Flow Interfaces

Compositional Abstractions of Concurrent Data Structures

Siddharth Krishna,
Dennis Shasha, and Thomas Wies

NYU | COURANT

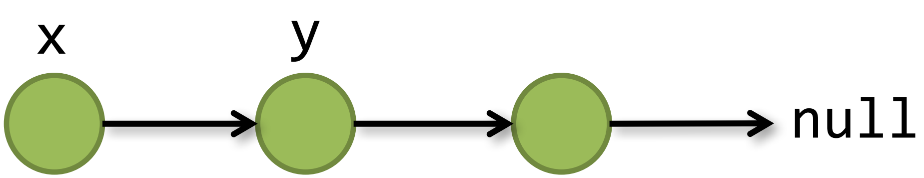
Background

Verifying programs, separation logic, inductive predicates

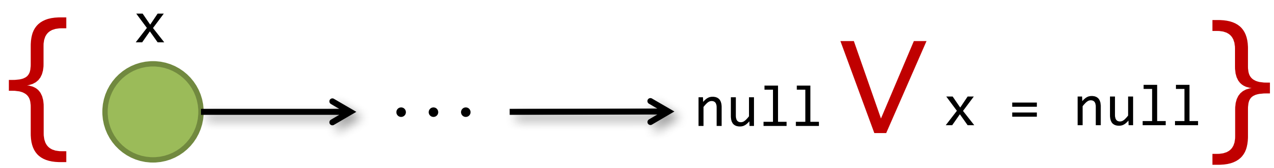
Slides courtesy of Thomas Wies

Verifying Data Structures

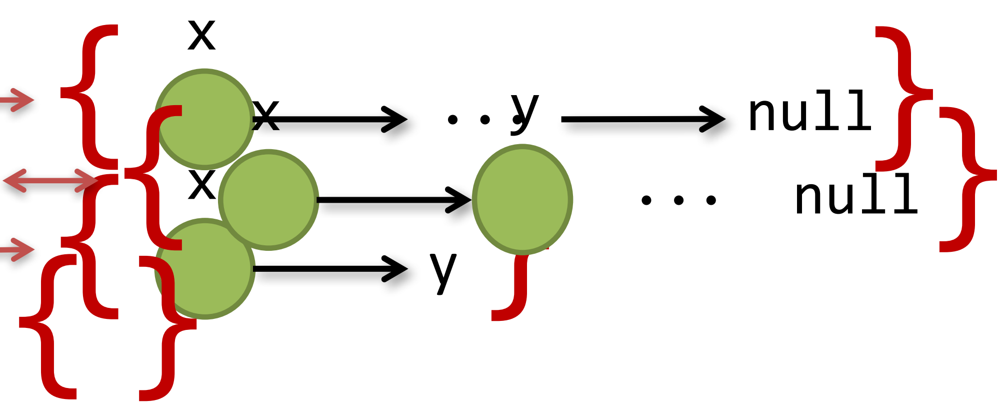
```
procedure delete(x: Node)
{
  if (x != null) {
    var y := x.next;
    delete(y);
    free(x);
  }
}
```



Proof by Hand-Waving

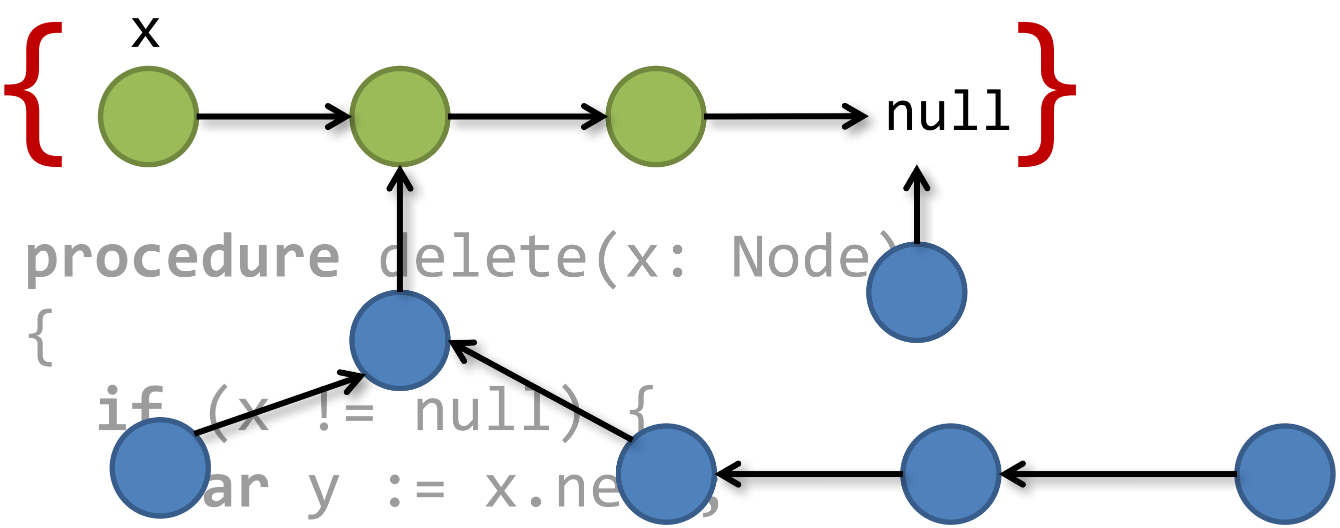



```
procedure delete(x: Node)
{
  if (x != null) {
    var y := x.next;
    delete(y);
    free(x);
  }
}
```



{ }

Proof by Hand-Waving



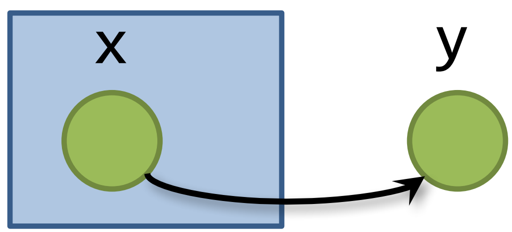

```
procedure delete(x: Node)
{
  if (x != null) {
    var y := x.next;
    delete(y);
    free(x);
  }
}
```

{ }

Separation Logic by Example

- Points-to predicate

$$x \mapsto y$$



Stack

x	10
y	42
…	

Heap

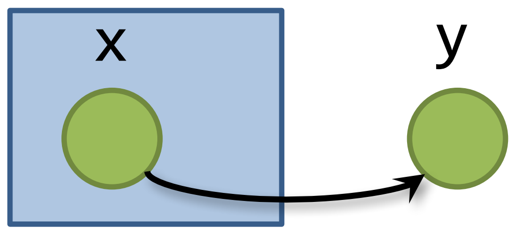
10	42
…	
42	?

A partial heap consisting of one allocated cell

Separation Logic by Example

- Points-to predicate

$$x \mapsto y$$



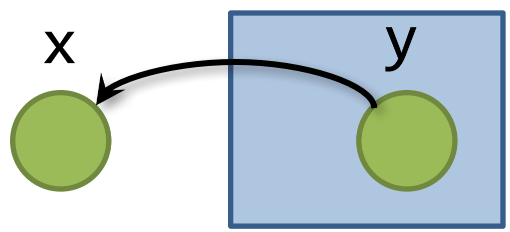
Points-to predicate expresses permission to access (i.e. read/write/deallocate) heap location x and nothing else!

SL assertions describe the part of the heap that a program is allowed to work with.

Separation Logic by Example

- Points-to predicate

$$y \mapsto x$$



Stack

x	10
y	42
…	

Heap

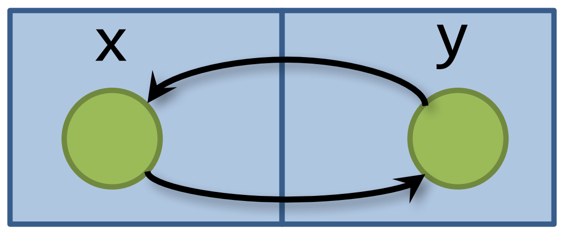
10	?
…	
42	10

A partial heap consisting of one allocated cell

Separation Logic by Example

- Separating conjunction

$$x \mapsto y \ * \ y \mapsto x$$



Stack

x	10
y	42
…	

Heap

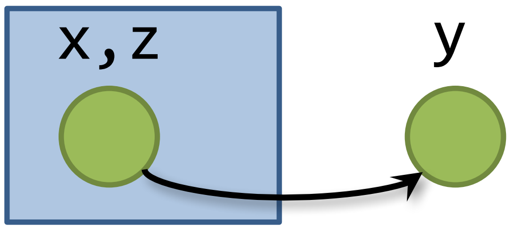
10	42
…	
42	10

Composition of disjoint partial heaps

Separation Logic by Example

- Equalities

$$x \mapsto y \wedge x = z$$



Stack

x	10
y	42
z	10

Heap

10	42
…	
42	?

Equalities only constrain the stack

Separation Logic by Example

- Separating conjunction

$$x \mapsto y \ * \ x \mapsto z$$

?

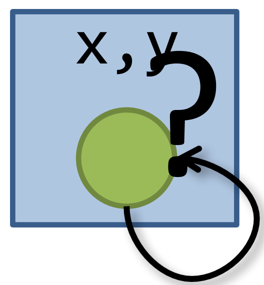
unsatisfiable

Subheaps must be disjoint
(x can't be at two different places at once)

Separation Logic by Example

- Classical conjunction

$$x \mapsto y \wedge y \mapsto x$$



Separation Logic by Example

- Separating conjunction

$$x \mapsto z_1 * y \mapsto z_2 \wedge x = y$$

?

still unsatisfiable

Convention: $\wedge$ has higher precedence than $*$

Inductive Predicates

$$ls(x) \stackrel{\text{def}}{=} x = null \wedge emp \ \vee \ \exists y.\, x \mapsto y \ * ls(y)$$

$$\begin{aligned}
ls(x) &\rightsquigarrow \exists y. && x \mapsto y \ * ls(y) \\
&\rightsquigarrow \exists y, z. && x \mapsto y \ * y \mapsto z \ * ls(z) \\
&\rightsquigarrow \exists y, z, w. && x \mapsto y \ * y \mapsto z \ * z \mapsto w \ * ls(w) \\
&\rightsquigarrow \exists y, z, w. && x \mapsto y \ * y \mapsto z \ * z \mapsto w \ * w = null \wedge emp \\
&\rightsquigarrow \exists y, z, w. && x \mapsto y \ * y \mapsto z \ * z \mapsto null
\end{aligned}$$

Proof by SL

$\{ls(x)\}$

```
procedure delete(x: Node)
{
  if (x != null) {
    var y := x.next;
    delete(y);
    free(x);
  }
}
```

