

# Thread-Modular Reasoning for Lock-Free Data Structures

---

Roland Meyer

based on joint work with Lukáš Holík, Tomáš Vojnar, and Sebastian Wolff.



Technische  
Universität  
Braunschweig

# Lock-Free Data Structures

---

## Key Take Aways:

- efficient but complex
- correctness = linearizability
- checking linearizability reduces to reachability



# Concept

---

- avoid locks
  - ➔ critical section cannot exist
- single commands are atomic
  - ➔ compare-and-swap (CAS)

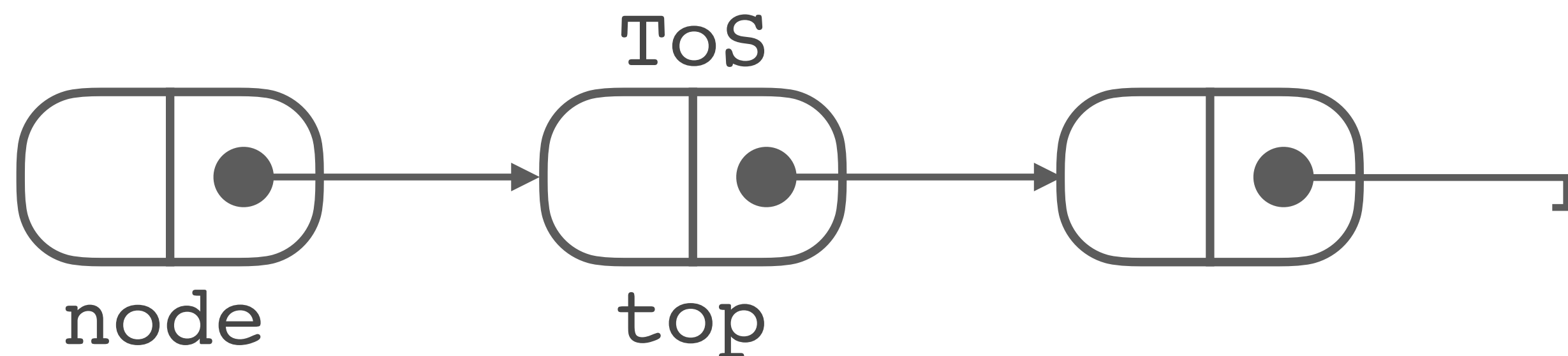
```
CAS(src, cmp, dst) := atomic {  
    if (src != cmp) return false;  
    src = dst;  
    return true;  
}
```

# Example: Treiber's Stack

---


```
➡ push(val):  
  node = new Node(val);  
  while (true) {  
    top = ToS;  
    node.next = top;  
    if (CAS(ToS, top, node))  
      return;  
  }
```

```
pop():  
  while (true) {  
    top = ToS;  
    if (top == NULL)  
      return EMPTY;  
    next = top.next;  
    if (CAS(ToS, top, next))  
      return top.data;  
  }
```

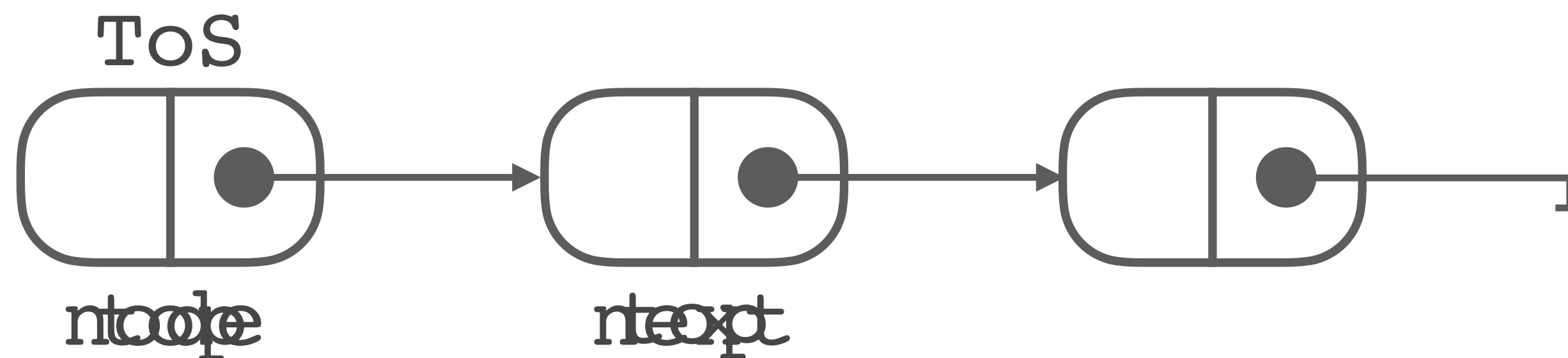


# Example: Treiber's Stack

---

```
push(val):  
  node = new Node(val);  
  while (true) {  
    top = ToS;  
    node.next = top;  
    if (CAS(ToS, top, node))  
       return;  
  }
```

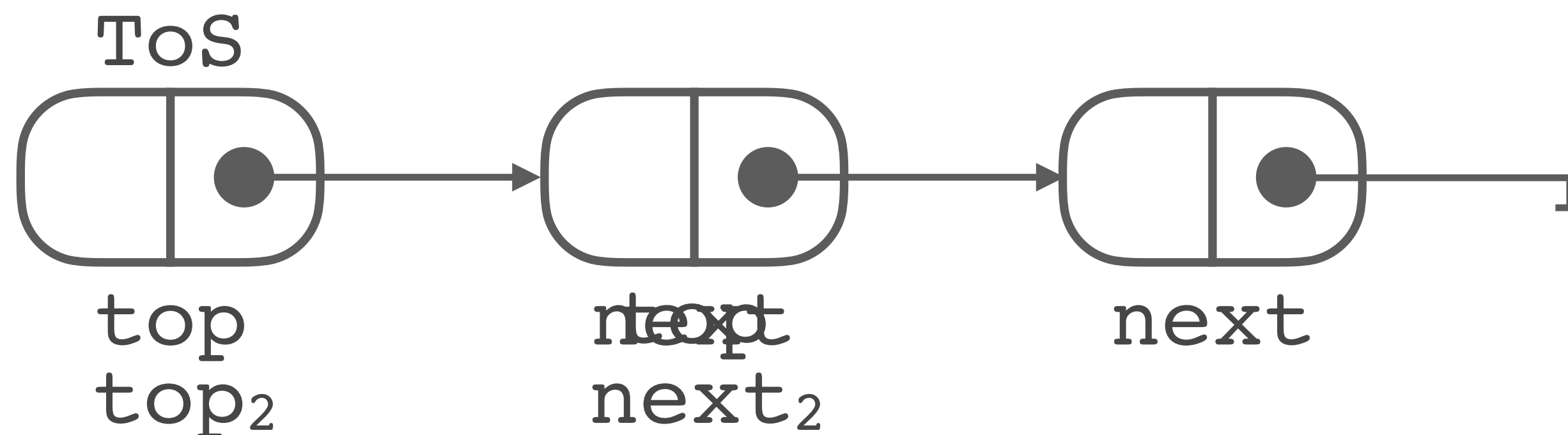
```
pop():  
  while (true) {  
    top = ToS;  
    if (top == NULL)  
      return EMPTY;  
    next = top.next;  
    if (CAS(ToS, top, next))  
      return top.data;  
  }
```



# Example: Treiber's Stack

```
push(val):  
  node = new Node(val);  
  while (true) {  
    top = ToS;  
    node.next = top;  
    if (CAS(ToS, top, node))  
      return;  
  }
```

```
pop():  
  while (true) {  
    top = ToS;  
    if (top == NULL)  
      return EMPTY;  
    next = top.next;  
    if (CAS(ToS, top, next))  
      return top.data;  
  }
```



# Correctness and Concurrency

---

- pre/post conditions meaningless
  - ➔ other correctness criteria required
- linearizability
  - ➔ every concurrent run must coincide with a sequential run
  - ➔ most common for lock-free data structures
  - ➔ illusion of sequentiality [Filipović et al. ESOP'09]:
    - linearizable  $\iff$  sequential and concurrent implementation are observationally equivalent

# Checking Linearizability

---

- check sequentiality illusion
  - ➔ sufficient: sequence of linearization points is valid [Abdulla et al. TACAS'13]  
(intuitively: linearization point = change of data structure takes effect)

$$\text{concurrent}(DS) \models \text{sequential}(DS)$$

$$\iff \text{linp}(DS) \subseteq \text{sequential}(DS)$$

$$\iff \text{linp}(DS) \cap \overline{\text{sequential}(DS)} = \emptyset$$

$$\iff \text{linp}(DS) \cap \text{observer}(DS) = \emptyset$$

- ➔ checking linearizability is a **reachability problem**



# Overview

---

1. thread-modular reasoning
2. ownership
3. summaries

# Thread-Modular Reasoning

[Qadeer, Flanagan SPIN'03]

