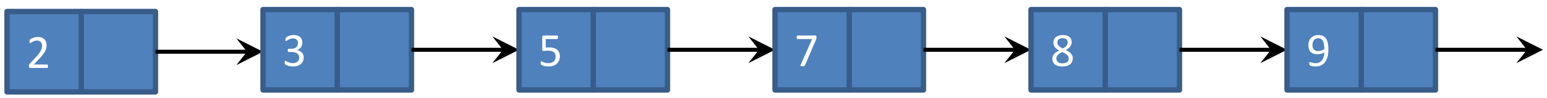
Introduction to Permission-Based Program Logics

Part II – Concurrent Programs

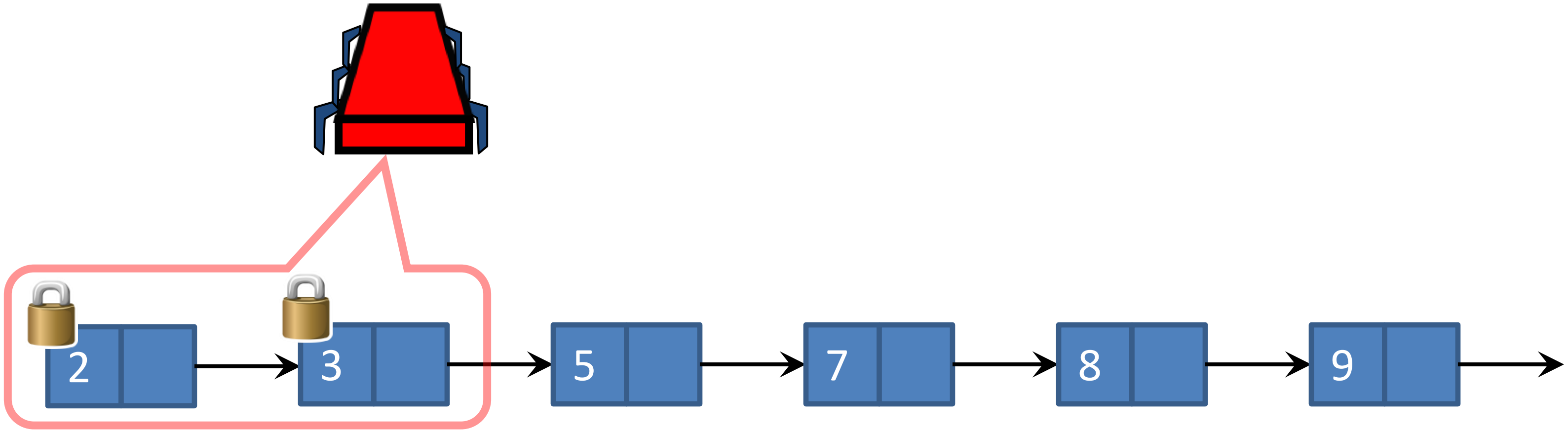
Thomas Wies
New York University

Example: Lock-Coupling List



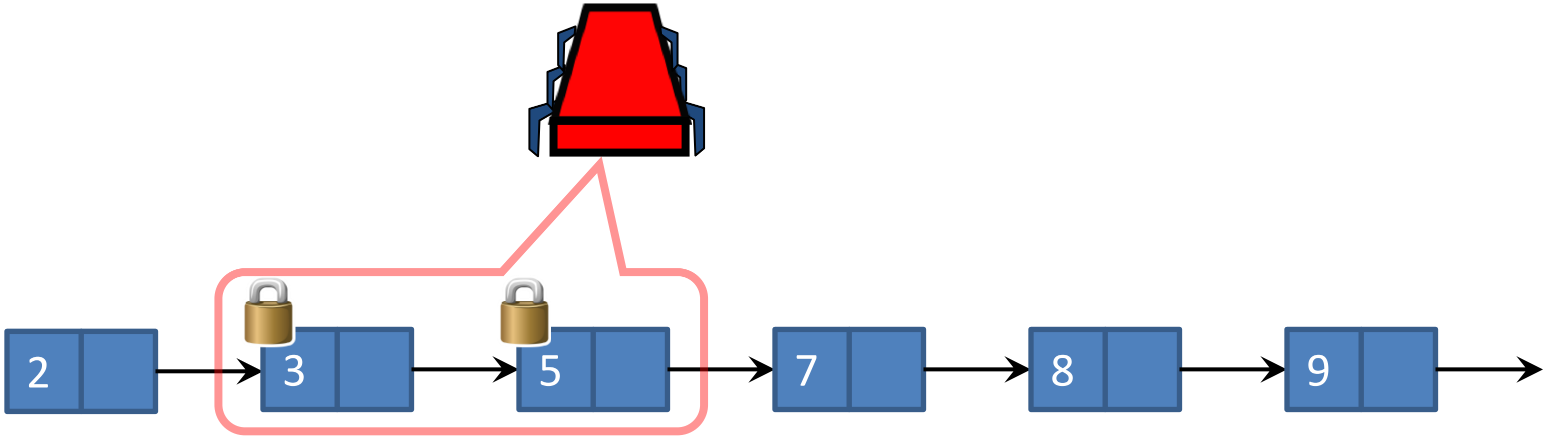
- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Example: Lock-Coupling List



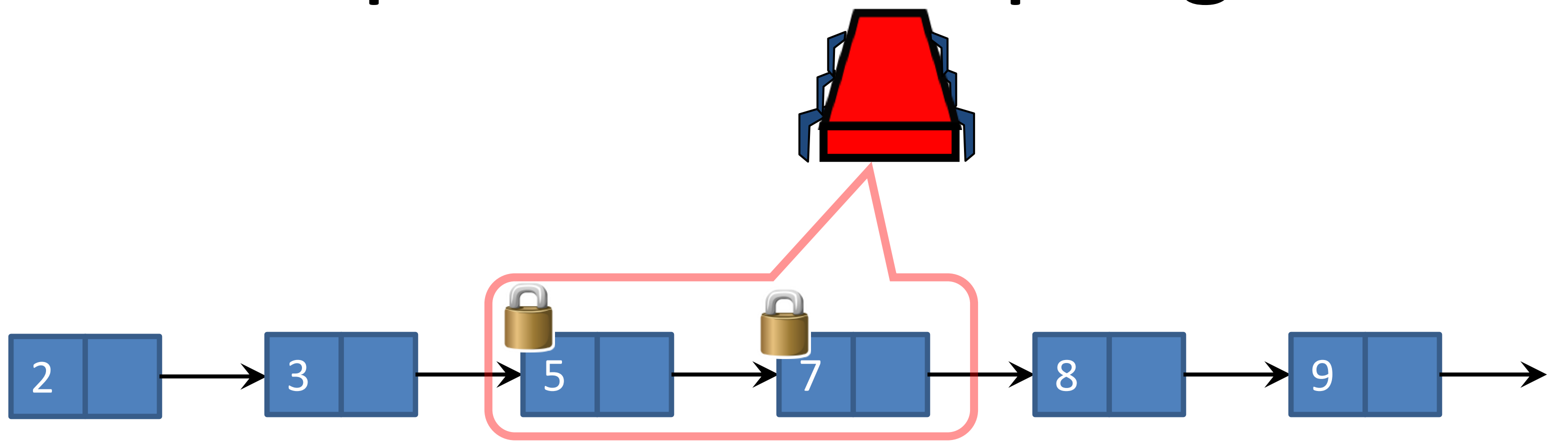
- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Example: Lock-Coupling List



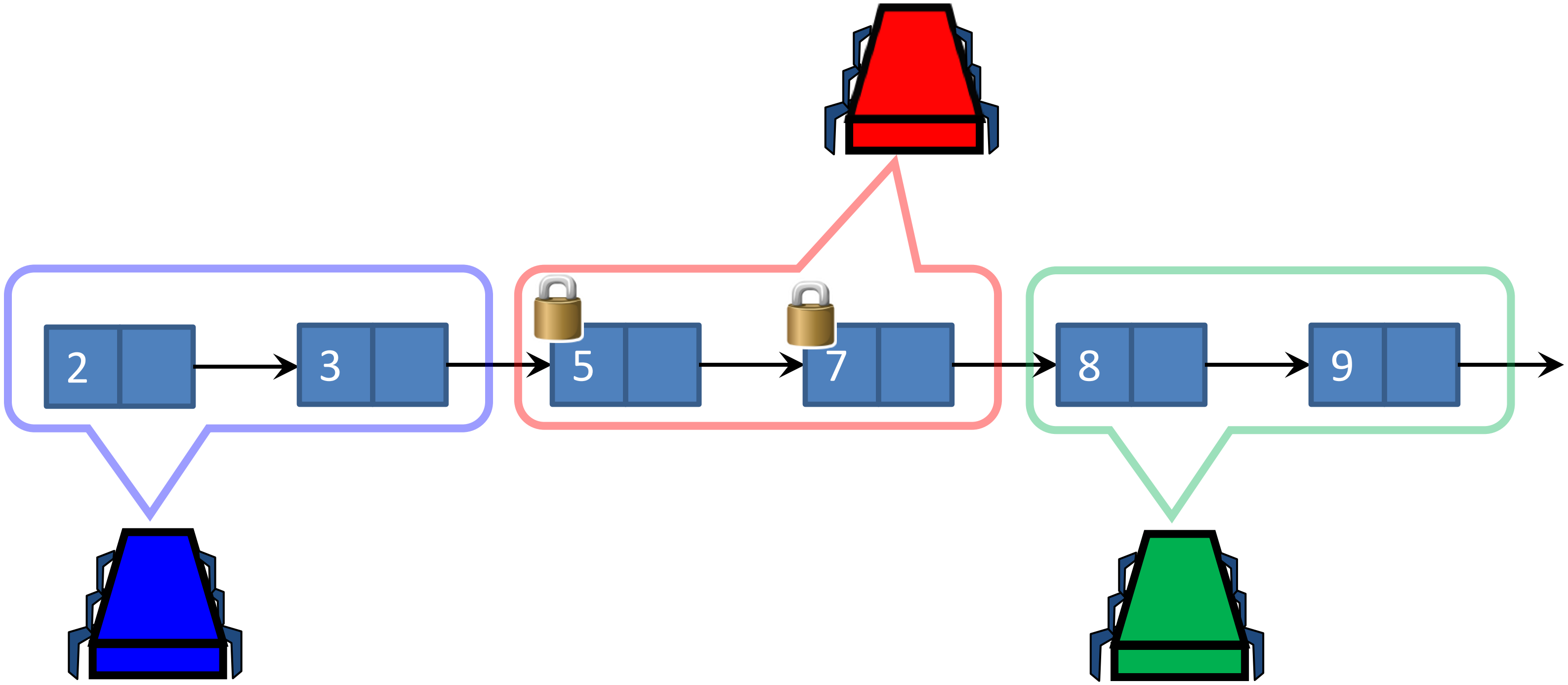
- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Example: Lock-Coupling List



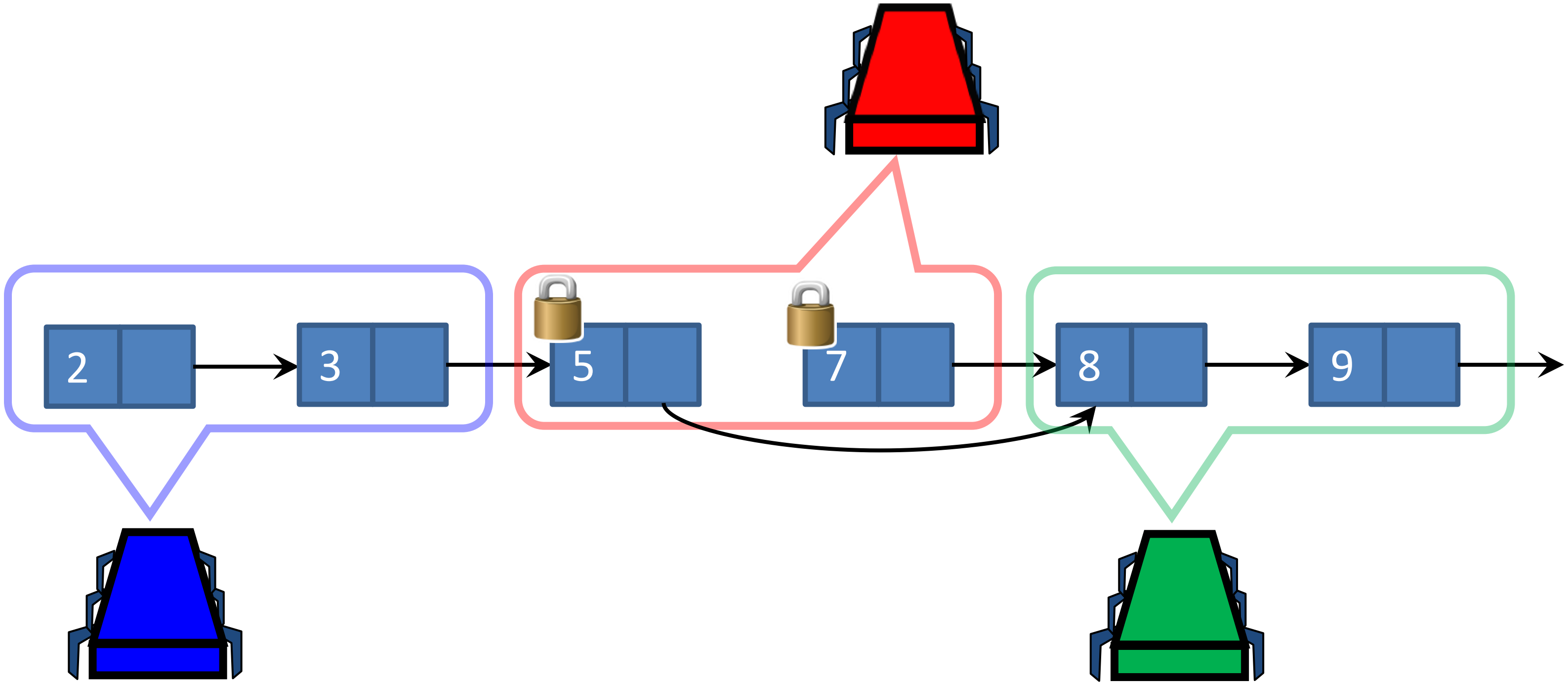
- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Example: Lock-Coupling List



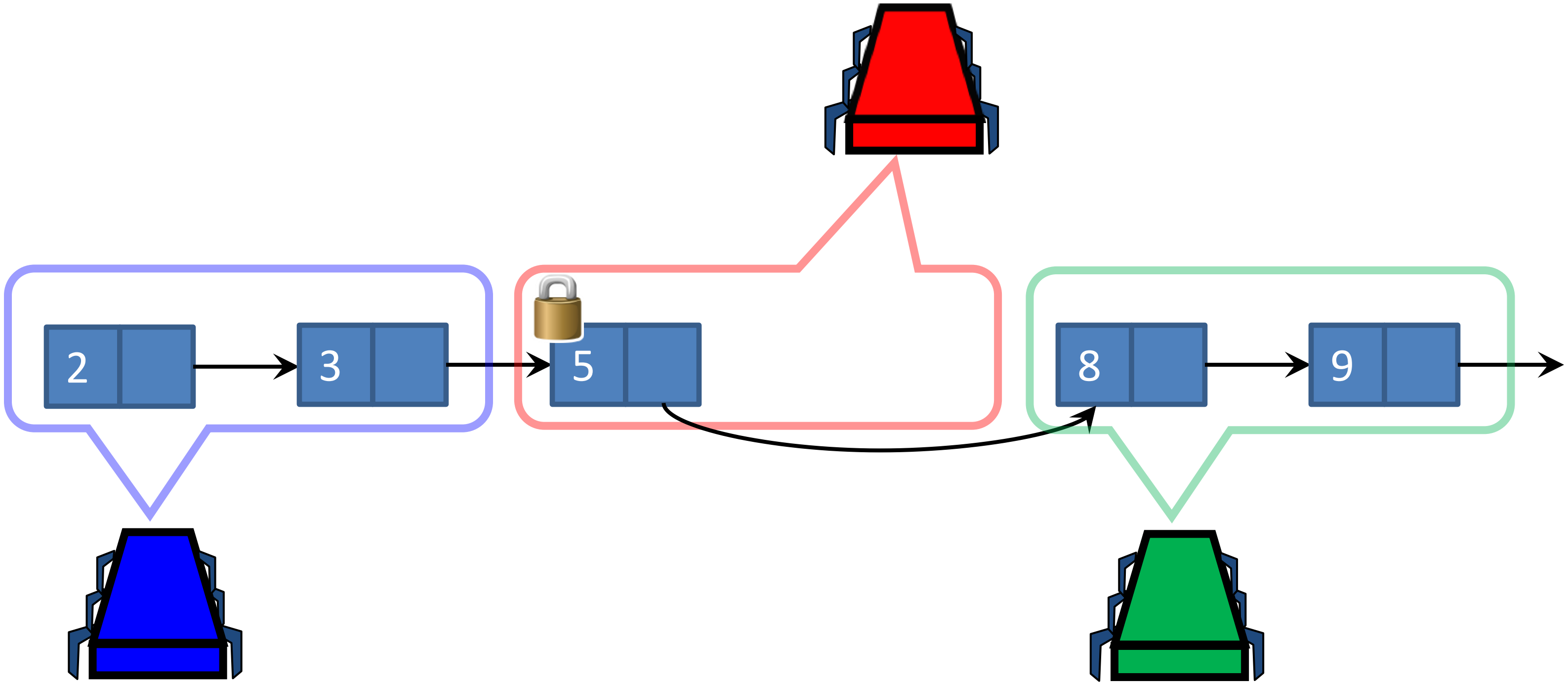
- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Example: Lock-Coupling List



- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Example: Lock-Coupling List



- There is one lock per node; threads acquire locks in a hand over hand fashion.
- If a node is locked, we can insert a node just after it.
- If two adjacent nodes are locked, we can remove the second.

Extensions of Separation Logic for Concurrent Programs

Extensions of Separation Logic for Concurrent Programs

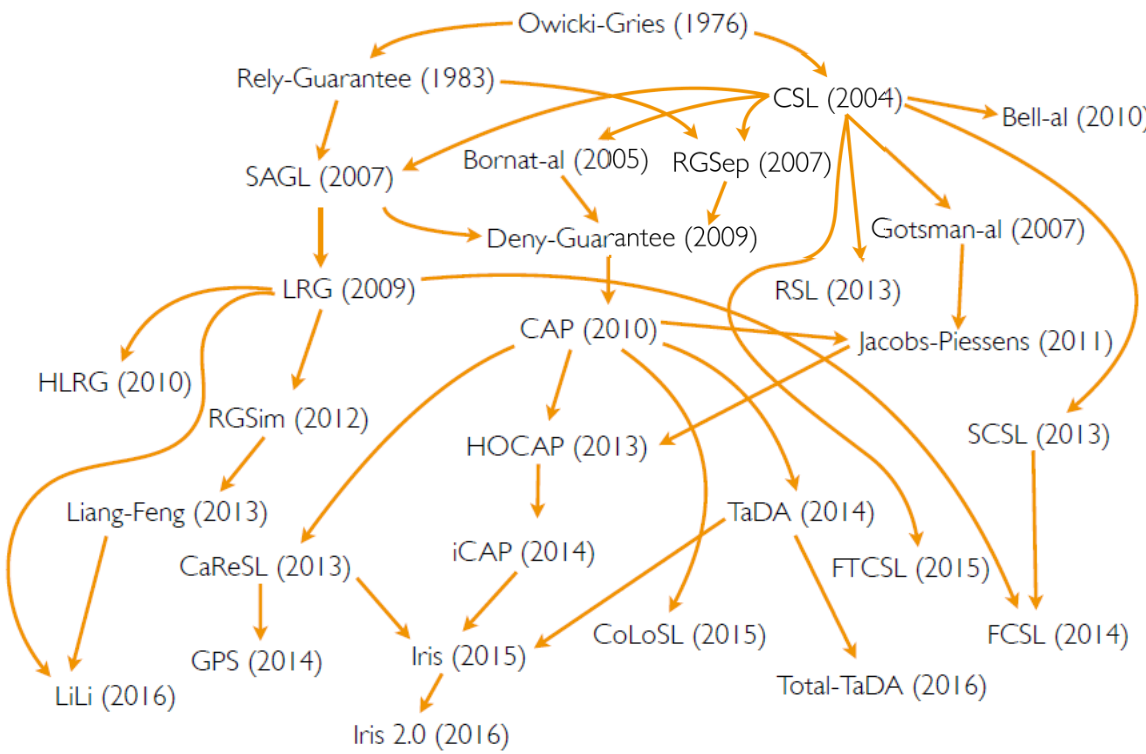
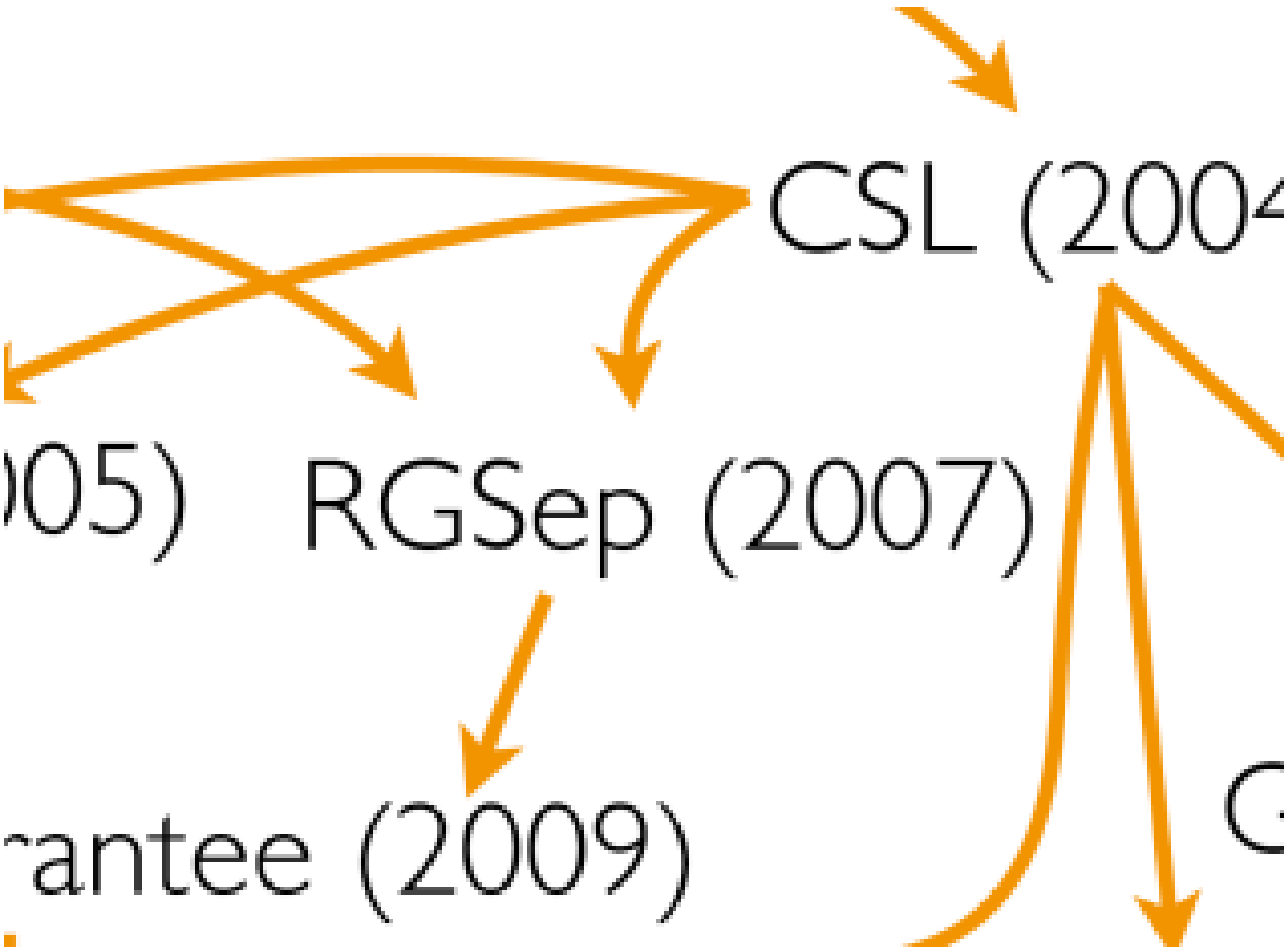


Fig. 1. CSL Family Tree (courtesy of Ilya Sergey)

Extensions of Separation Logic for Concurrent Programs



RGSep Primer

[courtesy of Viktor Vafeiadis]

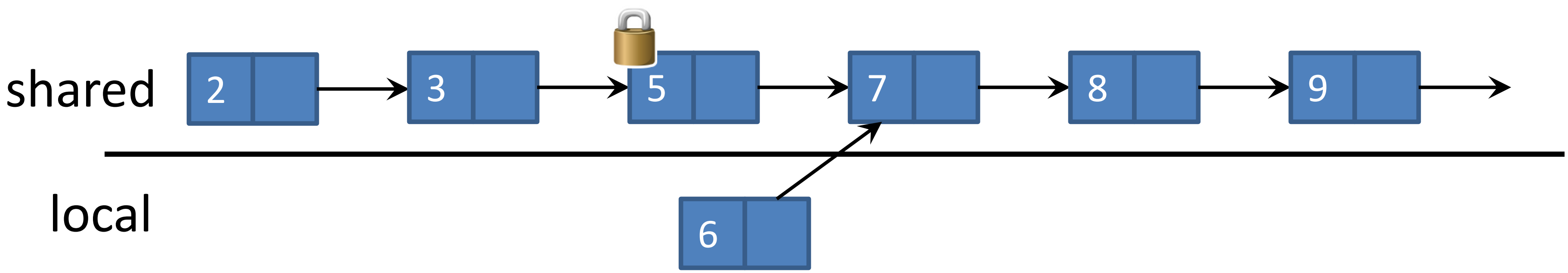
Program and Environment

- **Program:** the current thread being verified.
- **Environment:** all other threads of the system that execute in parallel with the thread being verified.
- **Interference:** The program interferes with the environment by modifying the shared state.

  Conversely, the environment interferes with the program by modifying the shared state.

Local & Shared State

- The total state is logically divided into two components:
  - ***Shared****:* accessible by all threads via synchronization
  - ***Local****:* accessible only by one thread, its owner



State of the lock-coupling list just before inserting a new node.

The node to be added is local because other threads cannot yet access it.

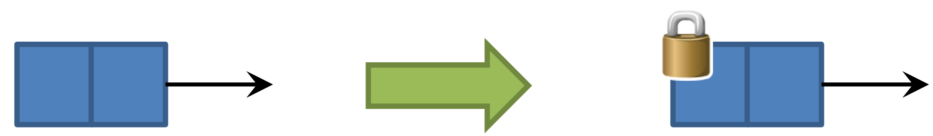
Program Specifications

- The specification of a program consists of two assertions (precondition & postcondition), and two sets of actions:
- **Rely:** Describes the interference that the program can tolerate from the environment; i.e. specifies how the environment can change the shared state.
- **Guarantee:** Describes the interference that the program imposes on its environment; i.e. specifies how the program can change the shared state.

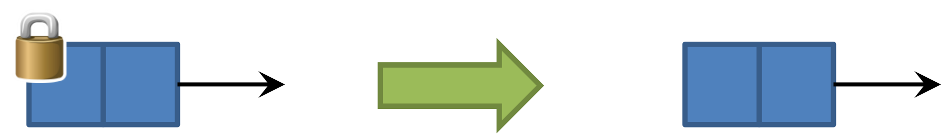
Rely/Guarantee Actions

Actions describe minimal atomic changes to the shared state.

Lock node



Unlock node

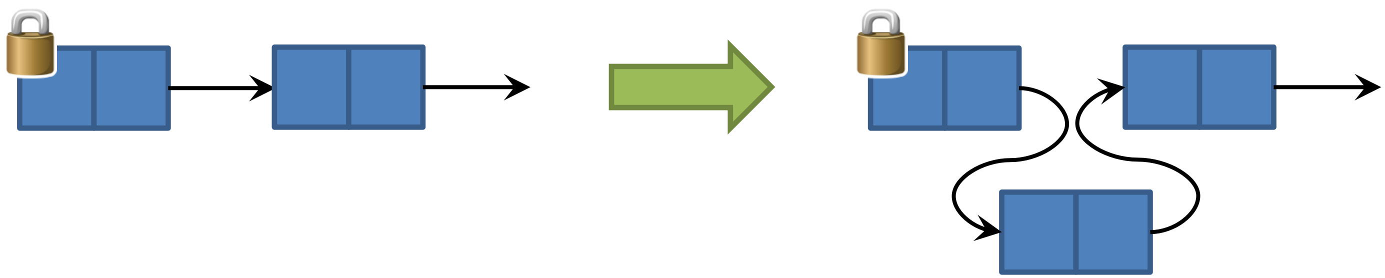


An action allows any part of the *shared state* that satisfies the LHS to be changed to a part satisfying the RHS, but the rest of the shared state must not be changed.

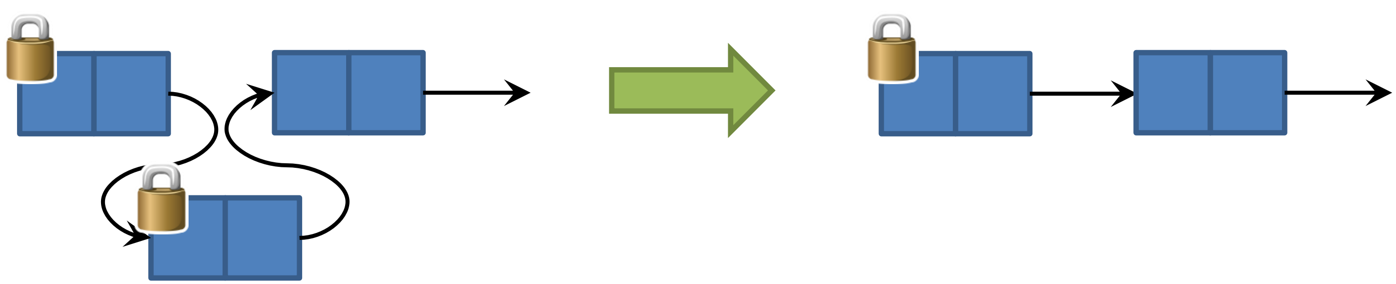
Rely/Guarantee Actions

Actions can adjust the boundary between local state and stared state.
This is also known as *transfer of ownership*.

Add node



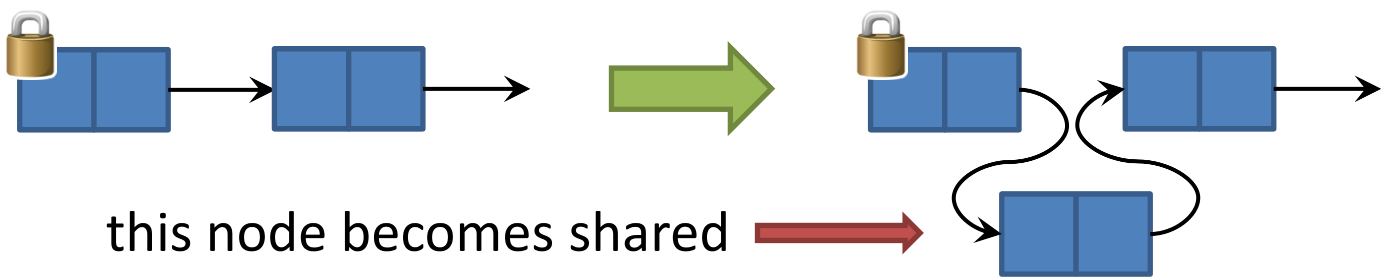
Delete node



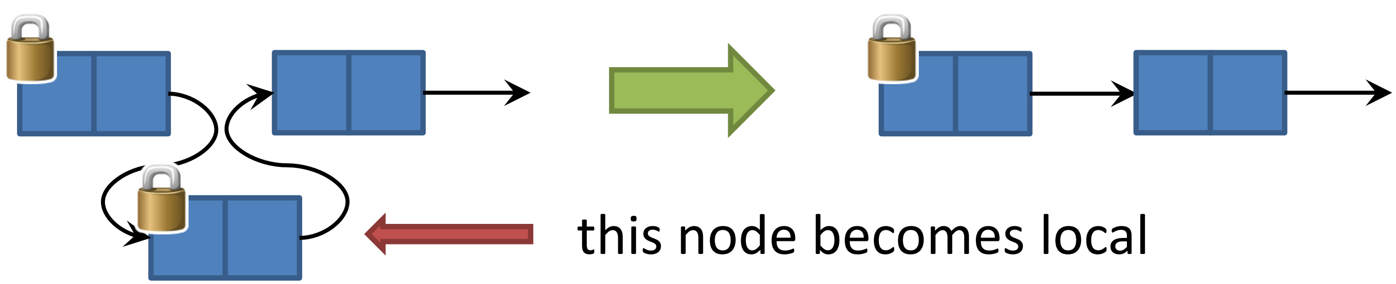
Rely/Guarantee Actions

Actions can adjust the boundary between local state and stared state.
This is also known as *transfer of ownership*.

