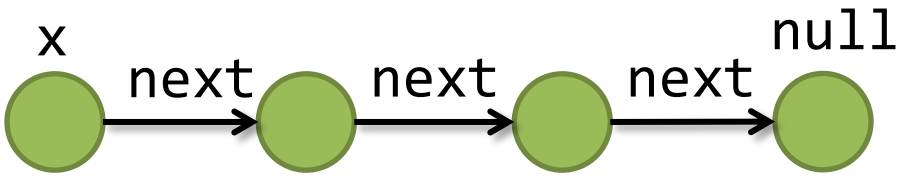
Automating Separation Logic using SMT

Thomas Wies
New York University

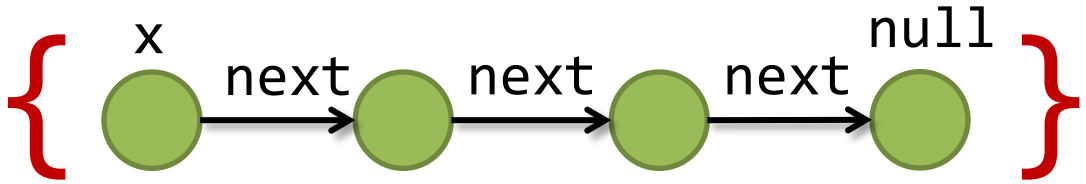
joint work with
Ruzica Piskac (Yale) and Damien Zufferey (MIT)

A Motivating Example

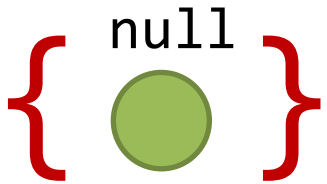
```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```



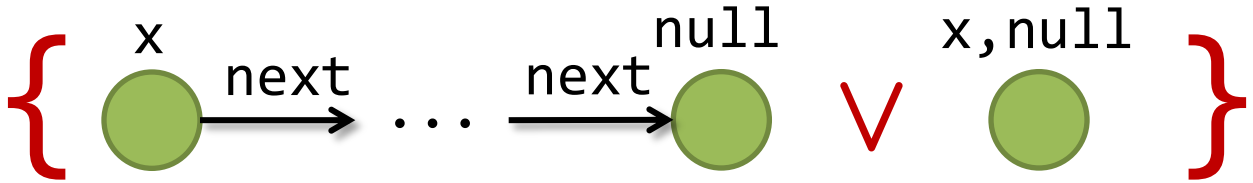
From Hand-Waving to Proofs



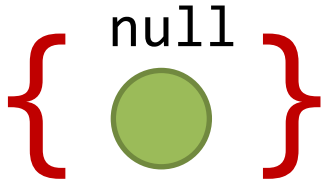

```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```



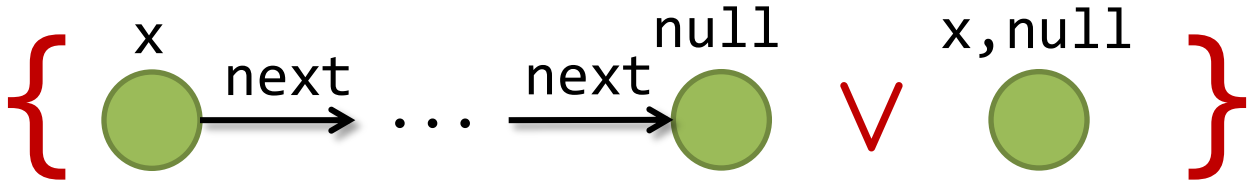
From Hand-Waving to Proofs



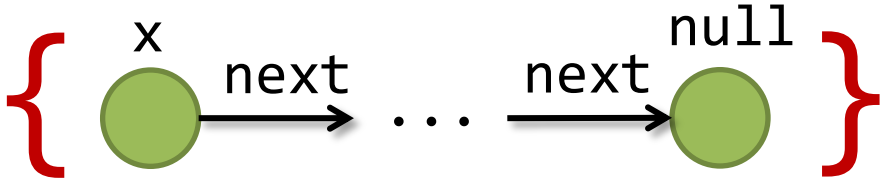

```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```

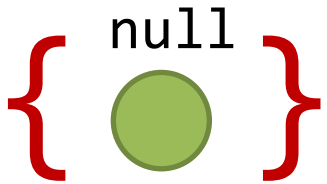


From Hand-Waving to Proofs

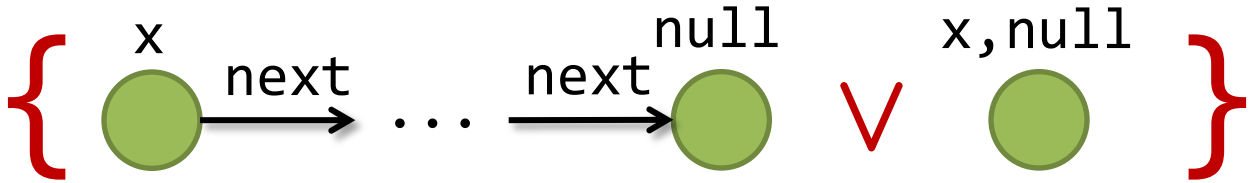



```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```

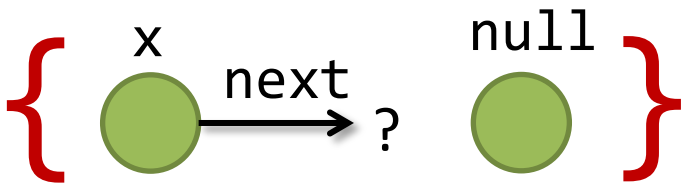


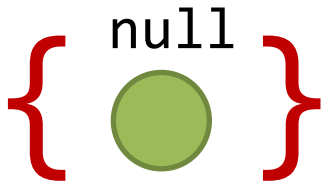


From Hand-Waving to Proofs

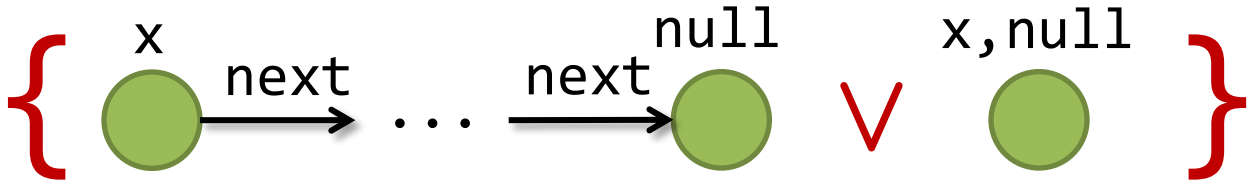



```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```

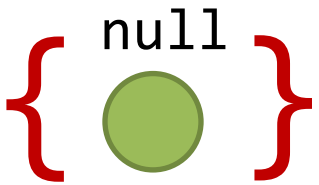


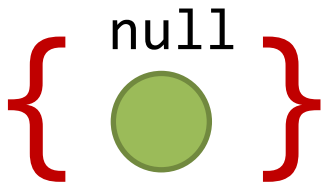


From Hand-Waving to Proofs

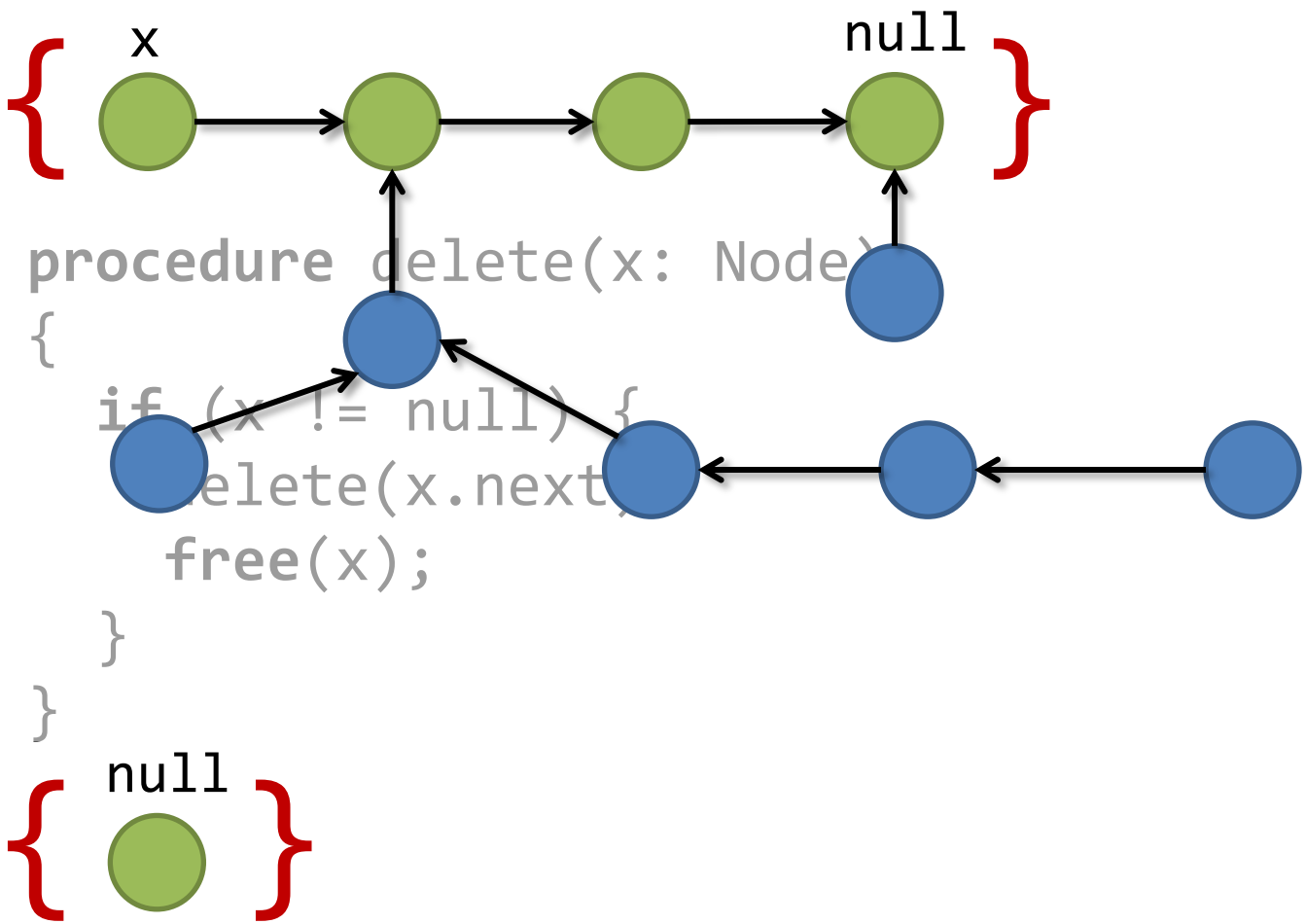



```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```





From Hand-Waving to Proofs

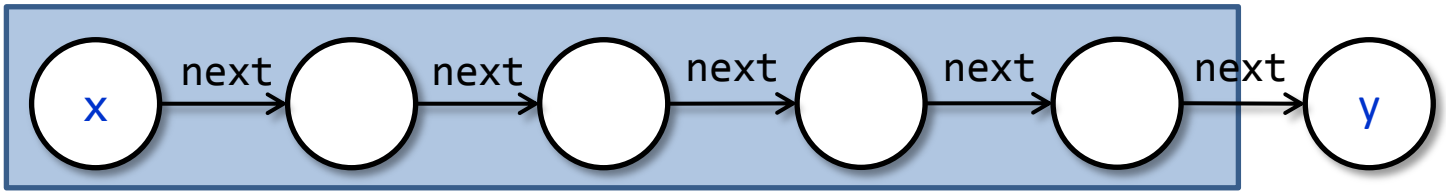


Separation Logic (SL)

[O'Hearn, Reynolds, Yang, 2001]

- acyclic list segment

```
lseg(x, y) =
    x = y ∨
    x ≠ y * acc(x) * lseg(x.next, y)
```



SL assertions describe heap regions (heaplets)

Delete with SL Specification

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
{ emp }
```

Frame Rule

```
{ lseg(x,null) * lseg(y,null) }
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
{ emp * lseg(y,null) }
```

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
{ emp }
```

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {  ⟷ {lseg(x,null) ∧ x≠null}
    delete(x.next);
    free(x);
  }
}
{ emp }
```

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

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

$\longleftrightarrow$ $\{\mathtt{lseg(x,null)} \wedge \mathtt{x \neq null}\}$

$\mathtt{lseg(x,null)} \wedge \mathtt{x \neq null} \vdash \mathtt{lseg(x.next,\ null)}$

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {  ⟷ {lseg(x,null) ∧ x≠null}
    delete(x.next);
    free(x);
  }
}
```

Frame inference:

$\mathtt{lseg(x,null)} \wedge \mathtt{x \neq null} \vdash \mathtt{lseg(x.next,\ null)} * ?$

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

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

$\longleftrightarrow$ $\{lseg(x,null) \wedge x \neq null\}$

Frame inference:

$acc(x) * lseg(x.next, null) \vdash lseg(x.next, null) * ?$

$lseg(x,null) \wedge x \neq null \vdash lseg(x.next, null) * ?$

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {  ⟷ {lseg(x,null) ∧ x≠null}
    delete(x.next);
    free(x);
  }
}
```

Frame inference:

acc(x) ⊢ ?

acc(x) * lseg(x.next, null) ⊢ lseg(x.next, null) * ?

lseg(x,null) ∧ x≠null ⊢ lseg(x.next, null) * ?

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {  ⟷ {lseg(x,null) ∧ x≠null}
    delete(x.next);
    free(x);
  }
}
```

Frame inference: ? = acc(x)

acc(x) ⊢ ?

acc(x) * lseg(x.next, null) ⊢ lseg(x.next, null) * ?

lseg(x,null) ∧ x≠null ⊢ lseg(x.next, null) * ?

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

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

{acc(x) * emp}

Frame inference: ? = acc(x)

acc(x) ⊢ ?

acc(x) * lseg(x.next, null) ⊢ lseg(x.next, null) * ?

lseg(x,null) ∧ x≠null ⊢ lseg(x.next, null) * ?

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
{ emp }
```

$\longleftrightarrow$ {emp * emp}

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn, 2005]

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
{ emp }
```

Why SMT?

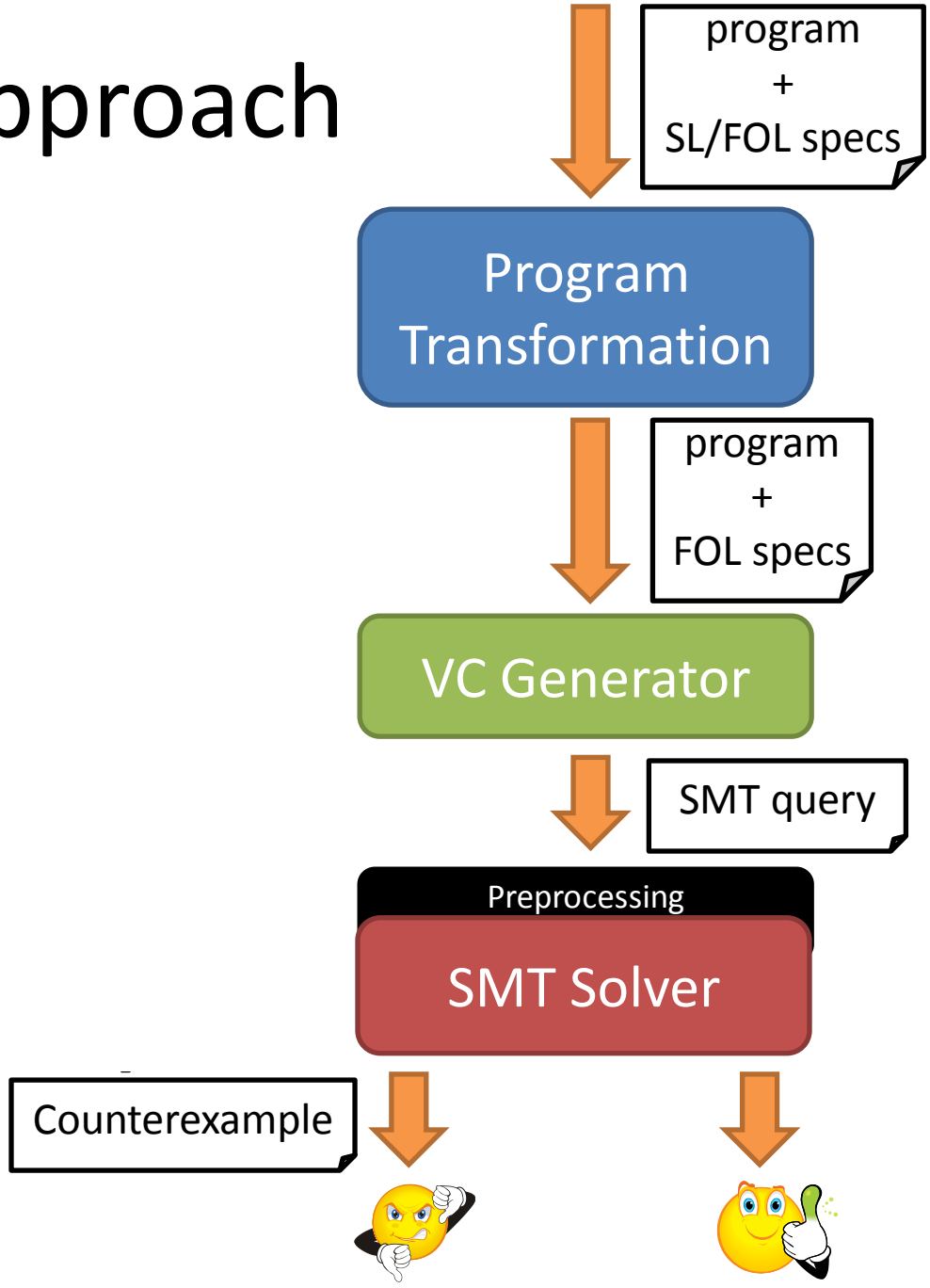
	Frontend / Specification	**Backend / Solver**
SL	+ succinct + intuitive	- tailor-made solvers - difficult to extend + local reasoning (frame inference)
FOL	+ flexible - complex	+ standardized solvers (SMT-LIB) + extensible (e.g. Nelson-Oppen)

Why SMT?

