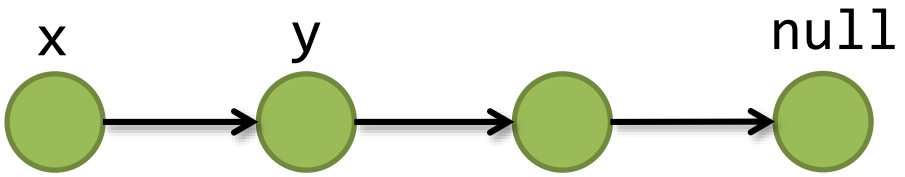
SMT-based Verification of Heap-manipulating Programs

Ruzica Piskac

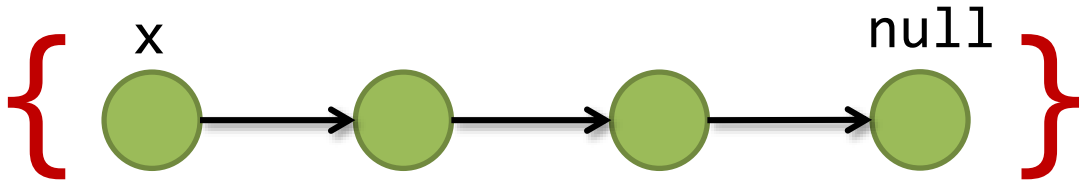
The 11th SRI Summer School on Formal Techniques
May 30 - June 5, 2022

A Motivating Example

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



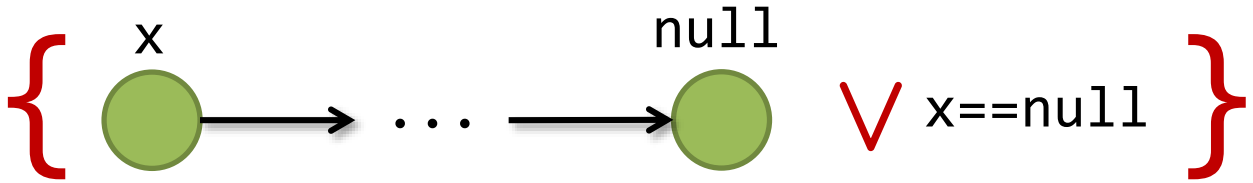
Proof by Hand-Waving



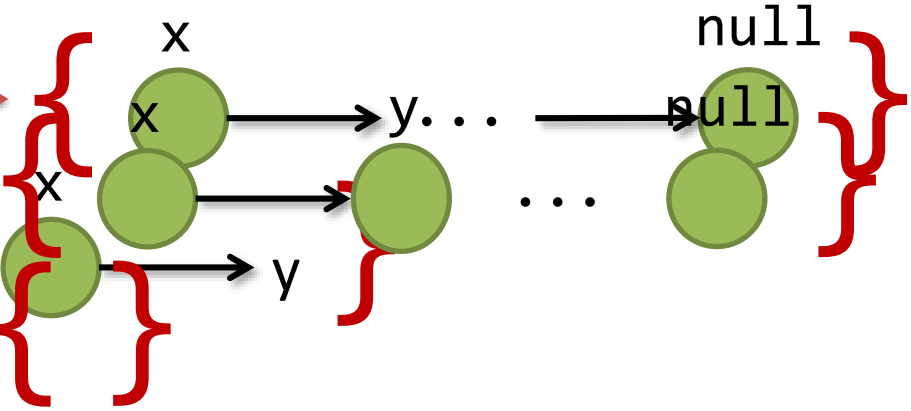

```
procedure delete(x: Node)
{
  if (x != null) {
    var y := x.next;
    delete(y);
    free(x);
  }
}
```

{ }

Proof by Hand-Waving

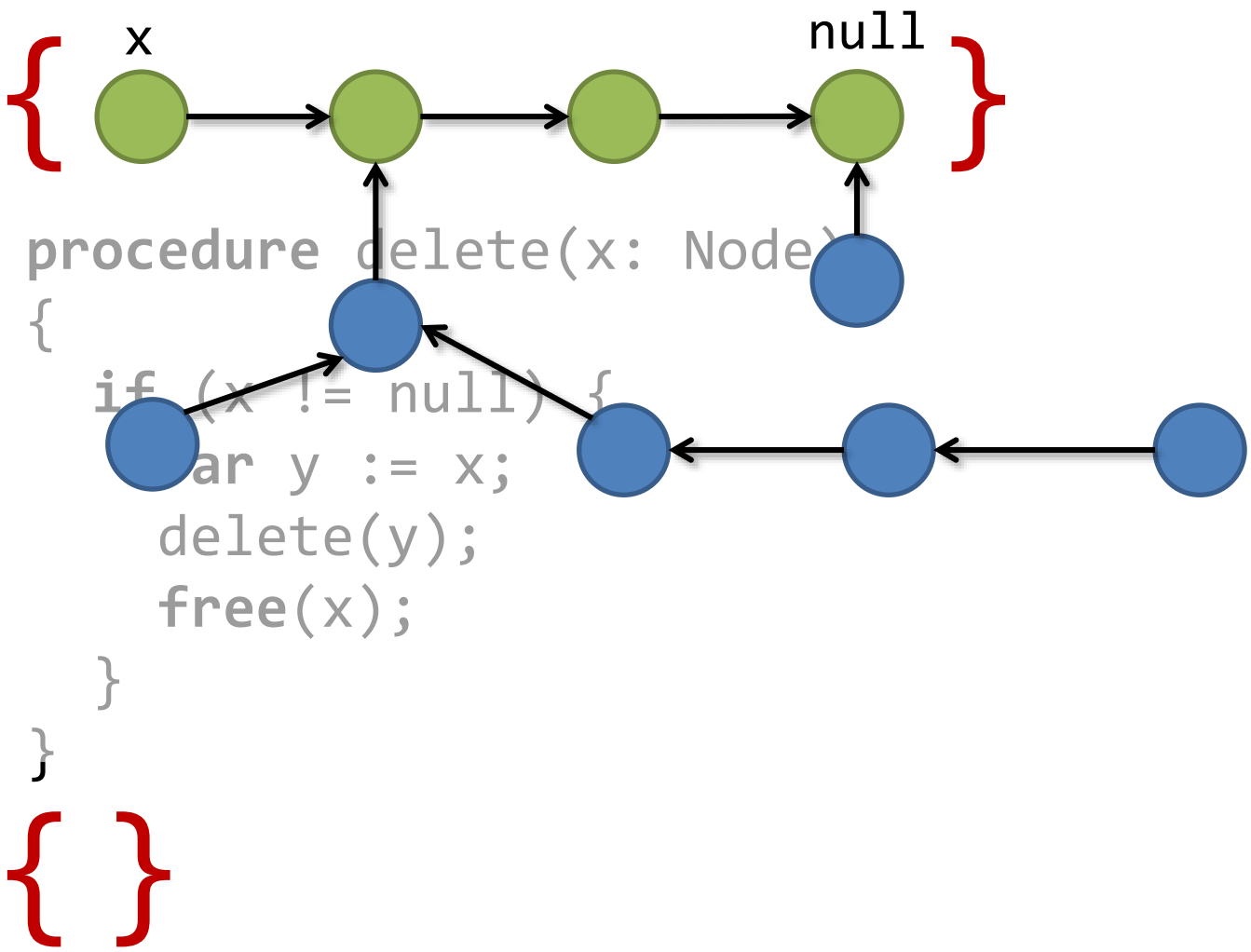



```
procedure delete(x: Node)
{
  if (x != null) {
    var y := x.next;
    delete(y);
    free(x);
  }
}
```



{ }

Proof by Hand-Waving



Implicit Dynamic Frames

Implicit Dynamic Frames by Example

- Pure assertions

```
x.next == y
```

Stack

x	10
y	42
…	

Heap

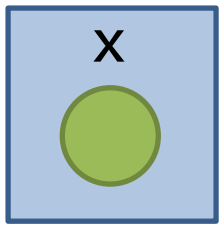
10	42
…	
42	?

```
struct Node {
  var next: Node;
}
```

Implicit Dynamic Frames by Example

- Permission predicates

acc(x)

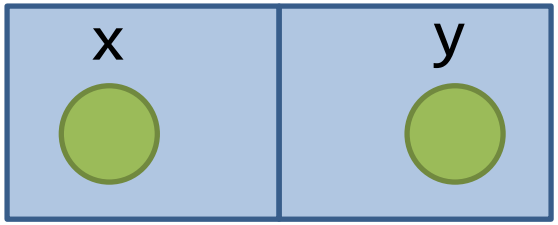


Expresses permission to access (i.e. read/write/deallocate) heap location x.
Assertions describe the program state **and** a set of locations that are allowed to be accessed.

Implicit Dynamic Frames by Example

- Separating conjunction

```
acc(x) * acc(y)
```

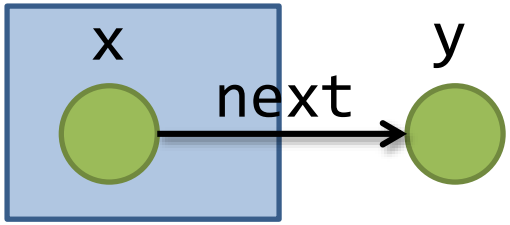


Yields union of permission sets of subformulas.
Permission sets of subformulas must be disjoint.

Implicit Dynamic Frames by Example

- Separating conjunction

```
acc(x) * x.next == y
```

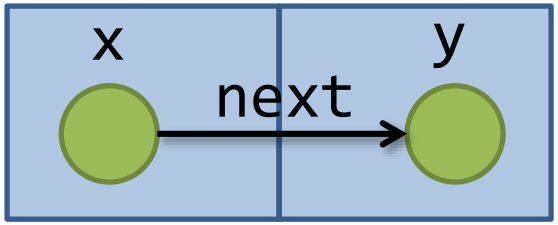


Pure assertions yield no permissions.

Implicit Dynamic Frames by Example

- Separating conjunction

```
acc(x) * acc(y) * x.next == y
```



Implicit Dynamic Frames by Example

- Separating conjunction

```
acc(x) * acc(x) * x.next == y
```

unsatisfiable ?

Implicit Dynamic Frames by Example

- Classical conjunction

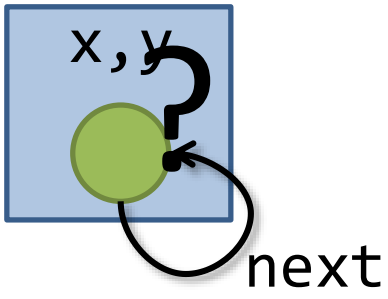
$$\texttt{acc(x)} \wedge \texttt{x.next == y}$$

unsatisfiable ?

Implicit Dynamic Frames by Example

- Classical conjunction

$$\texttt{acc(x)} \wedge \texttt{acc(y)} * \texttt{x.next == y}$$



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

Syntactic Short-hands

- Empty heap:

  $\texttt{emp} \equiv (\texttt{x} == \texttt{x})$

- Points-to predicates:

  $\texttt{x.next} \mapsto \texttt{y} \equiv \textbf{acc}(\texttt{x}) * \texttt{x.next} == \texttt{y}$

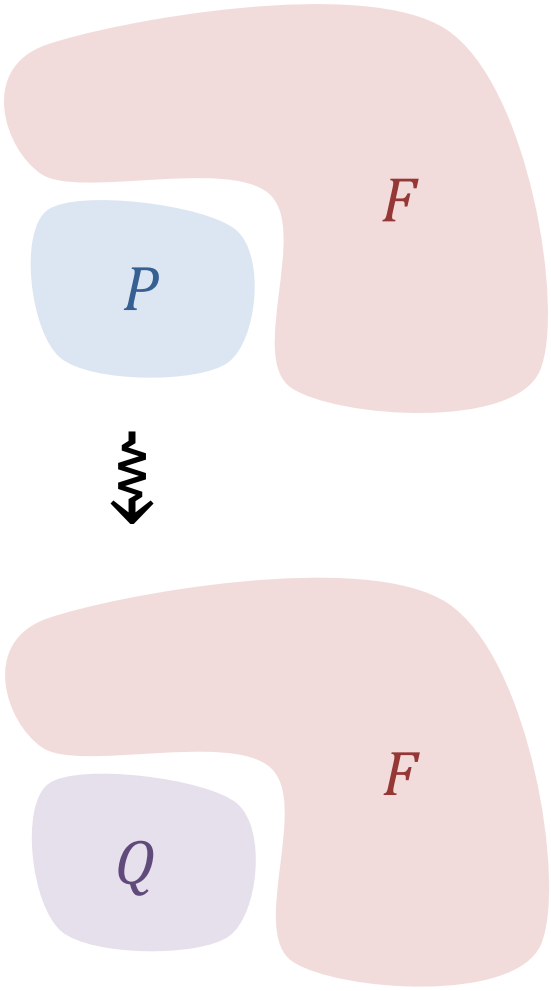
Implicit Dynamic Frames: Assertion Semantics

- M: first order structure, D: subset of M's universe

- $M,D \models P \quad \Leftrightarrow D = \emptyset$ and $M \models P$ if P is pure
- $M,D \models \mathbf{acc}(t) \quad \Leftrightarrow D = \{M(t)\}$
- $M,D \models P * Q \quad \Leftrightarrow$ exists D_1, D_2 s.t. $D = D_1 \uplus D_2$ and $M,D_1 \models P$ and $M,D_2 \models Q$
- $M,D \models P \wedge Q \quad \Leftrightarrow M,D \models P$ and $M,D \models Q$
- … everything else as in classical logic