---

Key Take Aways:

- compute reachability
- interference is key to scalability



# Concept

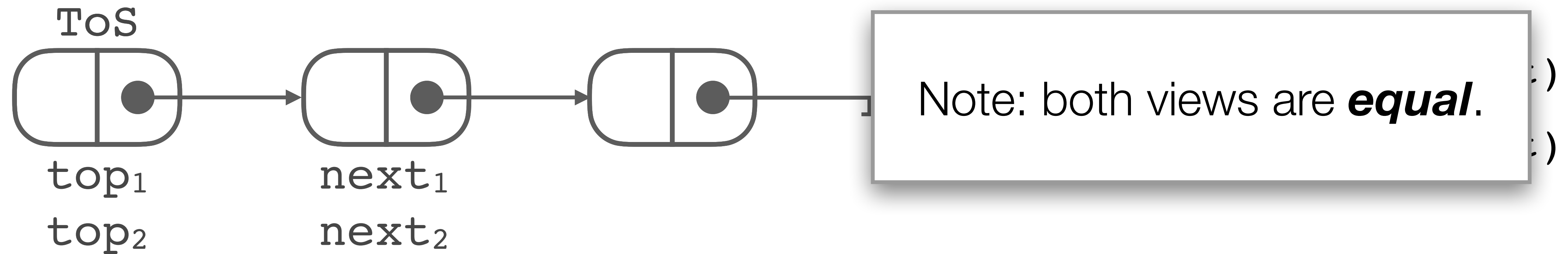
---

- view abstraction
  - ➔ split states into set of views
  - ➔ views capture perception of 1 thread (abstract from correlation)
- state exploration
  - ➔ fixed-point computation:

$$X = X \cup \textit{sequential}(X) \cup \textit{interference}(X)$$

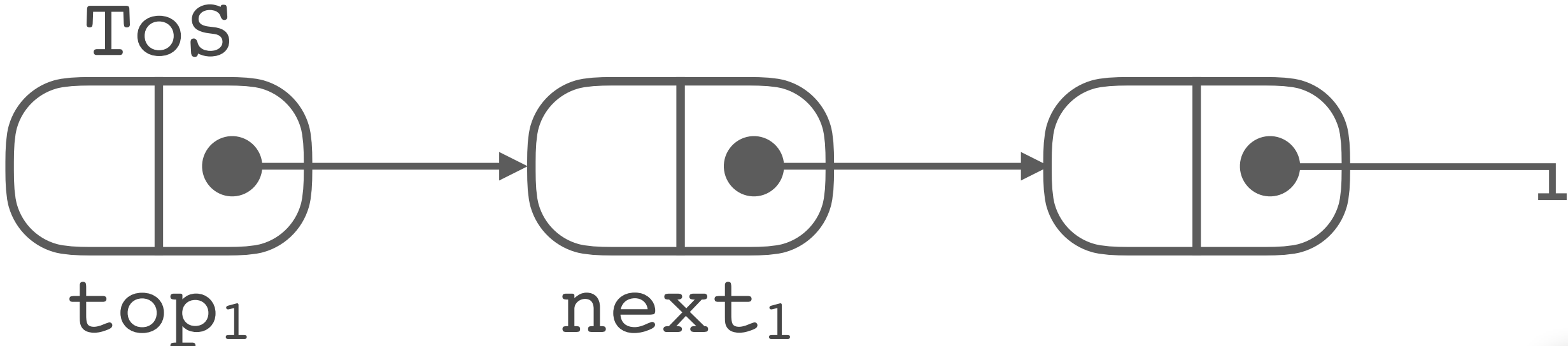
# Example: View Abstraction


$$X = X \cup \text{sequential}(X) \cup \text{interference}(X)$$



# Example: Sequential Step

$$X = X \cup \textit{sequential}(X) \cup \textit{interference}(X)$$

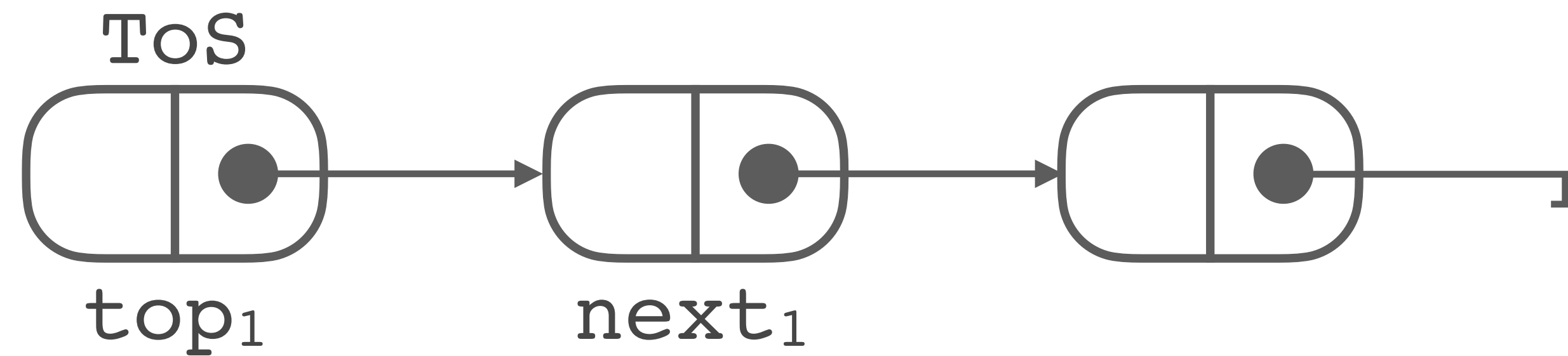


 `CAS(ToS, top, next)`

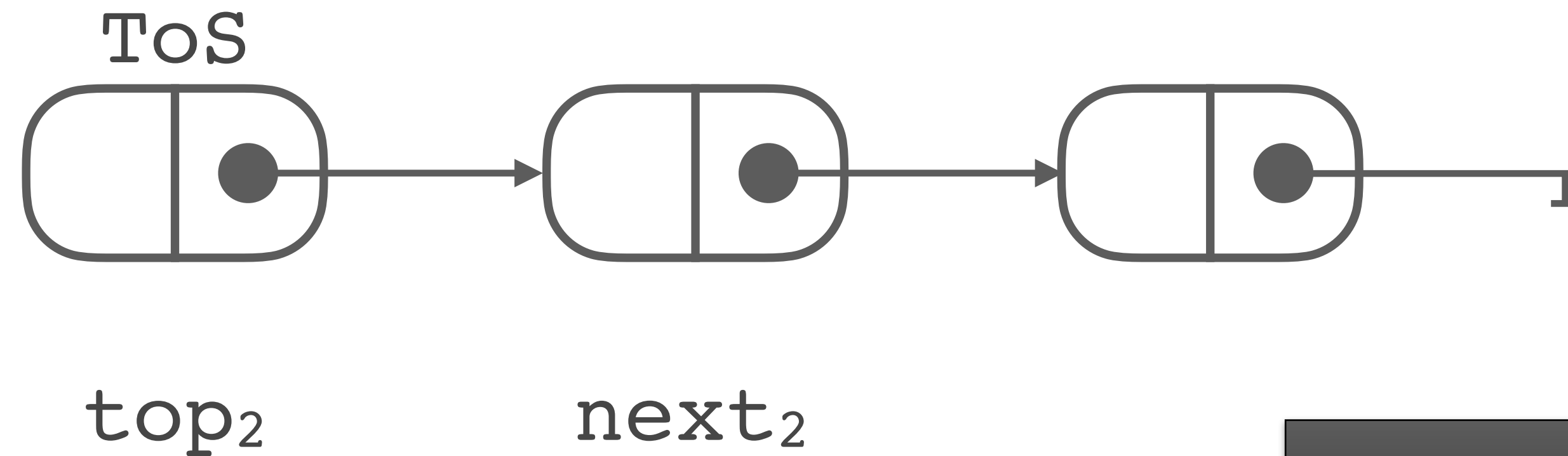
No concurrent behavior.