$\{emp\}$

$\{ls(x) \wedge x \neq null\}$

$\{x \mapsto y * ls(y)\}$

$\{x \mapsto y * emp\}$

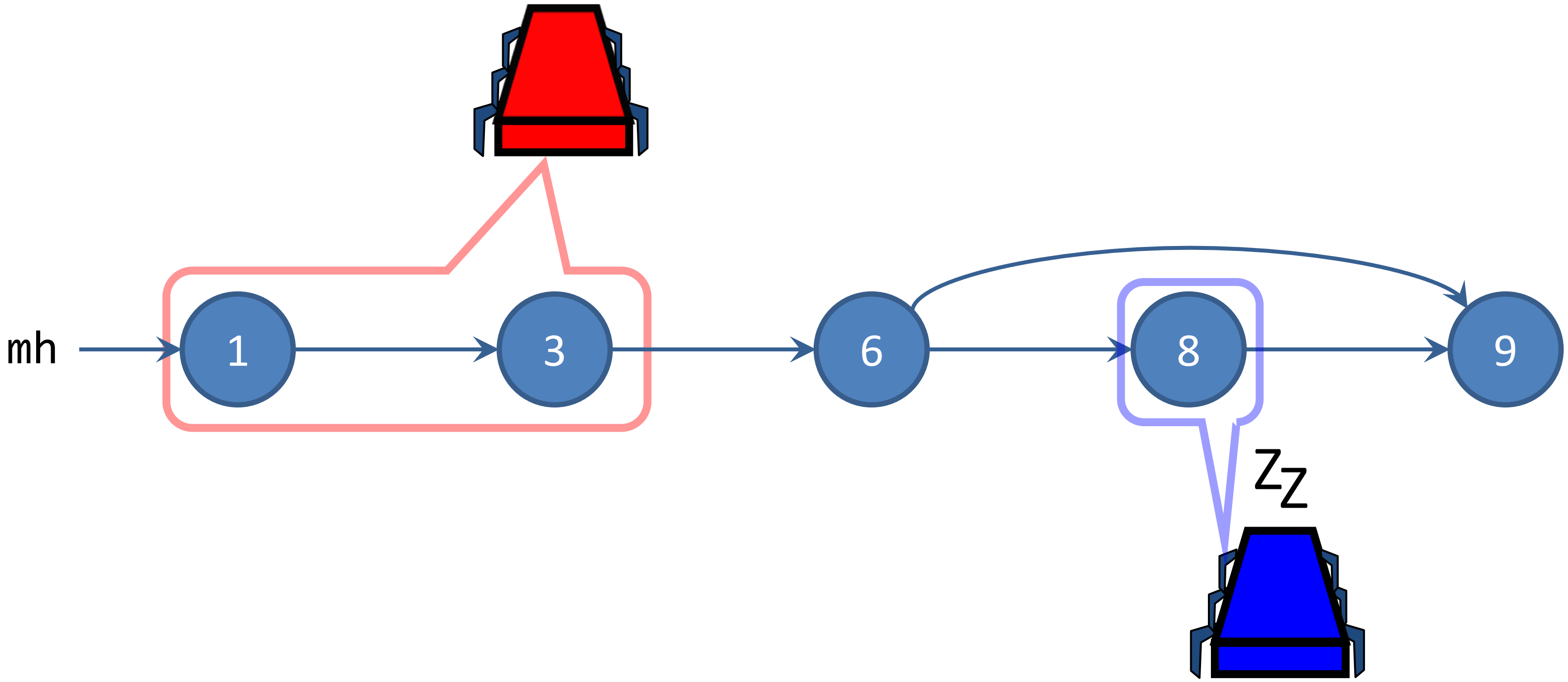
$\{emp\}$

Background

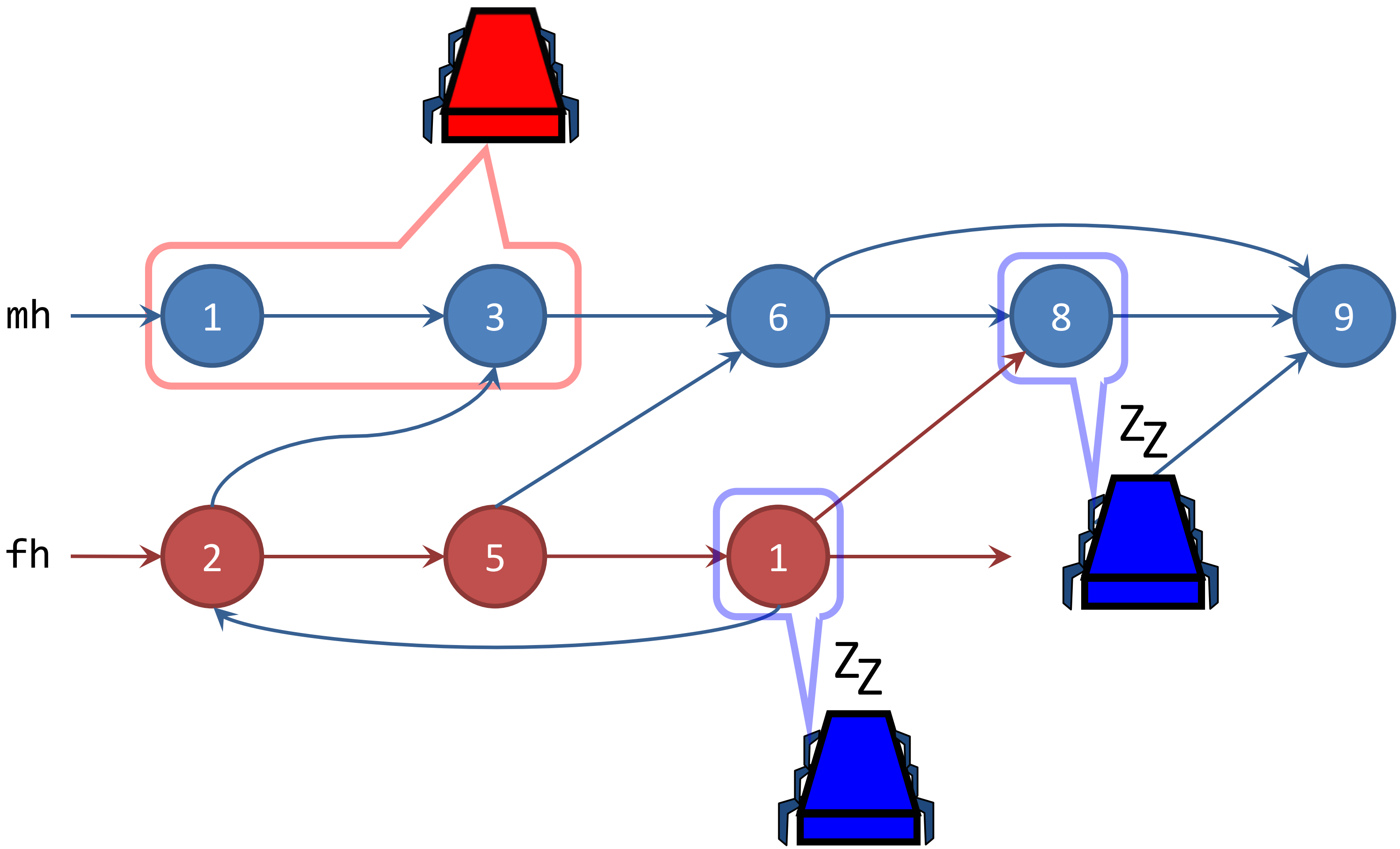
The Trouble with Inductive Predicates

Concurrent data structures are complex

Harris' Non-blocking List



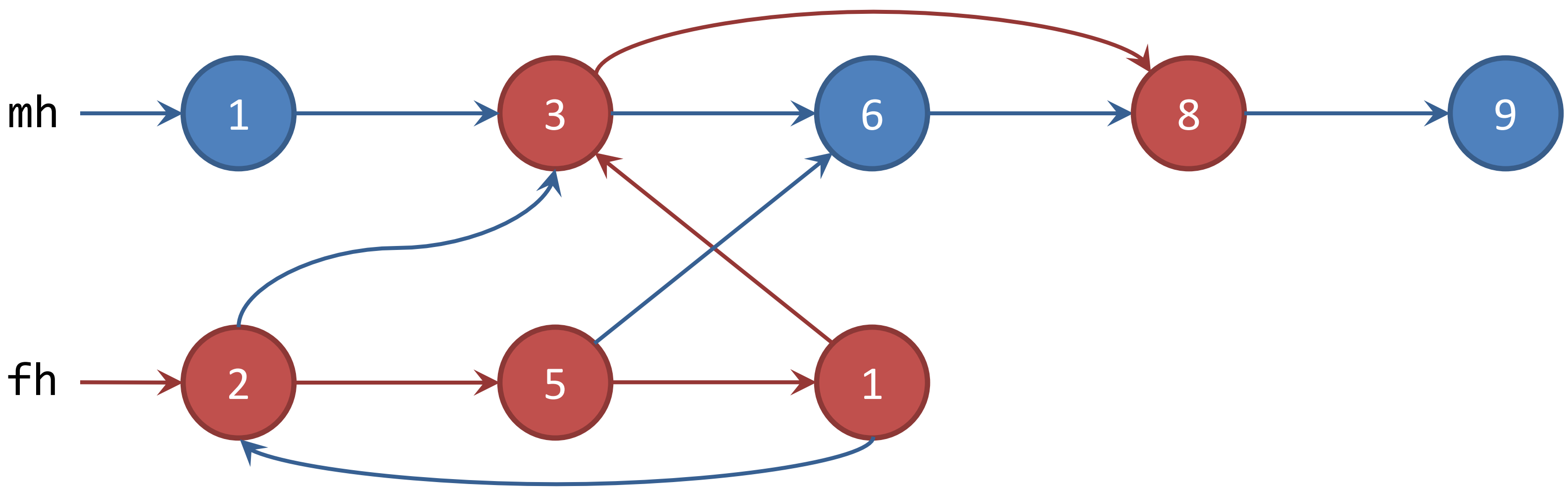
Harris' Non-blocking List



Limitations of Inductive Predicates

Traversals need to visit each node exactly once.

$$ls(mh, null) * ls(fh, null)$$

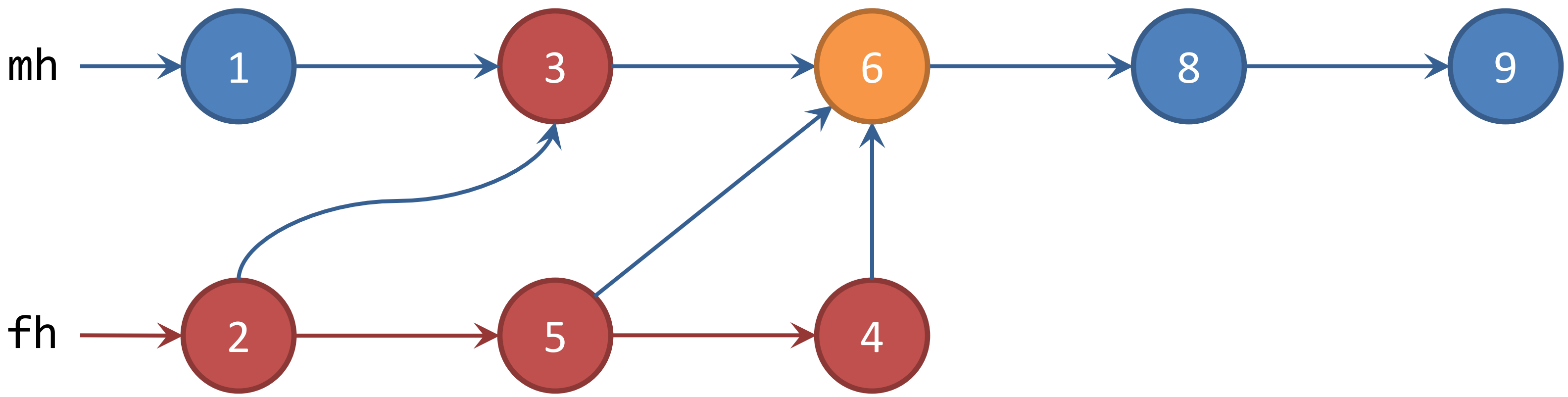


Overlays

Limitations of Inductive Predicates

Traversals need to visit each node exactly once.

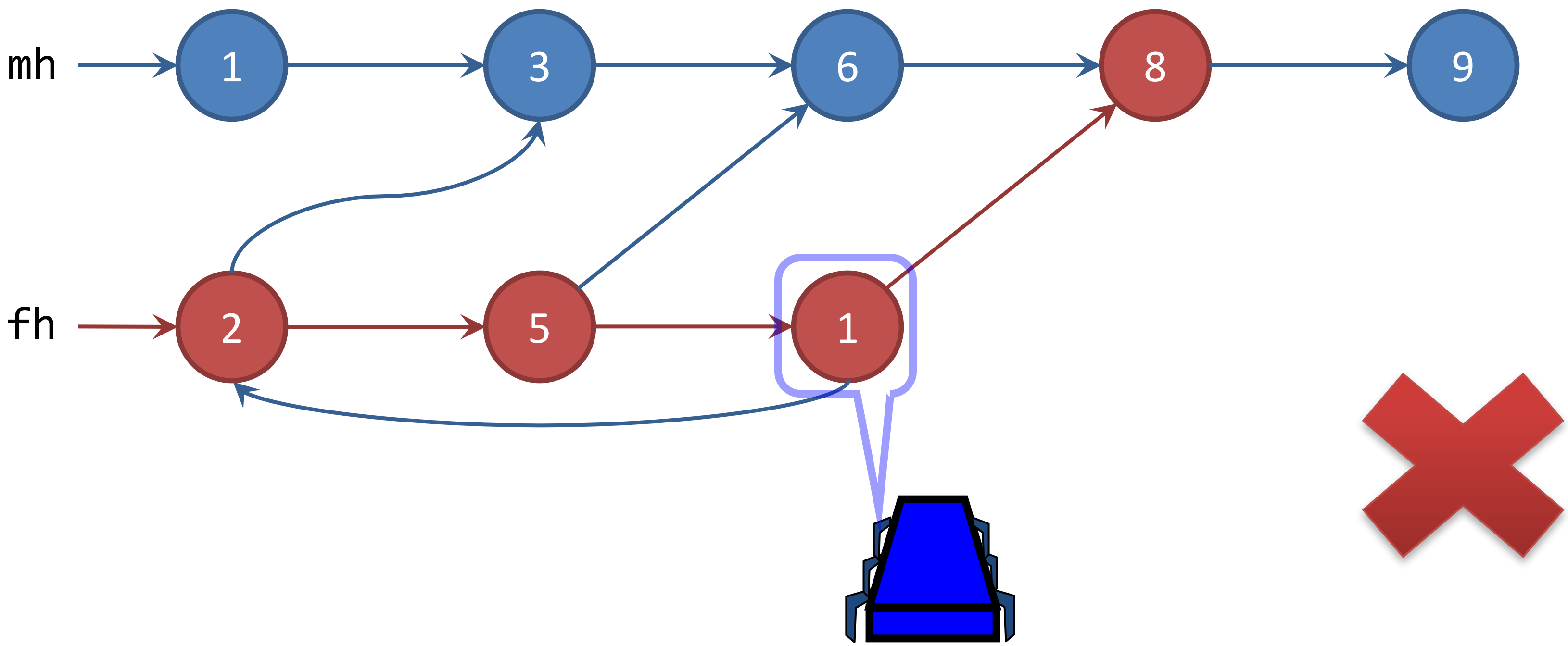
$$harris(mh, fh, null) \stackrel{\text{def}}{=} \ldots * harris(\ldots)$$



Unbounded sharing

Limitations of Inductive Predicates

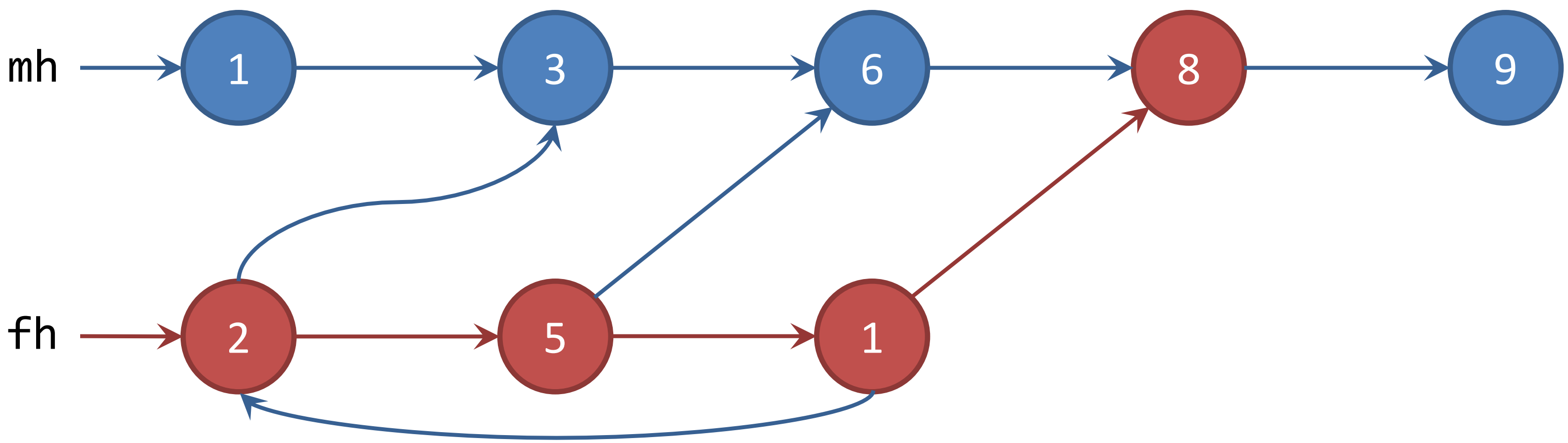
$$harris(mh, fh, null) \stackrel{\text{def}}{=} ls(mh, null) * \cdots$$



