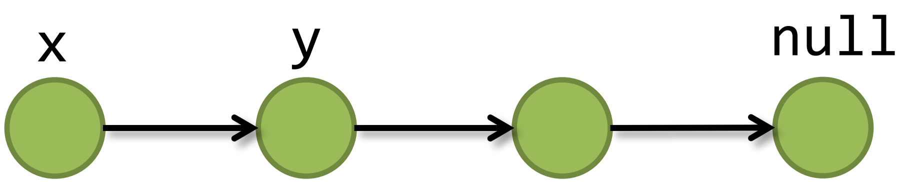
Introduction to Permission-Based Program Logics

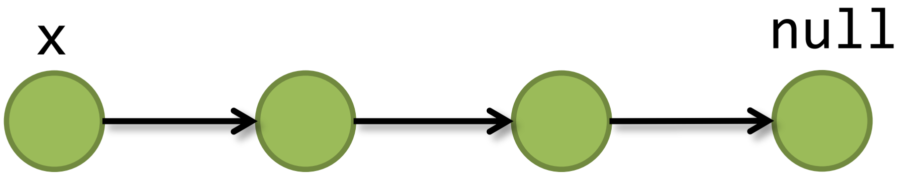
Thomas Wies
New York University

A Motivating Example

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



Proof by Hand-Waving

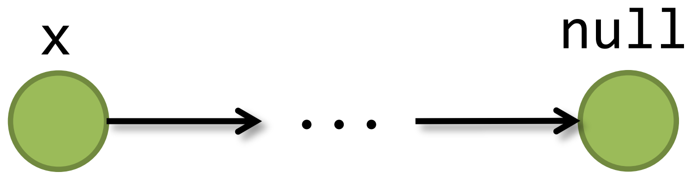
{


}

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

{ }

Proof by Hand-Waving

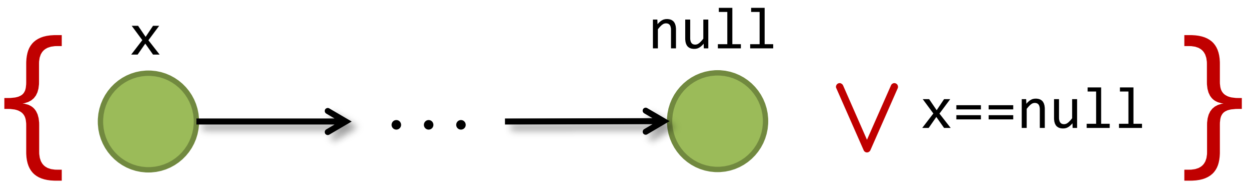
{ 

∨ x==null }

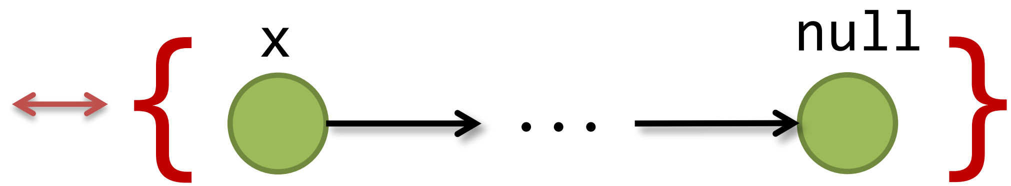
```
procedure delete(x: Node)
{
  if (x != null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
```

{ }

Proof by Hand-Waving

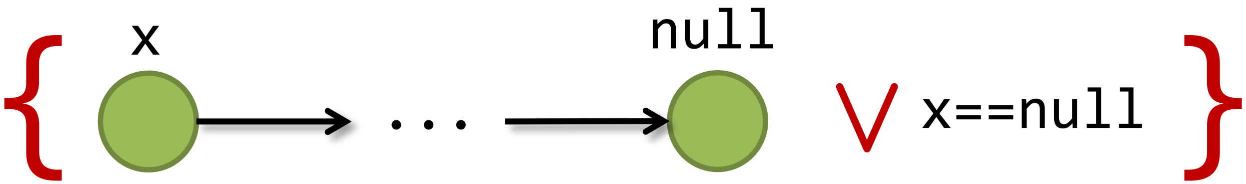



```
procedure delete(x: Node)
{
  if (x != null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
```

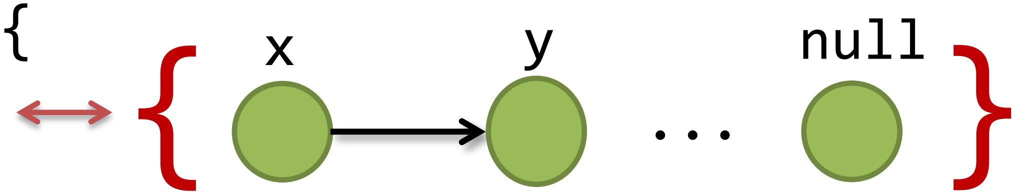


{ }

Proof by Hand-Waving

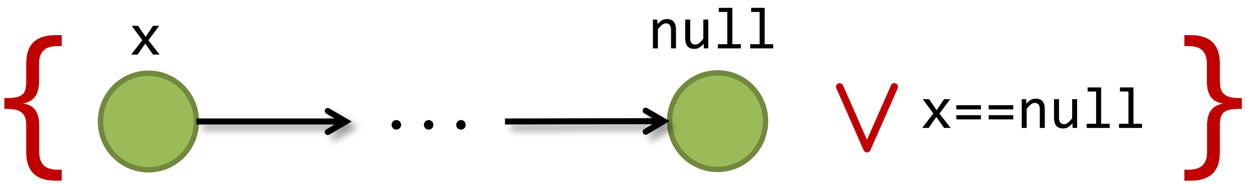



```
procedure delete(x: Node)
{
  if (x != null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
```

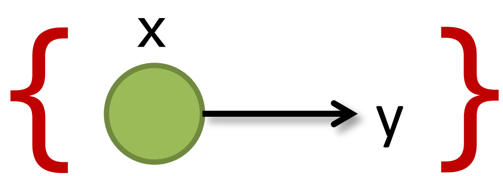


{ }

Proof by Hand-Waving

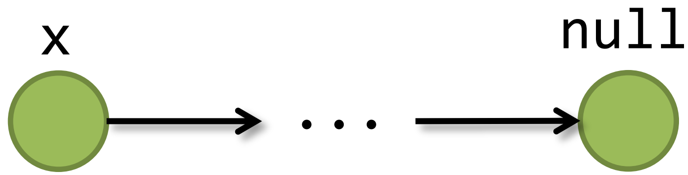



```
procedure delete(x: Node)
{
  if (x != null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
```



{ }

Proof by Hand-Waving

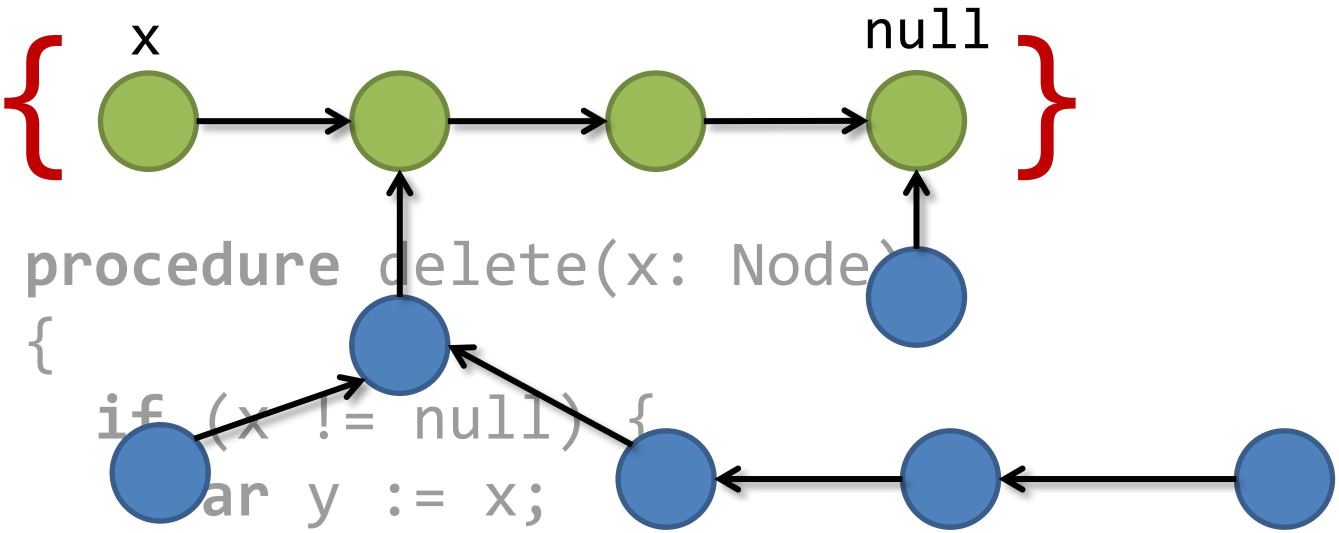
{ 

∨ x==null }

```
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;
    delete(y);
    free(x);
  }
}
```

{ }

Road Map

Part I – Sequential Programs

Part II – Concurrent Programs

Permission-based Logics

- Separation Logic
  - O'Hearn, Pym 1999 (boolean bunched implications)
  - O'Hearn, Reynolds, Yang 2001
  - Reynolds 2002
  - …
- Implicit Dynamic Frames
  - Smans, Jacobs, Piessens 2008
  - Parkinson, Summers 2011
  - …
- Linear maps
  - Lahiri, Qadeer, Walker 2011
- …

Tools and Projects using Permission-based Logics

- CompCert (Inria)
- L4.Verified (Data 61)
- Bedrock (MIT)
- …

interactive deductive verification

- Smallfoot (UCL, Imperial)
- Chalice (Microsoft)
- VeriFast (KU Leuven)
- HIP (Singapore)
- Viper (ETH)
- GRASShopper (NYU, Yale, MPI-SWS)
- …

automated deductive verifiers

- Space Invader (UCL, Imperial)
- SLAyer (Microsoft)
- Infer (Facebook)
- Xisa (Boulder, Paris, Berkeley)
- …

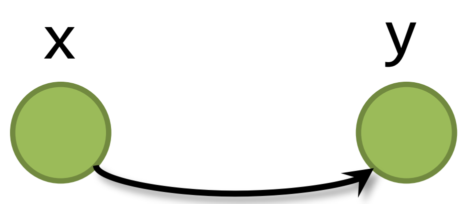
static program analysis tools

Separation Logic (SL)

Separation Logic by Example

- Points-to predicates

$$x \mapsto y$$



Stack

x	10
y	42
…	

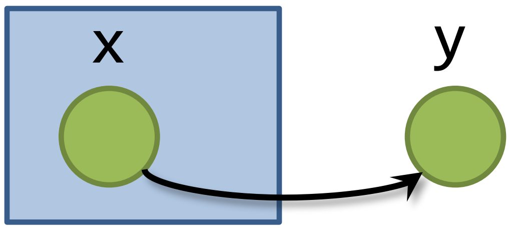
Heap

10	42
…	
42	?

Separation Logic by Example

- Points-to predicates

$$x \mapsto y$$



Stack

x	10
y	42
…	

Heap

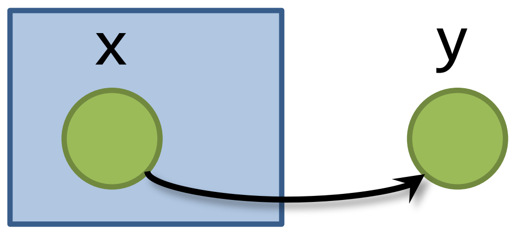
10	42
…	
42	?

A partial heap consisting of one allocated cell

Separation Logic by Example

- Points-to predicates

$$x \mapsto y$$



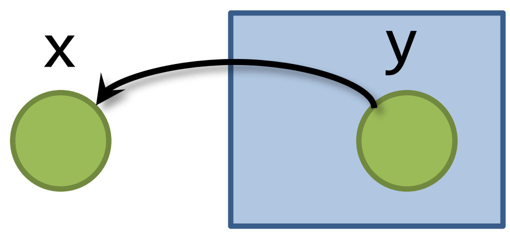
Points-to predicate Expresses permission to access (i.e. read/write/deallocate) heap location x and nothing else!

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

Separation Logic by Example

- Points-to predicates

$$y \mapsto x$$



Stack

x	10
y	42
…	

Heap

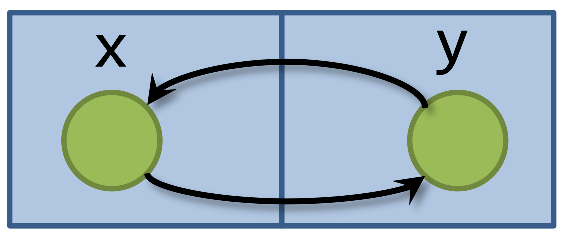
10	?
…	
42	10

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

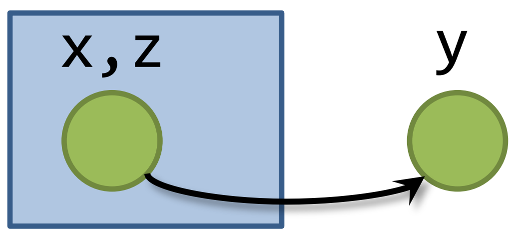
10	42
…	
42	10

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

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

?

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

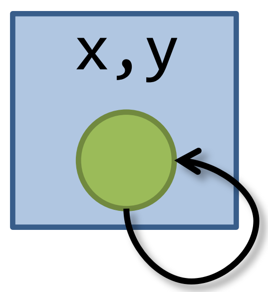
$$x \mapsto y \land y \mapsto x$$

?

Separation Logic by Example

- Classical conjunction

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



Separation Logic by Example

- Separating conjunction

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

?

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

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 $*$

Separation Logic: Syntax

- Terms e, t
  - variables: $x \in Var$
  - …

- Assertions P, Q
  - equalities: $e = t$
  - empty heap: emp
  - points-to: $e \mapsto t$
  - separating conjunction: $P * Q$
  - magic wand: $P -* Q$
  - classical conjunction: $P \wedge Q$
  - negation: $\neg P$
  - existential quantification: $\exists x. P$
  - (inductively defined predicates)

Separation Logic: Assertion Semantics

- Domains
  - addresses: Addr (= $\mathbb{N}$)
  - values: Val = Addr $\cup$ …
- A state $\sigma \in \Sigma$ is a pair (h, s) of a stack s and a heap h
  - s: Var $\rightarrow$ Val
  - h: Addr $\rightharpoonup$ Val
- Composition of states
  - $(h_1, s_1) \bullet (h_2, s_2) = (h_1 \cup h_2, s_1)$ if $s_1 = s_2$ and $h_1 \perp h_2$
  - $(h_1, s_1) \bullet (h_2, s_2)$ undefined otherwise

  Here, $h_1 \perp h_2$ means domains of h_1 and h_2 are disjoint

Separation Logic: Assertion Semantics

- t^s: denotation of term t in stack s
- $(h, s) \models e = t \quad \Leftrightarrow e^s = t^s$
- $(h, s) \models \mathrm{emp} \quad \Leftrightarrow h = \{\}$
- $(h, s) \models e \mapsto t \quad \Leftrightarrow h = \{e^s \mapsto t^s\}$
- $\sigma \models P * Q \quad \Leftrightarrow$ exists σ_1, σ_2 s.t. $\sigma = \sigma_1 \bullet \sigma_2$ and $\sigma_1 \models P$ and $\sigma_2 \models Q$
- $\sigma \models P \text{ -}{*}\ Q \quad \Leftrightarrow$ for all σ_1 s.t. $\sigma_1 \perp \sigma$, $\sigma_1 \models P$ implies $\sigma_1 \bullet \sigma \models Q$
- $\sigma \models P \land Q \quad \Leftrightarrow \sigma \models P$ and $\sigma \models Q$
- … everything else as in classical logic

Entailment

Entailment between assertions is defined as usual

P $\models$ Q $\Leftrightarrow$ for all σ, $\sigma \models$ P implies $\sigma \models$ Q

SL is a Substructural Logic

- Duplication of hypotheses is not allowed

$$P \not\models P * P$$

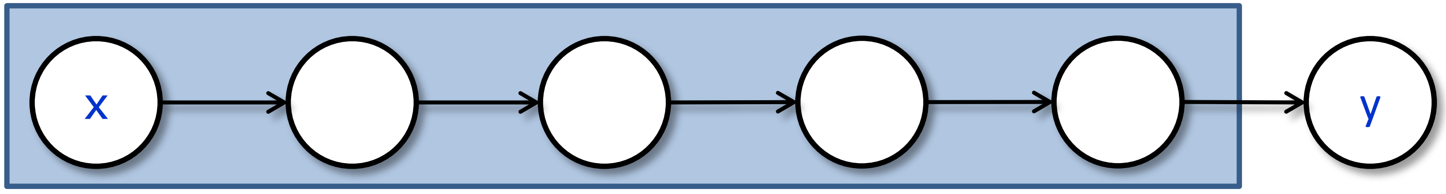
- Weakening is not allowed

$$P * Q \not\models P$$

Inductively Defined Predicates

- acyclic list segment

$$\mathbf{lseg}(x,\ y) \equiv$$
$$x = y \lor \exists z.\, x \neq y \land x \mapsto z * \mathbf{lseg}(z,\ y)$$



Inductively Defined Predicates

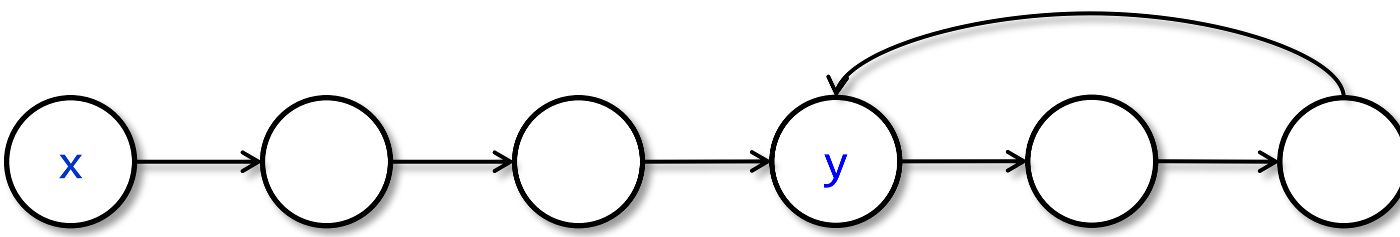
- acyclic list segment

$$\mathbf{lseg}(x, y) \equiv$$
$$x = y \vee \exists z. \quad x \mapsto z * \mathbf{lseg}(z, y)$$

Inductively Defined Predicates

- acyclic list segment

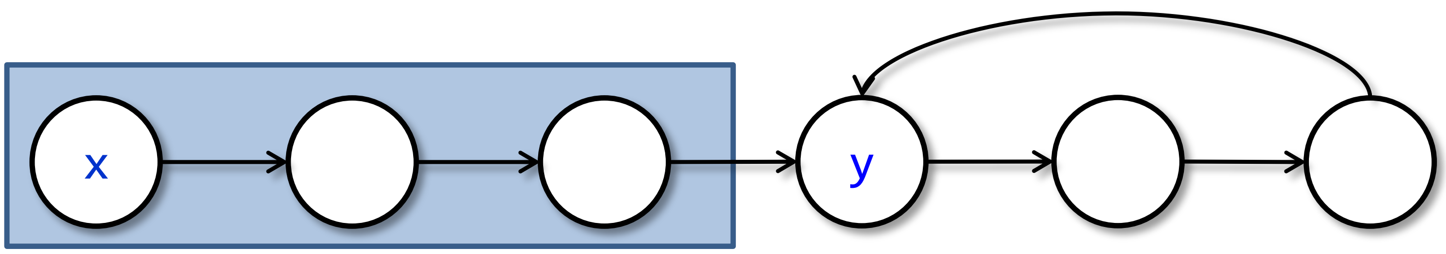
$$\mathbf{lseg}(x, y) \equiv$$
$$x = y \lor \exists z. \quad x \mapsto z * \mathbf{lseg}(z, y)$$



Inductively Defined Predicates

- acyclic list segment

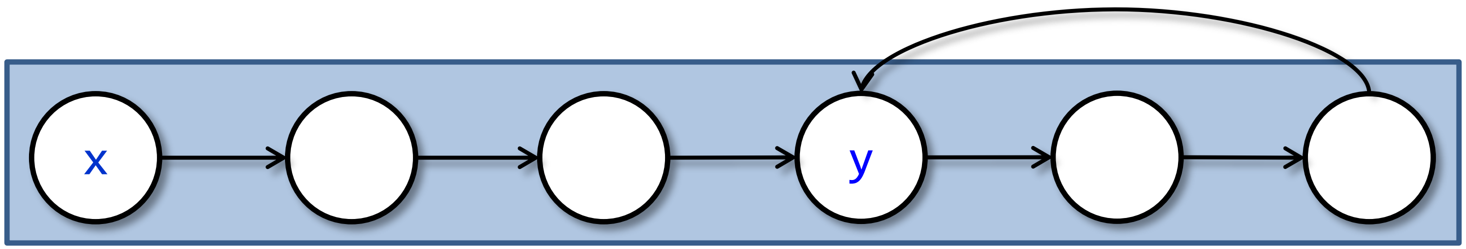
$$\mathbf{lseg}(x, y) \equiv$$
$$x = y \lor \exists z. \quad x \mapsto z * \mathbf{lseg}(z, y)$$



Inductively Defined Predicates

- acyclic list segment

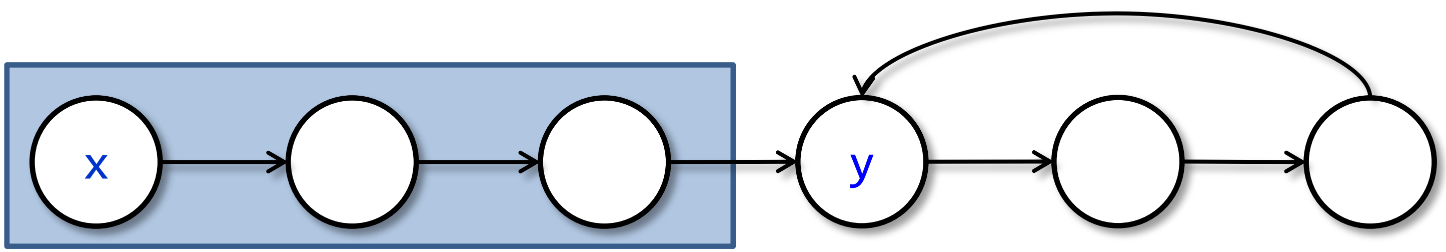
$$\mathbf{lseg}(x, y) \equiv$$
$$x = y \lor \exists z. \quad x \mapsto z * \mathbf{lseg}(z, y)$$



Inductively Defined Predicates

- acyclic list segment

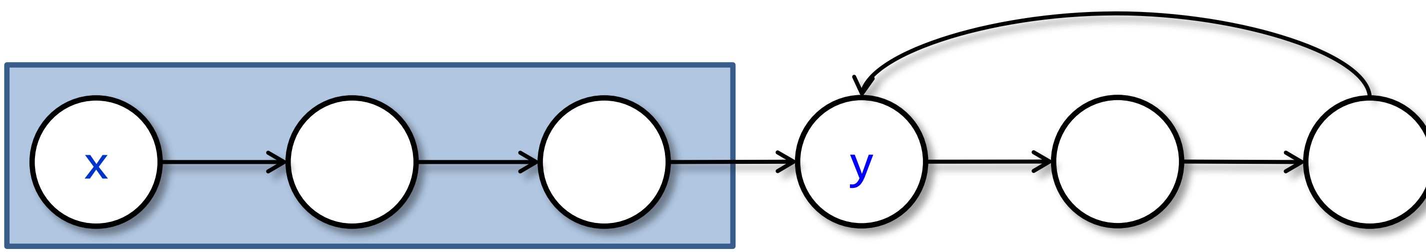
$$\mathbf{lseg}(x, y) \equiv$$
$$x = y \lor \exists z. \quad x \mapsto z * \mathbf{lseg}(z, y)$$



Inductively Defined Predicates

- acyclic list segment

$$\mathbf{lseg}(x, y) \equiv$$
$$x = y \vee \exists z. \quad x \mapsto z * \mathbf{lseg}(z, y)$$



Predicate is not *precise*

Abstract Separation Logic

[Calcagno, O'Hearn, Yang 2007]

- Theory of separation logic also works for other semantics than the standard heap model.
- In general, any *separation algebra* can be used.
- Separation algebra (Σ, $\bullet$, e) is a partial commutative cancelative monoid, i.e. for all $\sigma, \sigma_1, \sigma_2, \sigma_3 \in \Sigma$
  - unit: $\sigma \bullet e = \sigma$
  - associative: $(\sigma_1 \bullet \sigma_2) \bullet \sigma_3 = \sigma_1 \bullet (\sigma_2 \bullet \sigma_3)$
  - commutative: $\sigma_1 \bullet \sigma_2 = \sigma_2 \bullet \sigma_1$
  - cancelative: $\sigma \bullet \sigma_1 = \sigma \bullet \sigma_2 \Rightarrow \sigma_1 = \sigma_2$

Here, equality means either both sides are defined and equal or both sides are undefined.

