WRITE

$$\{\chi_s = \emptyset\}$$

`write` ($p : \text{ptr}, n : \text{int}$)

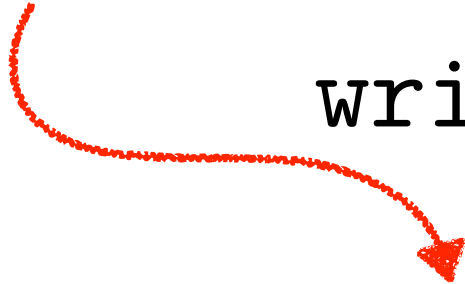
$$\{\exists \mathbf{t}. \chi'_s = \mathbf{t} \mapsto (\mathbf{p}, \mathbf{v}) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow \mathbf{t})\}$$

SPEC FOR WRITE

Self-History χ_s'
accounts for a
fresh write event

$$\{\chi_s = \emptyset\}$$

write ($p : \text{ptr}, n : \text{int}$)


$$\{\exists t. \chi_s' = t \mapsto (p, v) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

SPEC FOR WRITE

Self-History χ_s'
accounts for a
fresh write event

$$\{\chi_s = \emptyset\}$$

write ($p : \text{ptr}, n : \text{int}$)

$$\{\exists t. \chi_s' = t \mapsto (p, v) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

All other (environment)
write events

SPEC FOR WRITE

Self-History χ_s'
accounts for a
fresh write event

$$\{\chi_s = \emptyset\}$$

write ($p : \text{ptr}, n : \text{int}$)

$$\{\exists t. \chi_s' = t \mapsto (p, v) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

All other (environment)
write events

Global stable order...

SNAPSHOT OBJECT SPECIFICATION

STABLE DYNAMIC ORDER

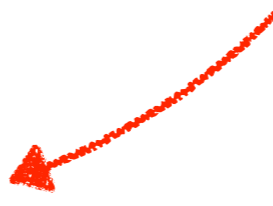
$$a \Omega b \hat{=} a = b$$
$$\vee E(a) < b$$
$$\vee a <_L b \wedge C(a) = \text{green}$$

SNAPSHOT OBJECT SPECIFICATION

STABLE DYNAMIC ORDER

$$a \Omega b \hat{=} \begin{aligned} &a = b \\ &\vee E(a) < b \\ &\vee a <_L b \wedge C(a) = \text{green} \end{aligned}$$

Non-overlapping write events



SNAPSHOT OBJECT SPECIFICATION

STABLE DYNAMIC ORDER

$$a \Omega b \hat{=} a = b$$

$$\vee E(a) < b$$

$$\vee a <_L b \wedge C(a) = \text{green}$$

Non-overlapping
write events

Green timestamps
are fixed on the left
of $<_L$

SNAPSHOT OBJECT SPECIFICATION

STABLE DYNAMIC ORDER

$$a \Omega b \hat{=} a = b$$

$$\vee E(a) < b$$

$$\vee a <_L b \wedge C(a) = \text{green}$$

Non-overlapping
write events

Green timestamps
are fixed on the left
of $<_L$

$$\text{stable: } \Omega \subseteq \Omega'$$

SNAPSHOT OBJECT SPECIFICATION

SCANNED PREFIX

$$\text{scanned } \Omega \hat{=} \{t \mid (\Omega \downarrow t) = (\prec_L \downarrow t) \\ \wedge \forall s \in (\Omega \downarrow t). C(s) = \text{green}\}$$

SNAPSHOT OBJECT SPECIFICATION

SCANNED PREFIX

Prefix of $\langle L$ up to t



$$\text{scanned } \Omega \hat{=} \{t \mid (\Omega \downarrow t) = (\langle L \downarrow t) \\ \wedge \forall s \in (\Omega \downarrow t). C(s) = \text{green}\}$$

SNAPSHOT OBJECT SPECIFICATION

SCANNED PREFIX

Prefix of $\langle L \downarrow t$ up to t



$$\text{scanned } \Omega \hat{=} \{t \mid (\Omega \downarrow t) = (\langle L \downarrow t) \wedge \forall s \in (\Omega \downarrow t). C(s) = \text{green}\}$$