Threads can enter main list at arbitrary points

Other Approaches

Overlapping conjunction: $\mathbin{\cup\!\!\!\!\!*}$

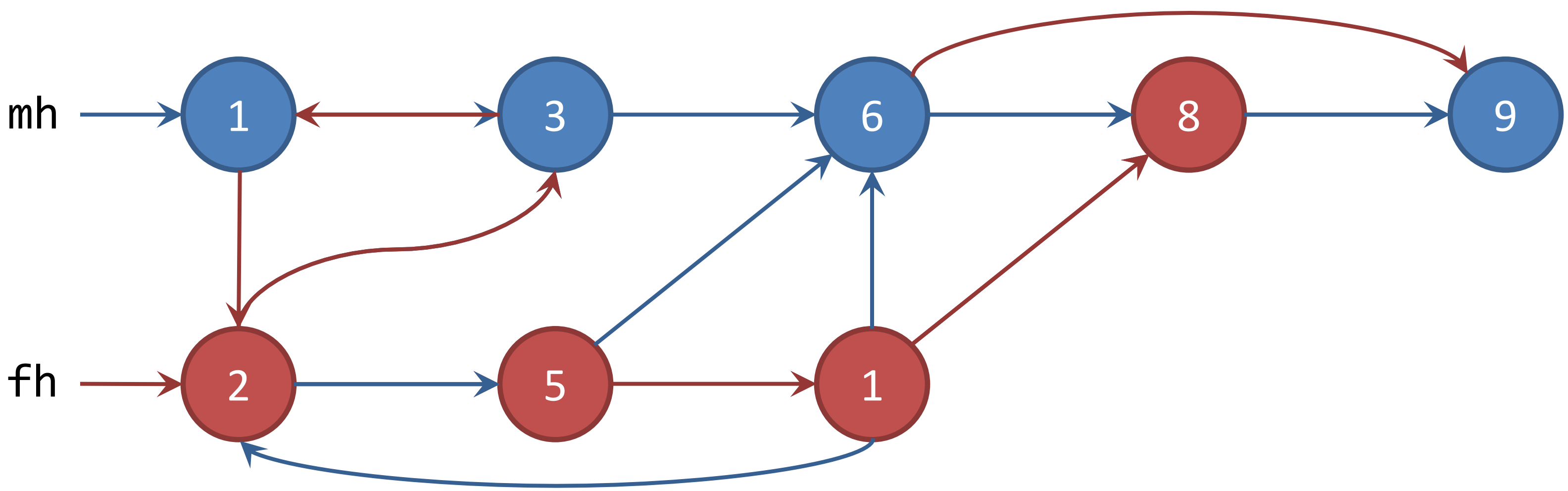


Potentially cyclic ⇒ complex predicate

Updates: ramifications, reason about $-\!\!*$

Other Approaches

Iterated separating conjunction: $\circledast_{x \in X}\ \phi(x)$



Can’t do better than closed set of nodes

Every node reachable – memory leaks

Other Issues

Entailment lemmas are specific to data structure.

Example:

$$ls(x, y) \ * \ ls(y, z) \Rightarrow ls(x, z)$$

but

$$sls(x, y) \ * sls(y, z) \nRightarrow sls(x, z)$$

Solution: add lower and upper bounds $sls(x, y, l, u)$

$$sls(x, y, l, v) \ * sls(y, z, w, u) \wedge v \leq w \Rightarrow sls(x, z, l, u)$$

Different composition lemma!

The Problem

An abstraction mechanism that can handle data structures like the Harris list?

(i.e. handle overlays, sharing, arbitrary traversals & have easy reasoning)

Background

The Trouble with Inductive Predicates

Flow Interfaces

A new abstraction for concurrent data structures

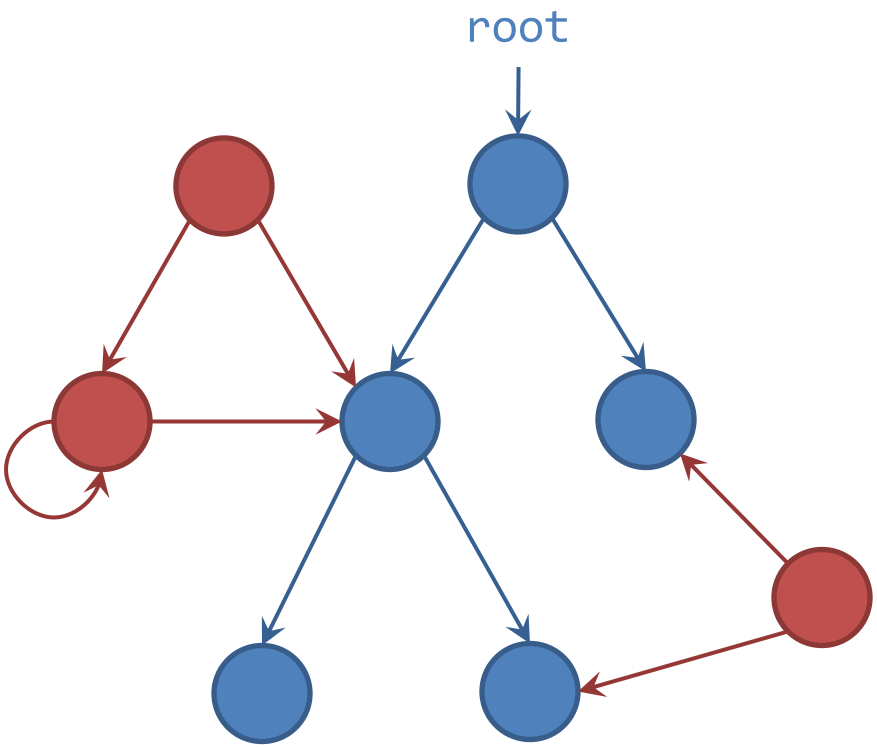
The Idea

- Inductive predicates:
  - Pro: inductive properties
  - Con: fixed traversals
- Iterated $*$: $\circledast_{x \in X} \phi(x)$
  - Pro: easy reasoning
  - Con: only local properties
- Best of both?
- Inductive properties $\rightarrow$ local conditions
- But allow dependence on inductive quantity: flow

The Idea

- Express all invariants w.r.t. local condition on nodes and flow.
- Introduce a graph composition that preserves local invariants on global flows.
- Introduce a generic predicate to abstract a heap region satisfying the local flow condition (the flow interface).

Local Data Structure Invariants with Flows

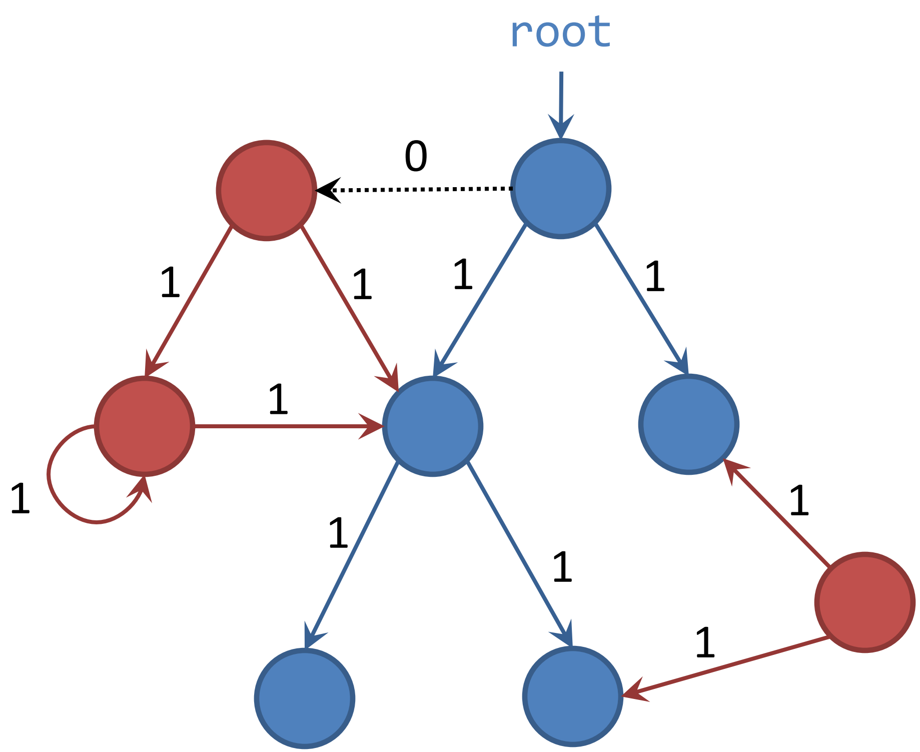


Path counting!

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

Flows

Step 1: Define the graph



Graph G = (N, e)

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

D is a flow domain

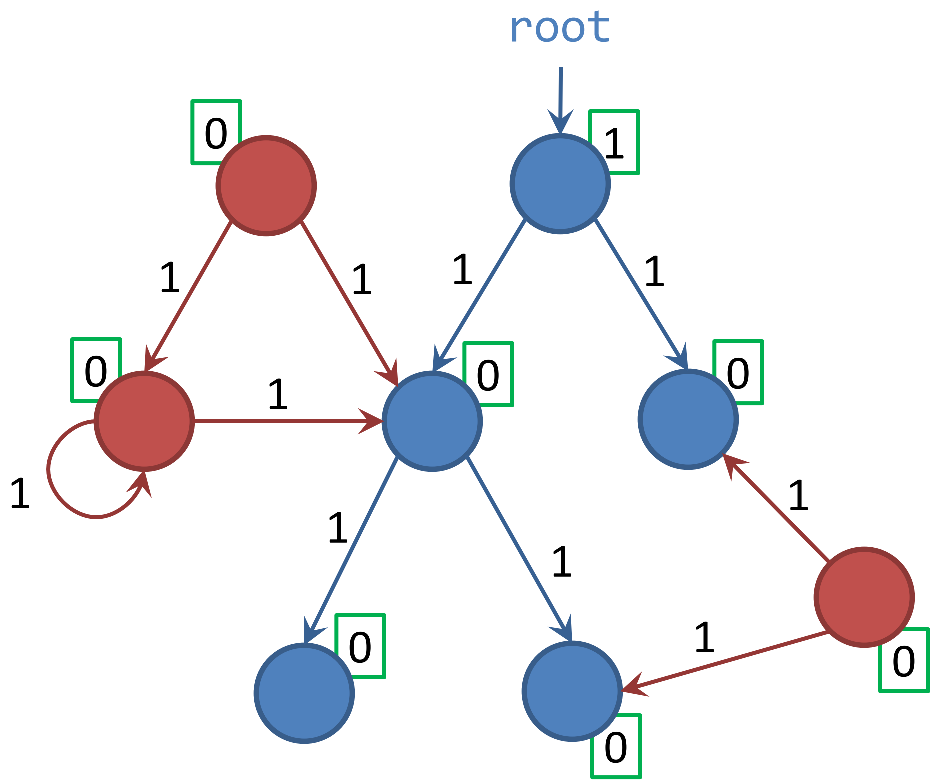
For example:

$$D = \mathbb{N} \cup \{\infty\}$$

Label each edge in the graph with 1.