Example: Fractional Permissions

[Bornat et al. 2005]

- Heaps with fractional permissions:
  - h: Addr $\times$ Val $\rightarrow [0, 1]$
  - $h_1 \bullet h_2 = h_1 + h_2$ if for all a,v: $h_1(a, v) + h_2(a, v) \leq 1$
  - $h_1 \bullet h_2$ undefined otherwise

- Useful for reasoning about concurrent programs

SL-based Hoare Logic

Programs: Syntax

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

Programs: Operational Semantics

- Reduction relation
  $\rightarrow \subseteq (\text{Com} \times \Sigma) \times (\text{Com} \times \Sigma) \uplus \{\mathbf{abort}\}$
  - Notation: $\langle$ C, σ $\rangle \rightarrow \langle$ C', σ' $\rangle$
  - Meaning: Command C takes a step in state σ, yielding continuation C' and state σ'

Operational Semantics

- $\langle$ **assume**(b), σ $\rangle \rightarrow \langle$ **skip**, σ $\rangle$ if $\sigma \models$ b
- $\langle$ [x] := y, (s, h) $\rangle \rightarrow \langle$ **skip**, (s, h[$x^s \mapsto y^s$]) $\rangle$ if $x^s \in$ dom(h)
- $\langle$ [x] := y, (s, h) $\rangle \rightarrow$ **abort** if $x^s \notin$ dom(h)
- …
- $$\frac{\langle C, \sigma \rangle \rightarrow \langle C', \sigma' \rangle}{\langle C^*, \sigma \rangle \rightarrow \langle C'; C^*, \sigma' \rangle}$$ $\langle C^*, \sigma \rangle \rightarrow \langle$ **skip**, σ $\rangle$

Locality of Operational Semantics

- Separation Logic works for any operational semantics that is *local*
- A command C is *local* iff for all σ, σ_1, σ_2, σ', C'
  - if $\langle$ C, $\sigma_1 \bullet \sigma_2$ $\rangle \rightarrow \langle$ C', σ' $\rangle$ then either
    - $\langle$ C, $\sigma_1 \rangle \rightarrow$ **abort**
    - exists σ_1' s.t. $\langle$ C, $\sigma_1 \rangle \rightarrow \langle$ C', σ_1' $\rangle$ and $\sigma' = \sigma_1' \bullet \sigma_2$
  - if $\langle$ C, σ $\rangle \rightarrow \langle$ C', σ' $\rangle$ and $\sigma \bullet \sigma_1$ defined, then
    - $\langle$ C, $\sigma \bullet \sigma_1 \rangle \rightarrow \langle$ C', $\sigma' \bullet \sigma_1 \rangle$

Hoare Logic

- Hoare triples $\{ P \}\ C\ \{ Q \}$
  - Meaning:
    - C executes without failure from any state satisfying P.
    - Moreover, if C terminates, then the final state satisfies Q.
  - Formally: $\{ P \}\ C\ \{ Q \}$ is valid iff for all σ, σ'
    - $\sigma \vDash P$ and $\langle\, C, \sigma \,\rangle \not\to^{*}$ **abort**
    - $\sigma \vDash P$ and $\langle\, C, \sigma \,\rangle \to^{*} \langle \mathbf{skip}, \sigma' \rangle$ implies $\sigma' \vDash Q$

Hoare Triples: Examples

- $\{ x = 15 \}$ `y := [x]` $\{ x = 15 \wedge y = 15 \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \land y = 15 \}$ Valid? 

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}$ y := [x] $\{ \mathit{true} \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}$ y := [x] $\{ \mathit{true} \}$ Valid? 

Hoare Triples: Examples

- $\{ x = 15 \}\ y := [x]\ \{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x = 15 \}\ y := [x]\ \{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}\ y := x\ \{ x = 15 \land y = 15 \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x = 15 \}$ y := [x] $\{ \textit{true} \}$ Valid? 
- $\{ x = 15 \}$ y := x $\{ x = 15 \land y = 15 \}$ Valid? 

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}$ y := [x] $\{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}$ y := x $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}$ y := [x] $\{ x \mapsto 15 \wedge y = 15 \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}$ y := [x] $\{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}$ y := x $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}$ y := [x] $\{ x \mapsto 15 \wedge y = 15 \}$ Valid? 

Hoare Triples: Examples

- $\{ x = 15 \}\ y := [x]\ \{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x = 15 \}\ y := [x]\ \{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}\ y := x\ \{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}\ y := [x]\ \{ x \mapsto 15 \land y = 15 \}$ Valid? 
- $\{ x \mapsto 15 * z \mapsto 42 \}\ y := [x]\ \{ x \mapsto 15 * z \mapsto 42 \land y = 15 \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}$ y := [x] $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}$ y := [x] $\{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}$ y := x $\{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}$ y := [x] $\{ x \mapsto 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 * z \mapsto 42 \}$ y := [x] $\{ x \mapsto 15 * z \mapsto 42 \wedge y = 15 \}$ Valid? 

Hoare Triples: Examples

- $\{ x = 15 \}$ $y := [x]$ $\{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x = 15 \}$ $y := [x]$ $\{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}$ $y := x$ $\{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}$ $y := [x]$ $\{ x \mapsto 15 \land y = 15 \}$ Valid? 
- $\{ x \mapsto 15 * z \mapsto 42 \}$ $y := [x]$ $\{ x \mapsto 15 * z \mapsto 42 \land y = 15 \}$ Valid? 
- $\{ y \mapsto 42 \land x = 15 \}$ $[y] := x$ $\{ 15 \mapsto 42 \land x = 15 \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}\ y := [x]\ \{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}\ y := [x]\ \{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}\ y := x\ \{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}\ y := [x]\ \{ x \mapsto 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 * z \mapsto 42 \}\ y := [x]\ \{ x \mapsto 15 * z \mapsto 42 \wedge y = 15 \}$ Valid? 
- $\{ y \mapsto 42 \wedge x = 15 \}\ [y] := x\ \{ 15 \mapsto 42 \wedge x = 15 \}$ Valid? 

Hoare Triples: Examples

- $\{ x = 15 \}\ y := [x]\ \{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x = 15 \}\ y := [x]\ \{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}\ y := x\ \{ x = 15 \land y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}\ y := [x]\ \{ x \mapsto 15 \land y = 15 \}$ Valid? 
- $\{ x \mapsto 15 * z \mapsto 42 \}\ y := [x]\ \{ x \mapsto 15 * z \mapsto 42 \land y = 15 \}$ Valid? 
- $\{ y \mapsto 42 \land x = 15 \}\ [y] := x\ \{ 15 \mapsto 42 \land x = 15 \}$ Valid? 
- $\{ y \mapsto 42 \land x = 15 \}\ [y] := x\ \{ y \mapsto 15 \land x = 15 \}$ Valid?

Hoare Triples: Examples

- $\{ x = 15 \}\ y := [x]\ \{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x = 15 \}\ y := [x]\ \{ \mathit{true} \}$ Valid? 
- $\{ x = 15 \}\ y := x\ \{ x = 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 \}\ y := [x]\ \{ x \mapsto 15 \wedge y = 15 \}$ Valid? 
- $\{ x \mapsto 15 * z \mapsto 42 \}\ y := [x]\ \{ x \mapsto 15 * z \mapsto 42 \wedge y = 15 \}$ Valid? 
- $\{ y \mapsto 42 \wedge x = 15 \}\ [y] := x\ \{ 15 \mapsto 42 \wedge x = 15 \}$ Valid? 
- $\{ y \mapsto 42 \wedge x = 15 \}\ [y] := x\ \{ y \mapsto 15 \wedge x = 15 \}$ Valid? 

Tight Axioms for Basic Commands

- Heap write
  $\{ x \mapsto z \}$ [x] := y $\{ x \mapsto y \}$

Tight Axioms for Basic Commands

- Heap write
  $\{ x \mapsto z \}$ [x] := y $\{ x \mapsto y \}$
- Heap read
  $\{ y \mapsto z \wedge x = n \}$ x := [y] $\{ y[n/x] \mapsto z \wedge x = z \}$

Tight Axioms for Basic Commands

- Heap write
  $\{\, x \mapsto z \,\}$ [x] := y $\{\, x \mapsto y \,\}$
- Heap read
  $\{\, y \mapsto z \wedge x = n \,\}$ x := [y] $\{\, y[n/x] \mapsto z \wedge x = z \,\}$
- Allocation
  $\{\, emp \,\}$ x := **new**() $\{\, x \mapsto z \,\}$

Tight Axioms for Basic Commands

- Heap write
  $\{\, x \mapsto z \,\}$ [x] := y $\{\, x \mapsto y \,\}$
- Heap read
  $\{\, y \mapsto z \land x = n \,\}$ x := [y] $\{\, y[n/x] \mapsto z \land x = z \,\}$
- Allocation
  $\{\, \mathrm{emp} \,\}$ x := **new**() $\{\, x \mapsto z \,\}$
- Deallocation
  $\{\, x \mapsto z \,\}$ **free**(x) $\{\, \mathrm{emp} \,\}$
- …

Structural Rules

- Frame rule

$$\frac{\{\, P \,\}\ C\ \{\, Q \,\}}{\{\, P * F \,\}\ C\ \{\, Q * F \,\}} \qquad \mathrm{mod}(C) \cap \mathrm{fv}(F) = \emptyset$$

$\mathrm{mod}(x := [y]) = \{x\}$, $\mathrm{mod}([x] := y) = \emptyset$, ...

Structural Rules

- Frame rule

$$\frac{\{\ P\ \}\ C\ \{\ Q\ \}}{\{\ P * F\ \}\ C\ \{\ Q * F\ \}} \qquad mod(C) \cap fv(F) = \emptyset$$

- Consequence rule

$$\frac{P' \vDash P \qquad \{\ P\ \}\ C\ \{\ Q\ \} \qquad Q \vDash Q'}{\{\ P'\ \}\ C\ \{\ Q'\ \}}$$

$mod(x := [\,y\,]) = \{x\}$, $mod([\,x\,] := y) = \emptyset$, ...

Structural Rules

- Frame rule

$$\frac{\{\ P\ \}\ C\ \{\ Q\ \}}{\{\ P * F\ \}\ C\ \{\ Q * F\ \}} \quad \mathrm{mod}(C) \cap \mathrm{fv}(F) = \emptyset$$

- Consequence rule

$$\frac{P' \vDash P \qquad \{\ P\ \}\ C\ \{\ Q\ \} \qquad Q \vDash Q'}{\{\ P'\ \}\ C\ \{\ Q'\ \}}$$

- $\exists$ introduction rule

$$\frac{\{\ P\ \}\ C\ \{\ Q\ \}}{\{\ \exists\, z.\ P\ \}\ C\ \{\ \exists\, z.\ Q\ \}} \quad z \notin \mathrm{fv}(C)$$

$\mathrm{mod}(x := [\,y\,]) = \{x\}$, $\mathrm{mod}([\,x\,] := y) = \emptyset$, ...

Structural Rules

- Frame rule

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P * F\,\}\ C\ \{\,Q * F\,\}} \qquad mod(C) \cap fv(F) = \emptyset$$

- Consequence rule

$$\frac{P' \models P \qquad \{\,P\,\}\ C\ \{\,Q\,\} \qquad Q \models Q'}{\{\,P'\,\}\ C\ \{\,Q'\,\}}$$

- $\exists$ introduction rule

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,\exists z.\ P\,\}\ C\ \{\,\exists z.\ Q\,\}} \qquad z \notin fv(C)$$

- Variable substitution rule

$$\frac{\{\,P\,\}\ C\ \{\,Q\,\}}{\{\,P\,\}\ C\ \{\,Q\,\}\ [e_1/x_1, \ldots, e_n/x_n]} \qquad \begin{array}{l} fv(P, C, Q) \subseteq \{x_1, \ldots, x_n\} \\ x_i \in mod(C) \Rightarrow e_i \in Var \setminus fv(\{e_j\}_{j \neq i}) \end{array}$$

$mod(x := [\,y\,]) = \{x\}$, $mod([\,x\,] := y) = \emptyset$, ...

Remaining Constructs as in Classical Hoare Logic

- Sequencing rule

$$\frac{\{\,P\,\}\; C_1\; \{\,R\,\} \quad \{\,R\,\}\; C_2\; \{\,Q\,\}}{\{\,P\,\}\; C_1\,;\; C_2\; \{\,Q\,\}}$$

- Choice rule

$$\frac{\{\,P\,\}\; C_1\; \{\,Q\,\} \quad \{\,P\,\}\; C_2\; \{\,Q\,\}}{\{\,P\,\}\; C_1 + C_2\; \{\,Q\,\}}$$

- Loop rule

$$\frac{\{\,I\,\}\; C\; \{\,I\,\}}{\{\,I\,\}\; C^*\; \{\,I\,\}}$$

Integration into Verification Tools

Any SL-based verification tool will have to implement at least the following two tasks:

- Mechanize Hoare Logic Rules using either
  - symbolic forward execution, or
  - verification condition generation
- Mechanize Validity Checking of SL Entailments

Symbolic Heap Fragment with Linked Lists

- Only consider assertions of the form
  - $\exists \mathbf{x}.\ P \land Q$

    where
  - P is a conjunction of equalities and disequalities
  - Q is a separating conjunction of
    - points-to predicates $x \mapsto y$
    - list segment predicates $\mathbf{lseg}(x, y)$
- Example: $\exists y.\ x \neq z \land x \mapsto y * \mathrm{lseg}(y, z)$

Symbolic Forward Execution

Define relation $H, c \rightsquigarrow H'$ that given H and c computes H' such that $\{ H \}\ c\ \{ H' \}$ is valid

Idea:

- Combine Hoare rules for basic commands with frame rule
  $\Rightarrow$ specialized rules for executing basic
  commands on symbolic heaps
- Specialize consequence rule to so-called *rearrangement* rules that materialize points-to predicates for heap accesses.

Symbolic Forward Execution Rules

- Variable assignment
  H, `x := t` $\rightsquigarrow$ $x = t[x'/x] \wedge H[x'/x]$

Symbolic Forward Execution Rules

- Variable assignment
  H, `x := t` $\rightsquigarrow$ $x = t[x'/x] \wedge H[x'/x]$
- Heap read
  $H * y \mapsto z$, `x := [y]` $\rightsquigarrow$ $x = z[x'/x] \wedge (H * y \mapsto z)[x'/x]$

Symbolic Forward Execution Rules

- Variable assignment
  $H, \texttt{x := t} \rightsquigarrow x = t[x'/x] \wedge H[x'/x]$
- Heap read
  $H * y \mapsto z, \texttt{x := [y]} \rightsquigarrow x = z[x'/x] \wedge (H * y \mapsto z)[x'/x]$
- Heap write
  $H * x \mapsto z, \texttt{[x] := y} \rightsquigarrow H * x \mapsto y$

Symbolic Forward Execution Rules

- Variable assignment
  H, `x := t` $\leadsto$ $x = t[x'/x] \wedge H[x'/x]$
- Heap read
  $H * y \mapsto z$, `x := [y]` $\leadsto$ $x = z[x'/x] \wedge (H * y \mapsto z)[x'/x]$
- Heap write
  $H * x \mapsto z$, `[x] := y` $\leadsto$ $H * x \mapsto y$
- Allocation
  H, `x := new()` $\leadsto$ $H[x'/x] * x \mapsto z$

Symbolic Forward Execution Rules

- Variable assignment
  $H, \texttt{x := t} \rightsquigarrow x = t[x'/x] \land H[x'/x]$
- Heap read
  $H * y \mapsto z, \texttt{x := [y]} \rightsquigarrow x = z[x'/x] \land (H * y \mapsto z)[x'/x]$
- Heap write
  $H * x \mapsto z, \texttt{[x] := y} \rightsquigarrow H * x \mapsto y$
- Allocation
  $H, \texttt{x := }\textbf{new}\texttt{()} \rightsquigarrow H[x'/x] * x \mapsto z$
- Deallocation
  $H * x \mapsto z, \textbf{free}\texttt{(x)} \rightsquigarrow H$

Rearrangement Rules

A(x) ::= [x] := y | y := [x]
P(x,y) ::= $x \mapsto y$ | **lseg**(x, y)

- $$\frac{H_0 * P(x, y), A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x = z}{H_0 * P(z, y), A(x) \rightsquigarrow H_1}$$

Rearrangement Rules

$A(x) ::= [x] := y \mid y := [x]$
$P(x,y) ::= x \mapsto y \mid \mathbf{lseg}(x, y)$

- $$\frac{H_0 * P(x, y), A(x) \rightsquigarrow H_1}{H_0 * P(z, y), A(x) \rightsquigarrow H_1} \quad H_0 \vdash x = z$$
- $$\frac{H_0 * x \mapsto z * \mathbf{lseg}(z, y), A(x) \rightsquigarrow H_1}{H_0 * \mathbf{lseg}(x, y), A(x) \rightsquigarrow H_1} \quad H_0 \vdash x \neq y$$

Rearrangement Rules

$A(x) ::= [x] := y \mid y := [x]$
$P(x,y) ::= x \mapsto y \mid \mathbf{lseg}(x, y)$

- $$\frac{H_0 * P(x, y), A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x = z}{H_0 * P(z, y), A(x) \rightsquigarrow H_1}$$
- $$\frac{H_0 * x \mapsto z * \mathbf{lseg}(z, y), A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x \neq y}{H_0 * \mathbf{lseg}(x, y), A(x) \rightsquigarrow H_1}$$
- $$\frac{H_0 * x \mapsto y, A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x \neq y}{H_0 * \mathbf{lseg}(x, y), A(x) \rightsquigarrow H_1}$$

Rearrangement Rules

$A(x)$ ::= [x] := y | y := [x]
$P(x,y)$ ::= $x \mapsto y$ | **lseg**(x, y)

- $$\frac{H_0 * P(x, y), A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x = z}{H_0 * P(z, y), A(x) \rightsquigarrow H_1}$$
- $$\frac{H_0 * x \mapsto z * \mathbf{lseg}(z, y), A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x \neq y}{H_0 * \mathbf{lseg}(x, y), A(x) \rightsquigarrow H_1}$$
- $$\frac{H_0 * x \mapsto y, A(x) \rightsquigarrow H_1 \qquad H_0 \vdash x \neq y}{H_0 * \mathbf{lseg}(x, y), A(x) \rightsquigarrow H_1}$$
- $$\frac{H_0 \nvdash \exists y.\ x \mapsto y * \mathrm{true}}{H_0, A(x) \rightsquigarrow \mathbf{abort}}$$

Symbolic Forward Execution: Example

$x \neq \text{null} \wedge \textbf{lseg}(x, \text{null})$, `y := [x]` $\rightsquigarrow$?

Symbolic Forward Execution: Example

$$\frac{x \neq \text{null} \land x \mapsto z * \mathbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow ? \qquad x \neq \text{null} \vdash x \neq \text{null}}{x \neq \text{null} \land \mathbf{lseg}(x, \text{null}),\ y := [x] \rightsquigarrow ?}$$

Symbolic Forward Execution: Example

$$\frac{x \neq \text{null} \wedge x \mapsto z * \textbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow x \neq \text{null} \wedge y = z \wedge x \mapsto z * \textbf{lseg}(z, \text{null})}{x \neq \text{null} \wedge x \mapsto z * \textbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow ?} \quad x \neq \text{null} \vdash x \neq \text{null}$$

$$\frac{}{x \neq \text{null} \wedge \textbf{lseg}(x, \text{null}),\ y := [x] \rightsquigarrow ?}$$

Symbolic Forward Execution: Example

$$x \neq \text{null} \land x \mapsto z * \textbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow$$
$$x \neq \text{null} \land y = z \land x \mapsto z * \textbf{lseg}(z, \text{null})$$

$$x \neq \text{null} \land x \mapsto z * \textbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow \qquad x \neq \text{null} \vdash x \neq \text{null}$$
$$x \neq \text{null} \land y = z \land x \mapsto z * \textbf{lseg}(z, \text{null})$$

$$x \neq \text{null} \land \textbf{lseg}(x, \text{null}),\ y := [x] \rightsquigarrow ?$$

Symbolic Forward Execution: Example

$$\frac{x \neq \text{null} \wedge x \mapsto z * \textbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow \quad x \neq \text{null} \wedge y = z \wedge x \mapsto z * \textbf{lseg}(z, \text{null})}{}$$

$$\frac{x \neq \text{null} \wedge x \mapsto z * \textbf{lseg}(z, \text{null}),\ y := [x] \rightsquigarrow \quad x \neq \text{null} \wedge y = z \wedge x \mapsto z * \textbf{lseg}(z, \text{null})}{x \neq \text{null} \wedge \textbf{lseg}(x, \text{null}),\ y := [x] \rightsquigarrow \quad x \neq \text{null} \wedge y = z \wedge x \mapsto y * \textbf{lseg}(y, \text{null})} \quad x \neq \text{null} \vdash x \neq \text{null}$$

Symbolic Forward Execution: Example

lseg(x, null), y := [x] ⇝ ?

Symbolic Forward Execution: Example

$$\frac{\mathrm{lseg}(x, \mathrm{null}) \nvdash \exists z.\ x \mapsto z * \mathrm{true}}{\mathbf{lseg}(x, \mathrm{null}),\ y := [x] \rightsquigarrow \mathbf{abort}}$$

Entailment Checking

- Various decidable fragments
  - Symbolic heaps with linked lists [Berdine, Calcagno, O'Hearn 2005], [Cook et al. 2011]
  - Propositional closure of symbolic heap with linked lists [Piskac, Zufferey, Wies 2013]
  - Recursive predicates of bounded tree width [Iosif, Rogalewicz, Simacek 2013]
  - …
  - also see survey [Demri, Deters 2015]

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn 2005]

Some of the proof rules:

$$\frac{P \vdash Q}{P * F \vdash Q * F}$$

$$\frac{x \neq y * x \mapsto \_ * y \mapsto \_ * P \vdash Q}{x \mapsto \_ * y \mapsto \_ * P \vdash Q}$$

$$\frac{x = y * P \vdash Q \qquad x \neq y * z \neq y * x \mapsto z * z \mapsto y * P \vdash Q}{\mathbf{lseg}(x, y) * P \vdash Q} \quad z \text{ fresh}$$

$$\frac{P \vdash Q}{P \vdash \mathbf{lseg}(x, x) * Q}$$

$$\frac{x \neq z * P \vdash \mathbf{lseg}(y, z) * Q}{x \neq z * x \mapsto y * P \vdash \mathbf{lseg}(x, z) * Q}$$

Procedure Calls and Frame Inference

$$\frac{P \vdash P'[\mathbf{a}/\mathbf{x}] * F \qquad \{ P' \}\ p(\mathbf{x})\ \{ Q' \}}{\{ P \}\ \texttt{call}\ p(\mathbf{a})\ \{ Q'[\mathbf{a}/\mathbf{x}] * F \}}$$

Procedure Calls and Frame Inference

Frame inference

$$\frac{P \vdash P'[\mathbf{a}/\mathbf{x}] * F \qquad \{\, P' \,\}\ p(\mathbf{x})\ \{\, Q' \,\}}{\{\, P \,\}\ \mathbf{call}\ p(\mathbf{a})\ \{\, Q'[\mathbf{a}/\mathbf{x}] * F \}}$$

Frame Inference: Example

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null}) * F_{?}$$

Frame Inference: Example

$$\mathbf{lseg}(x, t) * t \mapsto null * \mathbf{lseg}(y, null) \vdash_{?} \mathbf{lseg}(x, null)$$

$$\mathbf{lseg}(x, t) * t \mapsto null * \mathbf{lseg}(y, null) \vdash_{?} \mathbf{lseg}(x, null) * F_{?}$$

Frame Inference: Example

$$t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(t, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null}) * F_{?}$$

Frame Inference: Example

$$\textbf{lseg}(y, \text{null}) \nvdash \text{emp}$$

$$t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(t, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null}) * F_{?}$$

Frame Inference: Example

Proof failed!

$$\mathbf{lseg}(y, \text{null}) \nvdash \text{emp}$$

$$t \mapsto \text{null} * \mathbf{lseg}(y, \text{null}) \vdash_{?} \mathbf{lseg}(t, \text{null})$$

$$\mathbf{lseg}(x, t) * t \mapsto \text{null} * \mathbf{lseg}(y, \text{null}) \vdash_{?} \mathbf{lseg}(x, \text{null})$$

$$\mathbf{lseg}(x, t) * t \mapsto \text{null} * \mathbf{lseg}(y, \text{null}) \vdash_{?} \mathbf{lseg}(x, \text{null}) * F_{?}$$

Frame Inference: Example

Move residual LHS to RHS and propagate down.

$$\textbf{lseg}(y, \text{null}) \vdash \text{emp} * \textbf{lseg}(y, \text{null})$$

$$t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_? \textbf{lseg}(t, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_? \textbf{lseg}(x, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_? \textbf{lseg}(x, \text{null}) * F_?$$

Frame Inference: Example

Move residual LHS to RHS and propagate down.

$$\textbf{lseg}(y, \text{null}) \vdash \text{emp} * \textbf{lseg}(y, \text{null})$$

$$t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(t, \text{null}) * \textbf{lseg}(y, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_{?} \textbf{lseg}(x, \text{null}) * F_{?}$$

Frame Inference: Example

Move residual LHS to RHS and propagate down.

$$\mathbf{lseg}(y, null) \vdash emp * \mathbf{lseg}(y, null)$$

$$t \mapsto null * \mathbf{lseg}(y, null) \vdash_? \mathbf{lseg}(t, null) * \mathbf{lseg}(y, null)$$

$$\mathbf{lseg}(x, t) * t \mapsto null * \mathbf{lseg}(y, null) \vdash_? \mathbf{lseg}(x, null) * \mathbf{lseg}(y, null)$$

$$\mathbf{lseg}(x, t) * t \mapsto null * \mathbf{lseg}(y, null) \vdash_? \mathbf{lseg}(x, null) * F_?$$