	Frontend / Specification	Backend / Solver
SL	+ succinct + intuitive	- tailor-made solvers - difficult to extend + local reasoning (frame inference)
FOL	+ flexible - complex	+ standardized solvers (SMT-LIB) + extensible (e.g. Nelson-Oppen)

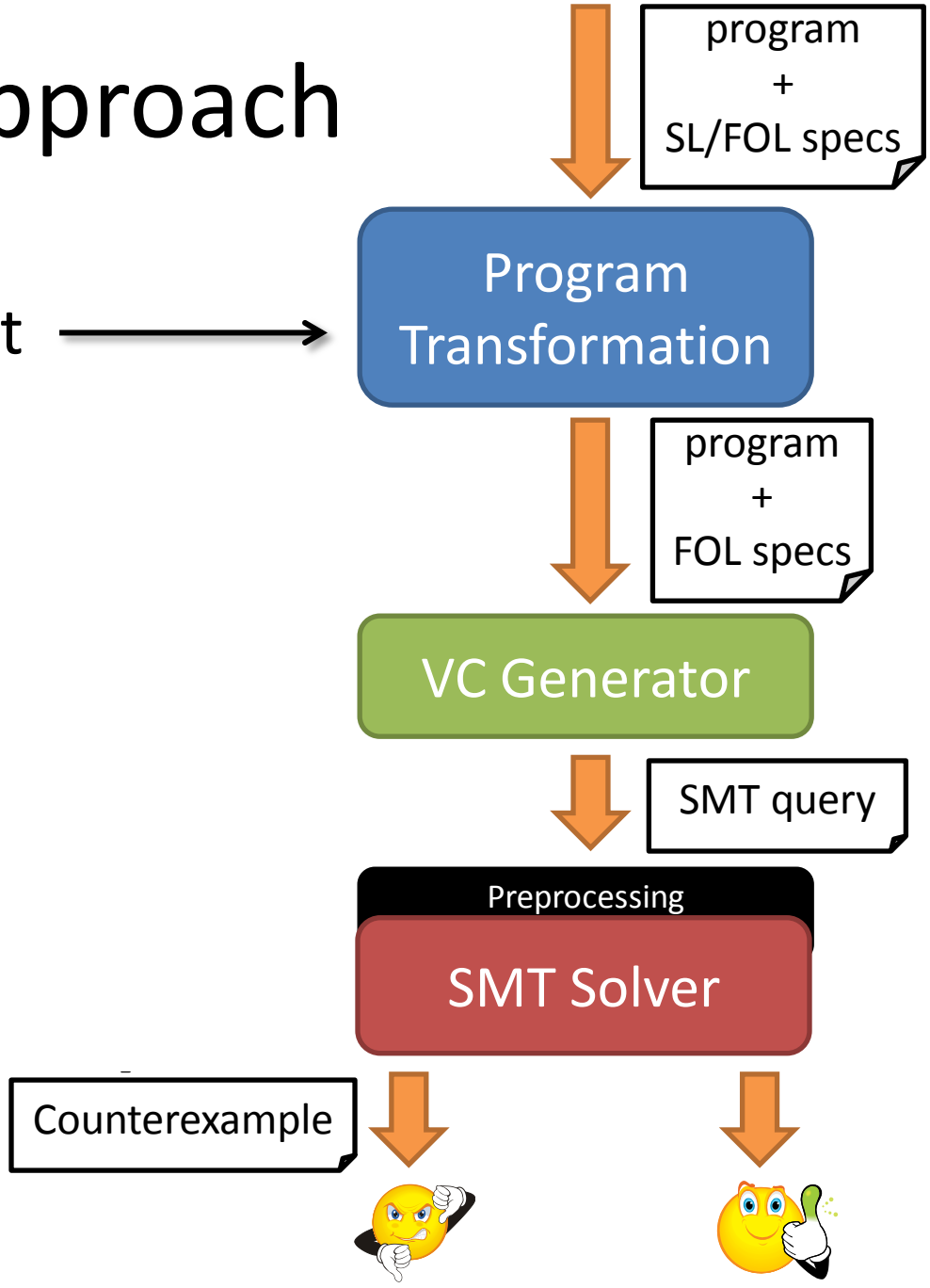
- Strong theoretical guarantees:
  sound, **complete**, **tractable complexity (NP)**
- Mixed specs: escape hatch when SL is not suitable.

Overview of Approach



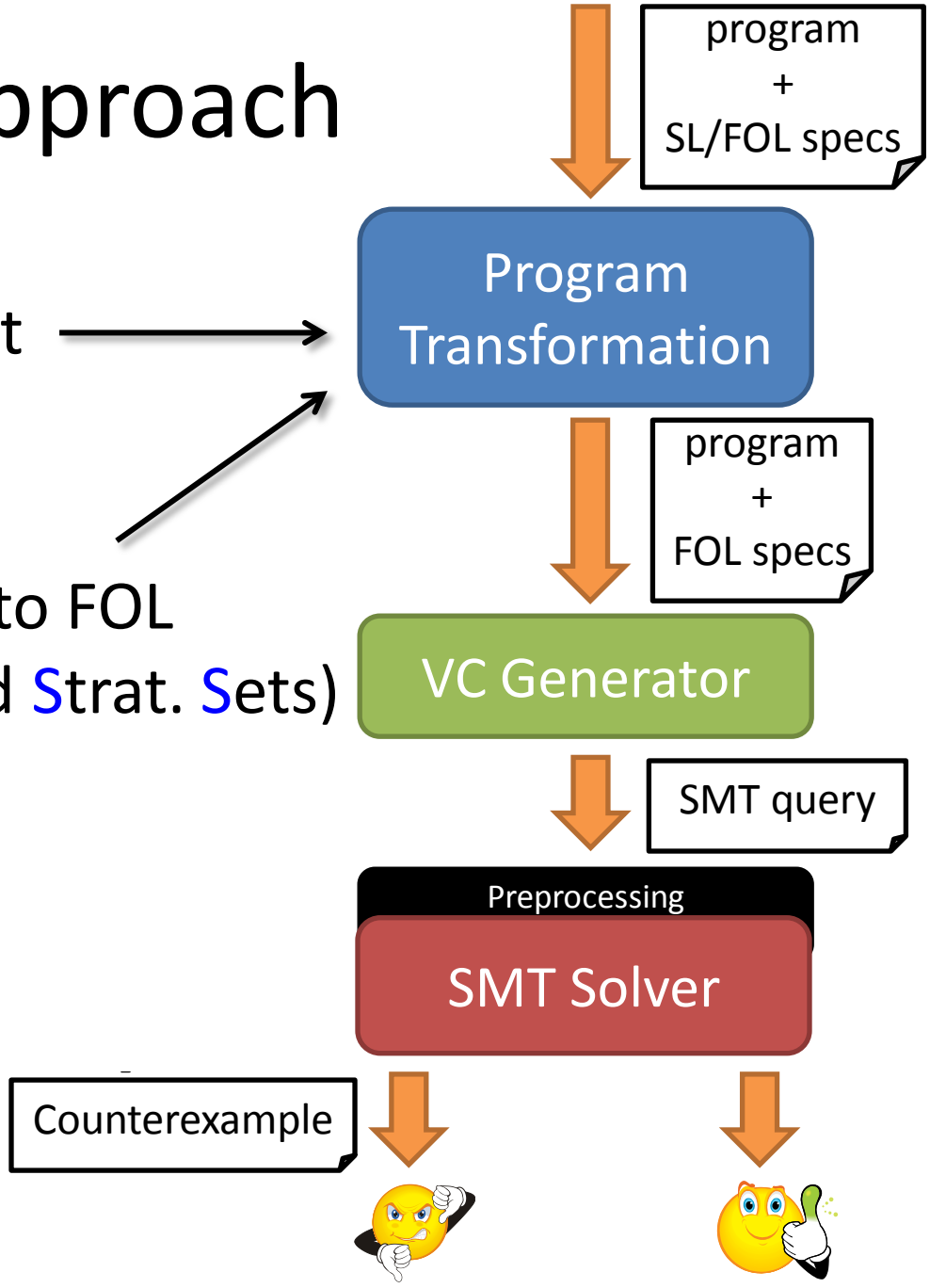
Overview of Approach

1. Make frame rule explicit
   [TACAS'14]



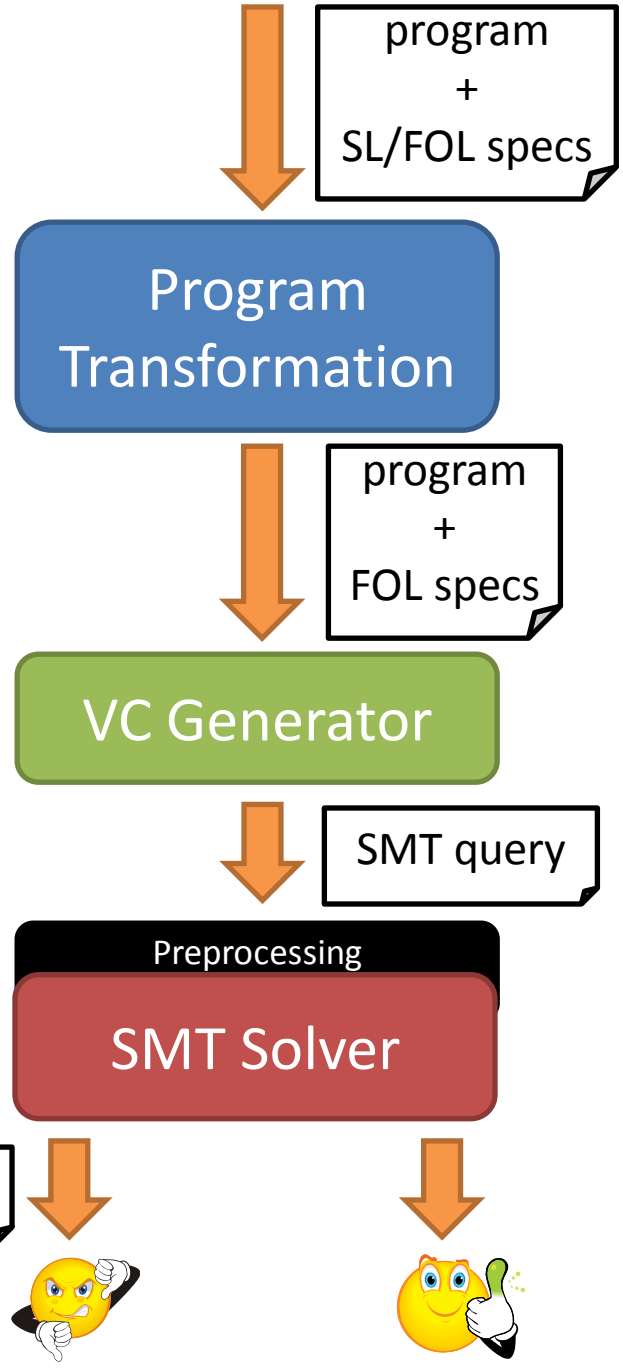
Overview of Approach

1. Make frame rule explicit
   [TACAS'14]
2. Translate SL assertions to FOL
   (Graph Reachability and Strat. Sets)
   [CAV'13]



Overview of Approach

1. Make frame rule explicit
   [TACAS'14]
2. Translate SL assertions to FOL
   (Graph Reachability and Strat. Sets)
   [CAV'13]
3. Decide generated VCs
   [CAV'13] + [TACAS'13] + [CAV'14] + [CAV'15]

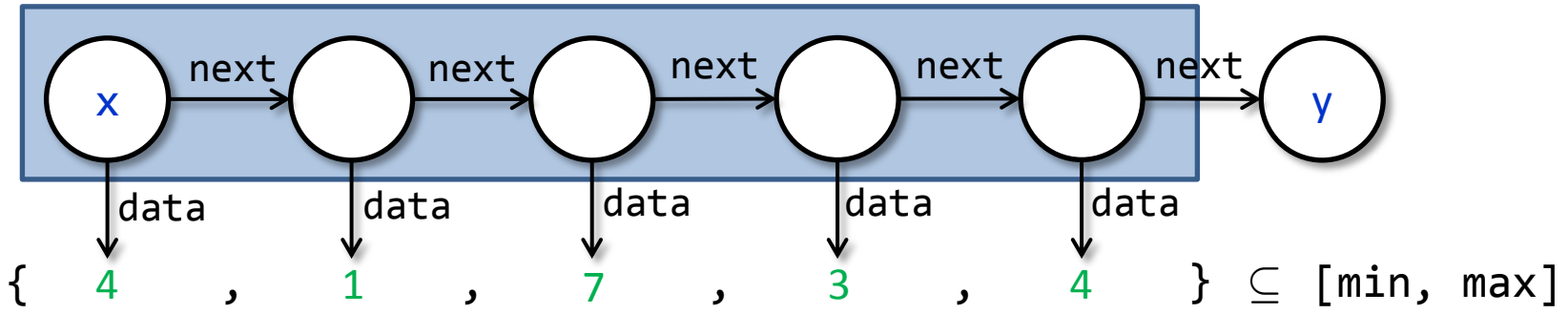


Reasoning about Heap and Data

Inductive Predicates with Data

- bounded list segment

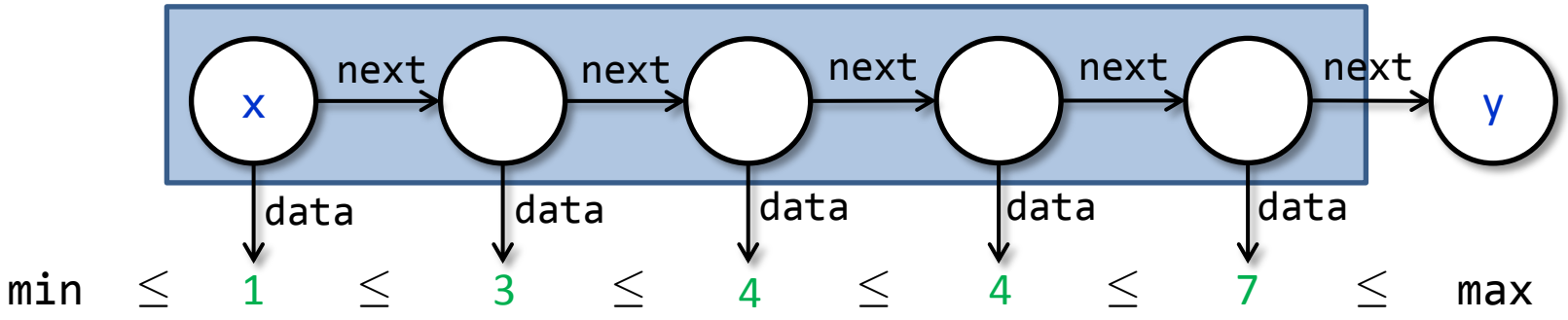
```
bnd_lseg(x, y, min, max) =
    x = y ∨
    x ≠ y * acc(x) * min ≤ x.data ≤ max *
    bnd_lseg(x.next, y, min, max)
```



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)
```

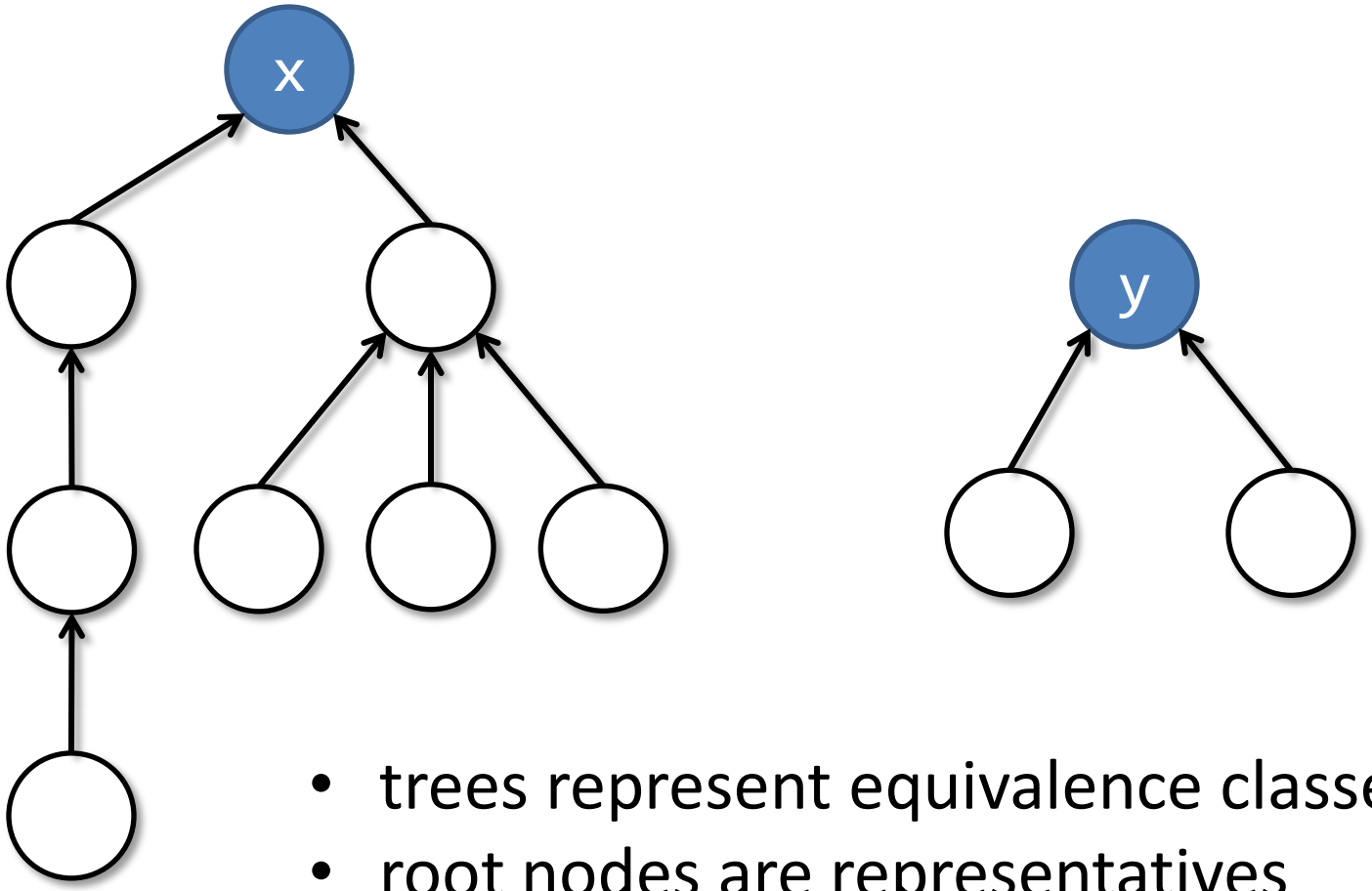


Example: Quicksort