# Example: Interference Step

$$X = X \cup \text{sequential}(X) \cup \text{interference}(X)$$



**1**  $\rightarrow$  CAS(ToS, top, next)

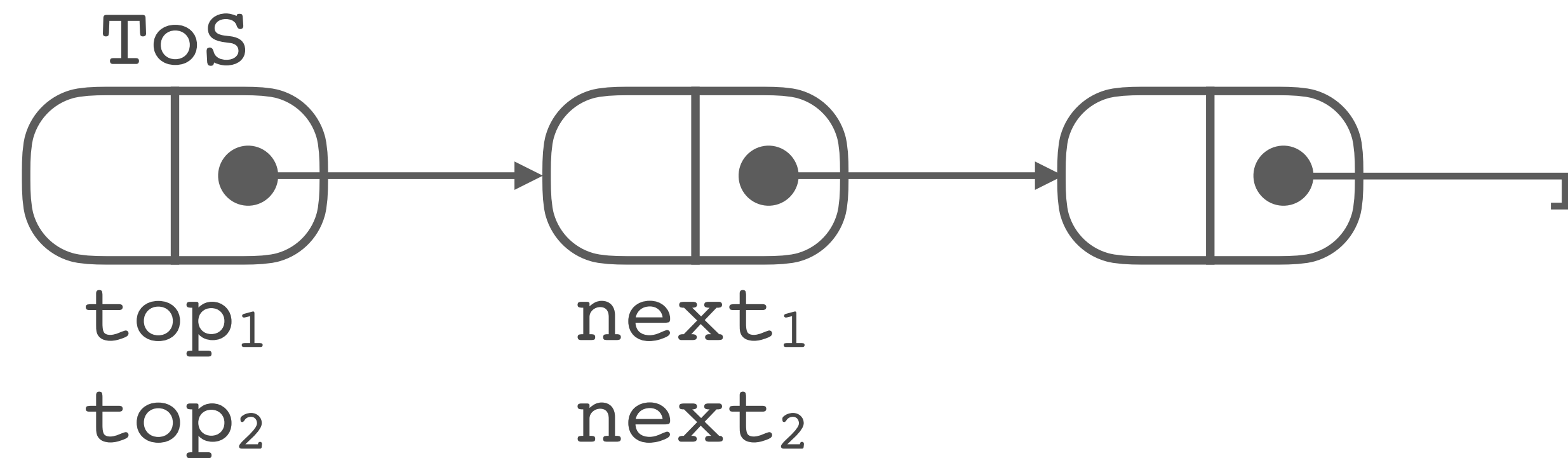


**2**  $\rightarrow$  CAS(ToS, top, next)

**1. combine**

# Example: Interference Step

$$X = X \cup \text{sequential}(X) \cup \text{interference}(X)$$



**1** → CAS (ToS, top, next)  
**2** → CAS (ToS, top, next)

**1. combine**

**2. step**

**3. project**

# Challenges with Interference

---

- number of possible combinations is enormous
  - ➔ not all combinations are reasonable
- need **pruning** to make the approach practical
  - ➔ precision
  - ➔ performance
- pruning must be sound



# Pruning Interferences

---

two types

- **matching**
  - ➔ Is it possible to combine at all? Skip if not.
- **correlation**
  - ➔ Which nodes should coincide?

# Matching: Complication

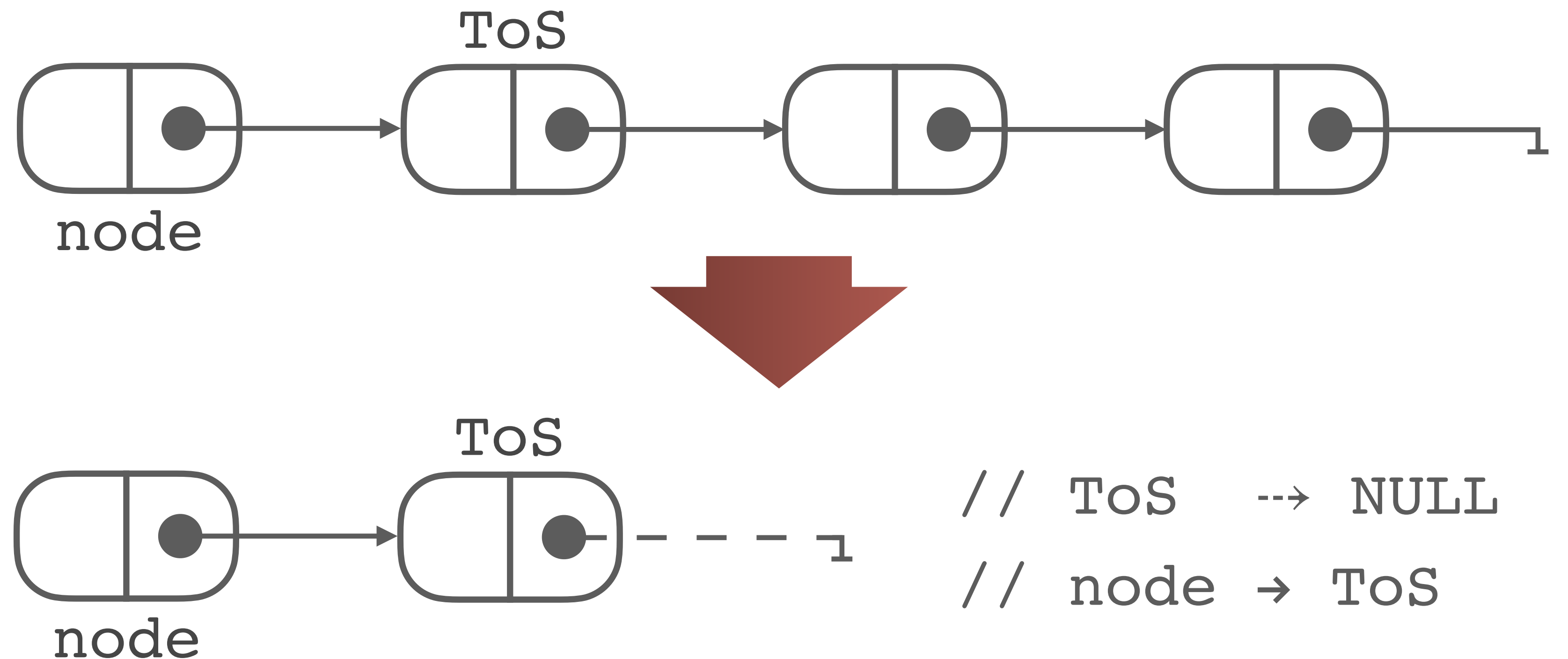
- matching gets harder due to **finite abstraction**
- we use reachability predicates (shape analysis):