Frame Inference: Example

Move residual LHS to RHS and propagate down.

$$\textbf{lseg}(y, \text{null}) \vdash \text{emp} * \textbf{lseg}(y, \text{null})$$

$$t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_? \textbf{lseg}(t, \text{null}) * \textbf{lseg}(y, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_? \textbf{lseg}(x, \text{null}) * \textbf{lseg}(y, \text{null})$$

$$\textbf{lseg}(x, t) * t \mapsto \text{null} * \textbf{lseg}(y, \text{null}) \vdash_? \textbf{lseg}(x, \text{null}) * F_?$$

$$F_? \equiv \textbf{lseg}(y, \text{null})$$

Specification of `delete`

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
{ emp }
```

Verifying `delete`

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
{ emp }
```

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {  ⟷ {lseg(x,null) ∧ x≠null}
    var y := [x];
    delete(y);
    free(x);
  }
}
{ emp }
```

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {  ⟷ {lseg(x,null) ∧ x≠null}
    var y := [x];  ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
{ emp }
```

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];  ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
{ emp }
```

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];   ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
```

$x \mapsto y * \mathbf{lseg}(y,null) \wedge x \neq null \vdash \mathbf{lseg}(y, null)$

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];  ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
```

Frame inference:

$x \mapsto y * \mathbf{lseg}(y,null) \wedge x \neq null \vdash \mathbf{lseg}(y, null) * ?$

Verifying delete

{ **lseg**(x,null) }

```
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];   ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
```

Frame inference:

x ↦ y ∧ x≠null ⊢ emp * ?

x ↦ y * **lseg**(y,null) ∧ x≠null ⊢ **lseg**(y, null) * ?

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];   ⟷  {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
```

Frame inference:

$x \mapsto y \wedge x \neq null \vdash ?$

$x \mapsto y \wedge x \neq null \vdash emp * ?$

$x \mapsto y * lseg(y,null) \wedge x \neq null \vdash lseg(y, null) * ?$

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];   ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);
    free(x);
  }
}
```

Frame inference: ? = x ↦ y ∧ x≠null

x ↦ y ∧ x≠null ⊢ ?

x ↦ y ∧ x≠null ⊢ emp * ?

x ↦ y * lseg(y,null) ∧ x≠null ⊢ lseg(y, null) * ?

Verifying delete

{ lseg(x,null) }

```
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];     ⟷ {x ↦ y * lseg(y,null)…}
    delete(y);        ⟷ {emp * x ↦ y ∧ x≠null}
    free(x);
  }
}
```

Frame inference: $? = x \mapsto y \land x \neq null$

$x \mapsto y \land x \neq null \vdash ?$

$x \mapsto y \land x \neq null \vdash emp * ?$

$x \mapsto y * lseg(y,null) \land x \neq null \vdash lseg(y, null) * ?$

Verifying delete

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];
    delete(y);  ⟷  {emp * x ↦ y ∧ x≠null}
    free(x);
  }
}
{ emp }
```

Verifying delete

{ **lseg**(x,null) }

```
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];
    delete(y);  ⟷ {emp ∗ x ↦ y ∧ x≠null}
    free(x);  ⟷ {emp ∗ emp ∧ x≠null}
  }
}
```

{ emp }

Verifying `delete`

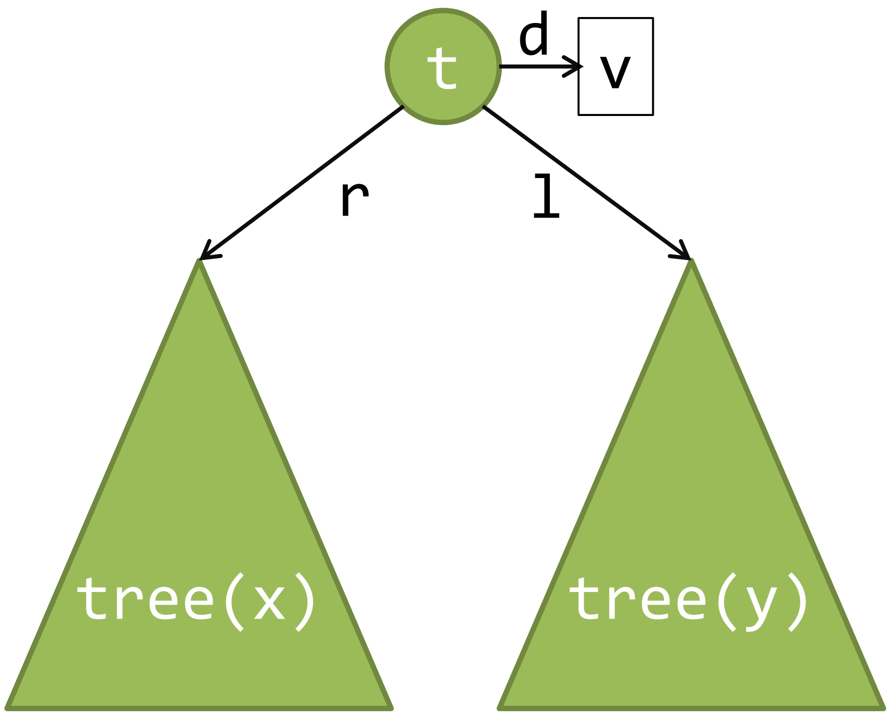
```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];
    delete(y);
    free(x);  ⟷  {emp * emp ∧ x≠null}
  }
}
{ emp }
```

Verifying `delete`

```
{ lseg(x,null) }
procedure delete(x: Node)
{
  if (x ≠ null) {
    var y := [x];
    delete(y);
    free(x);
  }
}
{ emp }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```



Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}

{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

```
                 tree(t)
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
                          tree(t) ∧ t != null
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
                    tree(t) ∧ t != null
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

```
tree(t) ∧ t != null
```

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

t ↦ (d:w, r:x, l:y) * tree(x) * tree(y)

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

t↦(d:w,r:x,l:y) * tree(x) * tree(y)

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

t ↦ (d:w, r:x, l:y) * tree(x) * tree(y)

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

```
t ↦ (d:w, r:x, l:y) * tree(x) * tree(y)
```

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

t ↦ (d:w, r:x, l:y) * tree(x) * tree(y)

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

t ↦ (d:w, r:x, l:y) * tree(x) * tree(y)

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
                          t↦(d:w,r:x,l:y) * tree(x) * tree(y)
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

```
tree(t)
```

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Binary Search Trees

```
predicate tree(t: Node) {
  t == null ∧ emp ∨
  ∃ v, x, y ::
    t ↦ (d: v, r: x, l: y) * tree(x) * tree(y)
}
```

tree(t)

```
{ tree(t) }
procedure search(t: Node, v: Int): Bool {
  if (t == null) return false;
  else if (t.d < v)
    return search(t.l, v);
  else if (t.d > v)
    return search(t.r, v);
  else return true;
}
{ tree(t) }
```

Permission Logics vs. Classical FOL

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

Permission Logics vs. Classical FOL

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

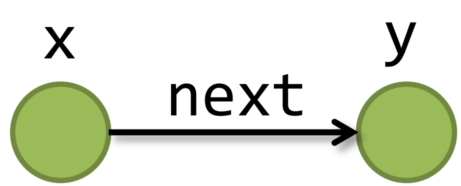
- Strong theoretical guarantees:
  sound, **complete**, **tractable complexity (NP)**
- Mixed specs: escape hatch when SL is not suitable.

Implicit Dynamic Frames

Implicit Dynamic Frames by Example

- Pure assertions

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



Stack

x	10
y	42
…	

Heap

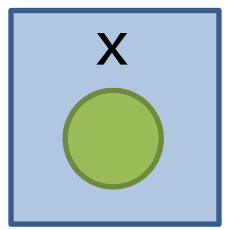
10	42
…	
42	?

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

Implicit Dynamic Frames by Example

- Permission predicates

acc(x)

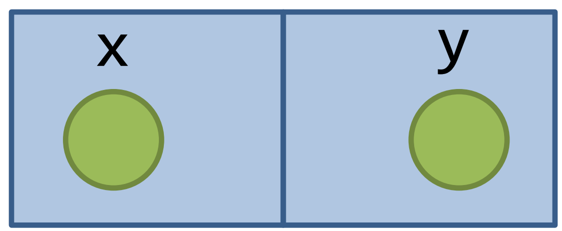


Expresses permission to access (i.e. read/write/deallocate) heap location x.
Assertions describe the program state **and** a set of locations that are allowed to be accessed.

Implicit Dynamic Frames by Example

- Separating conjunction

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

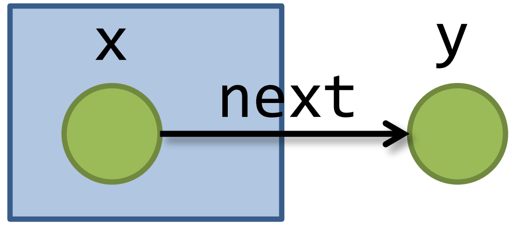


Yields union of permission sets of subformulas.
Permission sets of subformulas must be disjoint.

Implicit Dynamic Frames by Example

- Separating conjunction

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

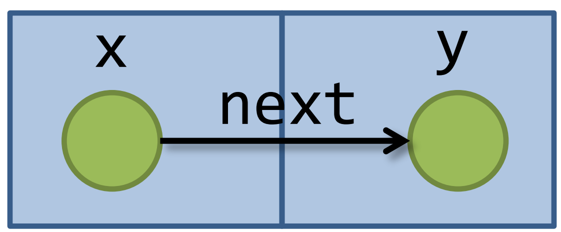


Pure assertions yield no permissions.

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
```

?

Implicit Dynamic Frames by Example

- Separating conjunction

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

unsatisfiable

Implicit Dynamic Frames by Example

- Classical conjunction

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

?

Implicit Dynamic Frames by Example

- Classical conjunction

```
acc(x) ∧ x.next == y
```

unsatisfiable

Implicit Dynamic Frames by Example

- Classical conjunction

```
acc(x) ∧ acc(y) * x.next == y
```

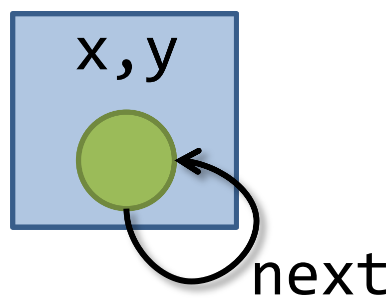
?

Convention: ∧ has higher precedence than *

Implicit Dynamic Frames by Example

- Classical conjunction

```
acc(x) ∧ acc(y) * x.next == y
```



Convention: ∧ has higher precedence than *

Syntactic Short-hands

- Empty heap:

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

- Points-to predicates:

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

Implicit Dynamic Frames: Assertion Semantics

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

- $M,D \models P \quad \Leftrightarrow D = \emptyset$ and $M \models P \quad$ if P is pure
- $M,D \models$ **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

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}
  ```
  ```

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}
  ```
  ```

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}

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}

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}
```

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}

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}
  ```
  
  ```

Note on Soundness of Frame Rule

- An assertion is *self-framing* if its truth value only depends on the heap locations it grants access to

- Example:
  `acc``(x) * y.next == x` and **`acc`**`(y.next)`
  are not self-framing

Note on Soundness of Frame Rule

- An assertion is *self-framing* if its truth value only depends on the heap locations it grants access to
- Example:
  acc(x) ∗ y.next == x and **acc**(y.next)
  are not self-framing

```
⊢ {acc(y)} y.next := z; {acc(y) * y.next == z}
⊬ {acc(y) * acc(y.next)}
    y.next := z;
   {acc(y) * y.next == z * acc(y.next)}
```

Ouch!

Note on Soundness of Frame Rule

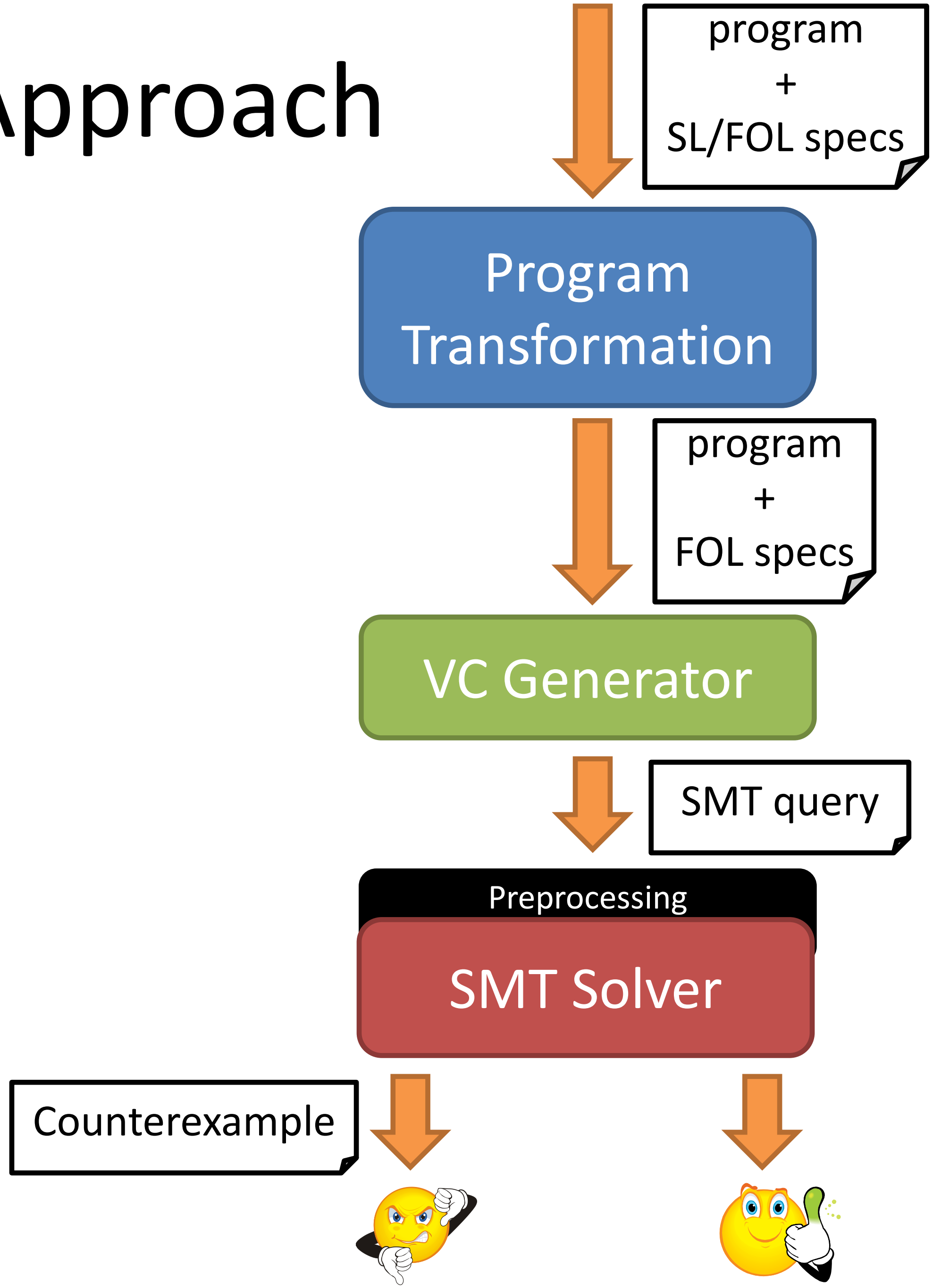
- An assertion is *self-framing* if its truth value only depends on the heap locations it grants access to

Self-framing can be enforced syntactically (separation logic) or semantically (implicit dynamic frames).

```
⊢ {acc(y)} y.next := z; {acc(y) * y.next == z}
⊬ {acc(y) * acc(y.next)}
     y.next := z;
    {acc(y) * y.next == z * acc(y.next)}
```

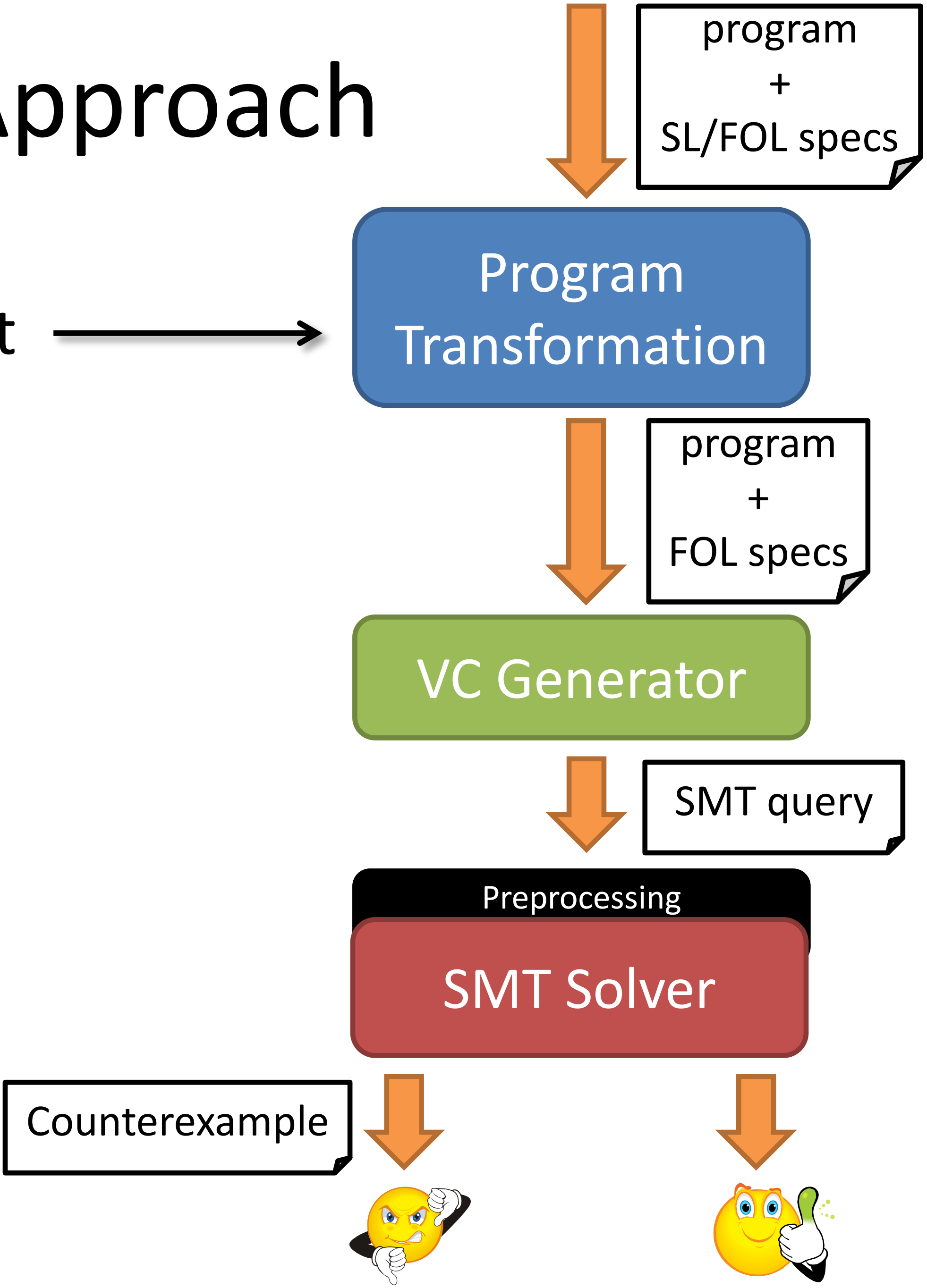
Ouch!

GRASShopper Approach



GRASShopper Approach

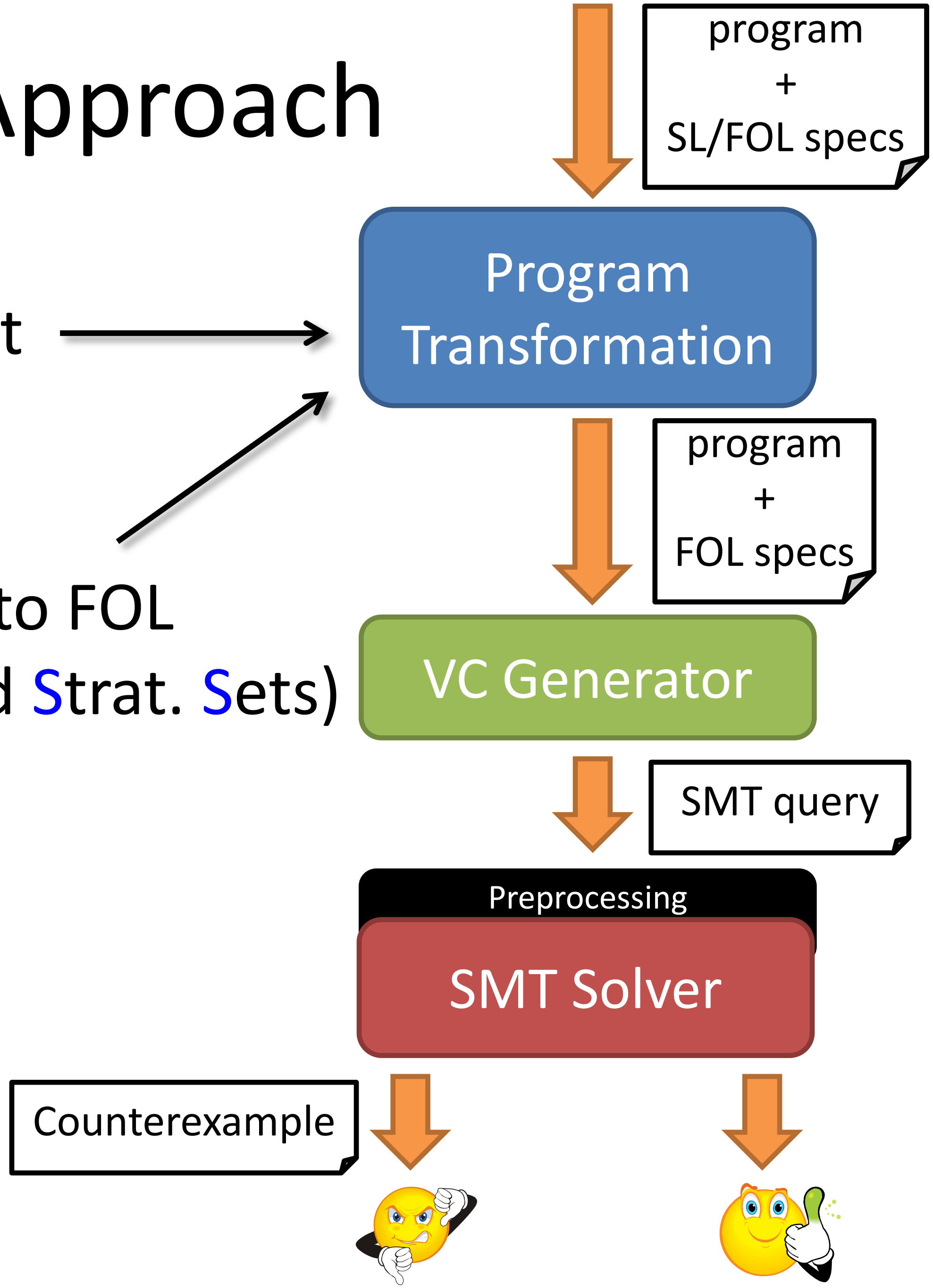
1. Make frame rule explicit
   [TACAS'14]



GRASShopper Approach

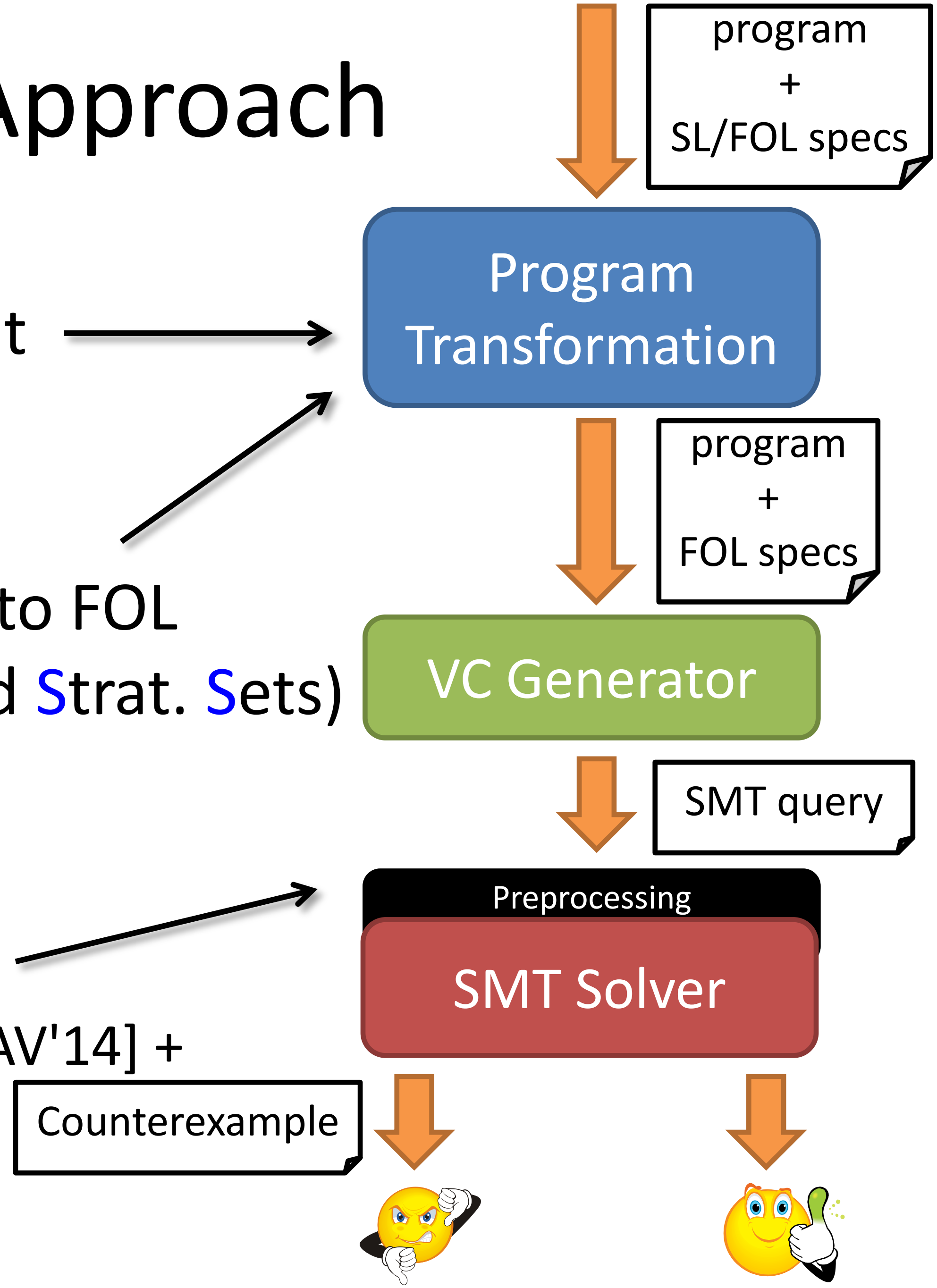
1. Make frame rule explicit
   [TACAS’14]