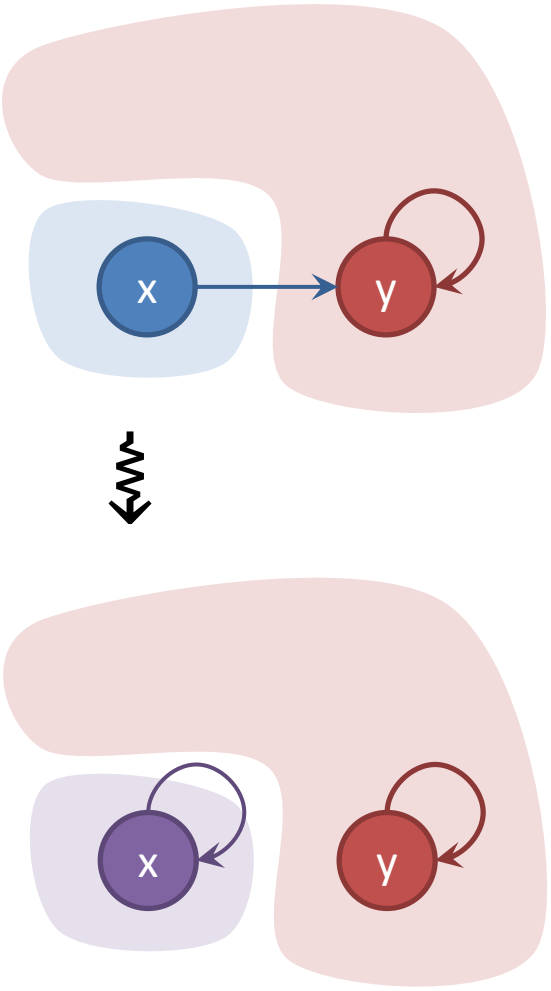
Frame Rule in Separation Logic

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



Frame Rule in Separation Logic

$$\frac{\{x \mapsto y\}\ \texttt{x. next := x}\ \{x \mapsto x\}}{\{x \mapsto y * y \mapsto y\}\ \texttt{x.next := x}\ \{x \mapsto x * y \mapsto y\}}$$



Hoare Rules

$\vdash \{\ \mathtt{acc(x)}\ \}\ \mathtt{x.f := y;}\ \{\ \mathtt{x.f} \mapsto \mathtt{y}\ \}$

$\vdash \{\ \mathtt{x.f} \mapsto \mathtt{z}\ \}\ \mathtt{y := x.f;}\ \{\ \mathtt{x.f} \mapsto \mathtt{z} * \mathtt{y} == \mathtt{z}\ \}$

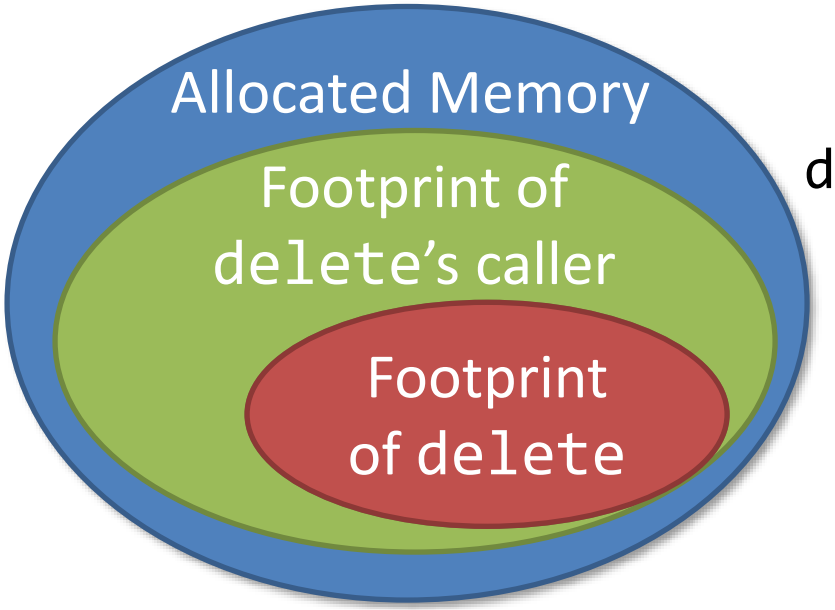
$\vdash \{\ \mathtt{emp}\ \}\ \mathtt{x := new\ T;}\ \{\ \mathtt{acc(x)}\ \}$

$\vdash \{\ \mathtt{acc(x)}\ \}\ \mathtt{free(x);}\ \{\ \mathtt{emp}\ \}$

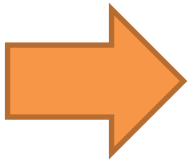
Some Examples of Hoare Triples

- {acc(x)} x.next := y; {acc(x) * x.next == y} 
- {acc(y)} x.next := y; {acc(y) * x.next == y} 
- {emp} x := new Node; {acc(x)} 
- {emp} free(x); {emp} 
- {acc(x)} free(x); {emp} 
- {acc(x) * acc(y) * y.next == z}
  x.next := y;
  {acc(x) * x.next == y * acc(y) * y.next == z} 

Encoding the Frame Rule




```
delete(x);
```

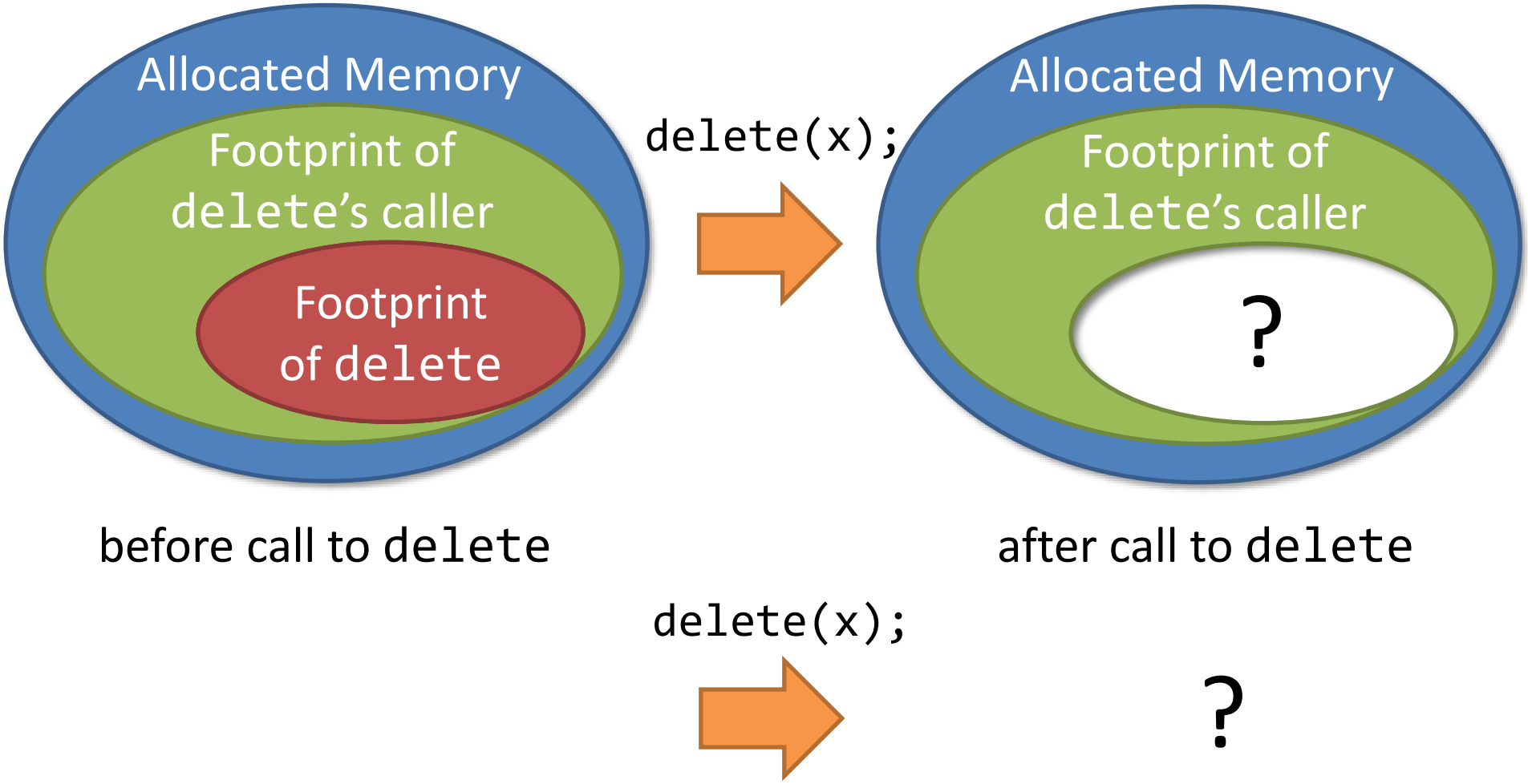


?