- 0-step: =

- 1-step:  $\rightarrow$

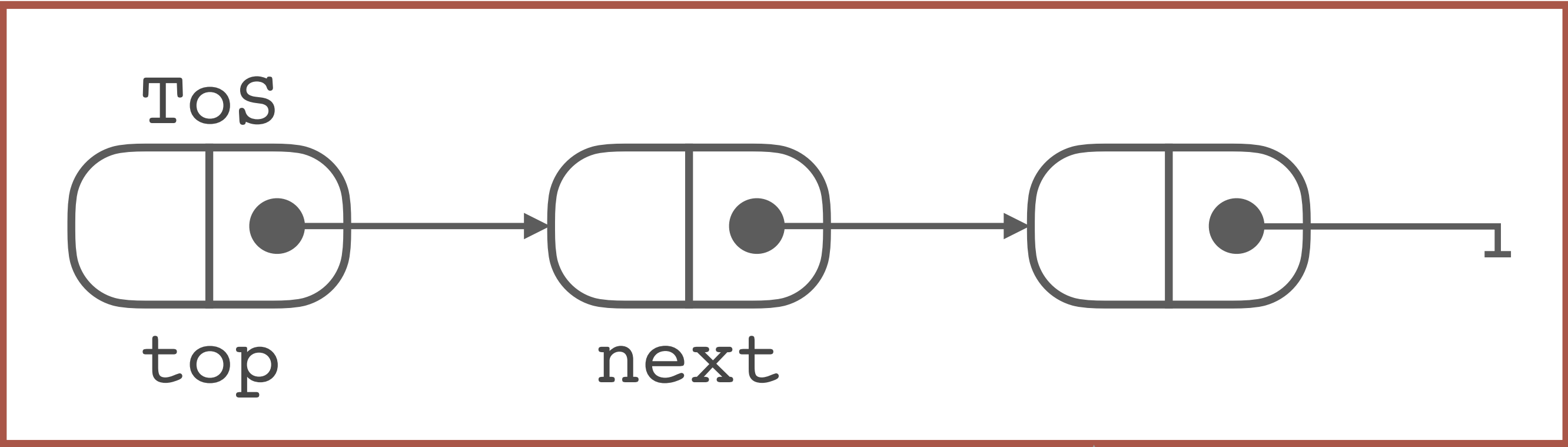
- n-step:  $\dashrightarrow$

- unreachable:  $\nexists$

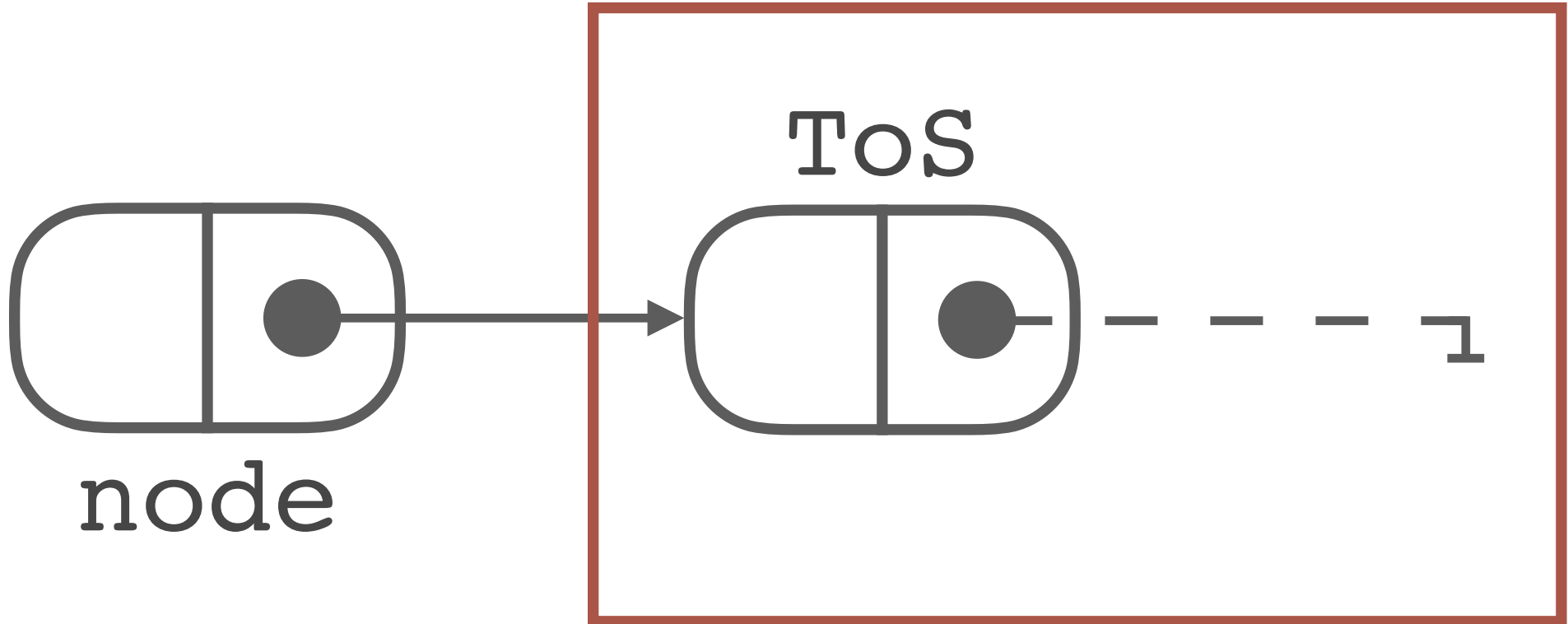


# Matching: Example

**Subgraph  
isomorphism:  
NP-complete!**

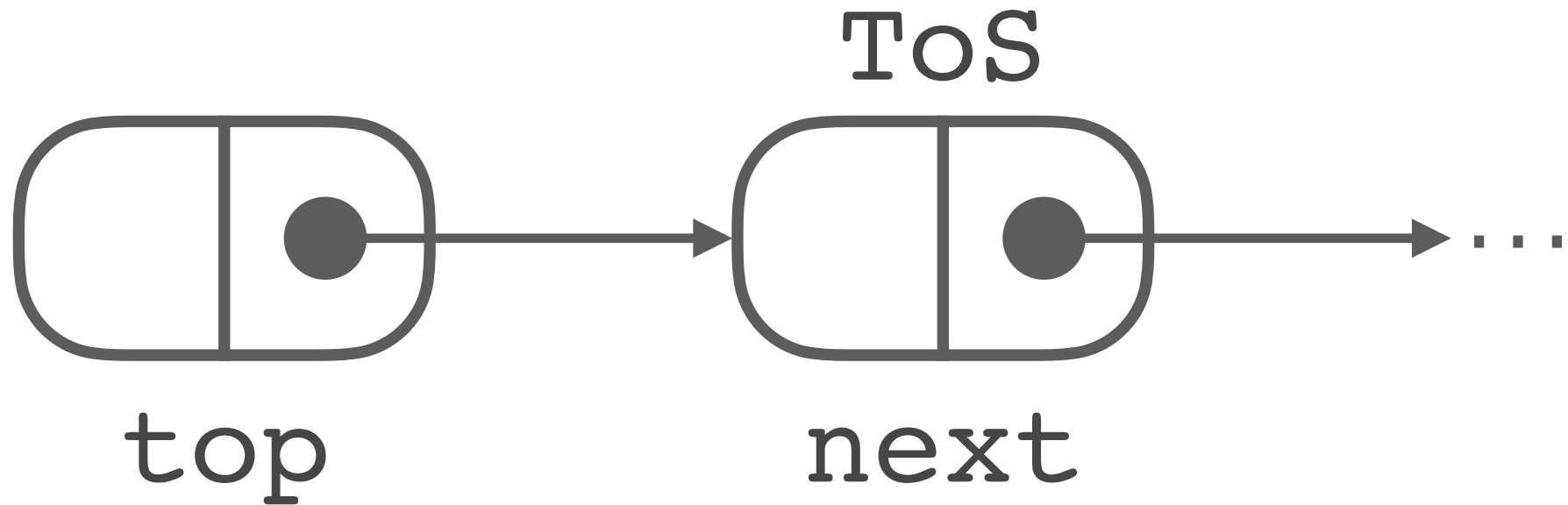
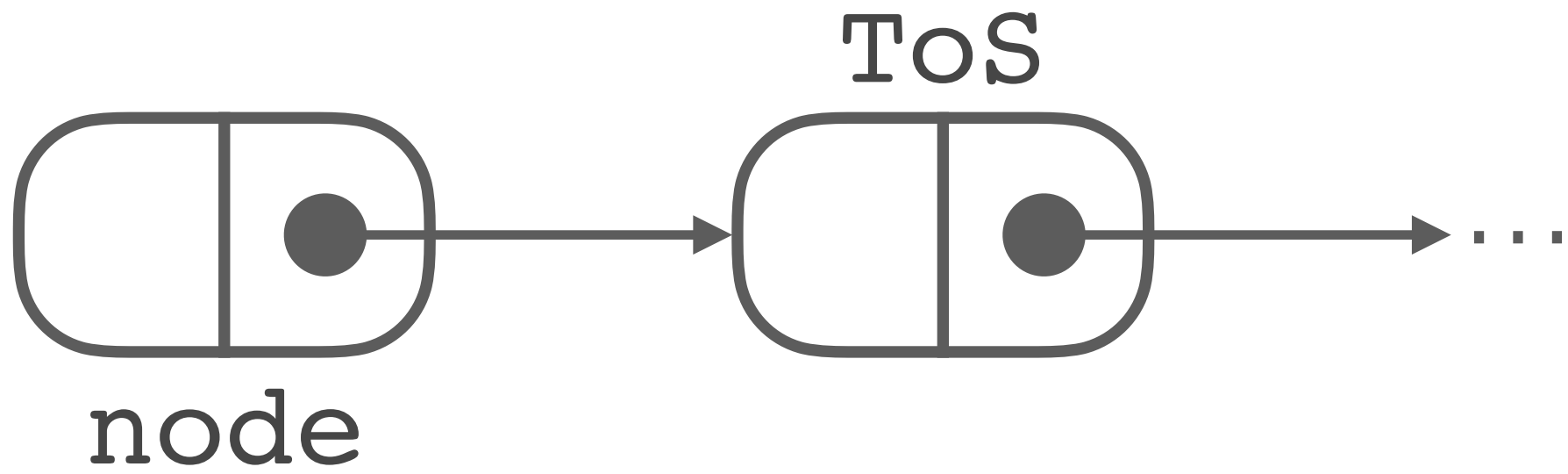