2. Translate SL assertions to FOL
   (Graph Reachability and Strat. Sets)
   [CAV’13]



GRASShopper 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’14] + [CAV'14] +
   [CAV'15]

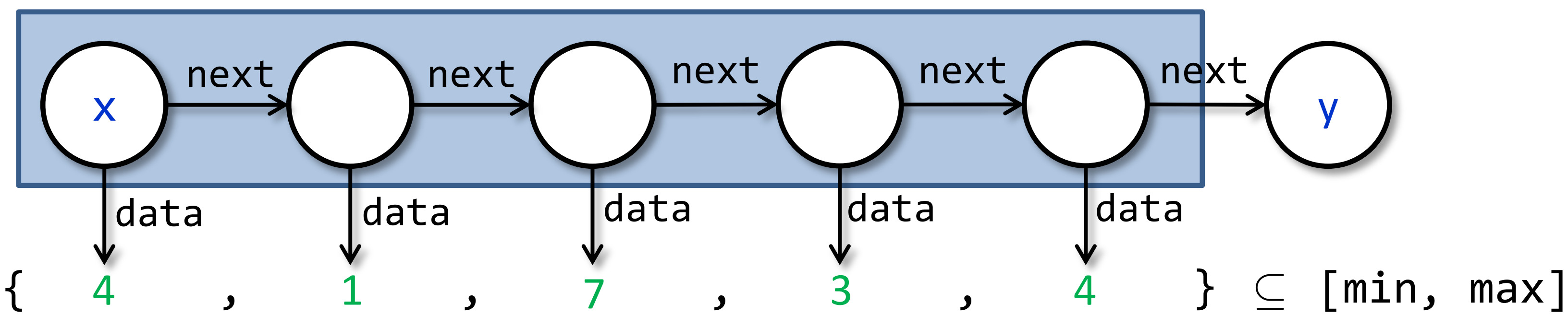


Reasoning about Heap and Data

Inductive Predicates with Data

- bounded list segment

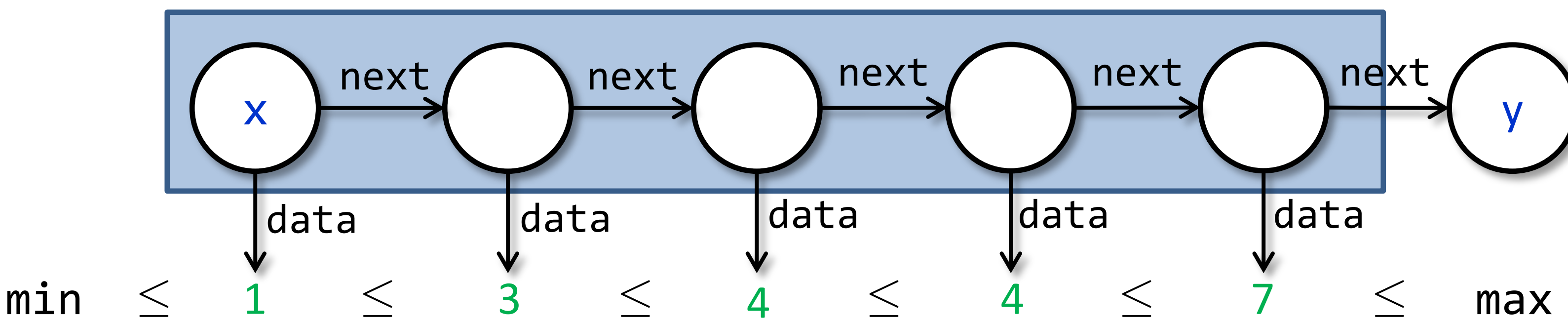
```
bnd_lseg(x, y, min, max) =
    x = y ∨
    x ≠ y * acc(x) * min ≤ x.data ≤ max *
    bnd_lseg(x.next, y, min, max)
```



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)
```

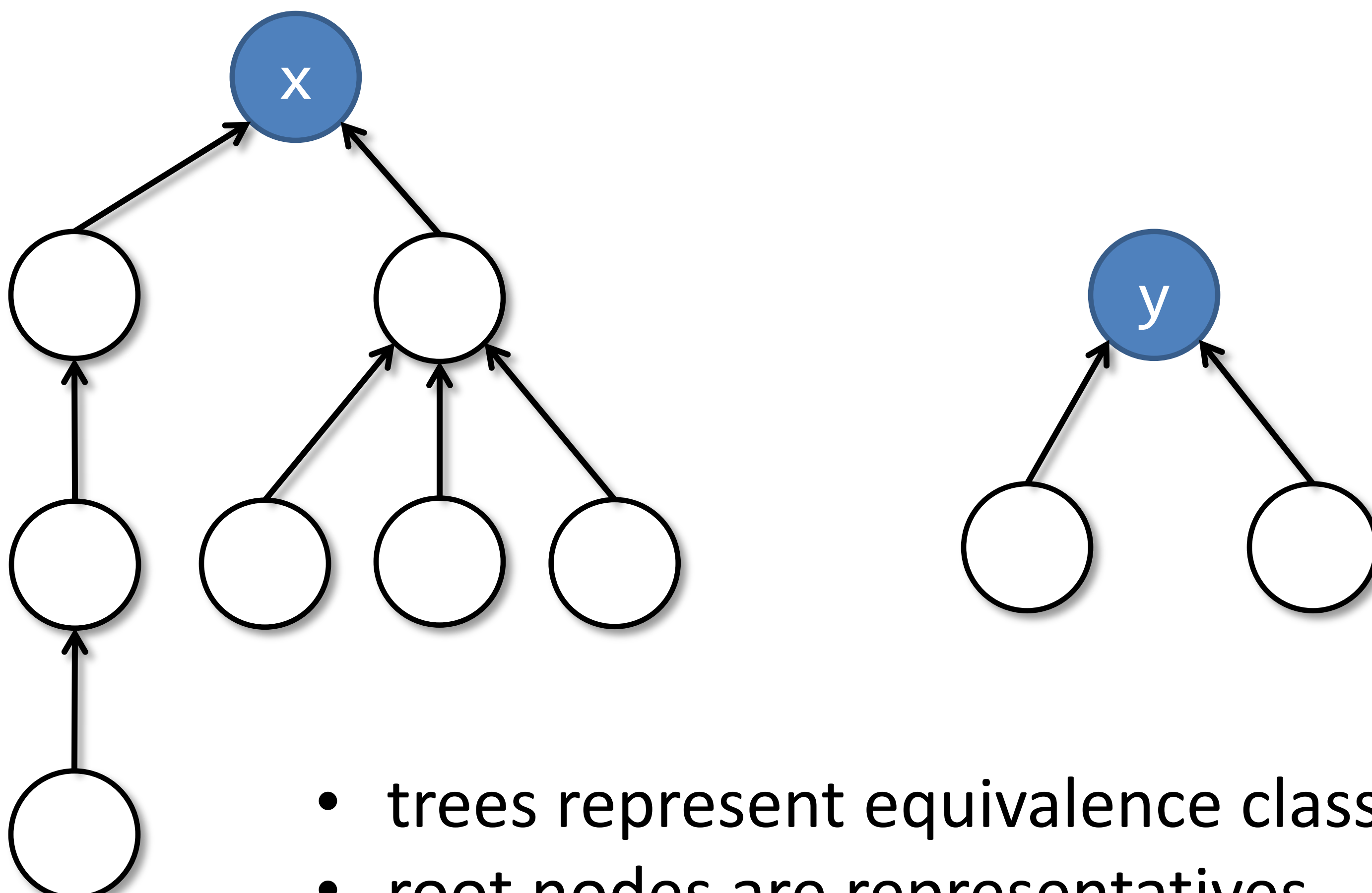


Example: Quicksort

```
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;
}
```

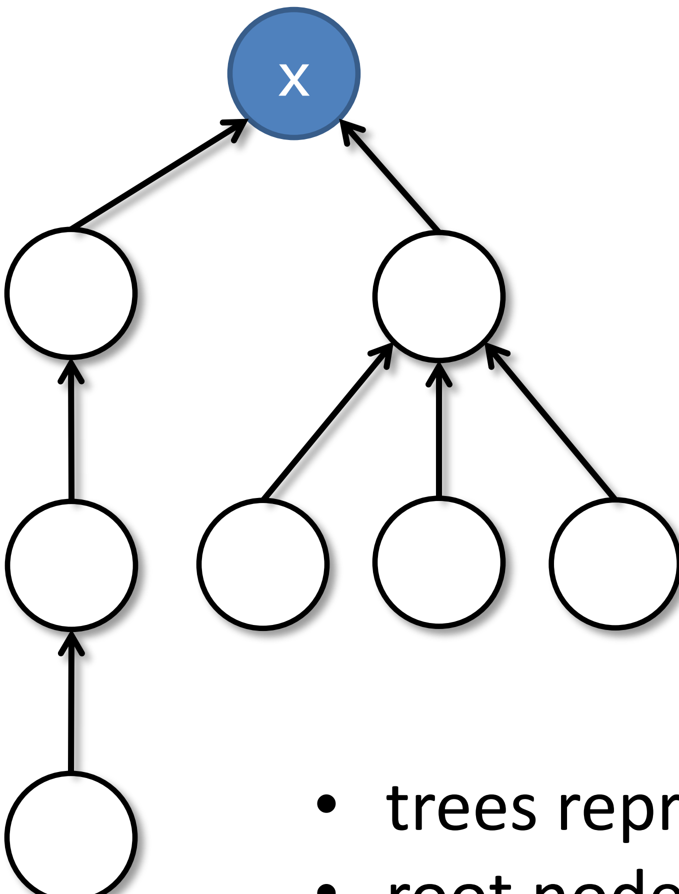
Mixed Specifications

Example: Union/Find Data Structure

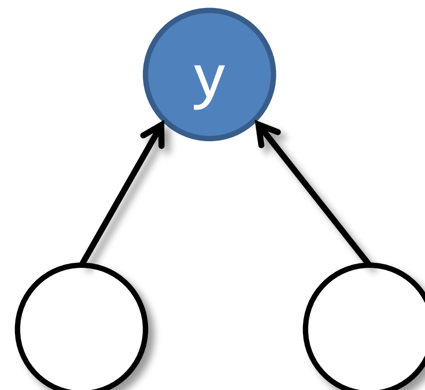


- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure

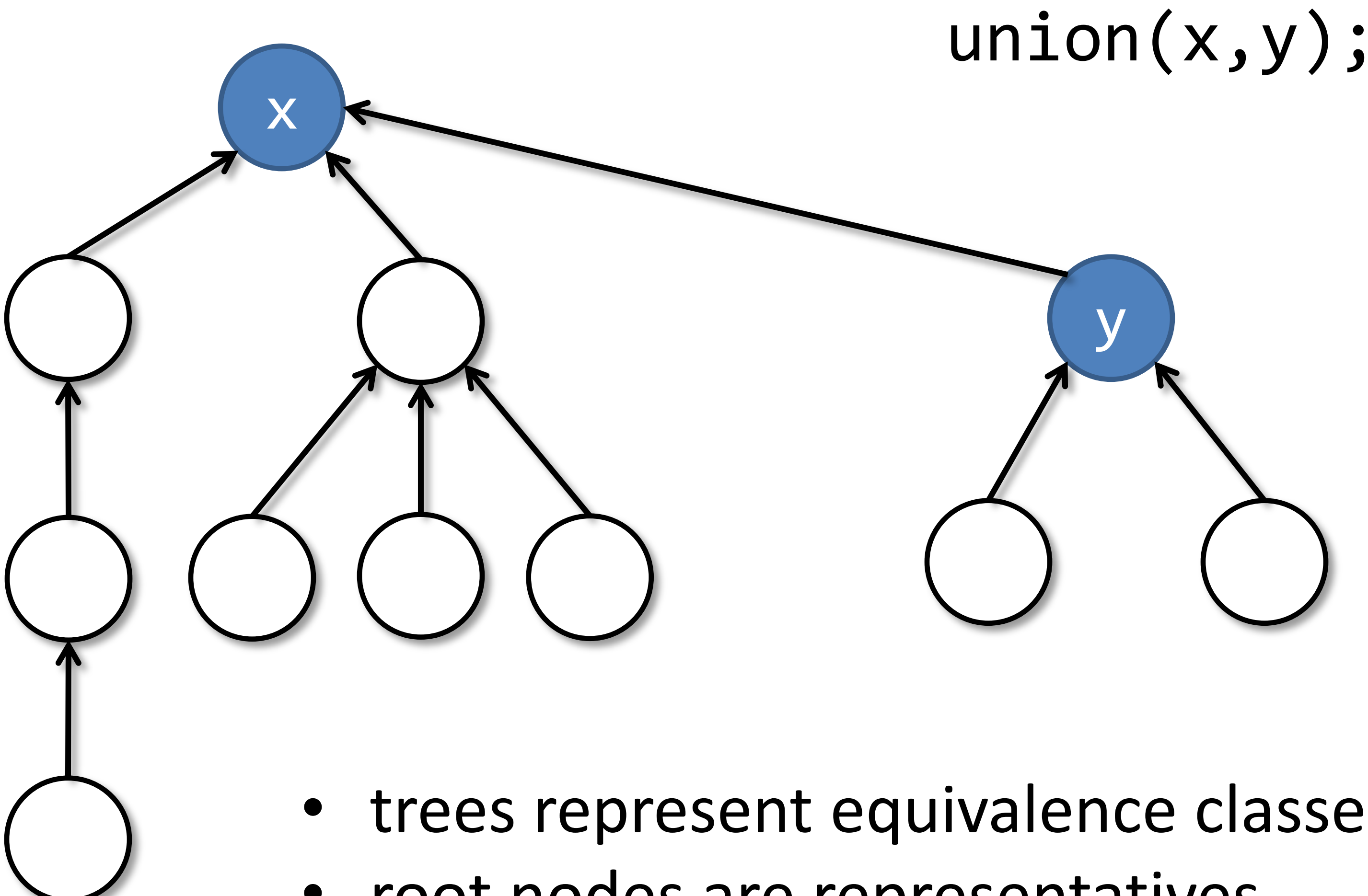



```
union(x,y);
```



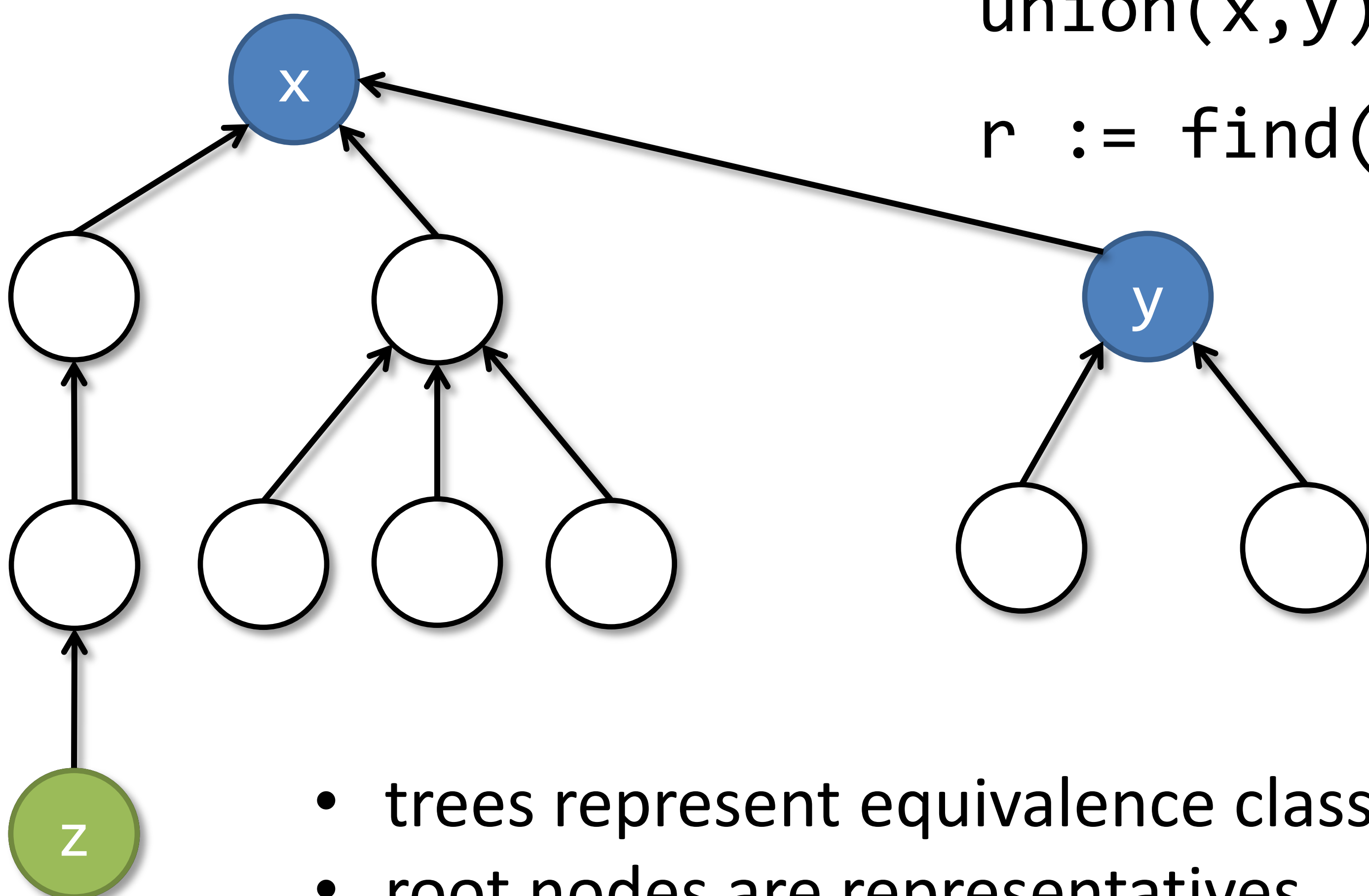
- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure



- trees represent equivalence classes
- root nodes are representatives

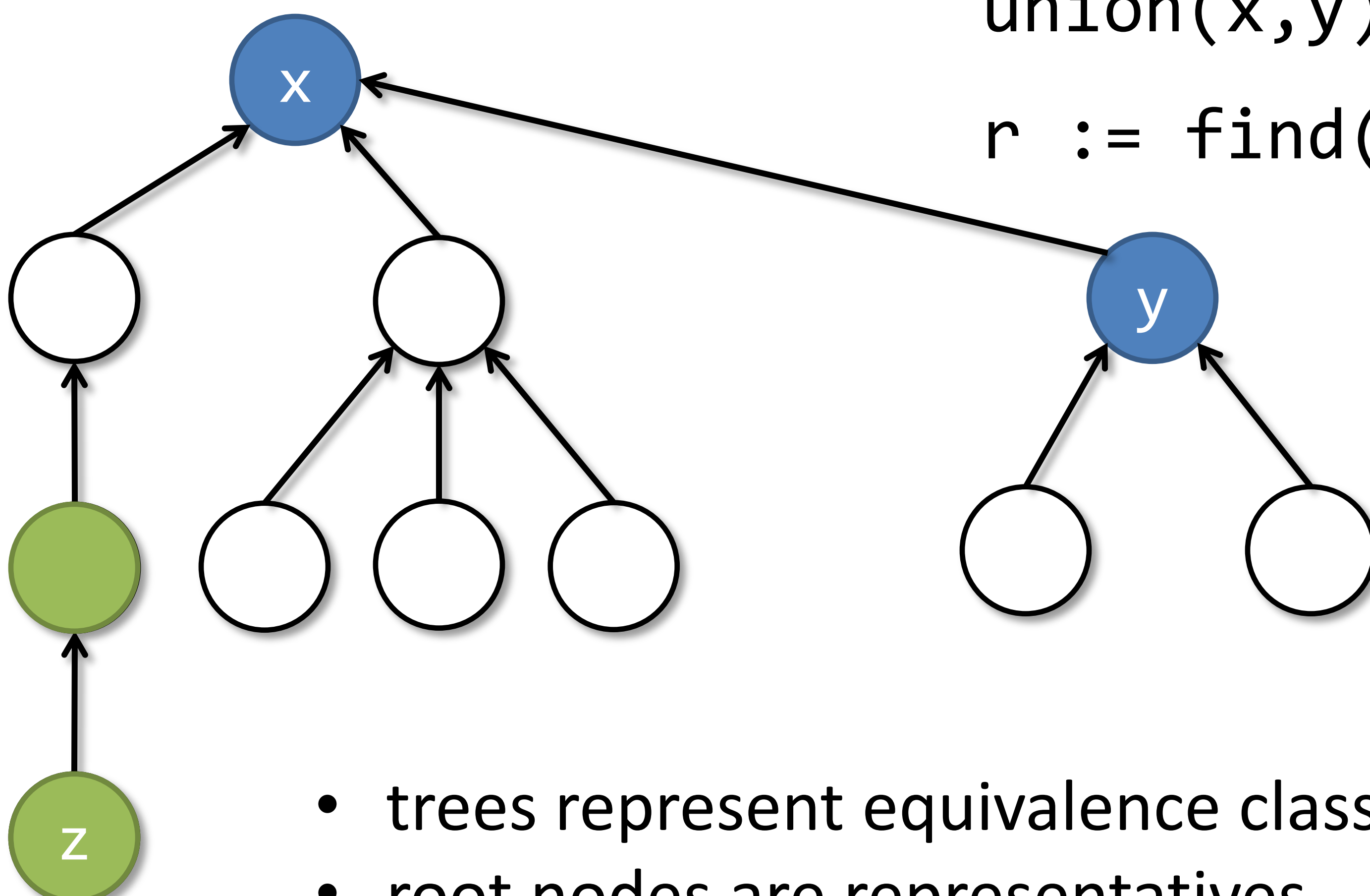
Example: Union/Find Data Structure




```
union(x,y);
r := find(z);
```

- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure

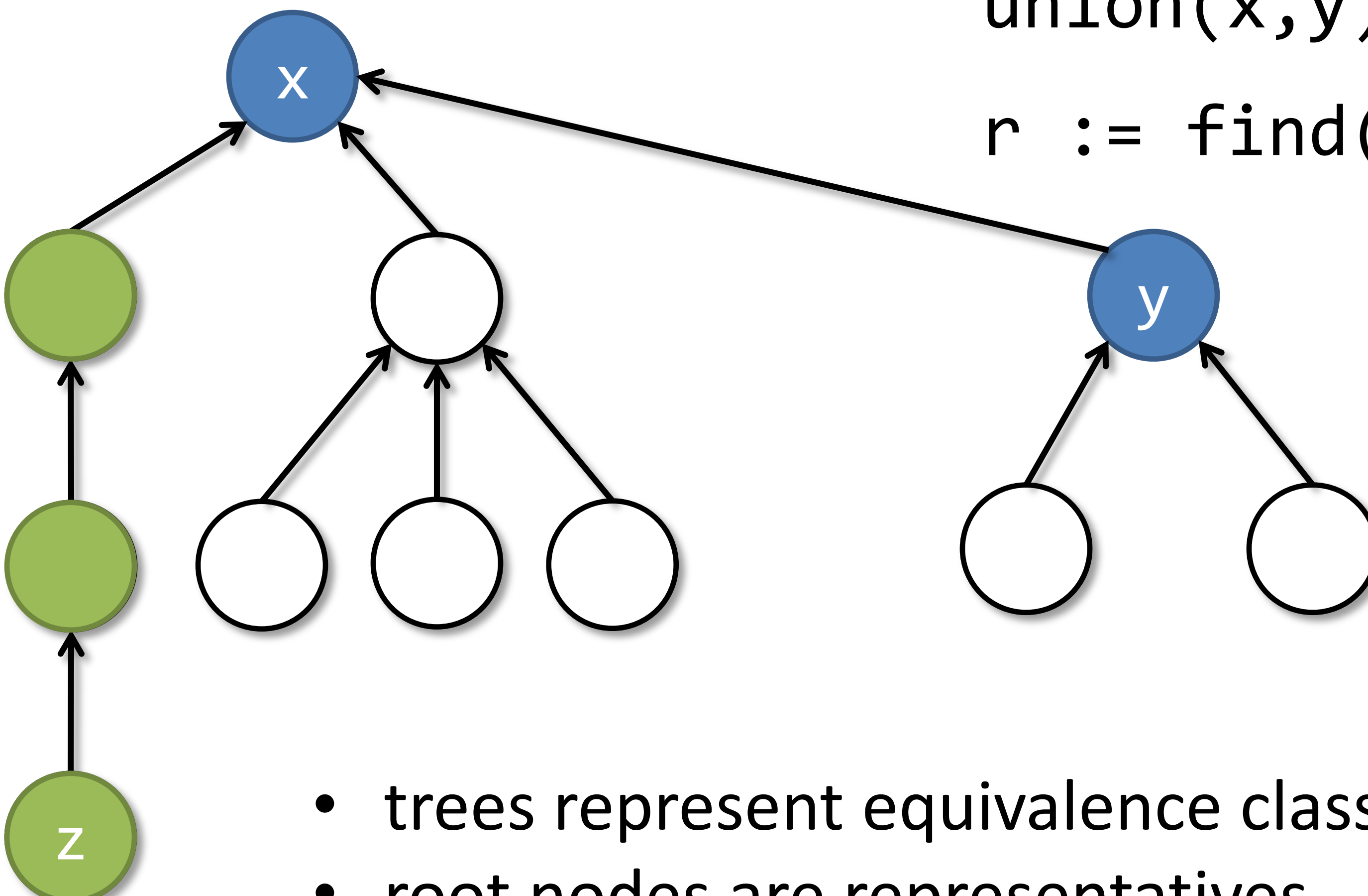