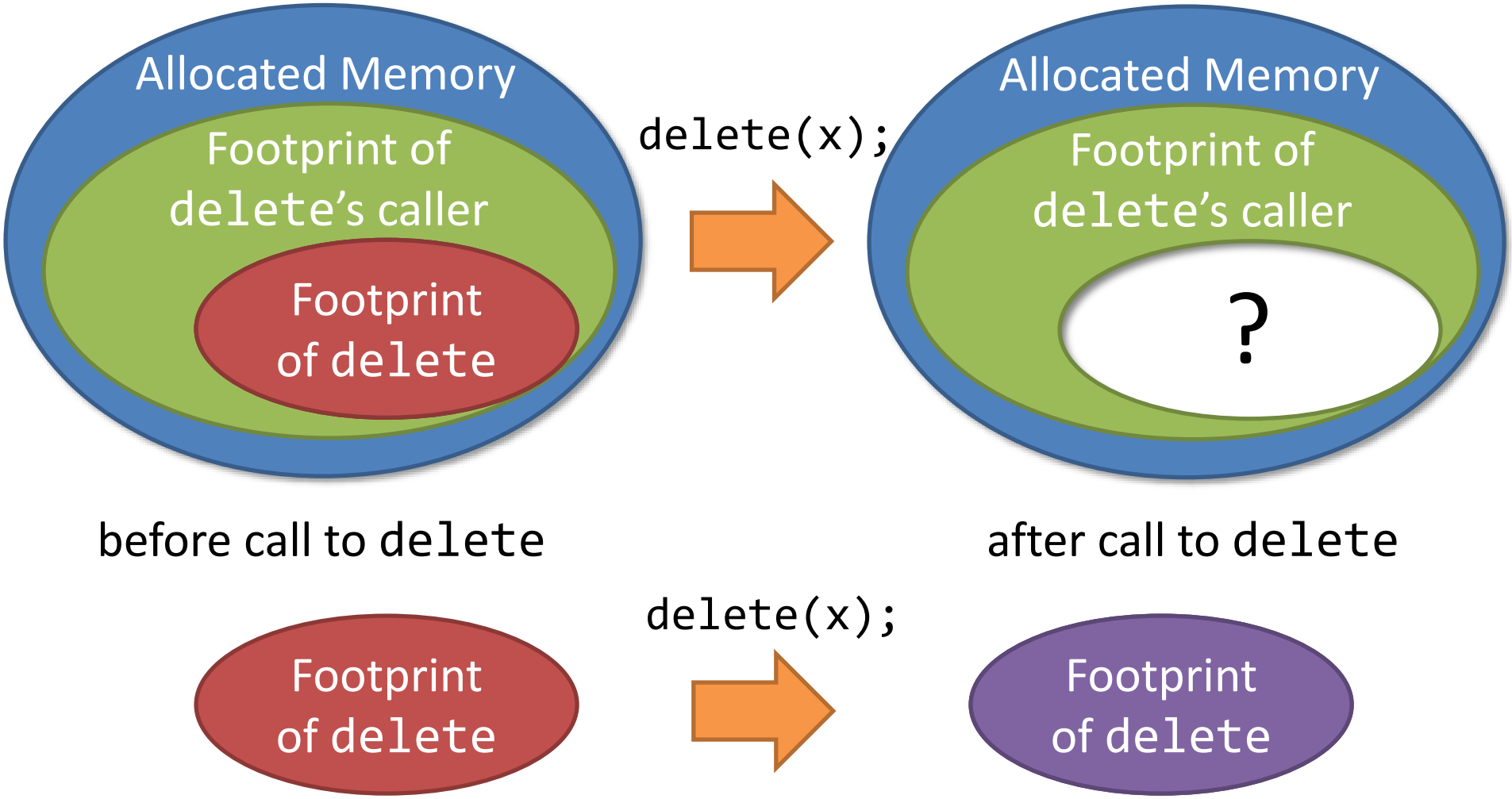
before call to `delete`

after call to `delete`

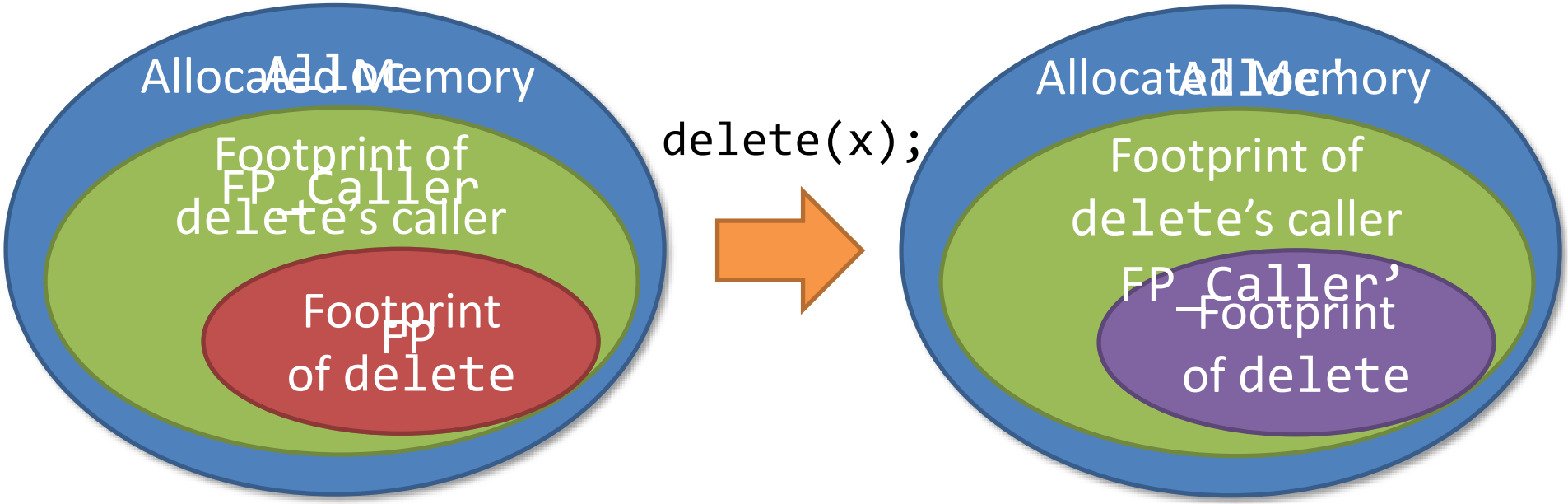
Encoding the Frame Rule



Encoding the Frame Rule



Encoding the Frame Rule



before call to `delete`

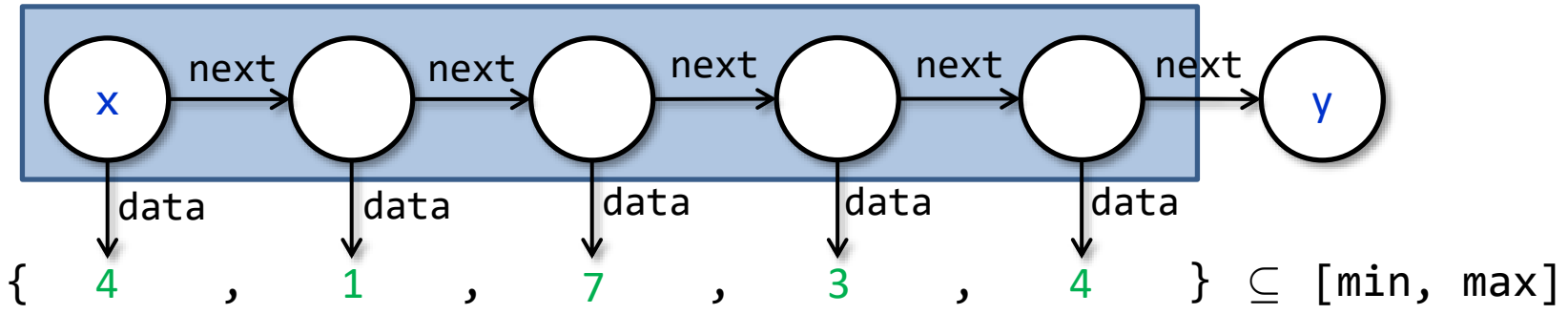
after call to `delete`

Reasoning about Heap and Data

Inductive Predicates with Data

- bounded list segment

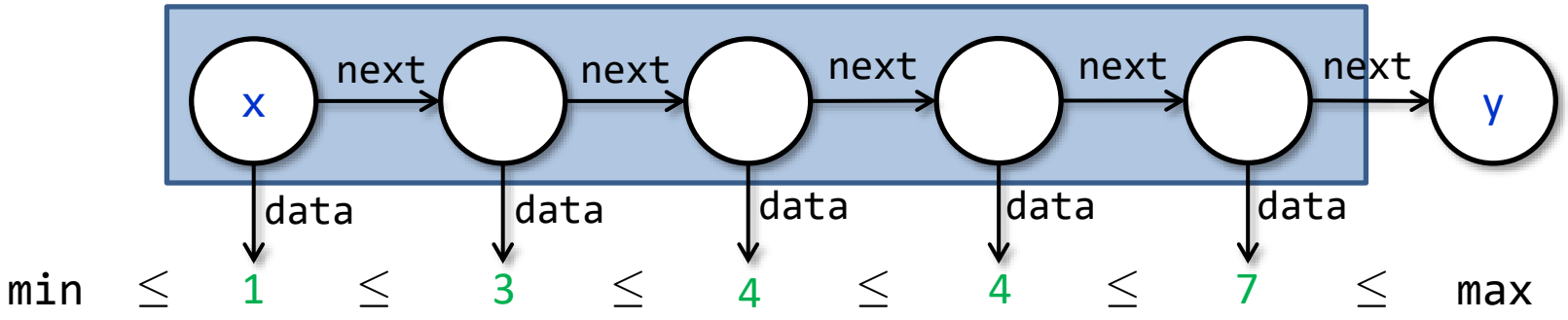
```
bnd_lseg(x, y, min, max) =
    x = y ∨
    x ≠ y * acc(x) * min ≤ x.data ≤ max *
    bnd_lseg(x.next, y, min, max)
```



Inductive Predicates with Data

- sorted list segment

```
srt_lseg(x, y, min, max) =
    x = y ∨
    x ≠ y * acc(x) * min ≤ x.data ≤ max *
    srt_lseg(x.next, y, x.data, max)
```

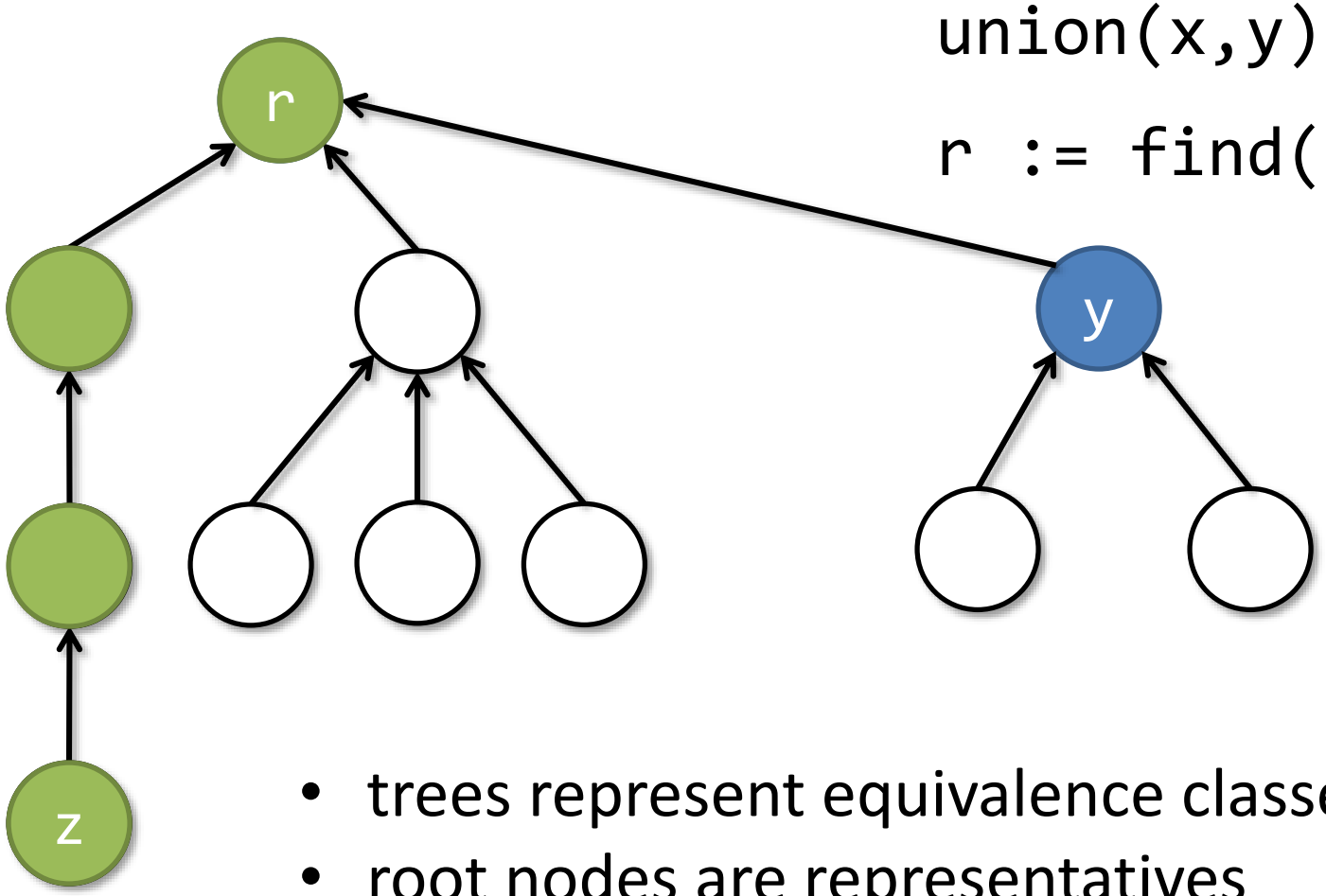


Example: Quicksort

```
procedure quicksort(x: Node, y: Node,
                    ghost min: int, ghost max: int)
  returns (z: Node)
  requires bnd_lseg(x, y, min, max)
  ensures srt_lseg(z, y, min, max)
{
  if (x != y && x.next != y) {
    var p: Node, w: Node;
    z, p := split(x, y, min, max);
    z := quicksort(z, p, min, p.data);
    w := quicksort(p.next, y, p.data, max);
    p.next := w;
  } else z := x;
}
```

Mixed Specifications

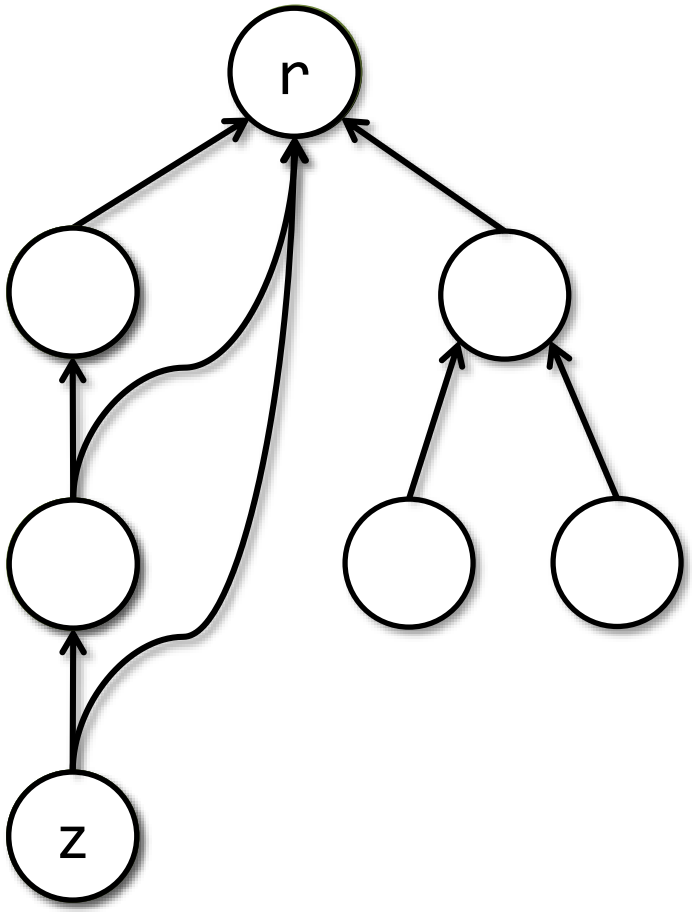
Example: Union/Find Data Structure



- trees represent equivalence classes
- root nodes are representatives

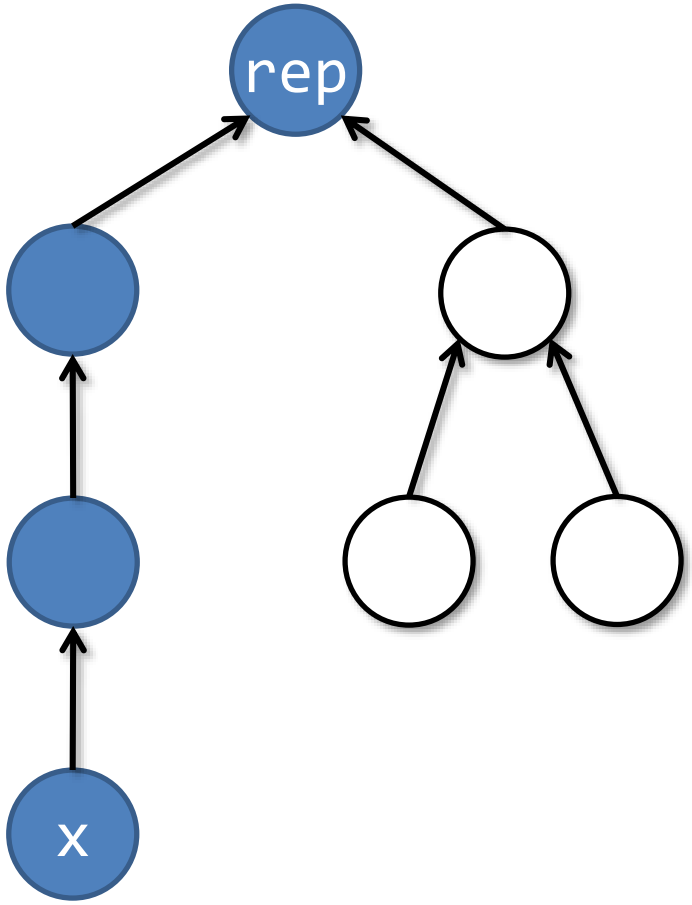
Find with Path Compression

```
procedure find(x: Node)
returns (r: Node)
{
  if (x != null) {
    r := find(x.next);
    x.next := r;
  } else {
    r := x;
  }
}
```



Find with SL Specification

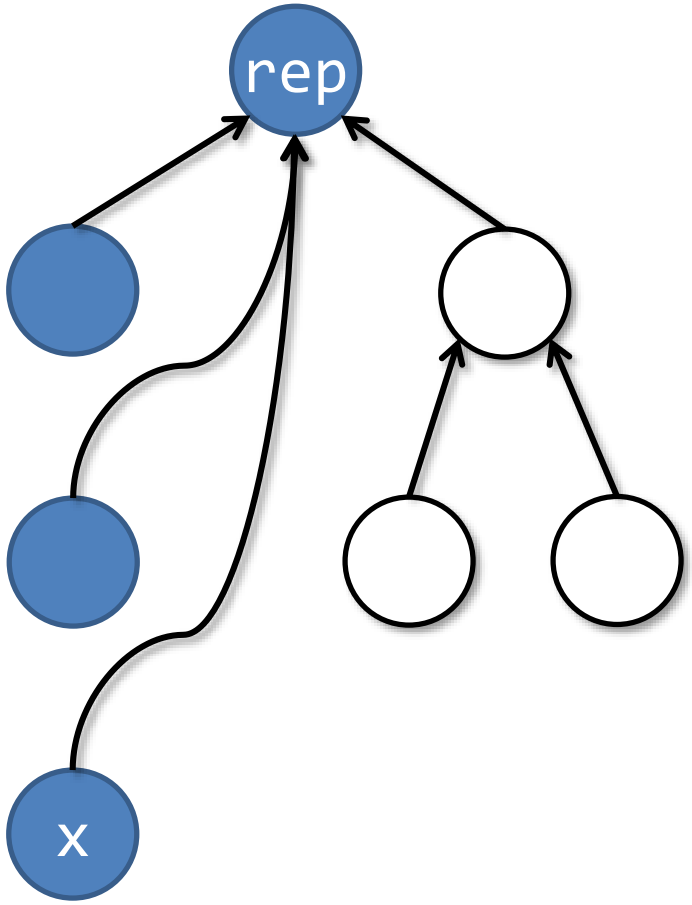
```
procedure find(x: Node, ghost rep: Node)
   returns (r: Node)
   requires lseg(x, rep)
   requires rep.next ↦ null
```