||

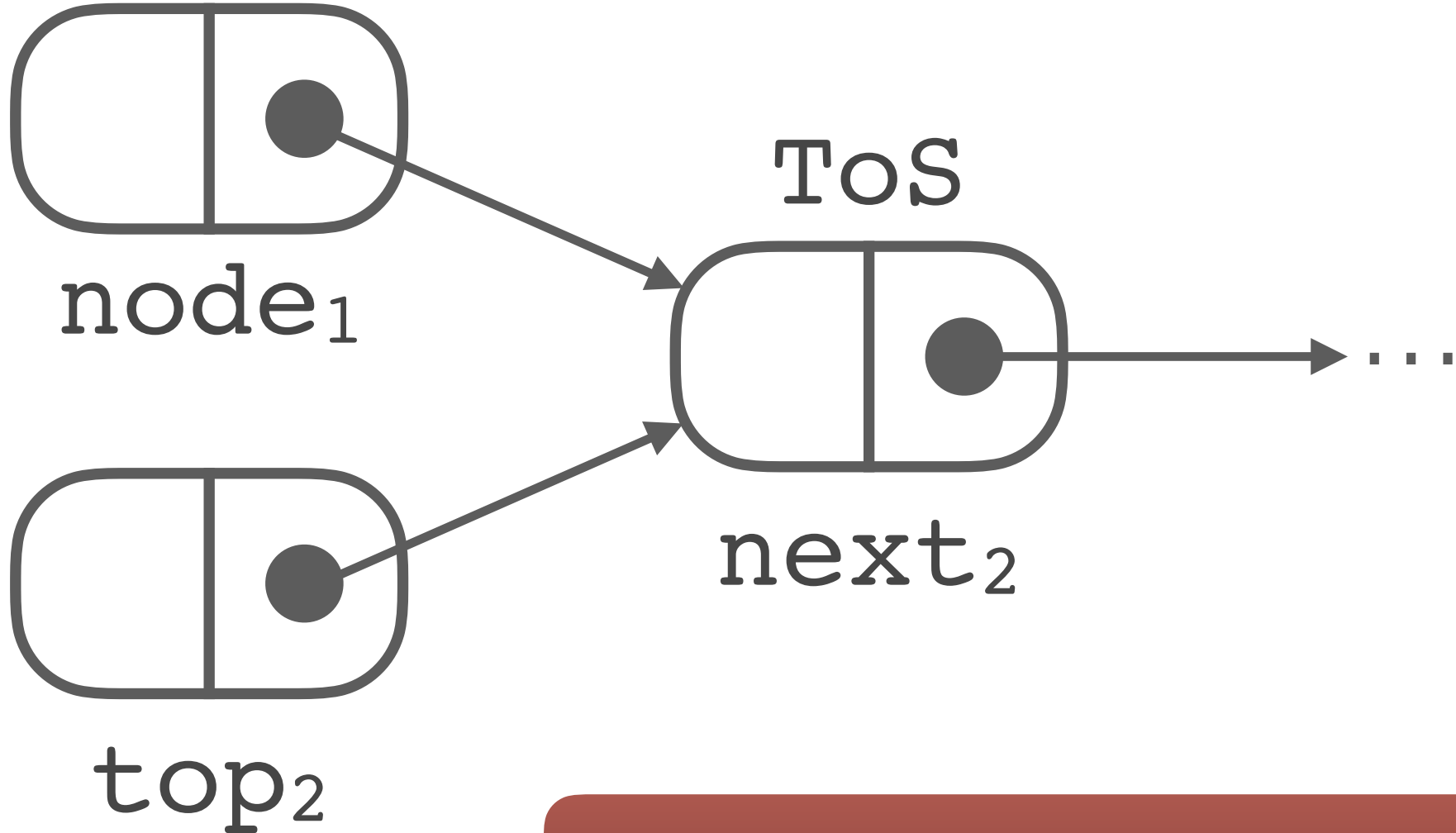


logical stack  
content

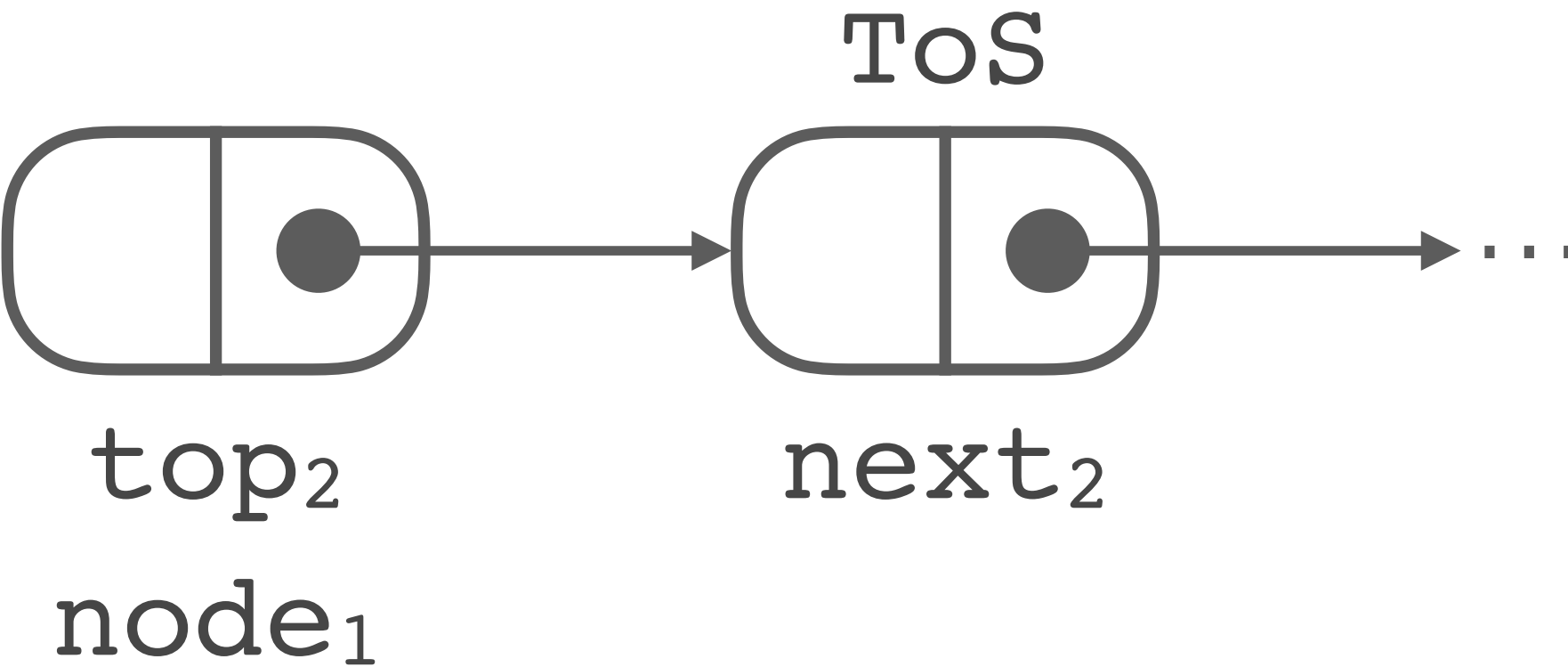
# Correlation: Example



??



**Exponentially many!**



# Practicality is about Interference

---

poor  
scalability

- interference
  - ➔ quadratic in size of state space
- matching
  - ➔ subgraph isomorphism (NP)
- correlation
  - ➔ exponential

fight imprecision  
(false-positives)

# Ownership

---

## Key Take Aways:

- ownership saves the day
- even under explicit memory management



# Concept

---

partition allocated heap into

- **owned**
  - ➔ exclusive access for a single thread
  - ➔ granted upon allocation
- **shared**
  - ➔ accessible by every thread
  - ➔ by publishing (e.g. making accessible via shared variables)