All events are green (= fixed)



SNAPSHOT OBJECT SPECIFICATION

SCANNED PREFIX

Prefix of $\langle L \downarrow t$ up to t

$$\text{scanned } \Omega \hat{=} \{t \mid (\Omega \downarrow t) = (\langle L \downarrow t) \wedge \forall s \in (\Omega \downarrow t). C(s) = \text{green}\}$$

All events are green (= fixed)

A growing
Linearization of the
data structure!

$$\text{scanned } \Omega \subseteq \text{scanned } \Omega'$$

SPEC FOR WRITE

$$\{\chi_s = \emptyset\}$$

`write` ($p : \text{ptr}, n : \text{int}$)

$$\{\exists \mathbf{t}. \chi'_s = \mathbf{t} \mapsto (\mathbf{p}, \mathbf{v})$$
$$\wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow \mathbf{t})\}$$

SPEC FOR WRITE

$$\{\chi_s = \emptyset\}$$

write ($p : \text{ptr}, n : \text{int}$)

$$\{\exists t. \chi'_s = t \mapsto (\mathbf{p}, \mathbf{v}) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

All previous,
Finished
Writes...



SPEC FOR WRITE

$$\{\chi_s = \emptyset\}$$

... and all previously scanned events...

write ($p : \text{ptr}, n : \text{int}$)

$$\{\exists t. \chi'_s = t \mapsto (p, v) \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

All previous,
Finished
Writes...

SPEC FOR WRITE

$$\{\chi_s = \emptyset\}$$

... and all previously scanned events...

write ($p : \text{ptr}, n : \text{int}$)

$$\{\exists t. \chi'_s = t \mapsto (p, v) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

All previous,
Finished
Writes...

...are sorted before the
new write event

SPEC FOR SCAN

`scan()` : array A

SPEC FOR SCAN

$$\{\chi_s = \emptyset\}$$

`scan()` : array A

SPEC FOR SCAN

$$\{\chi_s = \emptyset\}$$

scan() : array A

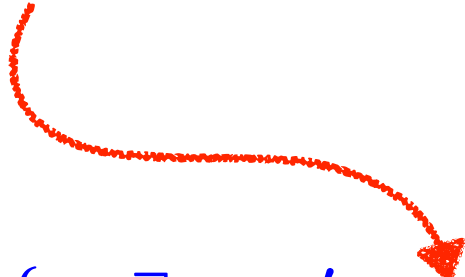
$$\{\mathbf{r}. \exists \mathbf{t}. \chi'_s = \emptyset \wedge \mathbf{r} = \text{eval } \mathbf{t} \Omega' \chi' \\ \wedge \text{dom}(\chi) \subseteq (\Omega' \downarrow \mathbf{t}) \wedge \mathbf{t} \in \text{scanned } \Omega'\}$$

SPEC FOR SCAN

scan observes
other threads'
contributions

$$\{\chi_s = \emptyset\}$$

scan() : array A


$$\{\mathbf{r}. \exists \mathbf{t}. \chi'_s = \emptyset \wedge \mathbf{r} = \text{eval } \mathbf{t} \Omega' \chi' \\ \wedge \text{dom}(\chi) \subseteq (\Omega' \downarrow \mathbf{t}) \wedge \mathbf{t} \in \text{scanned } \Omega'\}$$

SPEC FOR SCAN

scan observes
other threads'
contributions

$$\{\chi_s = \emptyset\}$$

t uniquely
determines a
snapshot in the
history, ...

scan() : array A

$$\{r. \exists t. \chi'_s = \emptyset \wedge r = \text{eval } t \Omega' \chi' \\ \wedge \text{dom}(\chi) \subseteq (\Omega' \downarrow t) \wedge t \in \text{scanned } \Omega'\}$$

SPEC FOR SCAN

scan observes
other threads'
contributions

$$\{\chi_s = \emptyset\}$$

t uniquely
determines a
snapshot in the
history, ...

scan() : array A

$$\{r. \exists t. \chi'_s = \emptyset \wedge r = \text{eval } t \Omega' \chi' \\ \wedge \text{dom}(\chi) \subseteq (\Omega' \downarrow t) \wedge t \in \text{scanned } \Omega'\}$$

... which considered
at least all previous
writes...