Find with SL Specification

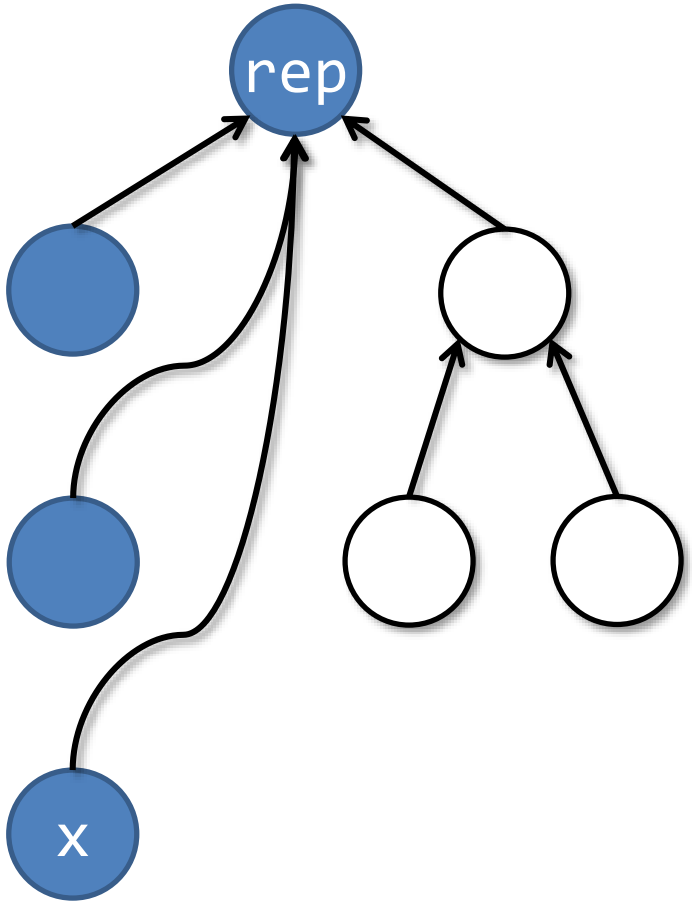
```
procedure find(x: Node, ghost rep: Node)
  returns (r: Node)
  requires rep.next ↦ null
  requires lseg(x, rep)
  ensures r == rep
  ensures rep.next ↦ null
  ensures ?
```

Postcondition needs to track an unbounded number of list segments.



Find with Mixed Specification

```
procedure find(x: Node, ghost rep: Node,
  implicit ghost X: Set<Node>)
  returns (r: Node)
  requires rep.next ↦ null
  requires lseg(x, rep) ∧ acc(X)
  ensures r == rep
  ensures rep.next ↦ null
  ensures acc(X)
  ensures ∀z ∈ X. z.next == rep
```



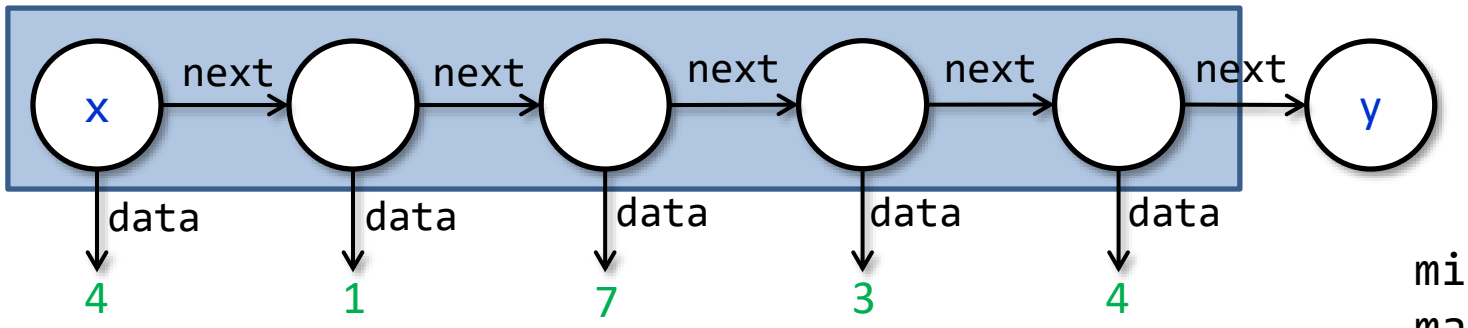
Completeness and Counterexamples

Quicksort Revisited

```
procedure quicksort(x: Node, y: Node,
                    ghost min: int, ghost max: int)
returns (z: Node)
  requires bnd_lseg(x, y, min, max)
  ensures srt_lseg(z, y, min, max)
{
  if (x != y && x.next != y) {
    var p: Node, w: Node;
    z, p := split(x, y, min, max);
    z := quicksort(z, p, min, p.data);
    w := quicksort(p.next, y, p.data, max);
    p.next := w;
  } else z := x;
}
```

Split with SL Specification

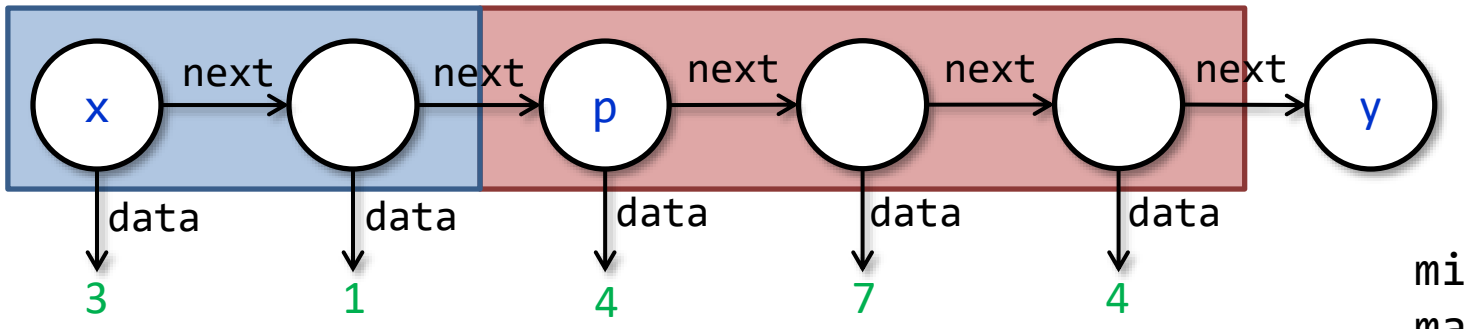
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```



Split with SL Specification

```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```

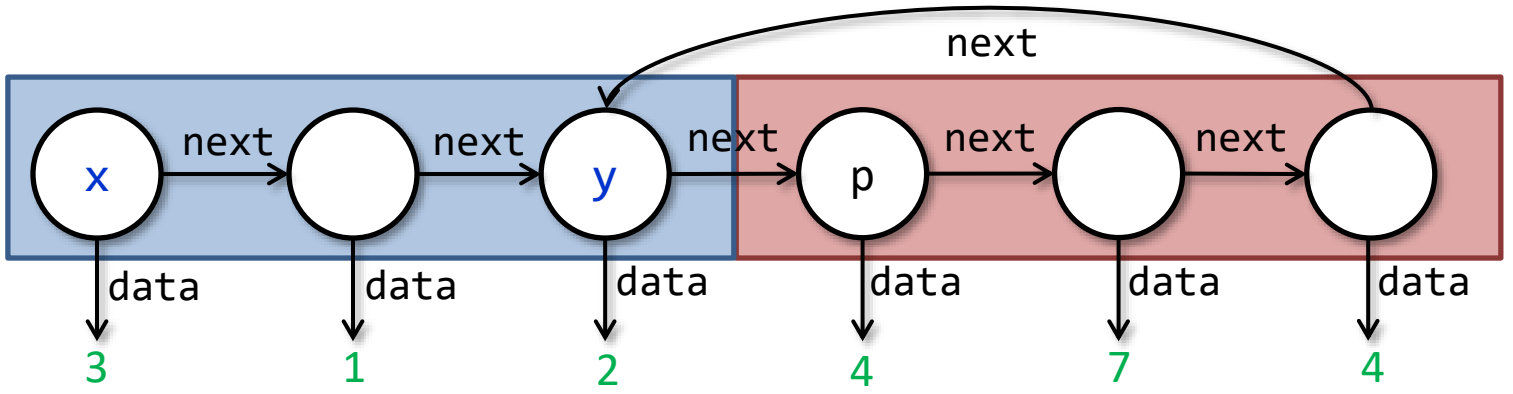
free memory



min = 1
max = 7

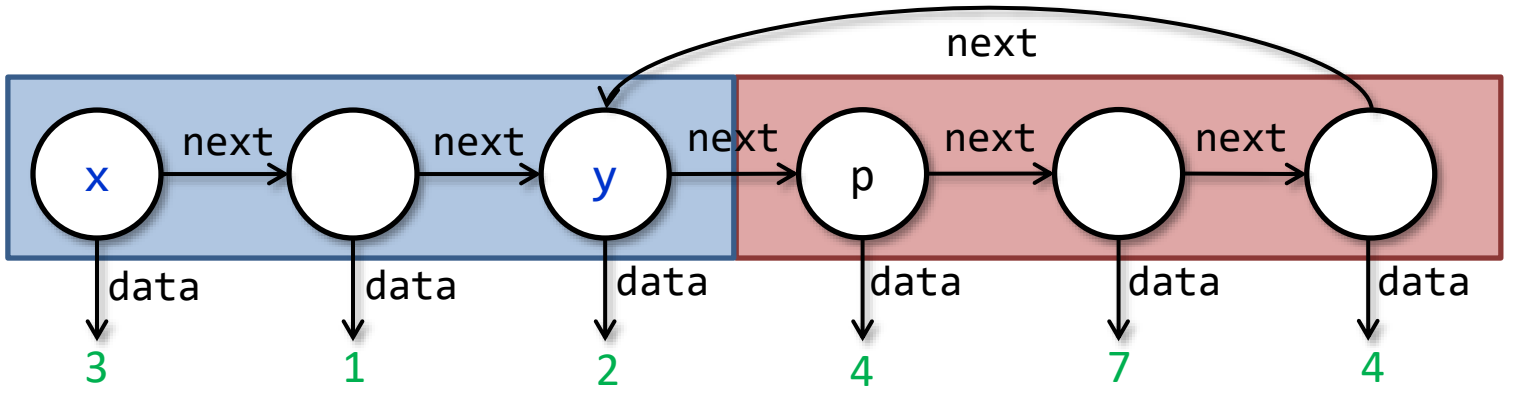
Counterexample for Quicksort Spec.

```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```



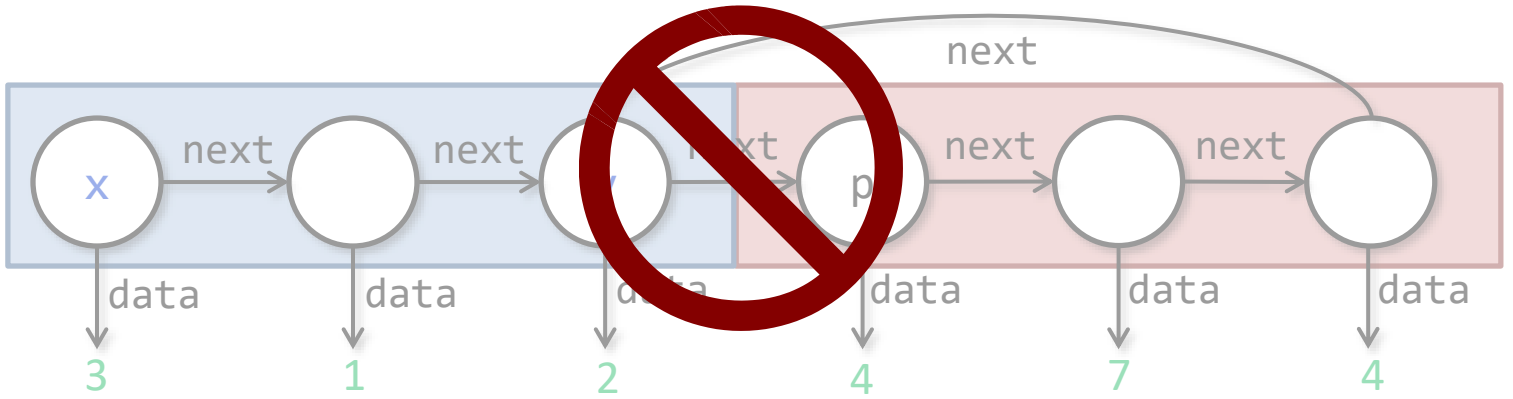
Split with Mixed Specification

```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max) * Btwn(next,x,p,y)
  ensures p ≠ y * min ≤ p.data ≤ max
```



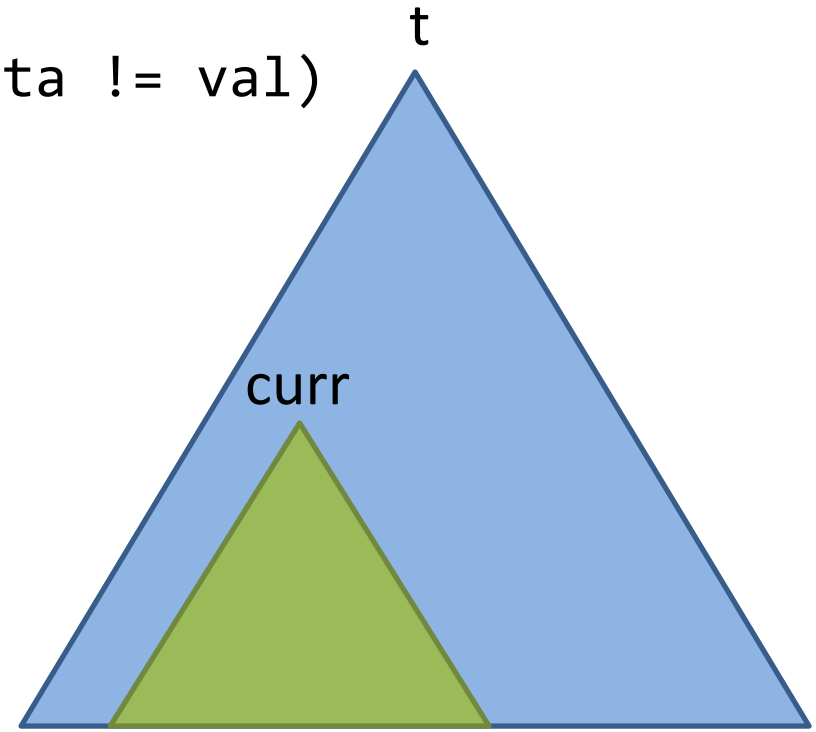
Split with Mixed Specification

```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max) * Btwn(next,x,p,y)
  ensures p ≠ y * min ≤ p.data ≤ max
```