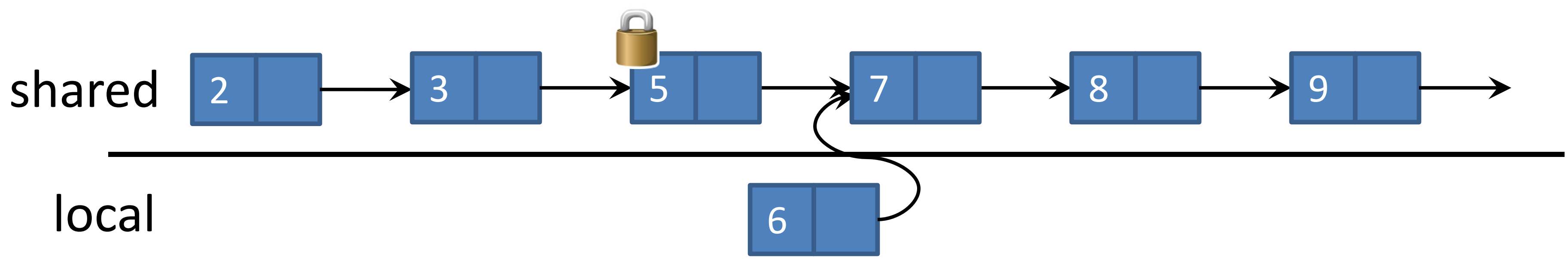
Add node



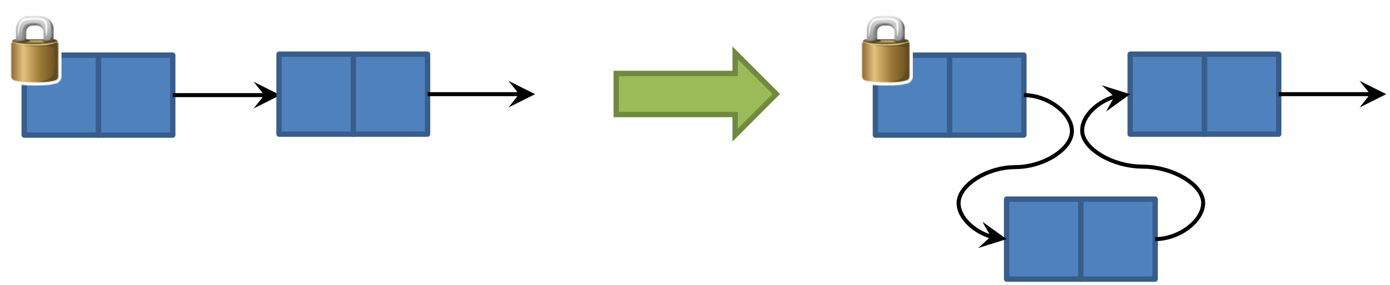
Delete node



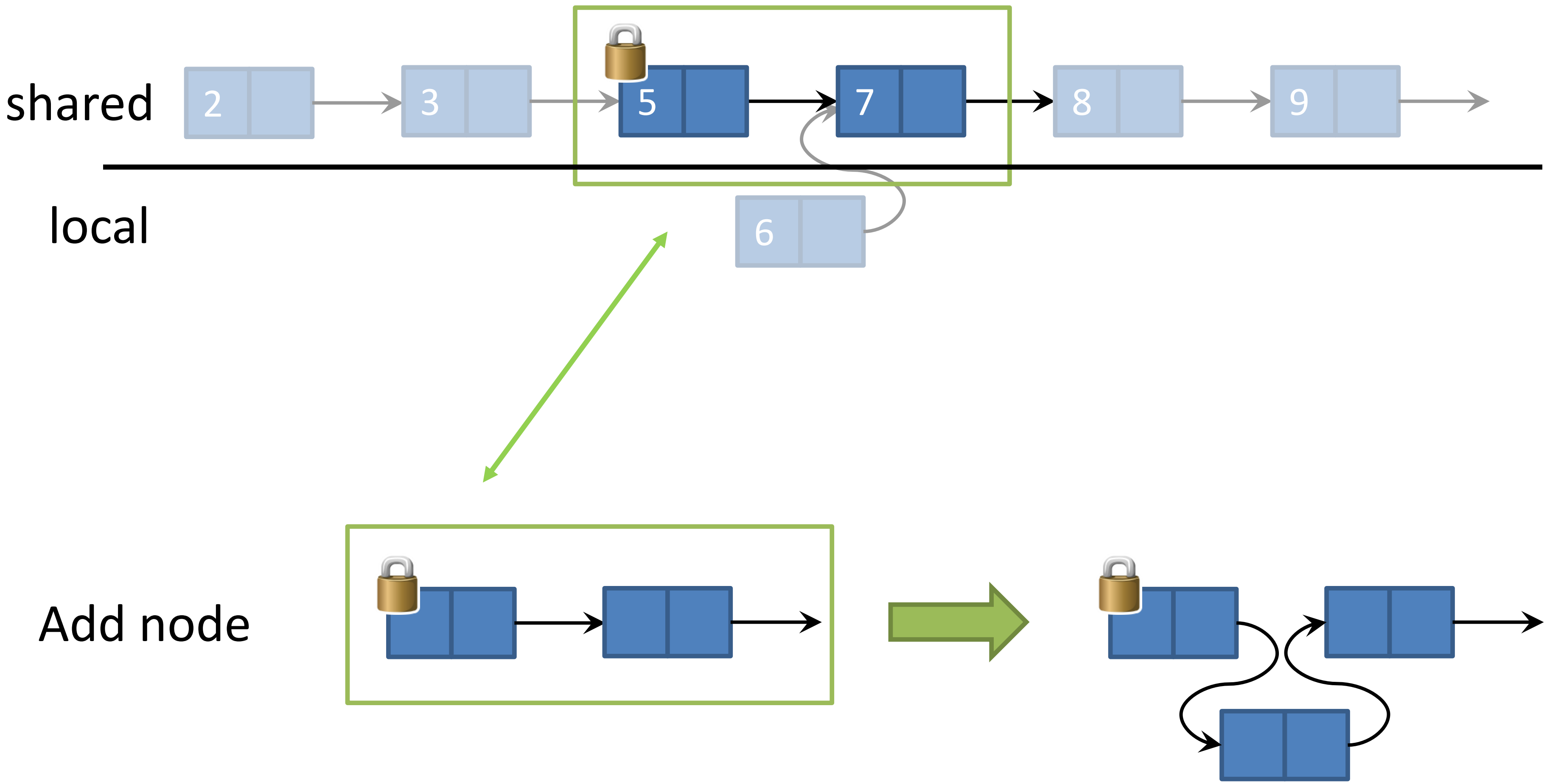
Rely/Guarantee Actions: Lock Coupling List



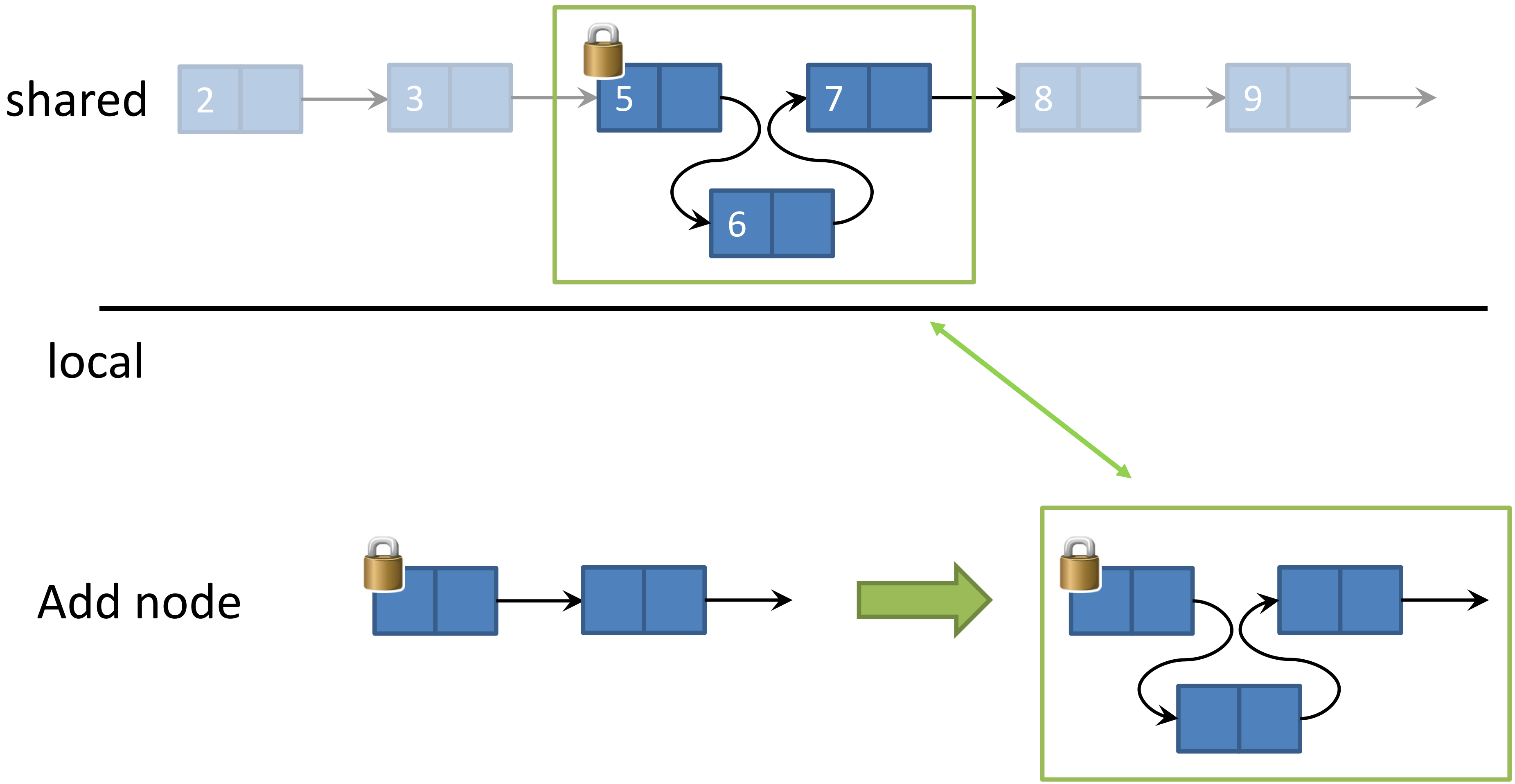
Add node



Rely/Guarantee Actions: Lock Coupling List

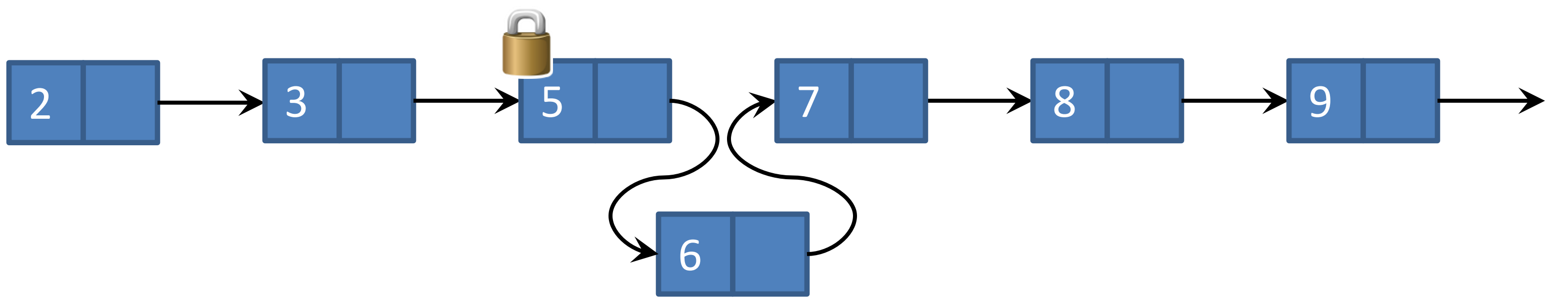


Rely/Guarantee Actions: Lock Coupling List



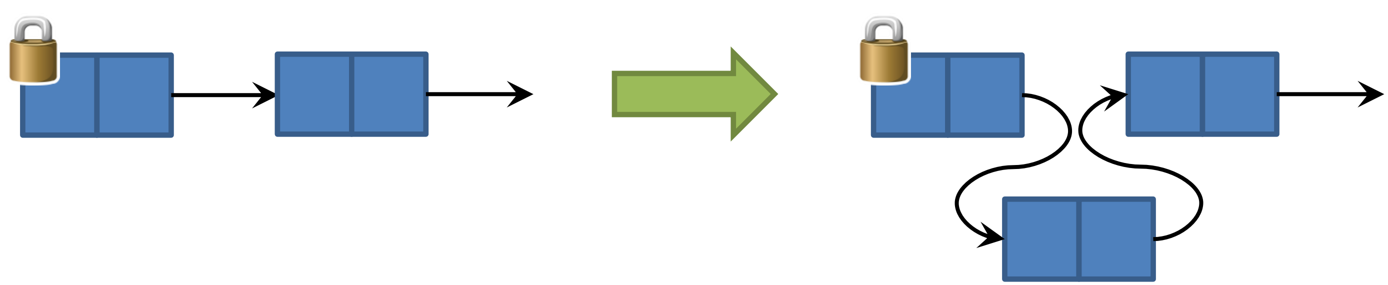
Rely/Guarantee Actions: Lock Coupling List

shared

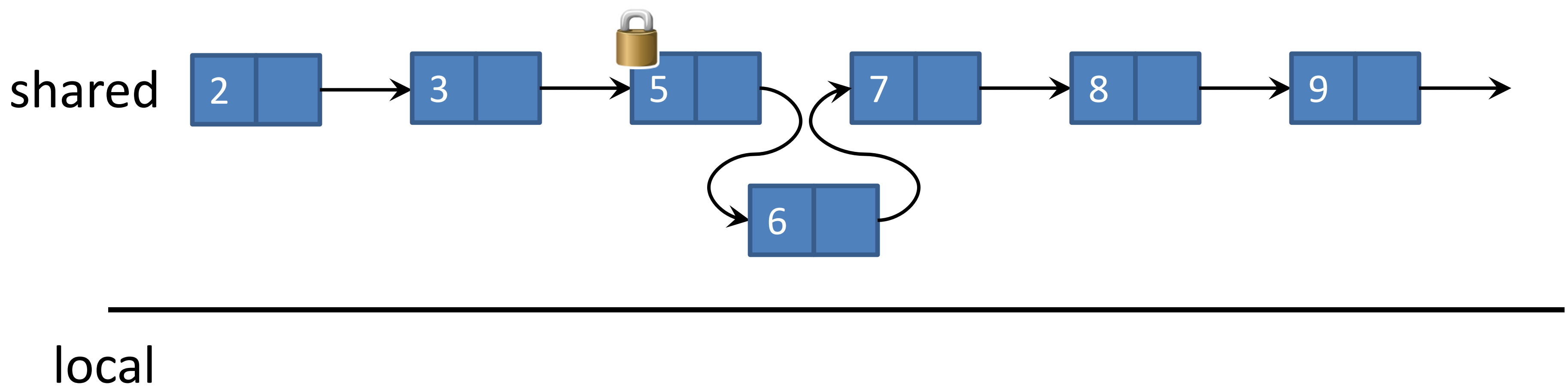


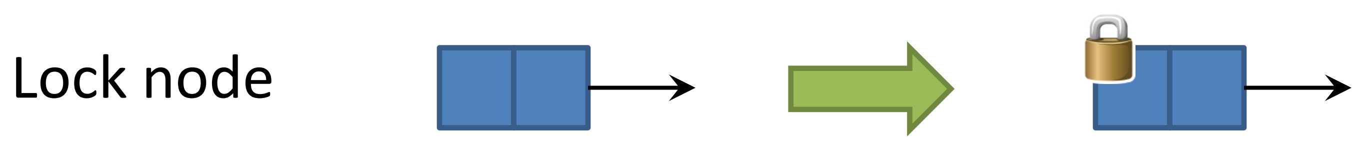
local

Add node

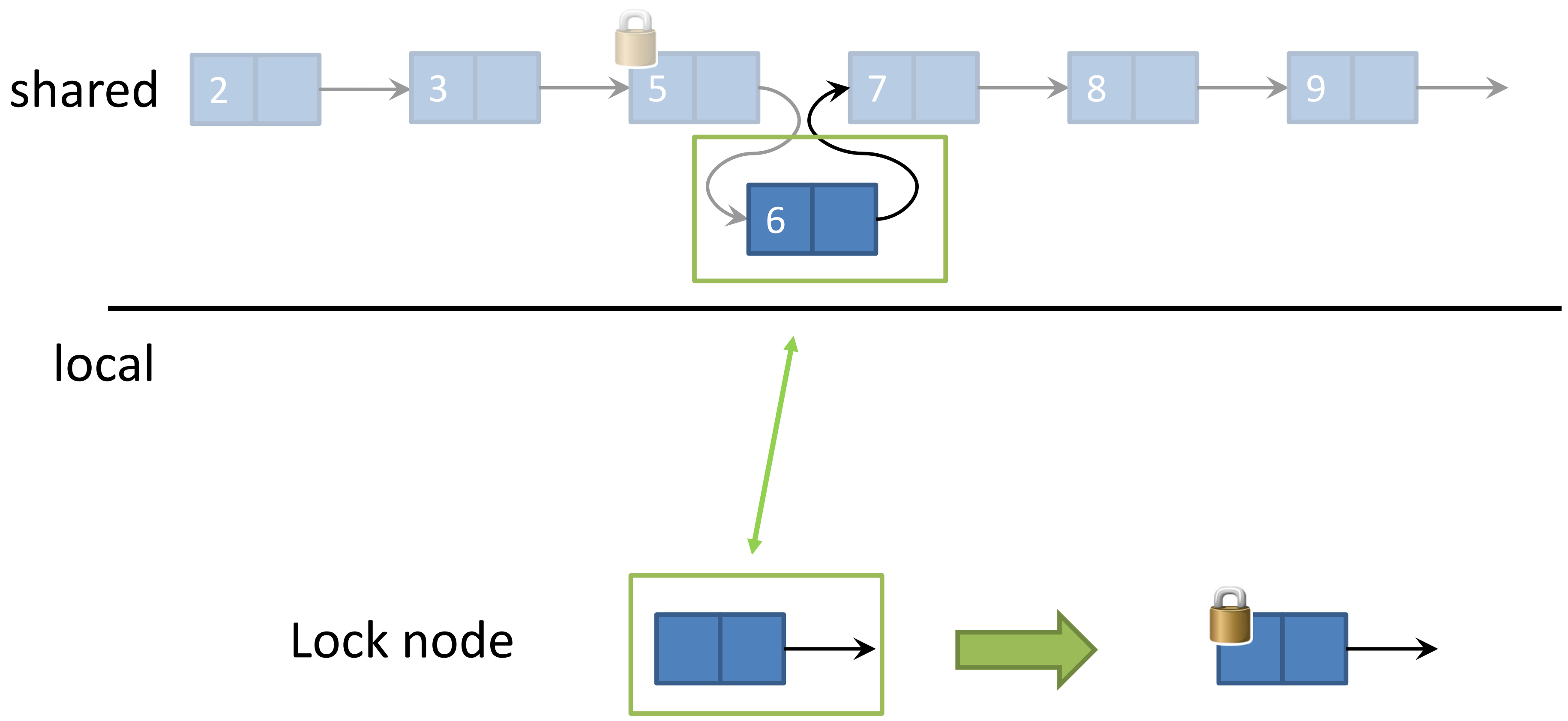


Rely/Guarantee Actions: Lock Coupling List

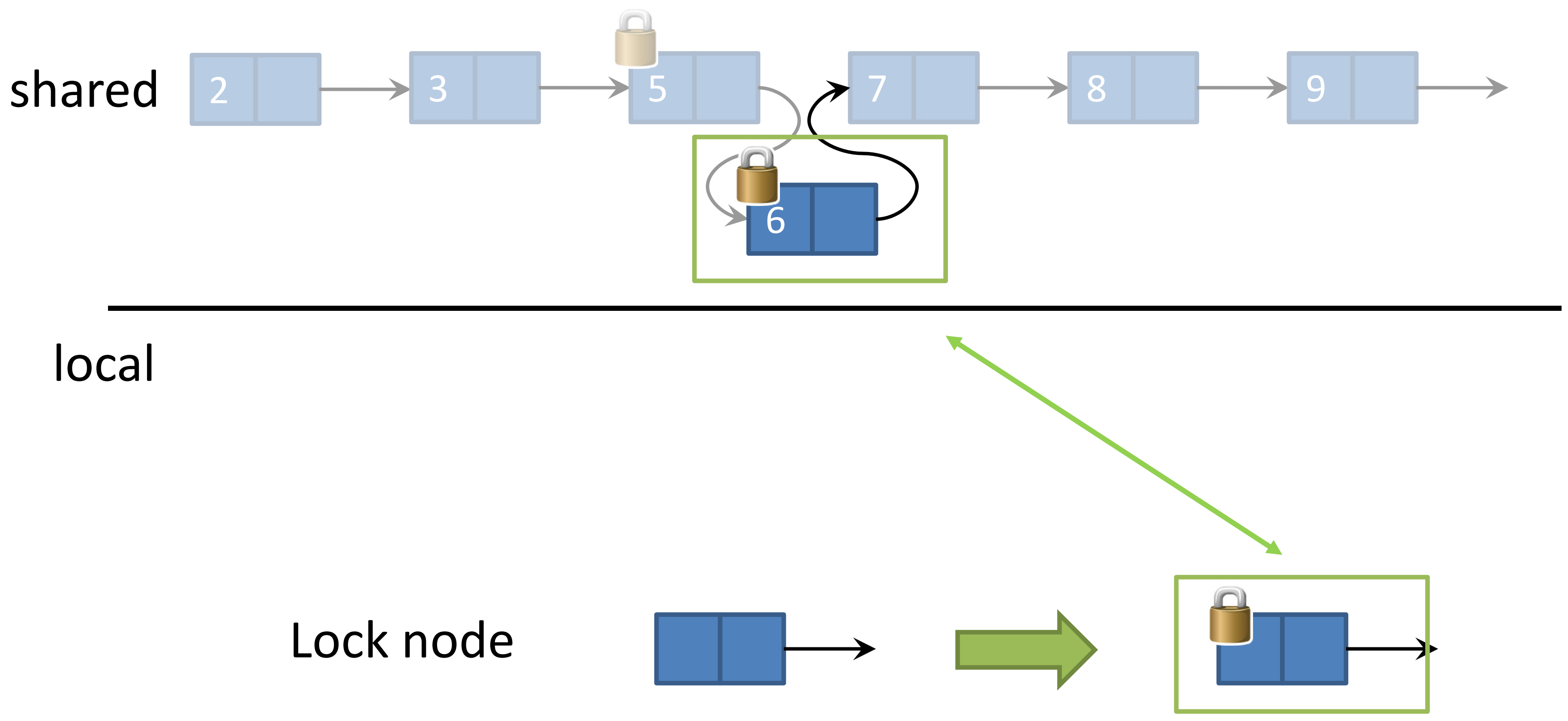




Rely/Guarantee Actions: Lock Coupling List

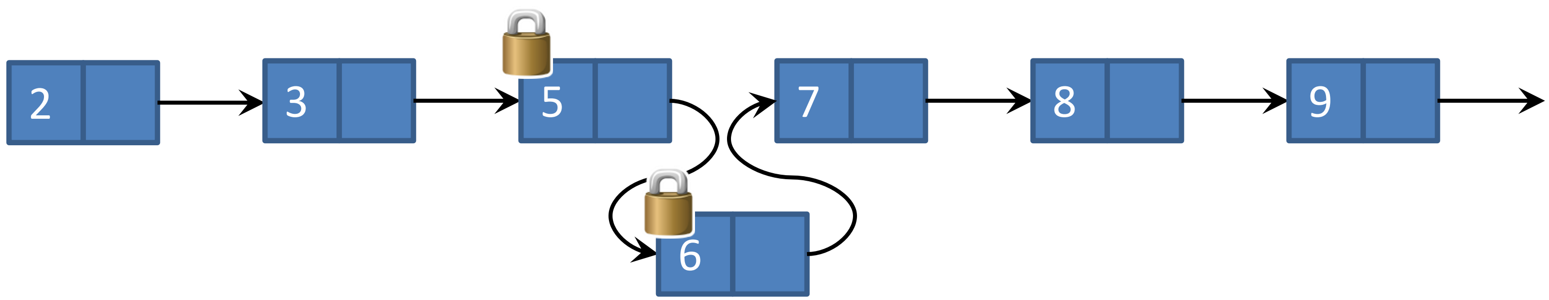


Rely/Guarantee Actions: Lock Coupling List



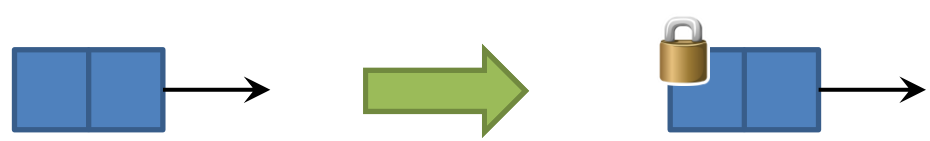
Rely/Guarantee Actions: Lock Coupling List

shared



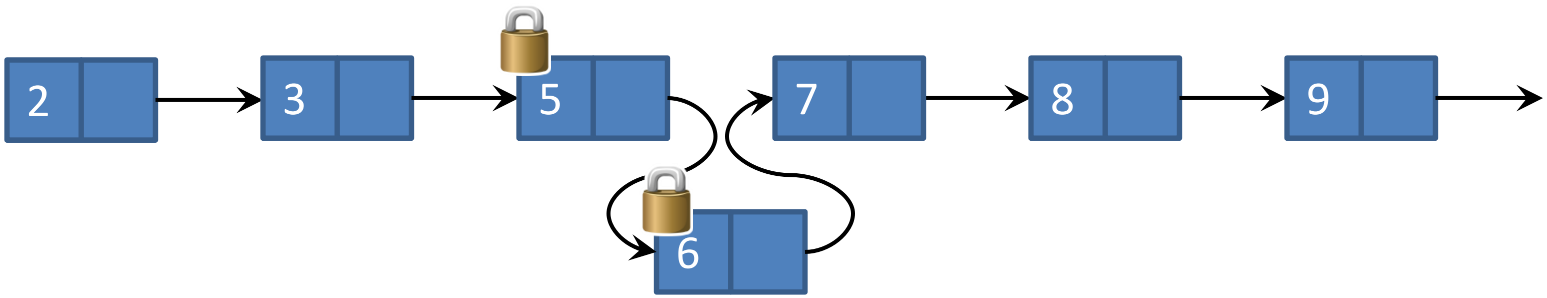
local

Lock node



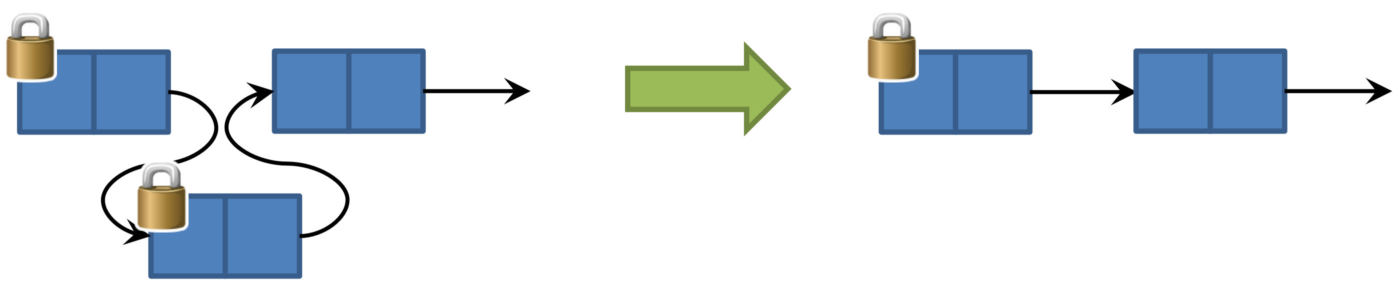
Rely/Guarantee Actions: Lock Coupling List