```
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

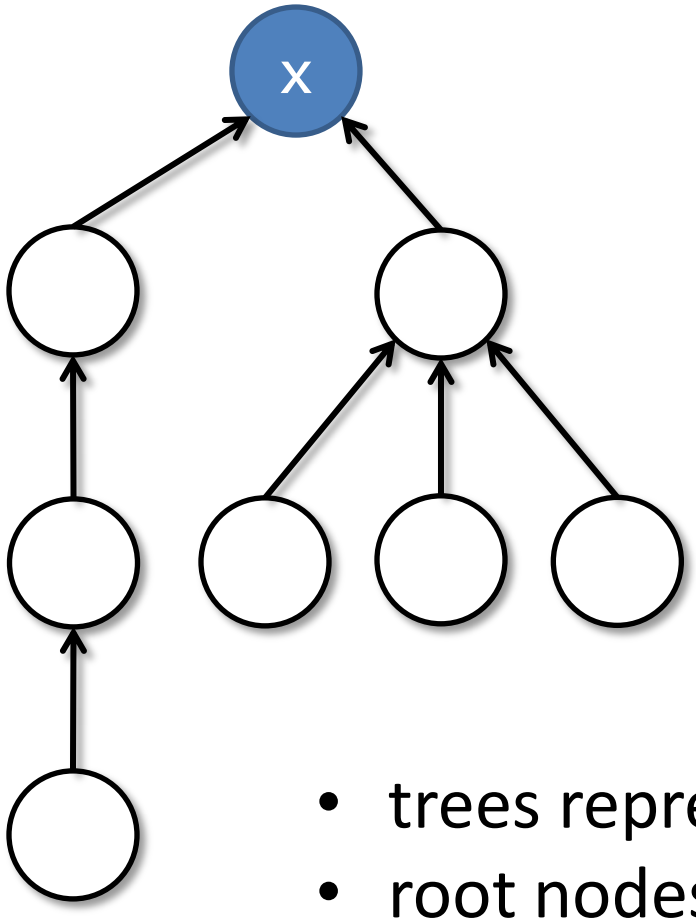
Example: Union/Find Data Structure

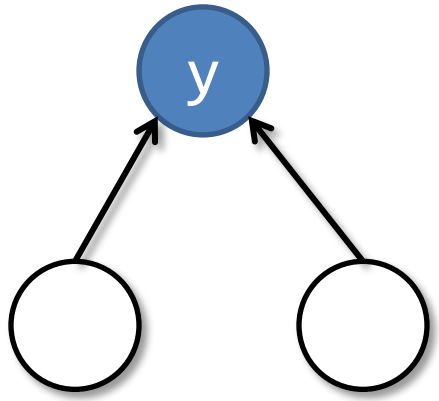


- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure

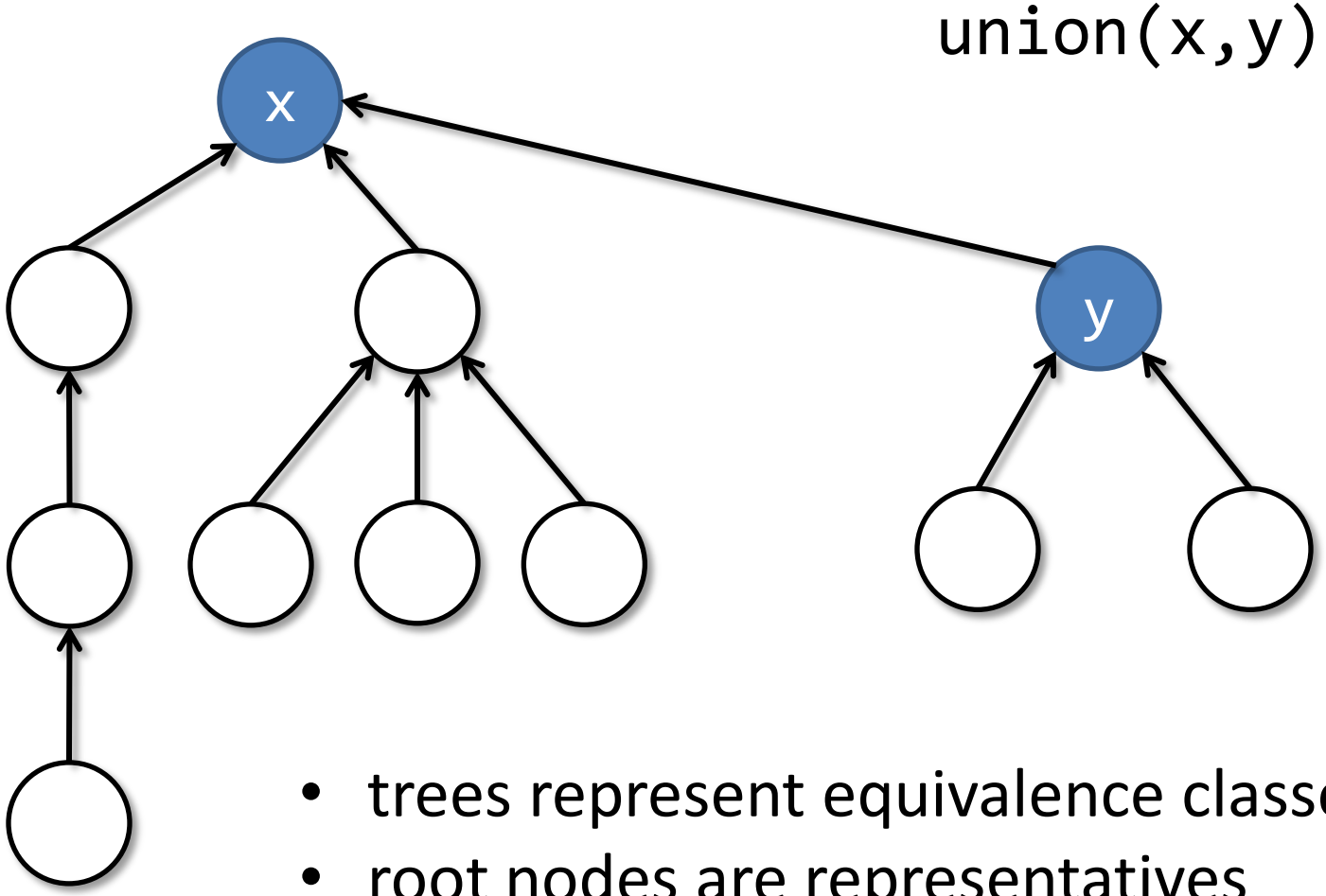
```
union(x,y);
```





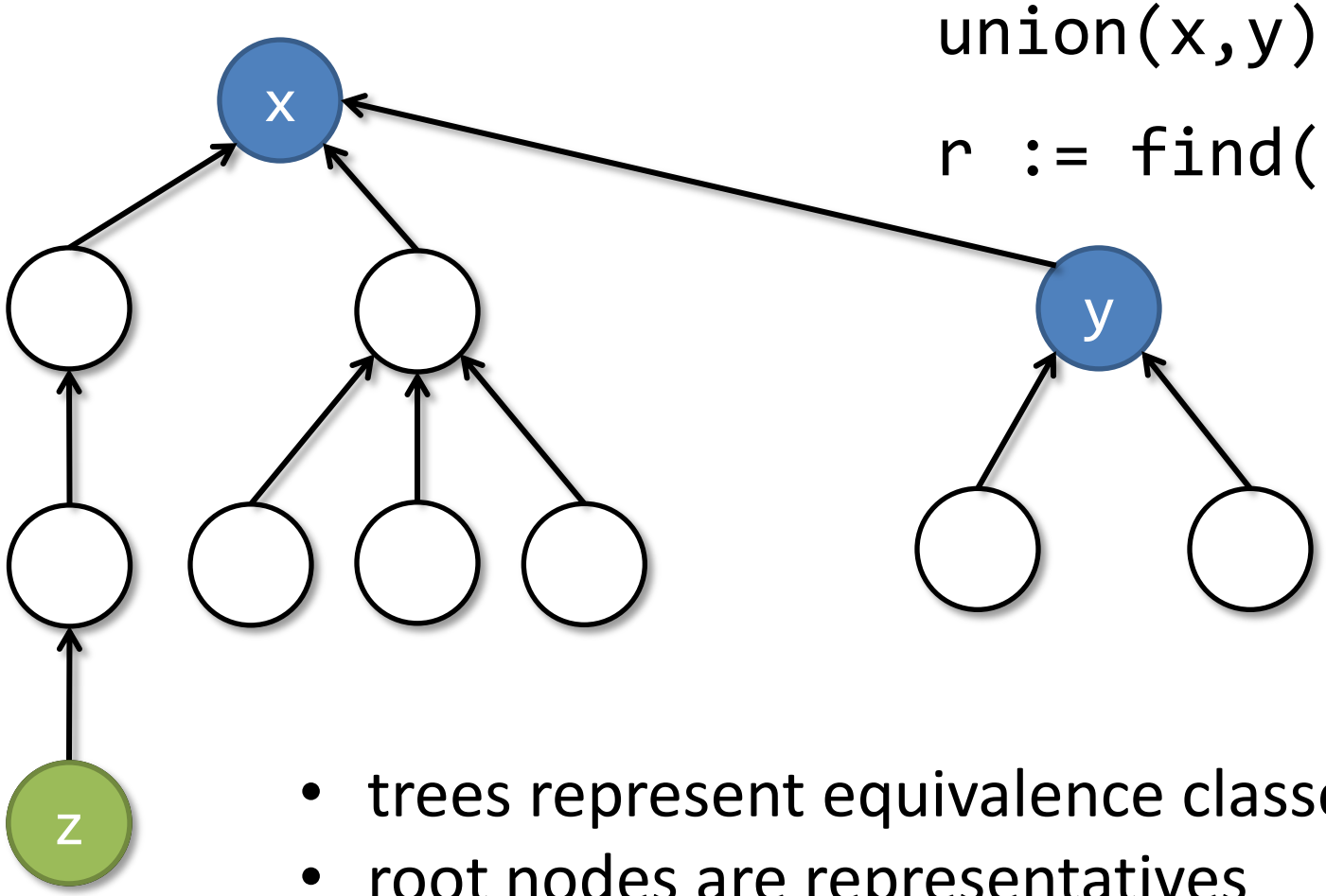
- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure



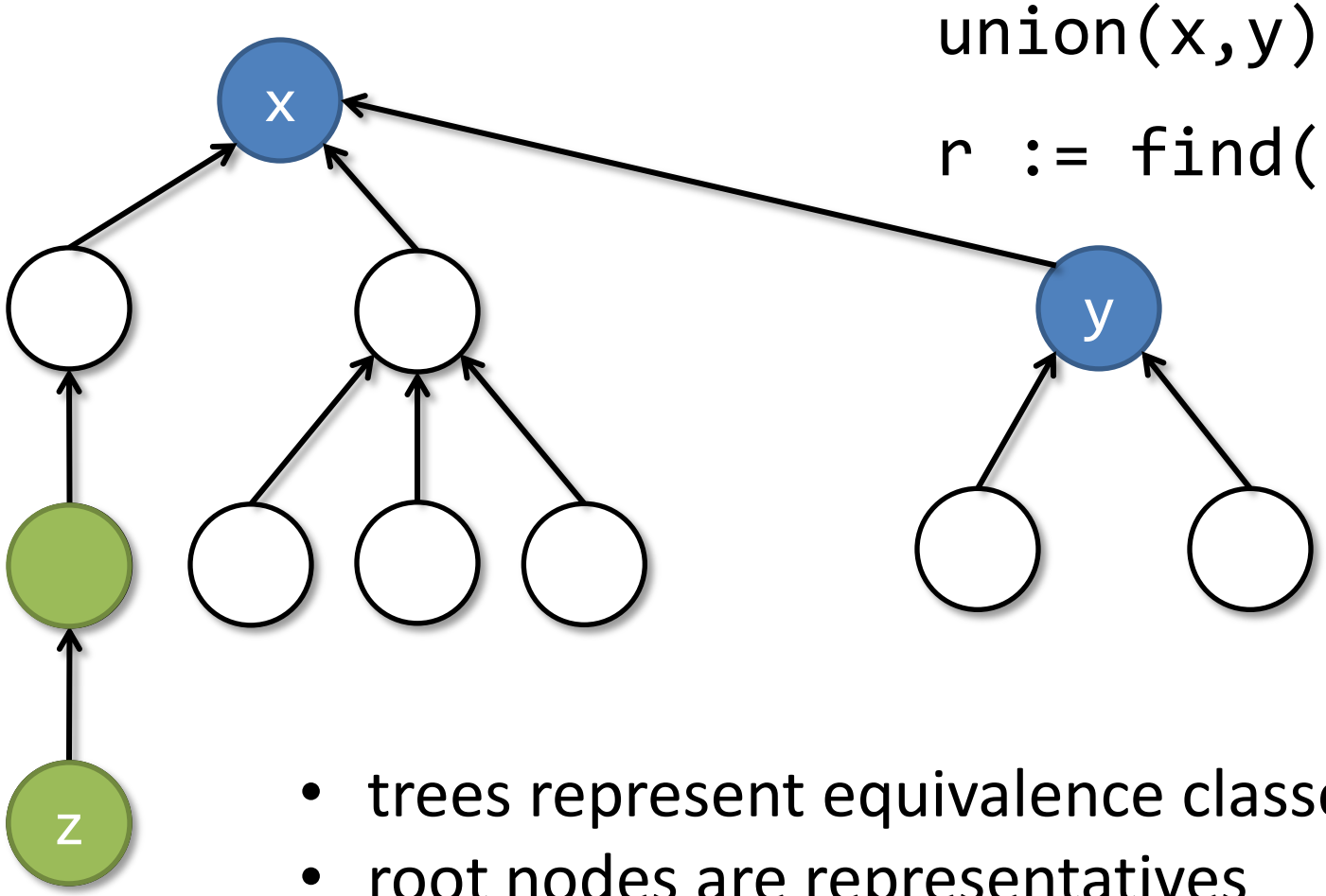
- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure



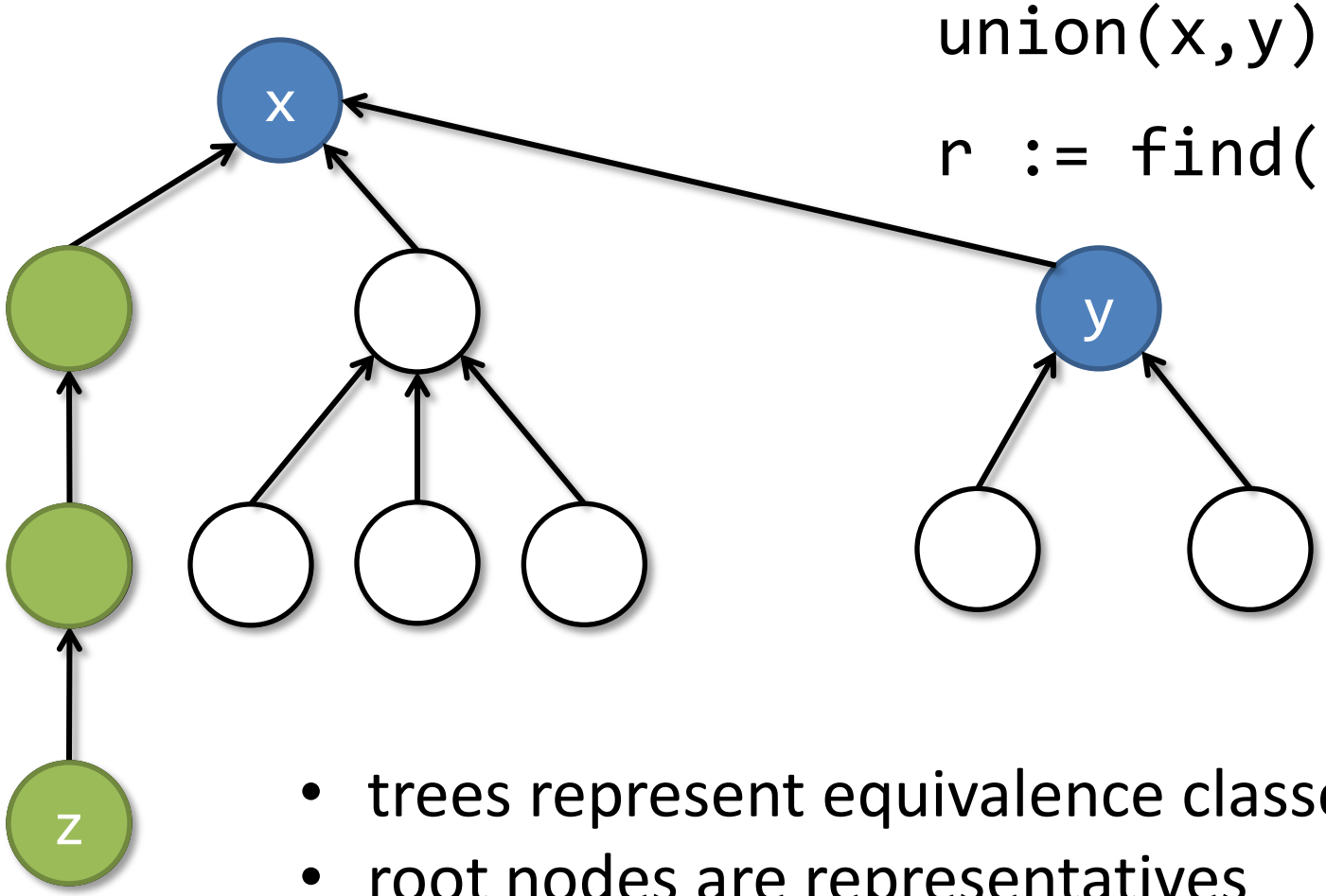
- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure



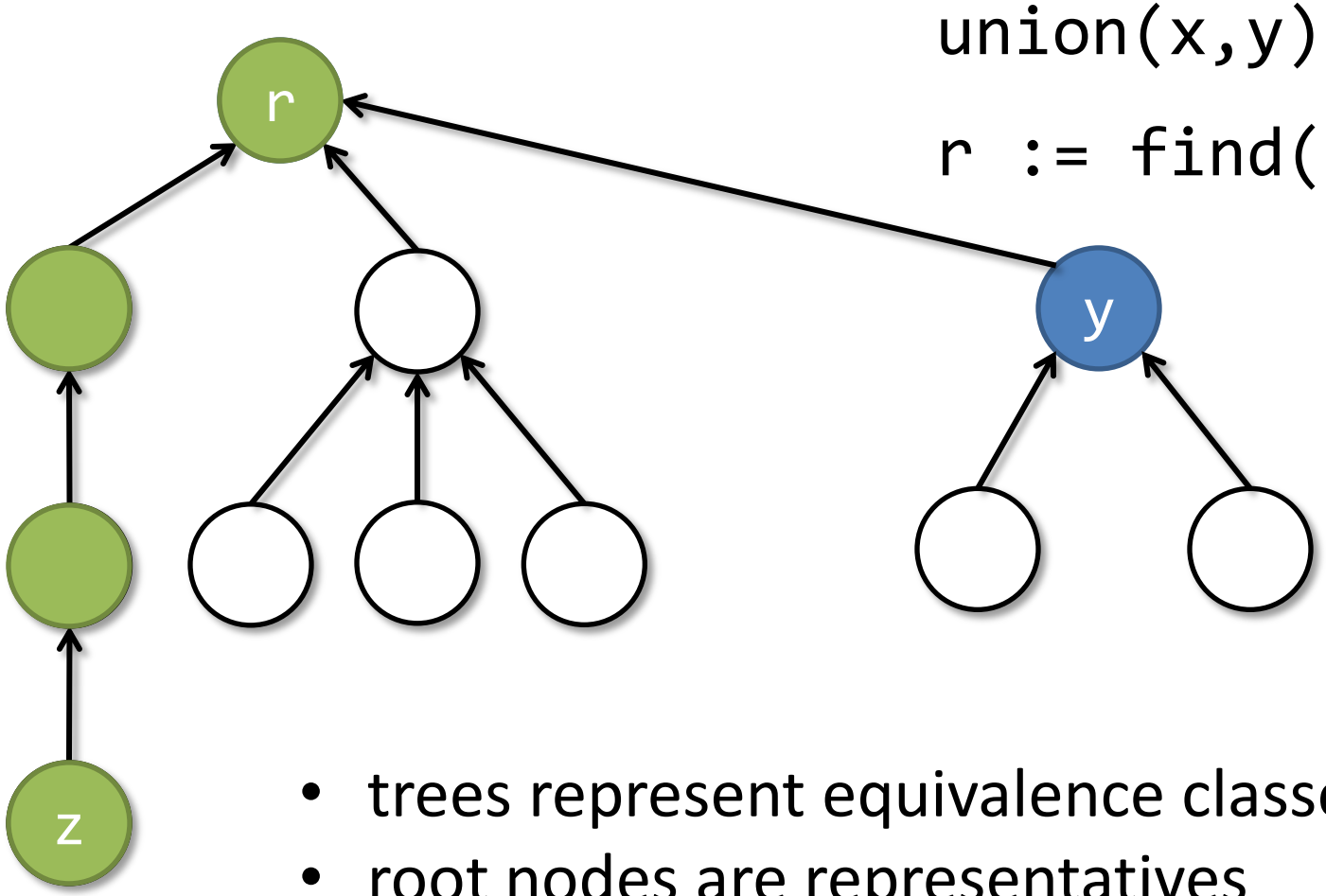
- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure