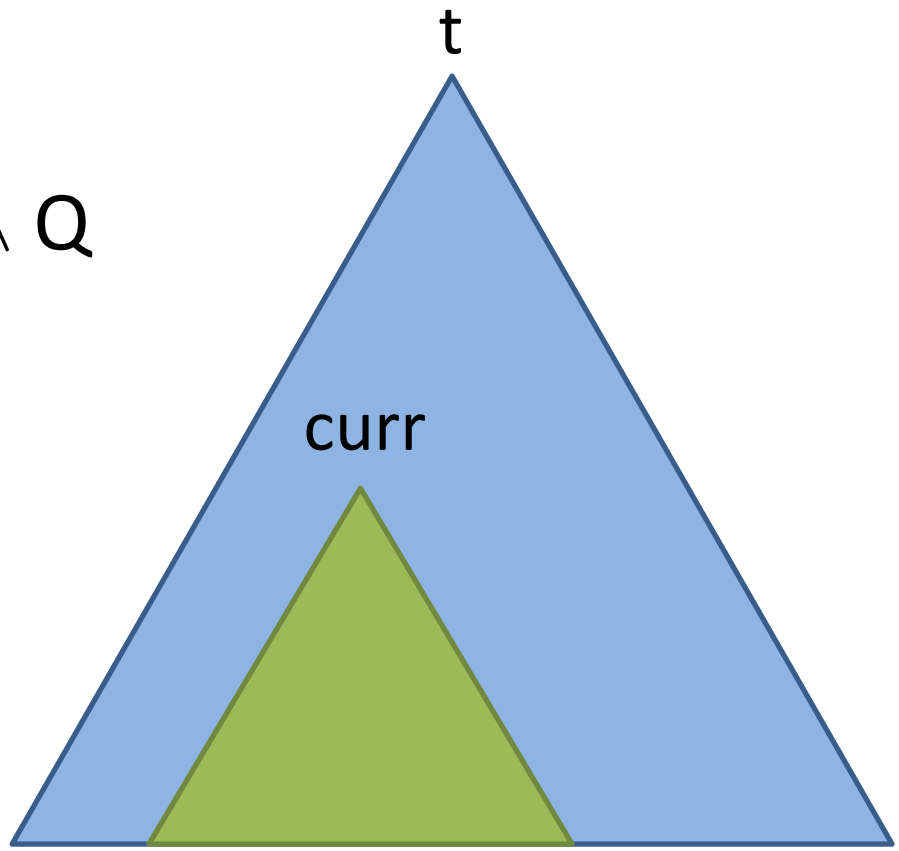
Tail-Recursive Tree Traversal

```
procedure contains(t: Tree, val: Int)
  returns (res: Bool)
  requires tree(t)
  ensures tree(t)
{
  var curr := t;
  while (curr != null && curr.data != val)
    invariant ?
  {
    if (curr.data > val)
      curr := curr.left;
    else if (curr.data < val)
      curr := curr.right;
  }
  return curr != null;
}
```



Poor Man's Magic Wand

$P \text{ -**} Q \equiv$
$(P * \text{true}) \wedge Q$



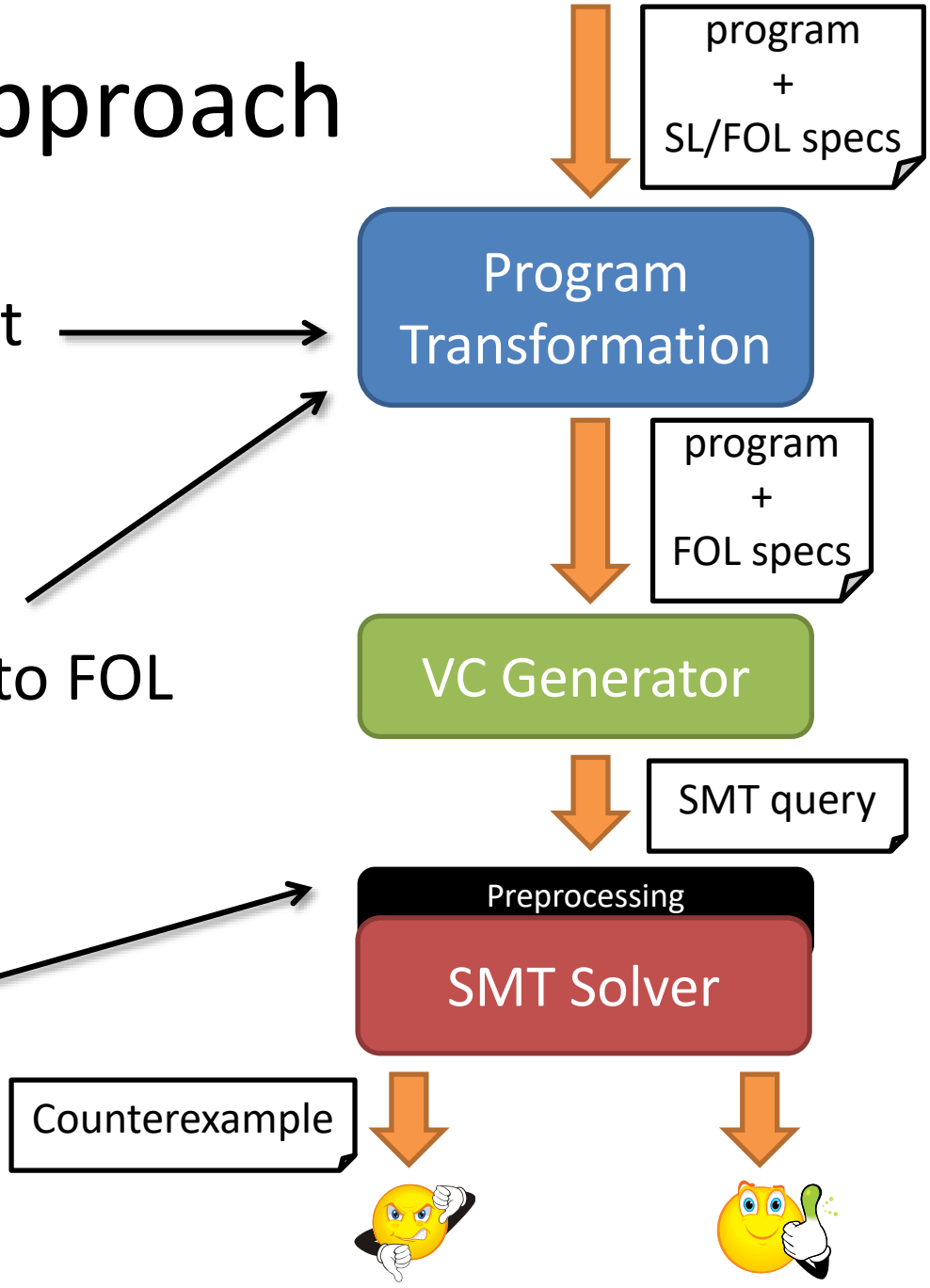

```
tree(curr) * (tree(curr) –* tree(t))
```

Poor Man's Magic Wand

```
procedure contains(t: Tree, val: Int)
  returns (res: Bool)
  requires tree(t)
  ensures tree(t)
{
  var curr := t;
  while (curr != null && curr.data != val)
    invariant tree(curr) -** tree(t)
  {
    if (curr.data > val)
      curr := curr.left;
    else if (curr.data < val)
      curr := curr.right;
  }
  return curr != null;
}
```

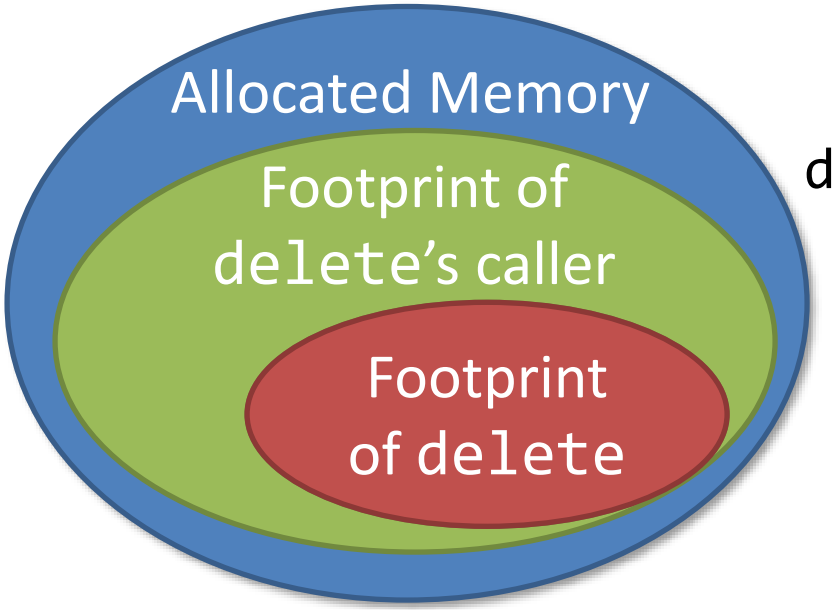
Overview of Approach

1. Make frame rule explicit
2. Translate SL assertions to FOL
3. Decide generated VCs

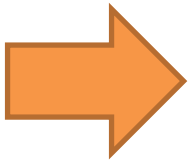


Step 1: Make Frame Rule Explicit

Encoding the Frame Rule




```
delete(x);
```

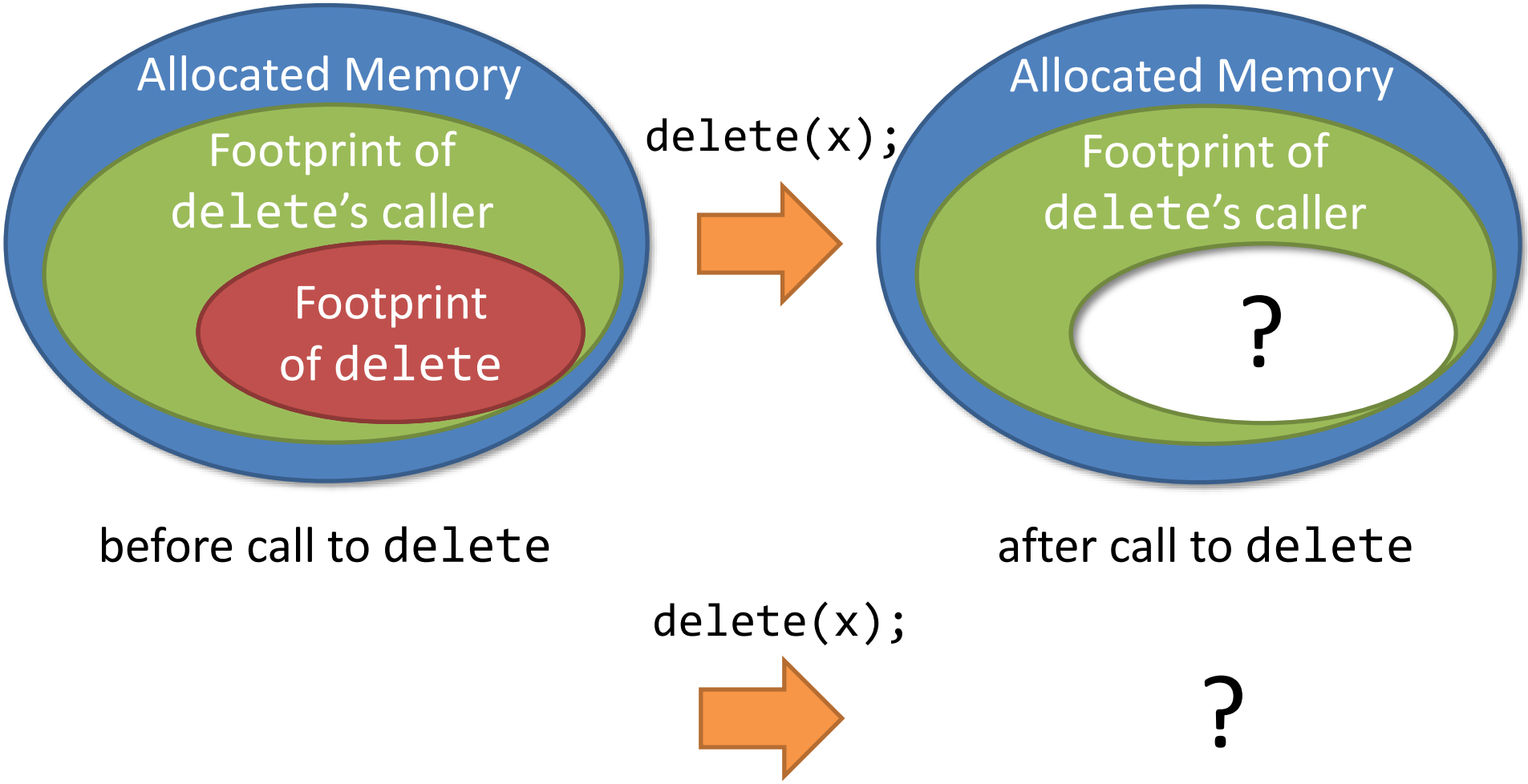


?