shared

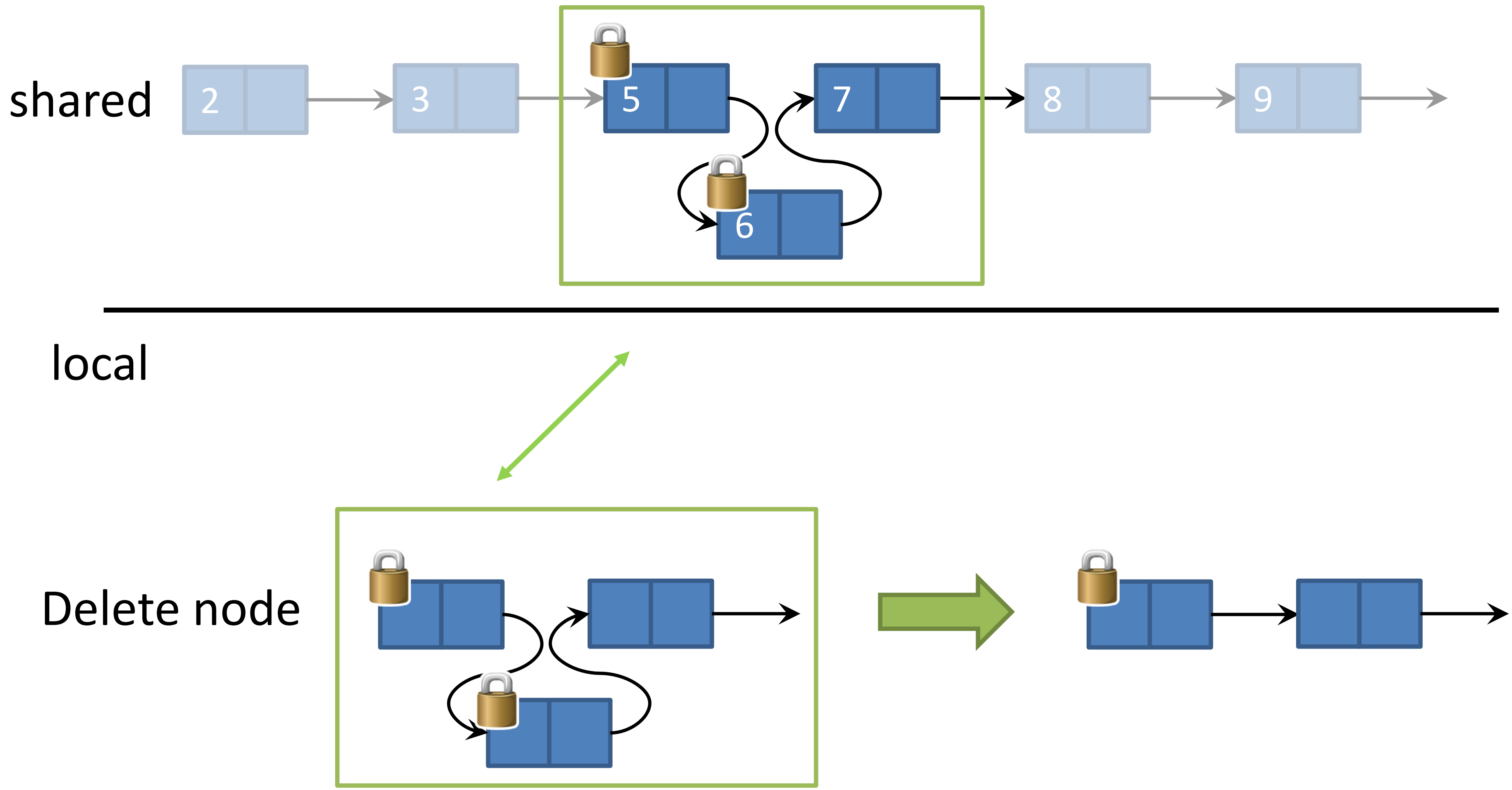


local

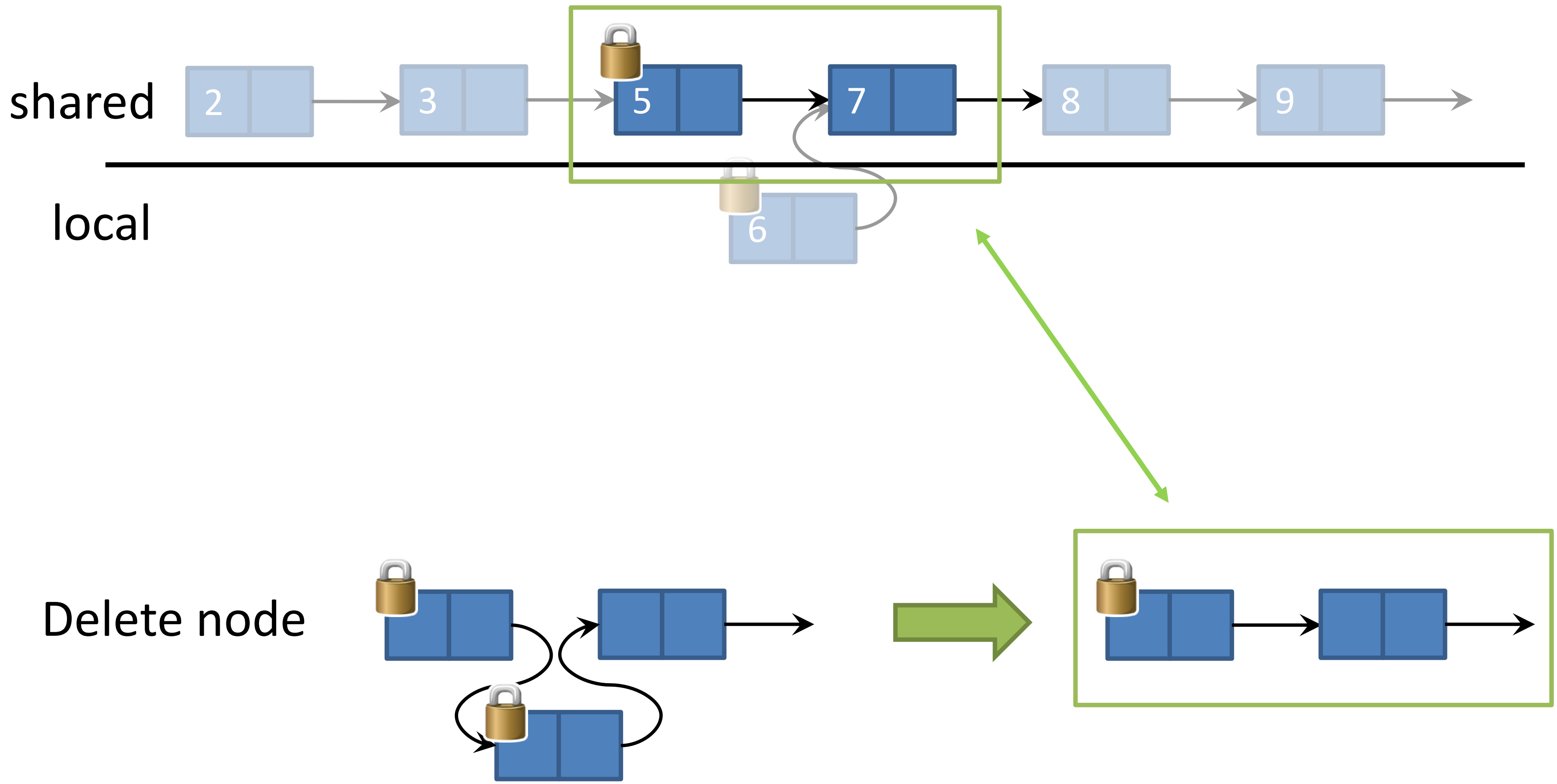
Delete node



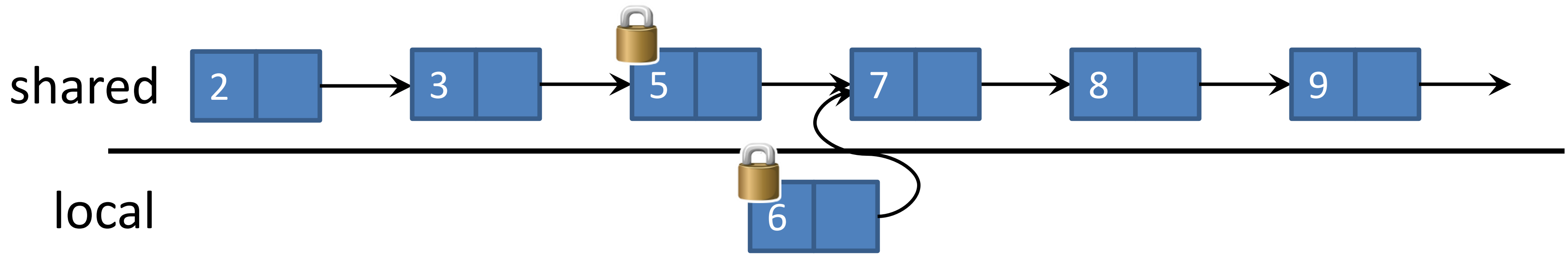
Rely/Guarantee Actions: Lock Coupling List



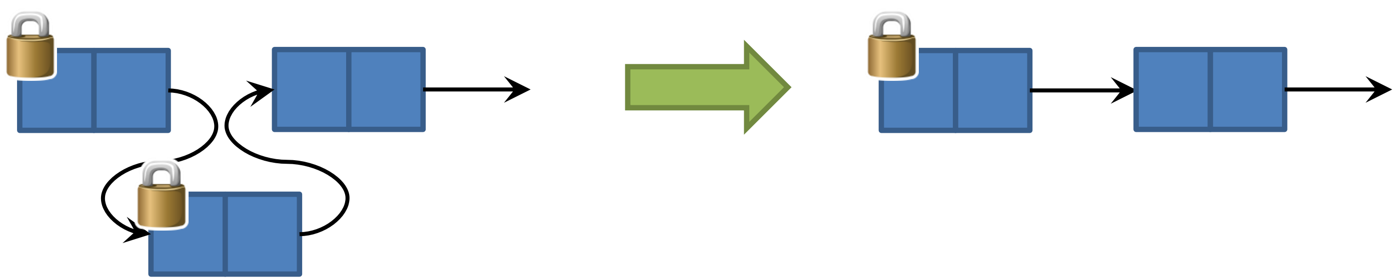
Rely/Guarantee Actions: Lock Coupling List



Rely/Guarantee Actions: Lock Coupling List



Delete node



Assertion Syntax

- Separation Logic

$$P, Q ::= e = e \mid e \neq e \mid e \mapsto (\mathbf{f}: \mathbf{e}) \mid P * Q \mid \ldots$$

- Extended Logic

$$p, q ::= P \mid \boxed{P} \mid p * q \mid \ldots$$

local

shared

Assertion Semantics

- $l, s \vDash P \quad \Leftrightarrow \quad l \vDash_{SL} P$
- $l, s \vDash \boxed{P} \quad \Leftrightarrow \quad s \vDash_{SL} P$ and $l = \{\}$
- $l, s \vDash p * q \quad \Leftrightarrow$ exists l_1, l_2 :
  $l = l_1 \bullet l_2$ and $l_1, s \vDash p$ and $l_2, s \vDash q$

Assertion Semantics

- $l, s \models P \quad \Leftrightarrow \quad l \models_{SL} P$
- $l, s \models \boxed{P} \quad \Leftrightarrow \quad s \models_{SL} P$ and $l = \{\}$
- $l, s \models p * q \quad \Leftrightarrow \quad$ exists l_1, l_2 :
  $l = l_1 \bullet l_2$ and $l_1, s \models p$ and $l_2, s \models q$

split local state

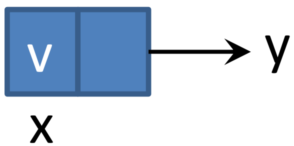
Assertion Semantics

- $l, s \vDash P \quad\Leftrightarrow\quad l \vDash_{SL} P$
- $l, s \vDash \boxed{P} \quad\Leftrightarrow\quad s \vDash_{SL} P \text{ and } l = \{\}$
- $l, s \vDash p * q \quad\Leftrightarrow\quad$ exists l_1, l_2 :
  $l = l_1 \bullet l_2$ and $l_1, s \vDash p$ and $l_2, s \vDash q$

share global state

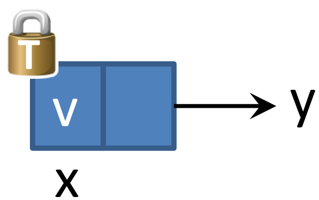
Assertions: Lock Coupling List

Unlocked node x holding value v and pointing to y



$x \mapsto (0, v, y)$

Node x holding value v and pointing to y, locked by thread T

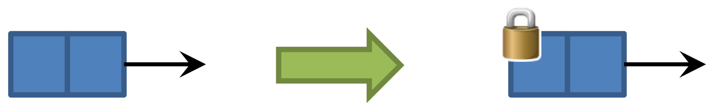


$x \mapsto (T, v, y)$

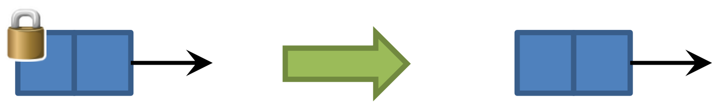
List segment from x to y of possibly locked nodes

`lseg`(x, y)

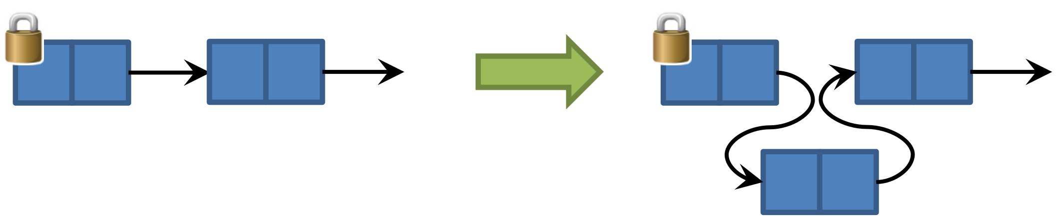
Rely/Guarantee Actions: Lock Coupling List



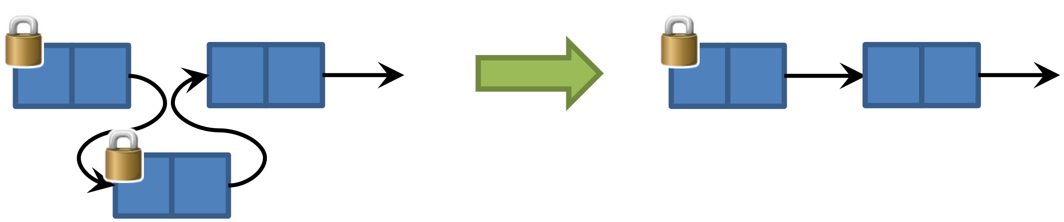
$x \mapsto (0, v, y) \rightarrow x \mapsto (T, v, y)$



$x \mapsto (T, v, y) \rightarrow x \mapsto (0, v, y)$



$x \mapsto (T, v, y) \rightarrow x \mapsto (T, v, z) * z \mapsto (0, w, y)$



$x \mapsto (T, v, z) * z \mapsto (T, w, y) \rightarrow x \mapsto (T, v, y)$

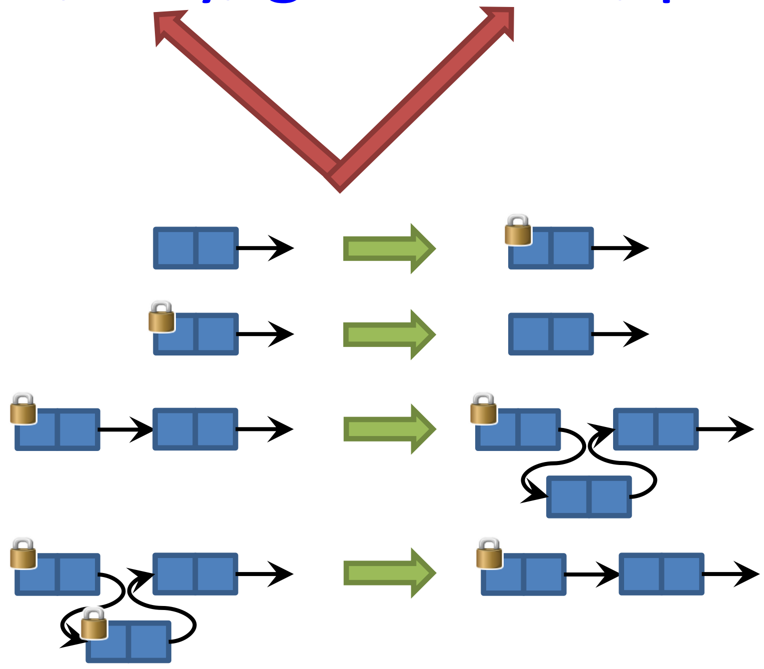
Programs: Syntax

- Basic commands c:
  - noop: **skip**
  - guard: **assume**(b)
  - heap write: [x] := y
  - heap read: x := [y]
  - allocation: x := **new**()
  - deallocation: **free**(x)
  - …
- Commands $C \in$ Com:
  - basic commands: c
  - seq. composition: C_1; C_2
  - nondet. choice: $C_1 + C_2$
  - looping: C*
  - atomic com.: **atomic** C
  - par. composition: $C_1 \mid C_2$

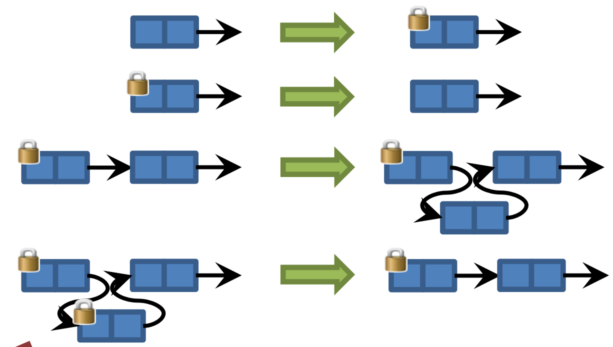
Rely/Guarantee Judgements

$\vdash$ C **sat** (p, R, G, q)

(precondition, rely, guarantee, postcondition)



Parallel Composition Rule



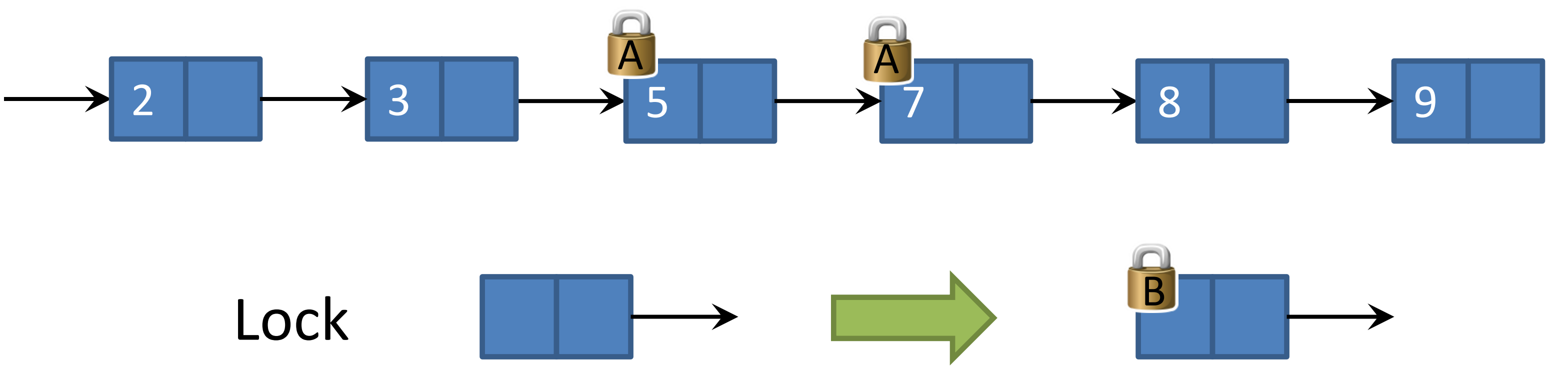
$$\vdash C_1 \ \mathbf{sat}\ (p_1, R \cup G_2, G_1, q_1)$$

$$\vdash C_2 \ \mathbf{sat}\ (p_2, R \cup G_1, G_2, q_2)$$

$$\overline{\vdash (C_1 \mid C_2)\ \mathbf{sat}\ (p_1 * p_2, R, G_1 \cup G_2, q_1 * q_2)}$$

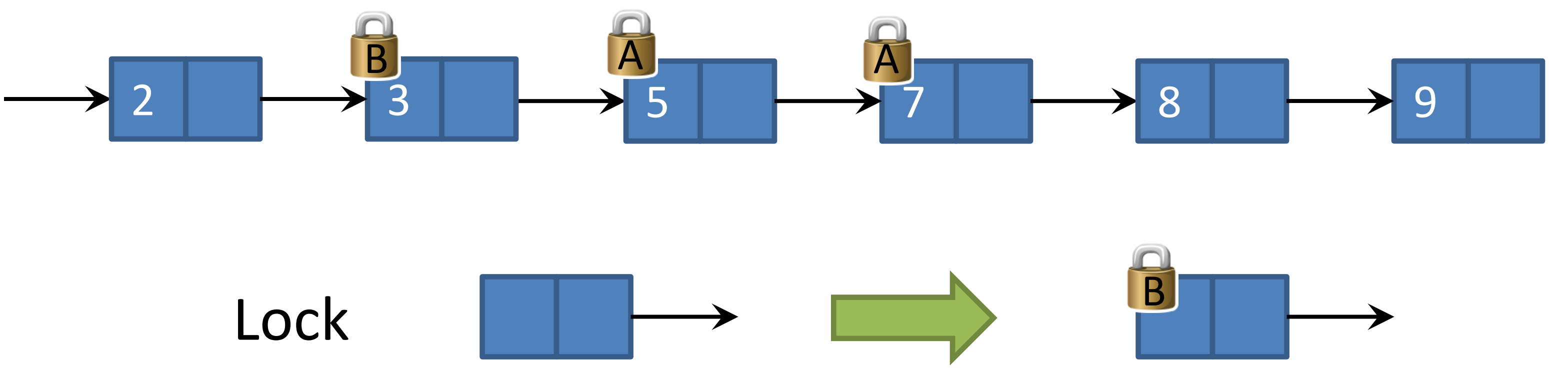
Stability

- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:



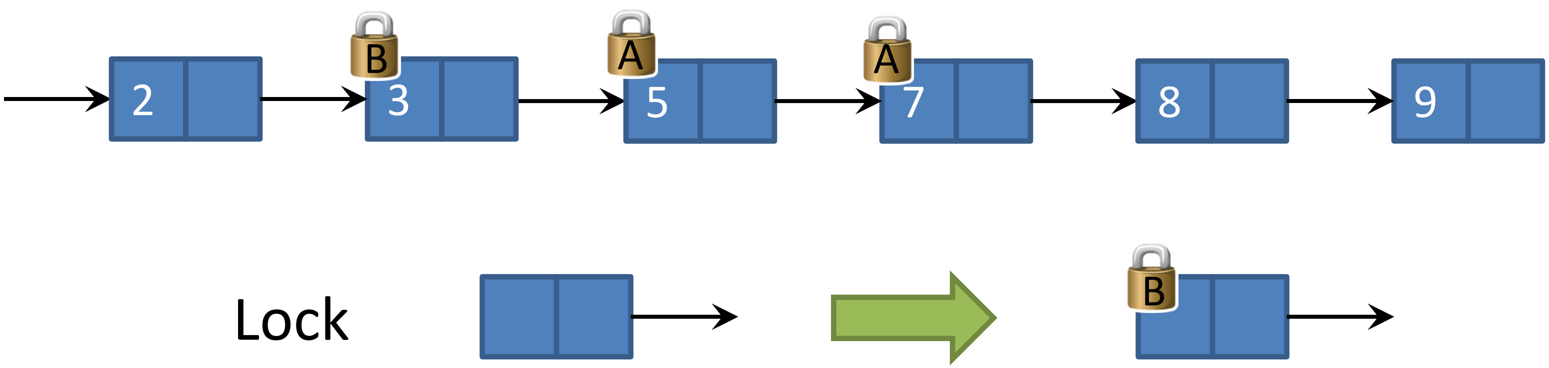
Stability

- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:



Stability

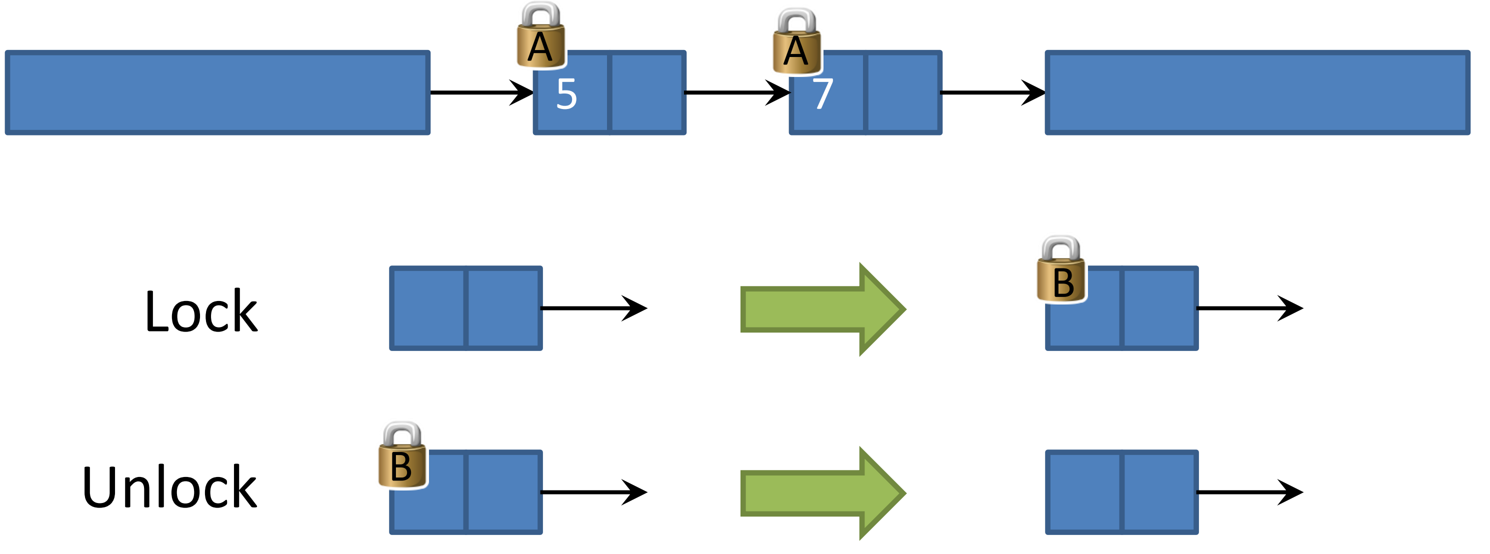
- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:



not stable!

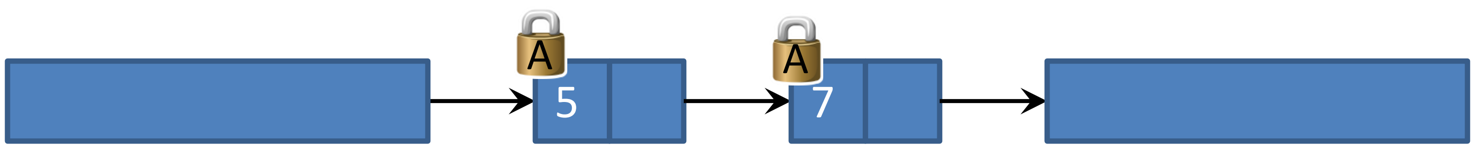
Stability

- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:

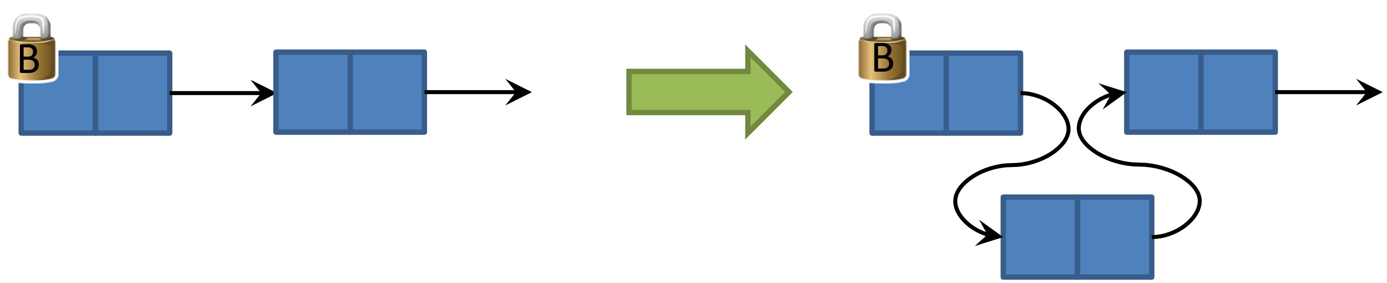


Stability

- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:

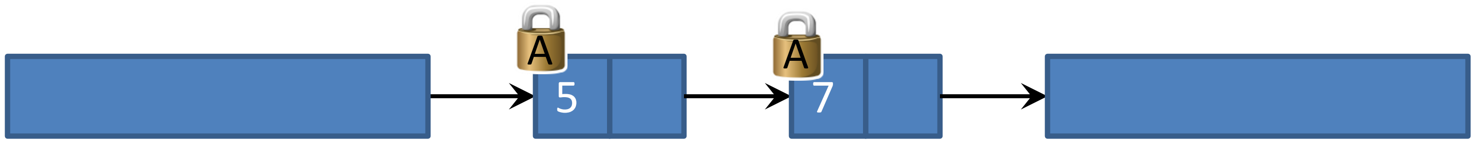


Add

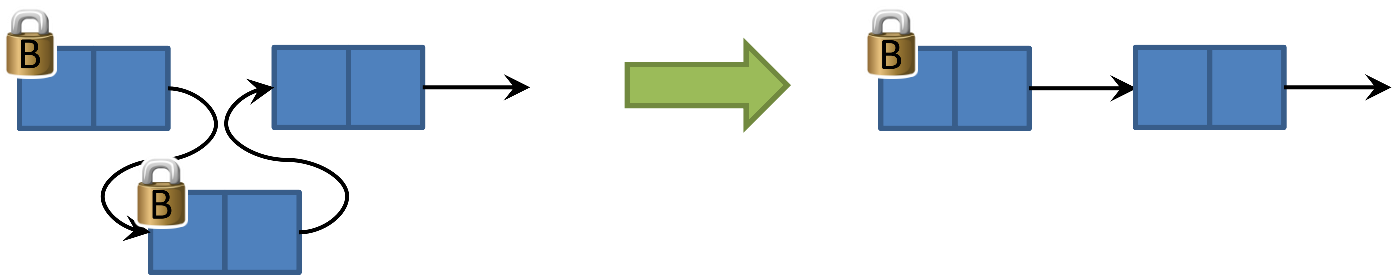


Stability

- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:

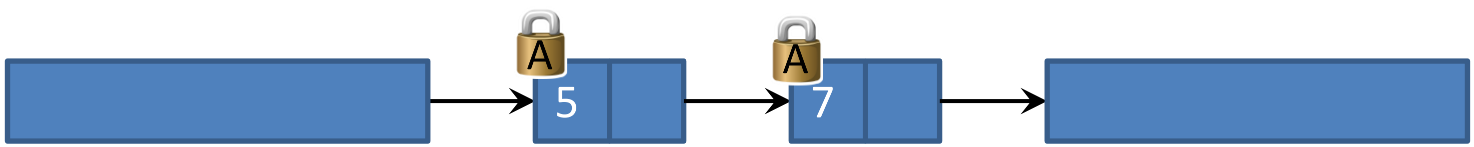


Delete

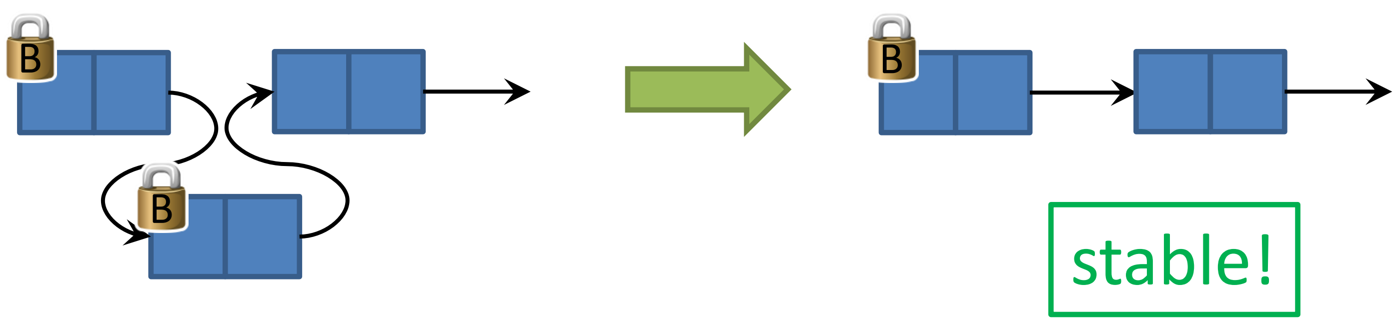


Stability

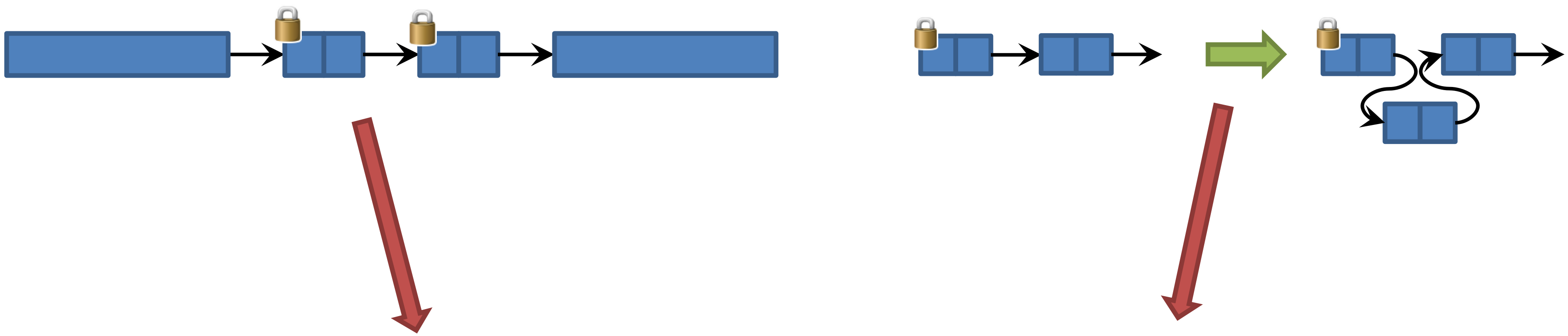
- An assertion is *stable* iff it is preserved under interference by other threads.
- **Example**:



Delete



Stability (Formally)



$\boxed{S}$ stable under $P \rightarrow Q$

iff

$(P \mathbin{-\!\circledast} S) * Q \models S$

where $P \mathbin{-\!\circledast} S \;:=\; \neg\,(\neg\, P \mathbin{-\!*} \neg\, S)$

Atomic Commands

$$\frac{\vdash \{\ P\ \}\ \mathsf{C}\ \{\ Q\ \}}{\vdash (\mathtt{atomic}\ \mathsf{C})\ \mathbf{sat}\ (P, \emptyset, \emptyset, Q)}$$

Atomic Commands

reduction to sequential SL

$$\frac{\vdash \{\, P \,\}\; C \;\{\, Q \,\}}{\vdash (\mathbf{atomic}\; C)\; \mathbf{sat}\; (P, \emptyset, \emptyset, Q)}$$

only local state

Atomic Commands

$$\frac{\vdash \{\, P \,\}\ C\ \{\, Q \,\}}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P, \emptyset, \emptyset, Q)}$$