```
union(x,y);

r := find(z);
```

- trees represent equivalence classes
- root nodes are representatives

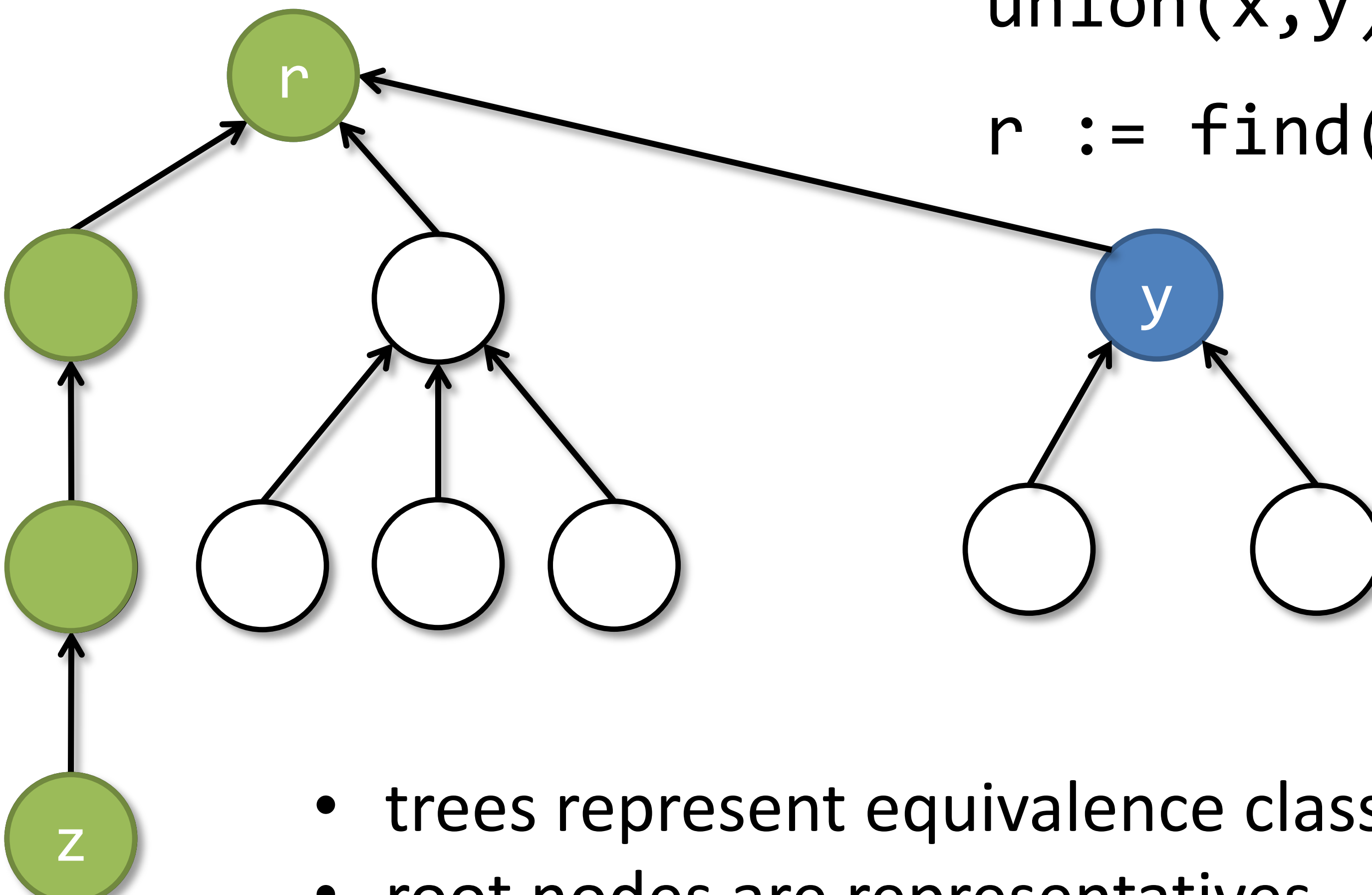
Example: Union/Find Data Structure




```
union(x,y);
r := find(z);
```

- trees represent equivalence classes
- root nodes are representatives

Example: Union/Find Data Structure



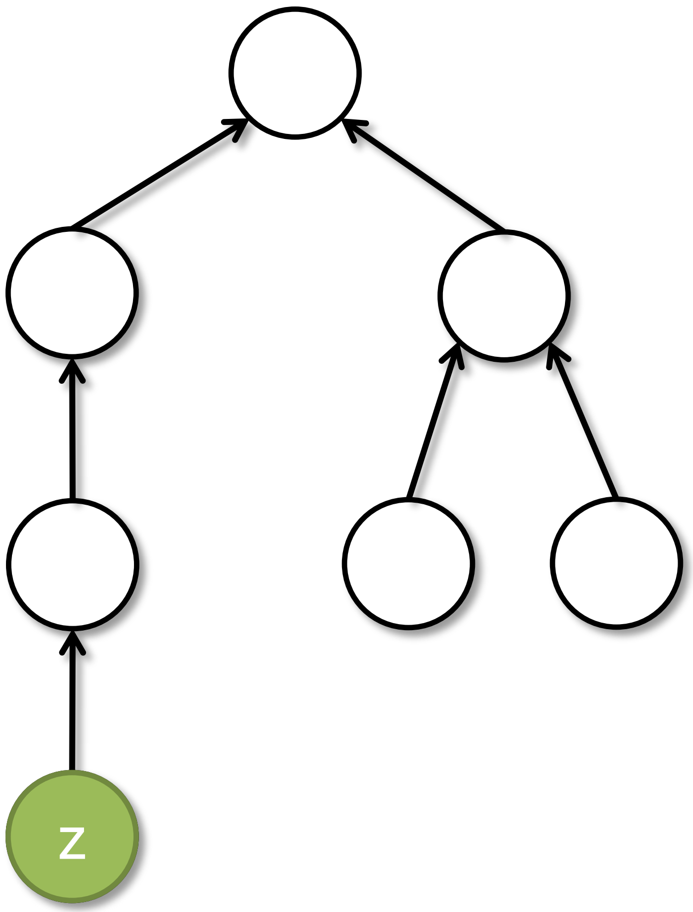

```
union(x,y);

r := find(z);
```

- trees represent equivalence classes
- root nodes are representatives

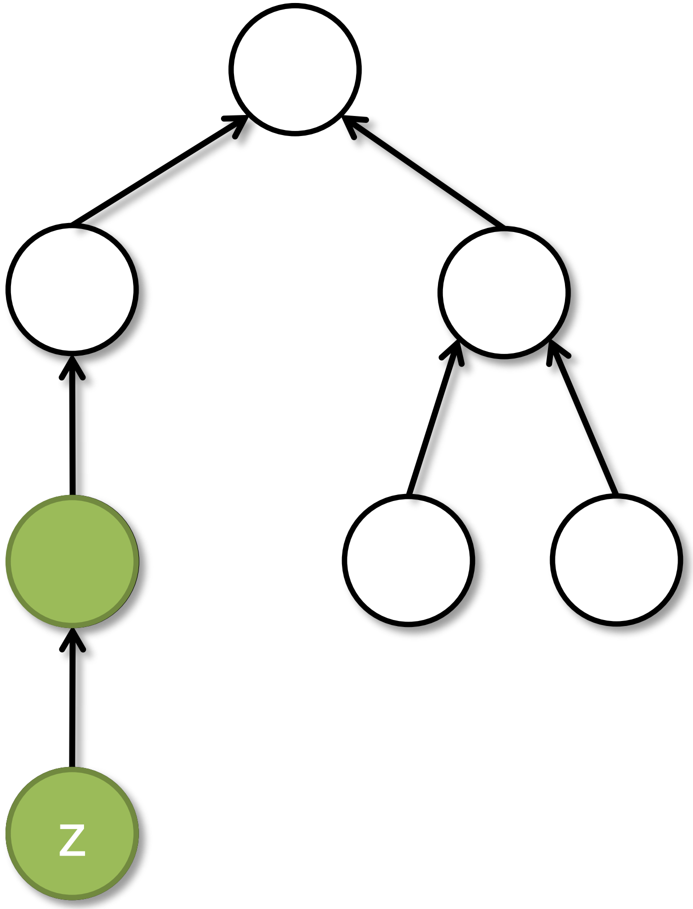
Find with Path Compression

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



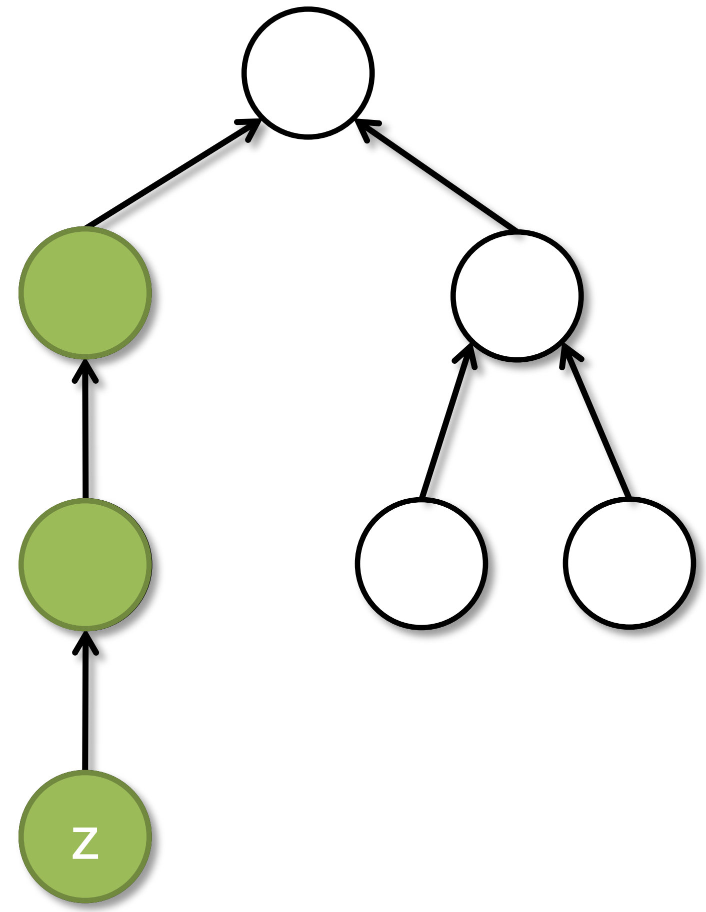
Find with Path Compression

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



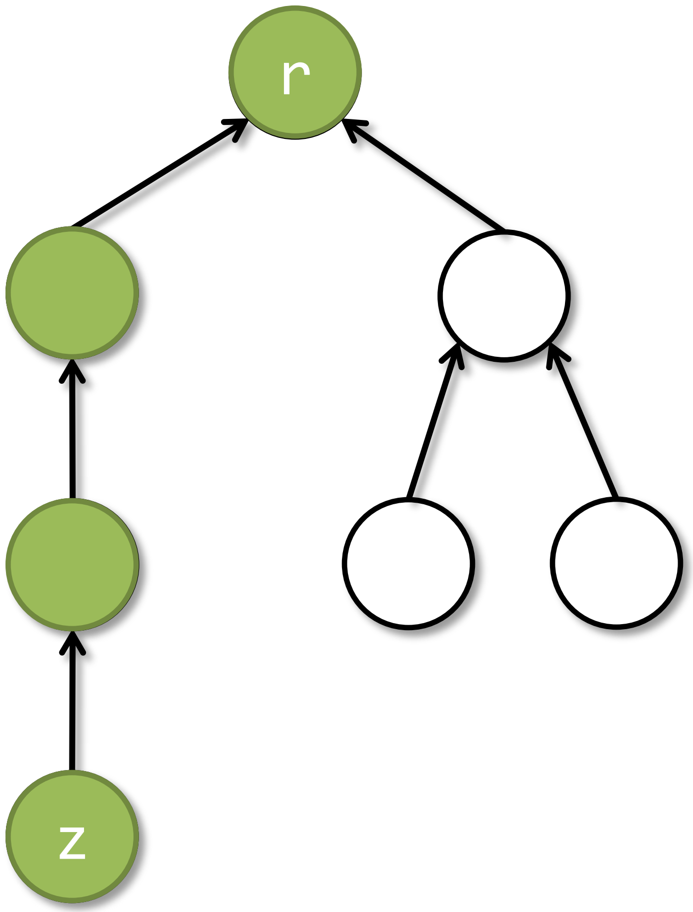
Find with Path Compression

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



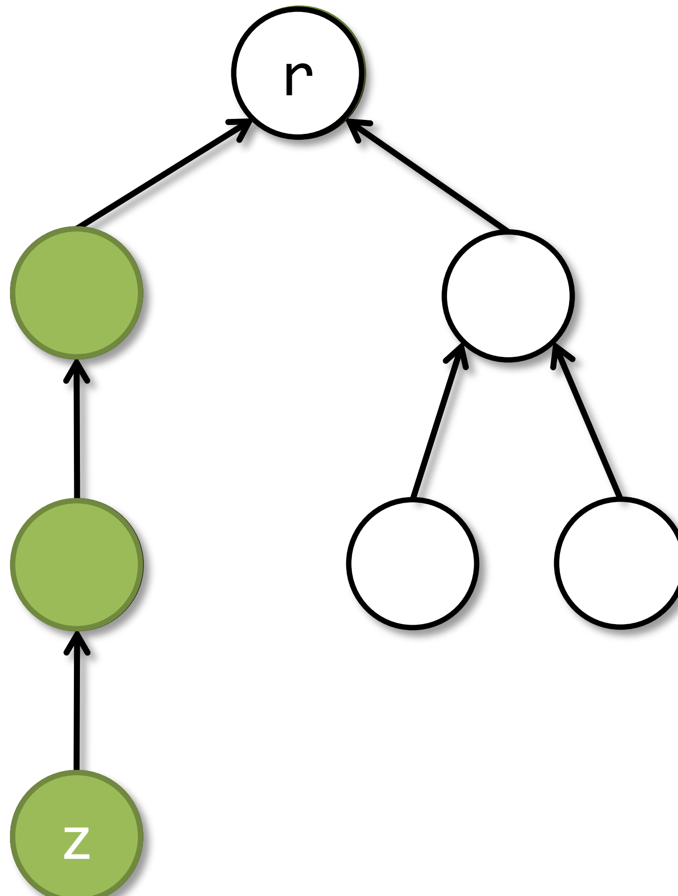
Find with Path Compression

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



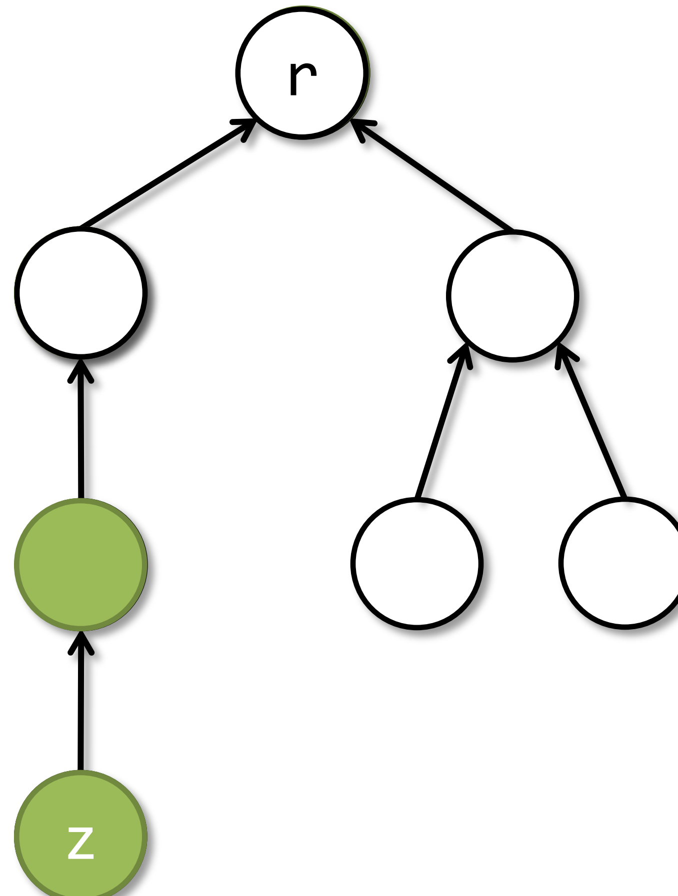
Find with Path Compression

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



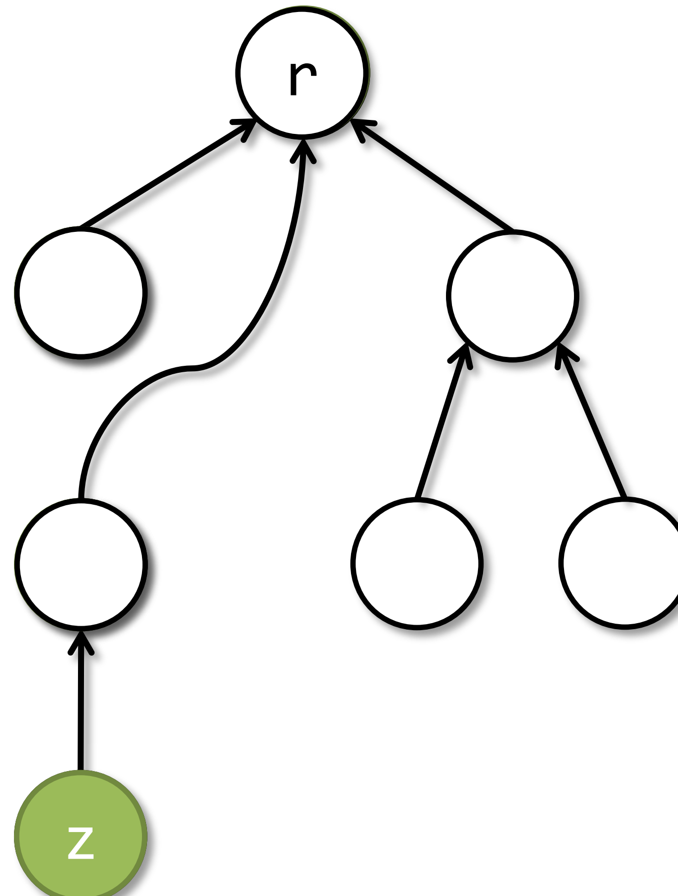
Find with Path Compression

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



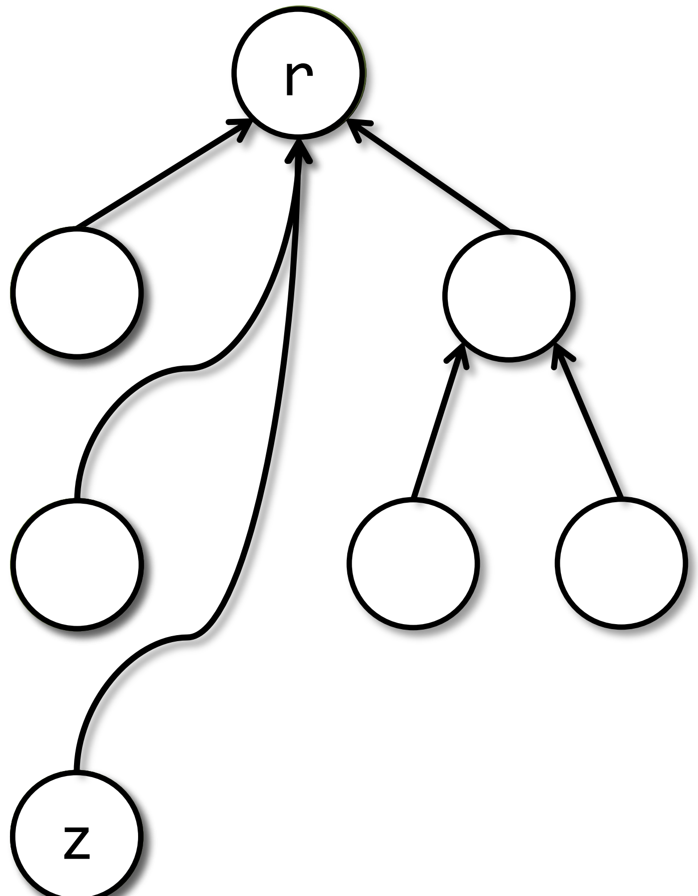
Find with Path Compression

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



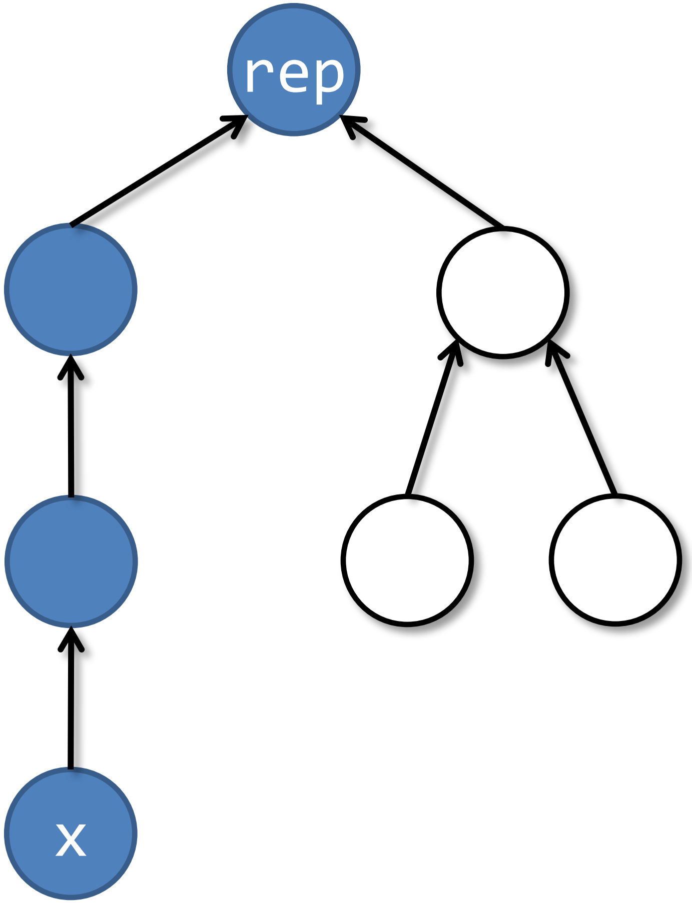
Find with Path Compression

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



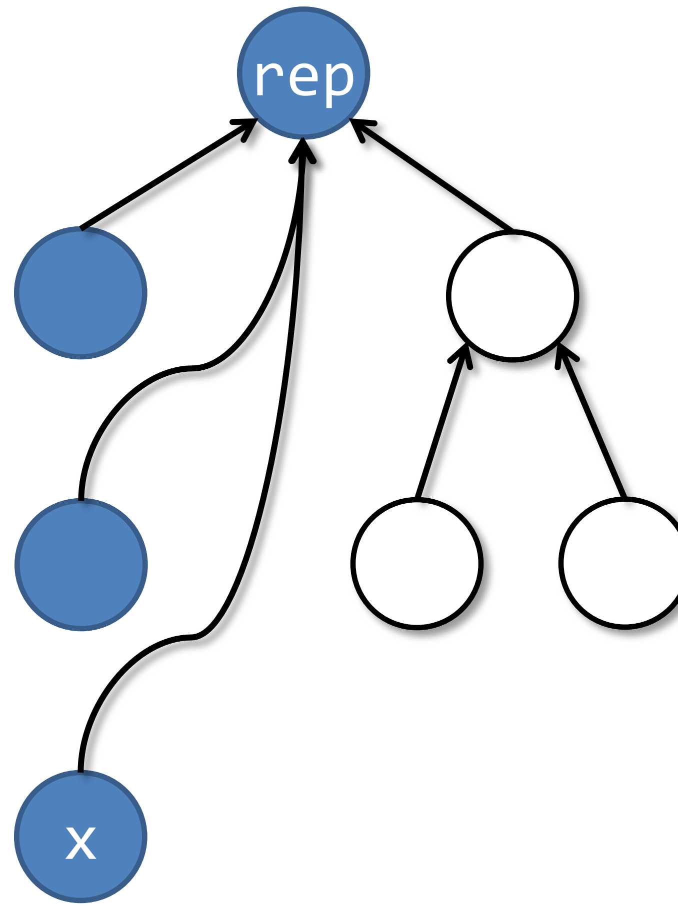
Find with SL Specification

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



Find with SL Specification

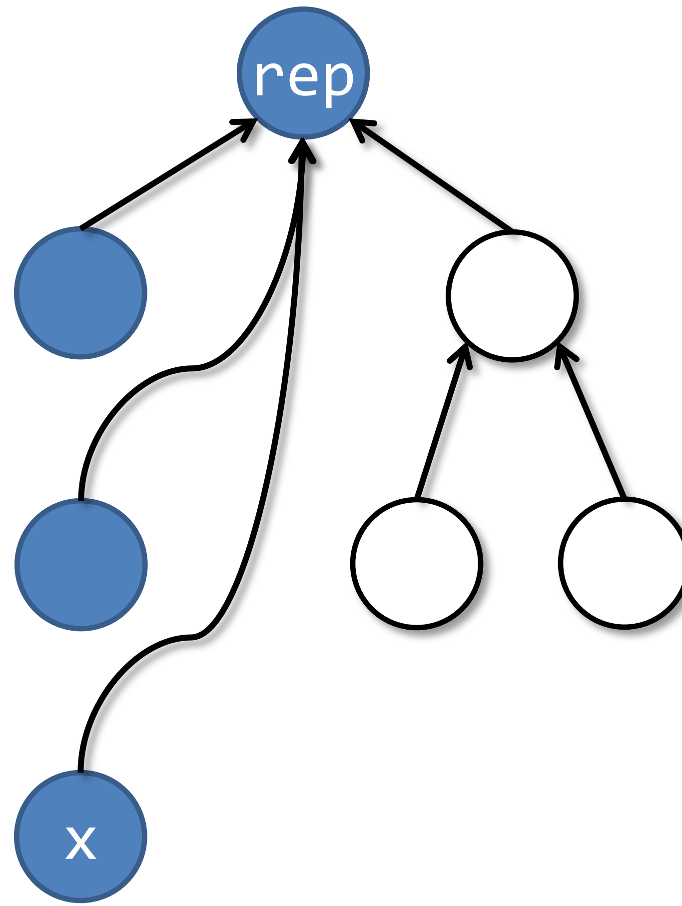
```
procedure find(x: Node, ghost rep: Node)
  returns (r: Node)
  requires rep.next ↦ null
  requires lseg(x, rep)
  ensures r == rep
  ensures rep.next ↦ null
  ensures ?
```



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



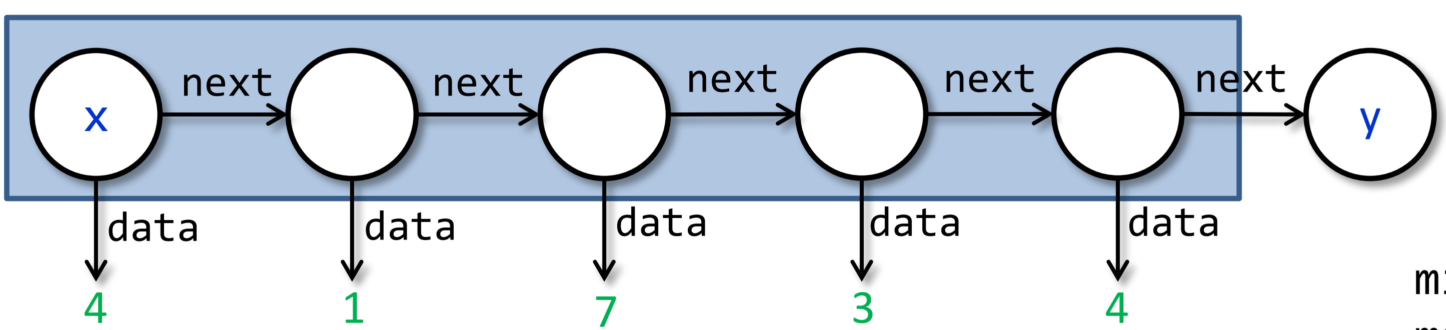
Completeness and Counterexamples

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

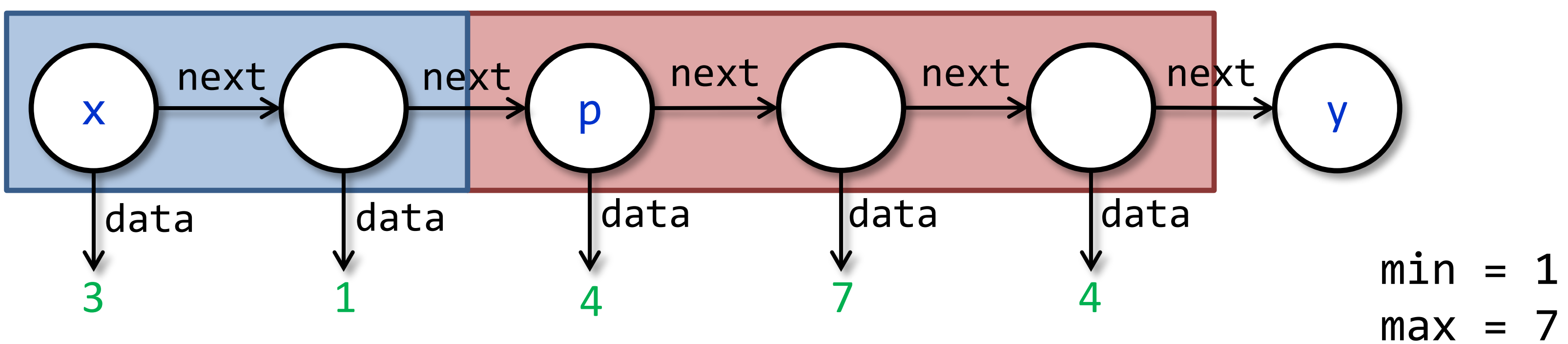
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```



min = 1
max = 7

Split with SL Specification

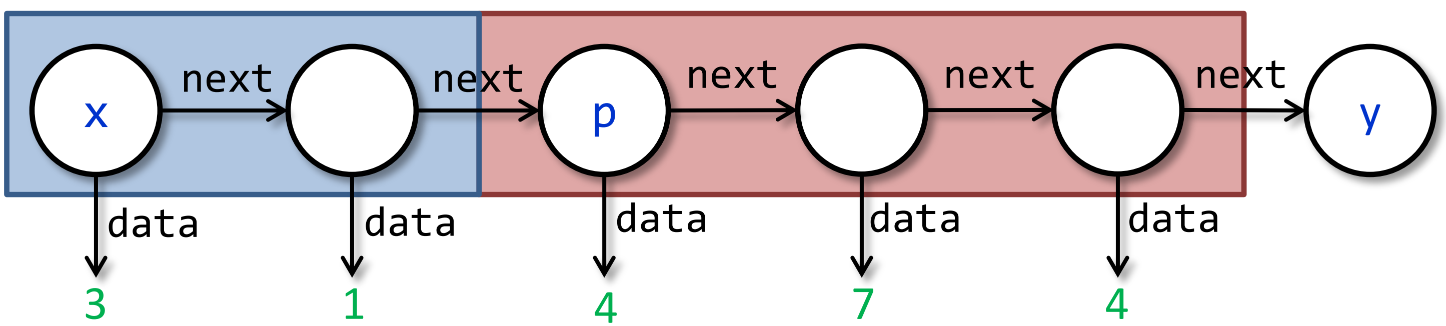
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```



Split with SL Specification

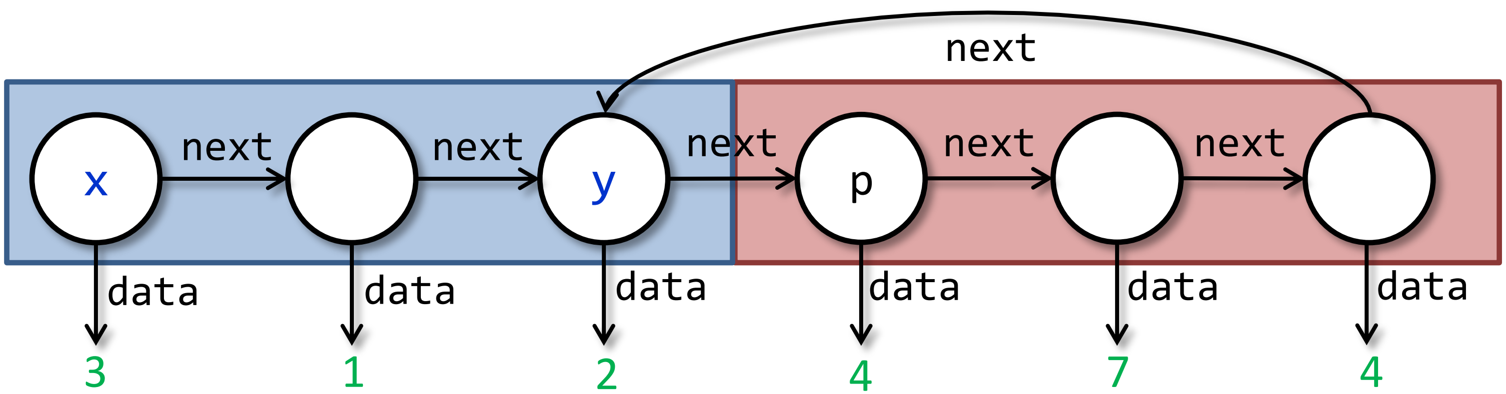
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max)
  ensures p ≠ y * min ≤ p.data ≤ max
```