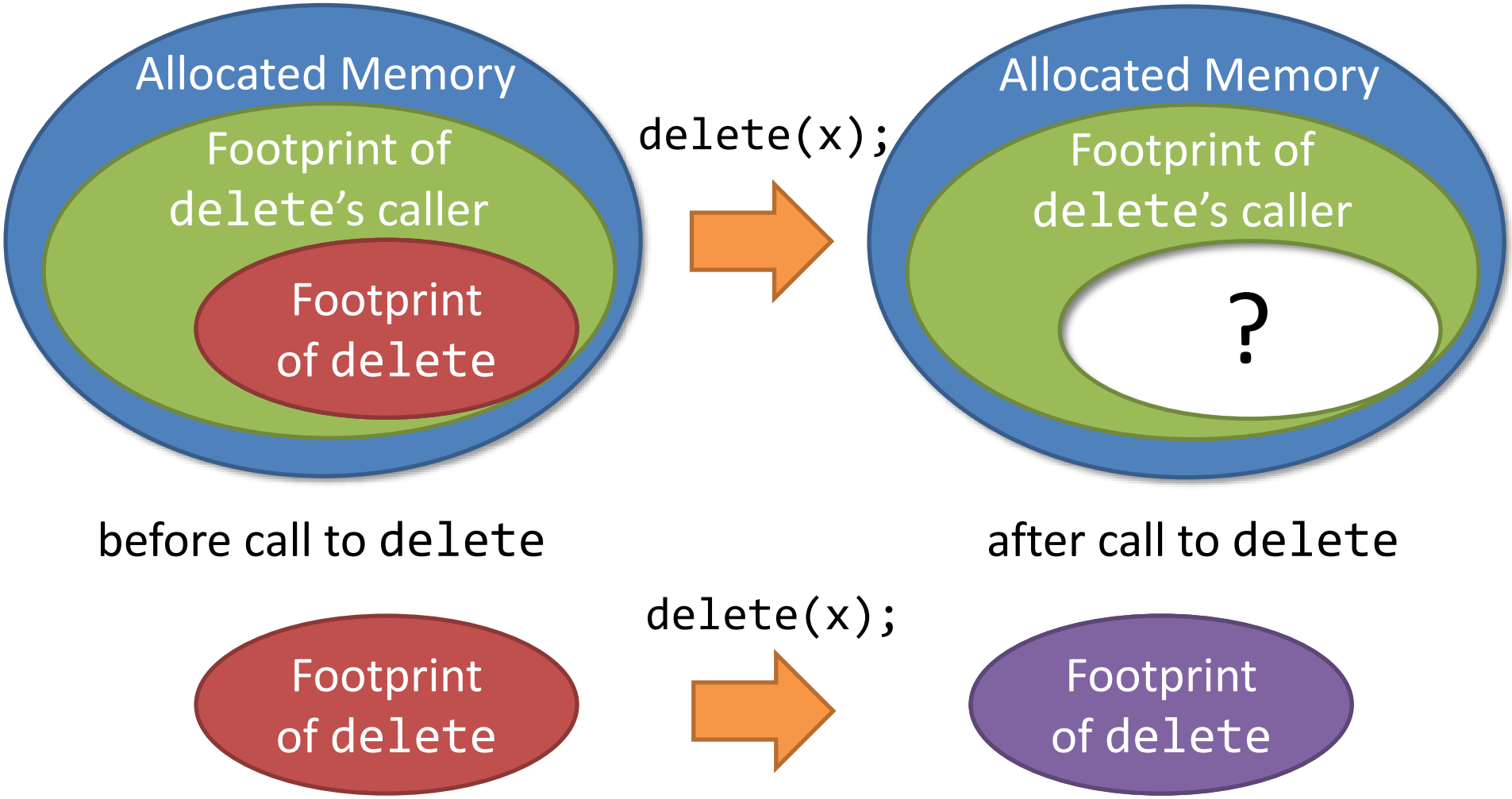
before call to `delete`

after call to `delete`

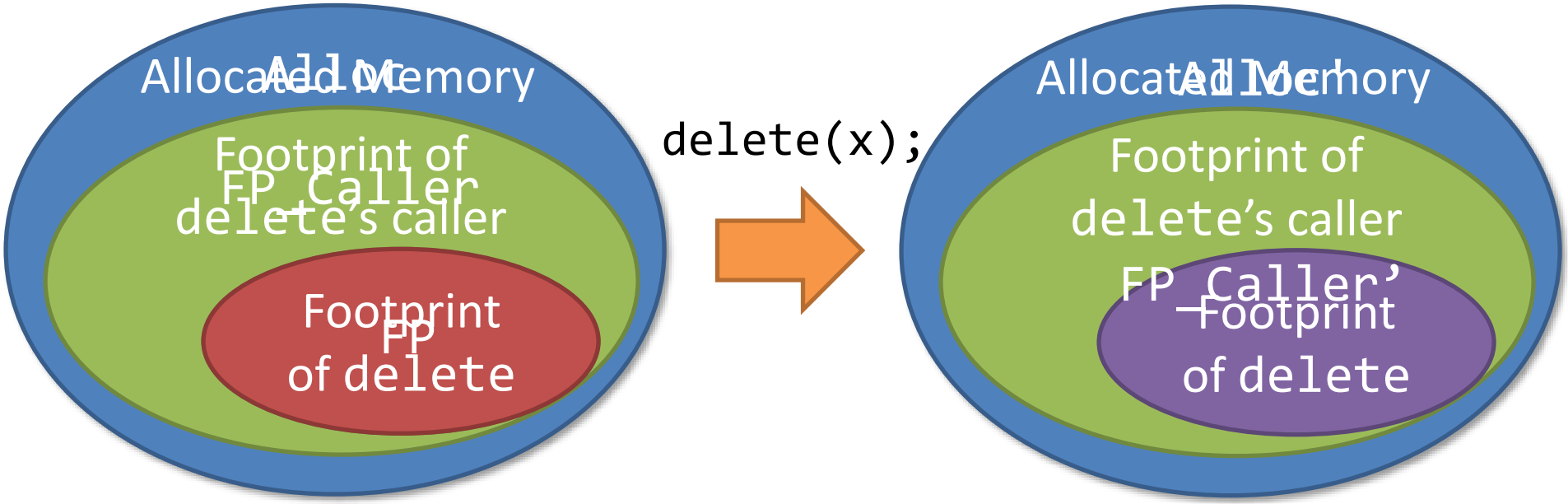
Encoding the Frame Rule



Encoding the Frame Rule



Encoding the Frame Rule

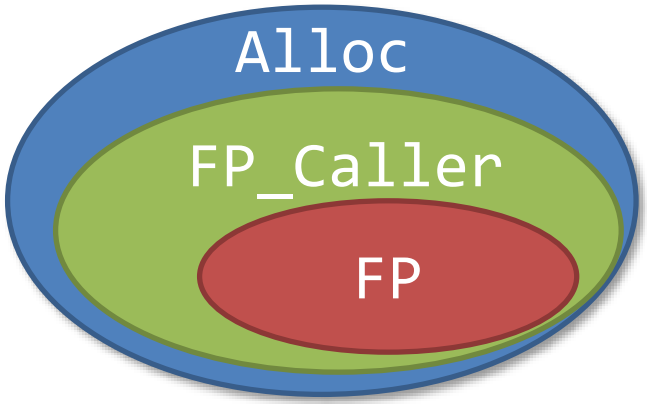


before call to `delete`

after call to `delete`

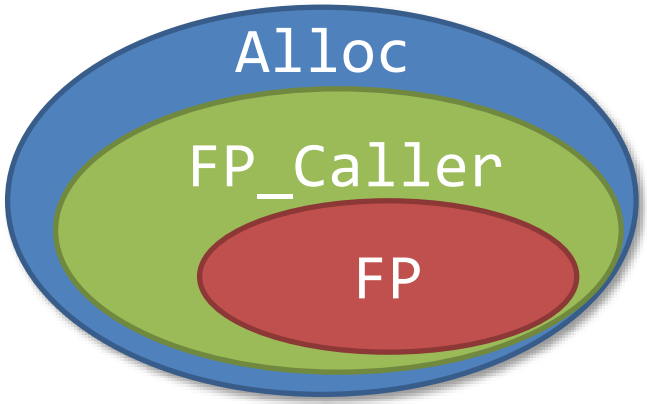
Encoding the Frame Rule

```
procedure delete(x: Node

                                        )

{

  if (x != null) {

    delete(x.next);
    free(x);
  }

}
```



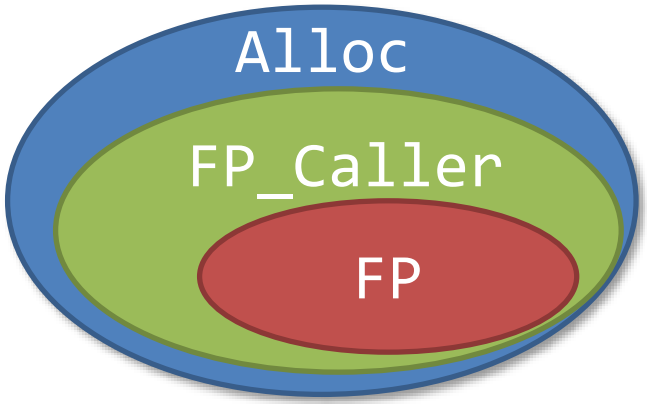
Encoding the Frame Rule

```
ghost var Alloc: Set<Node>;

procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
{

  if (x != null) {

    FP := delete(x.next, FP);
    FP := free(x, FP);
  }

}
```



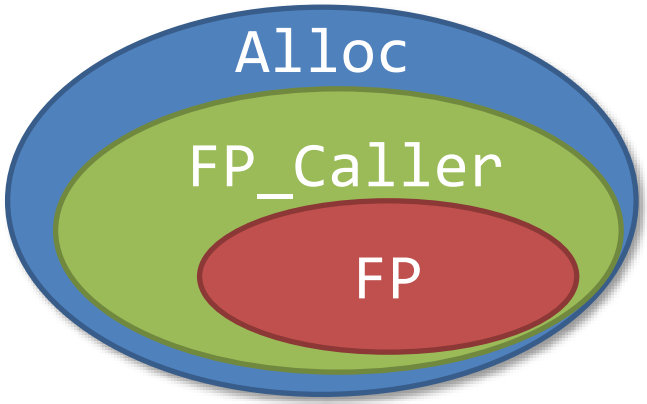
Encoding the Frame Rule

```
ghost var Alloc: Set<Node>;

procedure delete(x: Node)
                  ghost FP_Caller: Set<Node>,
                  implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
{
  FP_Caller' := FP_Caller \ FP;
  if (x != null) {

    FP := delete(x.next, FP);
    FP := free(x, FP);
  }
  FP_Caller' := FP_Caller' ∪ FP;
}
```



Encoding the Frame Rule

```
ghost var Alloc: Set<Node>;

procedure delete(x: Node)
                  ghost FP_Caller: Set<Node>,
                  implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
{
  FP_Caller' := FP_Caller \ FP;
  if (x != null) {
    pure assert x ∈ FP;
    FP := delete(x.next, FP);
    FP := free(x, FP);
  }
  FP_Caller' := FP_Caller' ∪ FP;
}
```



Encoding the Frame Rule

```
procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
  requires lseg(x, null)

  ensures emp

{ ... }
```

Encoding the Frame Rule

```
procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
  requires FP ⊆ FP_Caller
  requires Tr(lseg(x,null), FP)


  ensures Tr(emp, (Alloc ∩ FP) ∪ (Alloc\old(Alloc))
```

```
{ ... }
```

Encoding the Frame Rule

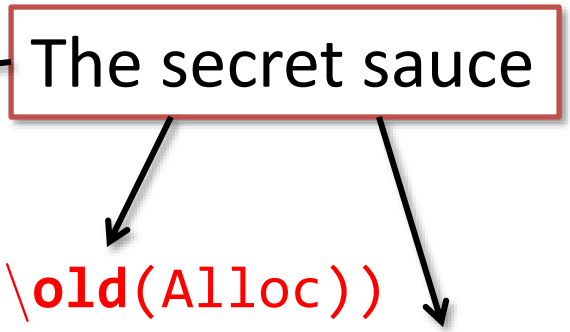
```
procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
  requires FP ⊆ FP_Caller
  free
  free
  ensu
  free
  free
                 (Alloc ∩ FP) ∪ (Alloc\old(Alloc))
  free ensures FP_Caller' ⊆ Alloc
  free ensures null ∉ Alloc
{ ... }
```

Encoding is inspired by implicit dynamic frames [Smans, Jacobs, Piessens, 2008]

Used, e.g., in the VeriCool and Chalice tools

Encoding the Frame Rule

```
procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
  returns (ghost FP_Caller': Set<Node>)
  requires FP ⊆ FP_Caller
  requires Tr(lseg(x,null), FP)
  free requires FP_Caller ⊆ Alloc
  free requires null ∉ Alloc
  ensures Tr(emp, (Alloc ∩ FP) ∪ (Alloc\old(Alloc)))
  free ensures Frame(old(Alloc), FP, old(next), next)
  free ensures FP_Caller' = (FP_Caller\FP) ∪
                (Alloc ∩ FP) ∪ (Alloc\old(Alloc))
  free ensures FP_Caller' ⊆ Alloc
  free ensures null ∉ Alloc
{ ... }
```



Step 2: Translating SL Assertions

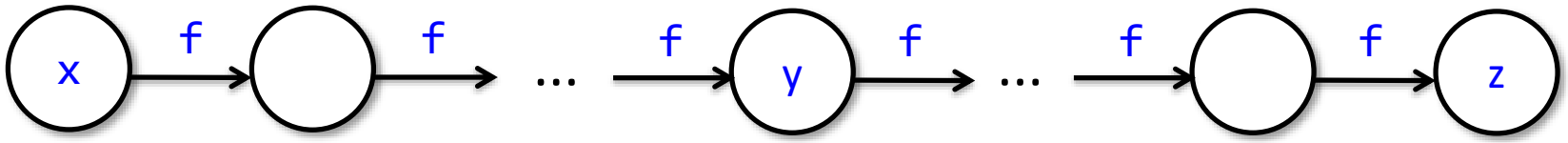
Target of Translation: GRASS (Graph Reachability and Stratified Sets)

- Theory of Reachability in Mutable Graphs
  - encodes structure of the heap (inductive predicates)
- Theory of Stratified Sets
  - encodes frame rule / separating conjunction

Reachability in Mutable Function Graphs

(Extension of [Nelson POPL'83], [Lahiri, Qadeer POPL'08])