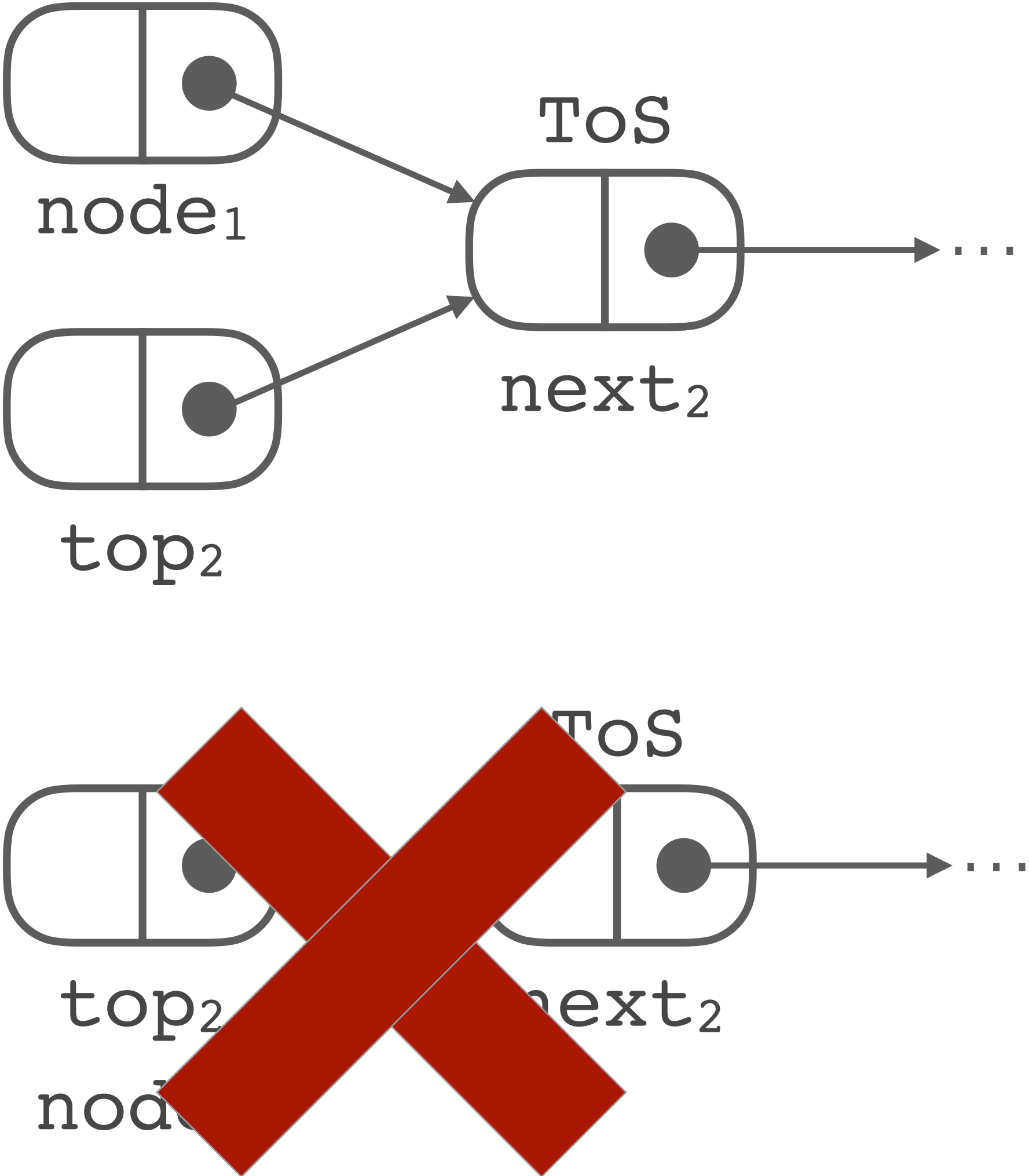
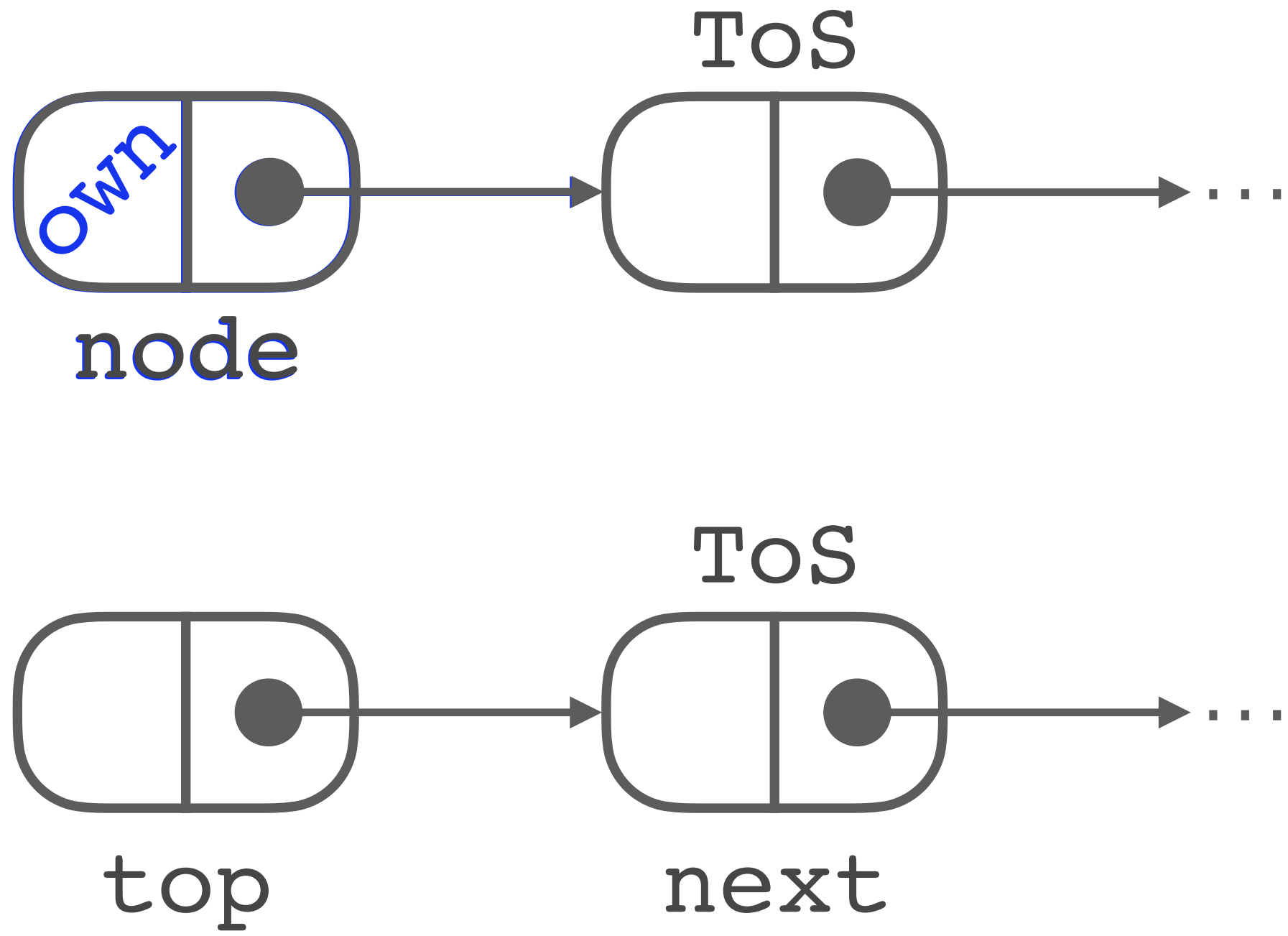
# Ownership in Thread-Modular Reasoning [Gotsman et al. PLDI'07]

---

- track ownership
  - ➔ small overhead
- matching
  - ➔ owned cells not contained
- correlation
  - ➔ owned cells not merged with other nodes



# Ownership and Correlation



# Ownership in Thread-Modular Reasoning

---

- helps a lot with
  - ➔ matching
  - ➔ correlation
- makes thread-modular reasoning practical
  - ➔ prunes false-positives