free memory



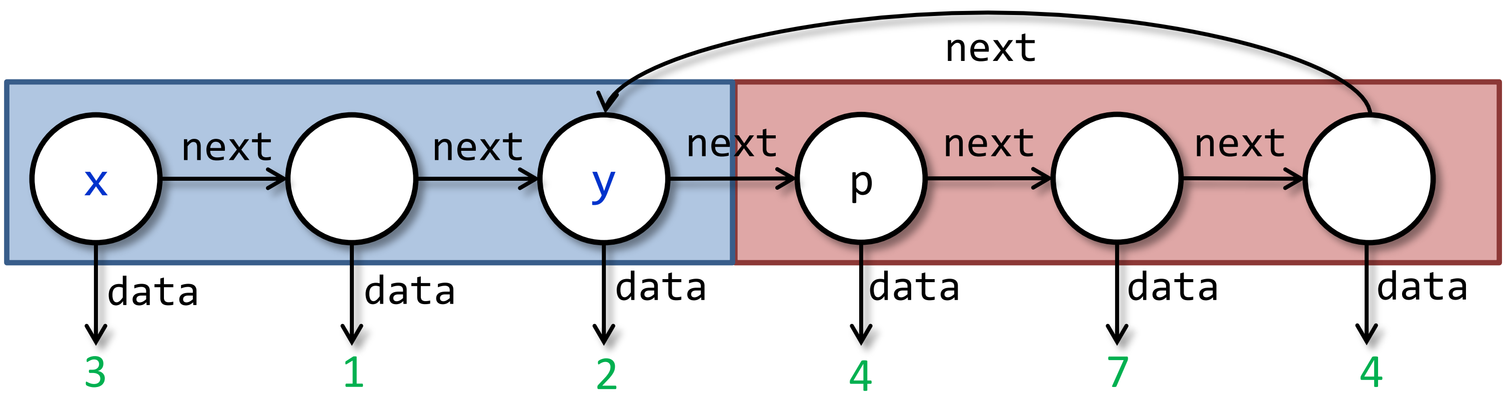
Counterexample for Quicksort Spec.

```
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
```



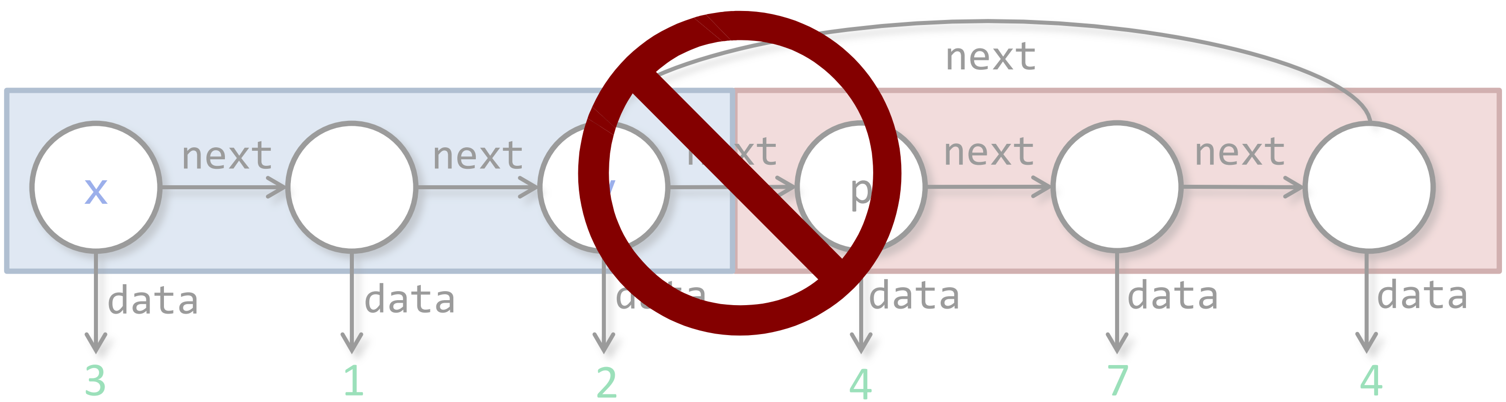
Split with Mixed Specification

```
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
```



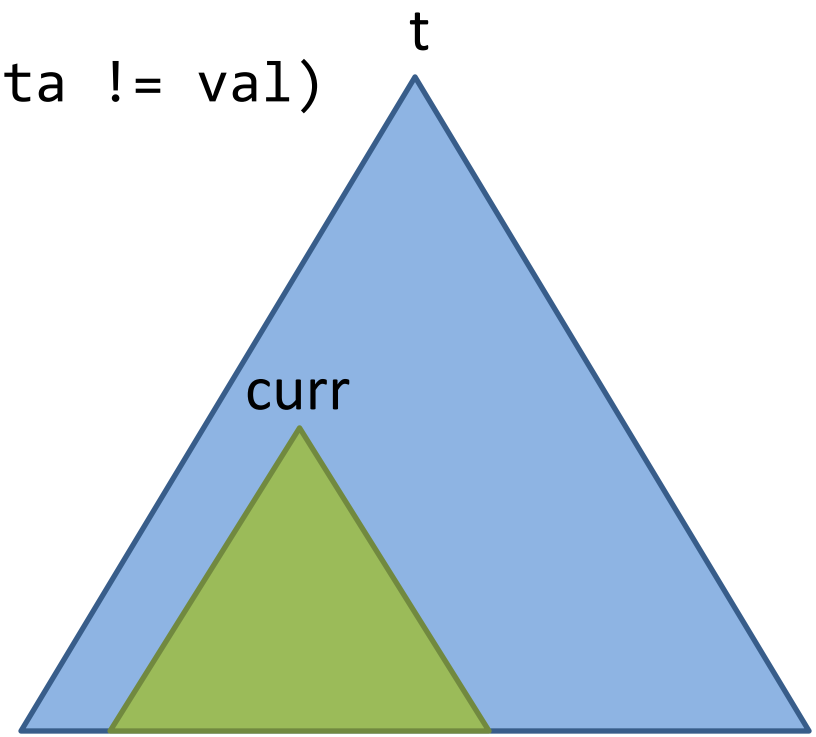
Split with Mixed Specification

```
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
```

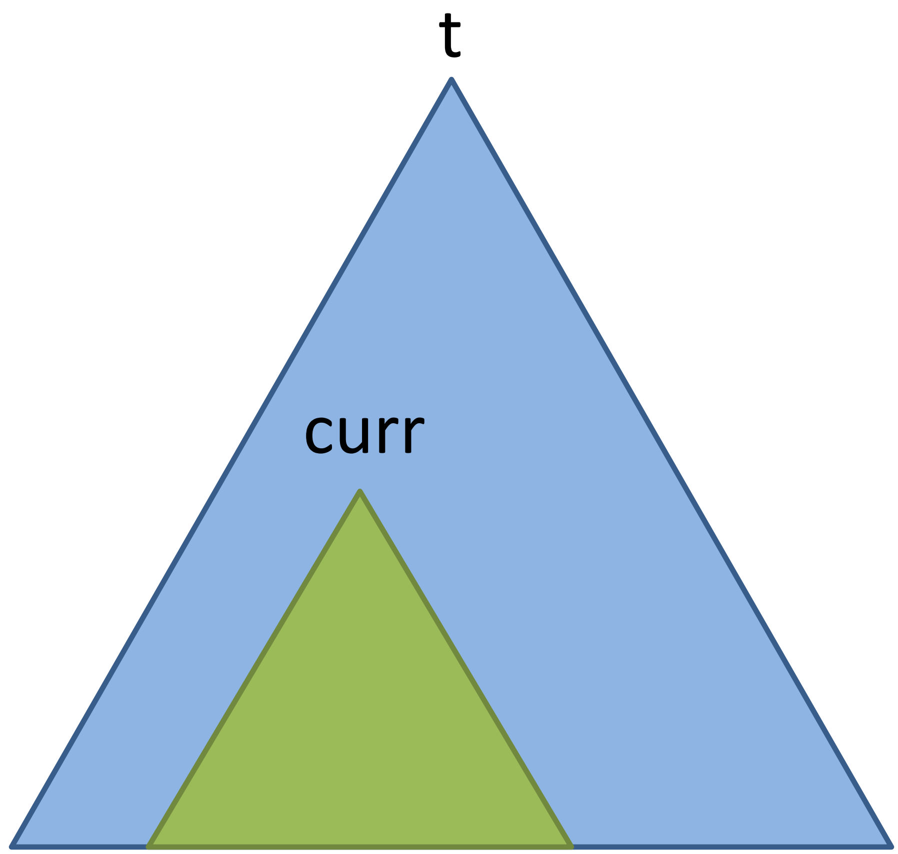


Tail-Recursive Tree Traversal

```
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;
}
```



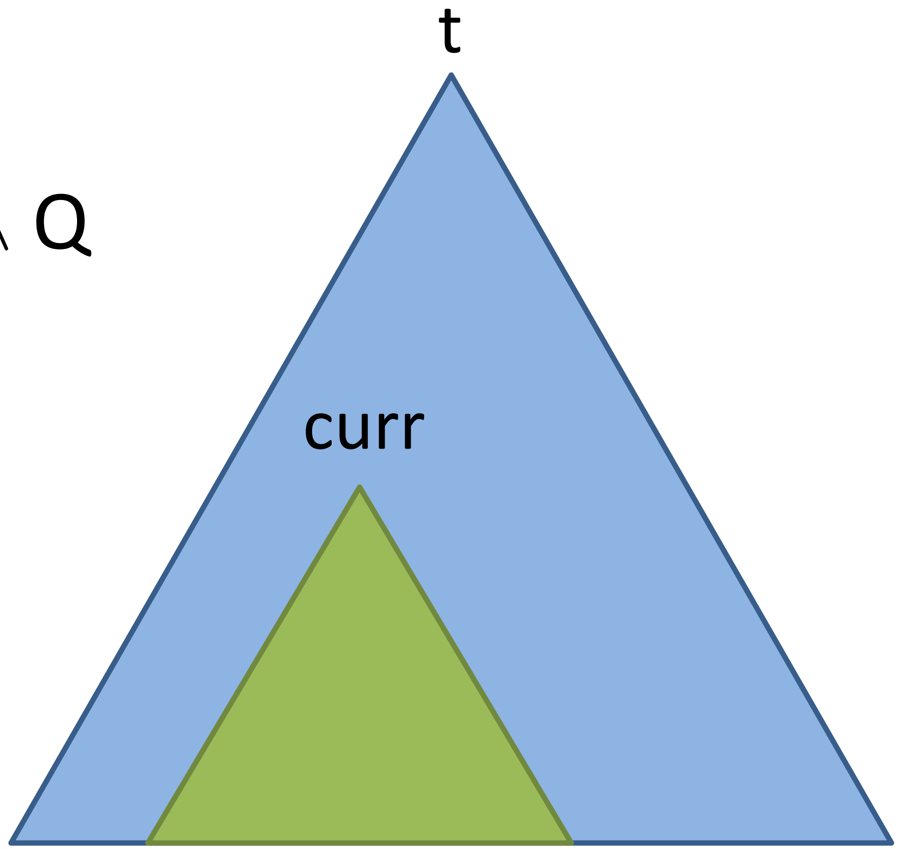
Poor Man's Magic Wand




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

Poor Man's Magic Wand

P -** Q ≡
(P * true) ∧ Q



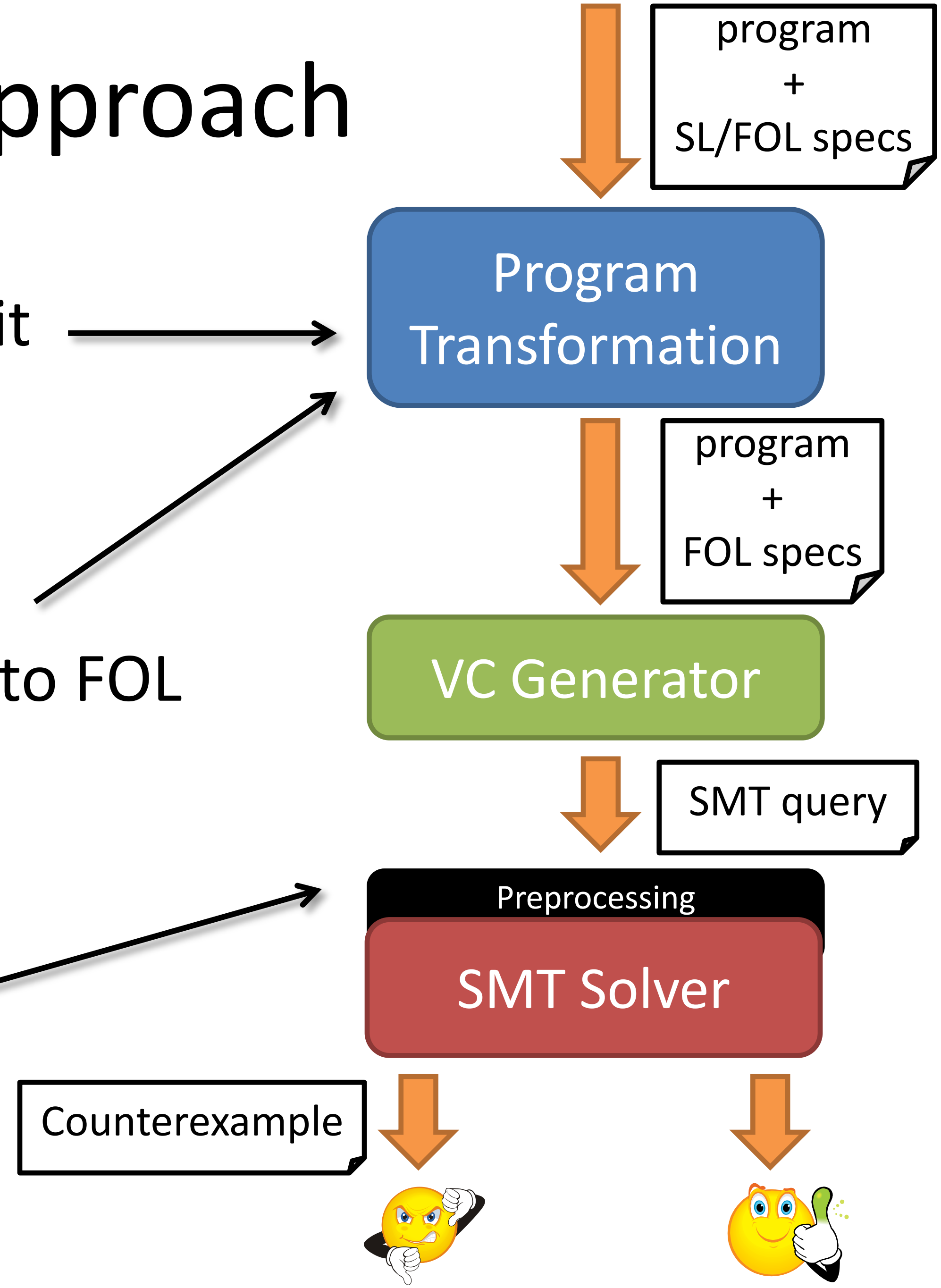
tree(curr) * (tree(curr) -* tree(t))