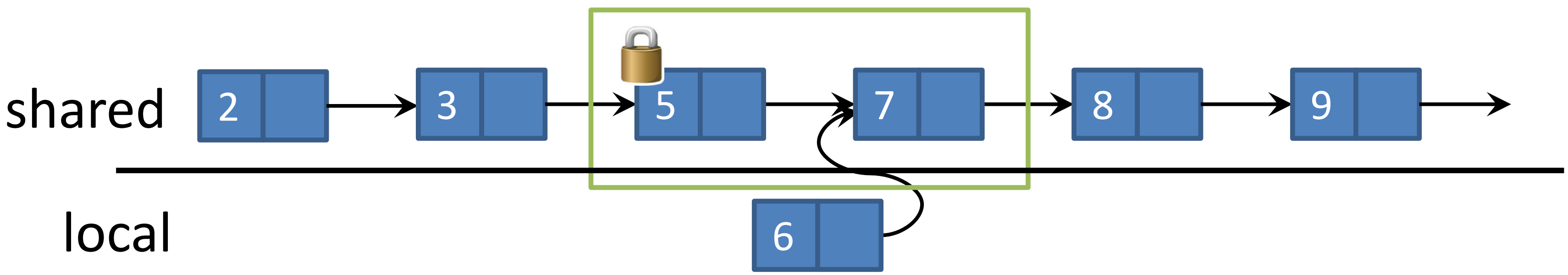
$$\frac{p, q \text{ stable under } R \qquad \vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (p, \emptyset, G, q)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (p, R, G, q)}$$

Atomic Commands

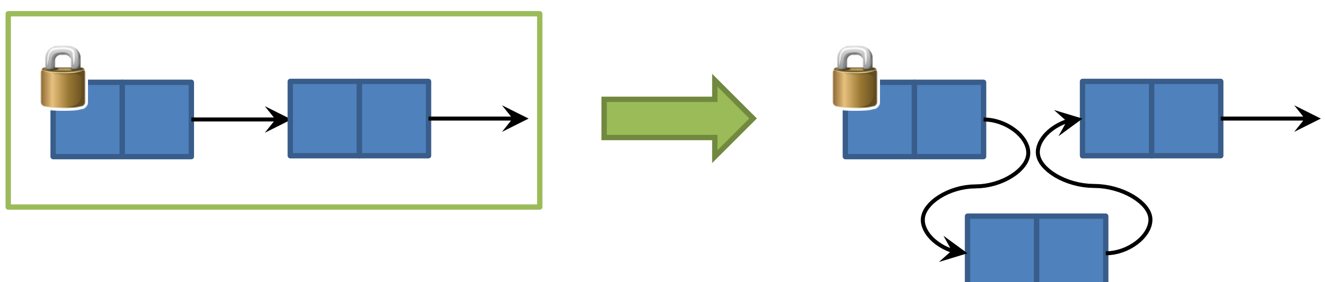
$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \to Q_2 \in G \qquad \vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$

Atomic Commands

$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \rightarrow Q_2 \in G \qquad \vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$

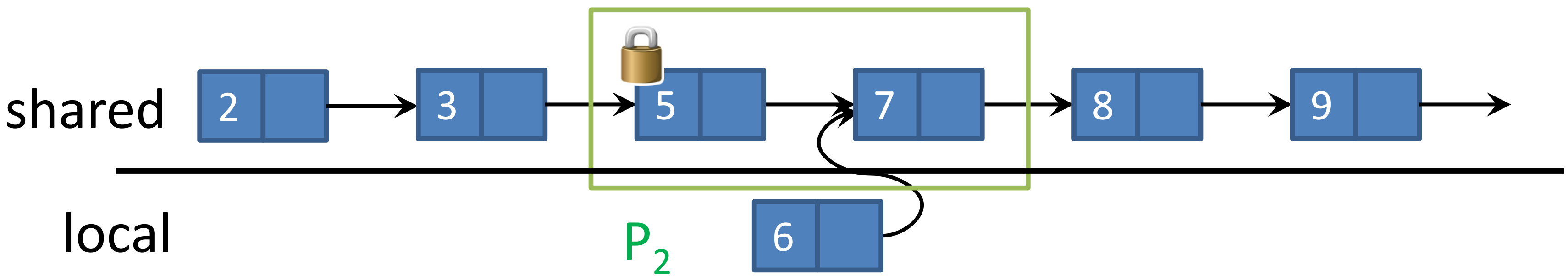


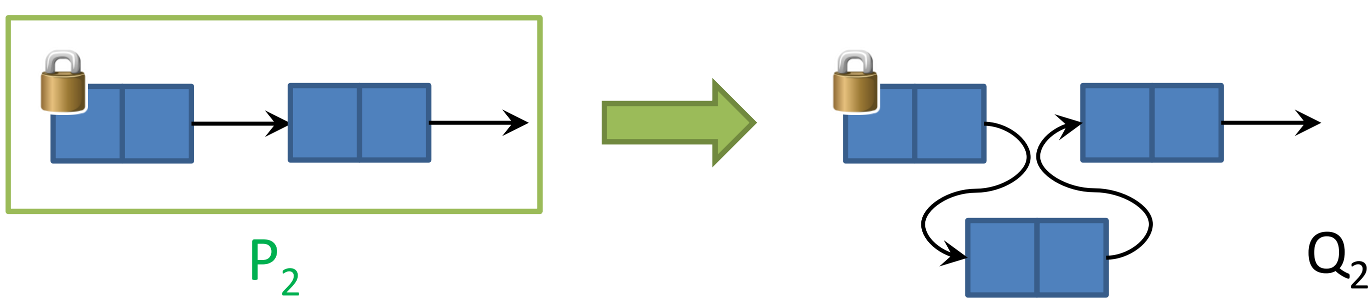
Add node



Atomic Commands

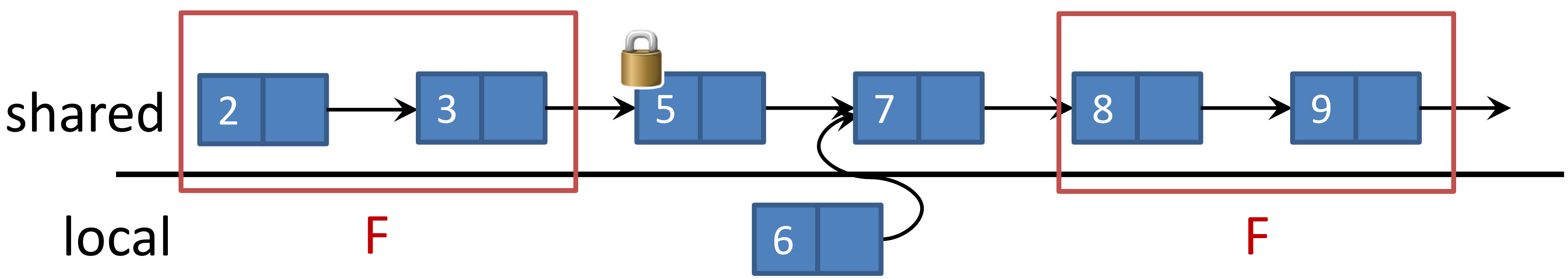
$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \to Q_2 \in G \qquad \vdash (\textbf{atomic}\ C)\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\textbf{atomic}\ C)\ \textbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$

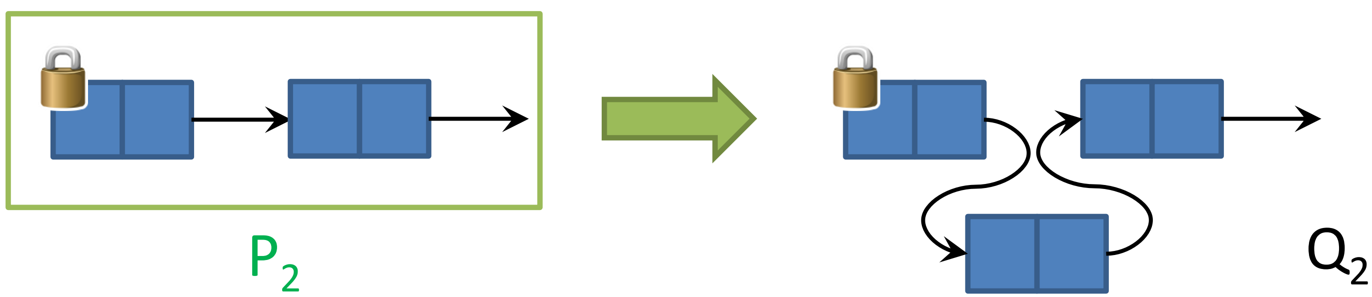




Atomic Commands

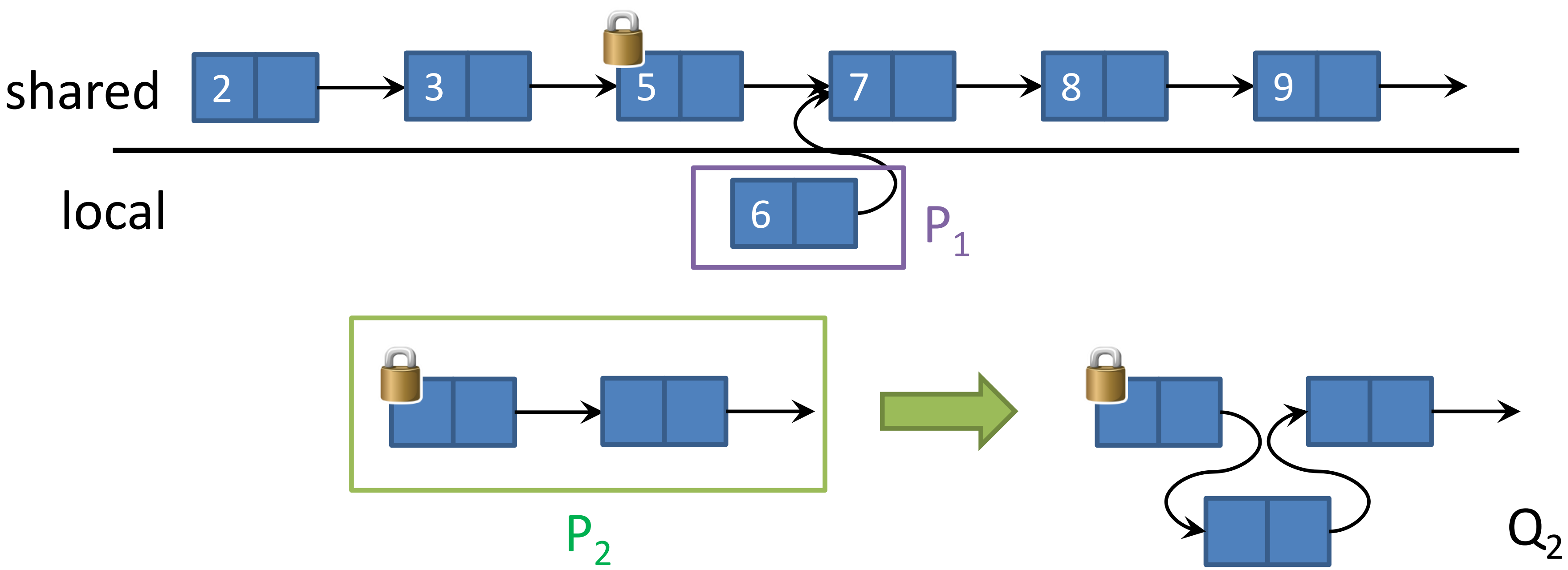
$$\frac{\begin{array}{c} P_2, Q_2 \text{ precise} \qquad P_2 \to Q_2 \in G \\ \vdash (\mathbf{atomic}\ C)\ \mathbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2) \end{array}}{\vdash (\mathbf{atomic}\ C)\ \mathbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$





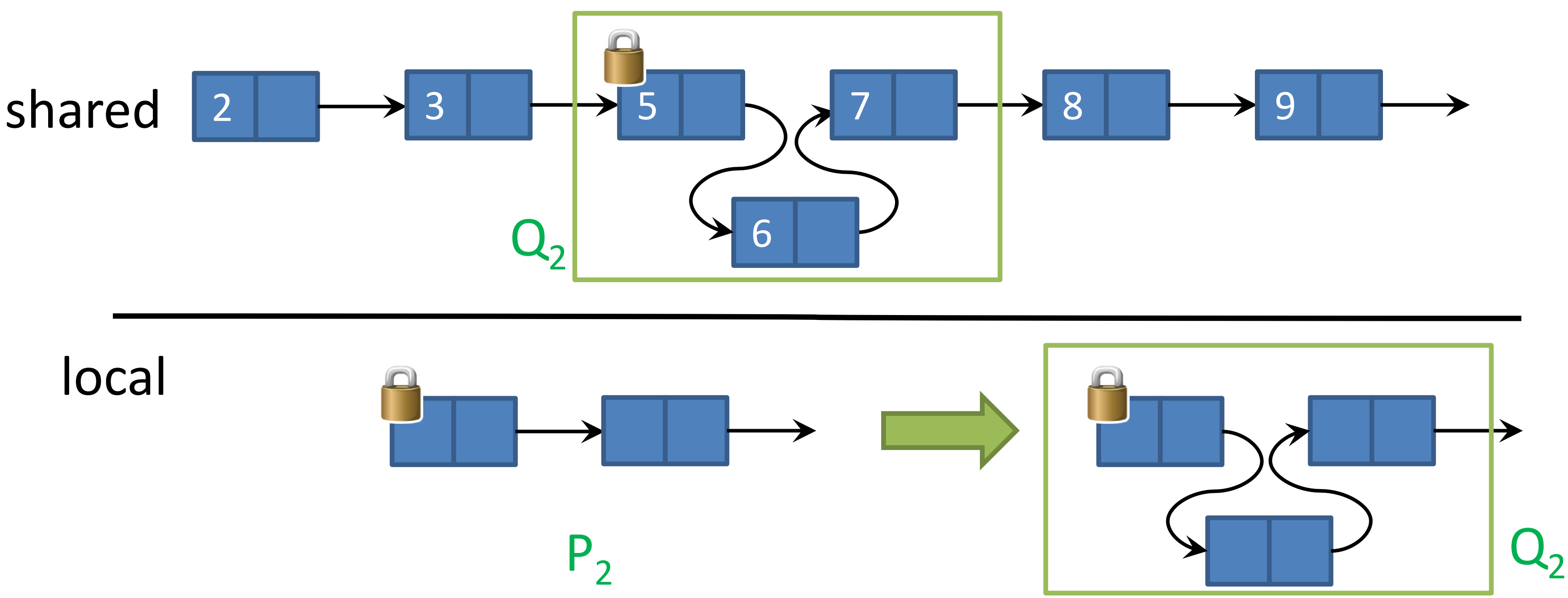
Atomic Commands

$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \to Q_2 \in G \qquad \vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$



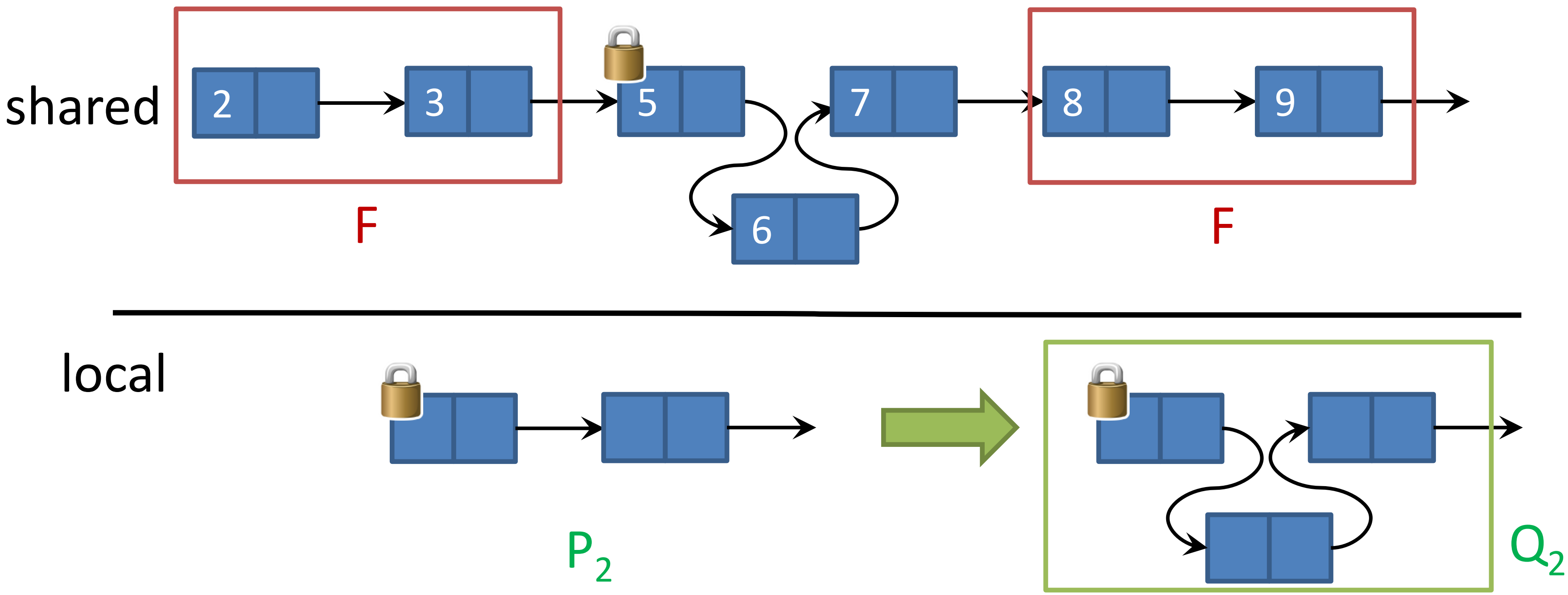
Atomic Commands

$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \rightarrow Q_2 \in G \qquad \vdash (\mathtt{atomic}\ C)\ \mathbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\mathtt{atomic}\ C)\ \mathbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$



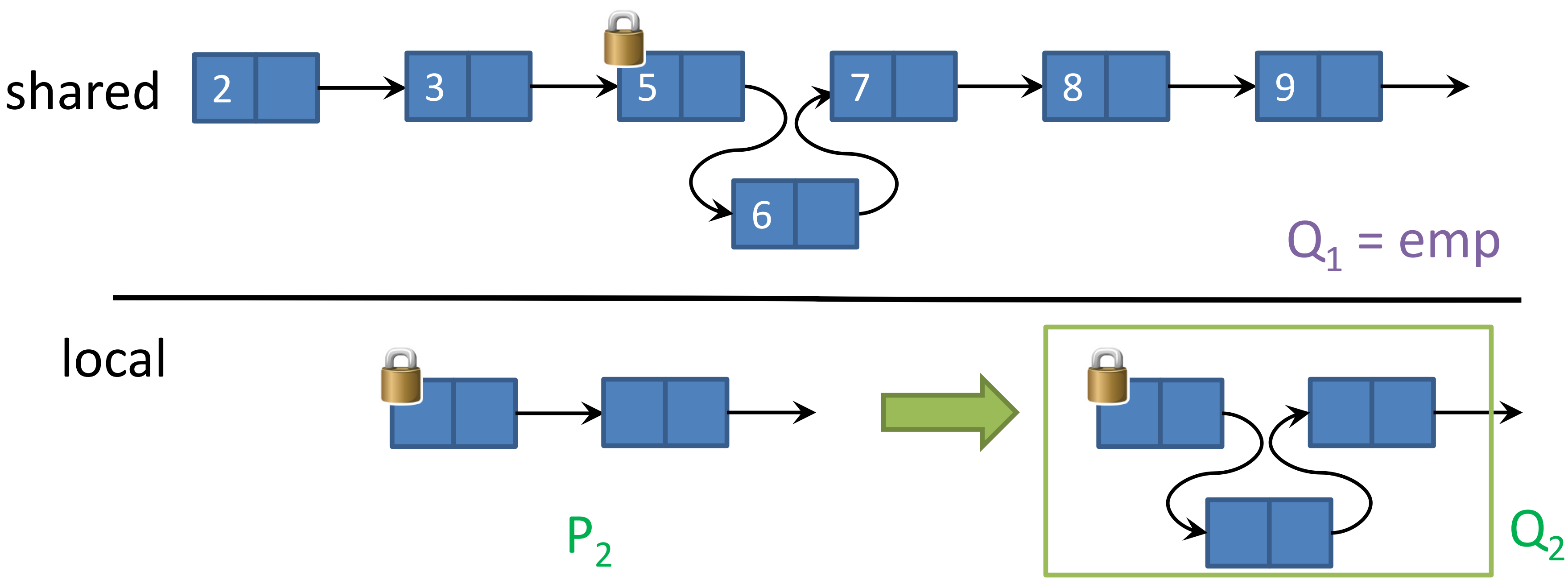
Atomic Commands

$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \rightarrow Q_2 \in G \qquad \vdash (\textbf{atomic}\ C)\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\textbf{atomic}\ C)\ \textbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$

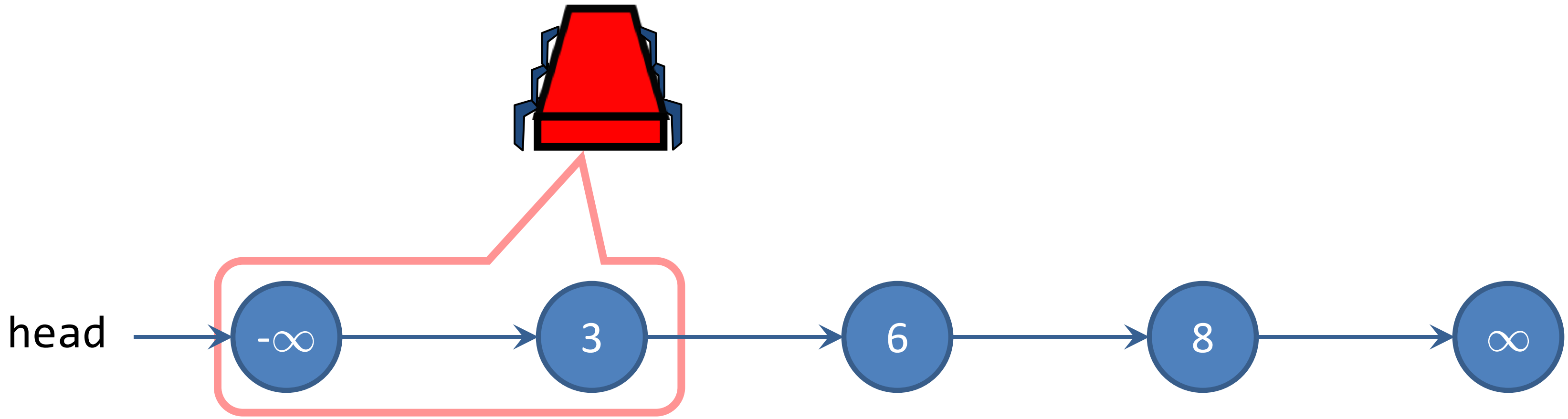


Atomic Commands

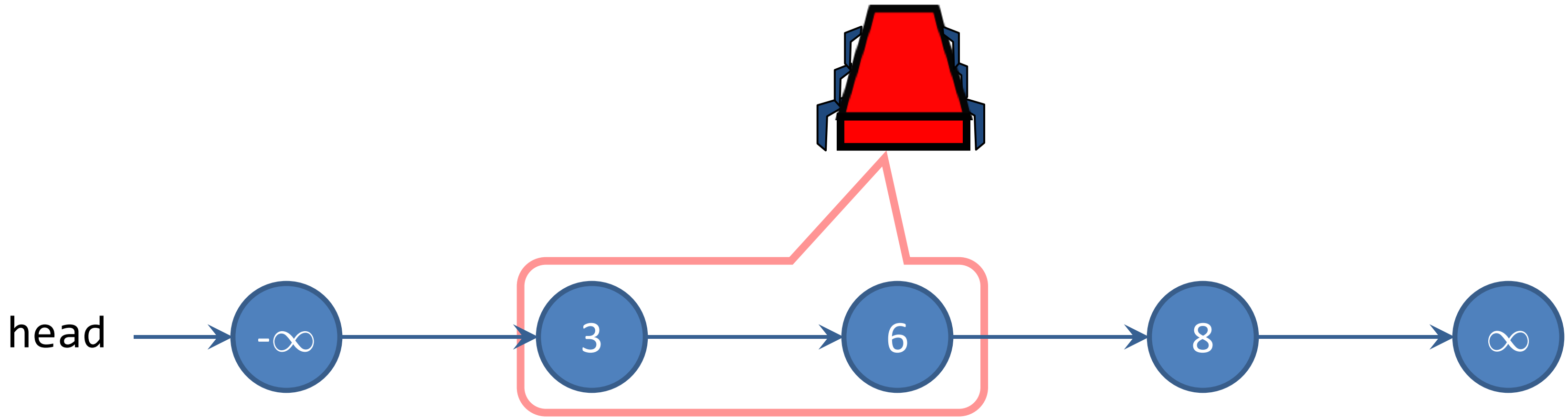
$$\frac{P_2, Q_2 \text{ precise} \qquad P_2 \rightarrow Q_2 \in G \qquad \vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * P_2, \emptyset, \emptyset, Q_1 * Q_2)}{\vdash (\texttt{atomic}\ C)\ \textbf{sat}\ (P_1 * \boxed{P_2 * F}, \emptyset, G, Q_1 * \boxed{Q_2 * F})}$$