Flows

Step 2: Calculate the flow

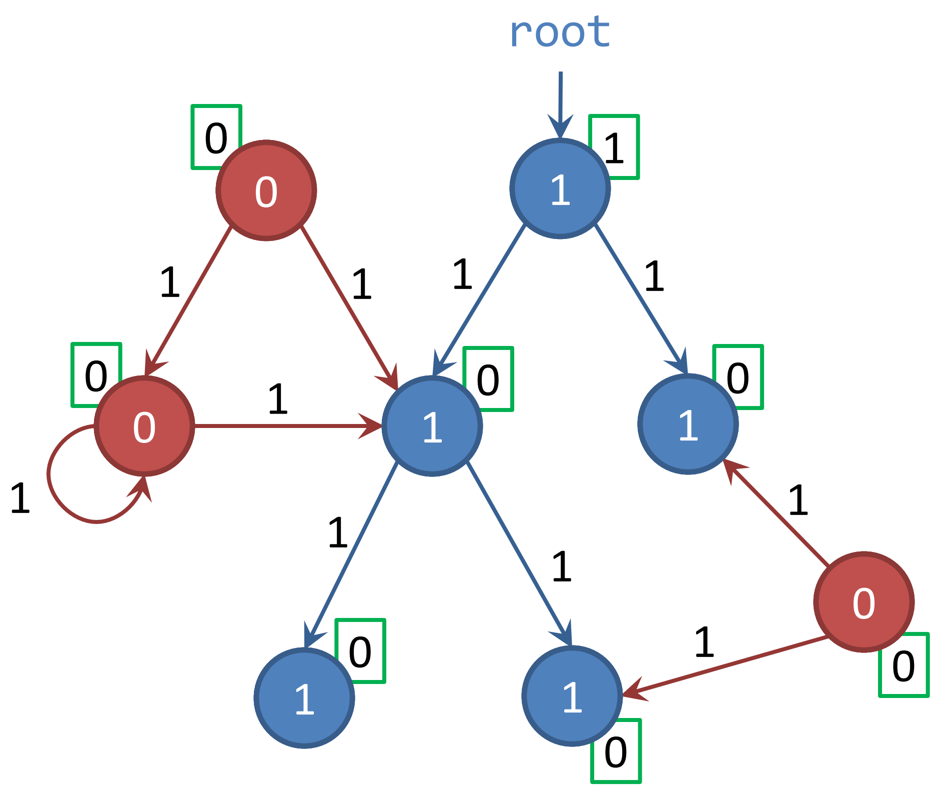


Given an inflow *in*: N → D

Here
$in(n) = ITE(n = \texttt{root}, 1, 0)$

Flows

Step 2: Calculate the flow



Start with

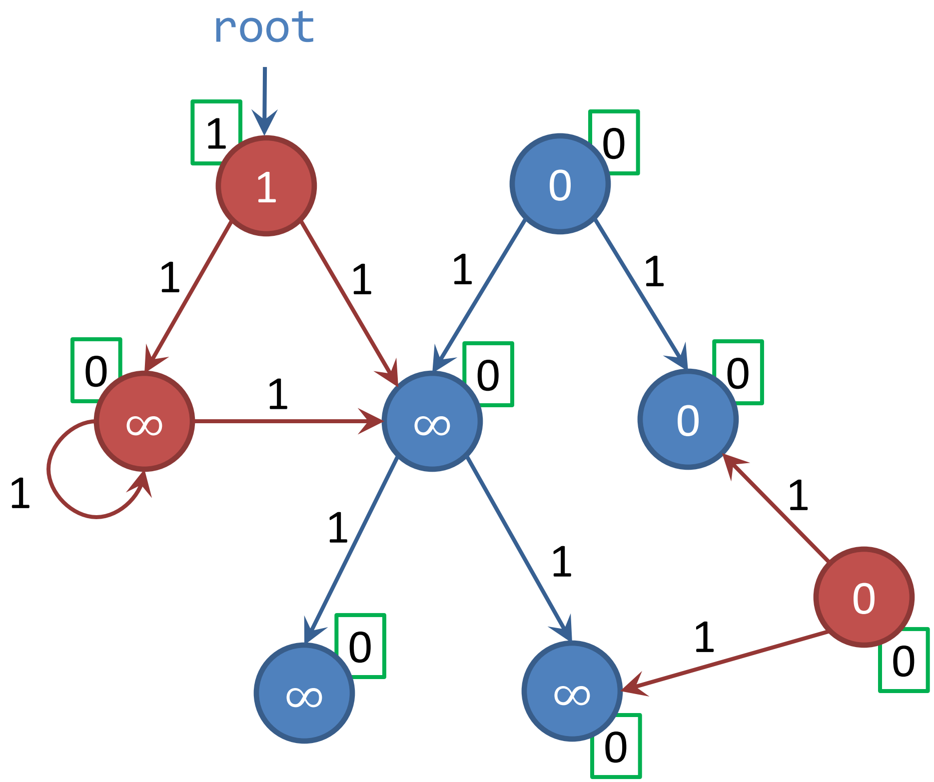
$$\text{flow}(n) \;=\; in(n)$$

And iterate until fixpoint

$$\text{flow}(n) = in(n) + \textstyle\sum_{n'} \text{flow}(n') \cdot e(n', n)$$

Flows

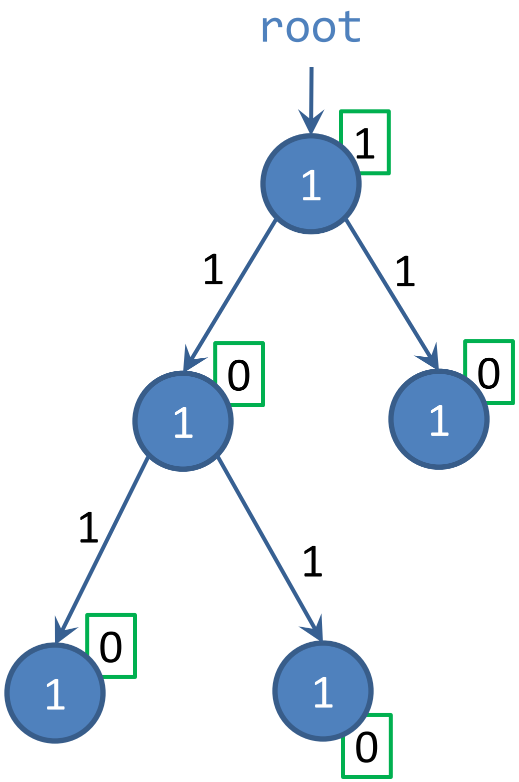
Step 2: Calculate the flow



Different inflows result in different flows

Flows

Step 3: Define invariant on flow



If every node satisfies the good condition

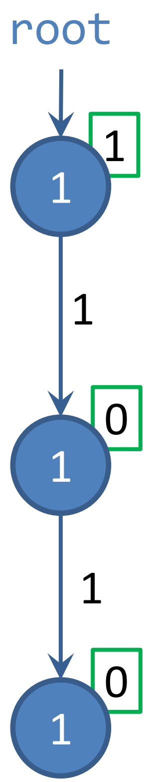
$$\gamma_t \stackrel{\text{def}}{=} \text{flow}(in, G)(n) = 1$$

for $in(n) = ITE(n = \texttt{root}, 1, 0)$

then G is "a tree rooted at `root`"

Flows

Step 3: Define invariant on flow



For lists, enforce at most 1 outgoing edge

$$\gamma_l \stackrel{\text{def}}{=} \mathsf{flow}(in, G)(n) = 1 \\ \wedge \left(e_n = \{(n, n') \rightarrowtail 1\} \vee e_n = \epsilon\right)$$

e_n: edge function restricted to nonzero edges leaving n

Flows
Alternate view

Flow graph G = (N, E)
Viewing E as a matrix,

${E^2}_{nn'} = \sum_m E_{nm} E_{mn'} =$ number of 2-length paths from n to n'

$E^l_{nn'} =$ number of l-length paths from n to n'

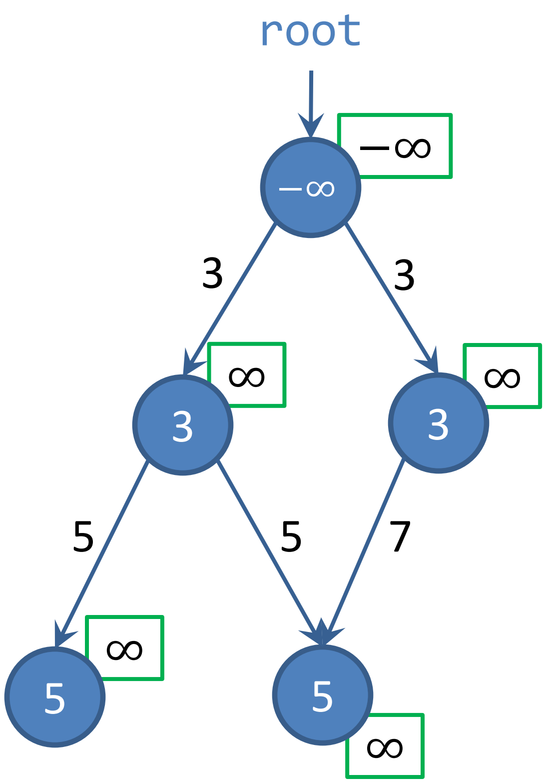
Capacity $C = I + E + E^2 + \cdots$

(converges if D is a flow domain)

$cap(G)(n, n') \stackrel{\text{def}}{=} C_{nn'}$ = the number of paths from n to n'

flow$(in, G)(n) \stackrel{\text{def}}{=} \sum_{n'} in(n') \cdot cap(G)(n', n)$

Data Invariants

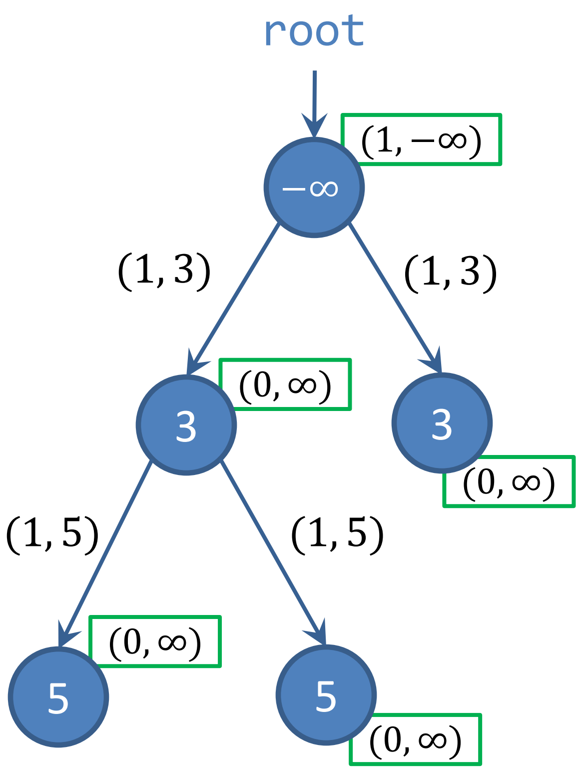


- Flow domain: ω-cpo & positive semiring
$$(D, \sqsubseteq, \sqcup, +, \cdot, 0, 1)$$
- Another Example: lower-bound domain
$$(\mathbb{Z} \cup \{-\infty, \infty\}, \geq, \min, \min, \max, \infty, -\infty)$$

- Label each edge with value of souce node
- Use $in(n) = ITE(n = \texttt{root}, -\infty, \infty)$
- flow$(in, G)(n)$ = max value in *some* path from root to n
- These paths are sorted if
$$\gamma_s \stackrel{\text{def}}{=} \text{flow}(in, G)(n) \leq a$$

where a is value at n

Stacking Flows



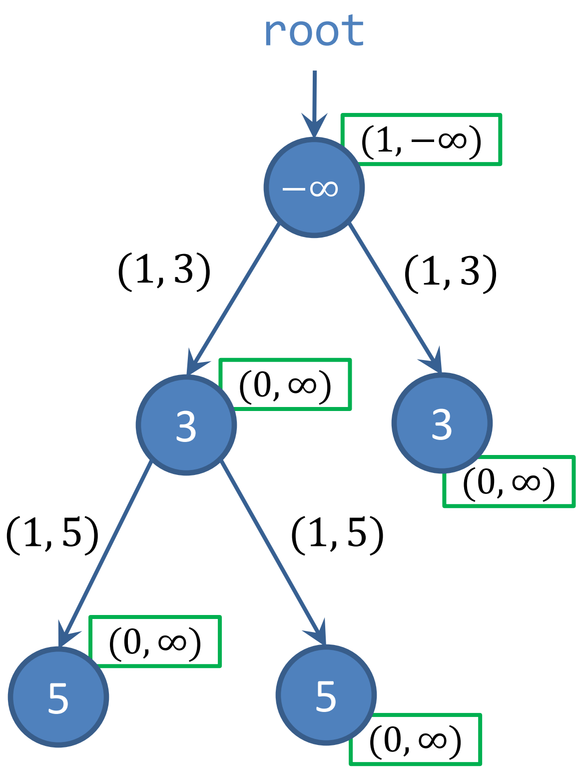
- Want shape & data invariants?
- Flow of product is product of flows!

- Example: min-heap
- Use product of path-counting and lower-bound domain
- Use $in(n) = ITE(n = \texttt{root}, (1, -\infty), (0, \infty))$
- And good condition

$$\gamma_h \stackrel{\text{def}}{=} \text{flow}(in, G)(n) = (1, l) \land l \leq a$$

where a is value at n

Stacking Flows



- Data invariants are decoupled from shape invariants

- Min-heap

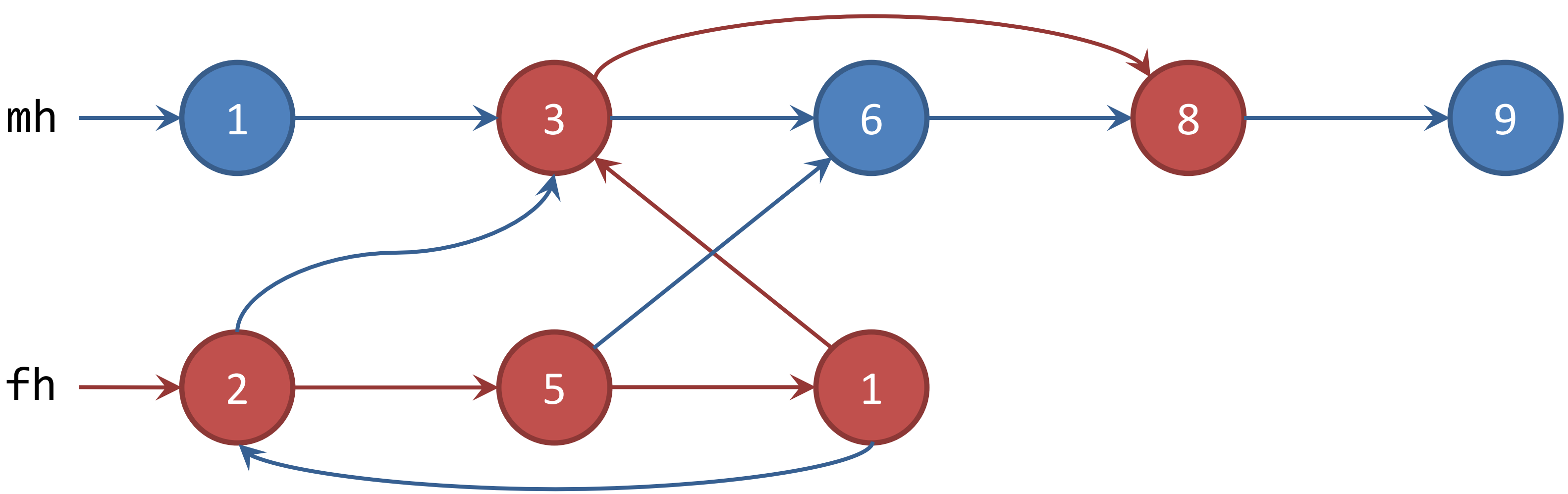
  $$\gamma_h \stackrel{\text{def}}{=} \text{flow}(in, G)(n) = (1, l) \land l \leq a$$