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;
}
```

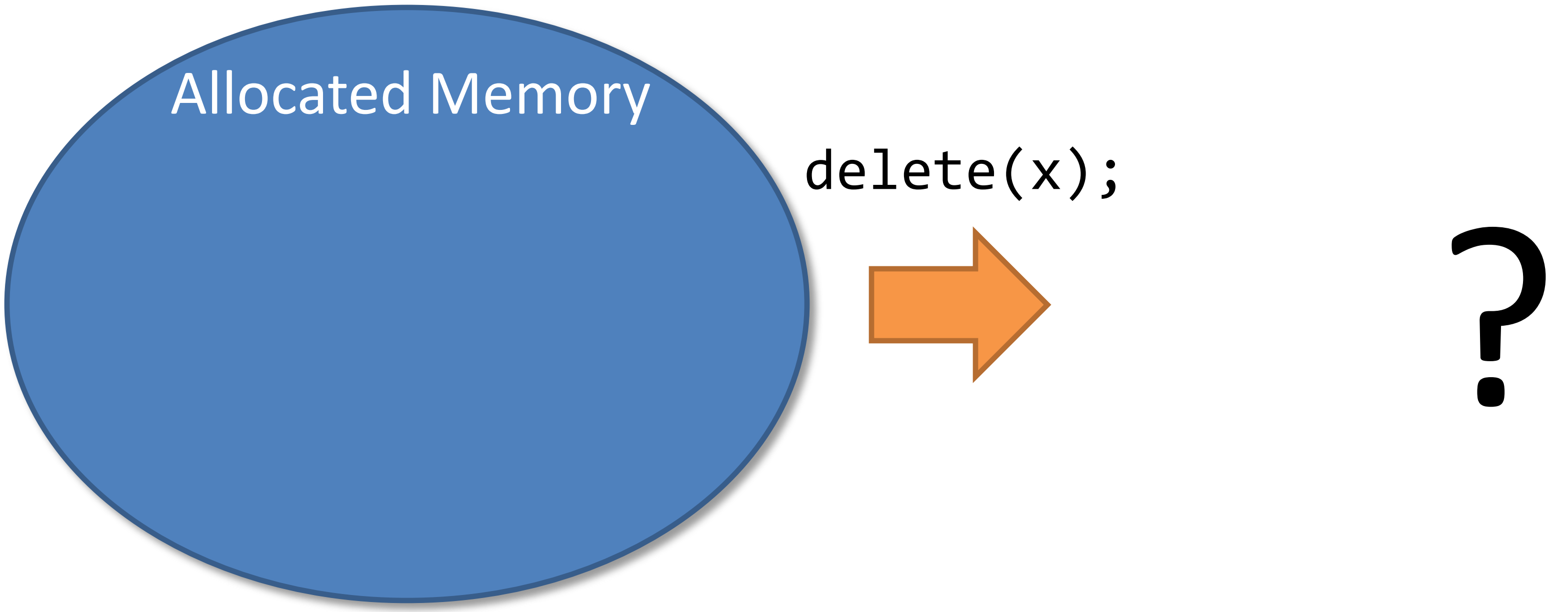
Overview of Approach

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



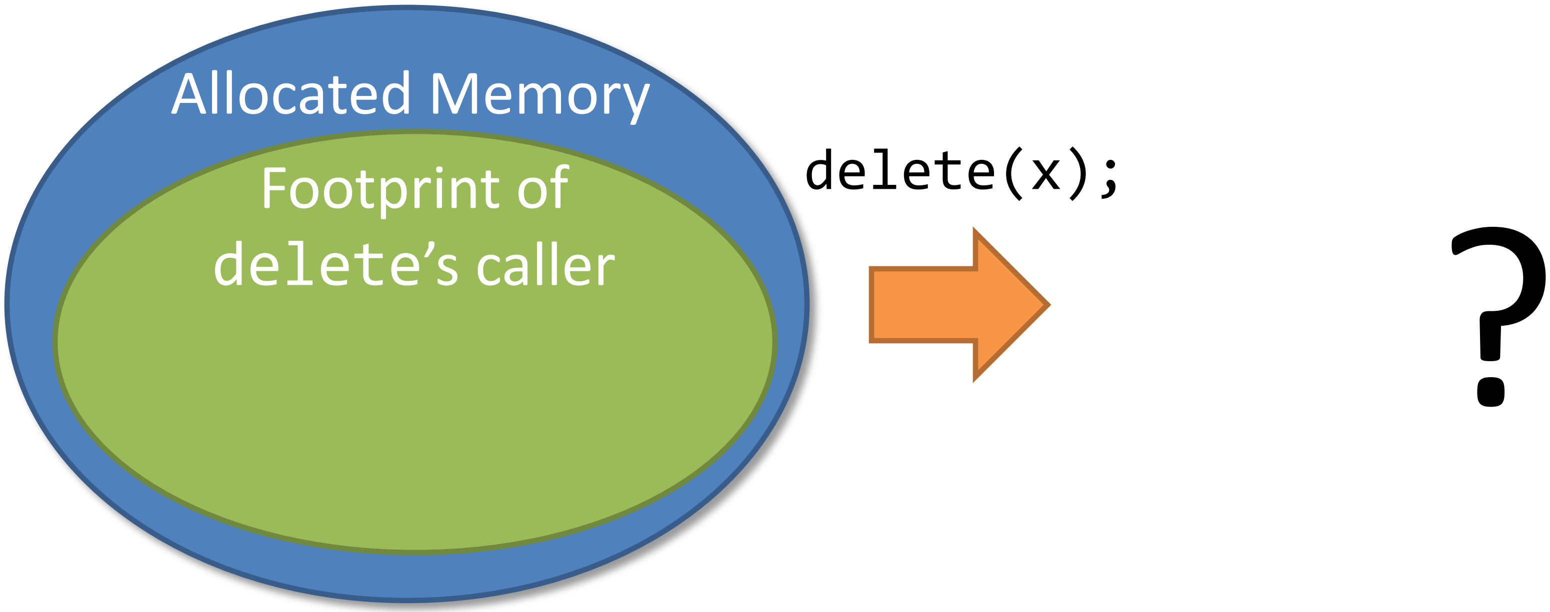
Step 1: Make Frame Rule Explicit

Encoding the Frame Rule



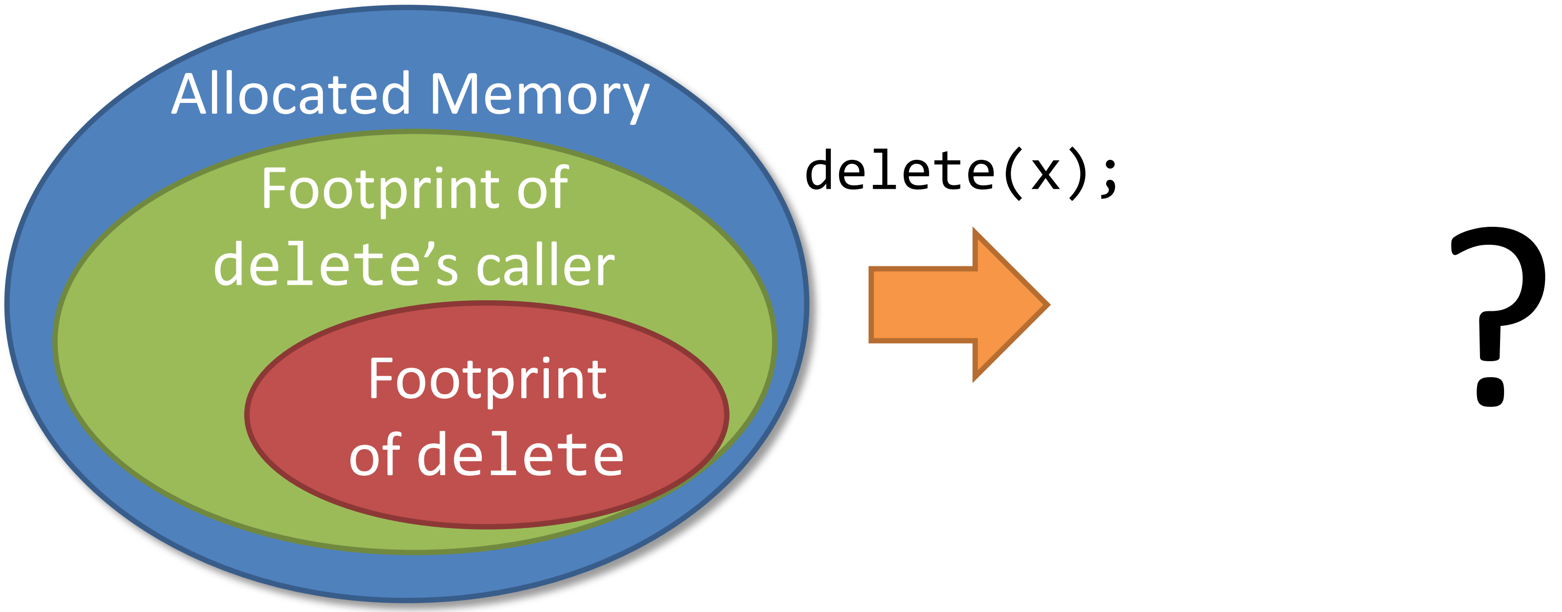
before call to `delete` after call to `delete`

Encoding the Frame Rule



before call to `delete` after call to `delete`

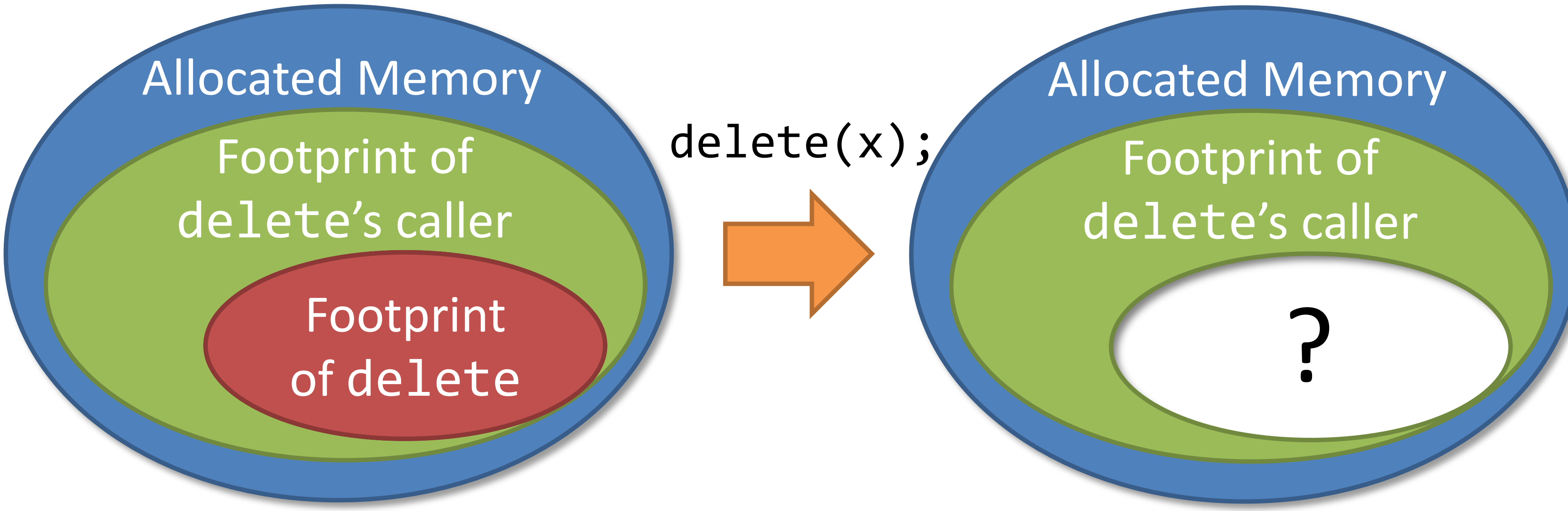
Encoding the Frame Rule



before call to delete

after call to delete

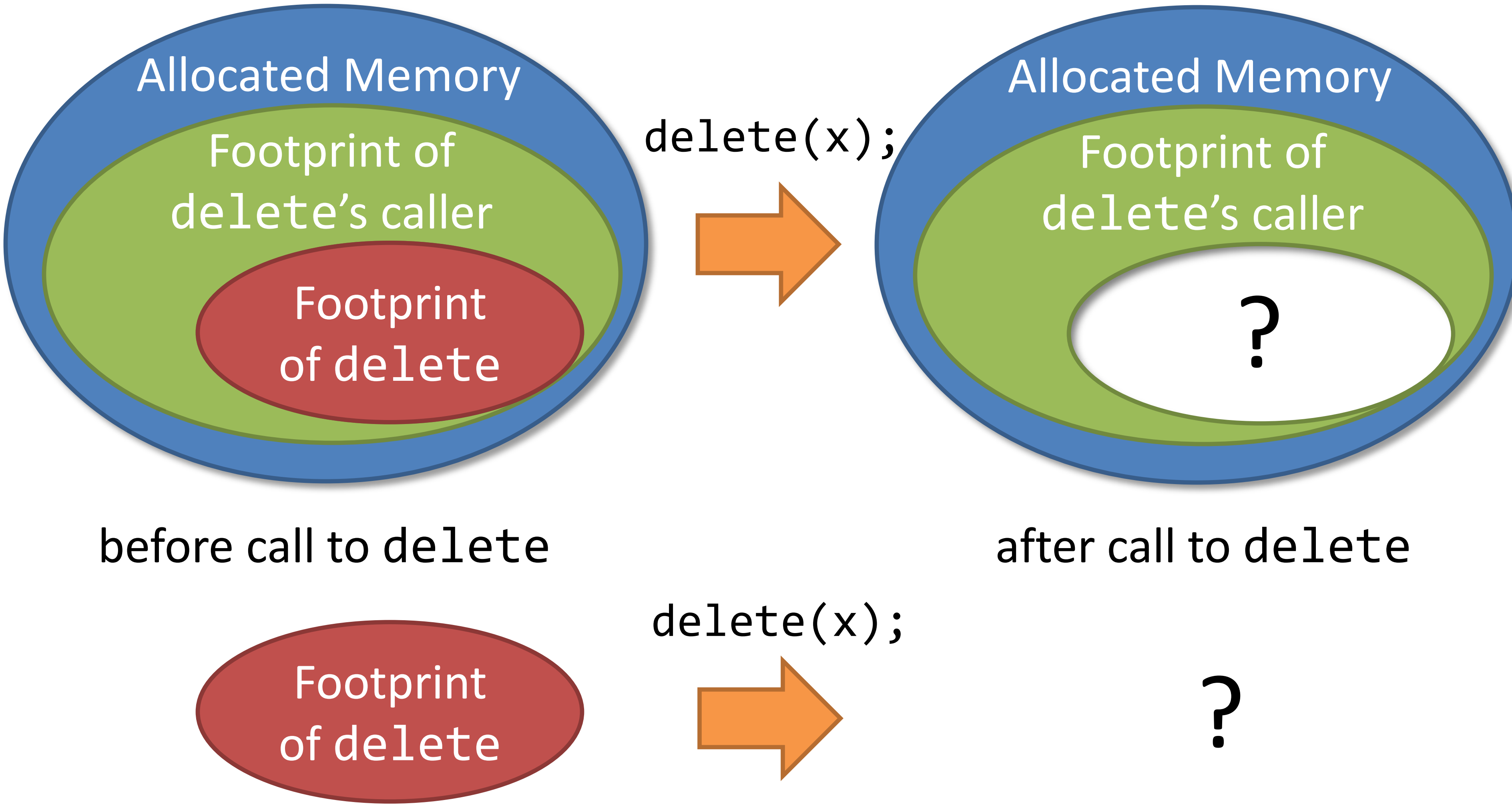
Encoding the Frame Rule



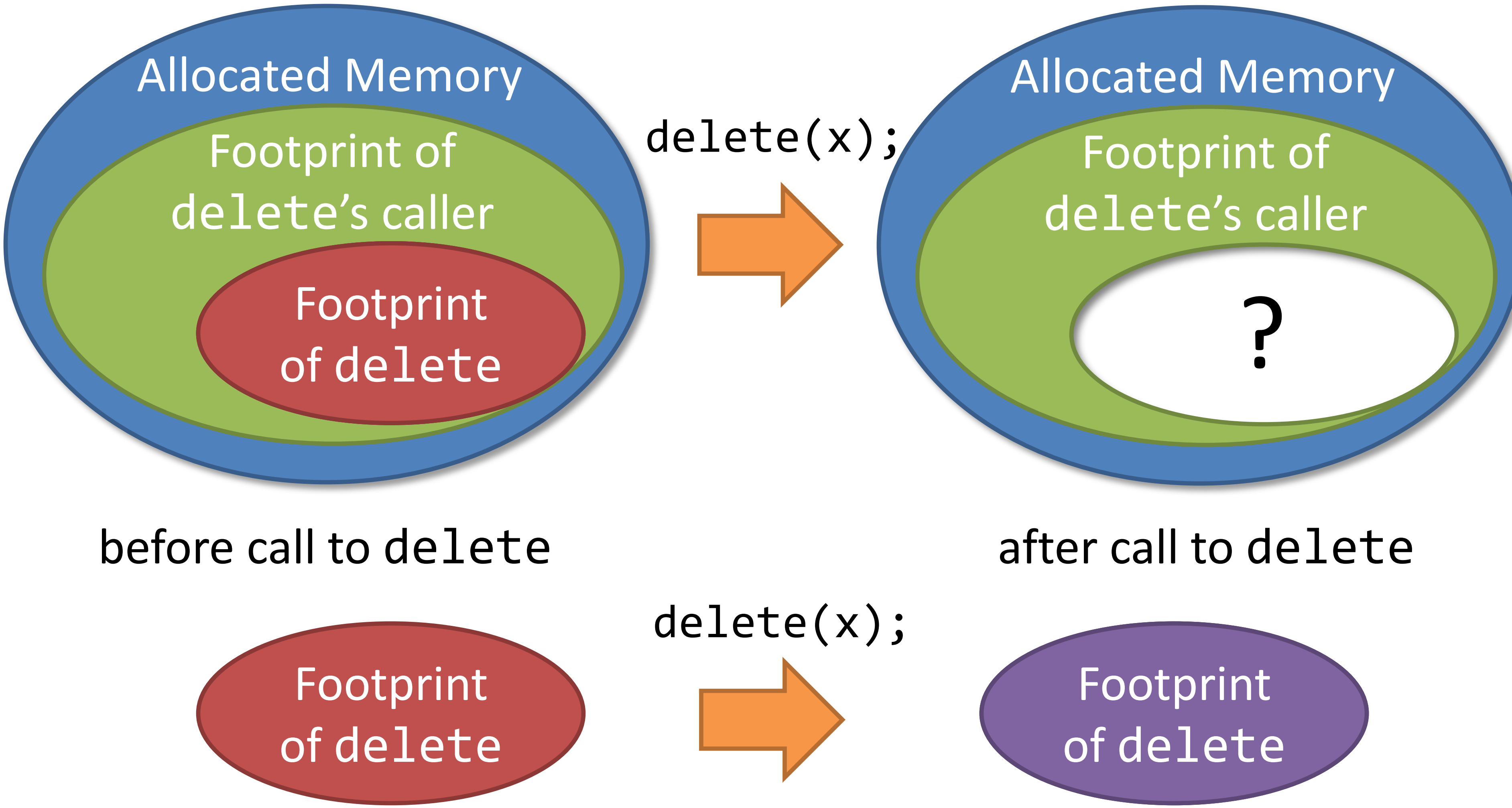
before call to delete

after call to delete

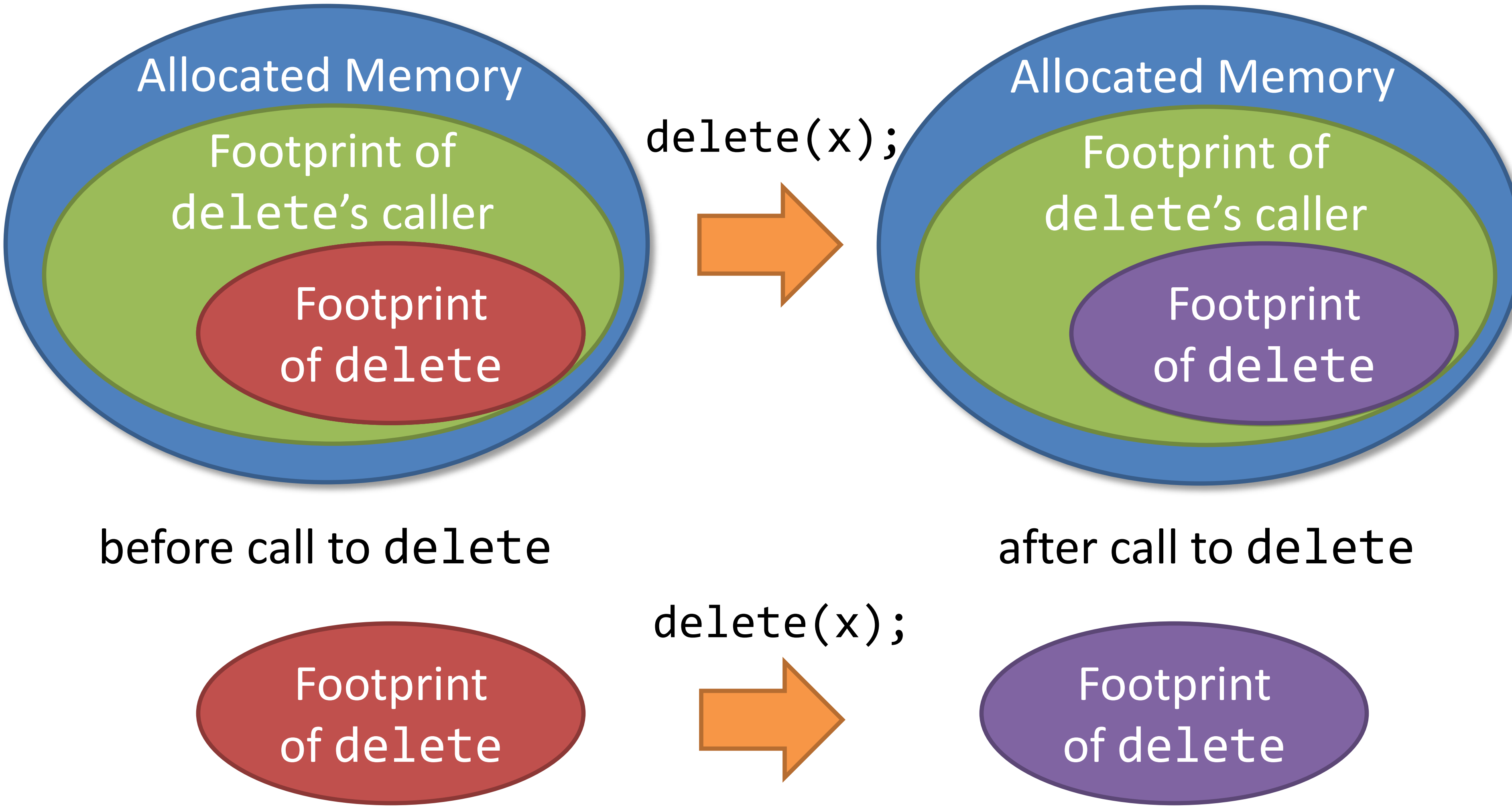
Encoding the Frame Rule



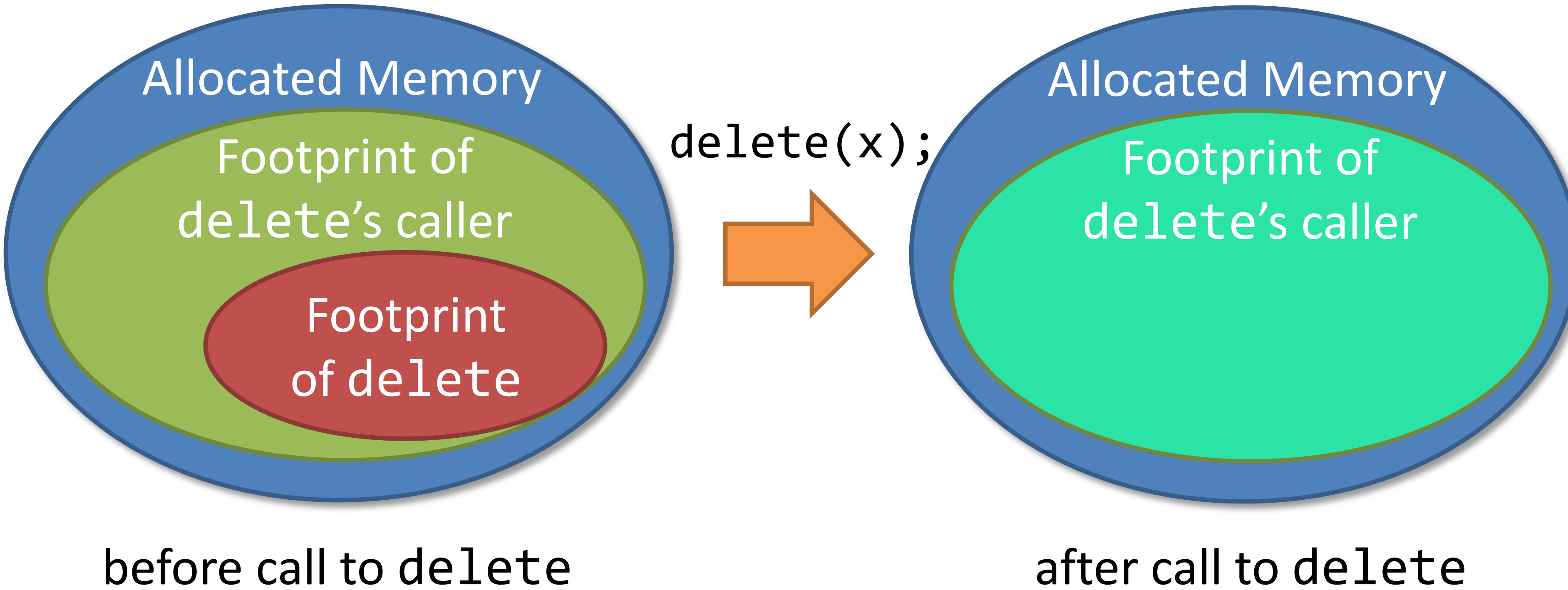
Encoding the Frame Rule



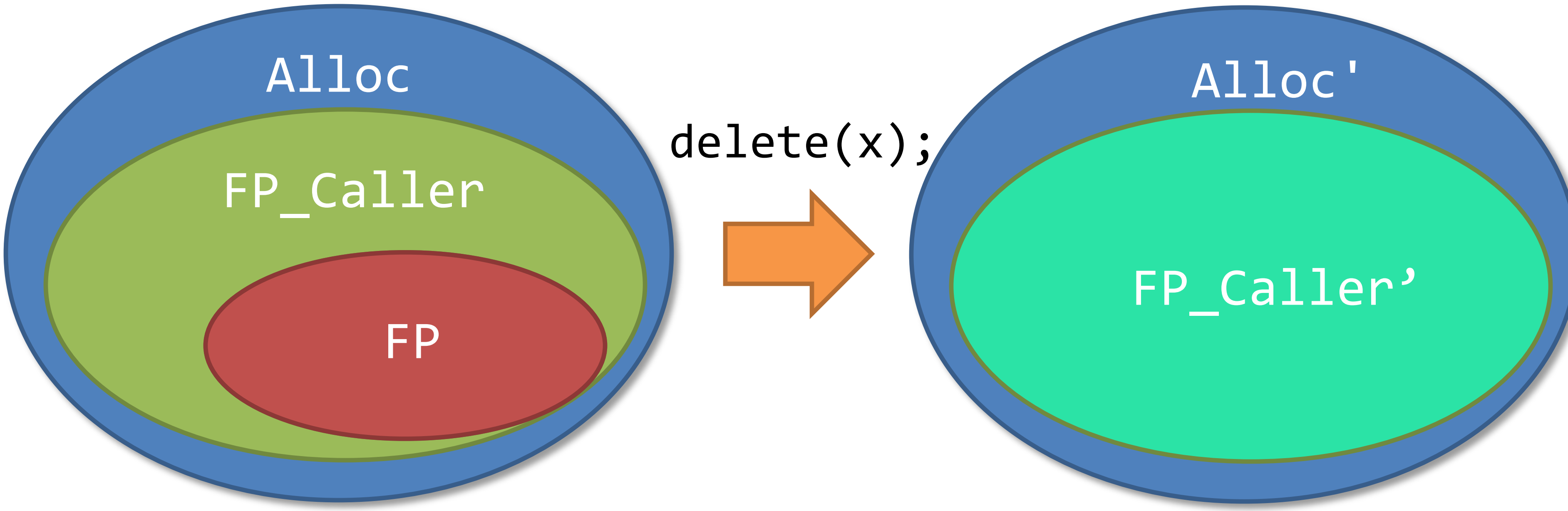
Encoding the Frame Rule



Encoding the Frame Rule



Encoding the Frame Rule



before call to delete

after call to delete

Encoding the Frame Rule