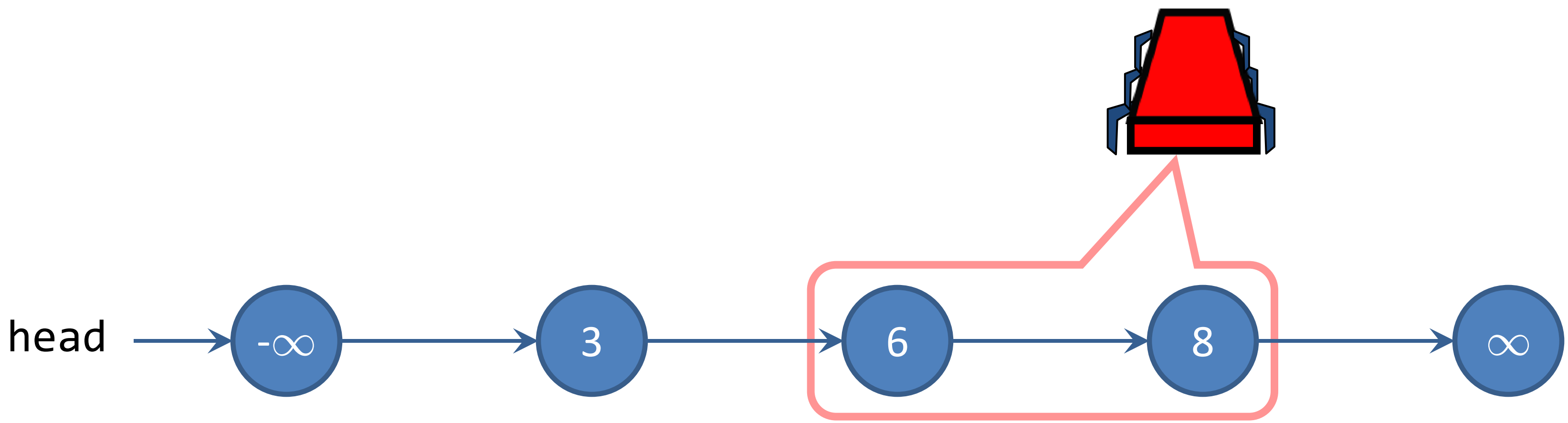
Challenge: Harris' Non-blocking List



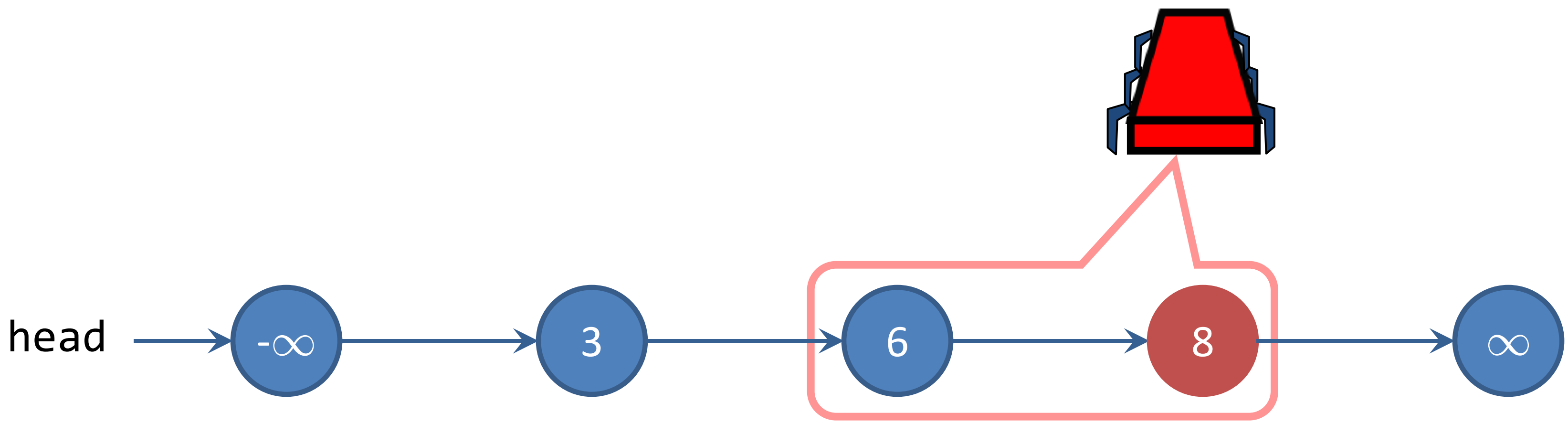
Challenge: Harris' Non-blocking List



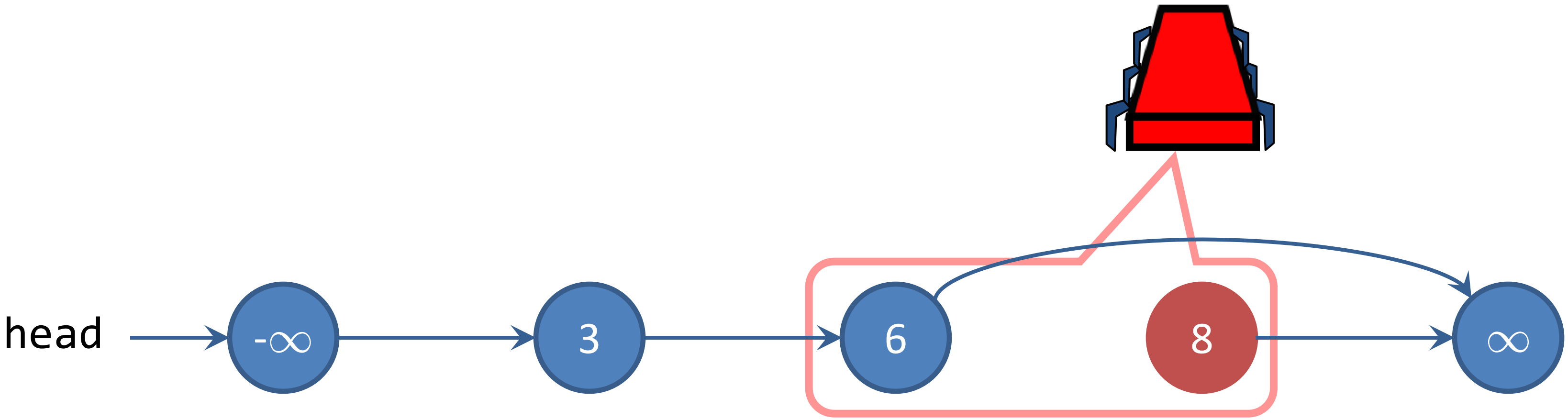
Challenge: Harris' Non-blocking List



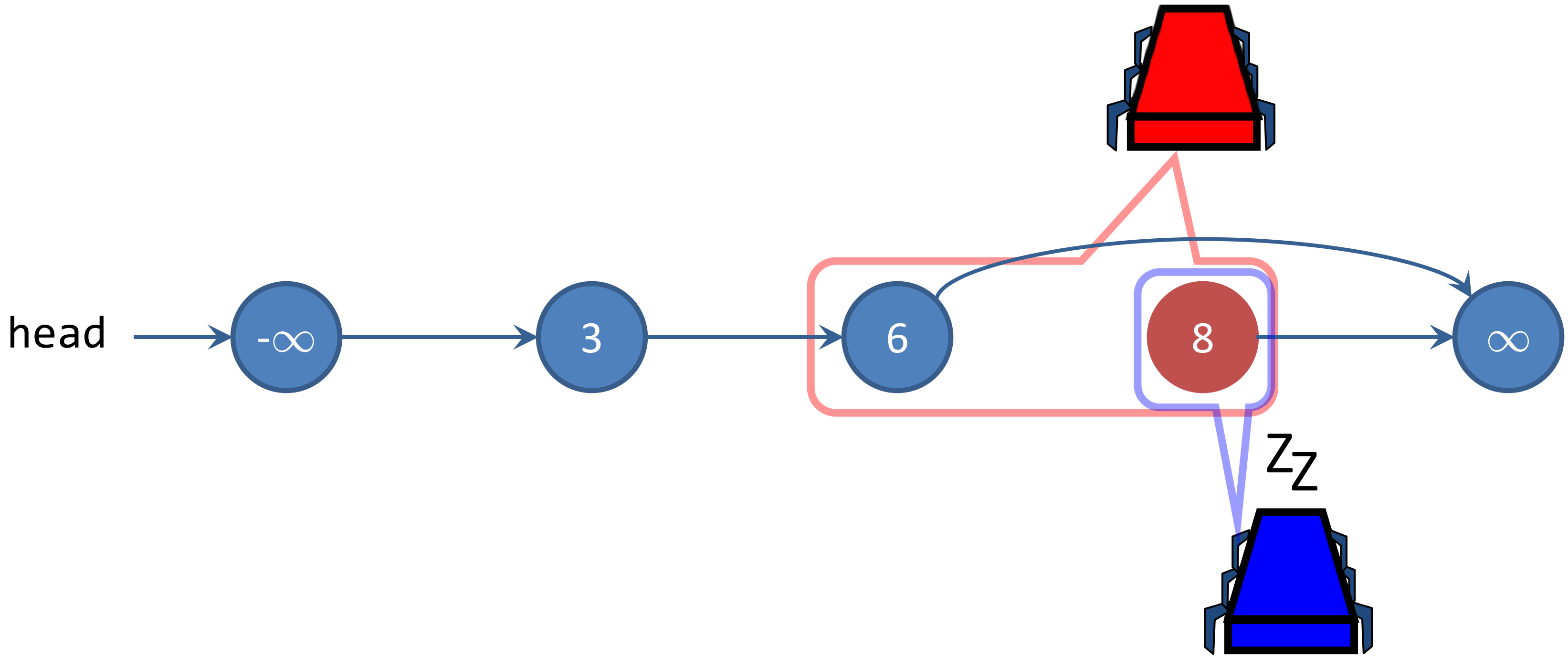
Challenge: Harris' Non-blocking List



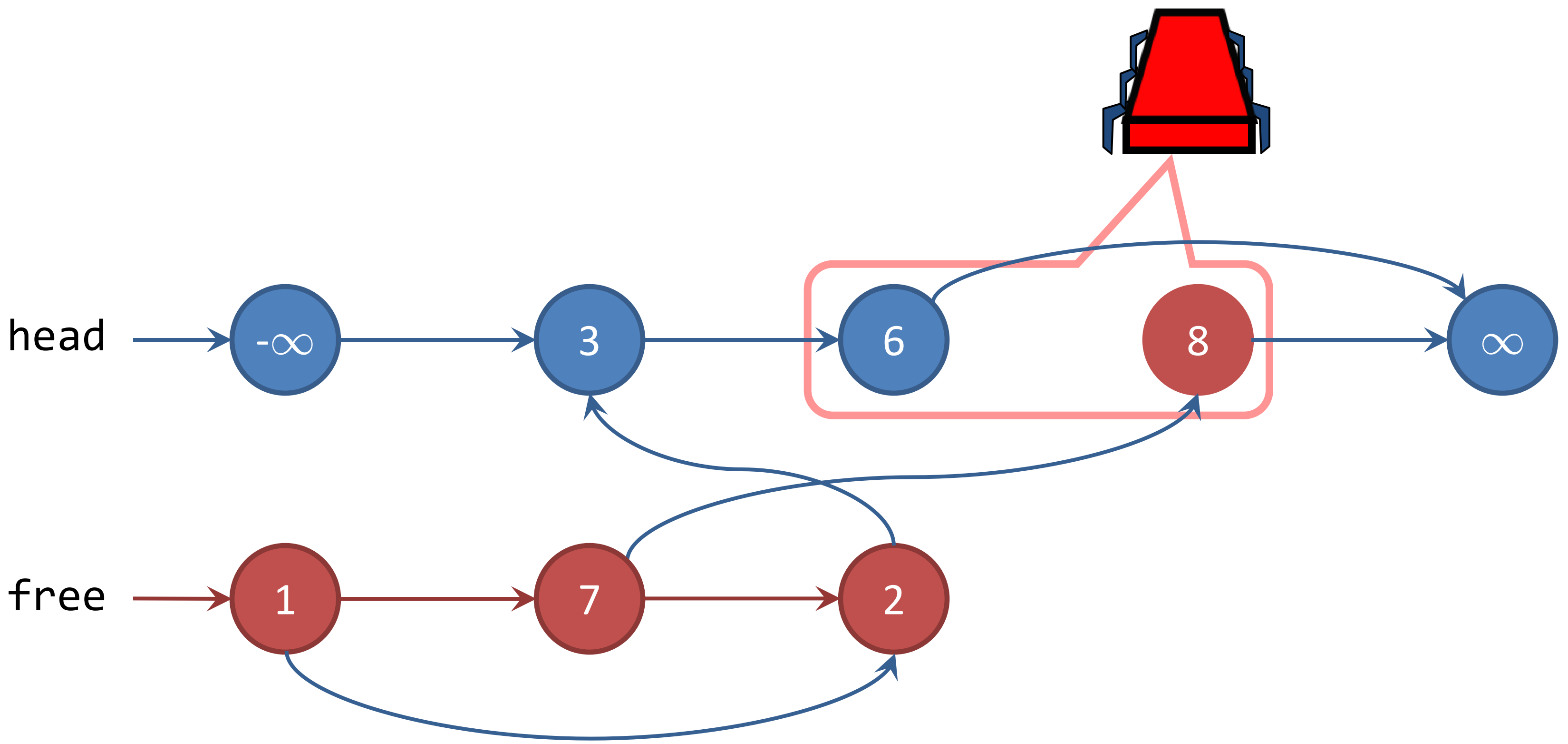
Challenge: Harris' Non-blocking List



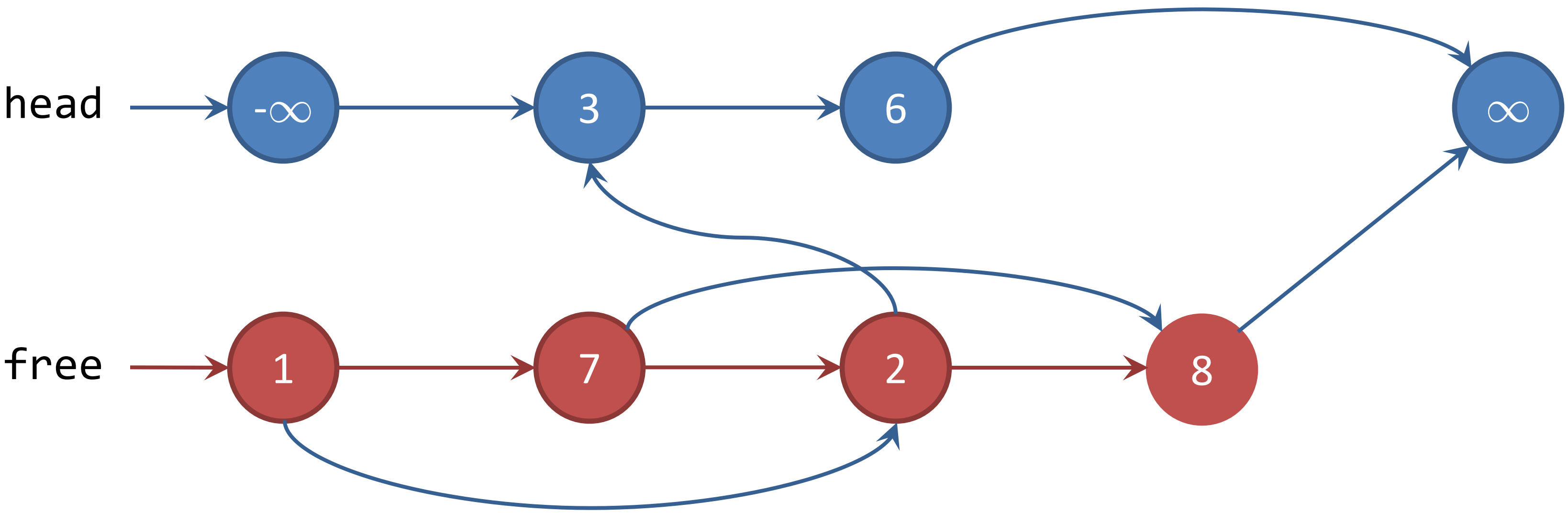
Challenge: Harris' Non-blocking List



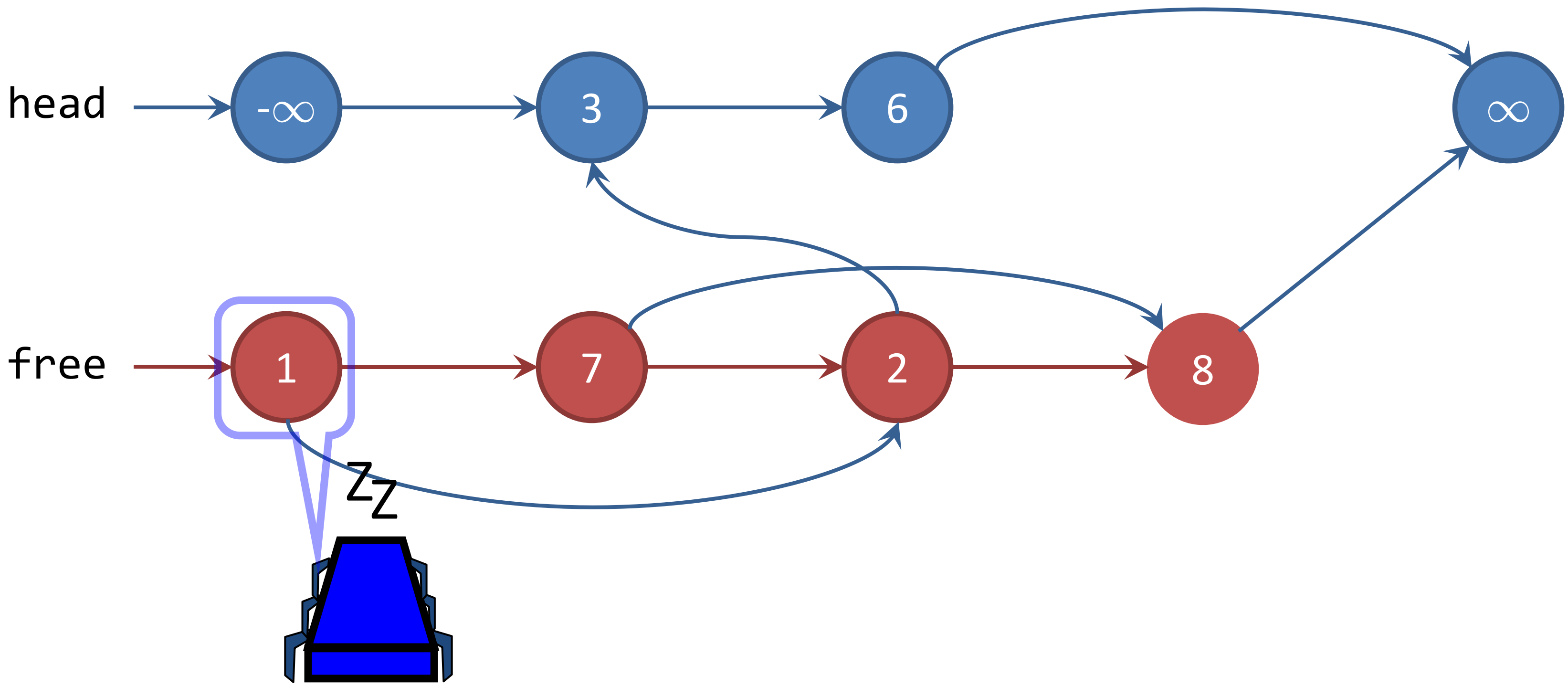
Challenge: Harris' Non-blocking List



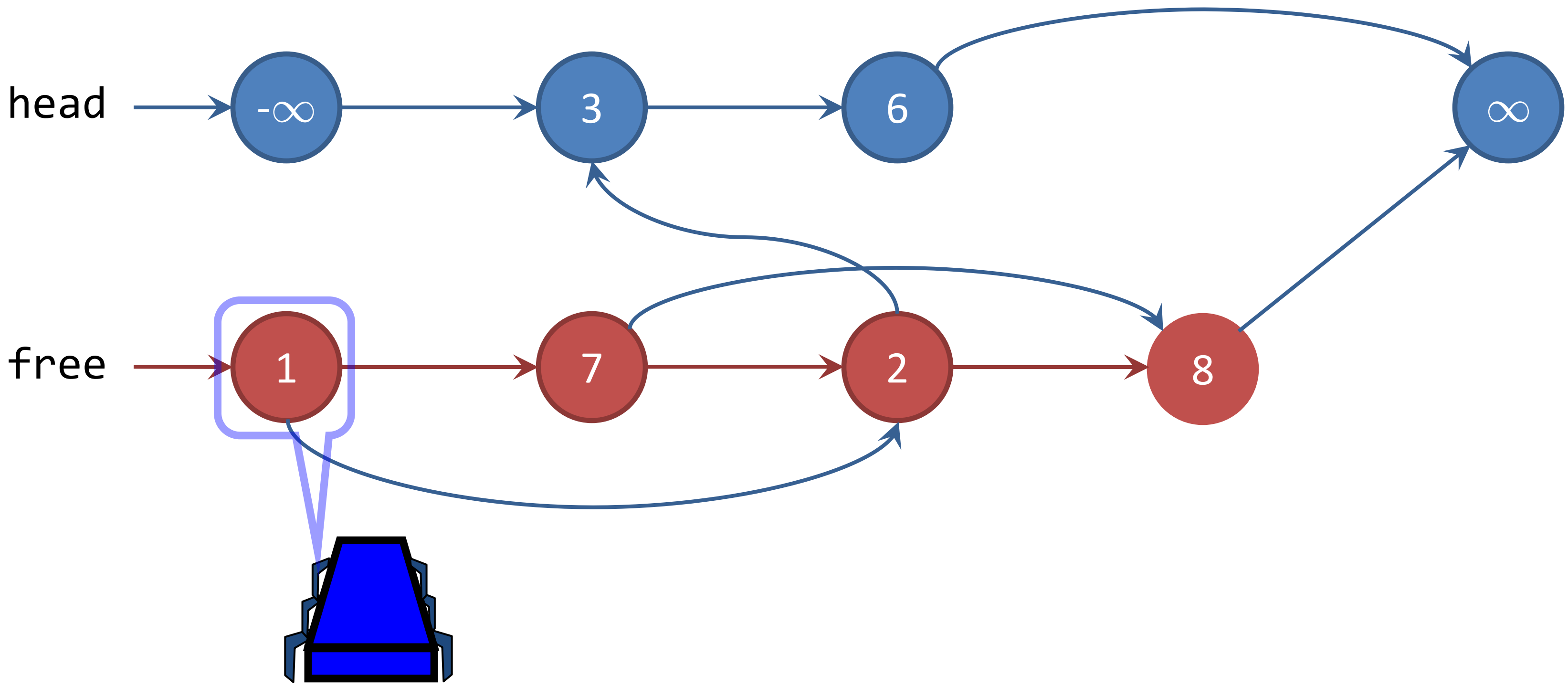
Challenge: Harris' Non-blocking List



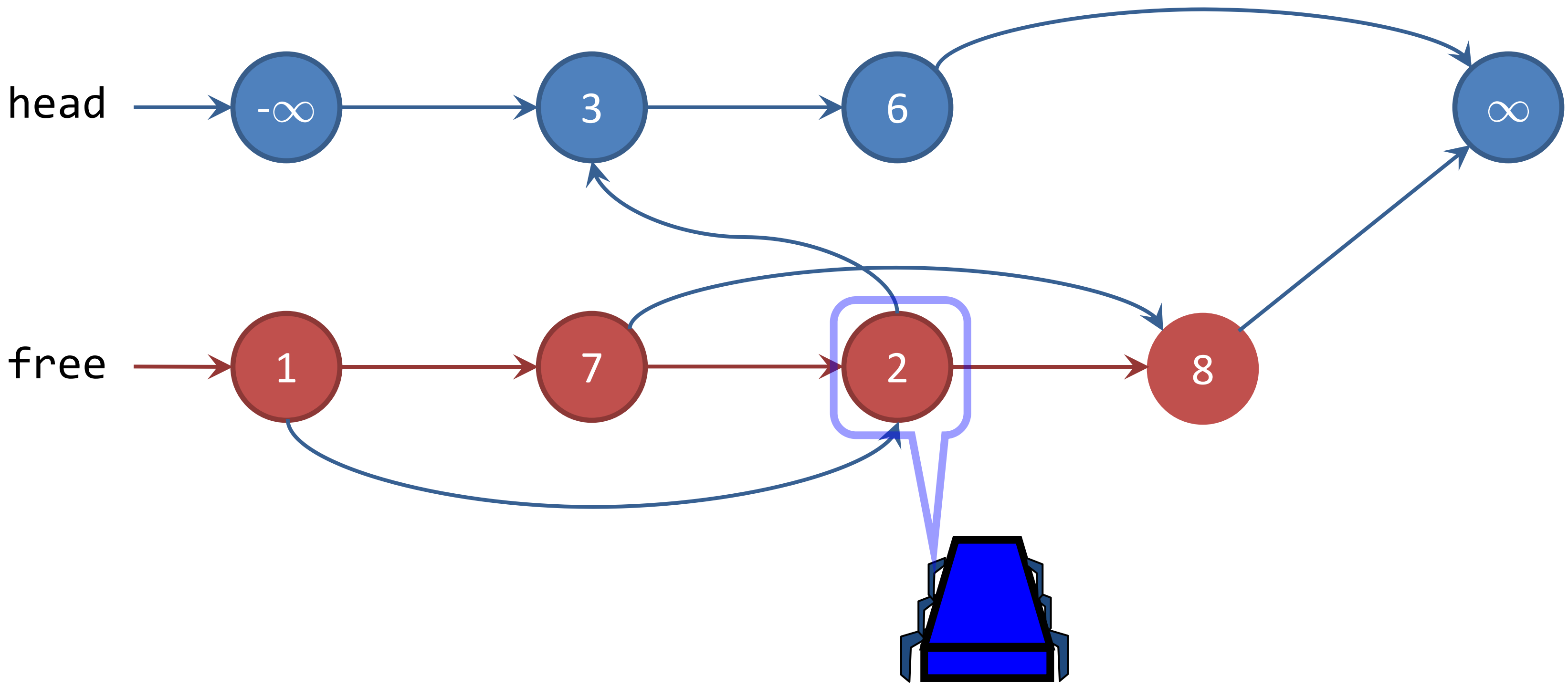
Challenge: Harris' Non-blocking List



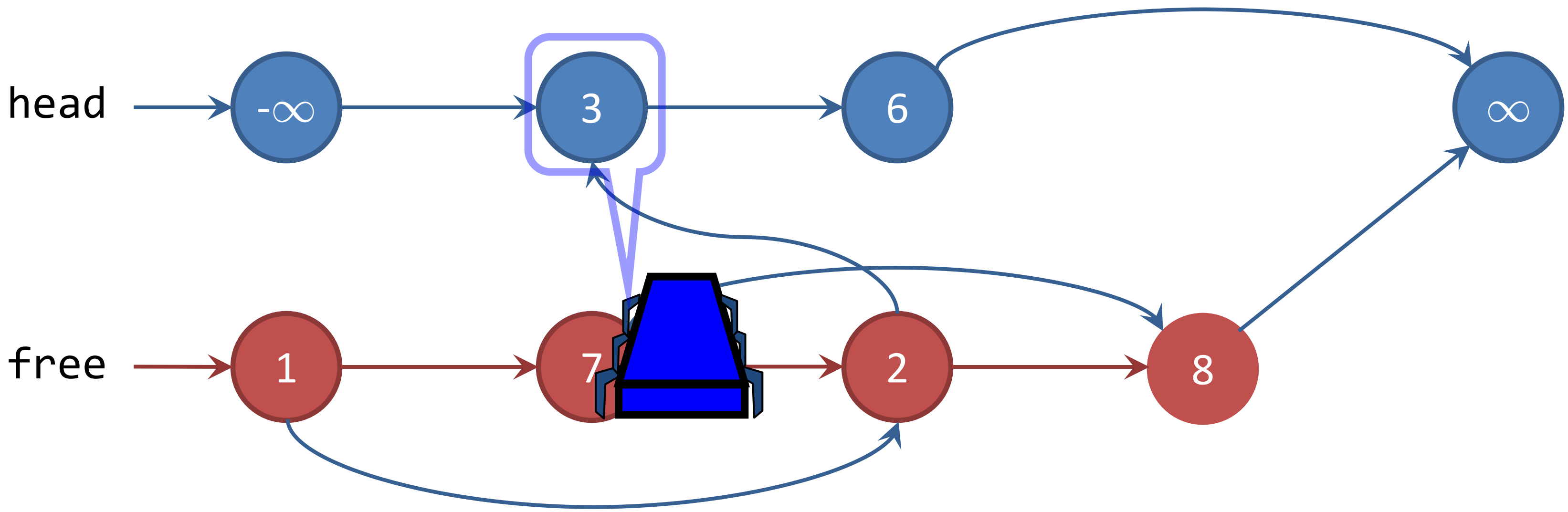
Challenge: Harris' Non-blocking List



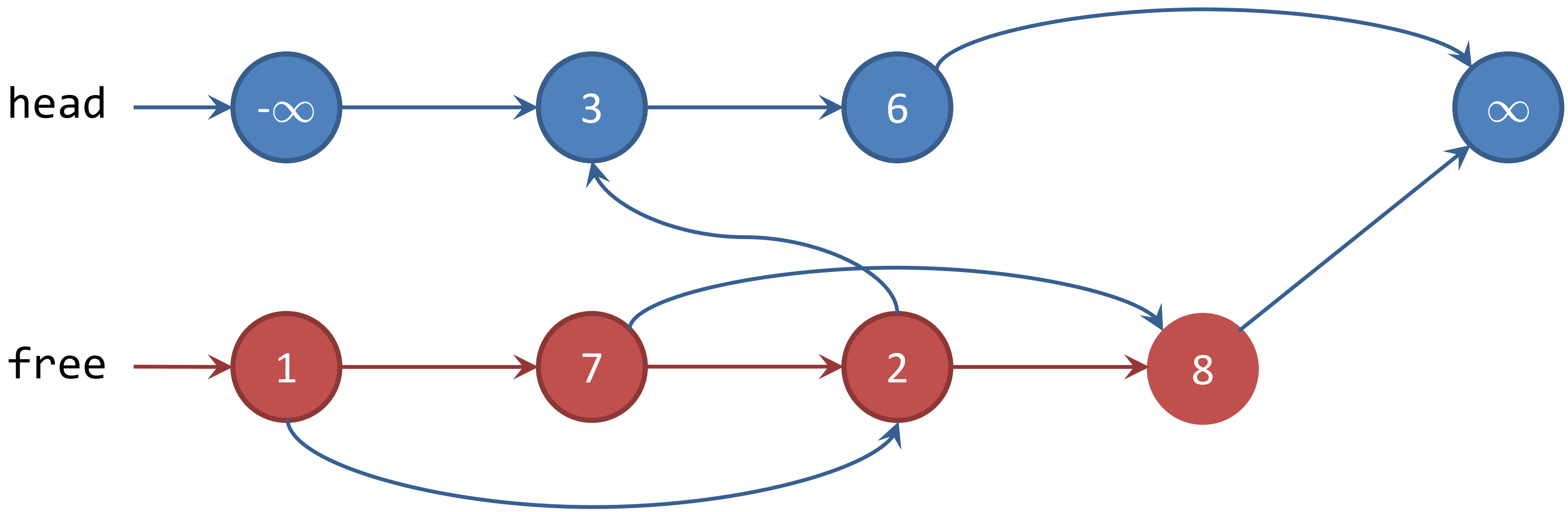
Challenge: Harris' Non-blocking List



Challenge: Harris' Non-blocking List



Challenge: Harris' Non-blocking List



Flow Interfaces

joint work with Siddharth Krishna and Dennis Shasha

Goal

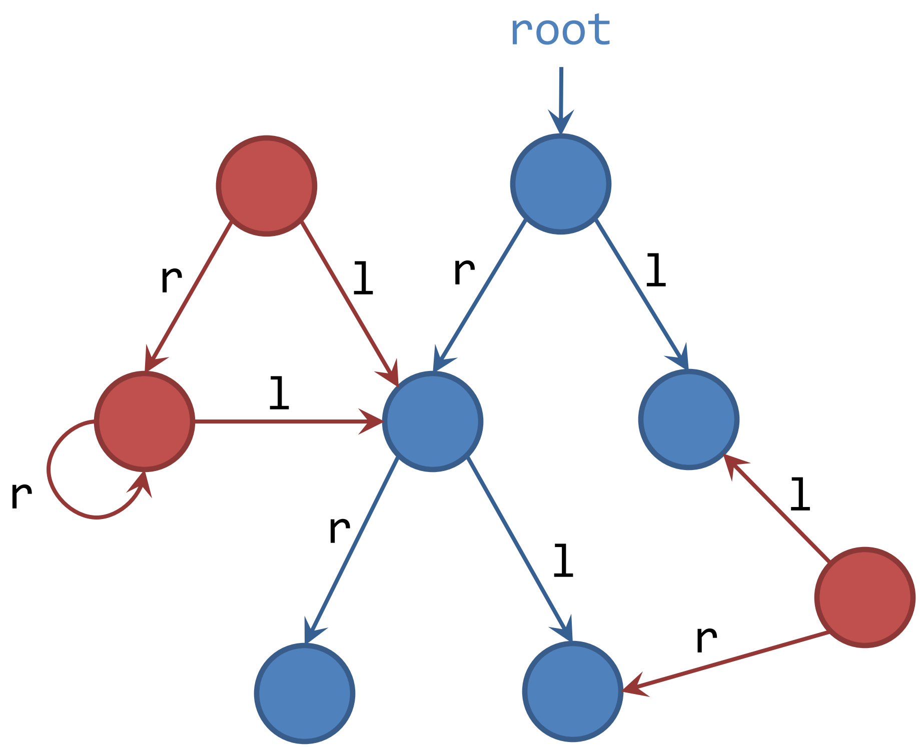
- Data structure abstractions that
  - can handle unbounded sharing and overlays
  - treat structural and data constraints uniformly
  - do not encode specific traversal strategies
  - provide data-structure-agnostic composition and decomposition rules
  - remain within general theory of separation logic

$\Rightarrow$ Flow Interfaces

High-Level Idea

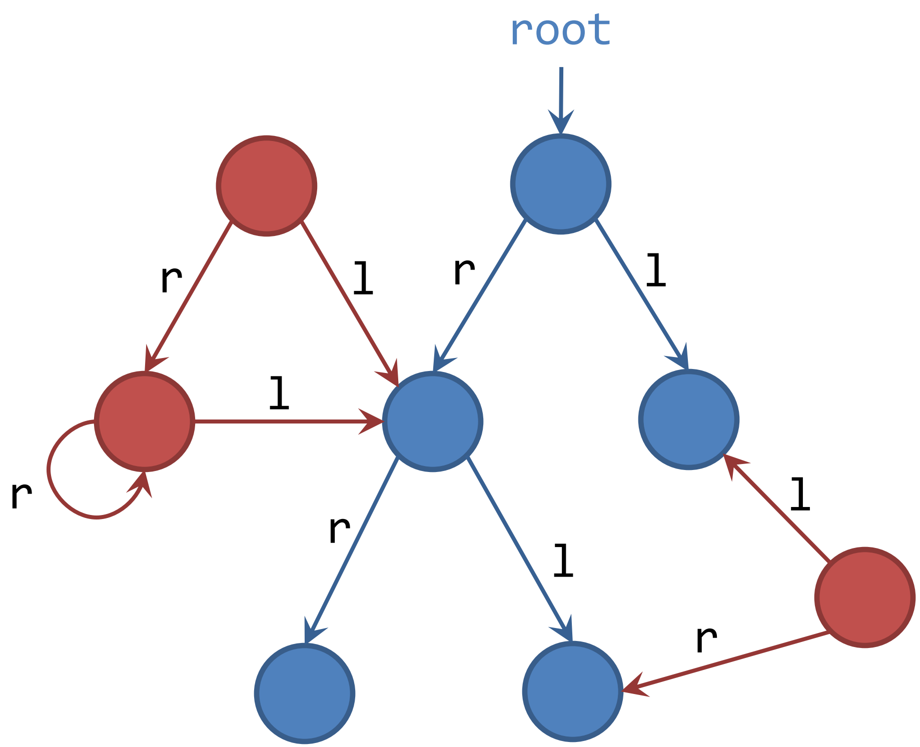
- Express all data structure invariants in terms of a local condition, satisfied by each node.
  - Local condition may depend on a quantity of the graph that is calculated inductively over the entire graph (the flow).
- Introduce a notion of graph composition that preserves local invariants of global flows.
- Introduce a generic *good graph* predicate that abstracts a heap region satisfying the local flow condition (the flow interface).

Local Data Structure Invariants with Flows



Can we express the property that `root` points to a tree as a local condition of each node in the graph?

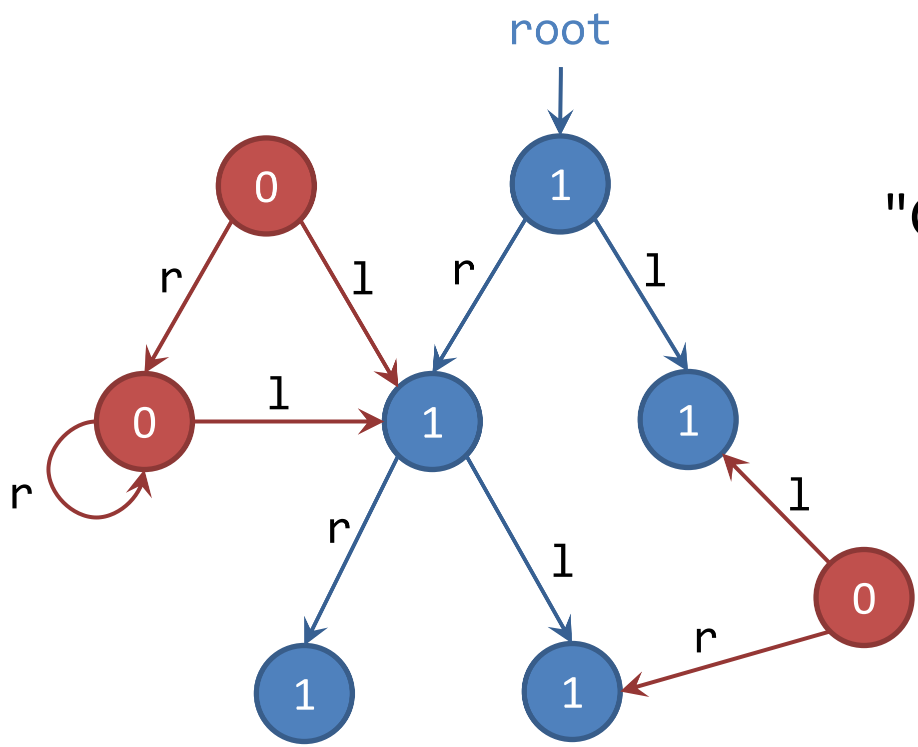
Local Data Structure Invariants with Flows



Path counting!

Can we express the property that `root` points to a tree as a local condition of each node in the graph?