SPEC FOR SCAN

scan observes
other threads'
contributions

$$\{\chi_s = \emptyset\}$$

t uniquely
determines a
snapshot in the
history, ...

scan() : array A

$$\{r. \exists t. \chi'_s = \emptyset \wedge r = \text{eval } t \Omega' \chi' \\ \wedge \text{dom}(\chi) \subseteq (\Omega' \downarrow t) \wedge t \in \text{scanned } \Omega'\}$$

... which considered
at least all previous
writes...

...and, it cannot be
invalidated in the
future

SNAPSHOT OBJECT SPECS

$$\{\chi_s = \emptyset\}$$

`write (p : ptr, n : int)`

$$\{\exists t. \chi'_s = t \mapsto (p, v) \\ \wedge \text{dom}(\chi_o) \cup \text{scanned } \Omega \subseteq (\Omega' \downarrow t)\}$$

$$\{\chi_s = \emptyset\}$$

`scan() : array A`

$$\{\mathbf{r}. \exists t. \chi'_s = \emptyset \wedge \mathbf{r} = \text{eval } t \Omega' \chi' \\ \wedge \text{dom}(\chi) \subseteq (\Omega' \downarrow t) \wedge t \in \text{scanned } \Omega'\}$$

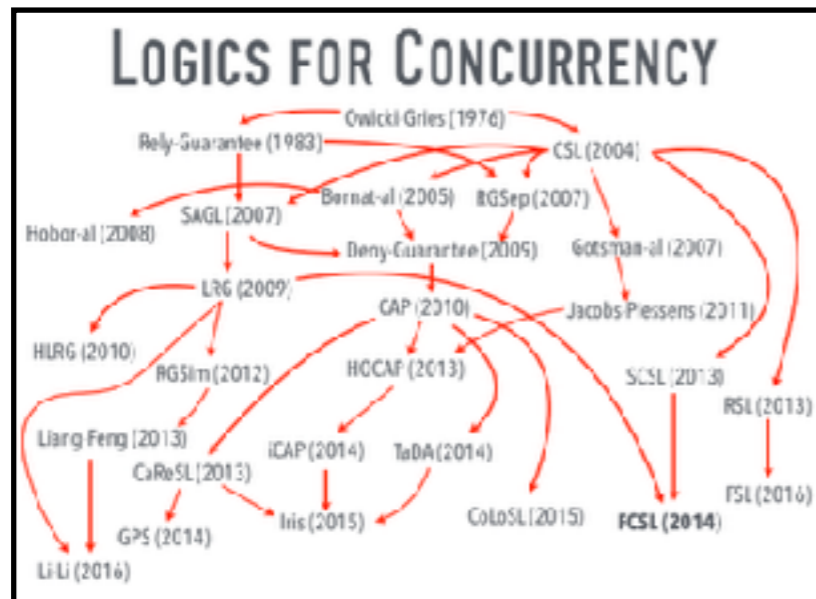
DATA STRUCTURES LINKED IN TIME

- Reasoning about **concurrent objects** is tricky, and tracking **linearization points** can be cumbersome.
- Creating a new **program logic/technique** for each new corner case/consistency criteria does not scale up.
- Introduced **Linking-in-time** as an alternative to explicit reasoning about **non-fixed , non-local, LPs**.
- Mechanized in **Coq**, including the **first formal correctness proof** of Jayanti's snapshot object.



THANKS

COPYRIGHT DISCLAIMERS



The CSL family tree slide is a variation of Ilya Sergey's original:
<http://ilyasergey.net/other/CSL-Family-Tree.pdf>



The Mighty Rooster is due to Lilia Asiminova
<http://www.liliaanisimova.com/>
Handle with caution!