- Sorted list

  $$\gamma_{sl} \stackrel{\text{def}}{=} \text{flow}(in, G)(n) = (1, l) \land l \leq a \land (e_n = \{(n, n') \rightarrowtail 1\} \lor e_n = \epsilon)$$

Harris List

- We can now describe Harris' List
- Flow domain: two path-counting flows
  - One from mh and one from fh
  - Every node is on at least one of these lists
- Nodes labelled: marked/unmarked
  - All nodes in free list are marked



Expressivity of Flows

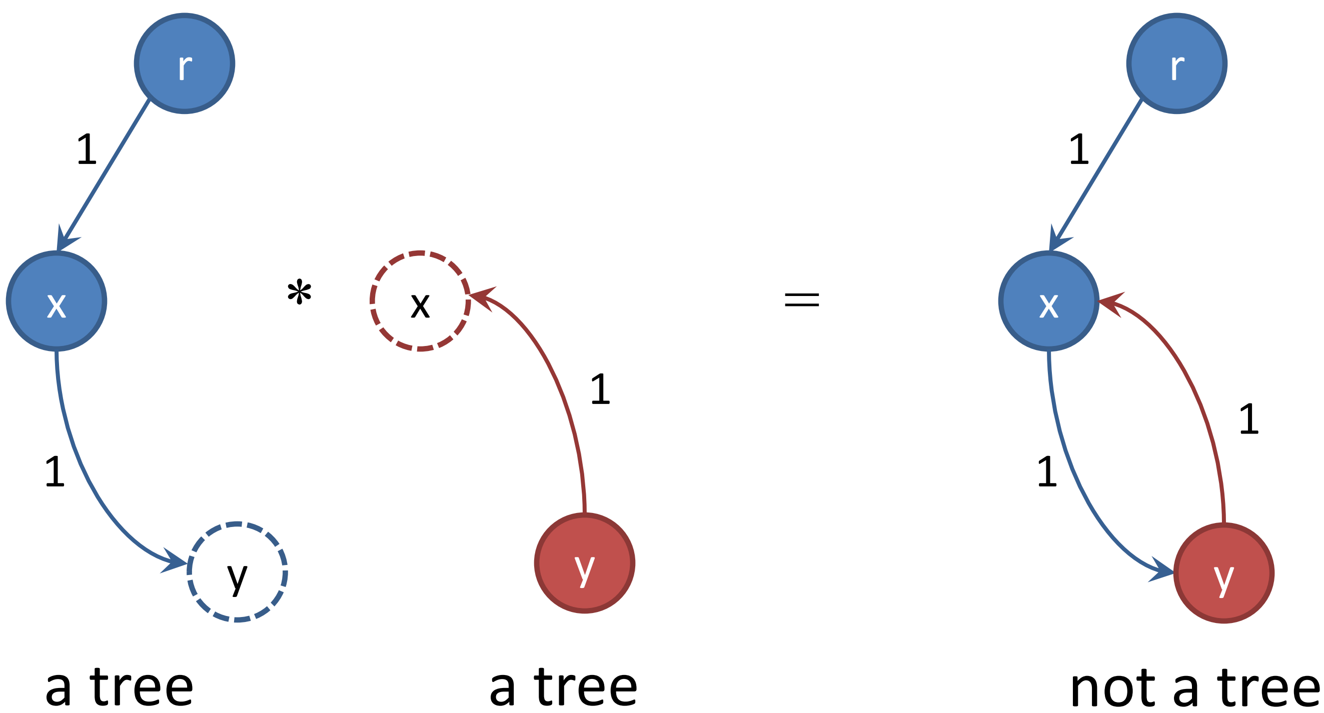
- Flows can describe:
  - Lists (singly and doubly linked, cyclic)
  - Trees (n-ary, arbitrary arity)
  - Nested combinations
  - Sorted lists, binary heaps, BSTs
  - Overlaid structures (threaded and B-link trees)
  - Irregular structures (DAGs, graphs)
  - Unbounded sharing & irregular traversals (Harris)
- But inductive predicates easier for:
  - Simple inductive structures/abstractions

Compositional Reasoning

But can we reason compositionally about flows and graphs à la SL?

Graph Composition

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

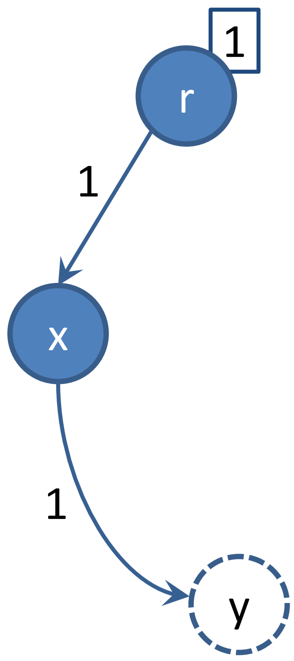


- Need a composition that preserves flow!

Flow Graphs

(in, G) is a flow graph iff

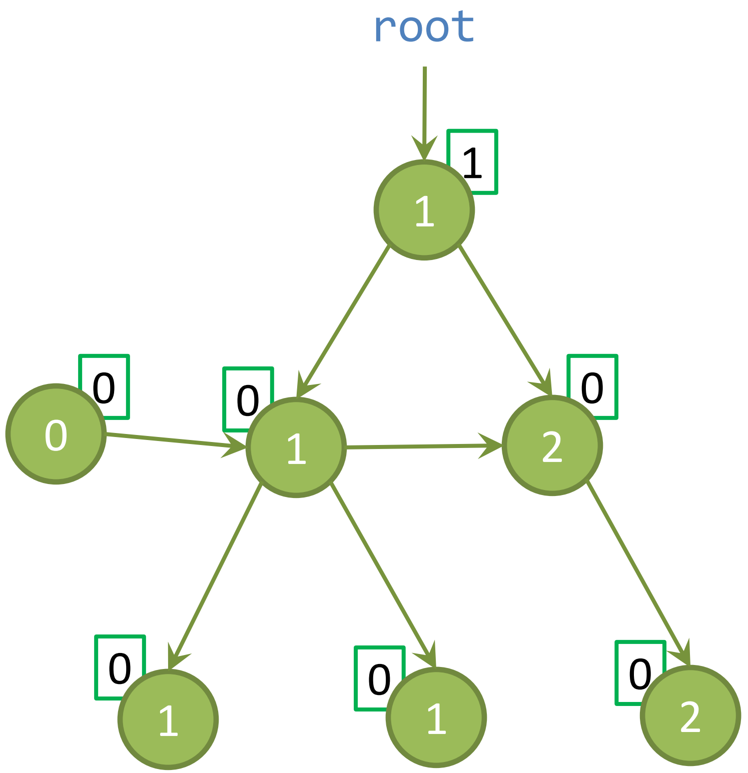
- $G = (N, N_o, e)$ is a partial graph with
  - N the set of internal nodes of the graph
  - N_o the set of *sink* nodes of the graph
  - $e: N \times (N \cup N_o) \rightarrow$ D is an edge function
- $in: N \rightarrow D$ is an inflow



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

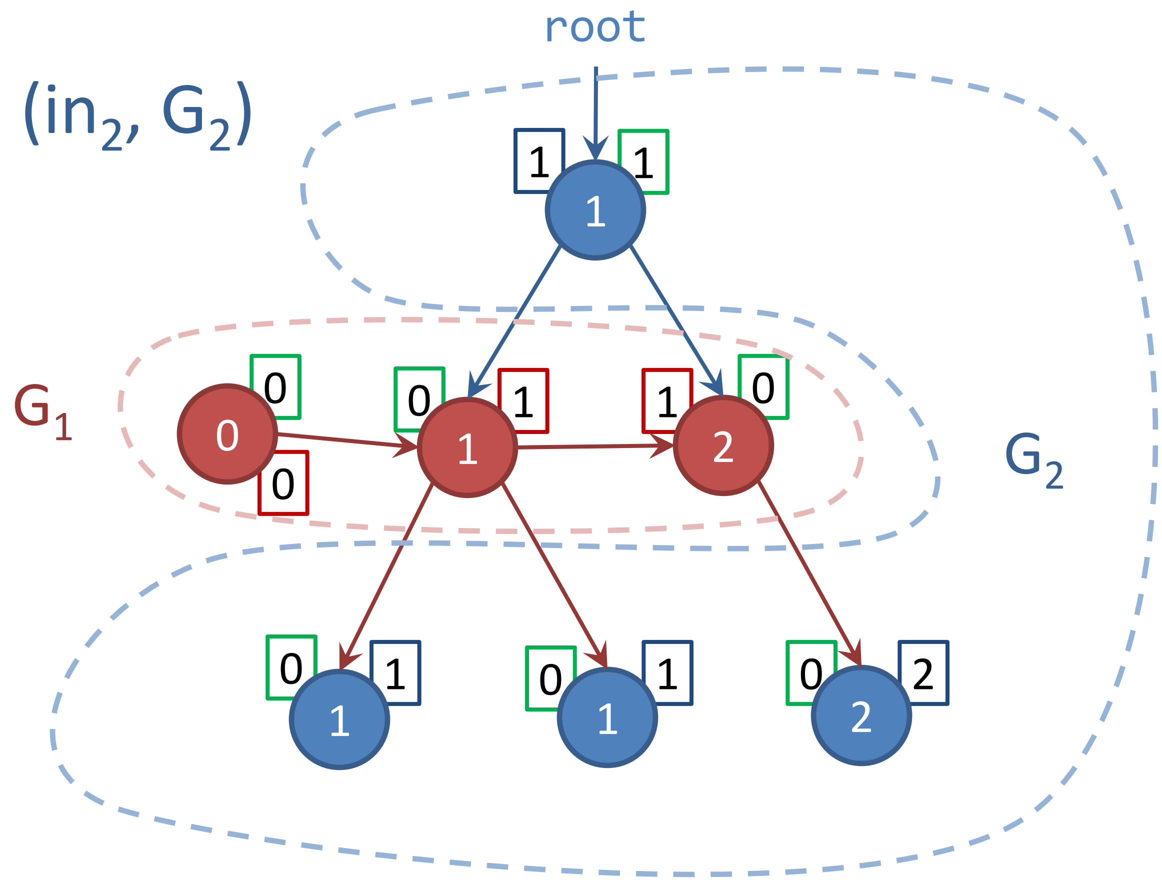
Flow Graph Composition

(in, G)



Flow Graph Composition

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



If $in_1(n) = in(n) + \sum_{n' \in G_2} \mathrm{flow}(in, G)(n') \cdot e(n', n)$

Flow Graph Composition

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

$\Rightarrow$ Flow graphs form a separation algebra.

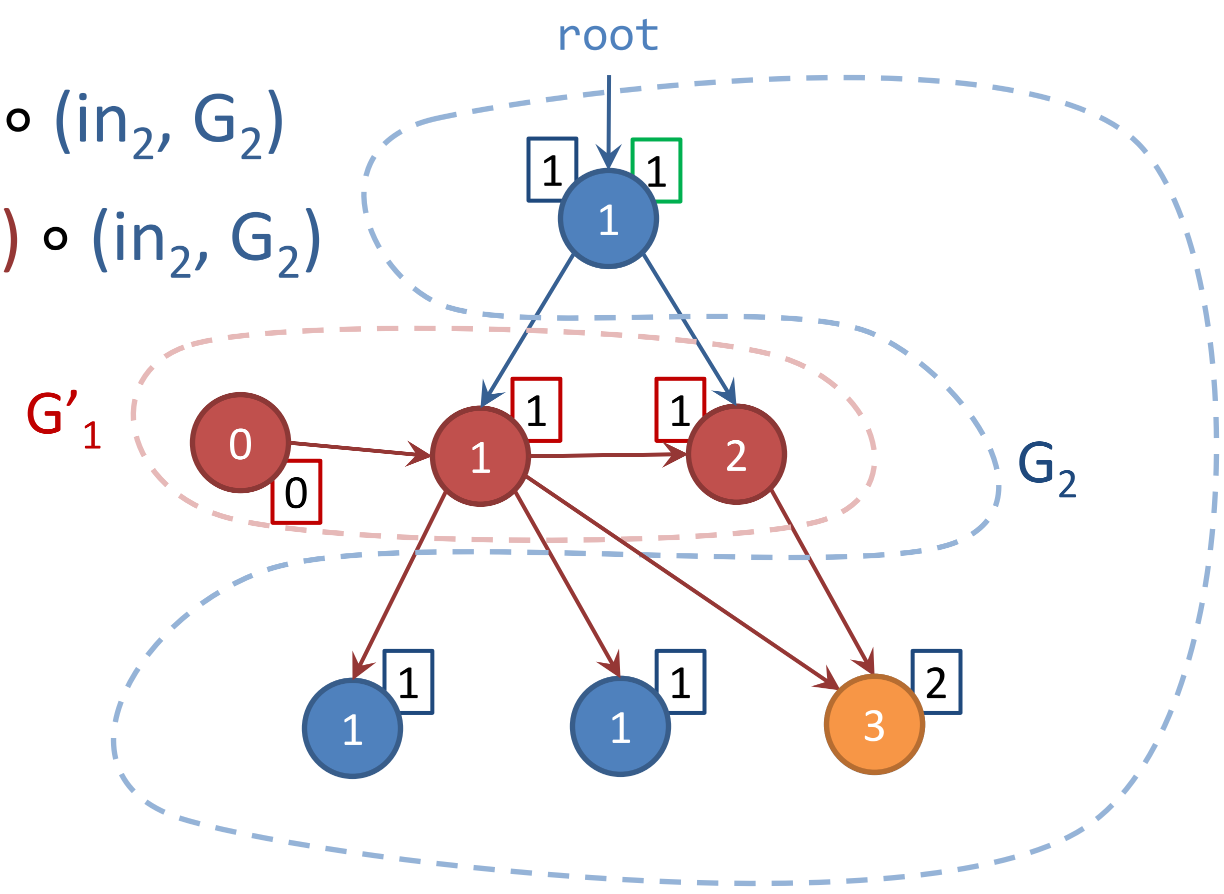
$\Rightarrow$ We can use them to give semantics to SL assertions.