- trees represent equivalence classes
- root nodes are representatives

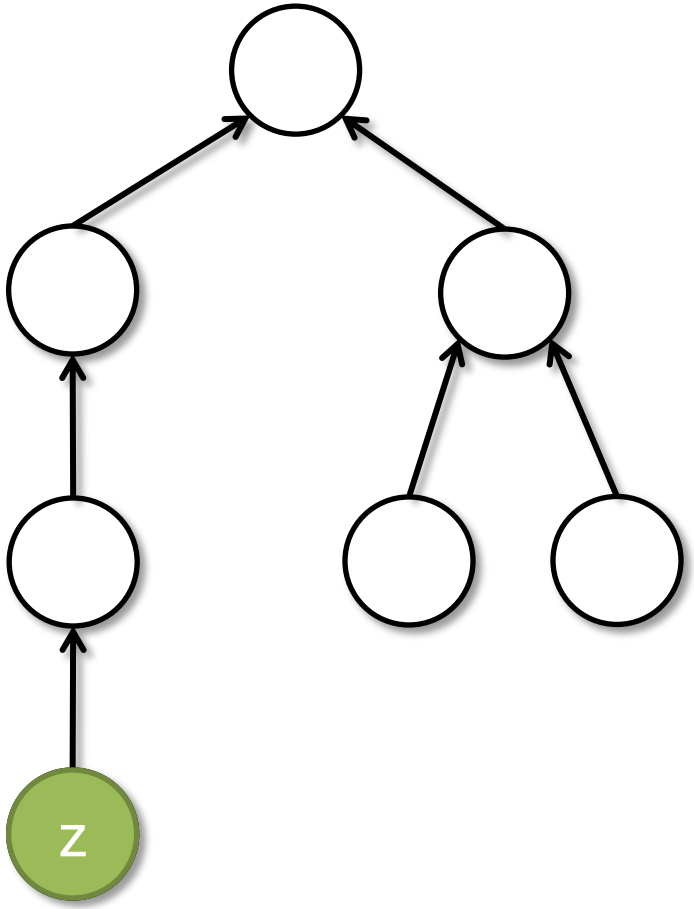
Example: Union/Find Data Structure



- trees represent equivalence classes
- root nodes are representatives

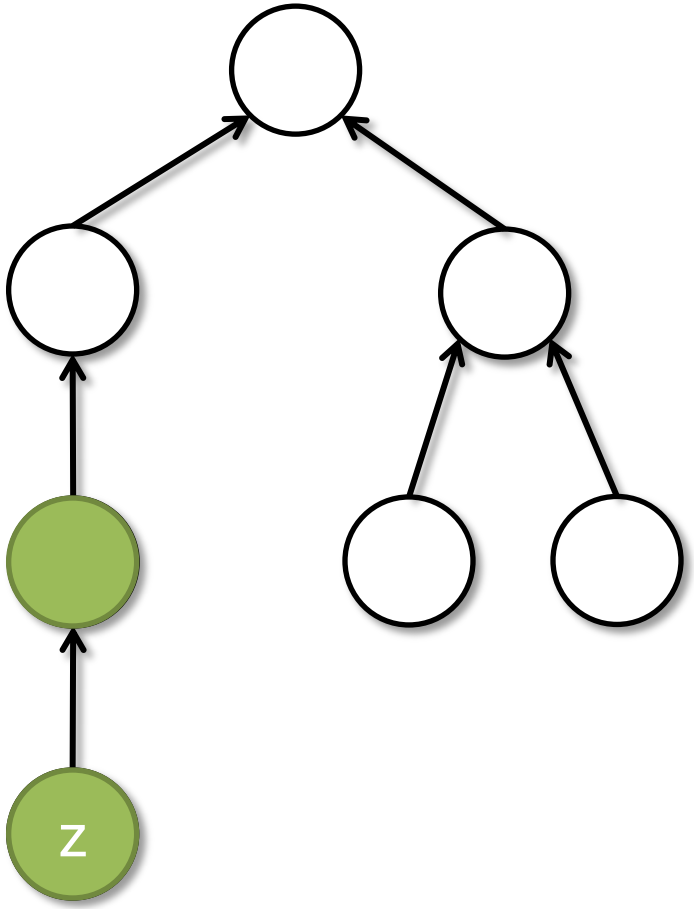
Find with Path Compression

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



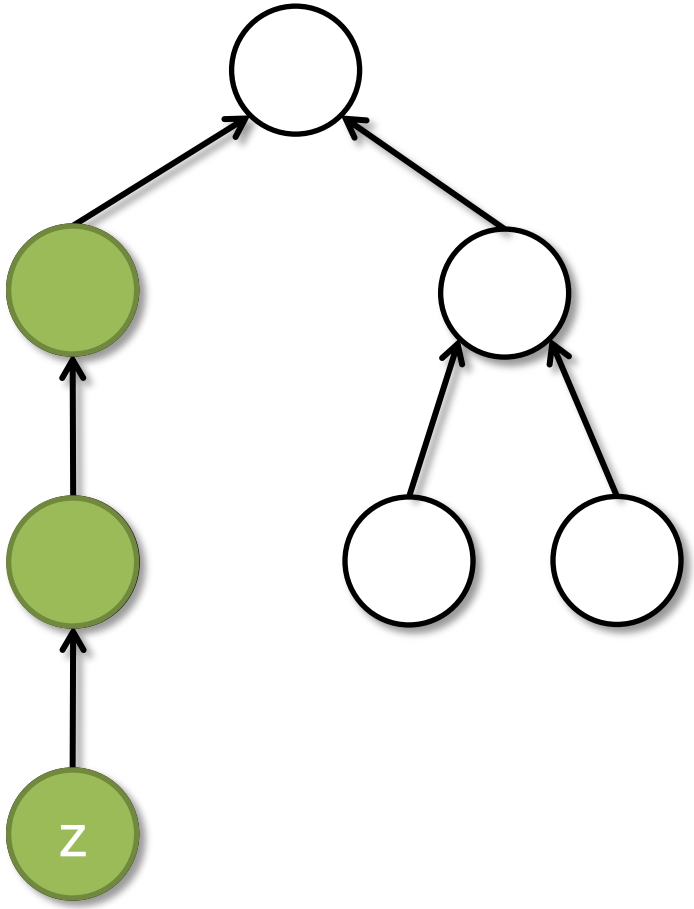
Find with Path Compression

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



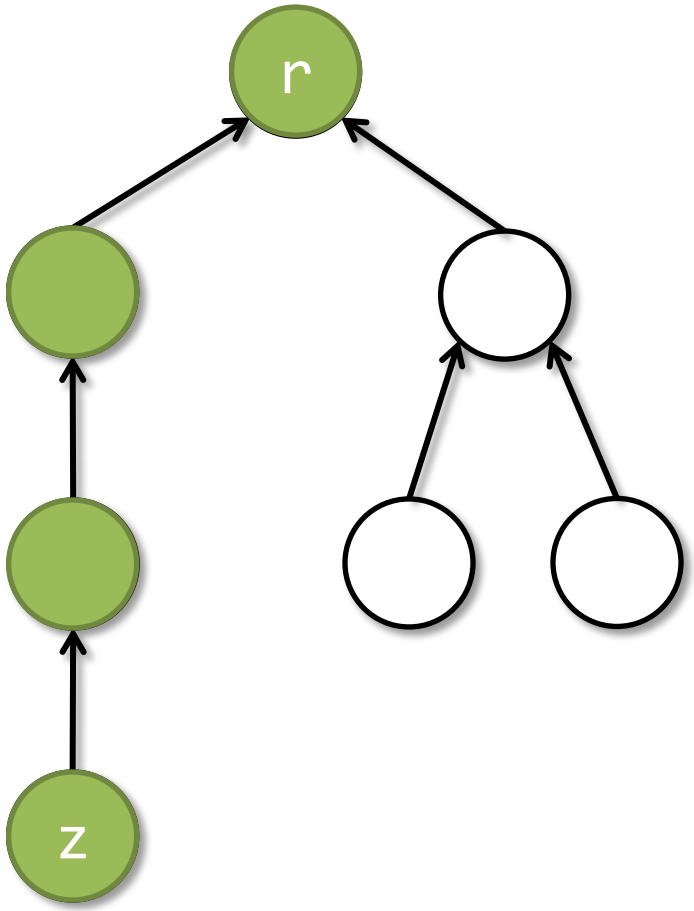
Find with Path Compression

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



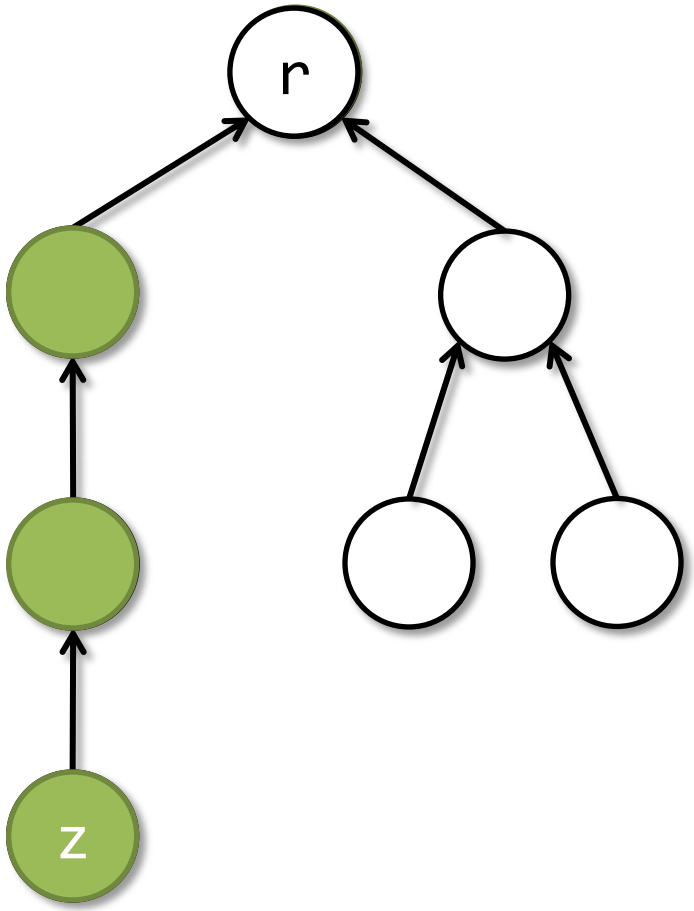
Find with Path Compression

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



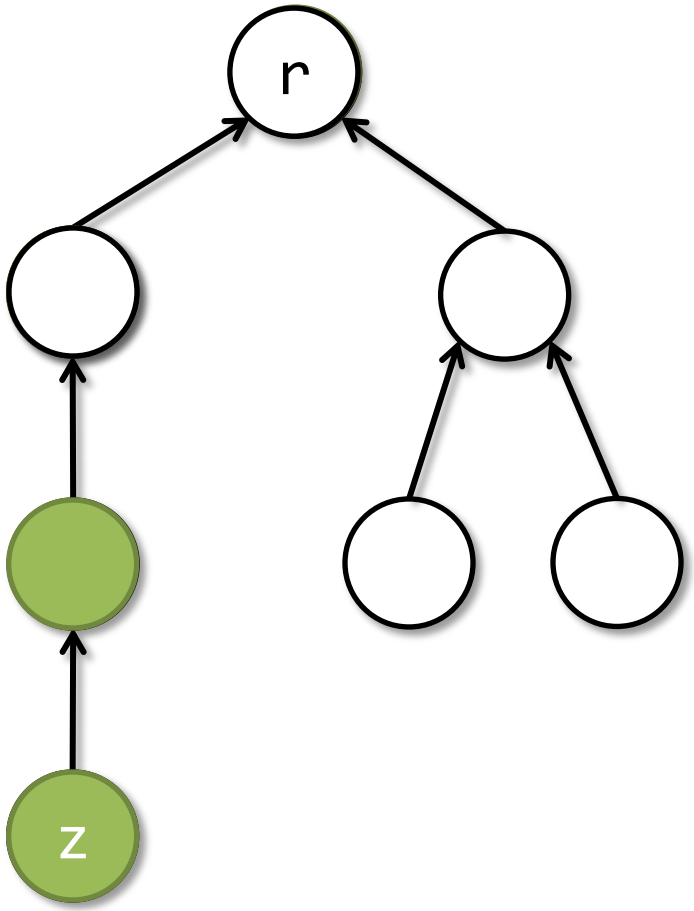
Find with Path Compression

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



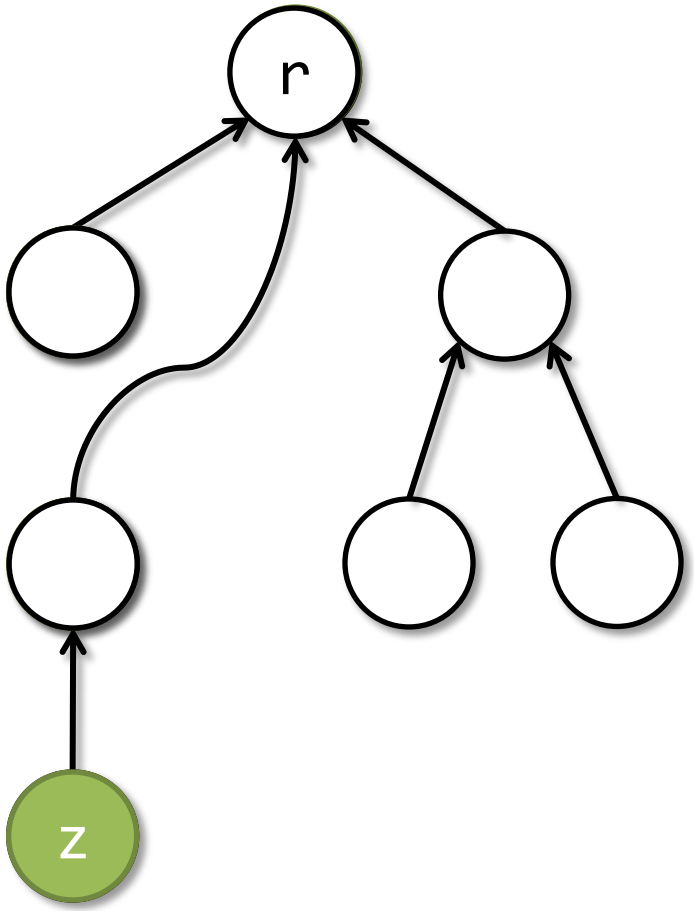
Find with Path Compression

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



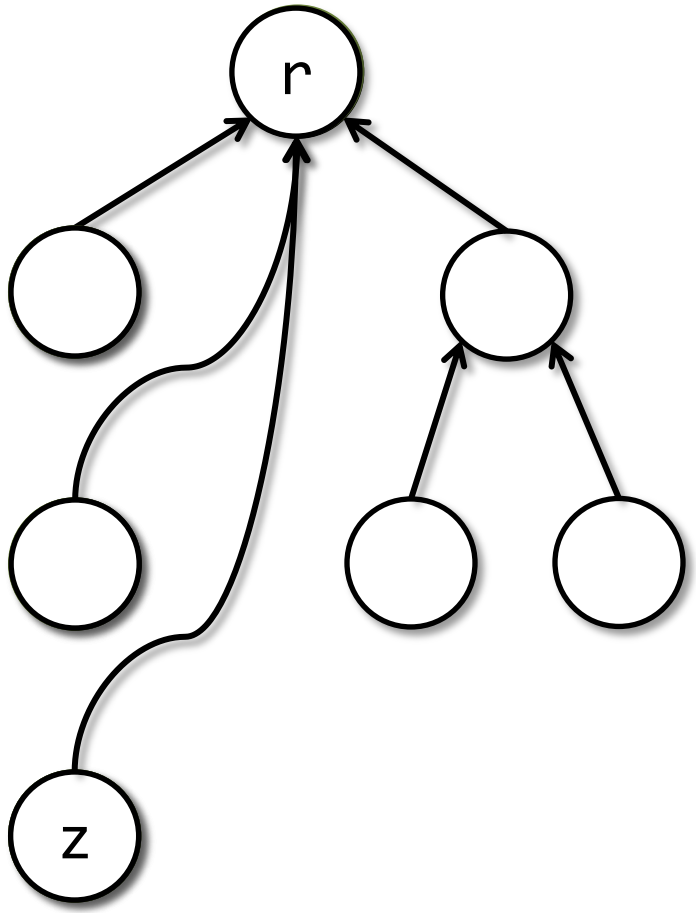
Find with Path Compression

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



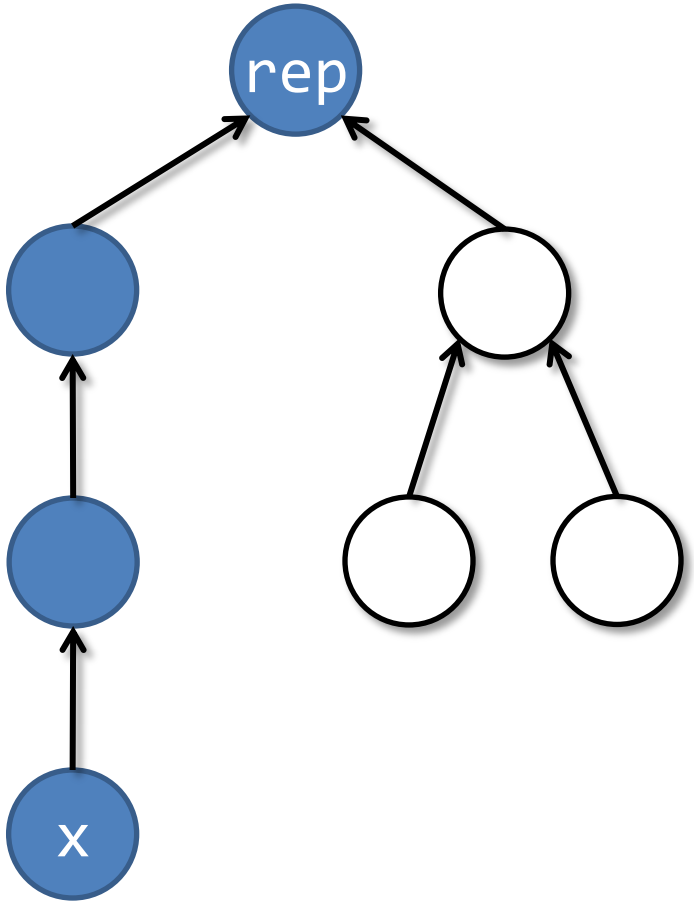
Find with Path Compression

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



Find with SL Specification

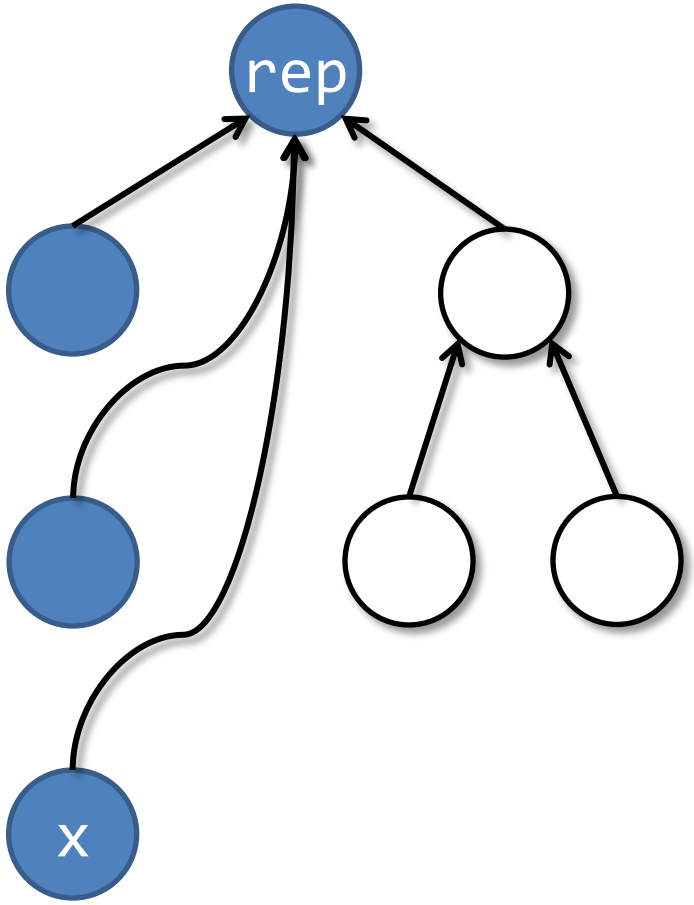
```
procedure find(x: Node, ghost rep: Node)
   returns (r: Node)
   requires lseg(x, rep)
   requires rep.next ↦ null
```