**Only for garbage collection (GC)!**

**What about explicit memory management (MM)?**

# Problem with MM

---

**Ownership does not exist under  
explicit memory management.**

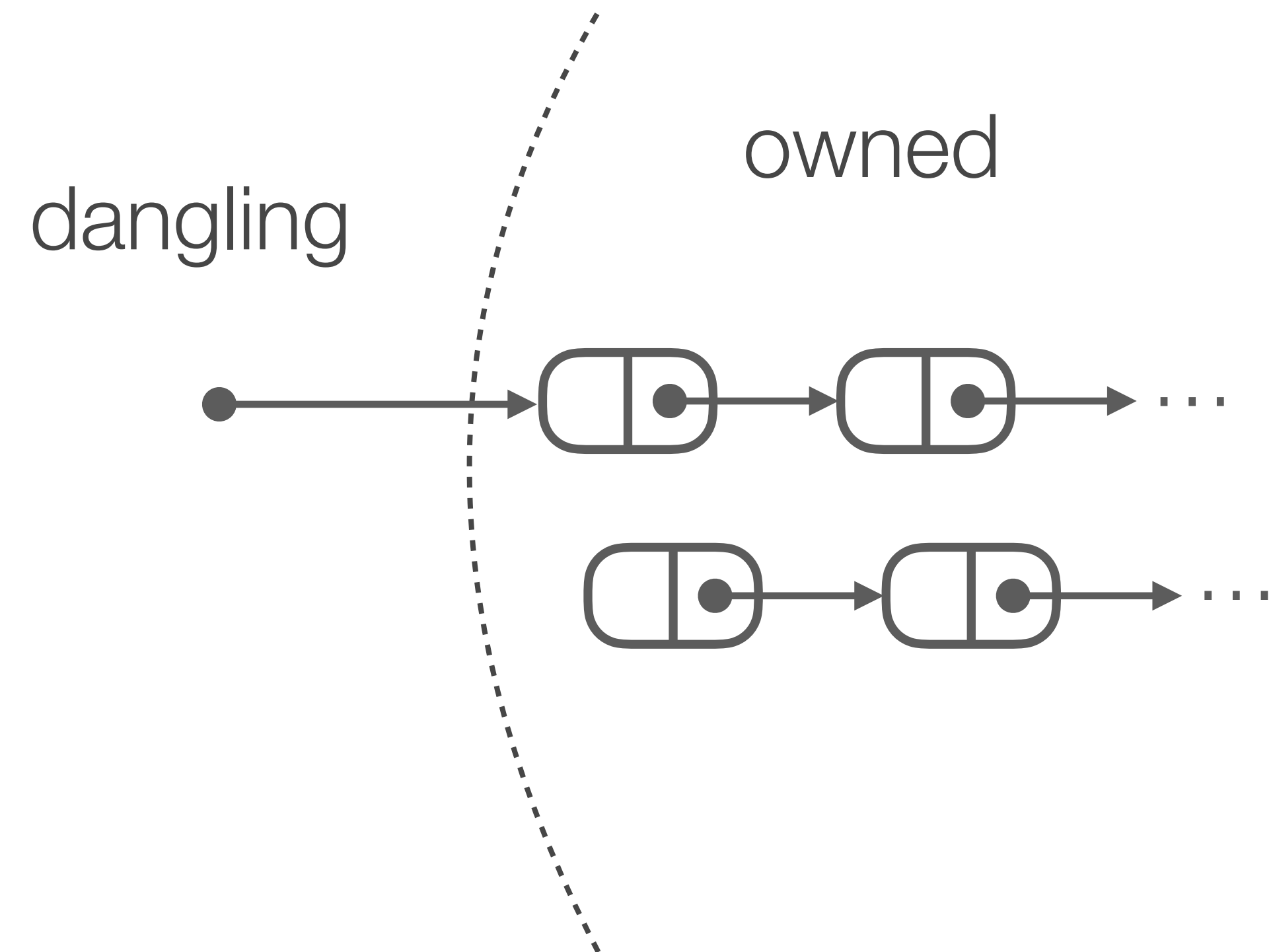
— folklore

- almost true
- indeed no exclusivity ➔ dangling pointers
- we introduced **weak ownership** in VMCAI'16

# Weak Ownership



[VMCAI'16]

- write exclusivity
  - ➔ only owners may write
- no read exclusivity **NEW!**
  - ➔ dangling readers allowed
  - ➔ dangling reads *unsafe*
  - ➔ only owner may rely on memory contents



# Weak Ownership in Thread-Modular Reasoning

[VMCAI'16]

- track dangling pointers
  - ➔ small overhead
- matching: like normal ownership
- correlation
  - ➔ -owned cells referenced by  only via dangling pointers
- dangling write accesses may be unsafe
  - ➔ report as bug

# Performance Impact

[VMCAI'16]

	MM without ownership		MM <b>with</b> ownership
Treiber's stack	944s	↓ :37	25.5s
	#116776	↓ :36	#3175
Michael&Scott's queue	false positive		11700s
	> #69000	<b>impractical</b>	#19742

# Accomplishments

---

- ownership helps with matching and correlation
- low overhead tracking additional info
- deeming unsafe accesses as bugs reflects programming practice
- performance improvements for analysis
- but: **not practical** yet
  - ➔ interference still computationally complex

# Summaries

---

## Key Take Aways:

- copy-and-check blocks
- statelessness
- efficient interference





# Observation

---

- lock-freedom relies on copy-and-check blocks
    1. create local copy of shared data
    2. make changes locally
    3. publish changes if copy up-to-date or retry otherwise
- ➔ updates appear atomically

```
push(val):  
    node = new Node(val);  
    while (true) {  
        ① top = ToS;  
        ② node.next = top;  
        ③ if (CAS(ToS, top, node))  
            return;  
    }
```

**Threads cannot observe the  
local behavior of other threads.**

— SAS'17

So why do interference for all intermediate steps?

- ➔ instead: apply updates in one shot
- ➔ potentially unsound: stay tuned

## Example: Summary for pop

---

```
atomic {  
    while (true) {  
        top = ToS;  
        if (top == NULL)  
            return EMPTY;  
        next = top.next;  
        if (CAS(ToS, top, next))  
            return top.data;  
    }  
}
```

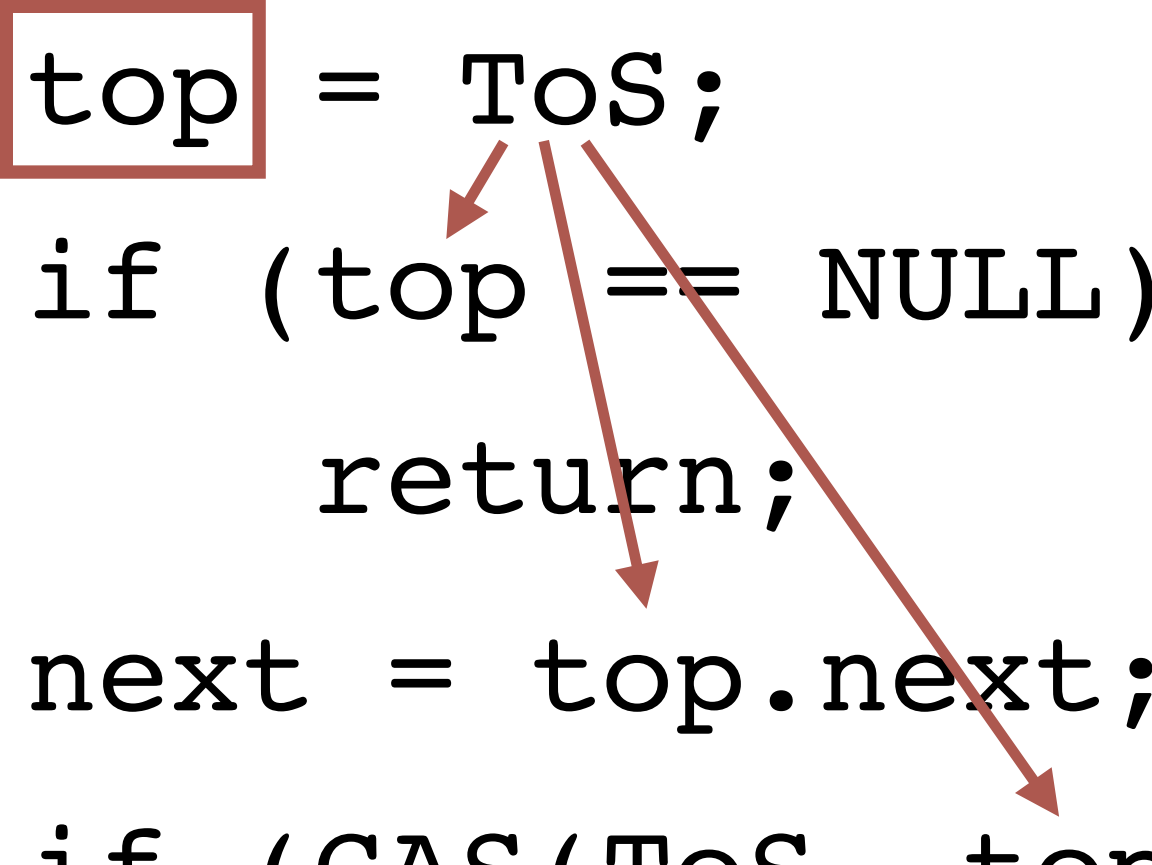
**1. make atomic**

**2. remove noise**

## Example: Summary for pop

---

```
atomic {  
    while (true) {  
        top = ToS;  
        if (top == NULL)  
            return;  
        next = top.next;  
        if (CAS(ToS, top, next))  
            return;  
    }  
}
```