- How do we abstract flow graphs?

Modifying the Graph

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

$(in, G') \neq (in_1, G'_1) \circ (in_2, G_2)$

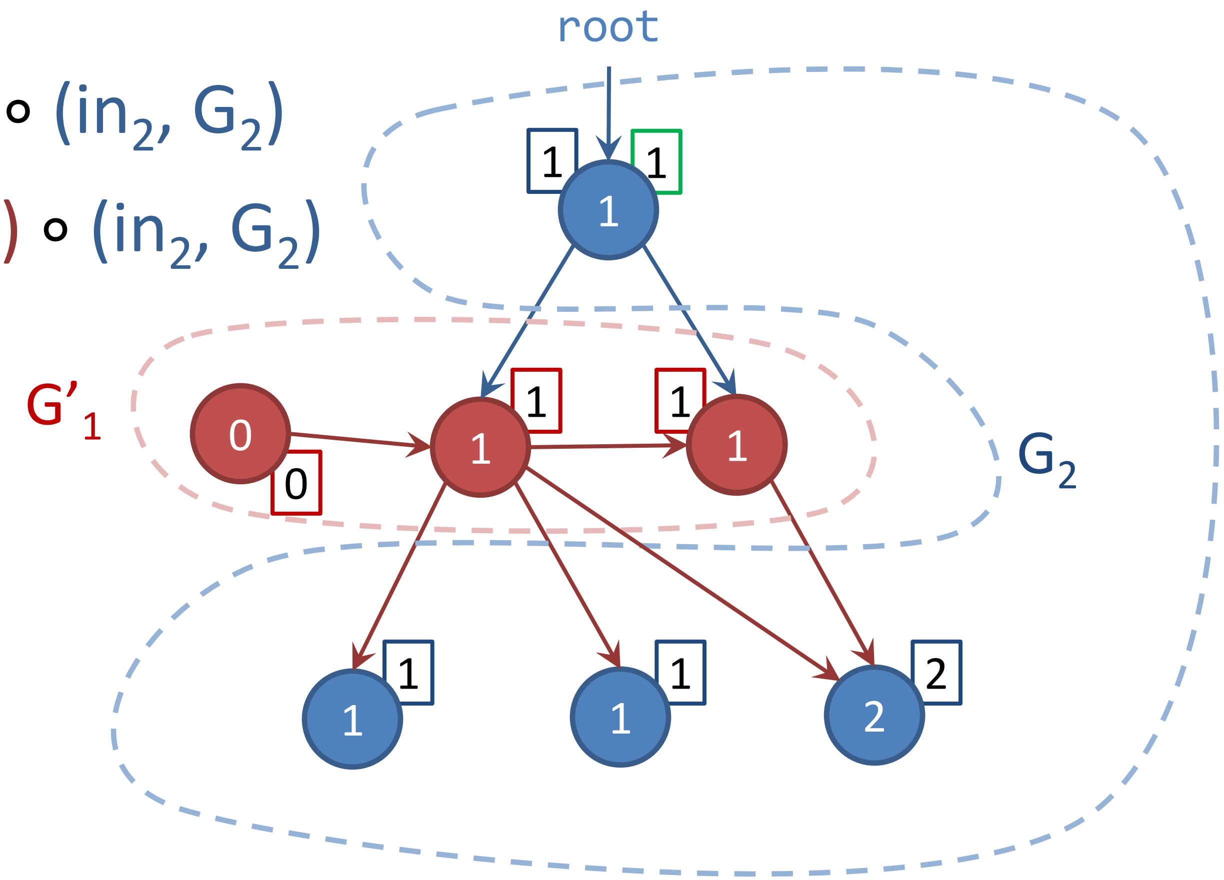


Need to preserve the flow into G_2...

Modifying the Graph

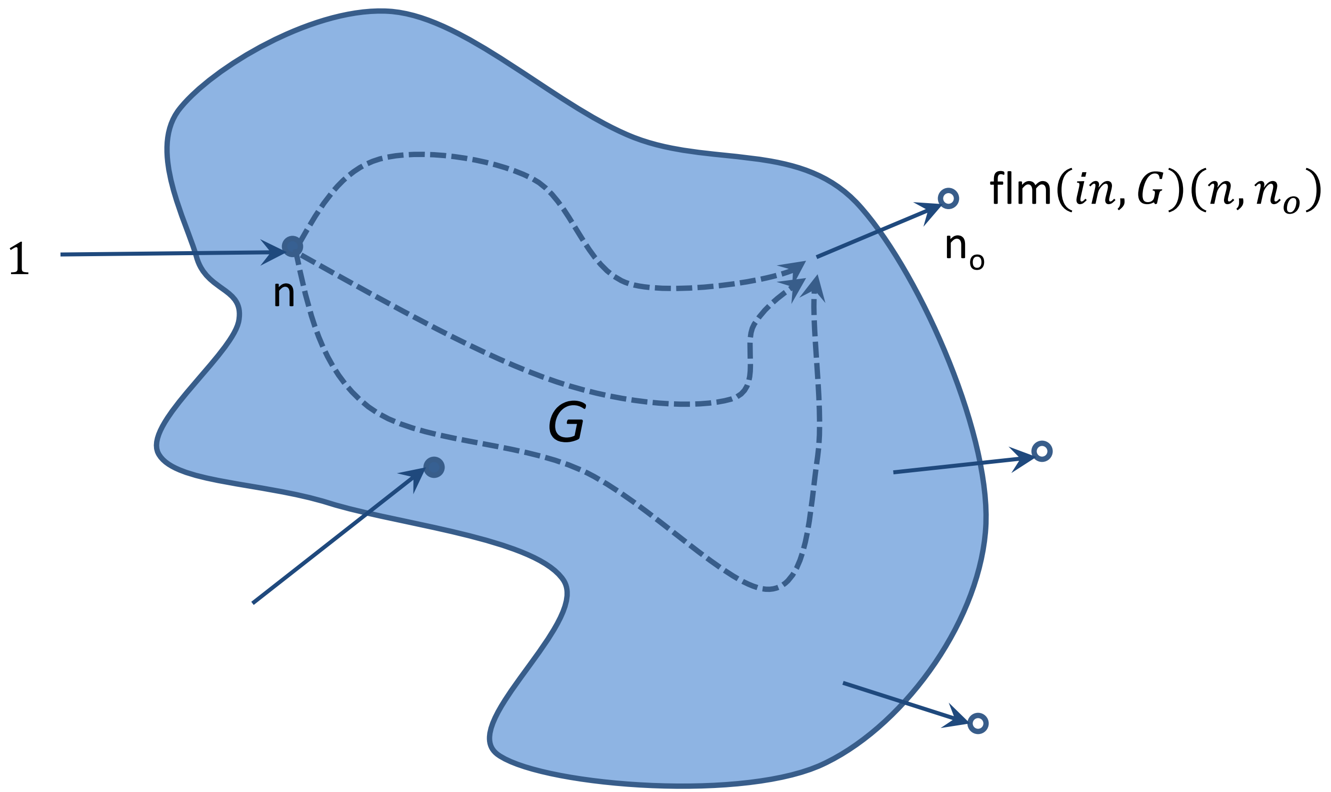
$(\mathrm{in}, G) = (\mathrm{in}_1, G_1) \circ (\mathrm{in}_2, G_2)$

$(\mathrm{in}, G') = (\mathrm{in}_1, G'_1) \circ (\mathrm{in}_2, G_2)$



Preserve the number of paths going through G_1?

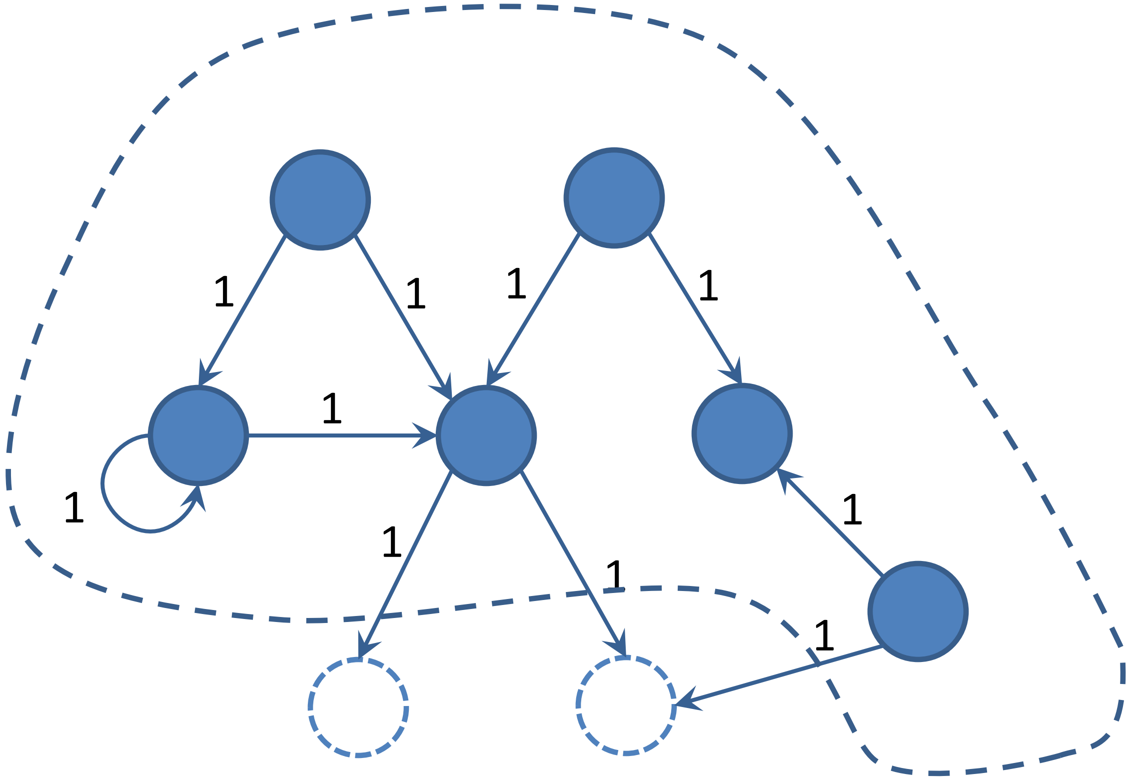
Flow Map of a Flow Graph



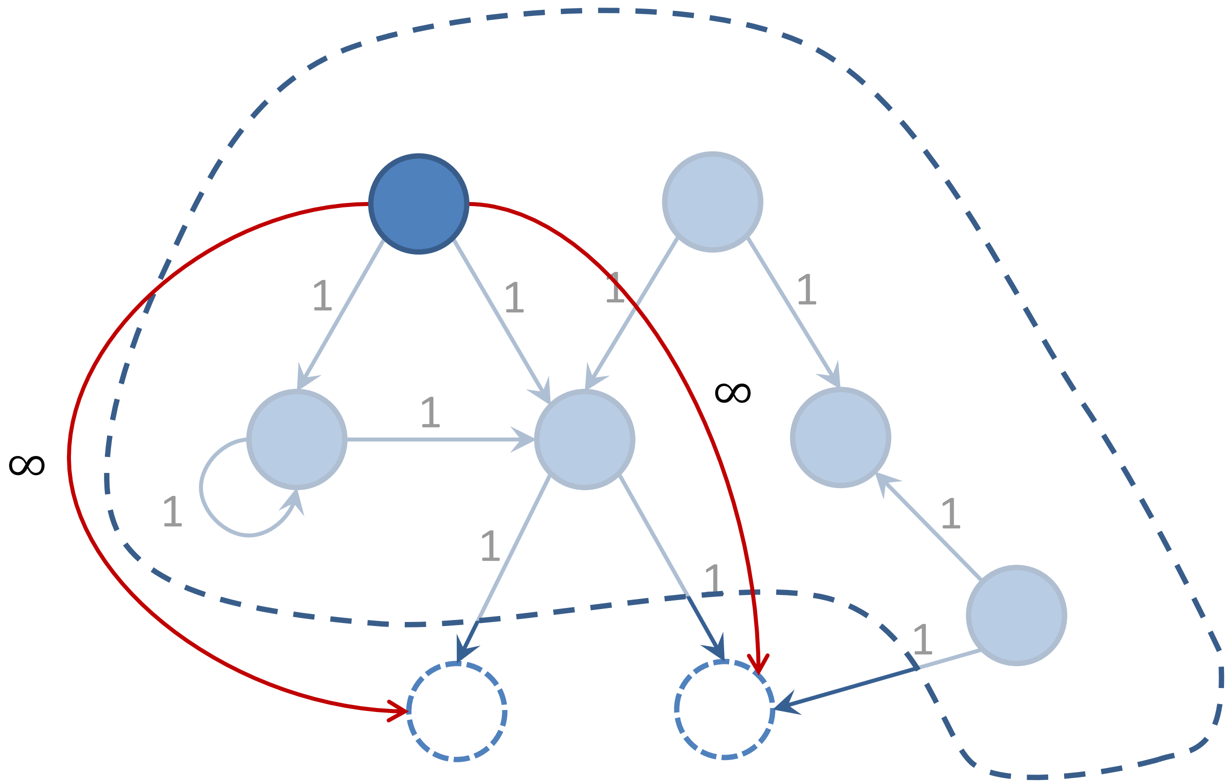
$\mathrm{flm}(in, G)(n, n_o) = \sum e(n, n_1) \cdots e(n_k, n_o)$ over all paths in G

$\mathrm{flow}(in, G)(n_o) = \sum_{n \in G} in(n) \cdot \mathrm{flm}(in, G)(n, n_o)$