Local Data Structure Invariants with Flows



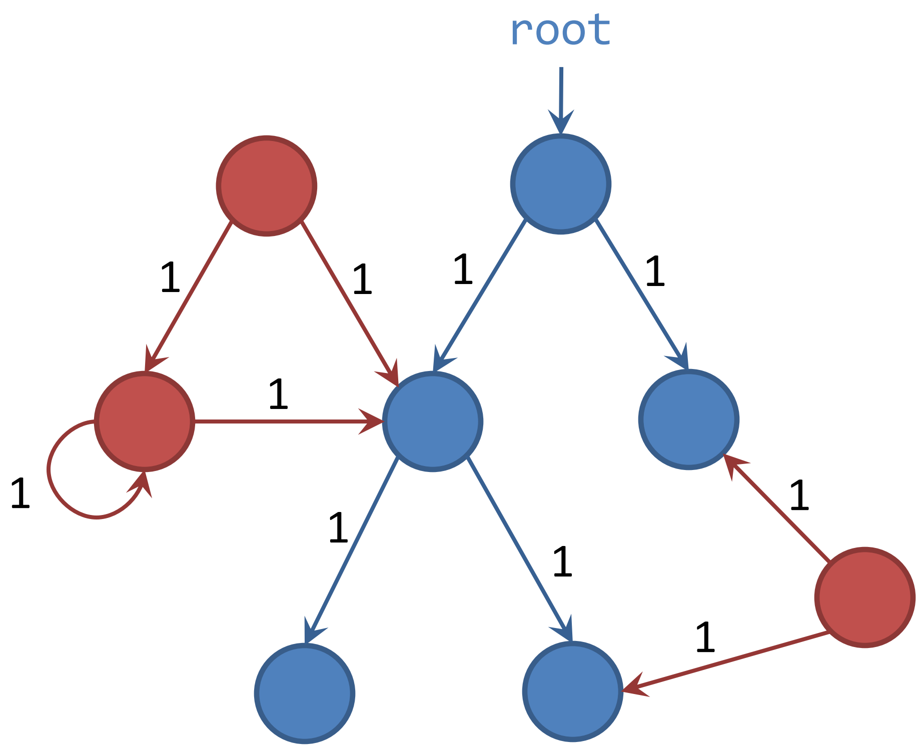
$\forall\, n \in N.\ pc(\texttt{root}, n) \leq 1$

"G contains a tree rooted at root"

Can we express the property that root points to a tree as a local condition of each node in the graph?

Flows

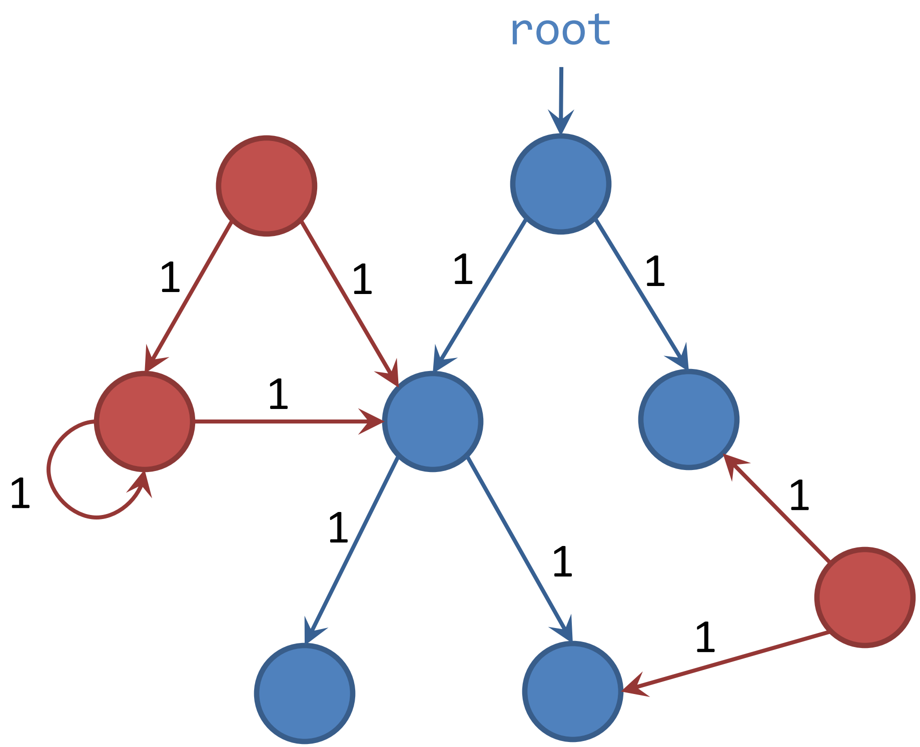
Step 1: Defining the Flow Graph



Label each edge in the graph with an element from some *flow domain* $(D, \sqsubseteq, +, \cdot, 0, 1)$

Flows

Step 1: Defining the Flow Graph



Requirements of flow domain:

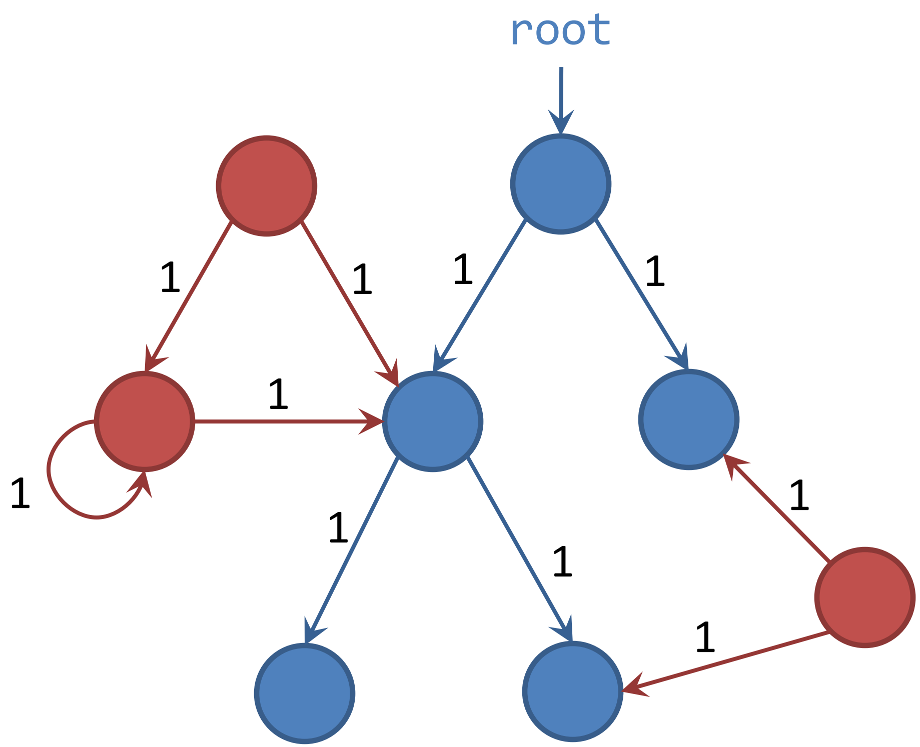
- (D, +, ·, 0, 1) is a semiring
- (D, $\sqsubseteq$) is ω-cpo with smallest element 0
- + and · are continuous

Path counting flow domain:
$(\mathbb{N} \cup \{\infty\}, \leq, +, \cdot, 0, 1)$

Label each edge in the graph with an element from some *flow domain* (D, $\sqsubseteq$, +, ·, 0, 1)

Flows

Step 1: Defining the Flow Graph



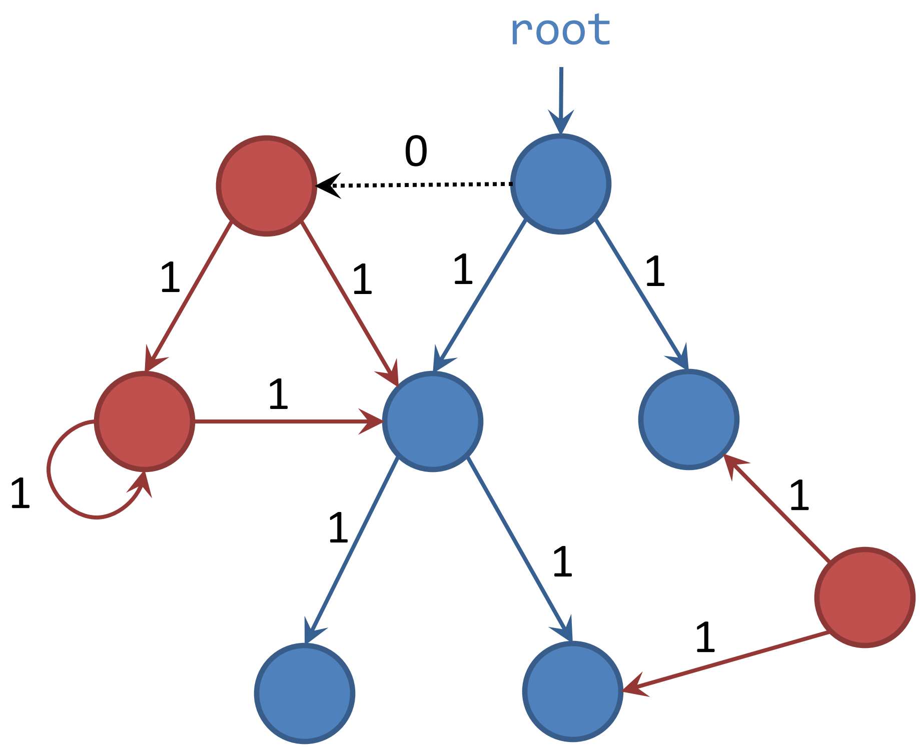
Flow graph $G = (N, e)$

- N finite set of nodes
- $e: N \times N \to D$

Label each edge in the graph with an element from some *flow domain* $(D, \sqsubseteq, +, \cdot, 0, 1)$

Flows

Step 1: Defining the Flow Graph



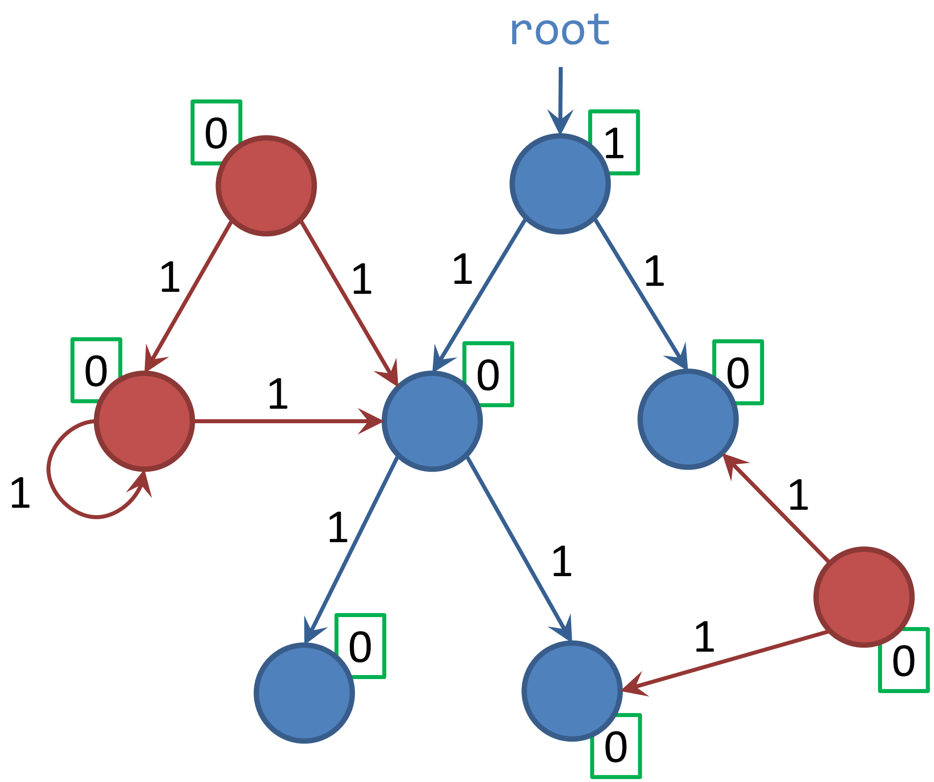
Flow graph $G = (N, e)$

- N finite set of nodes
- $e: N \times N \to D$

Label each edge in the graph with an element from some *flow domain* $(D, \sqsubseteq, +, \cdot, 0, 1)$

Flows

Step 2: Define the Inflow

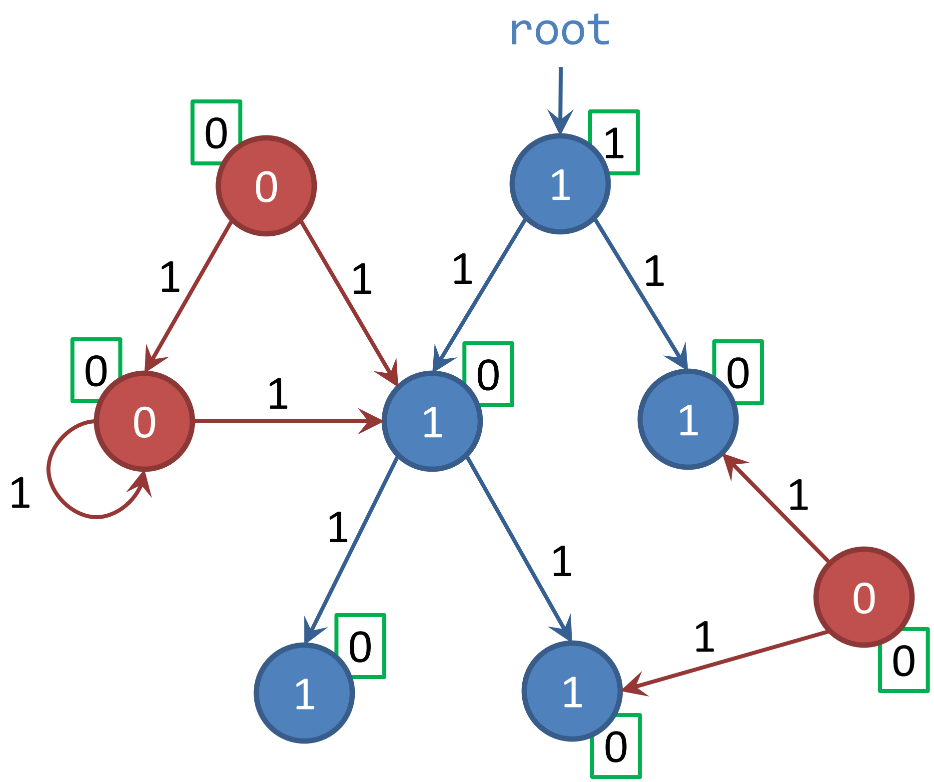


$$\color{green}{in}_{\color{blue}{\texttt{root}}}(n) = \begin{cases} 1, n = \color{blue}{\texttt{root}} \\ 0, n \neq \color{blue}{\texttt{root}} \end{cases}$$

Label each node using an *inflow in*: N → D

Flows

Step 3: Calculate the flow



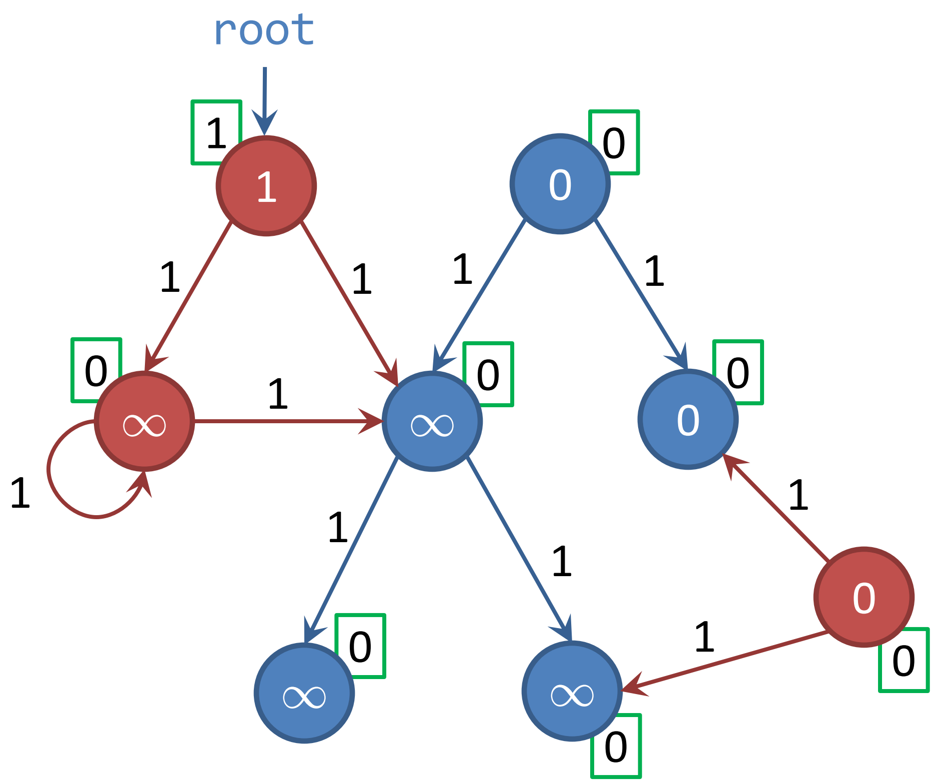
Flow graph G = (N, e)

flow$(in, G) : N \to D$

flow$(in, G) = \text{lfp}\left(\lambda C.\,\lambda n \in N.\, in(n) + \sum_{n' \in N} C(n') \cdot e(n', n)\right)$

Flows

Step 3: Calculate the flow



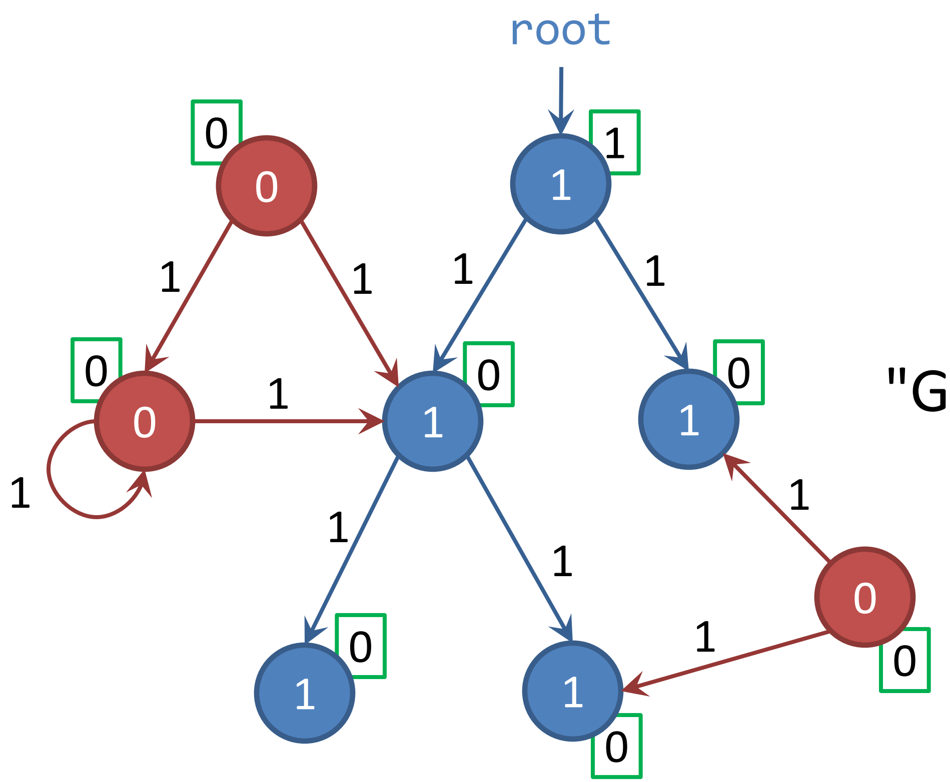
Flow graph G = (N, e)

$$\text{flow}(in, G) : N \to D$$

$$\text{flow}(in, G) = \text{lfp}\left(\lambda C.\, \lambda n \in N.\, in(n) + \sum_{n' \in N} C(n') \cdot e(n', n)\right)$$

Flows

Step 3: Calculate the flow



Flow graph G = (N, e)

$\forall$ n $\in$ N. flow(in_{root}, G)(n) $\leq$ 1

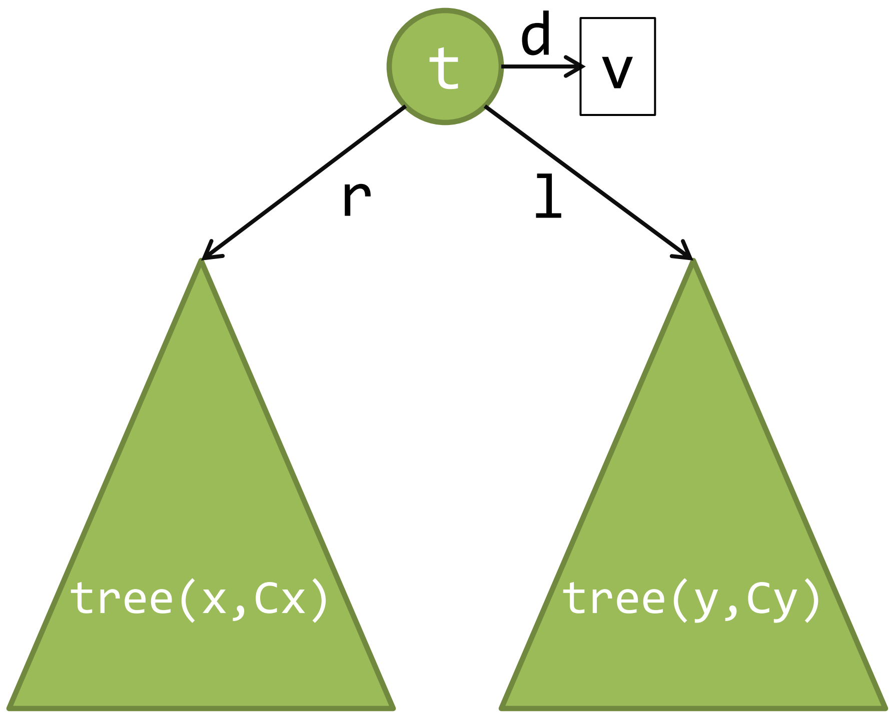
"G contains a tree rooted at root"

$$\text{flow}(in, G) : N \to D$$

$$\text{flow}(in, G) = \text{lfp}\left(\lambda C.\, \lambda n \in N.\, in(n) + \sum_{n' \in N} C(n') \cdot e(n', n)\right)$$

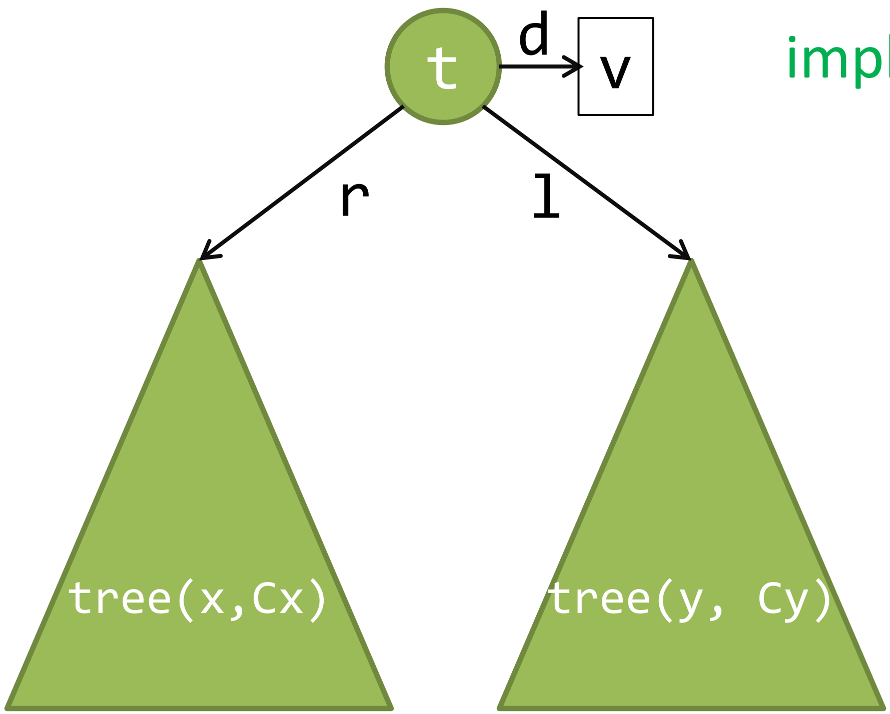
Data Constraints

```
predicate tree(t: Node, C: Set<Int>) {
  t == null ∧ emp ∧ C = ∅ ∨
  ∃ v, x, y, Cx, Cy ::
    t↦(d:v,r:x,l:y) * tree(x,Cx) * tree(y,Cy) ∧
    C = {v}∪Cx∪Cy ∧ v>Cx ∧ v<Cy
}
```



Data Invariants

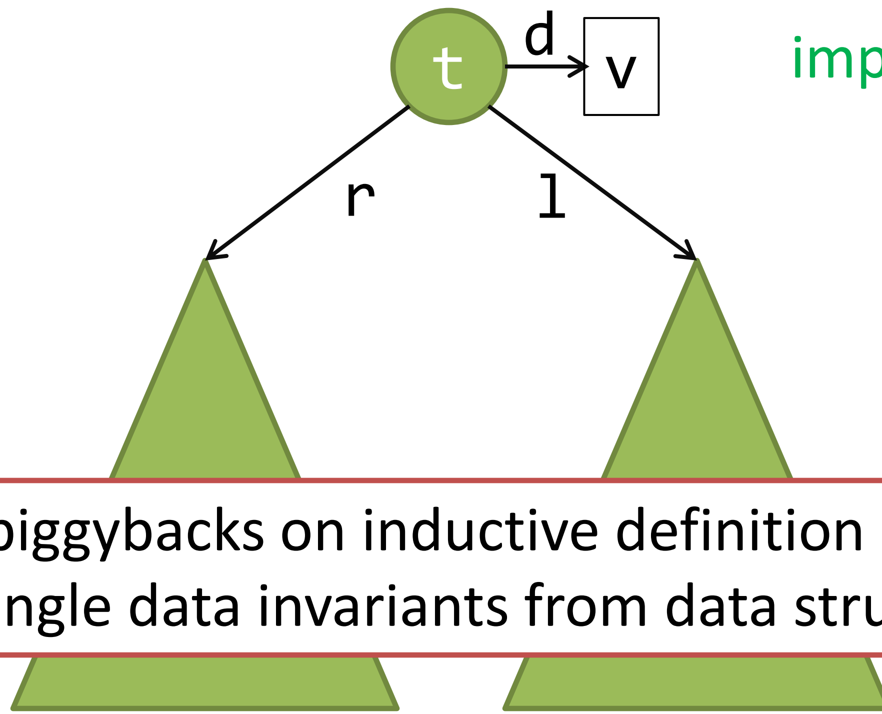
```
predicate tree(t: Node, C: Set<Int>) {
  t == null ∧ emp ∧ C = ∅ ∨
  ∃ v, x, y, Cx, Cy ::
    t↦(d:v,r:x,l:y) * tree(x,Cx) * tree(y,Cy) ∧
    C = {v}∪Cx∪Cy ∧ v>Cx ∧ v<Cy
}
```



implies $Cx \cap Cy = \emptyset$

Data Invariants

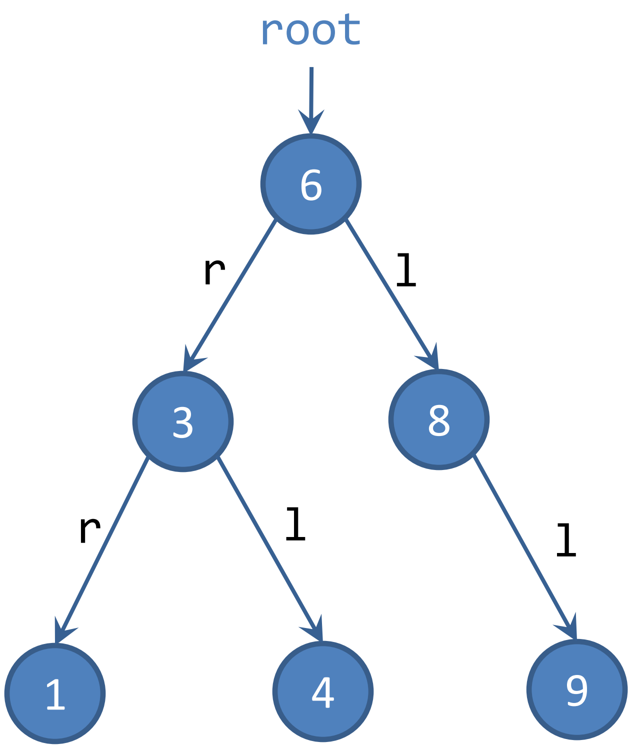
```
predicate tree(t: Node, C: Set<Int>) {
  t == null ∧ emp ∧ C = ∅ ∨
  ∃ v, x, y, Cx, Cy ::
    t↦(d:v,r:x,l:y) ∗ tree(x,Cx) ∗ tree(y,Cy) ∧
    C = {v}∪Cx∪Cy ∧ v>Cx ∧ v<Cy
}
```



implies $Cx \cap Cy = \emptyset$

Data invariant piggybacks on inductive definition of the tree.
⇒ hard to entangle data invariants from data structure specifics.

Inset Flows

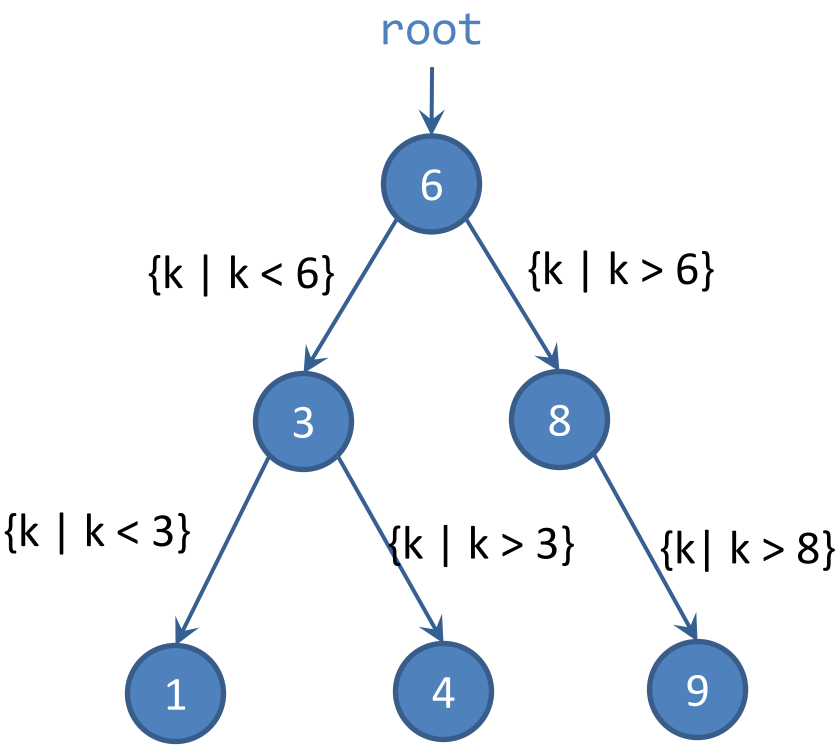


KS: the set of all search keys
e.g. KS = $\mathbb{Z}$

Inset flow domain:
$(2^{\mathrm{KS}}, \subseteq, \cup, \cap, \emptyset, \mathrm{KS})$

Label each edge with the set of keys that follow that edge in a search (edgeset).