Find with SL Specification

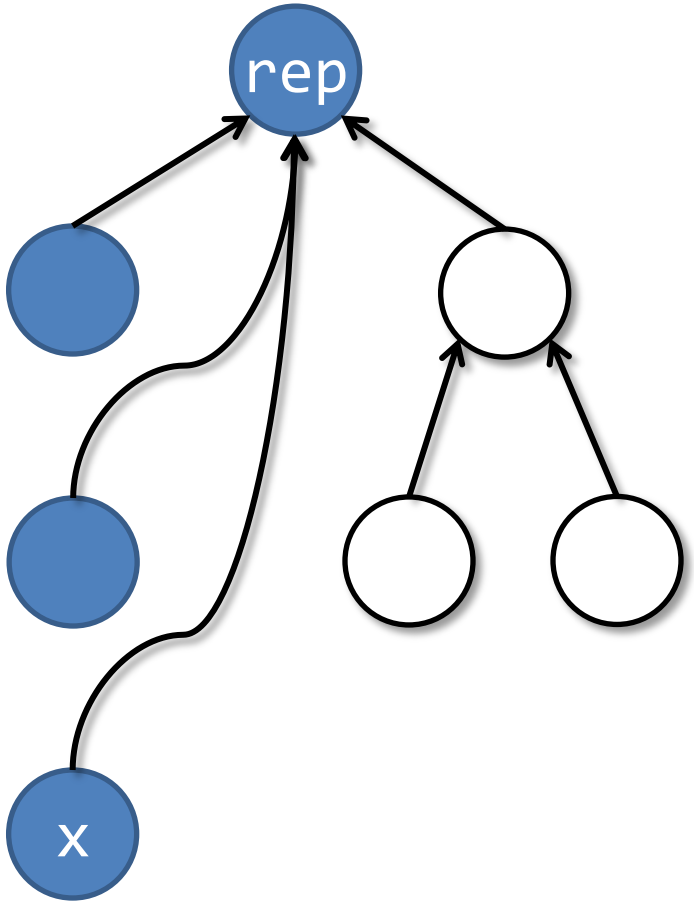
```
procedure find(x: Node, ghost rep: Node)
  returns (r: Node)
  requires rep.next ↦ null
  requires lseg(x, rep)
  ensures r = rep
  ensures rep.next ↦ null
  ensures ?
```

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_set(x, rep, X)
  ensures r = rep
  ensures rep.next ↦ null
  ensures acc(X)
  ensures ∀z ∈ X. z.next = rep
```



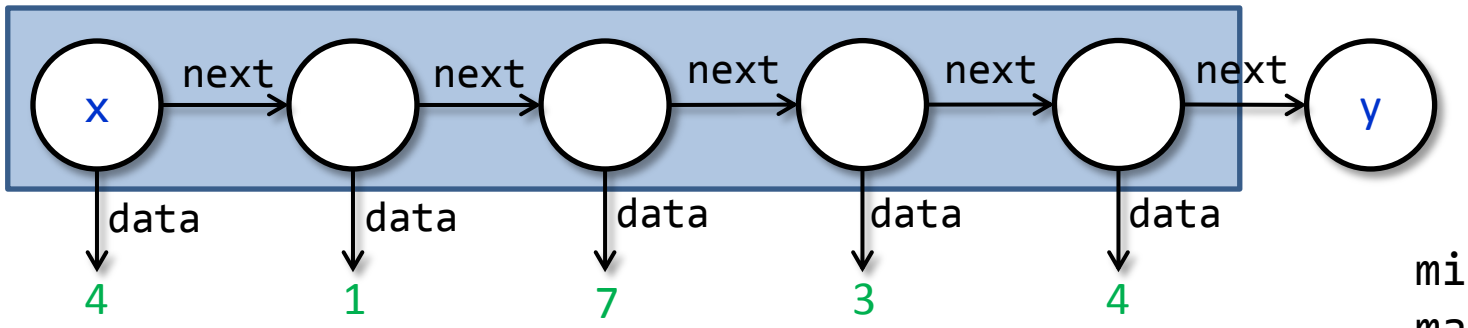
Completeness and Counterexamples

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;
}
```

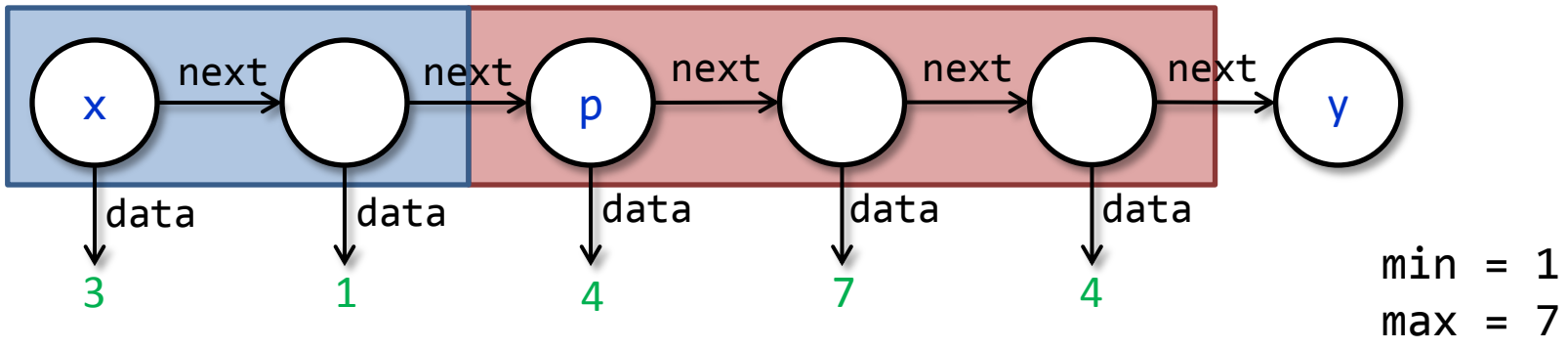
Split with SL Specification

```
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

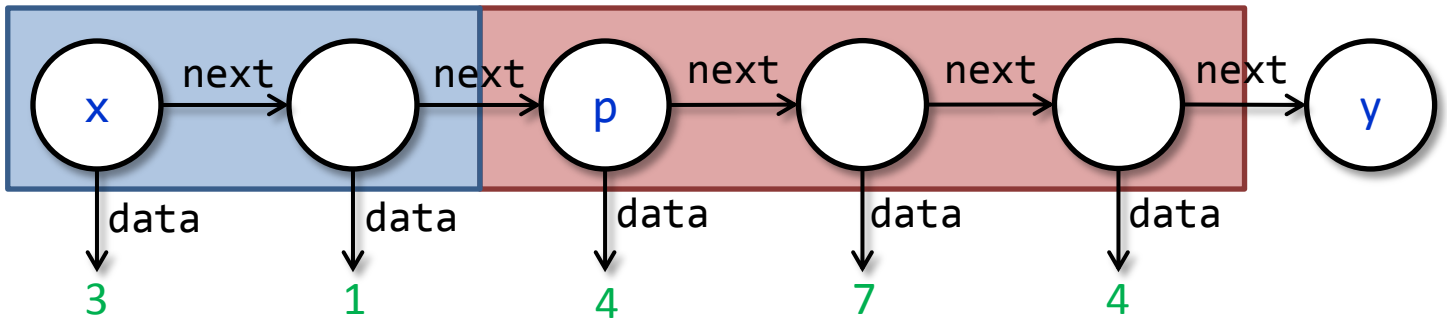
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```



Split with SL Specification

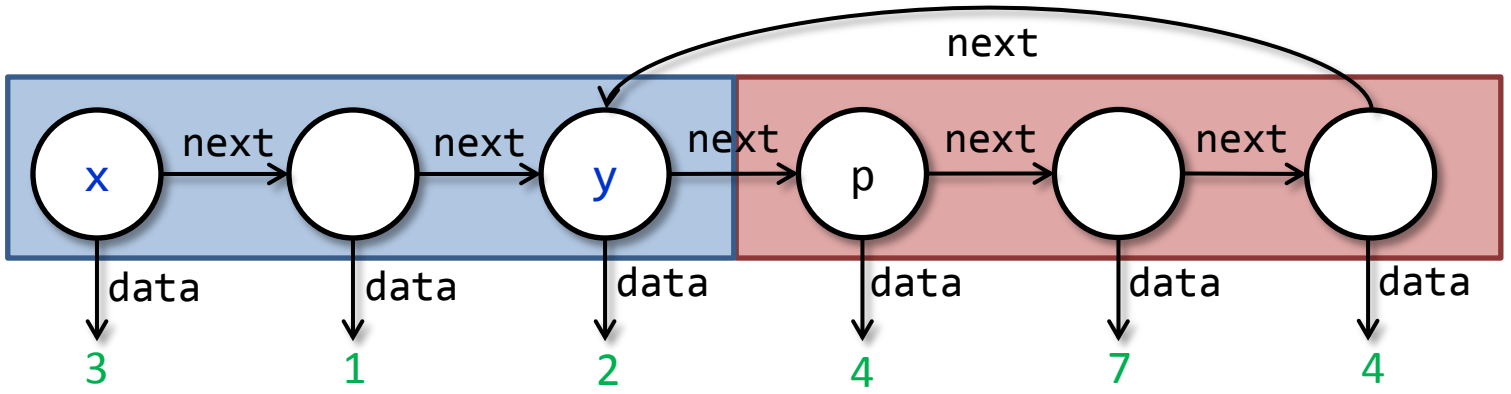
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```

free memory



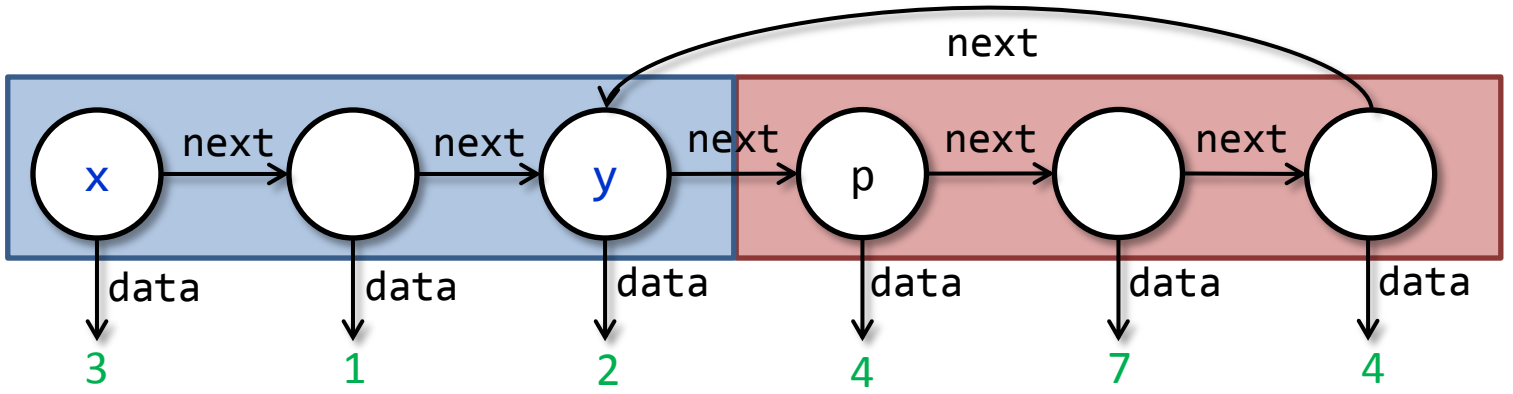
Counterexample for Quicksort Spec.

```
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
```



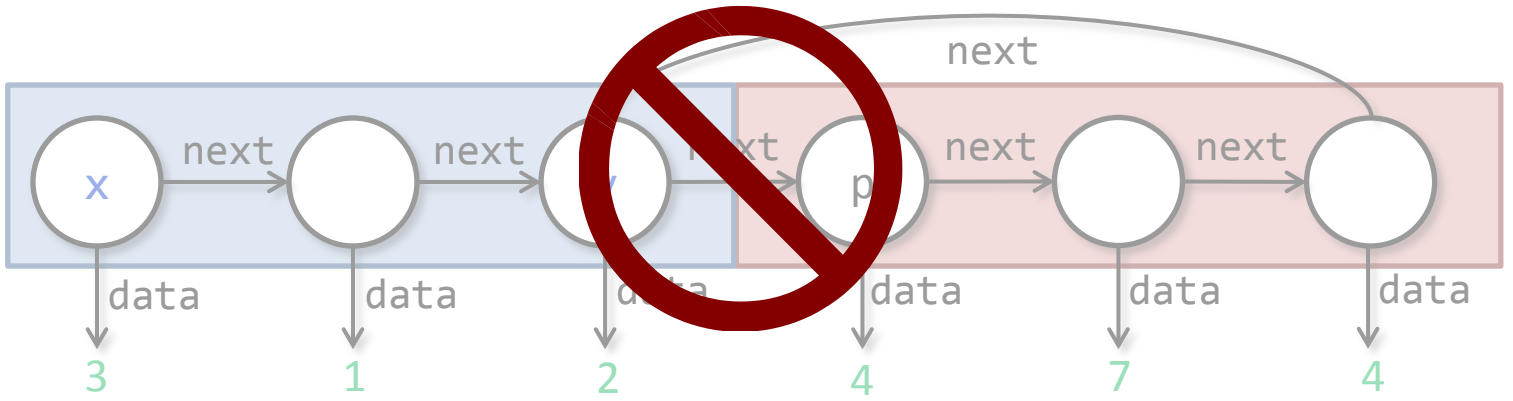
Split with Mixed Specification

```
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

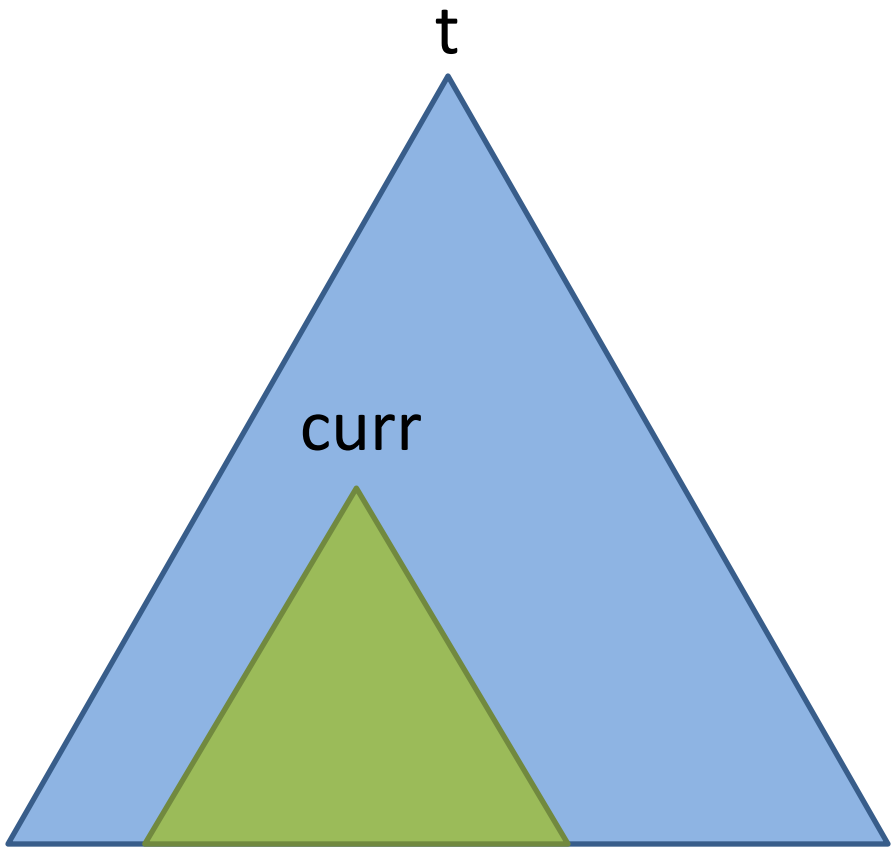
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max) * Btwn(next,x,p,y)
  ensures p ≠ y * min ≤ p.data ≤ max
```



Poor Man's Magic Wand

```
procedure contains(t: Tree, val: Int)
  returns (res: Bool)
  requires tree(t)
  ensures tree(t)
{
  var curr := root;
  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;
}
```

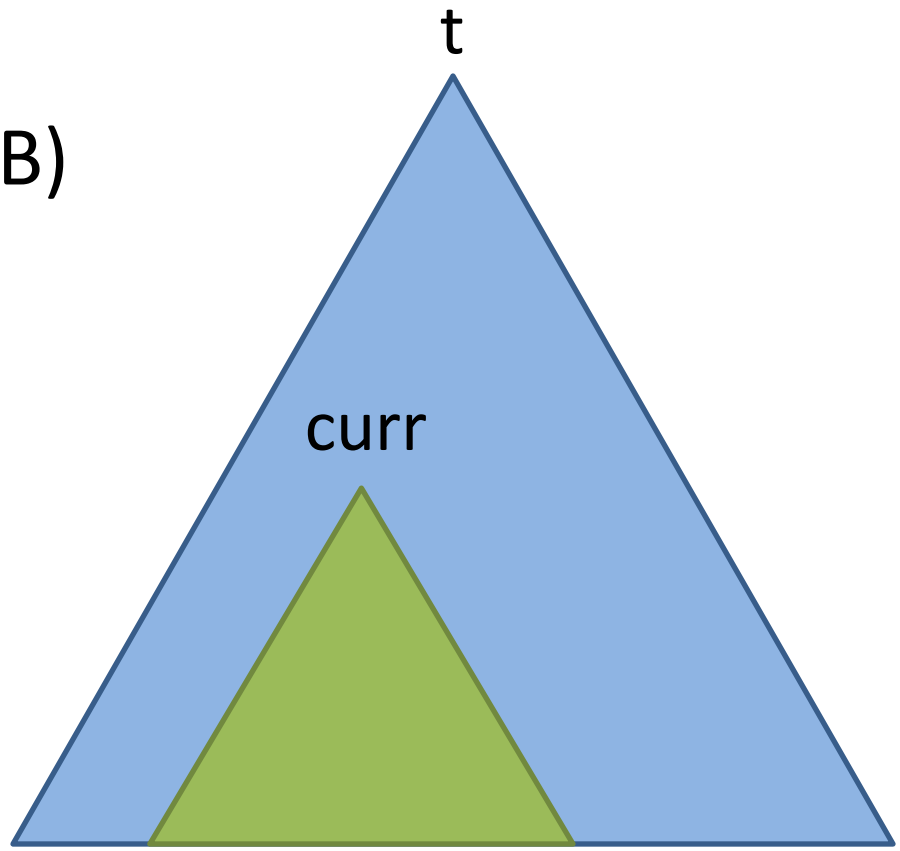
Poor Man's Magic Wand