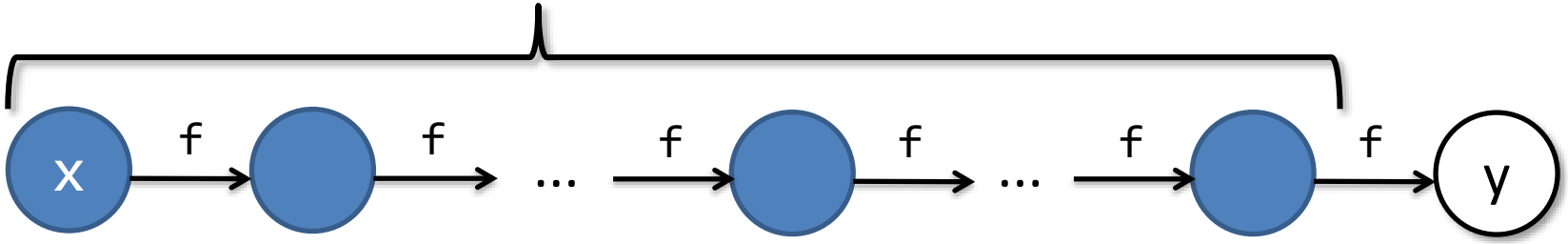
• sel(f, x)	field access	x.f
• upd(f,x,y)	field update	f[x := y]
• Btwn(f,x,y,z)	reachability	$x \xrightarrow{f} y \xrightarrow{f} z$



Btwn(f,x,y,z) means z is reachable from x via f
and y is on the shortest path between x and z

Stratified Sets

- operations: $X \cup Y$, $X \cap Y$, $X \setminus Y$, ...
- predicates: $x \in X$, $X \subseteq Y$, $X = Y$
- literals: $\{ x :: P(x) \}$
  - Examples:
    - $\{ z :: z = x \}$
    - $\{ z :: Btwn(f, x, z, y) \wedge z \neq y \}$



Translating SL Assertions to GRASS

- $Tr(emp, X) \equiv X = \emptyset$
- $Tr(\mathbf{acc}(t), X) \equiv X = t$
- $Tr(F, X) \equiv F \wedge X = \emptyset$ if F is pure
- $Tr(\mathbf{lseg}(x,y), X) \equiv Btwn(next,x,y,y) \wedge X = \{z :: Btwn(next,x,z,y) \wedge z \neq y\}$
- $Tr(F * G, X) \equiv \exists\, Y, Z :: Tr(F, Y) \wedge Tr(G, Z) \wedge X = Y \uplus Z$
- $Tr(F \mathbin{-\!\!*\!\!*} G, X) \equiv \exists\, Y :: Tr(F, Y) \wedge Tr(G, X) \wedge Y \subseteq X$
- $Tr(F \wedge G, X) \equiv Tr(F, X) \wedge Tr(G, X)$
- $Tr(\neg F, X) \equiv \neg Tr(F, X)$

Example: Delete

```
procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
returns (ghost FP_Caller': Set<Node>)
  requires FP ⊆ FP_Caller
  requires Tr(lseg(x,null), FP)
  free requires FP_Caller ⊆ Alloc
  free requires null ∉ Alloc
  ensures Tr(emp, (Alloc ∩ FP) ∪ (Alloc\old(Alloc))
  free ensures Frame(old(Alloc), FP, old(next), next)
  free ensures FP_Caller' = (FP_Caller\FP) ∪
                (Alloc ∩ FP) ∪ (Alloc\old(Alloc))
  free ensures FP_Caller' ⊆ Alloc
  free ensures null ∉ Alloc
{ ... }
```

Example: Delete

```
procedure delete(x: Node,
                 ghost FP_Caller: Set<Node>,
                 implicit ghost FP: Set<Node>)
returns (ghost FP_Caller': Set<Node>)
  requires FP ⊆ FP_Caller
  requires Btwn(next,x,y,y) ∧ FP = {z. Btwn(next,x,z,y) ∧ z ≠ y}
  free requires FP_Caller ⊆ Alloc
  free requires null ∉ Alloc
  ensures (Alloc ∩ FP) ∪ (Alloc\old(Alloc)) = ∅
  free ensures Frame(old(Alloc), FP, old(next), next)
  free ensures FP_Caller' = (FP_Caller\FP) ∪
                (Alloc ∩ FP) ∪ (Alloc\old(Alloc))
  free ensures FP_Caller' ⊆ Alloc
  free ensures null ∉ Alloc
{ ... }
```

Step 3: Deciding GRASS

Dealing with Second-Order Quantifiers

- $Tr(emp, X) \equiv X = \emptyset$
- $Tr(\mathbf{acc}(t), X) \equiv X = t$
- $Tr(F, X) \equiv F \wedge X = \emptyset$ if F is pure
- $Tr(\mathbf{lseg}(x,y), X) \equiv Btwn(next,x,y,y) \wedge X = \{z :: Btwn(next,x,z,y) \wedge z \neq y\}$
- $Tr(F * G, X) \equiv \exists\, Y, Z :: Tr(F, Y) \wedge Tr(G, Z) \wedge X = Y \uplus Z$
- $Tr(F \mathrel{-\!\!*\!\!*} G, X) \equiv \exists\, Y :: Tr(F, Y) \wedge Tr(G, X) \wedge Y \subseteq X$
- $Tr(F \wedge G, X) \equiv Tr(F, X) \wedge Tr(G, X)$
- $Tr(\neg F, X) \equiv \neg Tr(F, X)$

Permission sets are uniquely determined by formula structure

Quantifiers can be eliminated

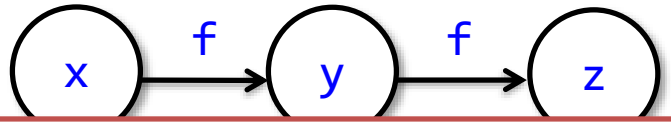
First-Order Axioms for Btwn

- $\forall$ f x. Btwn(f, x, x, x)
- $\forall$ f x. Btwn(f, x, x.f, x.f)
- $\forall$ f x y. Btwn(f, x, y, y) $\Rightarrow$ x = y $\vee$ Btwn(f, x, x.f, y)
- $\forall$ f x y. x.f = x $\wedge$ Btwn(f,x,y,y) $\Rightarrow$ x = y
- $\forall$ f x y. Btwn(f, x, y, x) $\Rightarrow$ x = y
- $\forall$ f x y z. Btwn(f,x,y,y) $\wedge$ Btwn(f,y,z,z) $\Rightarrow$ Btwn(f,x,y,z) $\vee$ Btwn(f,x,z,y)
- $\forall$ f x y z. Btwn(f,x,y,z) $\Rightarrow$ Btwn(f,x,y,y) $\wedge$ Btwn(f,y,z,z)
- $\forall$ f x y z. Btwn(f,x,y,y) $\wedge$ Btwn(f,y,z,z) $\Rightarrow$ Btwn(f,x,z,z)
- $\forall$ f x y z u. Btwn(f,x,y,z) $\wedge$ Btwn(f,y,u,z) $\Rightarrow$ Btwn(f,x,u,z) $\wedge$ Btwn(f,x,y,u)
- $\forall$ f x y z u. Btwn(f,x,y,z) $\wedge$ Btwn(f,x,u,y) $\Rightarrow$ Btwn(f,x,u,z) $\wedge$ Btwn(f,y,u,z)

But I thought transitive closure was not first-order definable!?

Completeness of Axioms for Btwn

- A model of Btwn(f,x,y,z)

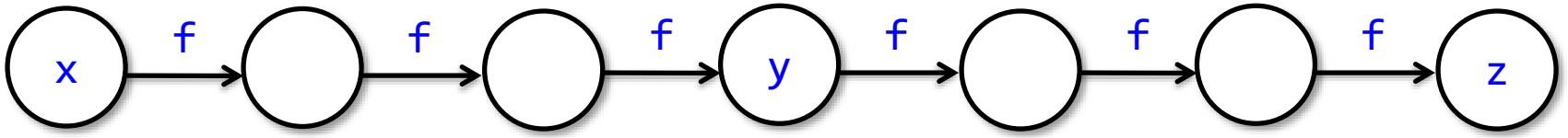


There are arbitrarily large finite models

+

Compactness Theorem $\Rightarrow$ there must also be infinite models

- and another

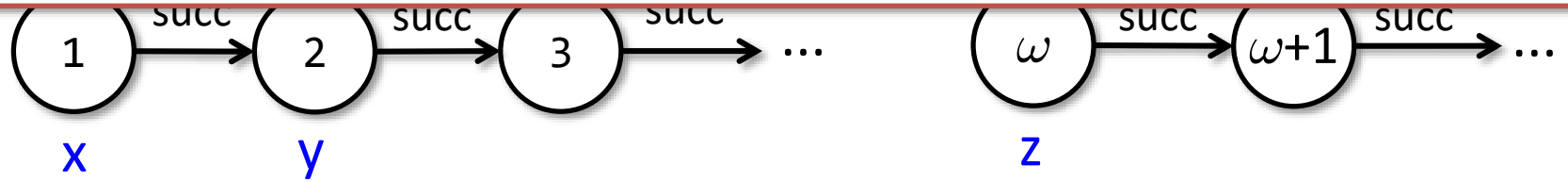


A *Degenerated* Infinite Model M of Btwn(f,x,y,z)

- M = ordinal numbers
- M(f) = succ

Completeness of first-order axioms for Btwn:

- Only infinite models can be degenerated
- If there is a model, then there is also a finite one



$\leq$ is not succ*

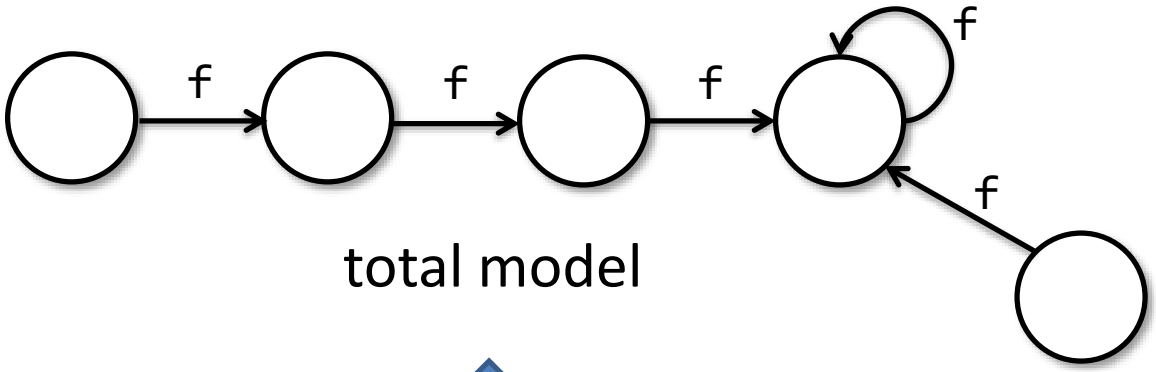
First-Order Axioms for Btwn

Almost in EPR!

- $\forall$ f x. Btwn(f, x, x, x)
- $\forall$ f x. Btwn(f,x, **sel(f,x)**, **sel(f,x)**)
- $\forall$ f x y. Btwn(f, x, y, y) $\Rightarrow$ x = y $\vee$ Btwn(f, x, **sel(f, x)**, y)
- $\forall$ f x y. **sel(f,x)** = x $\wedge$ Btwn(f,x,y,y) $\Rightarrow$ x = y
- $\forall$ f x y. Btwn(f, x, y, x) $\Rightarrow$ x = y
- $\forall$ f x y z. Btwn(f,x,y,y) $\wedge$ Btwn(f,y,z,z) $\Rightarrow$ Btwn(f,x,y,z) $\vee$ Btwn(f,x,z,y)
- $\forall$ f x y z. Btwn(f,x,y,z) $\Rightarrow$ Btwn(f,x,y,y) $\wedge$ Btwn(f,y,z,z)
- $\forall$ f x y z. Btwn(f,x,y,y) $\wedge$ Btwn(f,y,z,z) $\Rightarrow$ Btwn(f,x,z,z)
- $\forall$ f x y z u. Btwn(f,x,y,z) $\wedge$ Btwn(f,y,u,z) $\Rightarrow$ Btwn(f,x,u,z) $\wedge$ Btwn(f,x,y,u)
- $\forall$ f x y z u. Btwn(f,x,y,z) $\wedge$ Btwn(f,x,u,y) $\Rightarrow$ Btwn(f,x,u,z) $\wedge$ Btwn(f,y,u,z)

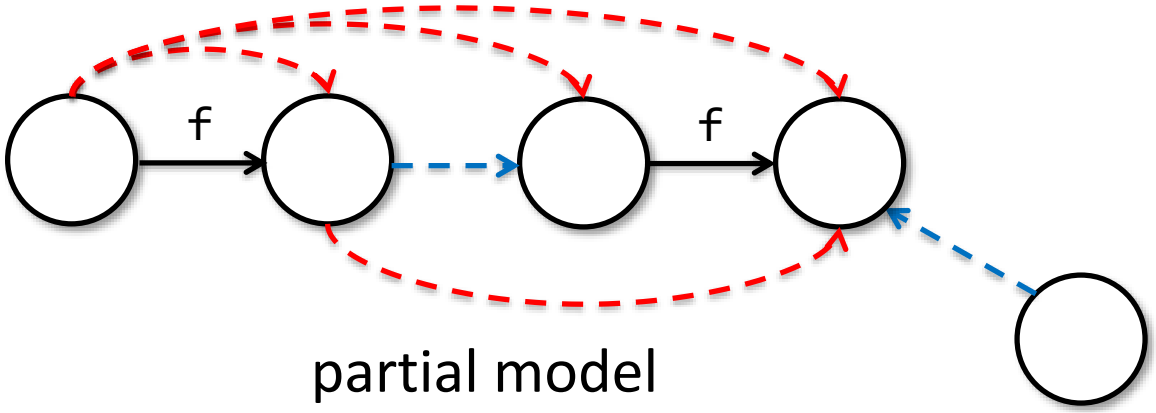
Need to consider more general decidable fragments:
Local Theory Extensions
[Sofronie-Stokkermans, CADE'05], [Bansal et al., CAV'15]