Flow Map: Example

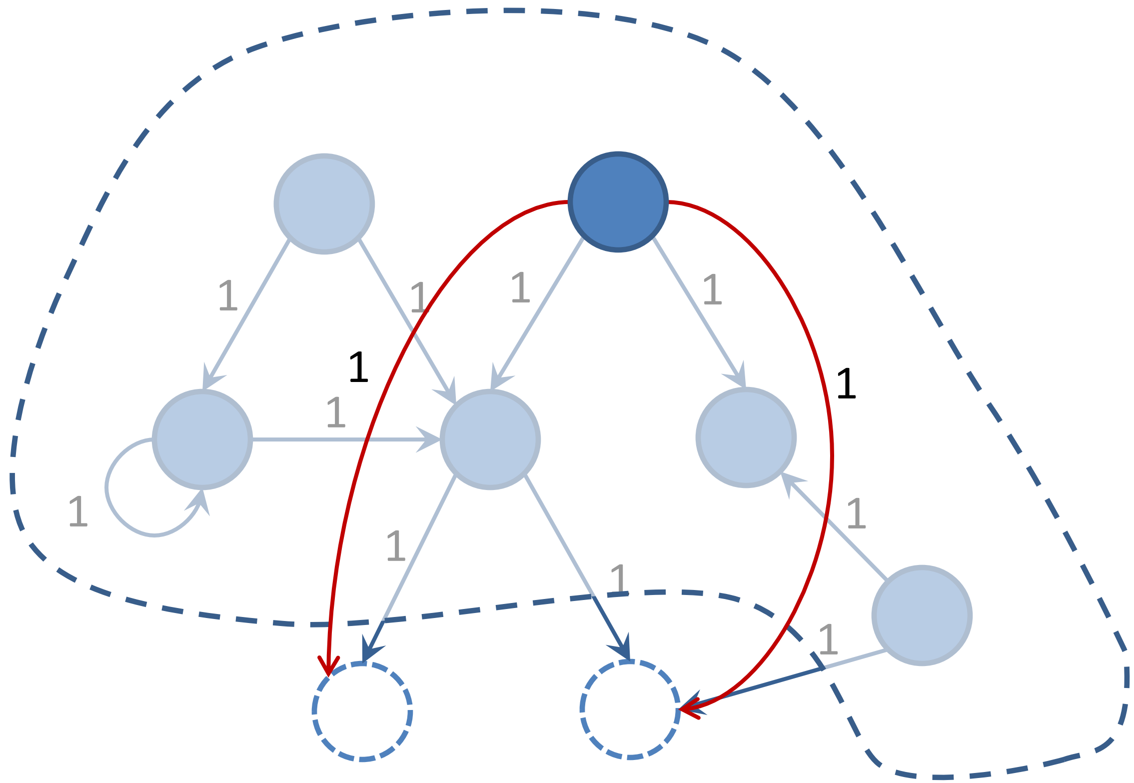


Flow Map: Example



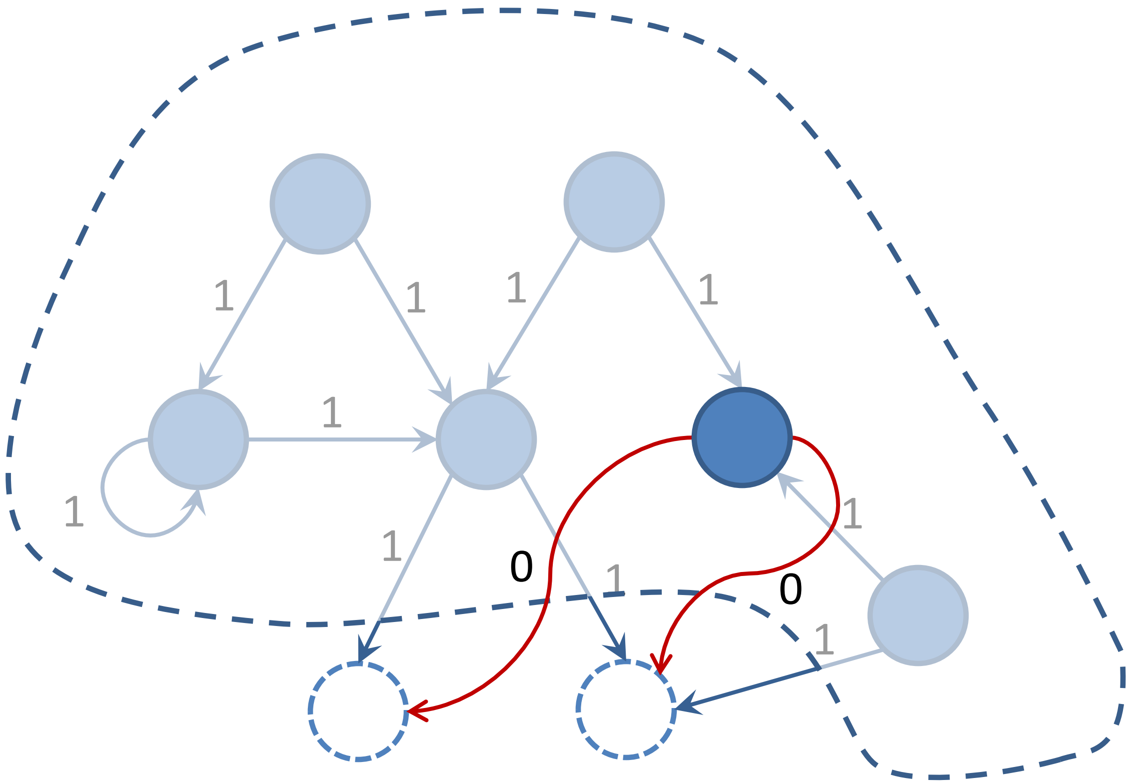
Flow map abstracts from internal structure of the graph

Flow Map: Example



Flow map abstracts from internal structure of the graph

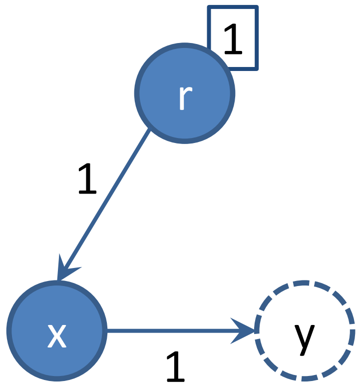
Flow Map: Example



Flow map abstracts from internal structure of the graph

Flow Interfaces

- $I = (in, a, f)$ is a flow interface
  - in: N $\rightarrow$ D is an inflow
  - $a \in A$ is the abstraction of node labels
  - f: N × N$_o$ $\rightarrow$ D is a flow map



$\in \;\; [\![(\{r \rightarrowtail 1, \ldots\}, \_\,, \{(r, y) \rightarrowtail 1\})]\!]$

Flow Interfaces

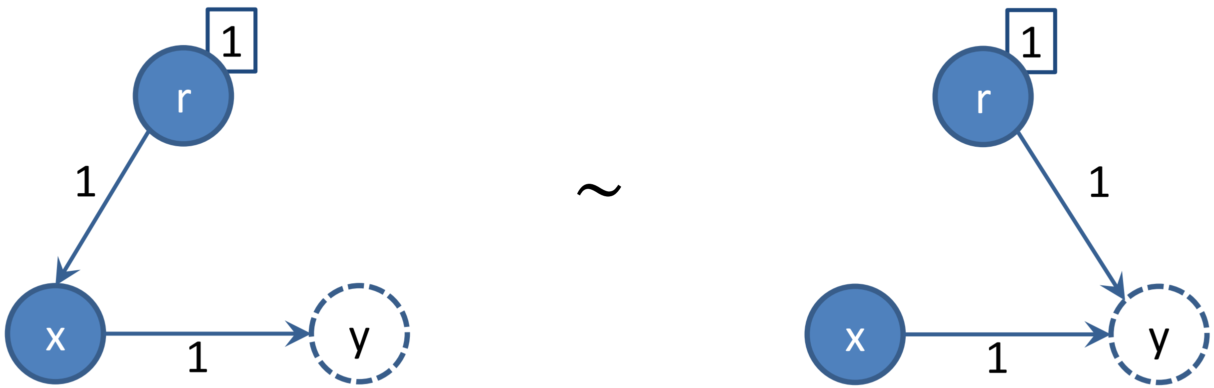
- Flow interfaces induce a congruence on flow graphs:

$$H_i, H_i' \in [\![I_i]\!] \wedge H_1 \circ H_2 \in [\![I]\!] \Rightarrow H_1' \circ H_2' \in [\![I]\!]$$

- Lift flow graph composition to interfaces: $\oplus$
- (Flow Interfaces, $\oplus$) also a separation algebra!

Reasoning about Modifications

- $\oplus$ congruence $\Rightarrow$ can replace G_1 with any G_1' with same interface



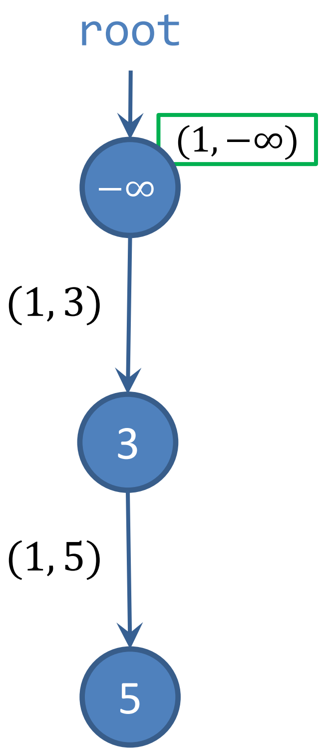
- Showing equivalence: requires fixpoint reasoning
- But concurrent algos modify small regions

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
- 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

Example

Sorted Linked List



$$\gamma(x, in, a, f) \stackrel{\text{def}}{=} n \mapsto k, n' \wedge\ in(n) = (1, l)$$
$$\wedge\, a = \{k\} \wedge l \leq k$$
$$\wedge\, f = ITE(n' = null, \epsilon, \{(n, n') \rightarrowtail 1\})$$

Global invariant:

$$\mathrm{Gr}(I) \wedge I^{in} + \mathbf{0} = \{r \rightarrowtail (1, -\infty)\} + \mathbf{0} \wedge I^{f} = \epsilon$$

Harris' List

$$\gamma(n, in, a, f) := \exists n', n''.\ n \mapsto n', n'' \wedge a \neq \top \wedge (M(n') \Leftrightarrow a \neq \diamond) \wedge (0,0) < in(n) \leq (1,1)$$
$$\wedge\ (in(n) \geq (0,1) \Rightarrow a \neq \diamond) \wedge (n = ft \Rightarrow in(n) \geq (0,1)) \wedge (in(n) \leq (1,0) \Rightarrow n'' = null)$$
$$\wedge\ f = \mathsf{ITE}(u(n') = null, \epsilon, \{(n, u(n')) \rightarrowtail (1,0)\}) + \mathsf{ITE}(n'' = null, \epsilon, \{(n, n'') \rightarrowtail (0,1)\}).$$

Global invariant:

$$\Phi \stackrel{\text{def}}{=} \exists I.\ \mathsf{Gr}(I) \wedge I^{in} + \mathbf{0} = \{mh \rightarrowtail (1,0)\} + \{fh \rightarrowtail (1,0)\} + \mathbf{0} \wedge I^{f} = \epsilon$$

Data-Structure-Agnostic Proof Rules

- Decomposition

$$\mathsf{Gr}(I) \land x \in I \;\models\; \exists I_1, I_2.\, \mathsf{N}(x, I_1) * \mathsf{Gr}(I_2) \land I = I_1 \oplus I_2$$

- Step

$$I = I_1 \oplus I_2 \land (x, y) \in I_1^f \land I^f = \epsilon \;\models\; y \in I_2$$

- Composition

$$\mathsf{Gr}(I_1) * \mathsf{Gr}(I_2) \;\models\; \mathsf{Gr}(I_1 \oplus I_2)$$

Harris: Insert

```
procedure insert() {
  var l := mh;
  var r := unmarked(l.next);
  while (r != null && ?? ) {
    l := r;
    r := unmarked(l.next);
  }
  …
}
```

$\{\Phi\}$

$\{\mathrm{Gr}(\mathrm{I}) \wedge mh \in I^{in}\}$

(Decomp)

$\{N(l, I_l) * \mathrm{Gr}(\mathrm{I}_2) \wedge I = I_l \oplus I_2\}$

Harris: Insert

```
procedure insert() {
  var l := mh;
  var r := unmarked(l.next);
  while (r != null && ?? ) {
    l := r;
    r := unmarked(l.next);
  }
  …
}
```

$\{\Phi\}$

$\{N(l, I_l) * \mathrm{Gr}(\mathrm{I}_2) \wedge I = I_l \oplus I_2\}$

$\left\{I = I_l \oplus I_2 \wedge (l, r) \in I_l^f \wedge I^f = \epsilon\right\}$

(Step), (Decomp)

$\{N(l, I_l) * N(r, I_r) * \mathrm{Gr}(\mathrm{I}_3) \wedge I = I_l \oplus I_r \oplus I_3\}$

Harris: Insert

```
procedure insert() {
  var l := mh;
  var r := unmarked(l.next);
  while (r != null && ?? ) {
    l := r;
    r := unmarked(l.next);
  }
  …
}
```

$\{\Phi\}$

$\{N(l, I_l) * \mathrm{Gr}(\mathrm{I}_2) \land I = I_l \oplus I_2\}$

$$\left\{\begin{matrix} N(l, I_l) * N(r, I_r) * \mathrm{Gr}(\mathrm{I}_3) \\ \land I = I_l \oplus I_r \oplus I_3 \end{matrix}\right\}$$

$$\Big\downarrow \text{(Comp)}$$

$\{N(r, I_r) * Gr(I_4) \land I = I_r \oplus I_4\}$

Harris: Insert

```
procedure insert() {
  var l := mh;
  var r := unmarked(l.next);
  while (r != null && ?? ) {
    l := r;
    r := unmarked(l.next);
  }
  …
}
```