Inset Flows

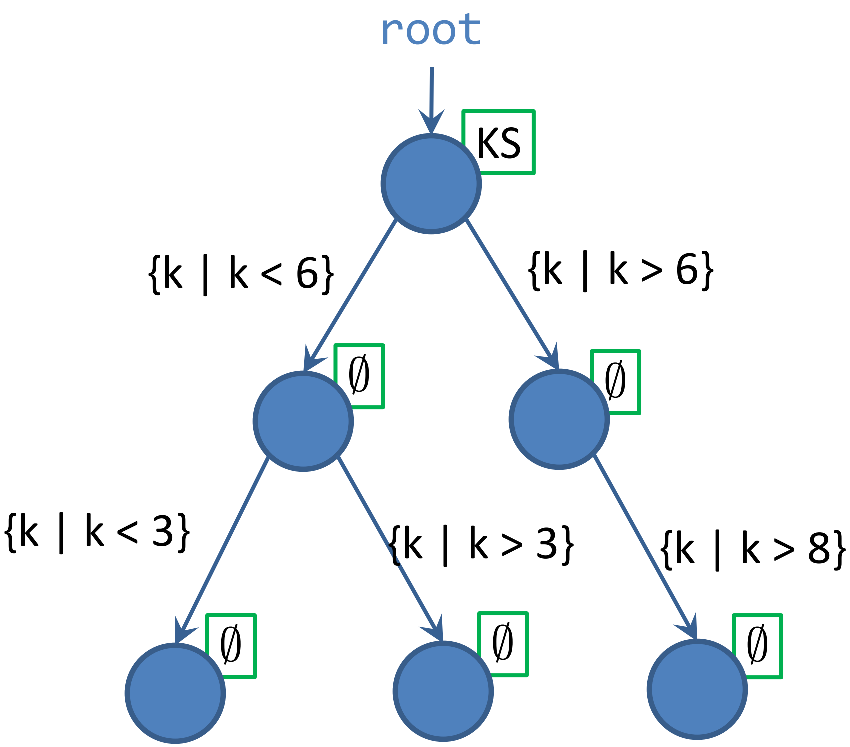


KS: the set of all search keys
e.g. KS = $\mathbb{Z}$

Inset flow domain:
$(2^{KS}, \subseteq, \cup, \cap, \emptyset, KS)$

Label each edge with the set of keys that follow that edge in a search (edgeset).

Inset Flows

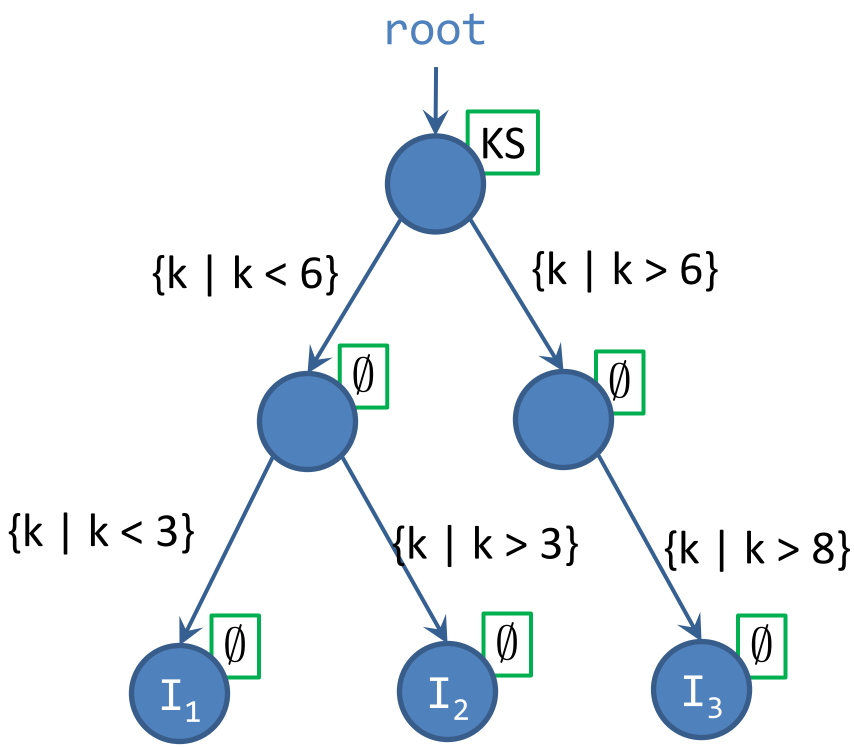


KS: the set of all search keys
e.g. KS = $\mathbb{Z}$

Inset flow domain:
$(2^{KS}, \subseteq, \cup, \cap, \emptyset, KS)$

Set inflow *in* of `root` to KS and to $\emptyset$ for all other nodes.

Inset Flows



$I_1 = \{k \mid 3 < k\}$

$I_2 = \{k \mid 3 < k < 6\}$

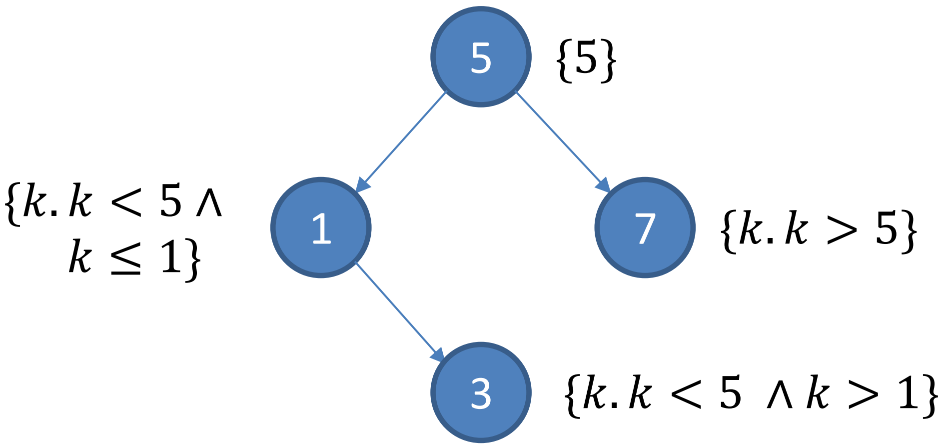
$I_3 = \{k \mid 8 < k\}$

flow(in, G)(n) is the *inset* of node n, i.e., the set of keys k such that a search for k will traverse node n.

From Insets to Keysets

$$outset(G)(n) = \bigcup_{n \in N} e(n, n')$$

$$keyset(in, G)(n) = inset(in, G)(n) \setminus outset(G)(n)$$



keyset(in, G)(n) is the set of keys that could be in n

Verifying Concurrent Search Data Structures

- Local data structure invariants
  - edgesets are disjoint for each n: $\{e(n,n')\}_{n' \in N}$ are disjoint
  - keyset of each n covers n's contents: $C(G)(n) \subseteq \text{keyset}(in, G)(n)$
- Observation: disjoint inflows imply disjoint keysets
  - If $\{in(n)\}_{n \in N}$ are disjoint (e.g. G has a single root)
  - then $\{\text{keyset}(in,G)(n)\}_{n \in N}$ are disjoint

$\Rightarrow$ Can be used to prove linearizability of concurrent search data structures in a data-structure-agnostic fashion

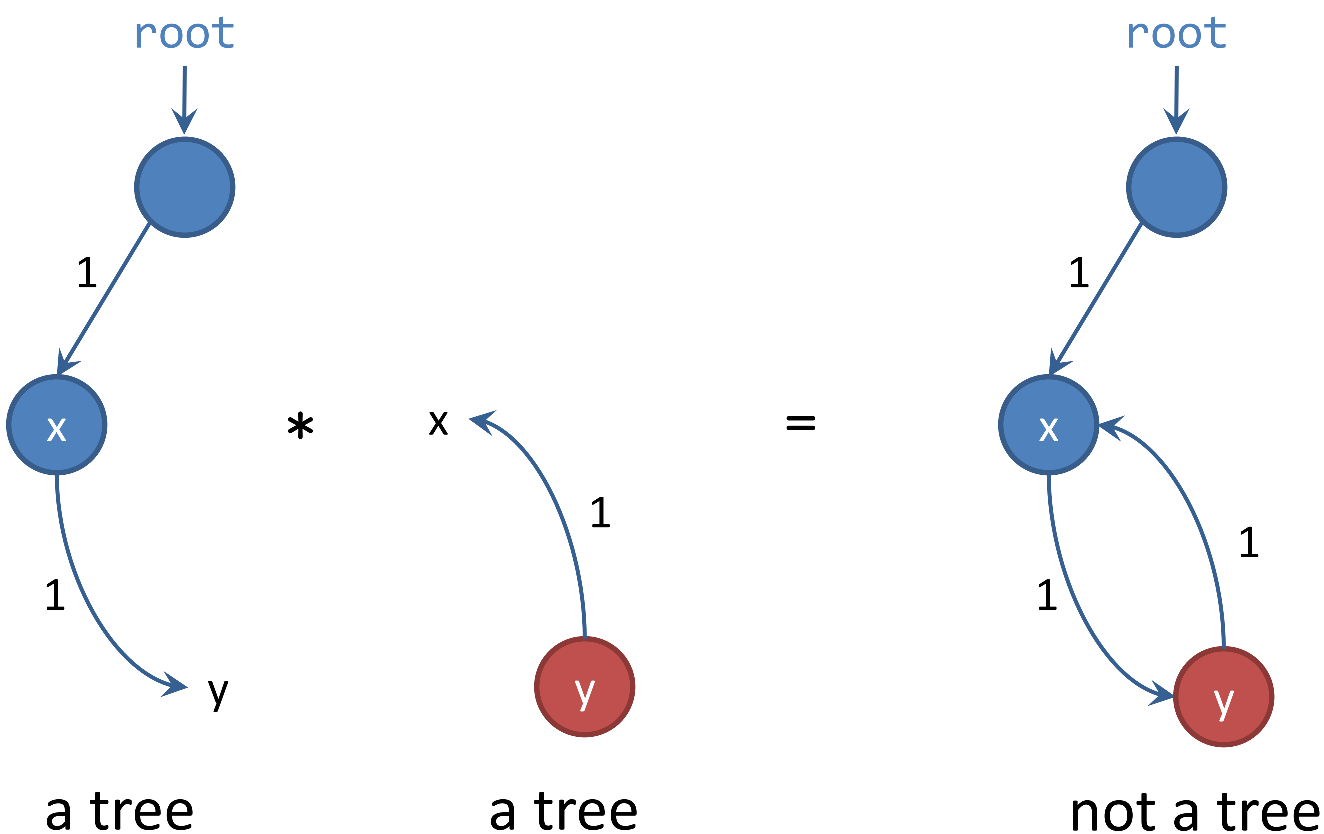
[Shasha and Goodman, 1988]

Compositional Reasoning

Can we reason compositionally about flows and flow graphs à la SL?

Flow Graph Composition

- Standard SL Composition (disjoint union) is too weak:



Flow Interface Graph

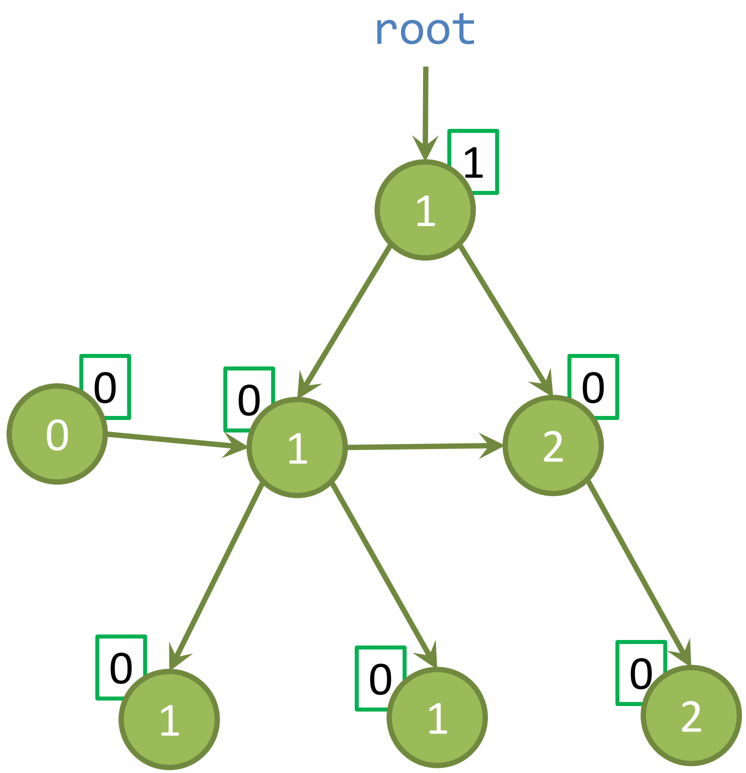
(in, G) is a *flow interface graph* iff

- $G = (N, N_o, \lambda, e)$ is a partial graph with
  - N the set of internal nodes of the graph
  - N_o the set of external nodes of the graph
  - λ: N $\rightarrow$ A a node labeling function
  - e: N x ($N \cup N_o$) $\rightarrow$ D is an edge function
- in: $N \rightarrow D$ is an inflow

Inflow *in* specifies rely of G on its context.

Flow Interface Graph Composition

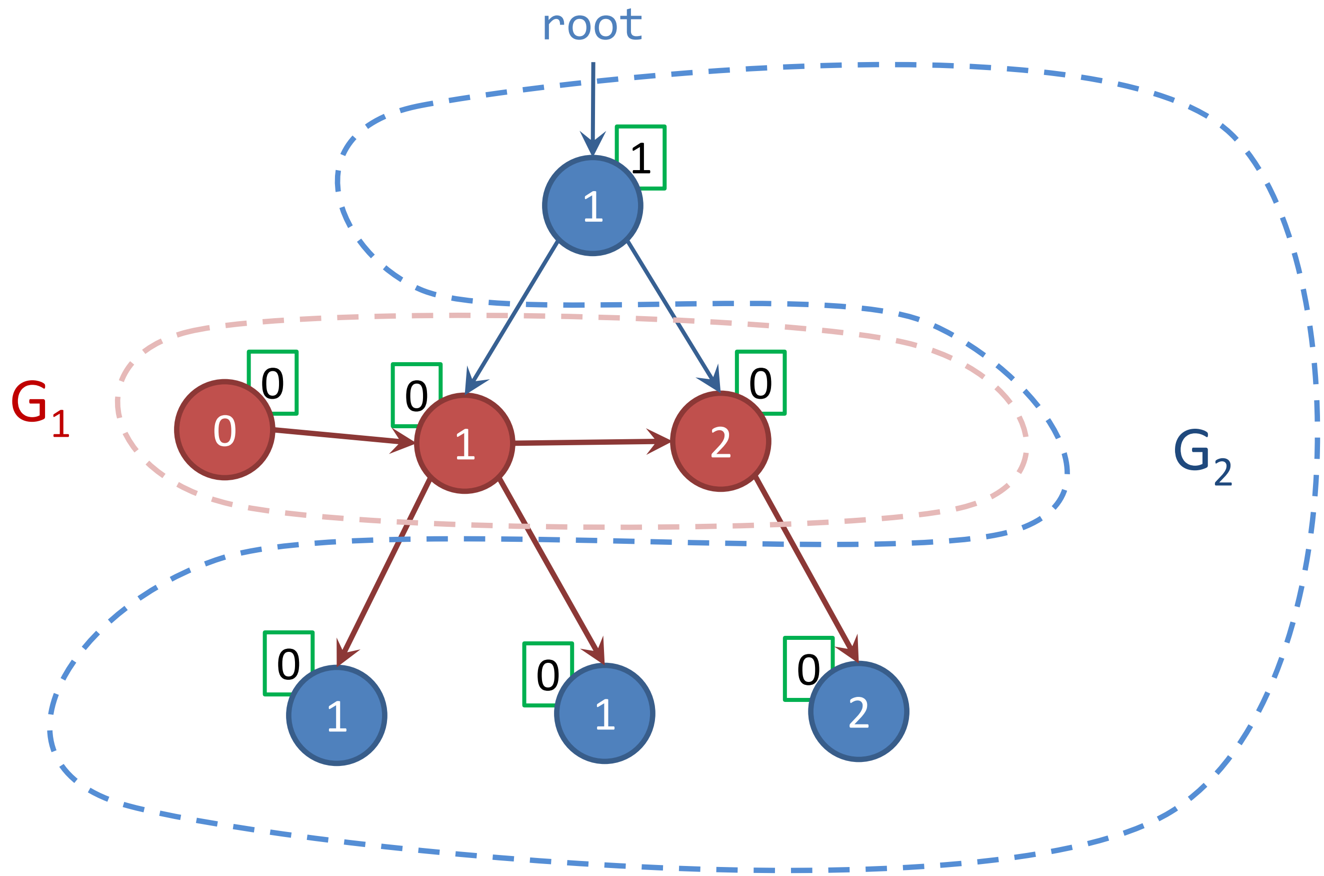
(in, G)



Flow Interface Graph Composition

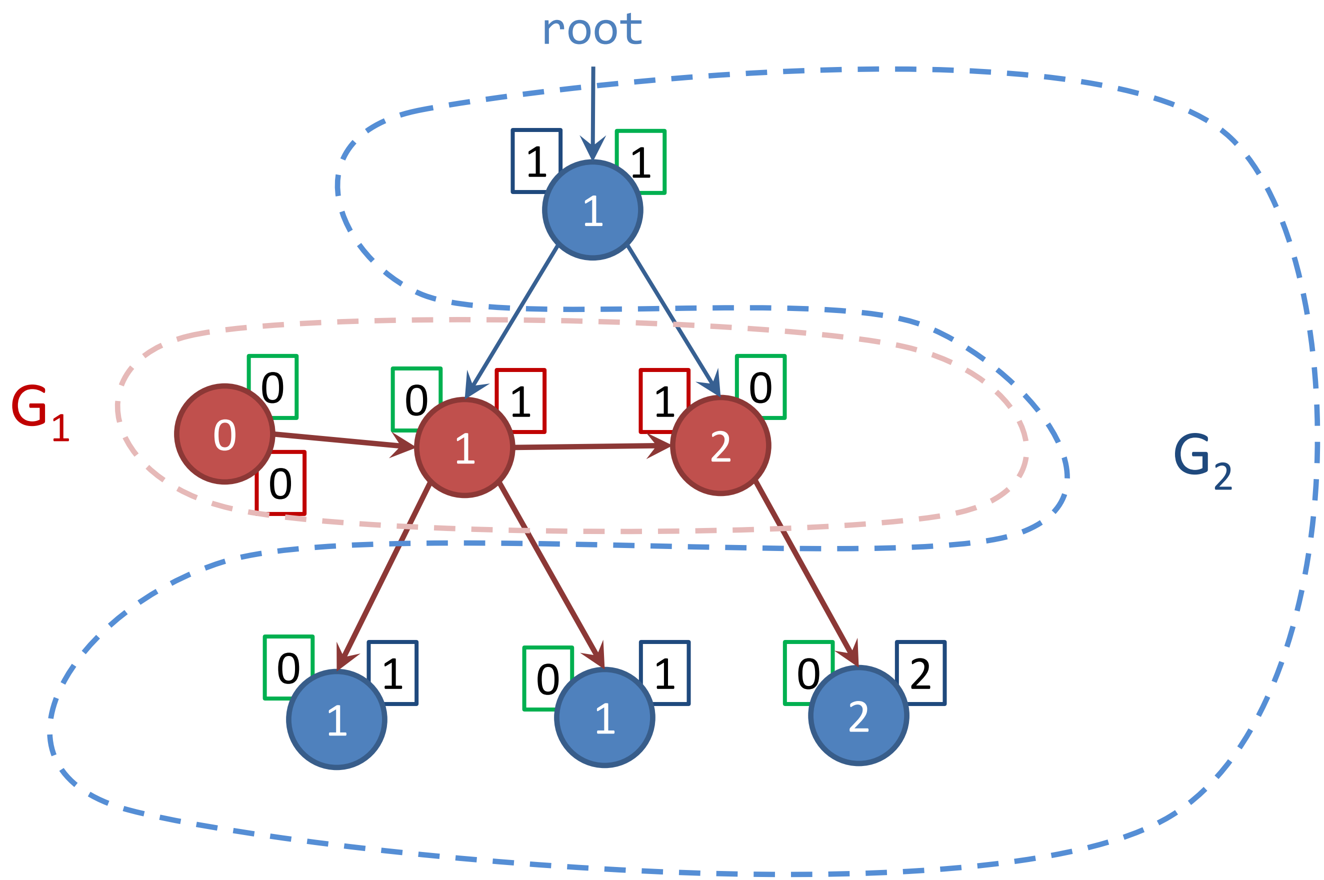
$(in, G) = (in_1, G_1) \bullet (in_2, G_2)$

$in_1 = ?$, $in_2 = ?$



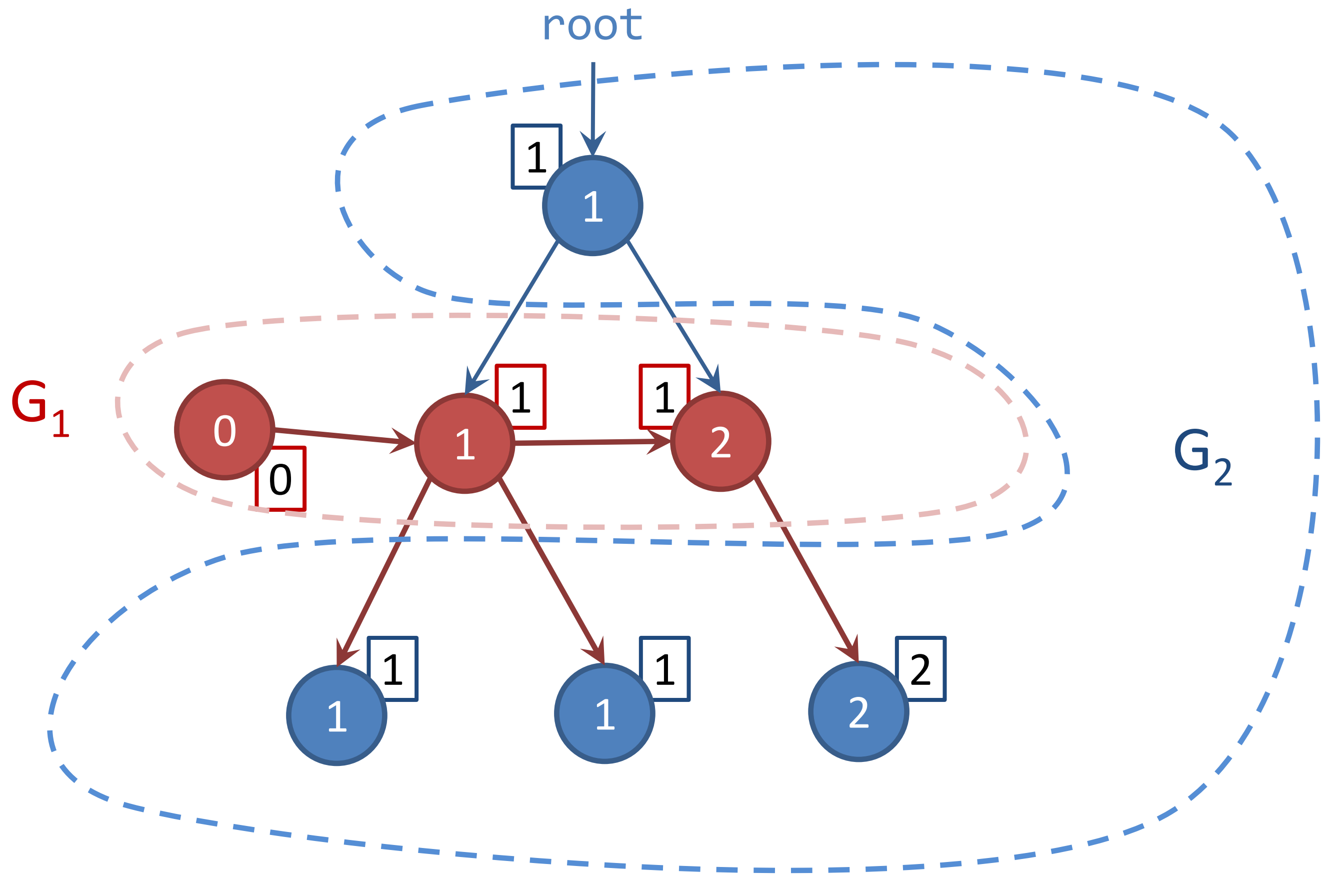
Flow Interface Graph Composition

$(in, G) = (in_1, G_1) \bullet (in_2, G_2)$



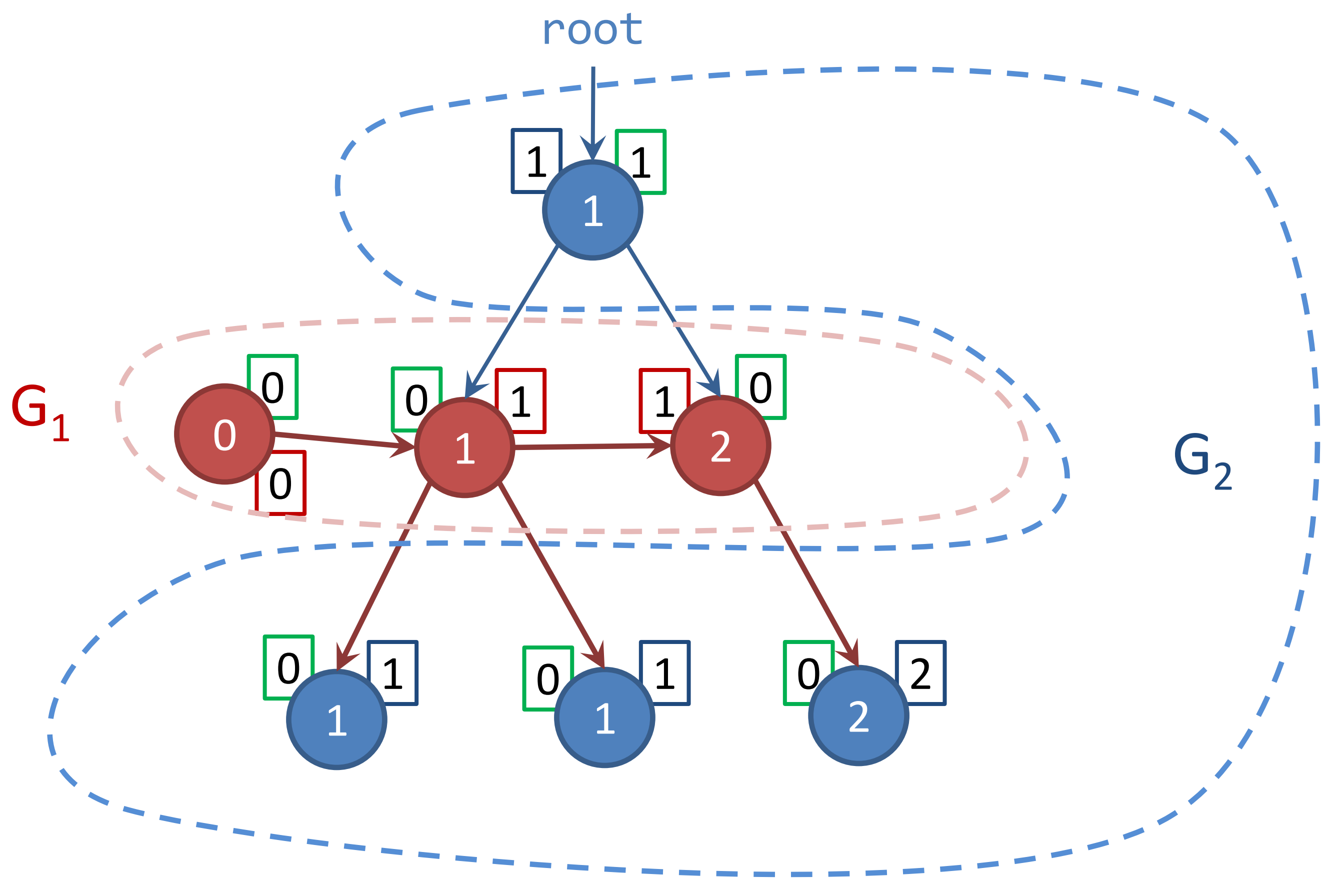
Flow Interface Graph Composition

$(in, G) = (in_1, G_1) \bullet (in_2, G_2)$



Flow Interface Graph Composition

$(in, G) = (in_1, G_1) \bullet (in_2, G_2)$



Flow Interface Graph Composition

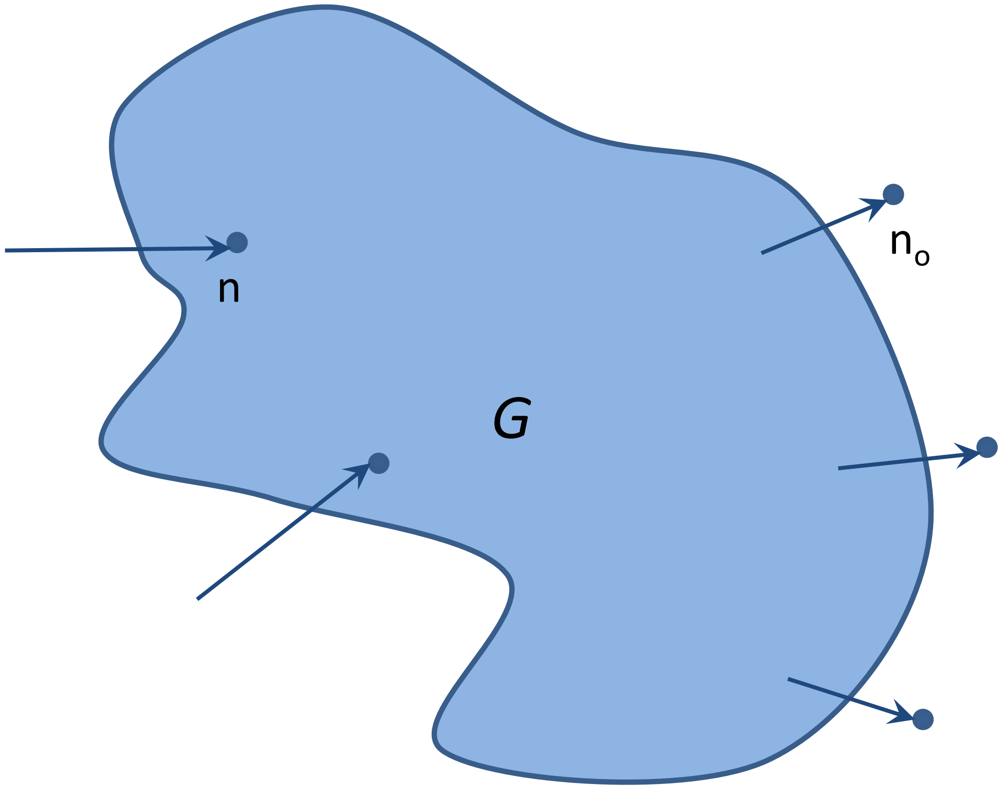
- $H_1 \bullet H_2$ is
  - commutative: $H_1 \bullet H_2 = H_2 \bullet H_1$
  - associative : $(H_1 \bullet H_2) \bullet H_3 = H_1 \bullet (H_2 \bullet H_3)$
  - cancelative: $\mathrm{H} \bullet H_1 = \mathrm{H} \bullet H_2 \Rightarrow H_1 = H_2$

$\Rightarrow$ Flow interface graphs form a *separation algebra*.

$\Rightarrow$ We can use them to give semantics to SL assertions.

- How do we abstract flow interface graphs?