Model Completion for Local Theory Extensions



total model

Yields NP decision procedure for GRASS



partial model

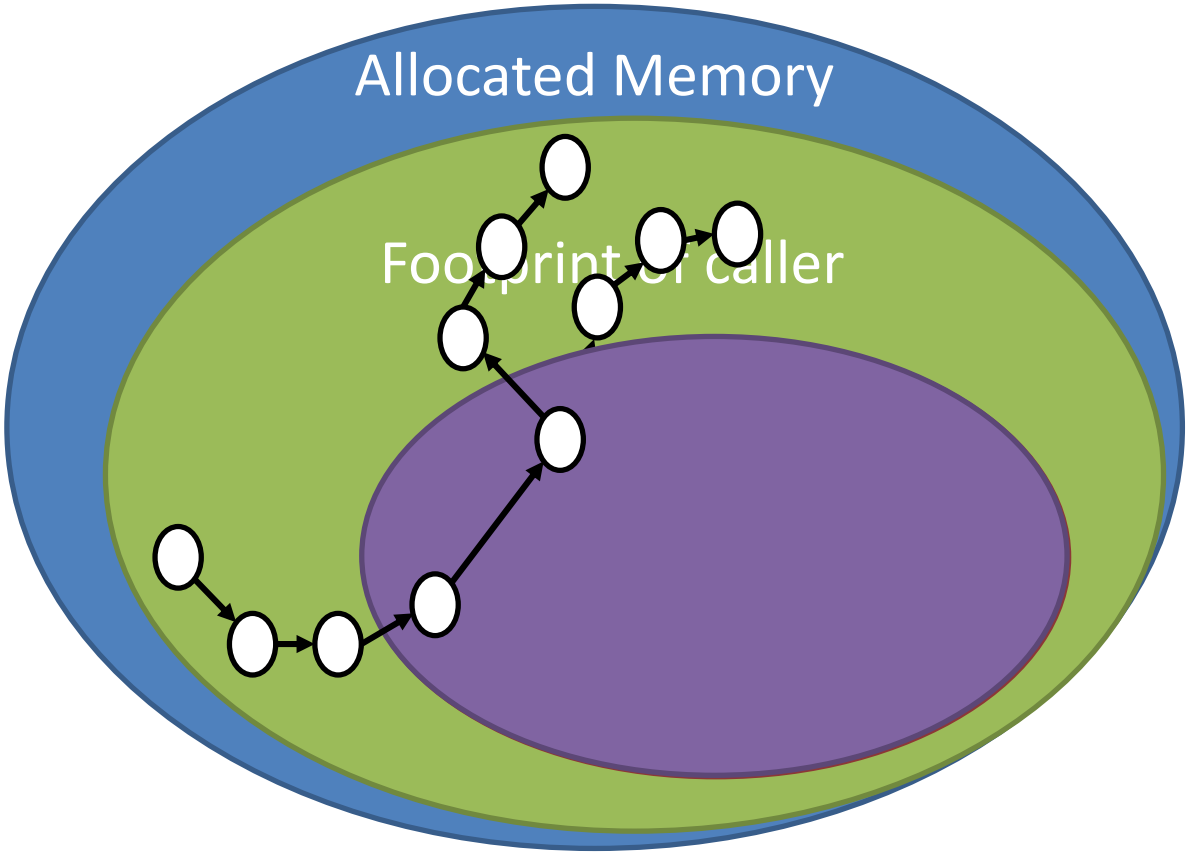
The Frame Predicate

- Frame(Alloc, FP, next, next') $\equiv$
  $\forall$ x. x $\in$ Alloc $\setminus$ FP $\Rightarrow$ x.next == x.next'

- Does not work with finite instantiation.

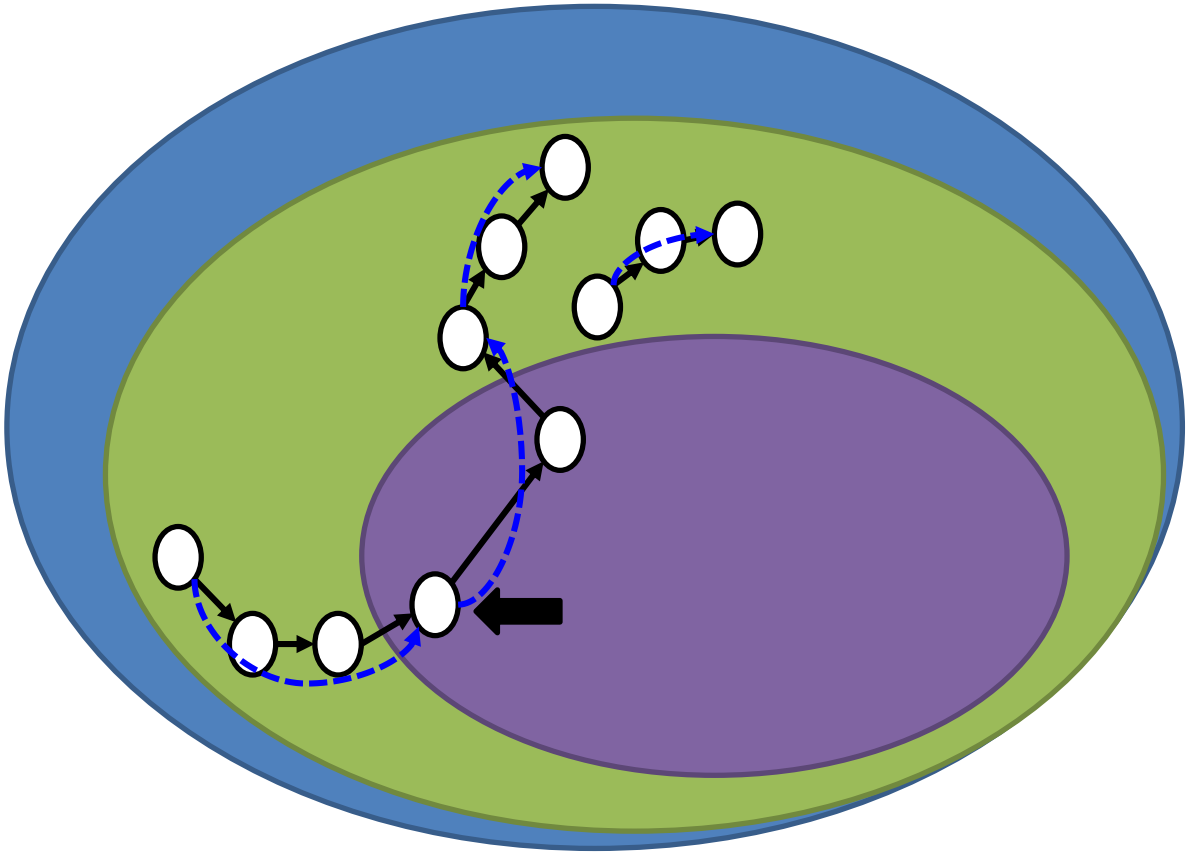
- Need to preserve reachability information in frame.

Axiomatizing the Frame Predicate



Local changes have global effect on reachability

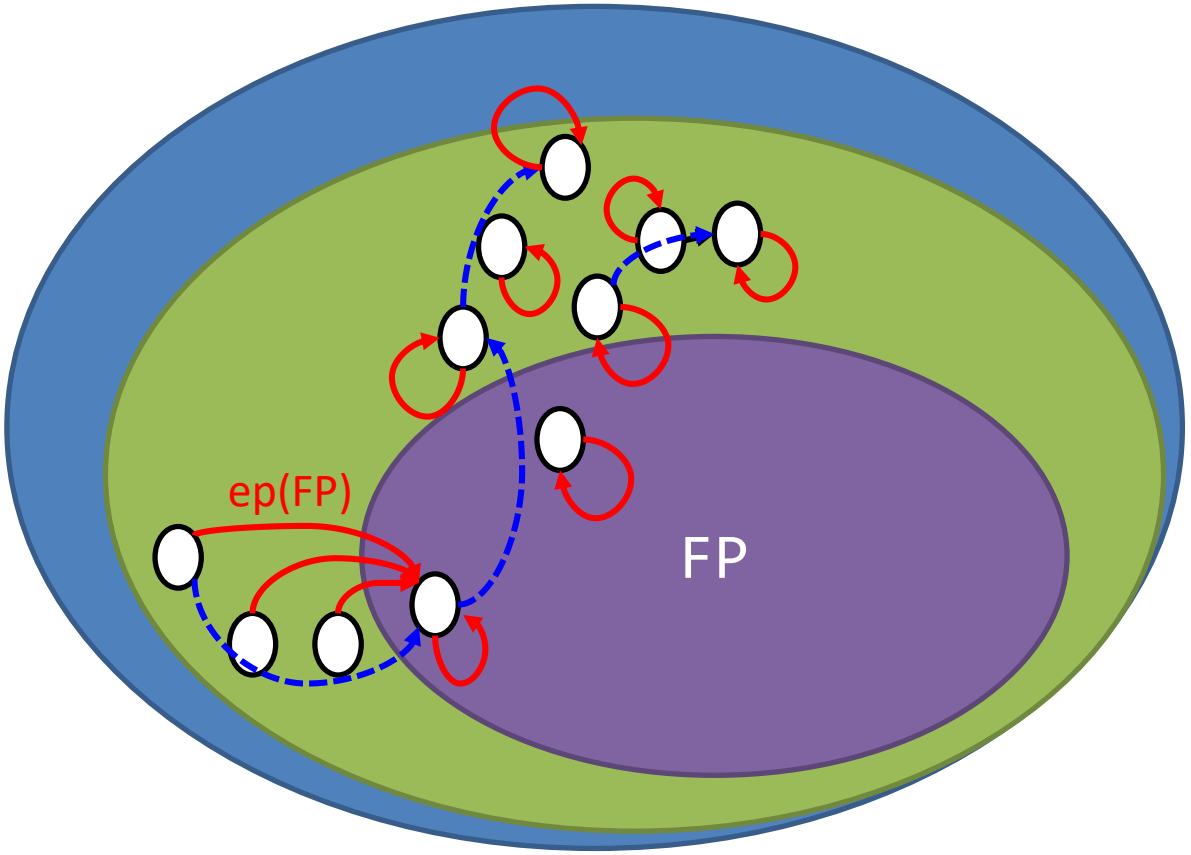
Axiomatizing the Frame Predicate



Local changes have global effect on reachability

Track **entry points (ep)** into footprint **FP**

Axiomatizing the Frame Predicate

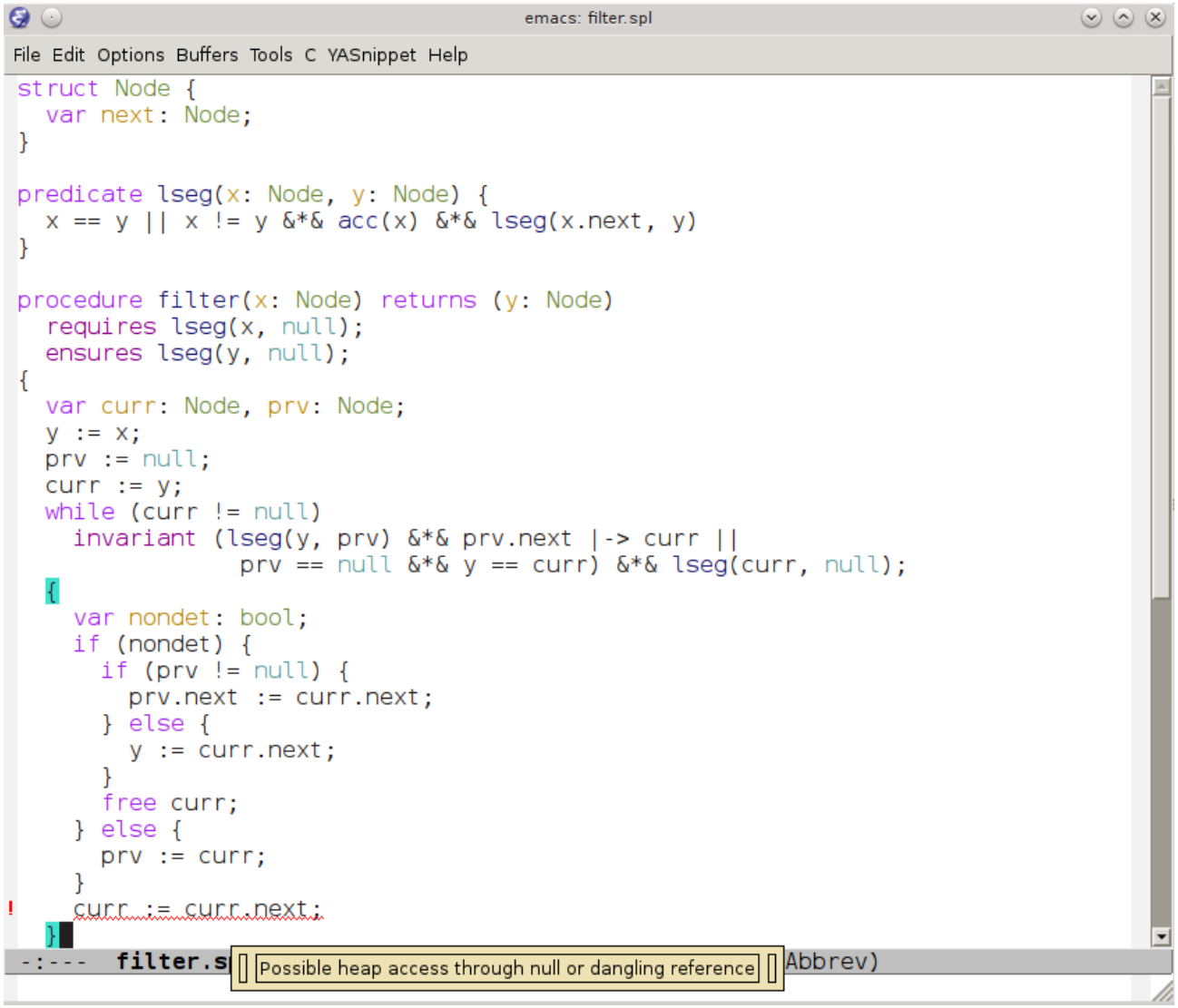


Local changes have global effect on reachability

Track **entry points (ep)** into footprint **FP**

GRASShopper

http://github.com/wies/grasshopper



GRASShopper

http://github.com/wies/grasshopper

- Key features
  - C-like language with mixed SL/FOL specifications
  - Compiles to C
  - Supported back-end solvers: Z3 and CVC4
- Benchmarks (several thousand LoC)
  - List data structures
    - Singly/doubly linked, bounded/sorted, with content, …
    - sorting algorithms, set containers, ...
  - Tree data structures (still in NP!)
    - Binary search trees, skew heaps, union/find, ...
  - Arrays
- Used as backend solver by other tools
  - Viper
  - Starling [Windsor et al. CAV'17]