```
tree(curr) * (tree(curr) -* tree(t))
```

Poor Man's Magic Wand

A ∗ (A -∗ B)

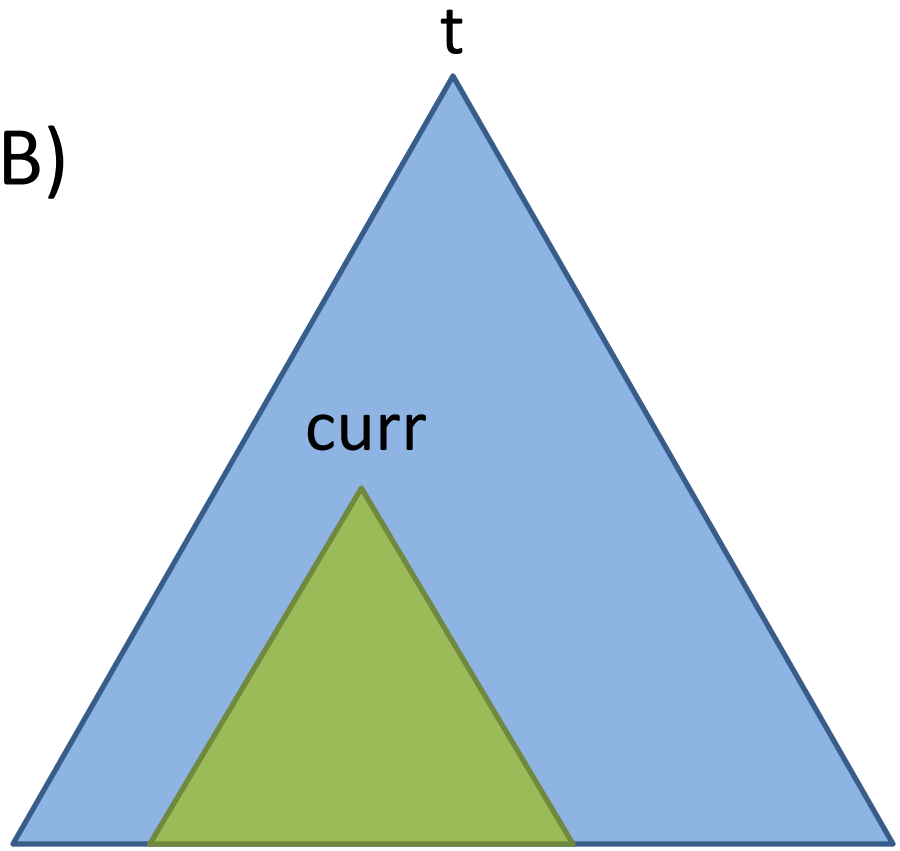



```
tree(curr) * (tree(curr) -* tree(t))
```

Poor Man's Magic Wand

A * (A -* B)

= A -** B



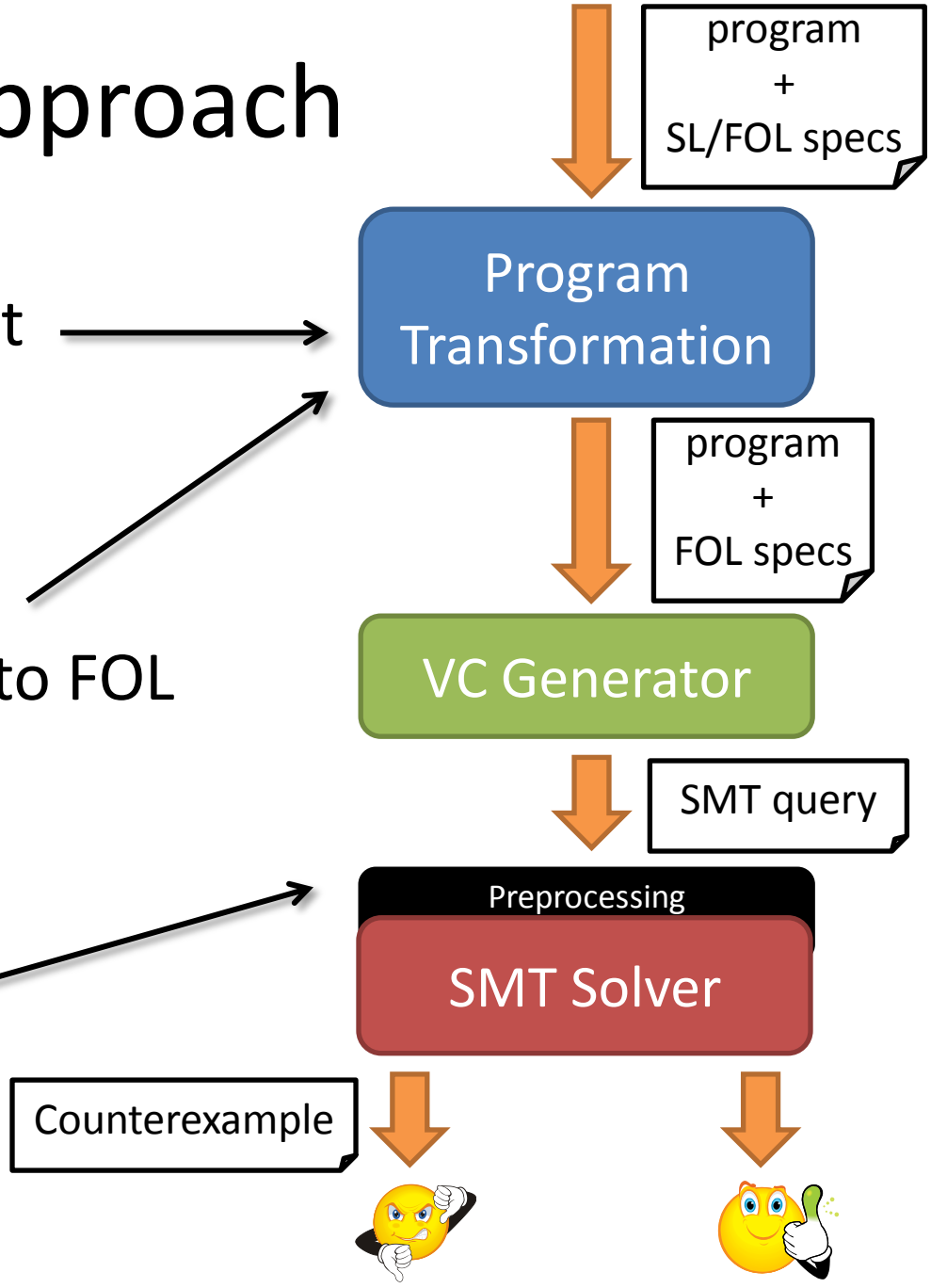

```
tree(curr) * (tree(curr) -* tree(t))
```

Poor Man's Magic Wand

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

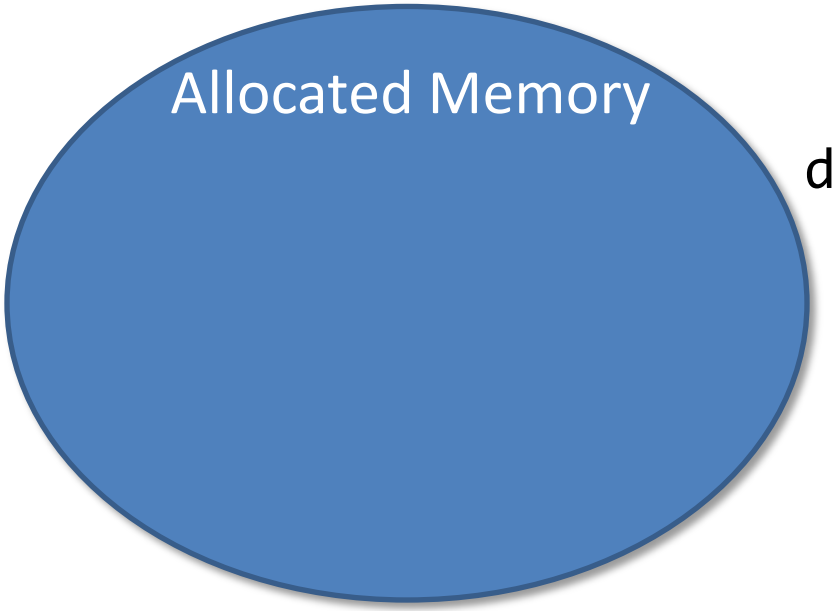
Overview of Approach

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

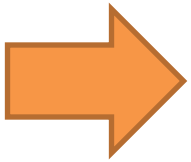


Step 1: Make Frame Rule Explicit

Encoding the Frame Rule



`delete(x);`

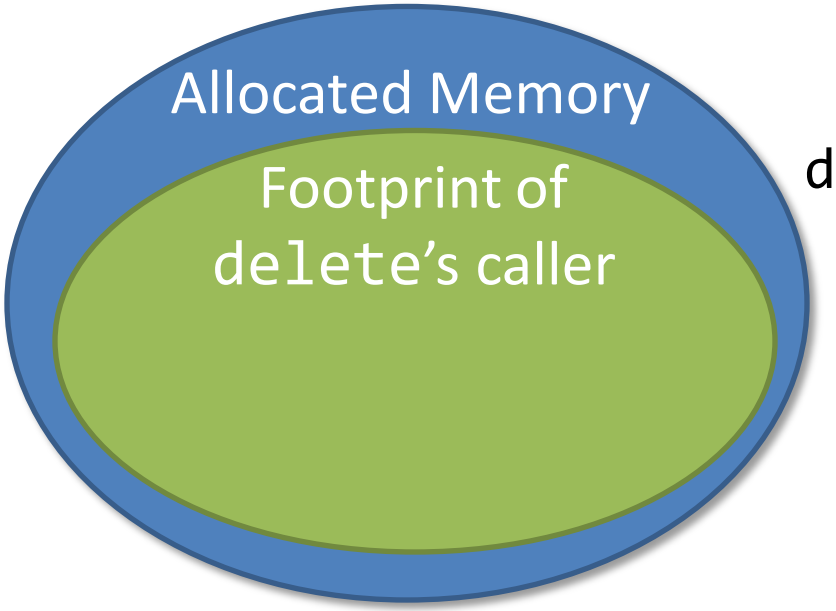


?

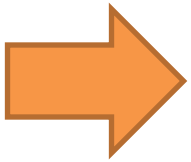
before call to `delete`

after call to `delete`

Encoding the Frame Rule



`delete(x);`

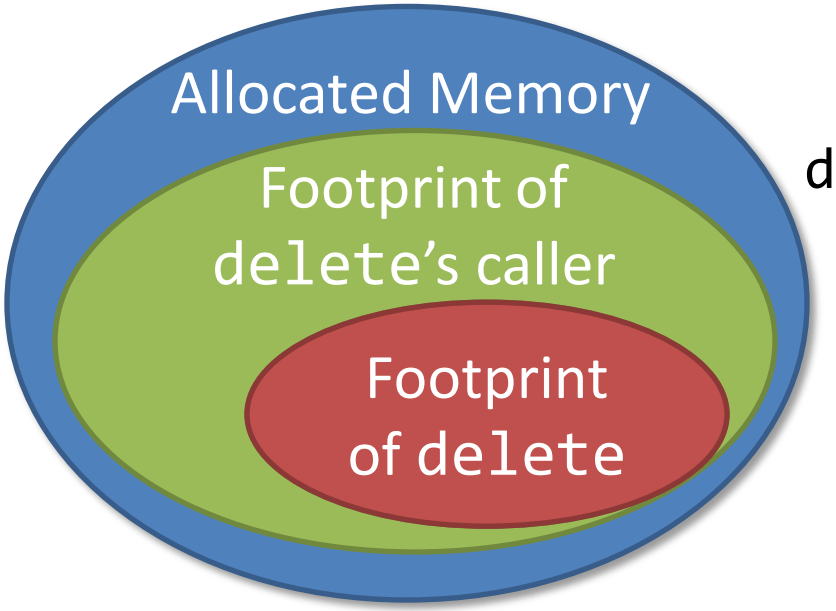


?

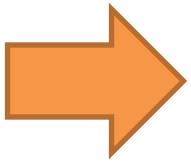
before call to `delete`

after call to `delete`

Encoding the Frame Rule