$\{\Phi\}$

$\{N(l, I_l) * \mathrm{Gr}(\mathrm{I}_2) \land I = I_l \oplus I_2\}$

$\{N(l, I_l) * \mathrm{Gr}(\mathrm{I}_4) \land I = I_l \oplus I_4\}$

Flow Interfaces

- 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

Background

The Trouble with Inductive Predicates

Flow Interfaces

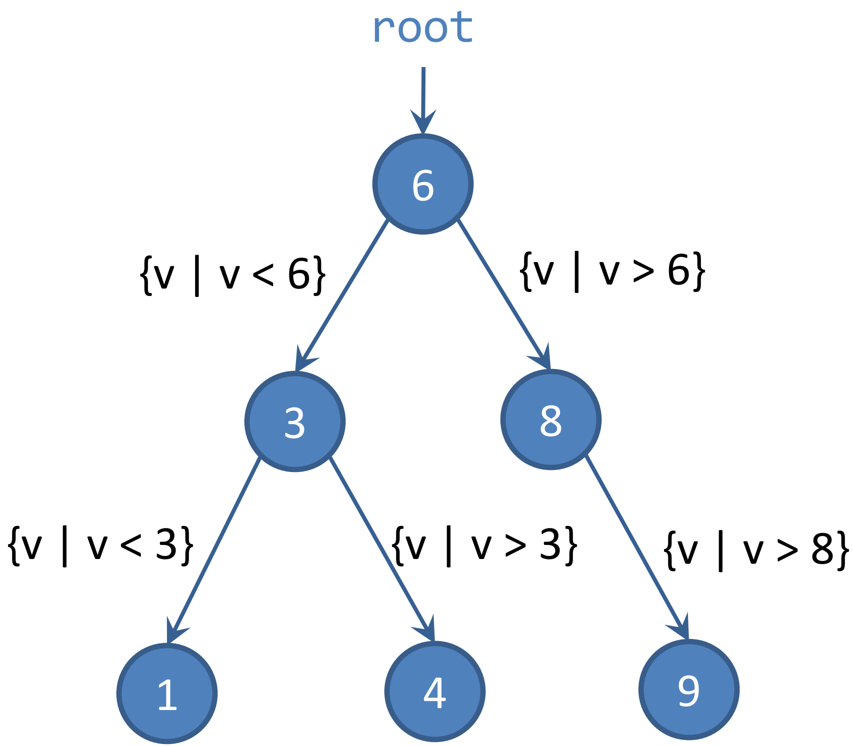
Concurrent Dictionaries

A memory-safe, linearizable, abstract template

Concurrent Dictionaries

- Dictionary: key-value store
- Examples: sorted linked lists, BSTs, B-trees, hash maps, …
- With flows: generic template + proof

Inset Flows

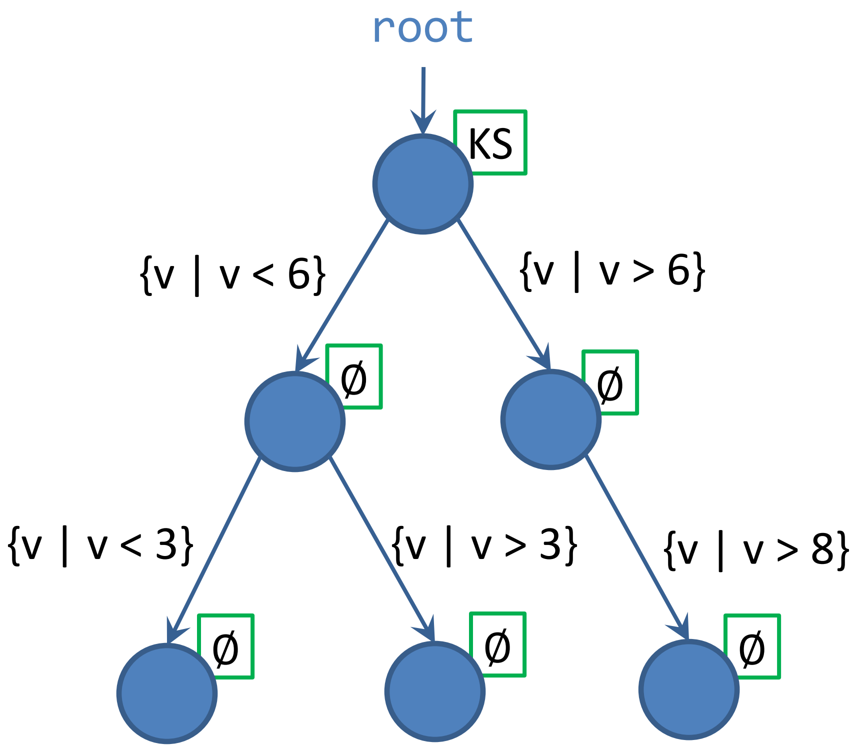


KS: the set of all search keys
e.g. KS = Int

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

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

Inset Flows



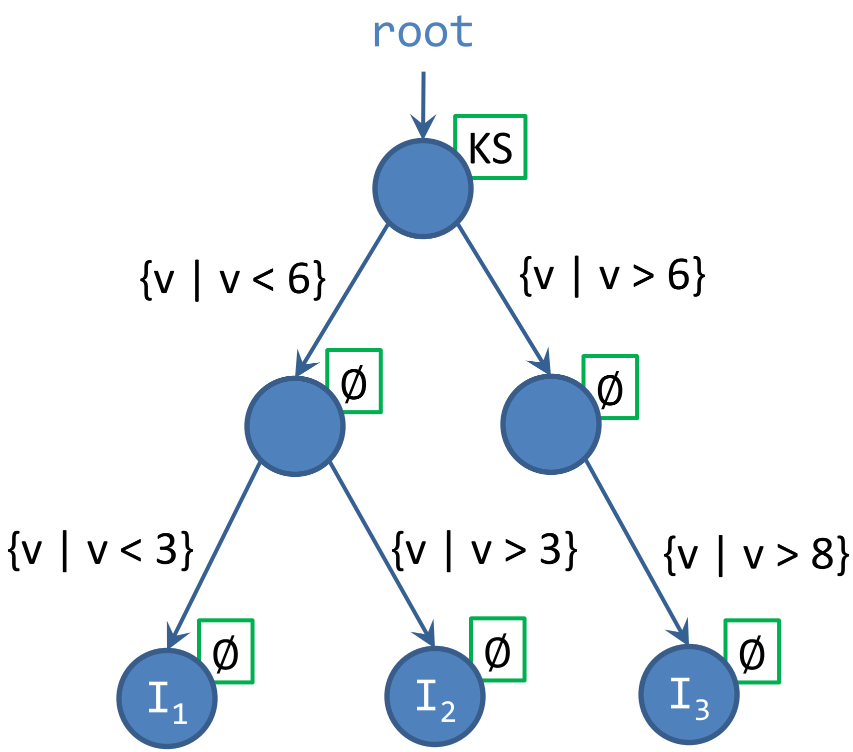
KS: the set of all search keys
e.g. KS = Int

Inset flow domain:

$$(2^{\mathrm{KS}}, \subseteq, \cup, \cup, \cap, \emptyset, \mathrm{KS})$$

Set inflow *in* of `root` to KS and to ∅ for all other nodes.

Inset Flows



I_1 = {v | 3 < v}

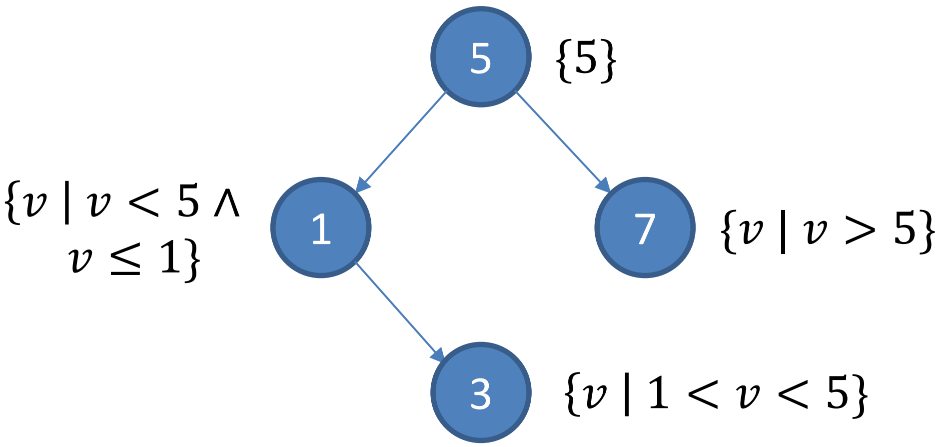
I_2 = {v | 3 < v < 6}

I_3 = {v | 8 < v}

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 Dictionaries

- Good state conditions
  - edgesets are disjoint for each n: $\{e(n,n')\}_{n' \in N}$ are disjoint
  - keyset of each n covers n's contents: $C(G)(n) \subseteq keyset(in, G)(n)$

- Keyset Theorem [Shasha and Goodman, 1988]

If all ops preserve good state, and methods search/insert/delete k at node s.t. $k \in$ keyset(n)

$\Rightarrow$ The implementation is linearizable

Encoding Using Flows

$$\gamma(x, in, (C, T), f) := \exists t.\ (\overbrace{\gamma_g(x, in, C, t, f)}^{\text{Description of good node}} \land T = \{t\} \lor \overbrace{\gamma_b(x, t) \land t \neq 0 \land T = \{\bar{t}\})}^{\text{Description of locked node}}$$

$$\land \underbrace{C \subseteq in(x) \land \forall y.\ (C \cap f(x, y) = \emptyset}_{\text{Contents are in keyset}} \land \underbrace{\forall z.\ f(x, y) \cap f(x, z) = \emptyset)}_{\text{Edgesets leaving node are disjoint}}$$

Global invariant:

$$\Phi := \exists I, in.\ \mathrm{Gr}(I) \land in \in I^{In} \land in + \mathbf{0} = \{r \rightarrowtail \mathrm{KS}\} + \mathbf{0} \land I^{f} = \epsilon$$

Give-up Template

`procedure dictionaryOp(Key k) {` $\left\{ \boxed{\Phi} \right\}$

`  var c := r;` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \right\}$

`  while (true) {` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \right\}$

`    lock(c);` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \right\}$

`    var n;`

`    if (inRange(c, k)) {` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \land k \in {I_c}^{In}(c) \right\}$

`      n := findNext(c, k);` $\left\{ \begin{array}{l} \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \land k \in {I_c}^{In}(c) \\ \land (n \neq null \land k \in {I_c}^f(c, n) \lor n = null \land \forall x \in {I_c}^f .\, k \notin {I_c}^f(c, x)) \end{array} \right\}$

`      if (n == null) break;`

$\left\{ \boxed{(\mathsf{N}(c, I_c) * \mathsf{N}(n, I_n)) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \right\}$

`    } else {`

`      n := r;` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \land n = r \right\}$

`    }` $\left\{ \boxed{(\mathsf{N}(c, I_c) * \mathsf{N}(n, I_n)) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \lor \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \land c = n = r \right\}$

`    unlock(c);`

`    c := n;` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \right\}$

`  }` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \land k \in {I_c}^{In}(c) \land \forall x \in {I_c}^f .\, k \notin {I_c}^f(c, x) \right\}$

`  var res := decisiveOp(c, k);` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \land {I_c}^a = (\_, \{t\}) \right\}$

`  unlock(c);` $\left\{ \boxed{\mathsf{N}(c, I_c) \mathrel{-\!\!*\!\!*} \Phi} \right\}$

`  return res;` $\left\{ \boxed{\Phi} \right\}$

`}`

Actions

$$t \in T \land \mathrm{N}(x, (\mathit{In}, (C, \{0\}), f)) \rightsquigarrow \mathrm{N}(x, (\mathit{In}, (C, T'), f)) \land T' \subseteq \{t, \bar{t}\} \tag{Lock}$$

$$t \in T \land \mathit{emp} \rightsquigarrow \mathrm{N}(x, (\{\{x \rightarrowtail 0\}\}, (\emptyset, \{\bar{t}\}), \epsilon)) \tag{Alloc}$$

$$t \in T \land \mathrm{Gr}(I) \land I^a \sqsubseteq (\_, \{t, \bar{t}\}) \rightsquigarrow \mathrm{Gr}(I') \land I'^a \sqsubseteq (\_, \{0, t, \bar{t}\}) \land I \precsim I' \tag{Sync}$$

Keyset Theorem with Flows

Theorem: An implementation of the template with appropriate γ_g, γ_b that satisfy the spec below (with some side conditions) is memory safe and linearizable.

$$\left\{ \begin{array}{l} \boxed{\mathsf{N}(c, I_c) \twoheadrightarrow\!\!\!* \; \Phi} \wedge {I_c}^{a} = (C, \{t\}) \wedge k \in {I_c}^{In}(c) \\ \wedge \, \forall x \in {I_c}^{f} .\, k \notin {I_c}^{f}(c, x) \end{array} \right\} \texttt{res := decisiveOp(c, k);} \; \left\{ \boxed{\mathsf{N}(c, I'_c) \twoheadrightarrow\!\!\!* \; \Phi} \wedge I_c \approx I'_c \wedge \Psi \right\}$$

$$\text{where } \Psi := \begin{cases} {I'_c}^{a} = (C, \{t\}) \wedge res \Leftrightarrow k \in C & \text{for } \texttt{member} \\ {I'_c}^{a} = (C \cup \{k\}, \{t\}) \wedge res \Leftrightarrow k \notin C & \text{for } \texttt{insert} \\ {I'_c}^{a} = (C \setminus \{k\}, \{t\}) \wedge res \Leftrightarrow k \in C & \text{for } \texttt{delete} \end{cases}$$

Background

The Trouble with Inductive Predicates

Flow Interfaces

Concurrent Dictionaries

Conclusion

A new way to reason about data structures

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 can abstract from the details of the specific data structure implementation.

Siddharth Krishna, Dennis Shasha, and Thomas Wies. Go with the Flow: Compositional Abstractions for Concurrent Data Structures. POPL 2018.