**1. make atomic**

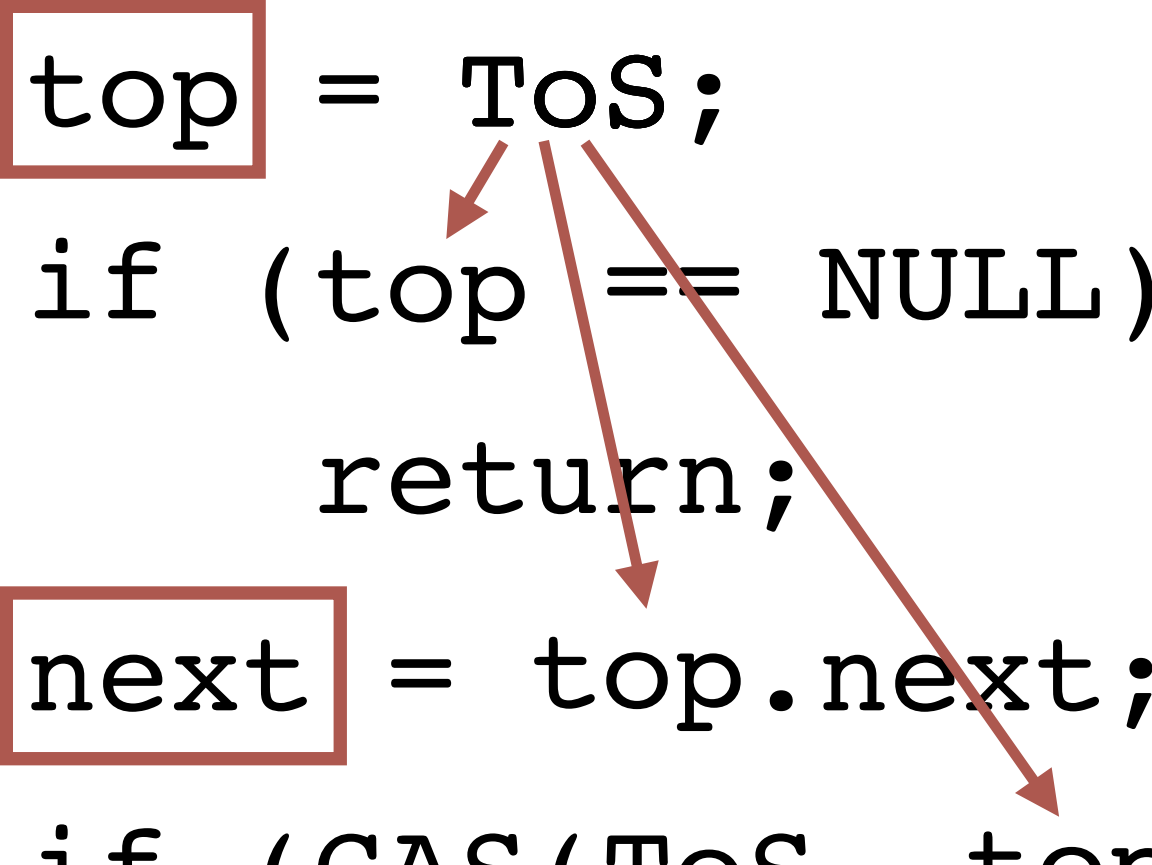
**2. remove noise**

**3. copy propagation**

## Example: Summary for pop

---

```
atomic {  
    while (true) {  
        top = ToS;  
        if (top == NULL)  
            return;  
        next = top.next;  
        if (CAS(ToS, top, next))  
            return;  
    }  
}
```



**1. make atomic**

**2. remove noise**

**3. copy propagation**

## Example: Summary for pop

---

```
atomic {  
    while (true) {  
  
        if (ToS == NULL)  
            return;  
  
        if (CAS(ToS, ToS, ToS.next))  
            return;  
  
    }  
}
```

**1. make atomic**

**2. remove noise**

**3. copy propagation**

**4. remove noise**

**5. rewrite CAS**

## Example: Summary for pop

---

```
atomic {  
  
    if (ToS == NULL)  
        return;  
  
    if (CAS(&ToS, ToS, ToS.next))  
        return;  
  
}
```

**1. make atomic**

**2. remove noise**

**3. copy propagation**

**4. remove noise**

**5. rewrite CAS**

## Example: Summary for pop

---

```
atomic {  
  
    assume(&TOS != NULL);  
    return;  
  
    TOS = TOS.next;  
    return;  
  
}
```

**1. make atomic**

**2. remove noise**

**3. copy propagation**

**4. remove noise**

**5. rewrite CAS**

**6. rewrite guard**



## Example: Summary for `pop`

---

```
atomic {
```

```
    assume(ToS != NULL);
```

```
    ToS = ToS.next;
```

```
}
```

**1. make atomic**

**2. remove noise**

**3. copy propagation**

**4. remove noise**

**5. rewrite CAS**

**6. rewrite guard**

## Example: Summary for `pop`

---

```
atomic {  
    assume(ToS != NULL);  
    ToS = ToS.next;  
}
```

- easy to compute  
    ➔ similar for push
- compact form beneficial for analysis  
    (and understandability)

**1. make atomic**

**2. remove noise**

**3. copy propagation**

**4. remove noise**

**5. rewrite CAS**

**6. rewrite guard**

**Summaries are stateless.**

— SAS'17

learn about an object's state from shared variables (`assume`);  
and execute atomically

→ no concurrency:  $\prod_i summary_i = \sum_i summary_i$

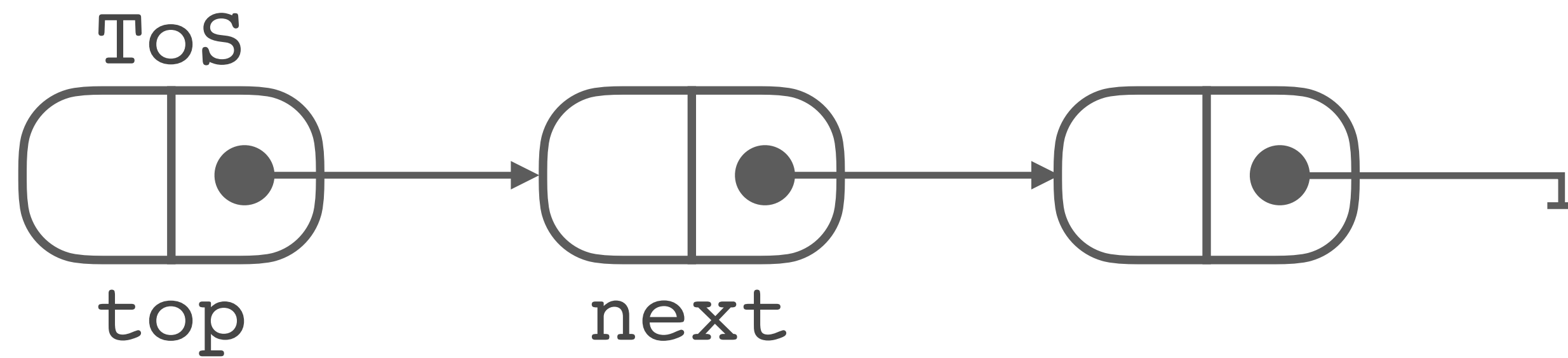
→ no interference for summaries needed

**Finally an efficient  
interference algorithm!**



# Example: New Interference

---



```
CAS(ToS, top, next)
```



```
atomic {  
    assume(ToS != NULL);  
    ToS = ToS.next;  
}
```

# Soundness

---

- soundness requires summaries to
  1. capture all possible effects of the implementation
  2. be stateless
- both can be checked on the fixed point
  1. for each effect check whether some summary can do it
  2. summaries must not rely on uninitialized local variables

# Accomplishments

---

- improved interference
  - ➔ matching: NP  $\implies$  **not needed**
  - ➔ correlation: exponential  $\implies$  **constant** (one)
  - ➔ interference: quadratic (in fixed-point approximant)  $\implies$  **linear**
- *sound approach despite unsound abstraction*
- works for explicit memory (requires ownership transfer, skipped)

# Performance Impact: GC

	classical		summaries
Coarse Stack	0.29s	↓ :10	0.03s
Coarse Queue	0.49s	↓ :10	0.05s
Treiber's stack	1.99s	↓ :33	0.06s
Michael&Scott's queue	11.0s	↓ :28	0.39s
DGLM queue	9.56s	↓ :25	0.37s

# Performance Impact: MM

	classical		summaries
Coarse Stack	1.89s	↓:10	0.19s
Coarse Queue	2.34s	↓:2	0.98s
Treiber's stack	25.5s	↓:15	1.64s
Michael&Scott's queue	11700s	↓:114	102s
DGLM queue	false-positive	✗	violation



# Related Work

---

## Key Take Aways:

- Abdulla et al.
- Vafeiadis et al.



## Abdulla et al.

---

- improve precision of interference
  - ➔ first to make it work for explicit memory management
  - ➔ without weak ownership
- increase threads per view to 2
  - ➔ could restore precision for matching and correlation
- poor scalability due to increased state space

## Vafeiadis et al.

---

- relies on RGSep (separation logic + rely guarantee)
- fixed point:
  - ➔ interference recorded per thread in every step
  - ➔ applied to others in next iteration
- corresponds to learning summaries
  - ➔ no freedom: sound in every step
  - ➔ linear in fixed point (here: linear in program size)
- only considered garbage collection

# Future Work

---

- stateful summaries
- go beyond singly-linked objects
- more benchmarks



An aerial, black and white photograph of a city, likely Cambridge, Massachusetts, showing a dense urban landscape with numerous buildings, streets, and green spaces. A large, multi-lane highway is visible in the lower-left quadrant. The word "FIN" is superimposed in large, bold, black capital letters in the center of the image.

**FIN**