```
delete(x);
```

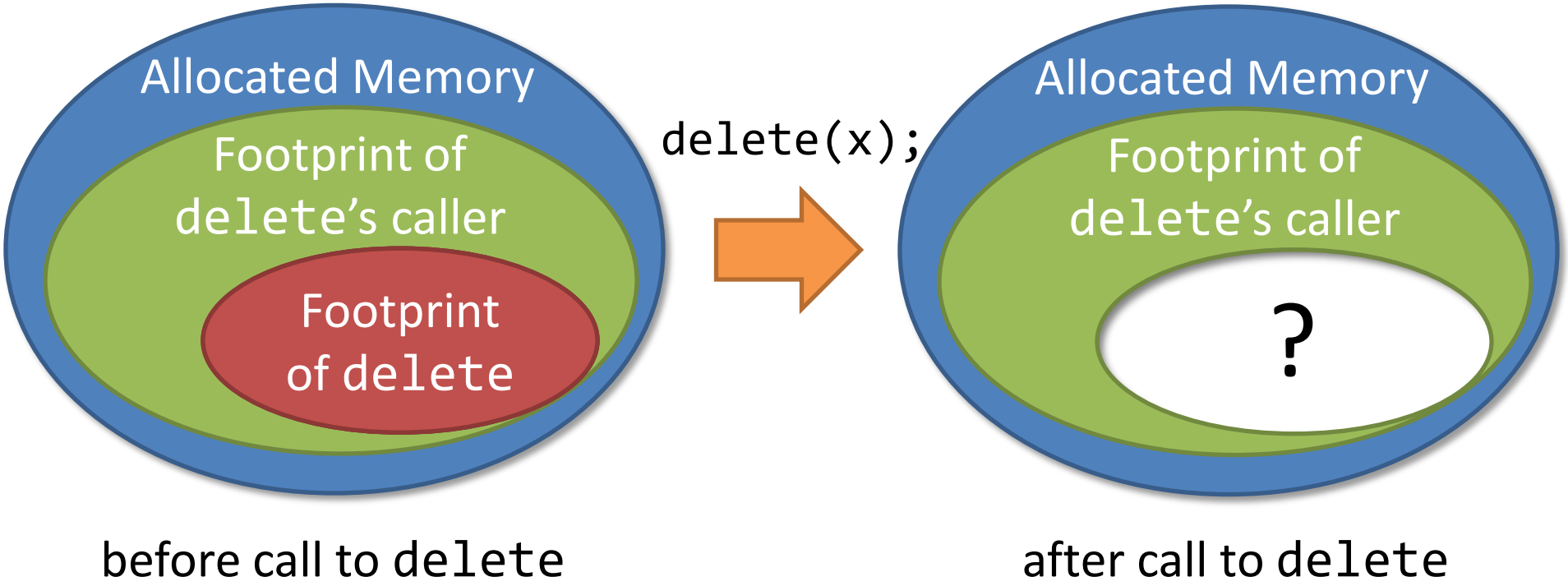


?

before call to `delete`

after call to `delete`

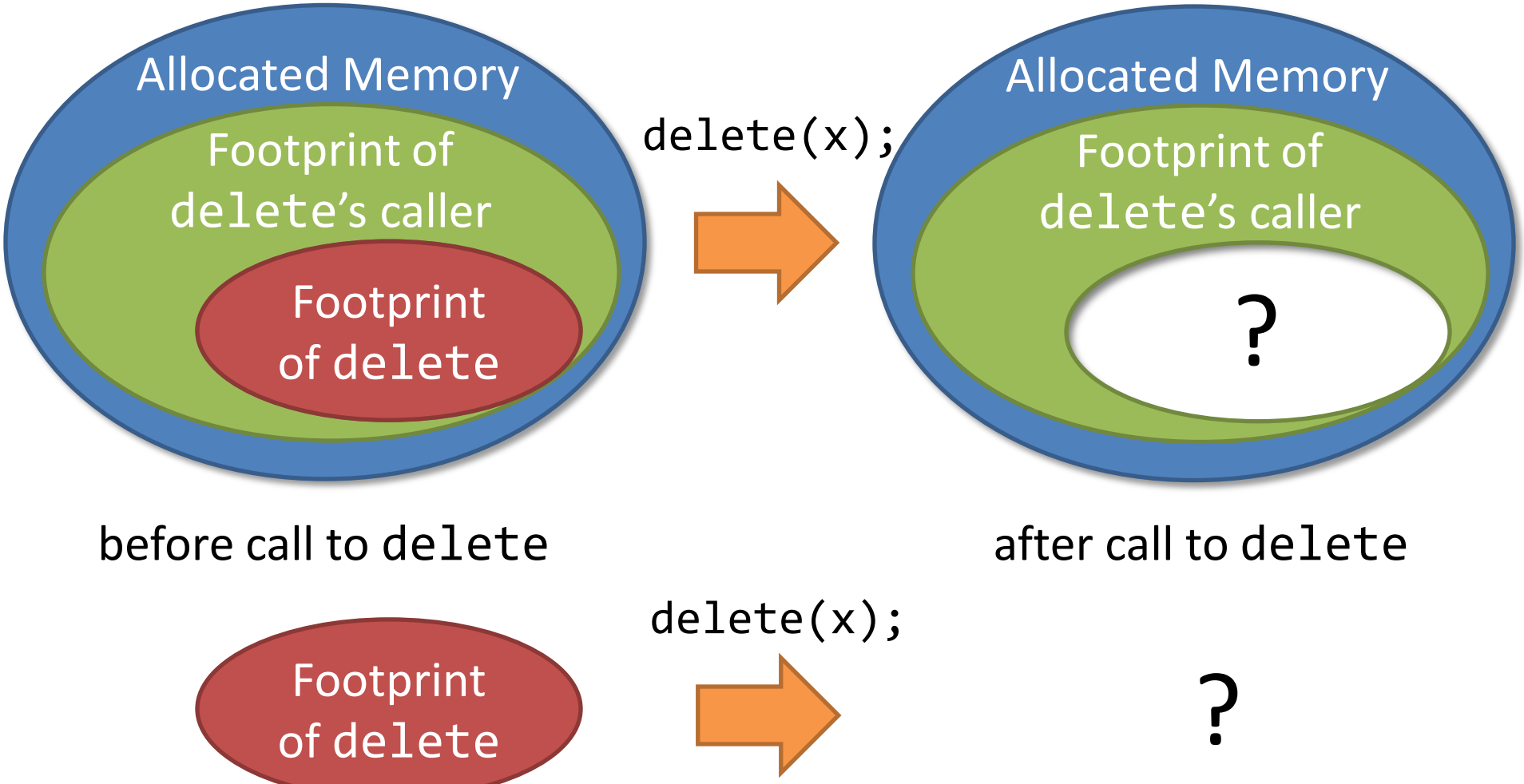
Encoding the Frame Rule



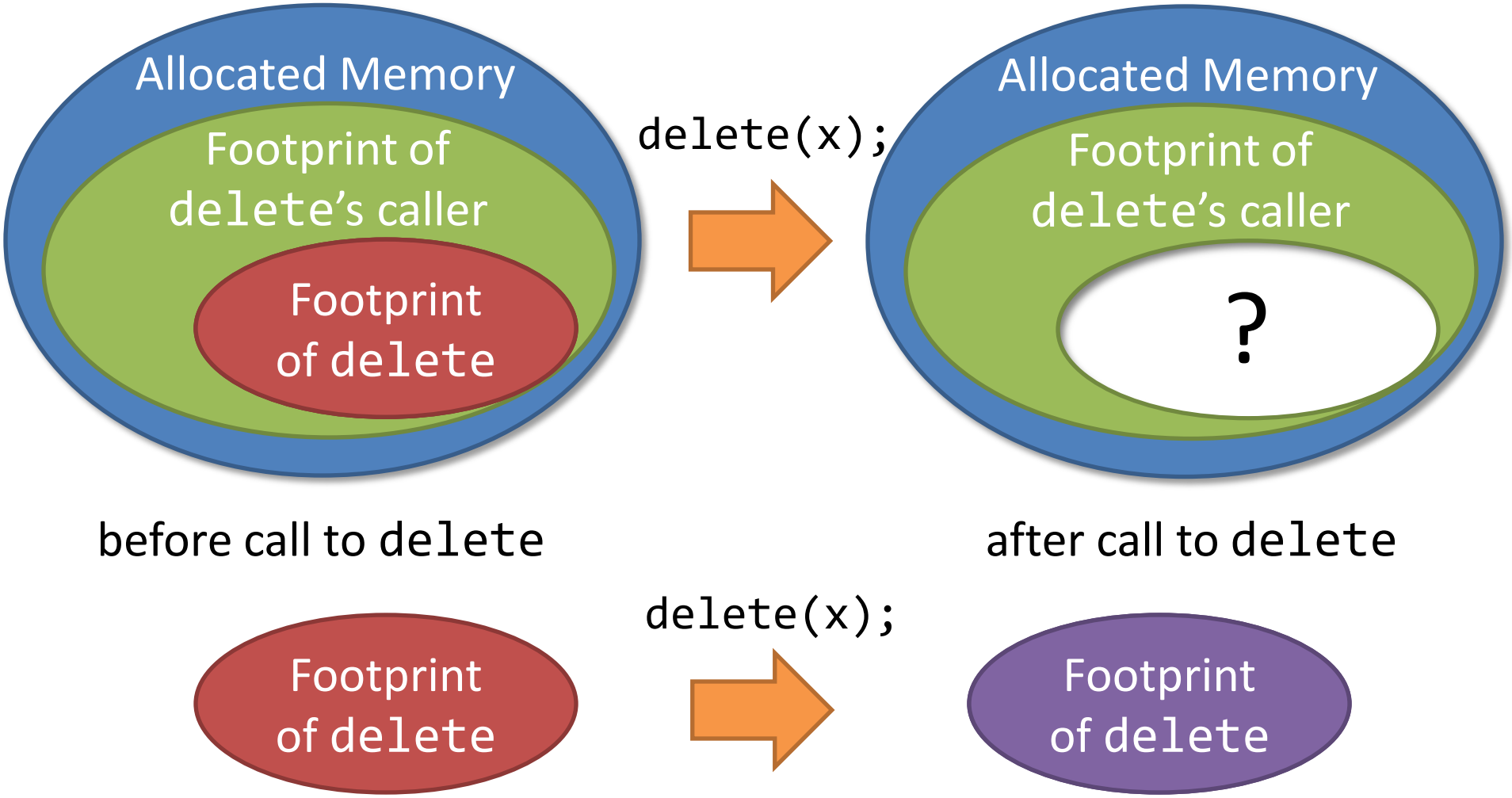
before call to `delete`

after call to `delete`

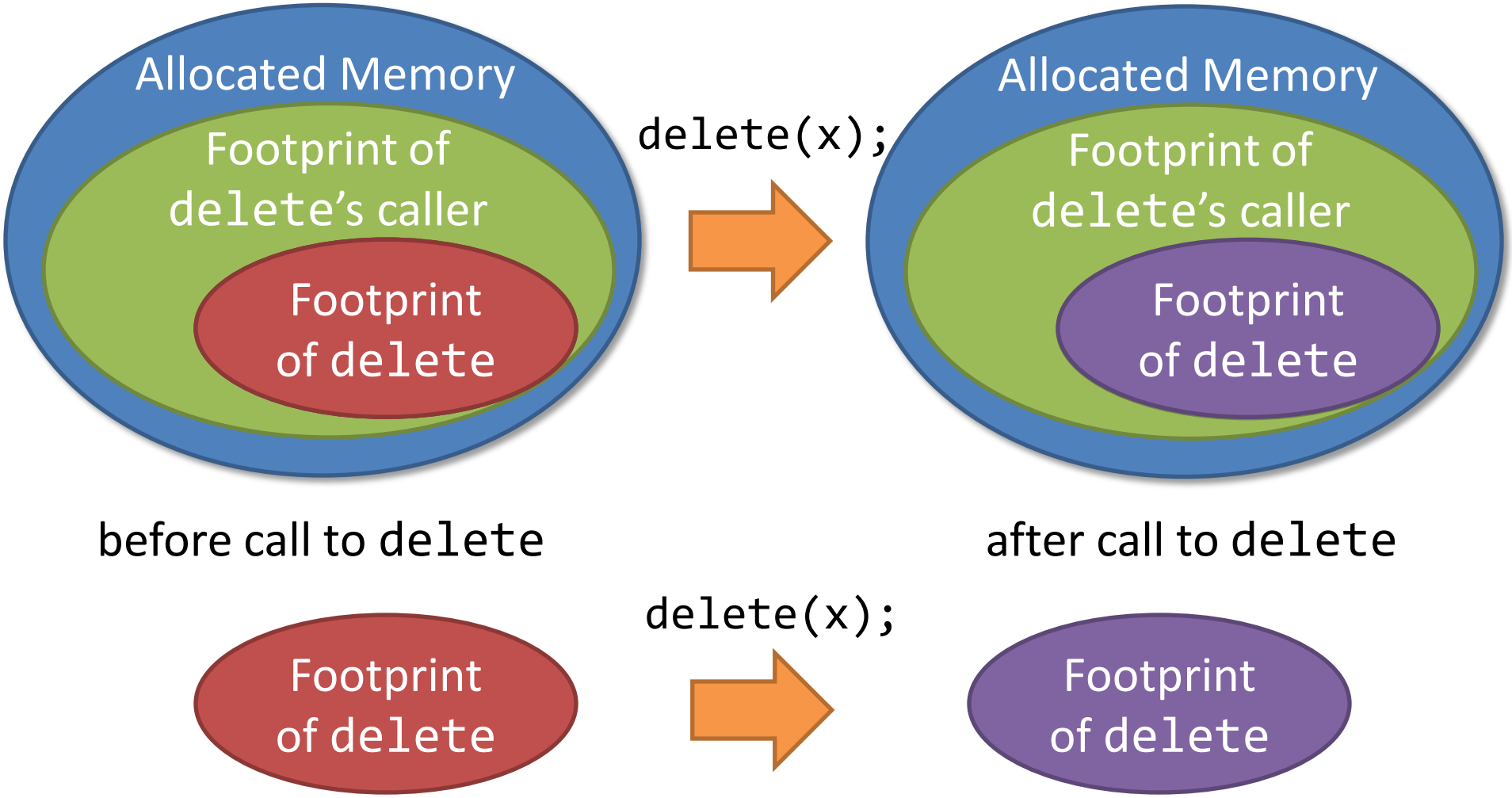
Encoding the Frame Rule



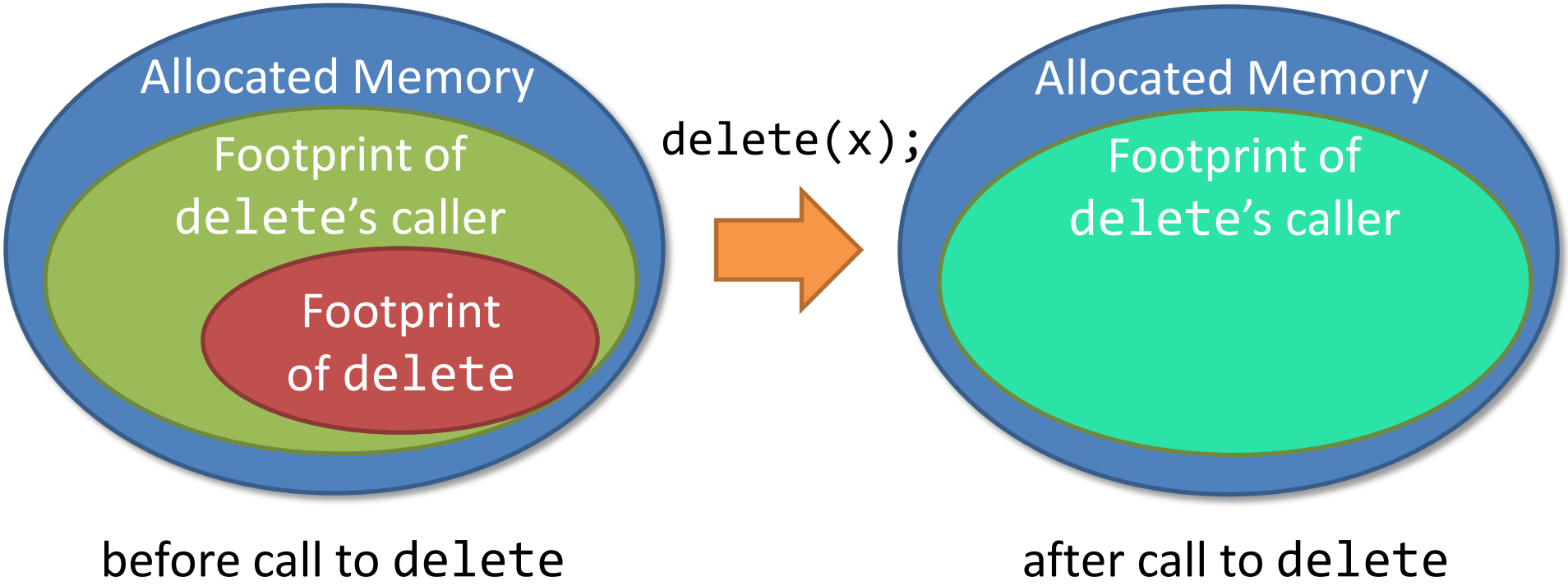
Encoding the Frame Rule



Encoding the Frame Rule



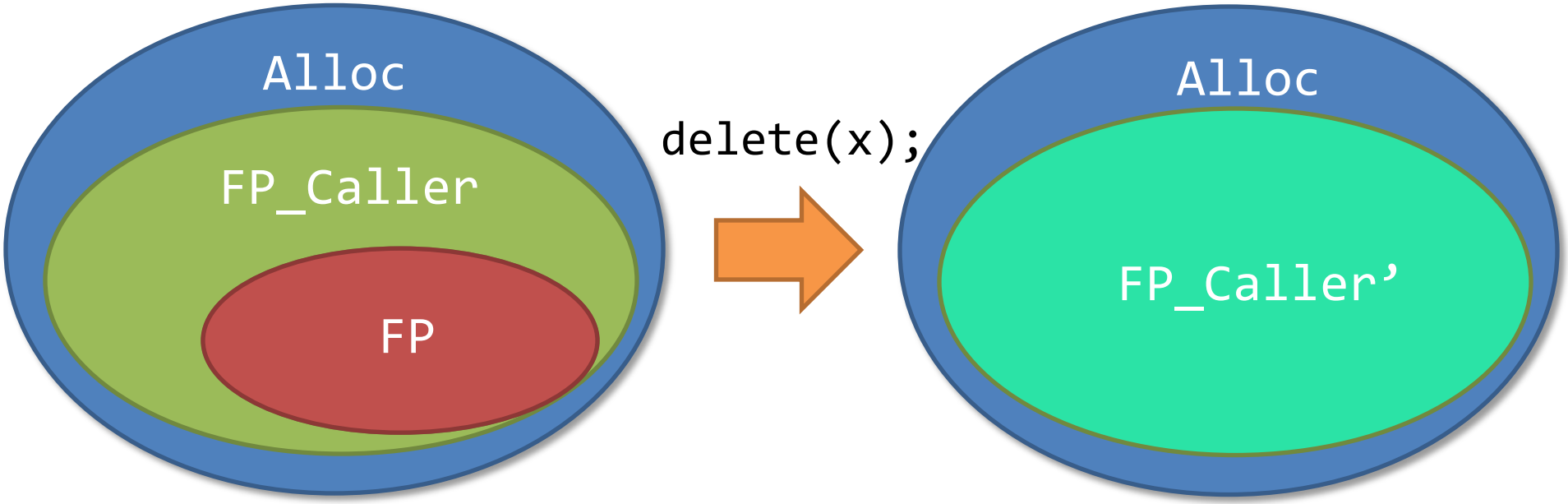
Encoding the Frame Rule



before call to `delete`

after call to `delete`

Encoding the Frame Rule

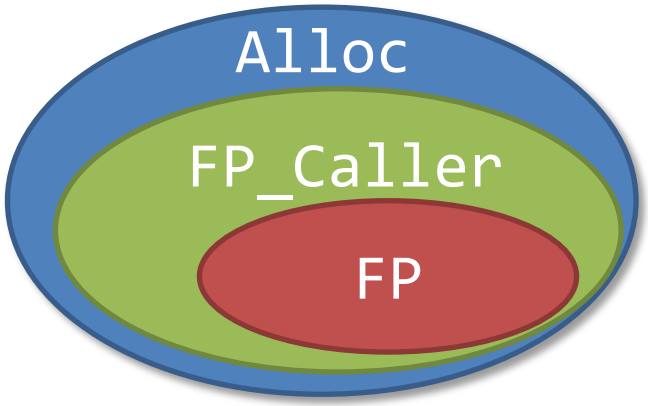


before call to `delete`

after call to `delete`

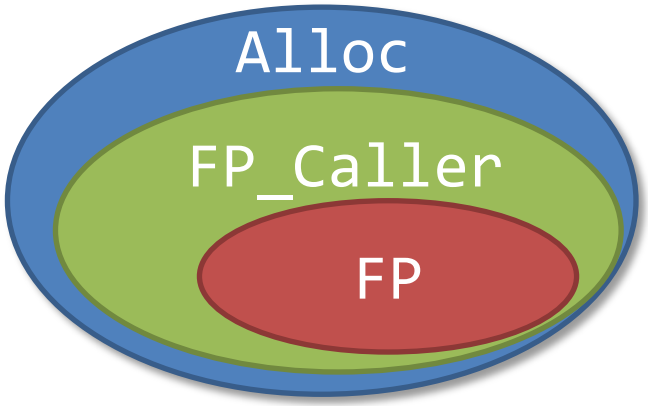
Encoding the Frame Rule

```
procedure delete(x: Node

                                        )

{

  if (x != null) {

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

}
```



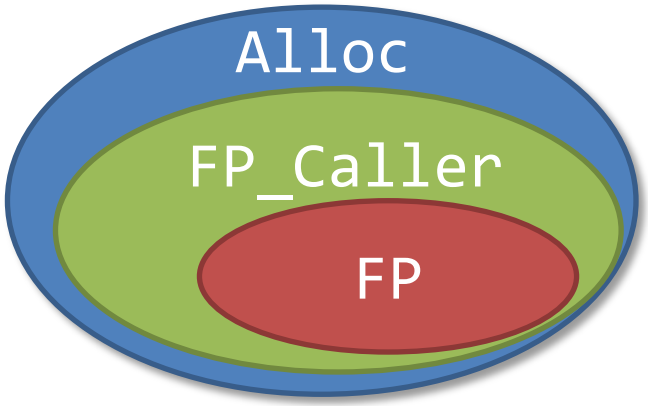
Encoding the Frame Rule

```
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);
  }

}
```



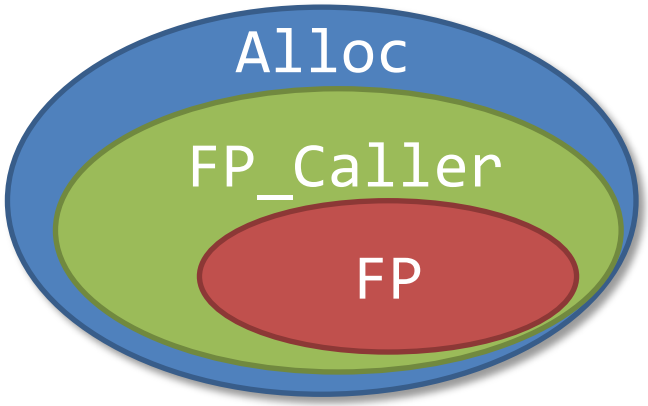
Encoding the Frame Rule

```
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) {
    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;
  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;
{ ... }
```

Encoding the Frame Rule

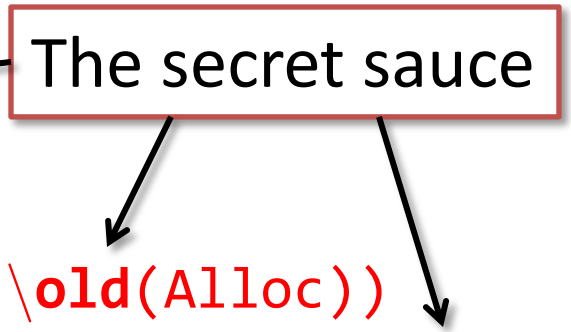
```
procedure delete(x: Node,
                 ghost FP_Caller: set<Node>,
                 implicit ghost FP: set<Node>)
returns (ghost FP_Caller': set<Node>)
  requires FP ⊆ FP_Caller;
  req
  fre
  fre
  ens
  fre                                                    t);
  fre
                 (Alloc ∩ 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

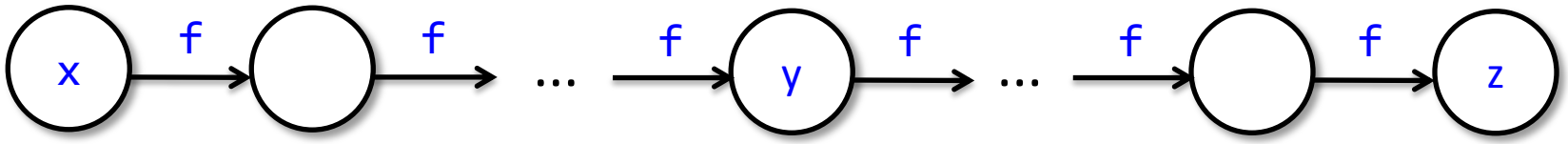
Target of Translation: GRASS (Graph Reachability and Stratified Sets)

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

Reachability in Mutable Function Graphs

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

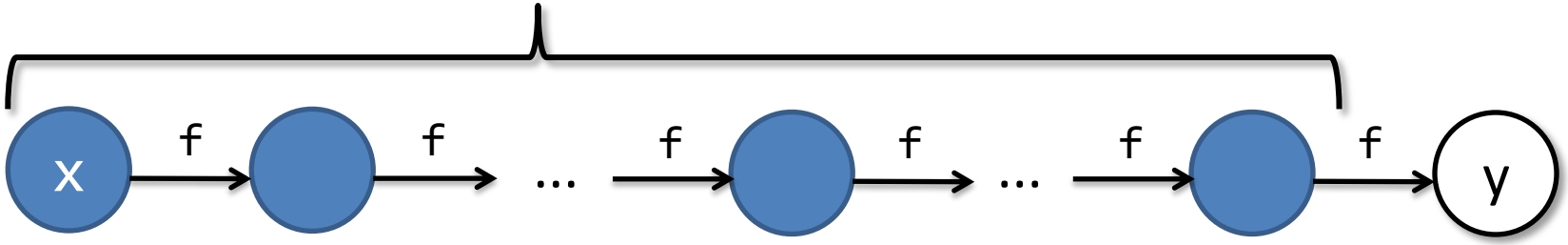
• sel(f, x)	field access	x.f
• upd(f,x,y)	field update	f[x := y]
• Btwn(f,x,y,z)	reachability	$x \xrightarrow{f} y \xrightarrow{f} z$



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

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

Translating SL Assertions to GRASS

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

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

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)

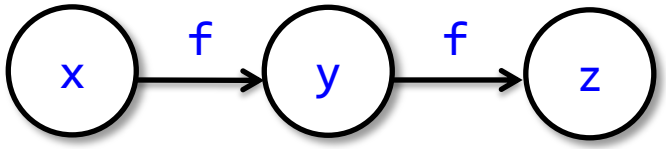
First-Order Axioms for Btwn

- $\forall$ 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)

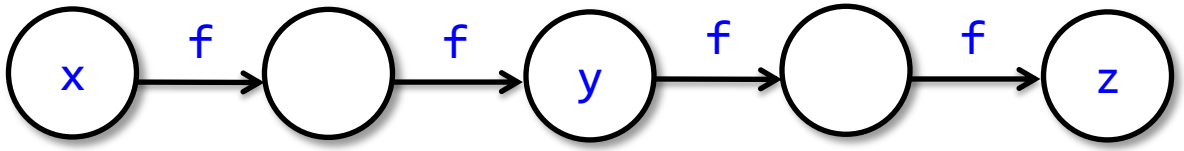
But I thought transitive closure was not first-order definable!?