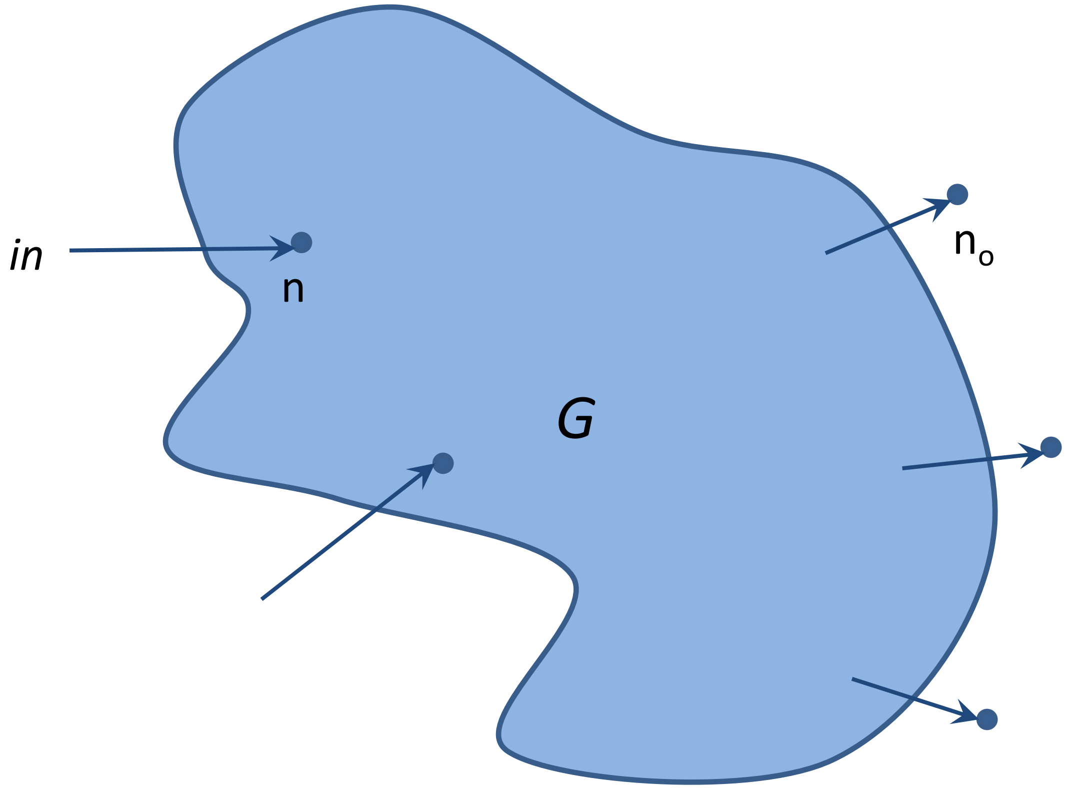
Flow Map of a Flow Interface Graph



$\mathrm{fm}(G)(n, n_o) = \sum \{ \, \mathit{pathproduct}(p) \mid p \text{ path from } n \text{ to } n_o \text{ in } G \}$

$\mathrm{flow}(in, G)(n_o) = \sum \{ \, \mathrm{in}(\mathrm{n}) \cdot \mathrm{fm}(G)(n, n_o) \mid \mathrm{n} \in \mathrm{G} \}$

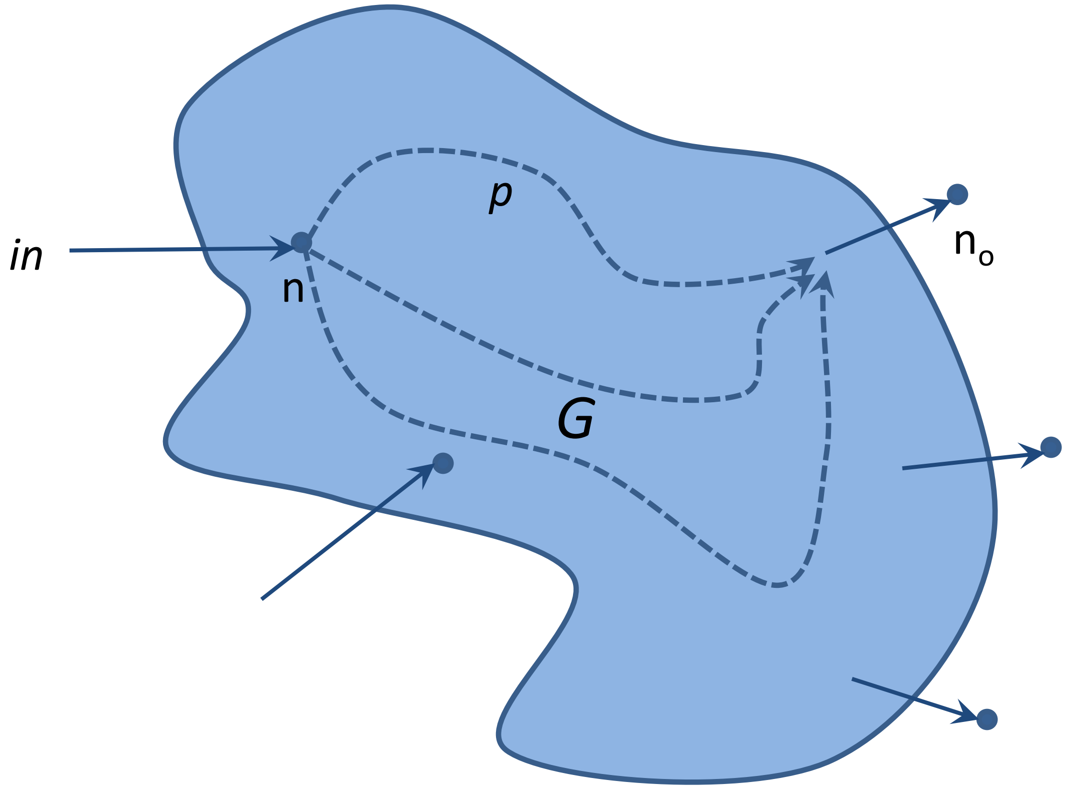
Flow Map of a Flow Interface Graph



$\mathrm{fm}(G)(n, n_o) = \sum \{\, \mathit{pathproduct}(p) \mid p \text{ path from } n \text{ to } n_o \text{ in } G \}$

$\mathrm{flow}(in, G)(n_o) = \sum \{\, \mathrm{in}(\mathrm{n}) \cdot \mathrm{fm}(G)(n, n_o) \mid \mathrm{n} \in \mathrm{G} \}$

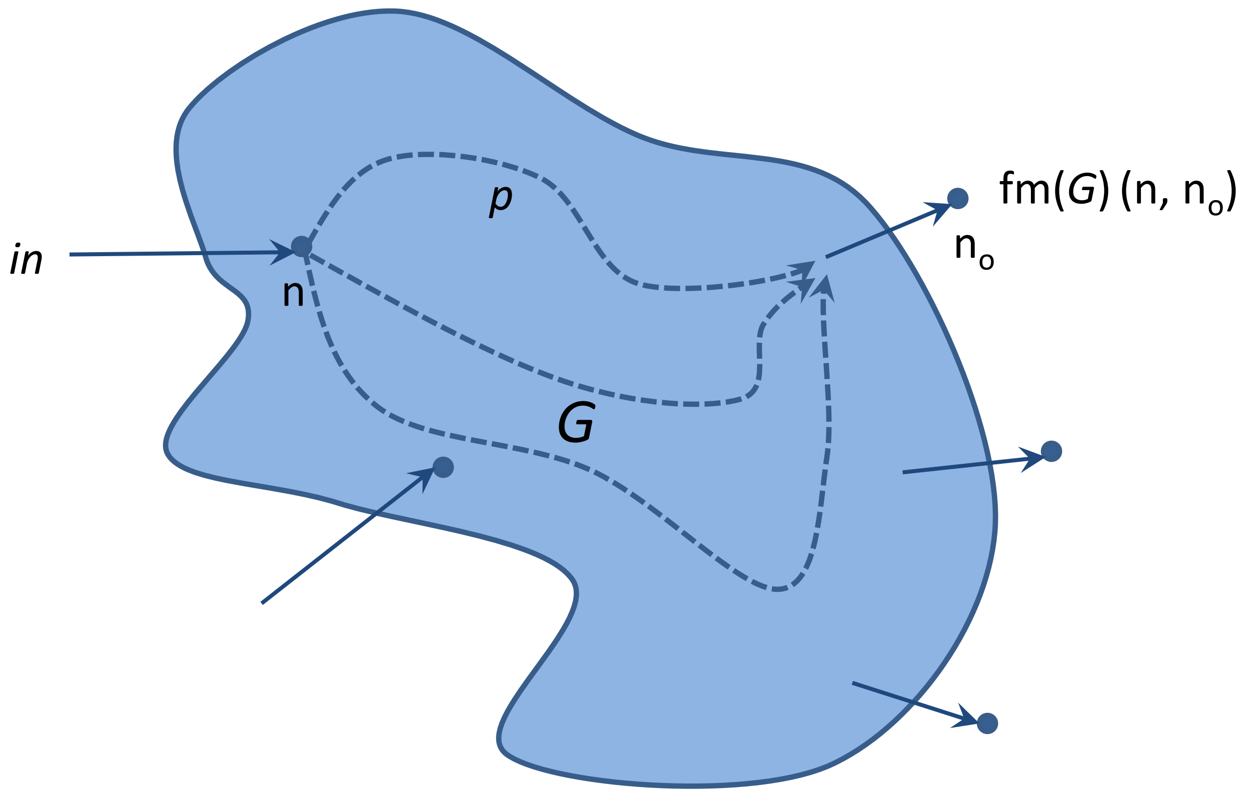
Flow Map of a Flow Interface Graph



$\mathrm{fm}(G)(n, n_o) = \sum \{ \, pathproduct(p) \mid p$ path from n to n_o in $G\}$

$\mathrm{flow}(in, G)(n_o) = \sum \{ \, \mathrm{in}(n) \cdot \mathrm{fm}(G)(n, n_o) \mid \mathrm{n} \in \mathrm{G}\}$

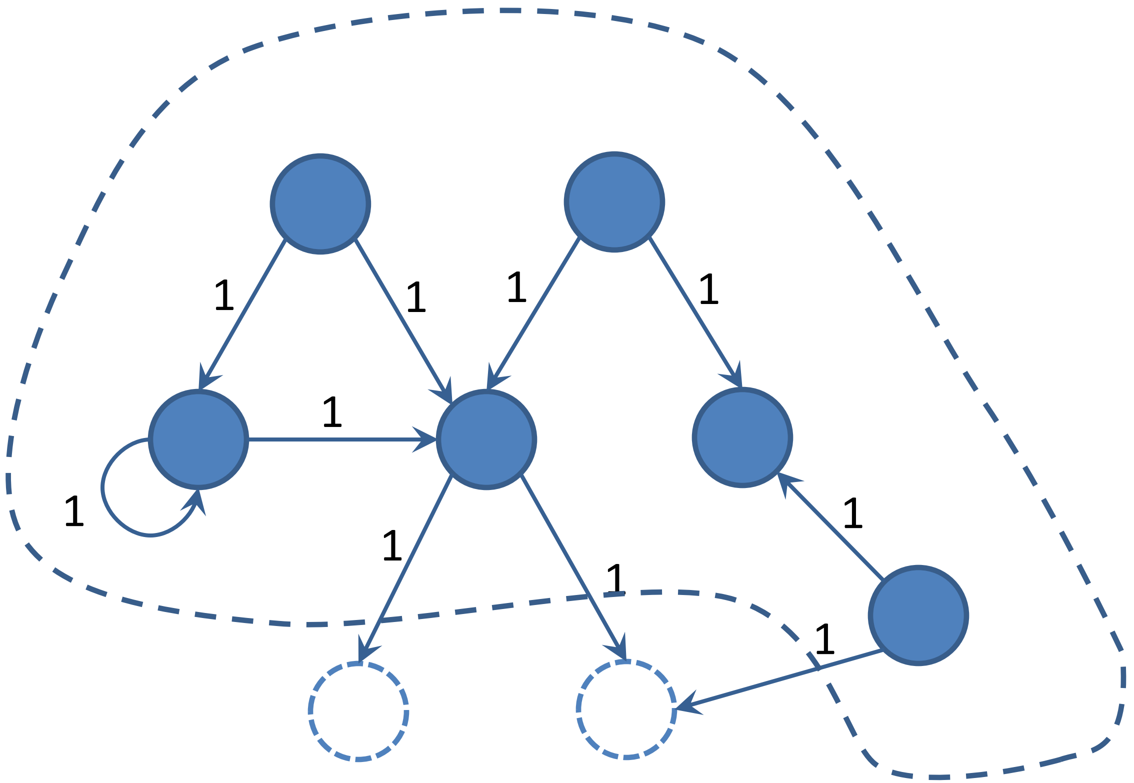
Flow Map of a Flow Interface Graph



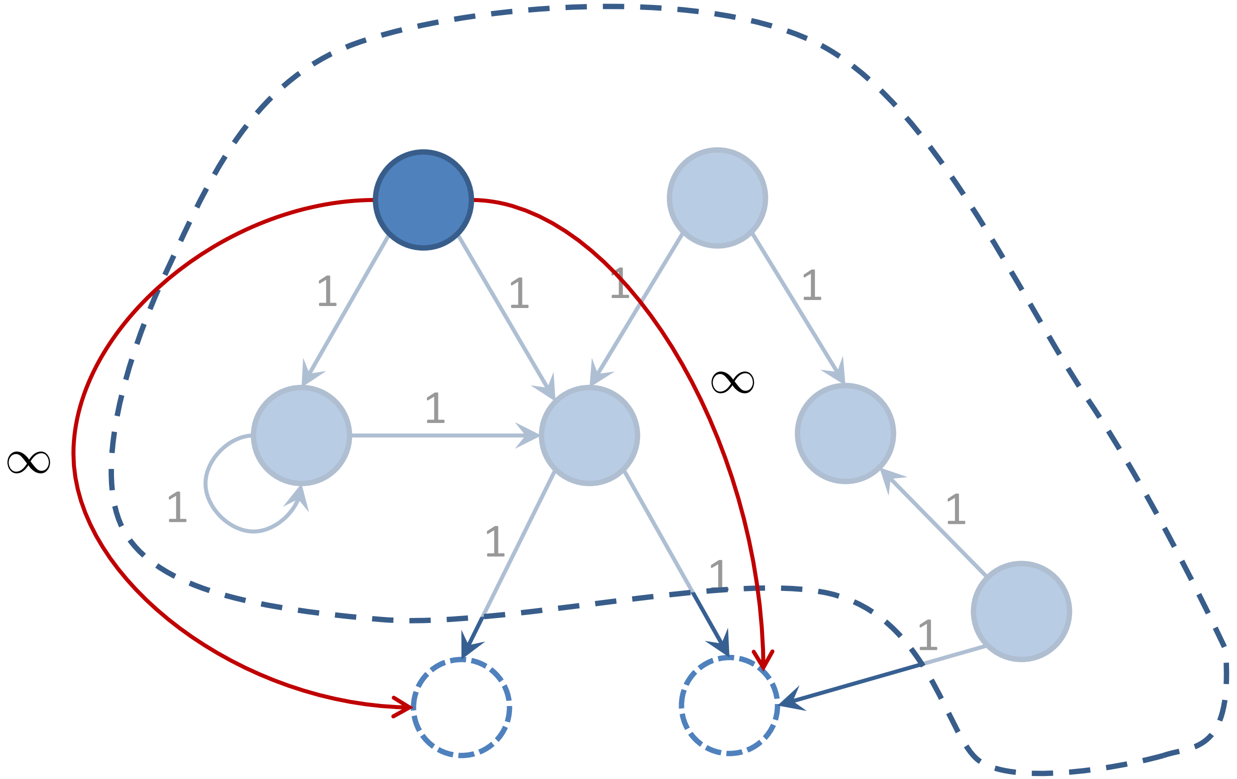
$\mathrm{fm}(G)(n, n_o) = \sum \{\, pathproduct(p) \mid p \text{ path from } n \text{ to } n_o \text{ in } G \}$

$\mathrm{flow}(in, G)(n_o) = \sum \{\, \mathrm{in}(\mathrm{n}) \cdot \mathrm{fm}(G)(n, n_o) \mid \mathrm{n} \in \mathrm{G} \}$

Flow Map: Example

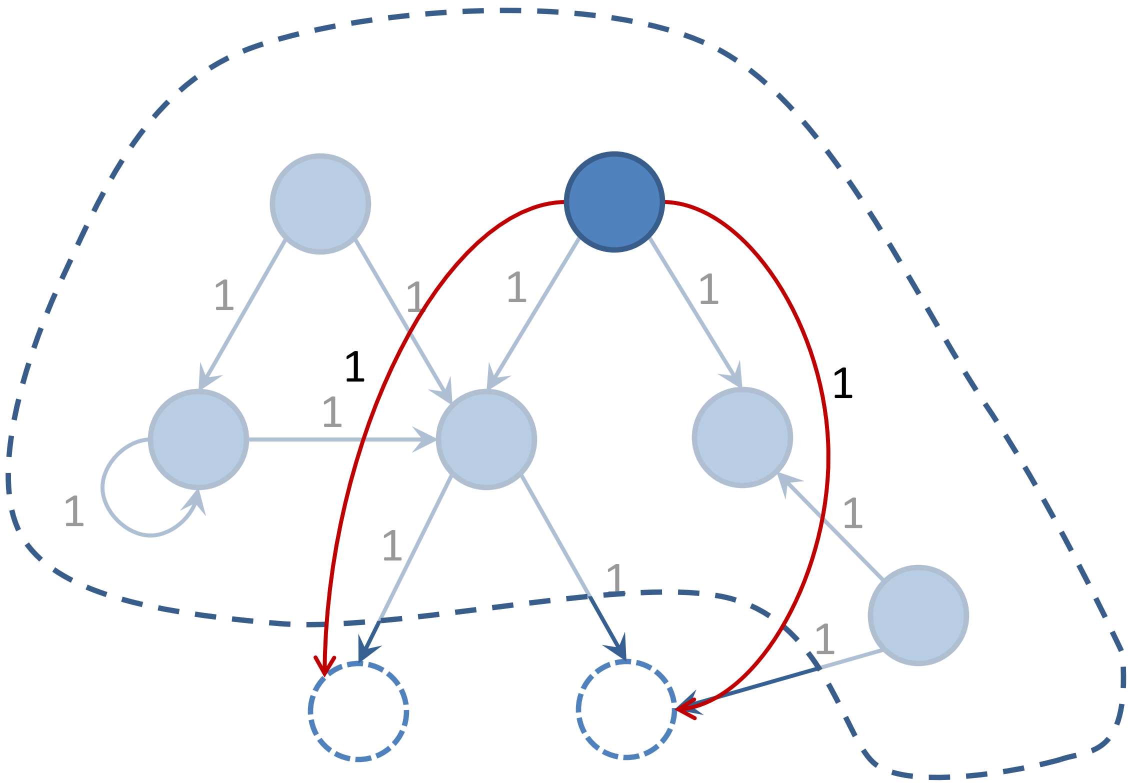


Flow Map: Example



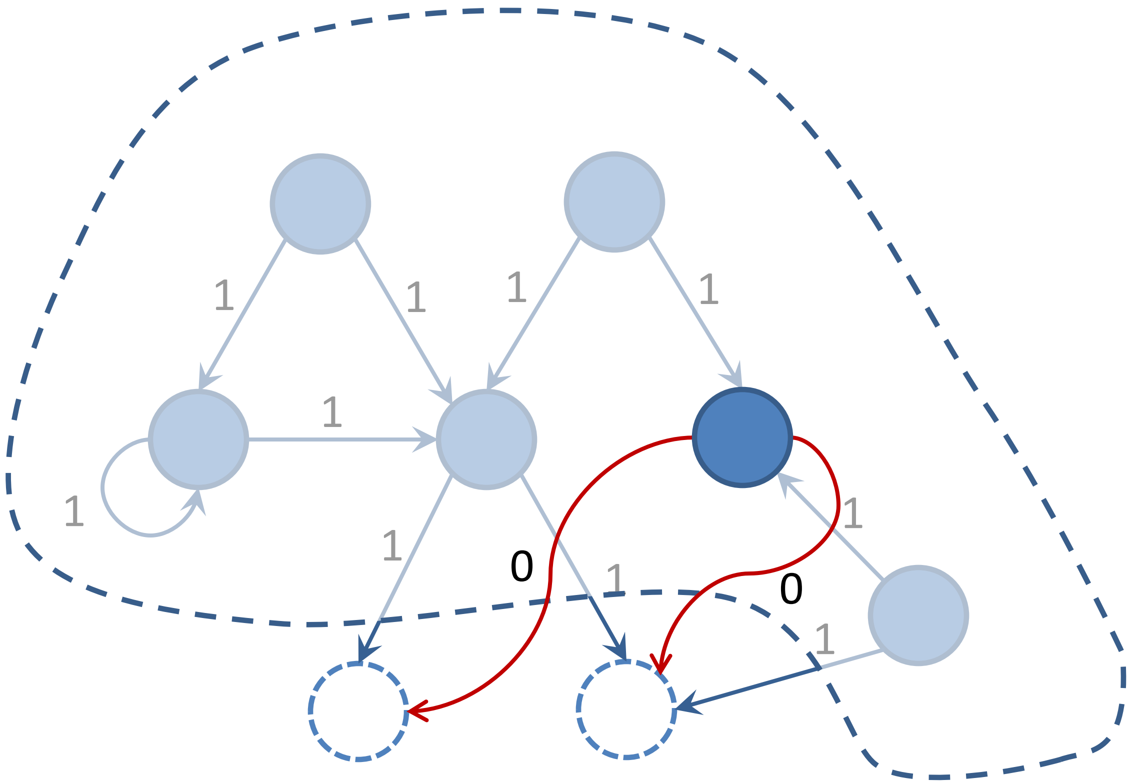
Flow map abstracts from internal structure of the graph

Flow Map: Example



Flow map abstracts from internal structure of the graph

Flow Map: Example



Flow map abstracts from internal structure of the graph

Flow Interfaces

- I = (*in*, *f*) is a *flow interface* if
  - in: N $\to$ D is an inflow
  - f: N $\times$ N_o $\to$ D is a flow map

- $[\![(in, f)]\!]_{good}$ denotes all flow interface graphs (in, G) s.t.
  - fm(G) = f
  - for all n $\in$ N *good*(*in*(n), $G|_n$) holds
- where *good* is some *good node* condition
  - e.g. *good*(i, _) = i $\leq$ 1

Flow Interfaces with Node Abstraction

- $I = (in, \alpha, f)$ is a *flow interface* if
  - in: $N \to D$ is an inflow
  - f: $N \times N_o \to D$ is a flow map
  - $\alpha \in A$ is a node label
- $[\![(in, \alpha, f)]\!]_{\text{good}}$ denotes all flow interface graphs (in, G) s.t.
  - $\text{fm}(G) = f$
  - $\alpha = \sqcup \{ \lambda_G(n) \mid n \in N \}$
  - for all $n \in N$ $good(in(n), G|_n)$ holds
- where *good* is some *good node* condition
  - e.g. $good(i, \_) = i \leq 1$

Flow Interface Composition

Composition of flow interface graphs can be lifted to flow interfaces:

- $I \in I_1 \oplus I_2$ iff $\exists$ H, H_1, H_2 such that
  - $H \in [\![I]\!]$, $H_1 \in [\![I_1]\!]$, and $H_2 \in [\![I_2]\!]$
  - $H = H_1 \bullet H_2$

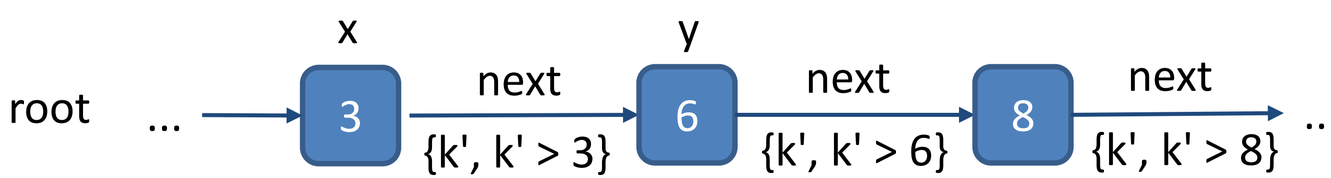
Some nice properties of $\oplus$

- $\oplus$ is associative and commutative
- $[\![I_1]\!] \bullet [\![I_2]\!] \subseteq [\![I_1 \oplus I_2]\!]$
- if $I \in I_1 \oplus I_2$, then for all $H_1 \in [\![I_1]\!]$, $H_2 \in [\![I_2]\!]$, $H_1 \bullet H_2$ defined
- …

Separation Logic with Flow Interfaces

- Good graph predicate $\mathrm{Gr}_\gamma(\mathtt{I})$
  - γ: SL predicate that defines good node condition and abstraction of heap onto nodes of flow graph
  - $\mathtt{I}$: flow interface term
- Good node predicate $\mathrm{N}_\gamma(\mathrm{x}, \mathtt{I})$
  - like Gr but denotes a single node
  - definable in terms of Gr
- Dirty region predicate $[\mathrm{P}]_{\gamma,\mathtt{I}}$
  - P: SL predicate
  - denotes heap region that is expected to satisfy interface $\mathtt{I}$ but may currently not

Graph Predicate: Linked List



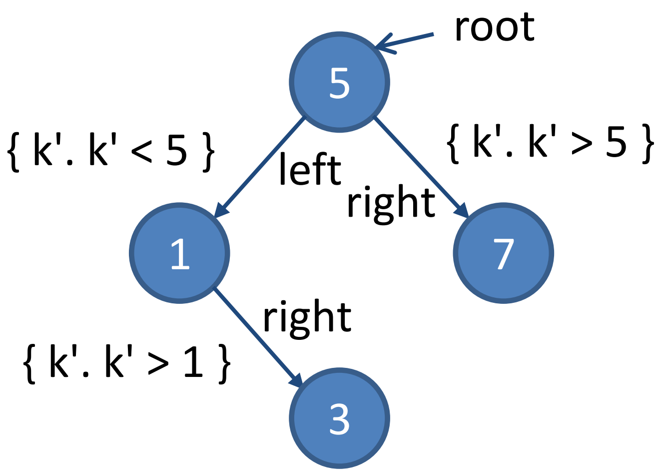
- Abstraction of linked list node

$$\begin{aligned}\gamma(x, in, C, f) = \exists k, y.\ x \mapsto (\text{data: } k, \text{next: } y) \land\ & \\ C = \{k\} \land k \in in \land\ & \\ f = \text{ITE}(\text{y = null}, \epsilon, \{ (x,y) \mapsto \{k'.\ k' > k\} \})&\end{aligned}$$

- Invariant

$$\exists \mathtt{I} :: \text{Gr}_\gamma(\mathtt{I}) \land \mathtt{I}^{in} = \{\text{root} \mapsto \text{KS}\}.\mathbf{0} \land \mathtt{I}^f = \epsilon$$

Graph Predicate: Binary Search Tree



- Abstraction of BST node

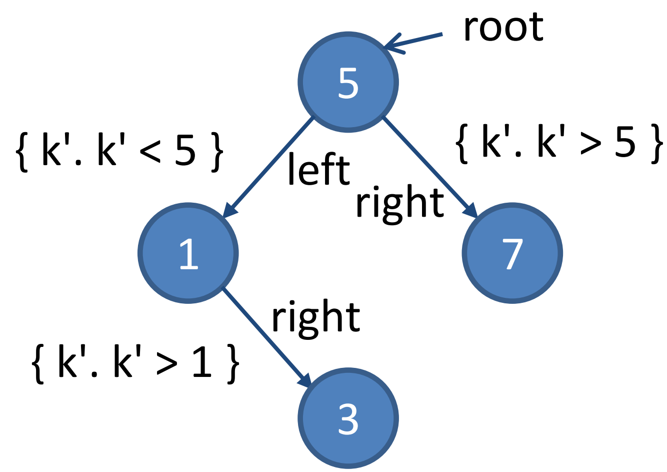
$$\gamma(x, in, C, f) = \exists k, y, z.\ x \mapsto (\text{data: } k, \text{left: } y, \text{right: } z) \wedge$$
$$C = \{k\} \wedge k \in in \wedge$$
$$f = \mathrm{ITE}(y = \text{null}, \epsilon, \{ (x,y) \mapsto \{k'.\ k' < k\} \}.$$
$$\mathrm{ITE}(z = \text{null}, \epsilon, \{ (x,z) \mapsto \{k'.\ k' > k\} \}$$

- Invariant

$$\exists \mathtt{I} :: \mathrm{Gr}_\gamma(\mathtt{I}) \wedge \mathtt{I}^{in} = \{\text{root} \mapsto \mathrm{KS}\}.\mathbf{0} \wedge \mathtt{I}^{f} = \epsilon$$

Graph Predicate: Binary Search Tree

Need tree invariant?



- Abstraction of BST node

$$\begin{aligned}\gamma(x, in, C, f) = \exists k, y, z.\ x \mapsto (\text{data: } k, \text{left: } y, \text{right: } z) \wedge \\ C = \{k\} \wedge k \in in \wedge \\ f = \text{ITE}(\text{y} = \text{null}, \epsilon, \{ (x,y) \mapsto \{k'.\ k' < k\} \}. \\ \text{ITE}(\text{z} = \text{null}, \epsilon, \{ (x,z) \mapsto \{k'.\ k' > k\} \}\end{aligned}$$

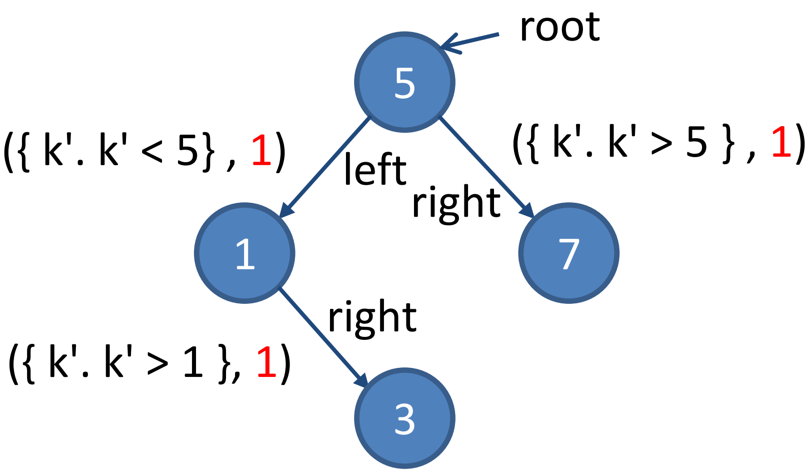
- Invariant

$$\exists \mathrm{I} :: \mathrm{Gr}_\gamma(\mathrm{I}) \wedge \mathrm{I}^{in} = \{\text{root} \mapsto \text{KS}\}.\mathbf{0} \wedge \mathrm{I}^{f} = \epsilon$$

Graph Predicate: Binary Search Tree

Need tree invariant?

No problem!



- Abstraction of BST node

$$\begin{aligned}\gamma(x, (in, pc), C, f) = \exists k, y, z.\ & x \mapsto (\text{data: } k, \text{left: } y, \text{right: } z) \wedge \\ & C = \{k\} \wedge k \in in \wedge\ pc = 1 \wedge \\ & f = \text{ITE}(y = \text{null}, \epsilon, \{ (x,y) \mapsto (\{k'.\ k' < k\}, 1) \}. \\ & \quad \text{ITE}(z = \text{null}, \epsilon, \{ (x,z) \mapsto (\{k'.\ k' > k\}, 1) \}\end{aligned}$$

- Invariant

$$\exists \mathtt{I} :: \mathrm{Gr}_\gamma(\mathtt{I}) \wedge \mathtt{I}^{in} = \{\text{root} \mapsto (\text{KS}, 1)\}.\mathbf{0} \wedge \mathtt{I}^{f} = \epsilon$$

Data-Structure-Agnostic Proof Rules

Decomposition

$$\frac{\mathrm{Gr}(\mathtt{I}) \wedge x \in \mathtt{I}^{\mathrm{in}}}{\mathrm{N}(x, \mathtt{I}_1) * \mathrm{Gr}(\mathtt{I}_2) \wedge \mathtt{I} \in \mathtt{I}_1 \oplus \mathtt{I}_2}$$

Abstraction

$$\frac{\mathrm{Gr}(\mathtt{I}_1) * \mathrm{Gr}(\mathtt{I}_2) \wedge \mathtt{I} \in \mathtt{I}_1 \oplus \mathtt{I}_2}{\mathrm{Gr}(\mathtt{I}) \wedge \mathtt{I} \in \mathtt{I}_1 \oplus \mathtt{I}_2}$$

Replacement

$$\frac{\mathtt{I} \in \mathtt{I}_1 \oplus \mathtt{I}_2 \wedge \mathtt{I}_1 \precsim \mathtt{J}_1}{\mathtt{J} \in \mathtt{J}_1 \oplus \mathtt{I}_2 \wedge \mathtt{I} \precsim \mathtt{J}}$$

Generic R/G Actions

- Lock node $N(x, (in, 0, f)) \rightarrow N(x, (in, T, f))$
- Unlock node $N(x, (in, T, f)) \rightarrow N(x, (in, 0, f))$
- Dirty $[\mathit{true}]_{\mathtt{I}} \wedge \mathtt{I}^{\alpha} = t \rightarrow [\mathit{true}]_{\mathtt{I}}$
- Sync $[\mathit{true}]_{\mathtt{I}} \wedge \mathtt{I}^{\alpha} = t \rightarrow Gr(\mathtt{I}') \wedge \mathtt{I} \precsim \mathtt{I}'$

Conclusion

- Radically new approach for building compositional abstractions of data structures.
- Fits in existing (concurrent) separation logics.
- Enables simple correctness proofs of concurrent data structure algorithms
- Proofs abstract from the details of the specific data structure implementation.