Completeness of Axioms for Btwn

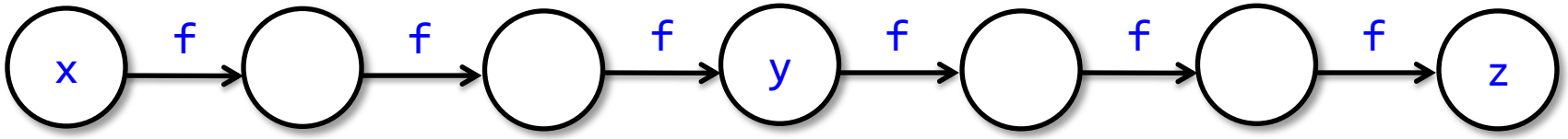
- A model of Btwn(f,x,y,z)



- and another model

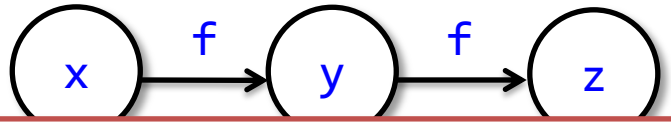


- and another



Completeness of Axioms for Btwn

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

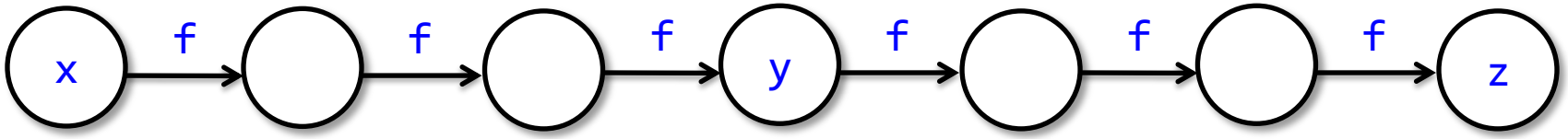


There are arbitrarily large finite models

+

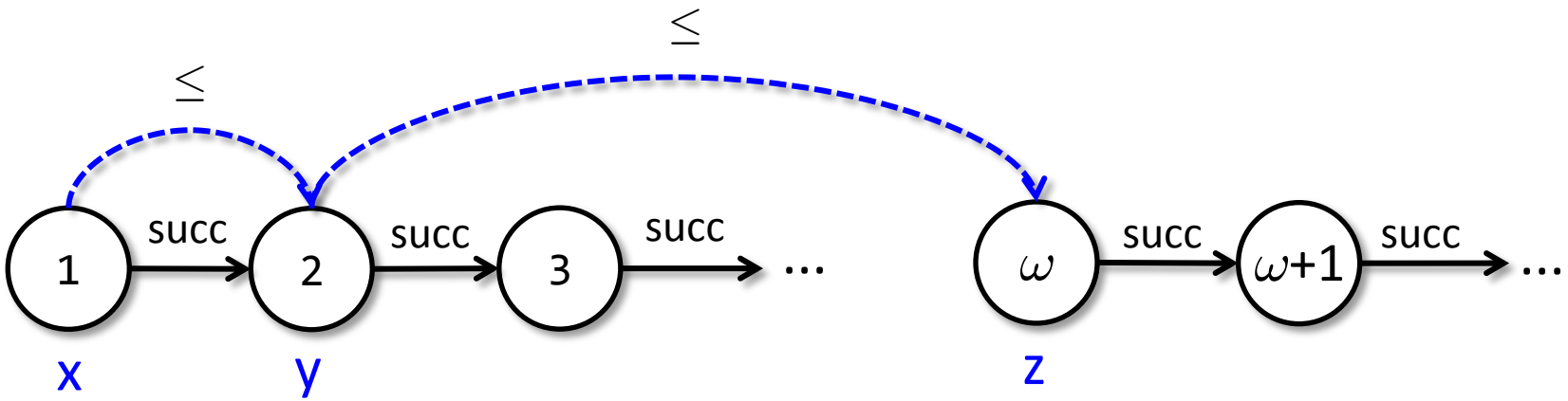
Compactness Theorem $\Rightarrow$ there must also be infinite models

- and another



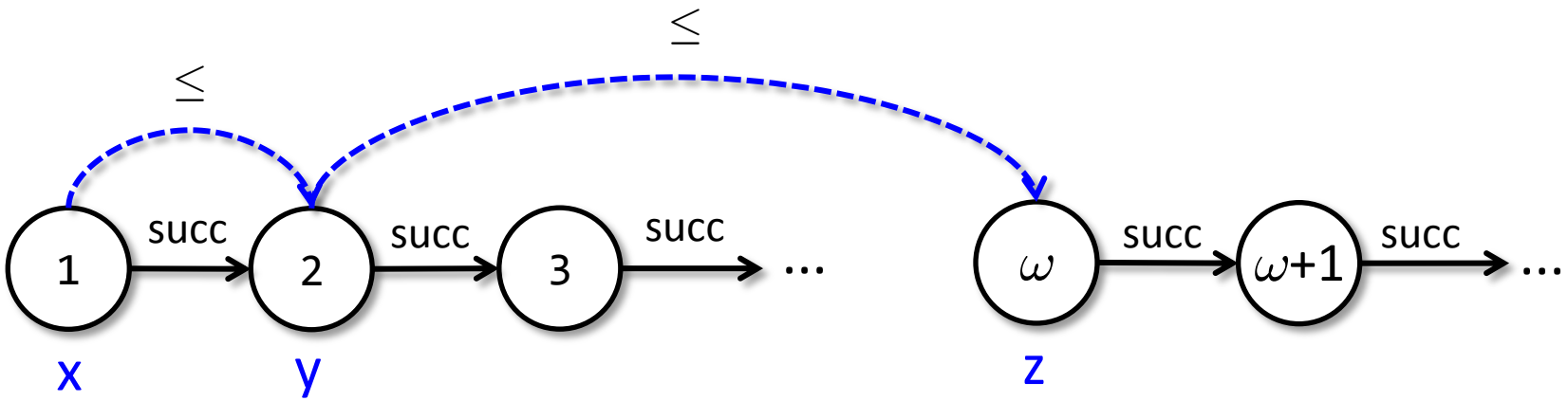
A *Degenerated* Infinite Model M of Btwn(f,x,y,z)

- M = ordinal numbers
- M(f) = succ
- M(Btwn) = $\lambda uvw.\ u \leq v \wedge v \leq w$



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

- M = ordinal numbers
- M(f) = succ
- M(Btwn) = $\lambda uvw.\ u \leq v \wedge v \leq w$



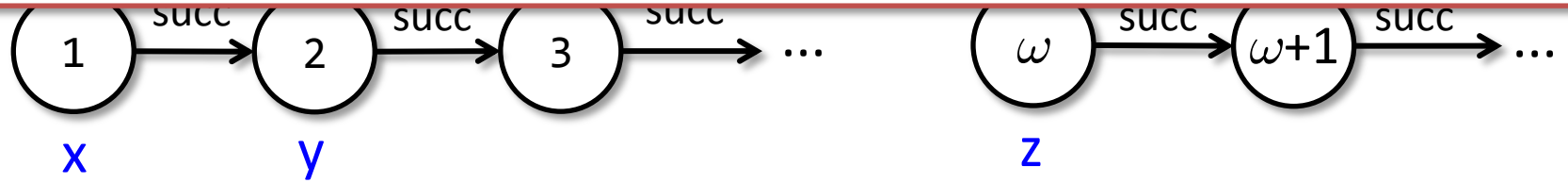
$\leq$ is not succ*

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)

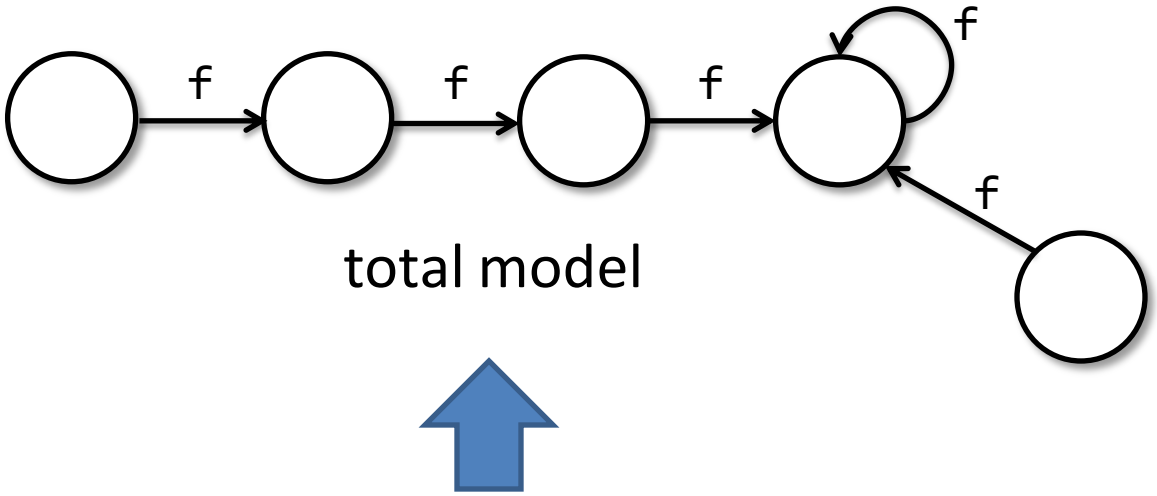
First-Order Axioms for Btwn

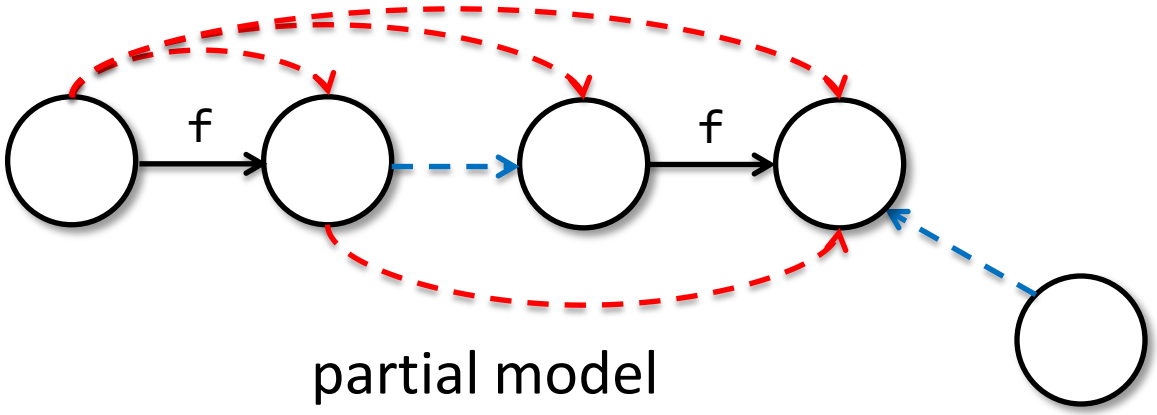
Almost in EPR!

- $\forall$ 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)

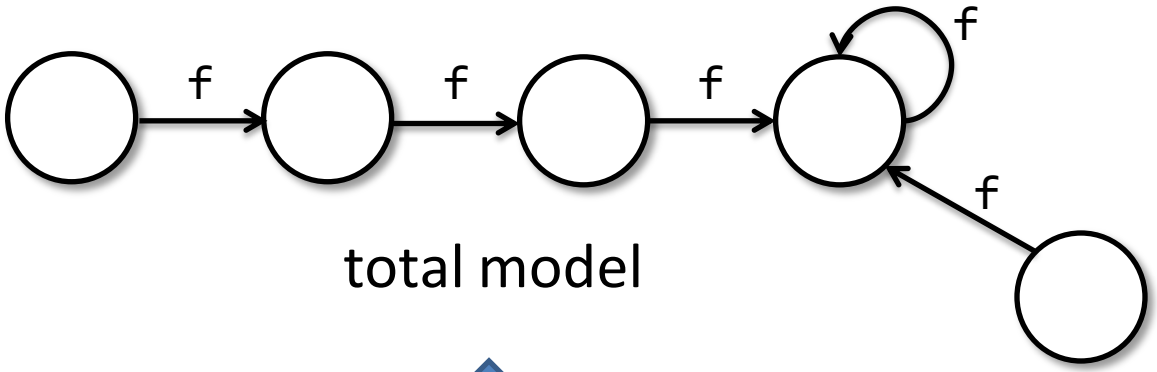
Need to consider more general decidable fragments:
Local Theory Extensions
[Sofronie-Stokkermans, CADE'05], [Bansal et al., CAV'15]

Model Completion for Local Theory Extensions



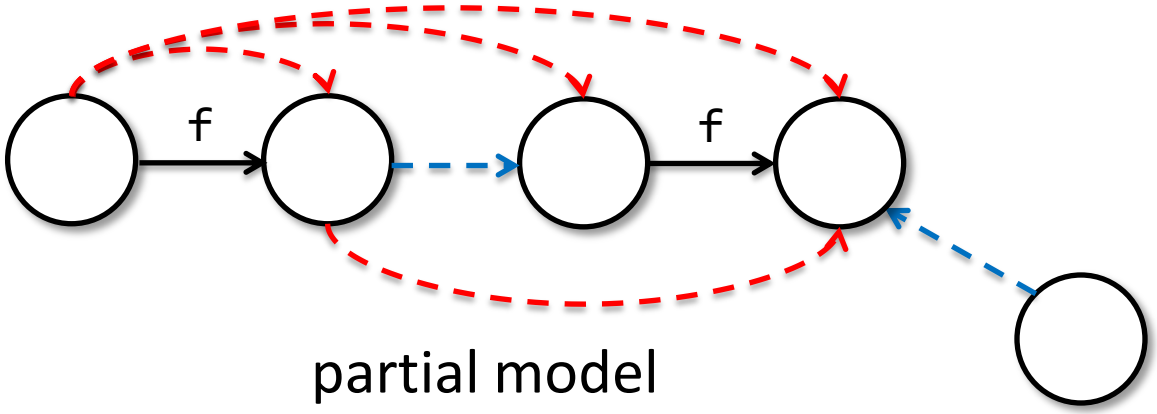


Model Completion for Local Theory Extensions



total model

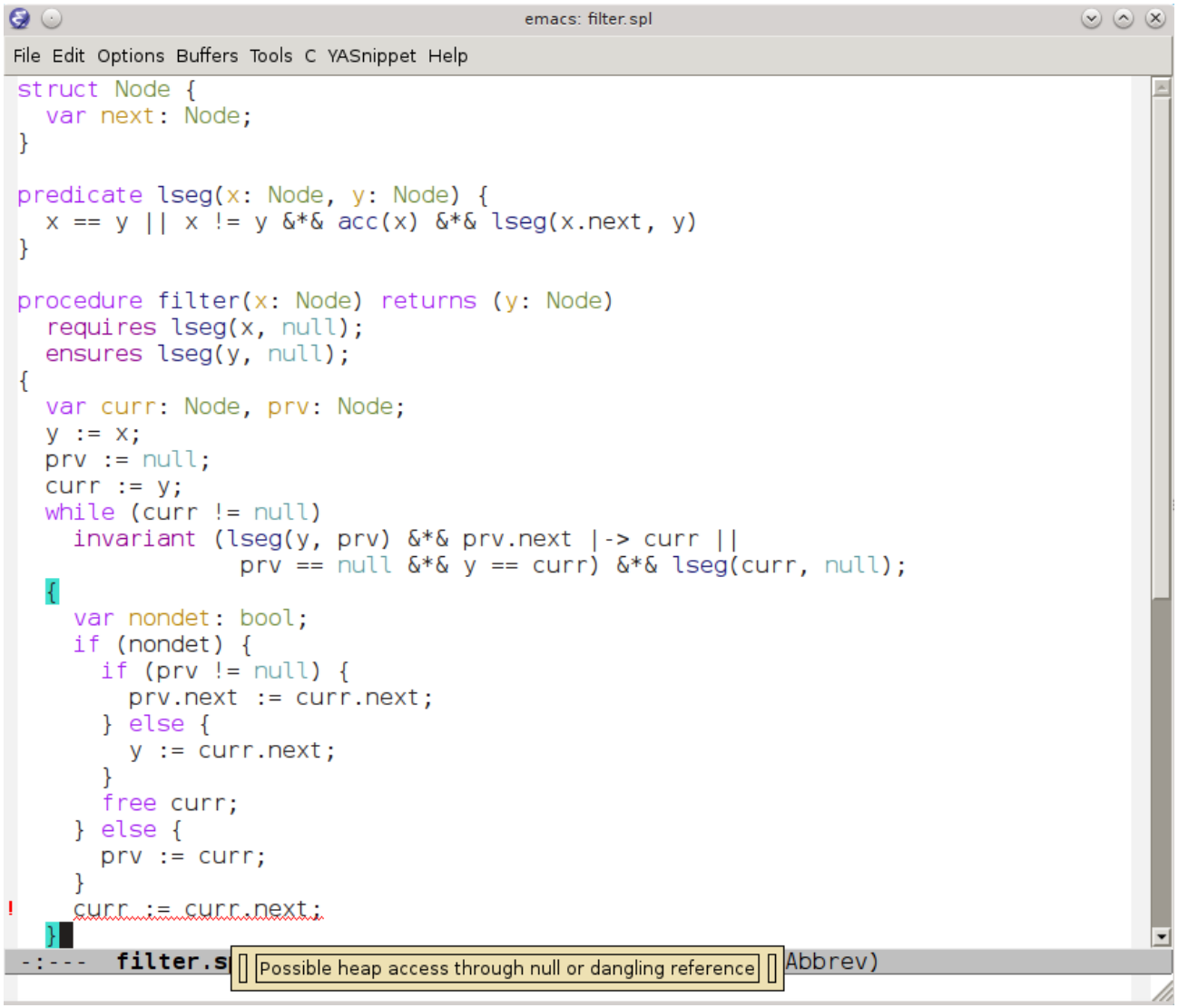
Yields NP decision procedure for GRASS



partial model

GRASShopper

http://cs.nyu.edu/wies/software/grasshopper



GRASShopper

http://cs.nyu.edu/wies/software/grasshopper

- Solvers
  - Frontend: C-like language with mixed SL/FOL specifications
  - Supported backend solvers: CVC4, Z3
- Benchmarks (~2000 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 (work in progress)

Conclusions

- What we did: reduce reasoning in separation logic to reasoning in first-order predicate logic
- Pluses
  - best of both worlds (succinct specs, flexibility)
  - reduction remains in decidable fragments
  - theory combination for free
- Minuses
  - no intuitive proofs
  - somewhat heavier machinery