```
procedure delete(x: Node

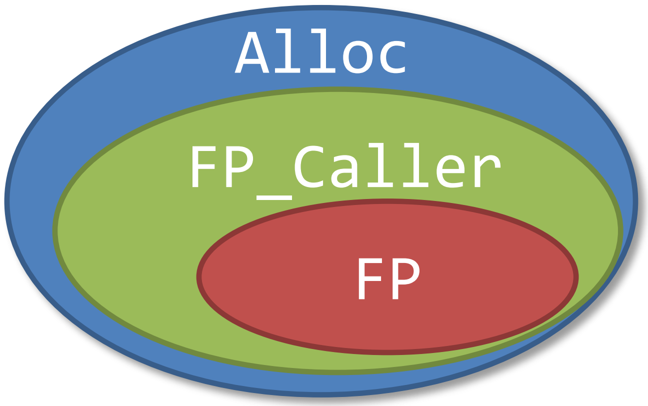
                                            )

{

  if (x != null) {

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

}
```



Encoding the Frame Rule

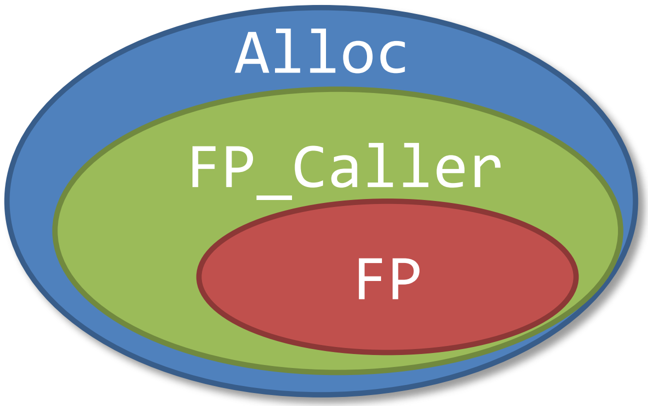
```
ghost var Alloc: Set<Node>;

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);
  }

}
```

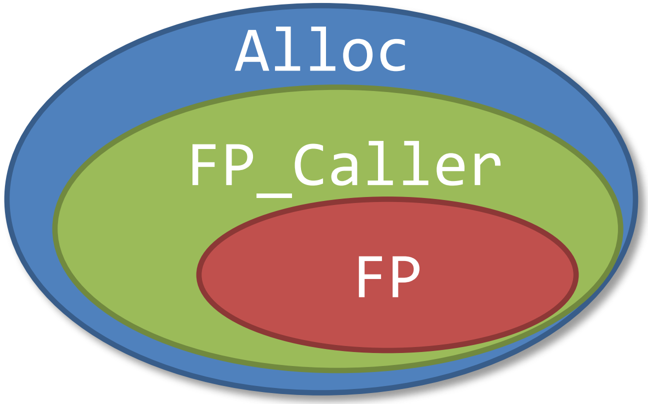


Encoding the Frame Rule

```
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) {

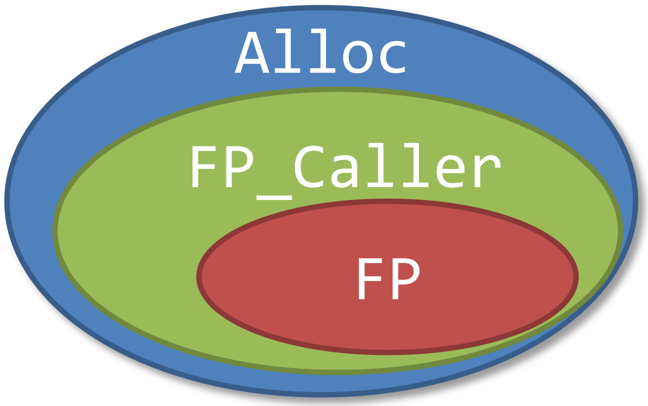
    FP := delete(x.next, FP);
    FP := free(x, FP);
  }
  FP_Caller' := FP_Caller' ∪ FP;
}
```



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
  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
  requ
  free
  free
  ensu
  free                                          t)
  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
{ ... }
```

The secret sauce

Step 2: Translating SL Assertions

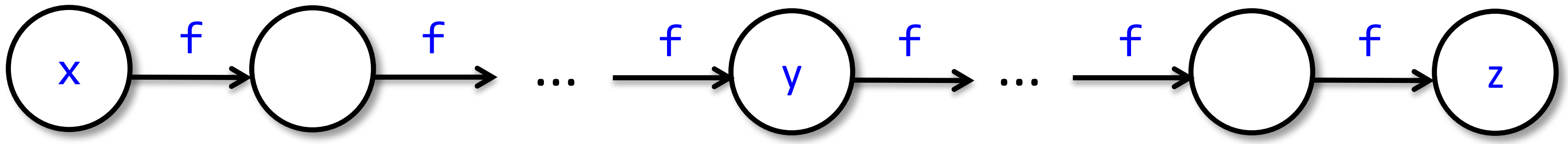
Target of Translation: GRASS (Graph Reachability and Stratified Sets)

- 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])

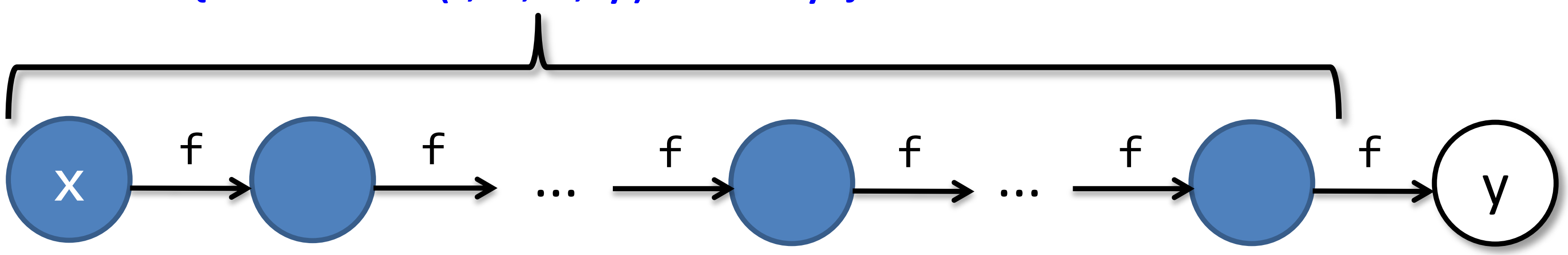
• sel(f, x)	field access	x.f
• upd(f,x,y)	field update	f[x := y]
• Btwn(f,x,y,z)	reachability	$x \xrightarrow{f} y \xrightarrow{f} z$



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

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

Translating SL Assertions to GRASS

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

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(**acc**(t), X) $\equiv$ X = t
- 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 -$**$ 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)

Dealing with Second-Order Quantifiers

Permission sets are uniquely determined by formula structure

- Tr(emp, X) $\equiv$ X = $\emptyset$
- Tr(**acc**(t), X) $\equiv$ X = t
- 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 -$**$ 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)

Dealing with Second-Order Quantifiers

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

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)

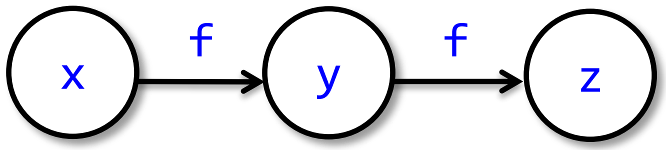
First-Order Axioms for Btwn

- $\forall$ 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)

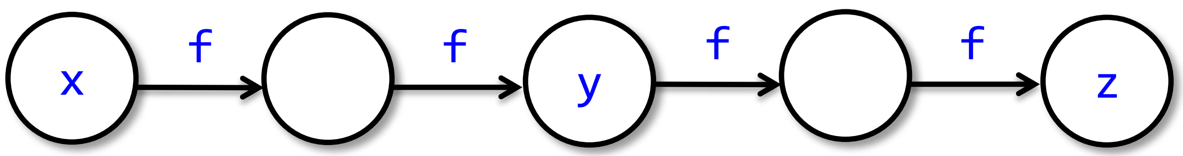
But I thought transitive closure was not first-order definable!?

Completeness of Axioms for Btwn

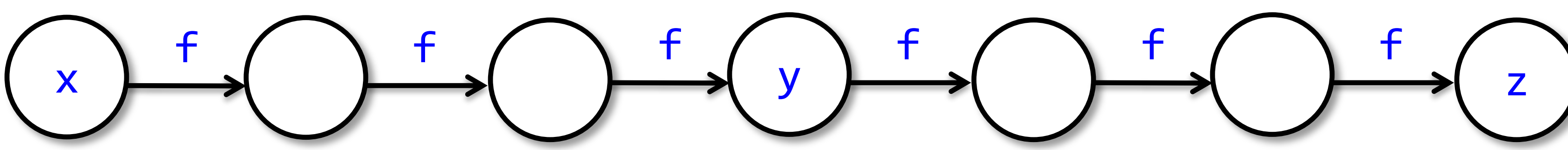
- A model of Btwn(f,x,y,z)



- and another model

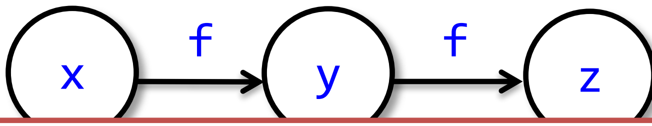


- and another



Completeness of Axioms for Btwn

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

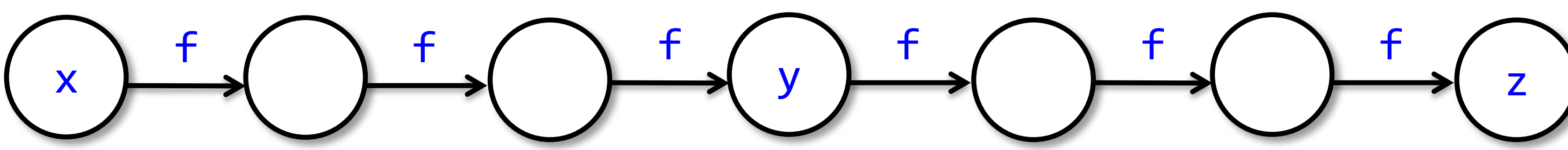


There are arbitrarily large finite models

+

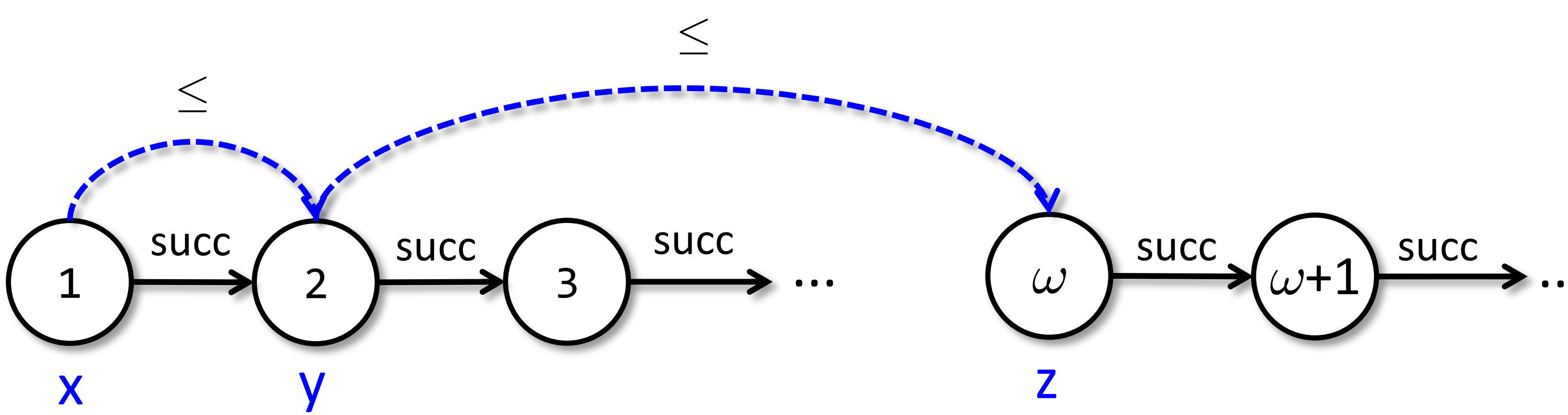
Compactness Theorem $\Rightarrow$ there must also be infinite models

- and another



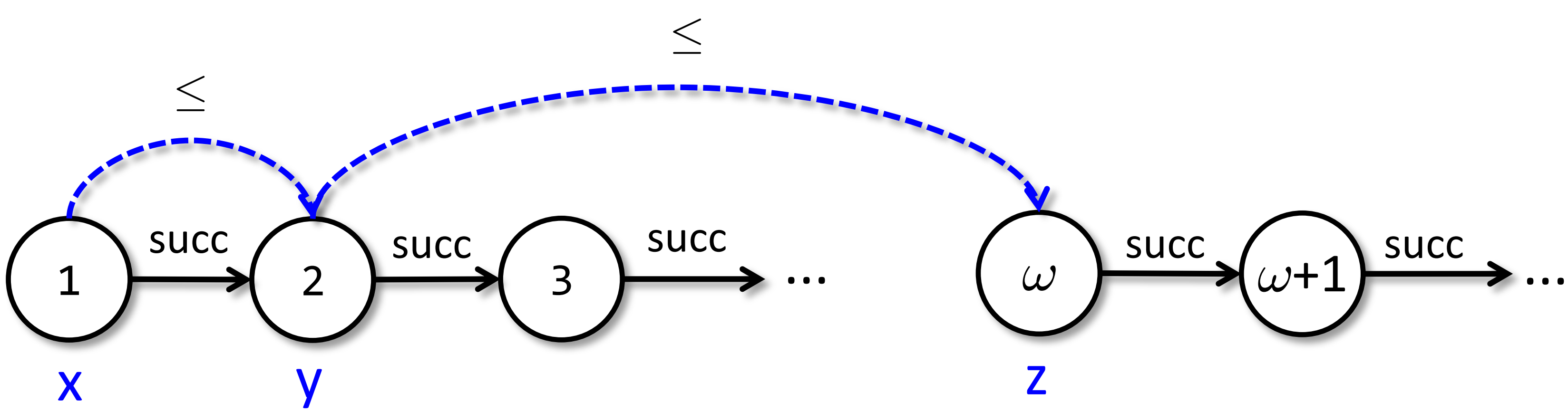
A *Degenerated* Infinite Model M of Btwn(f,x,y,z)

- 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) = $\lambda uvw.\ u \leq v \wedge v \leq w$



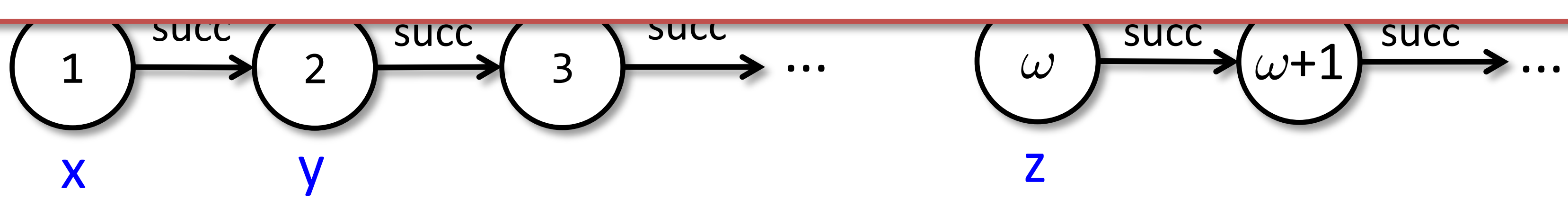
$\leq$ is not succ*

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)

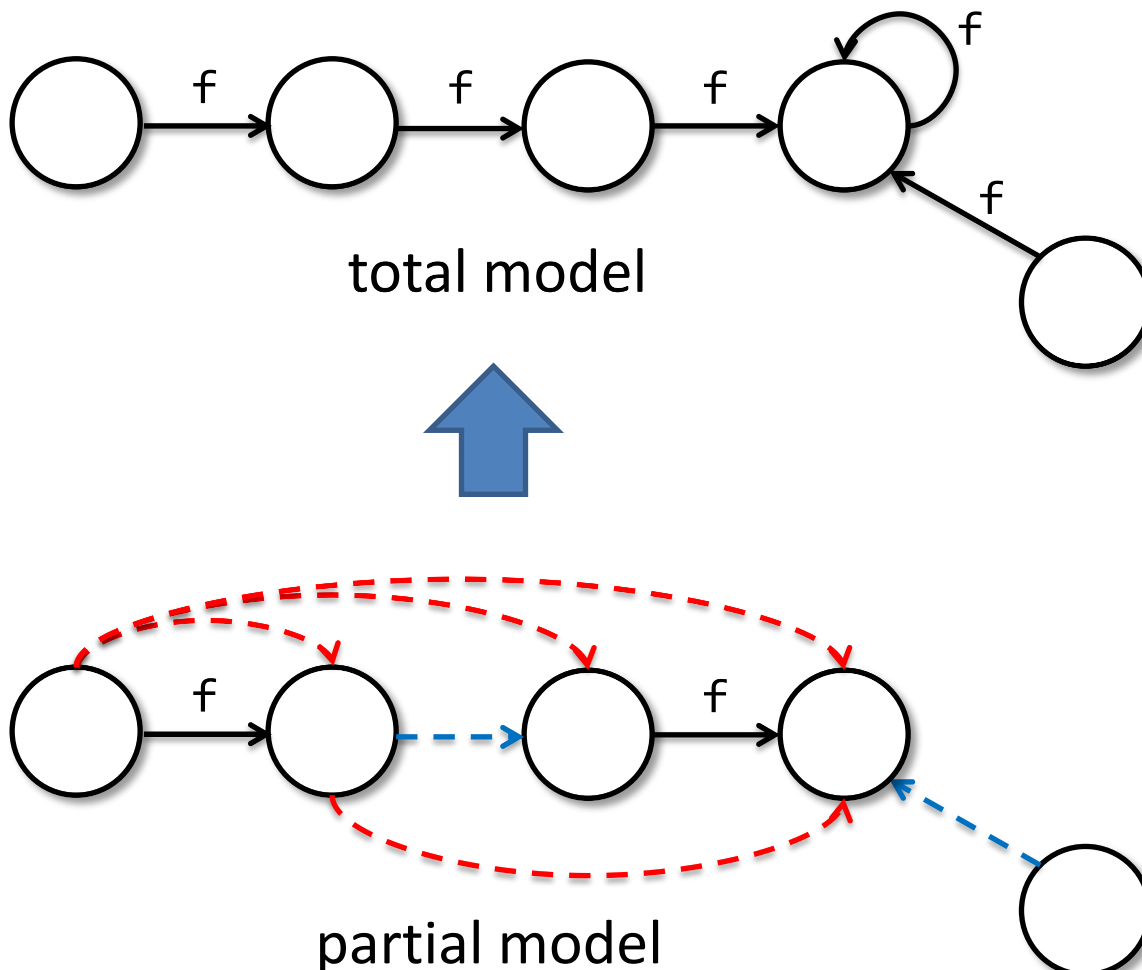
First-Order Axioms for Btwn

Almost in EPR!

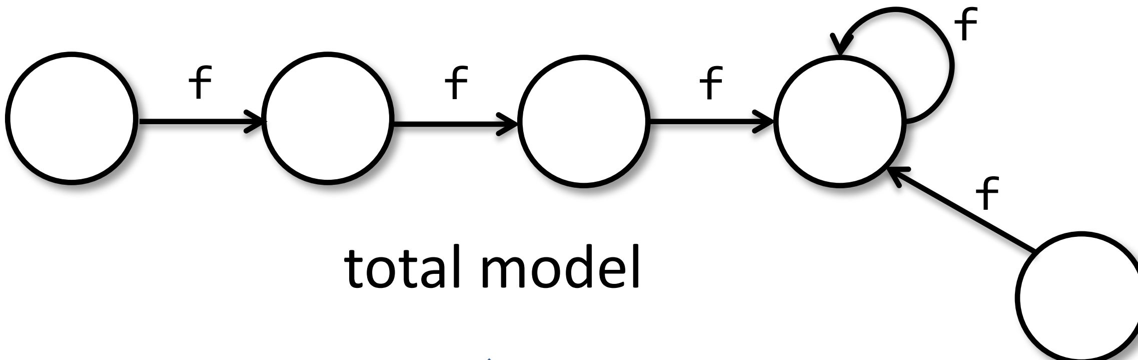
- $\forall f\ x.\ \mathrm{Btwn}(f, x, x, x)$
- $\forall f\ x.\ \mathrm{Btwn}(f, x, \mathbf{sel(f,x)}, \mathbf{sel(f,x)})$
- $\forall f\ x\ y.\ \mathrm{Btwn}(f, x, y, y) \Rightarrow x = y \lor \mathrm{Btwn}(f, x, \mathbf{sel(f, x)}, y)$
- $\forall f\ x\ y.\ \mathbf{sel(f,x)} = x \land \mathrm{Btwn}(f,x,y,y) \Rightarrow x = y$
- $\forall f\ x\ y.\ \mathrm{Btwn}(f, x, y, x) \Rightarrow x = y$
- $\forall f\ x\ y\ z.\ \mathrm{Btwn}(f,x,y,y) \land \mathrm{Btwn}(f,y,z,z) \Rightarrow \mathrm{Btwn}(f,x,y,z) \lor \mathrm{Btwn}(f,x,z,y)$
- $\forall f\ x\ y\ z.\ \mathrm{Btwn}(f,x,y,z) \Rightarrow \mathrm{Btwn}(f,x,y,y) \land \mathrm{Btwn}(f,y,z,z)$
- $\forall f\ x\ y\ z.\ \mathrm{Btwn}(f,x,y,y) \land \mathrm{Btwn}(f,y,z,z) \Rightarrow \mathrm{Btwn}(f,x,z,z)$
- $\forall f\ x\ y\ z\ u.\ \mathrm{Btwn}(f,x,y,z) \land \mathrm{Btwn}(f,y,u,z) \Rightarrow \mathrm{Btwn}(f,x,u,z) \land \mathrm{Btwn}(f,x,y,u)$
- $\forall f\ x\ y\ z\ u.\ \mathrm{Btwn}(f,x,y,z) \land \mathrm{Btwn}(f,x,u,y) \Rightarrow \mathrm{Btwn}(f,x,u,z) \land \mathrm{Btwn}(f,y,u,z)$

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

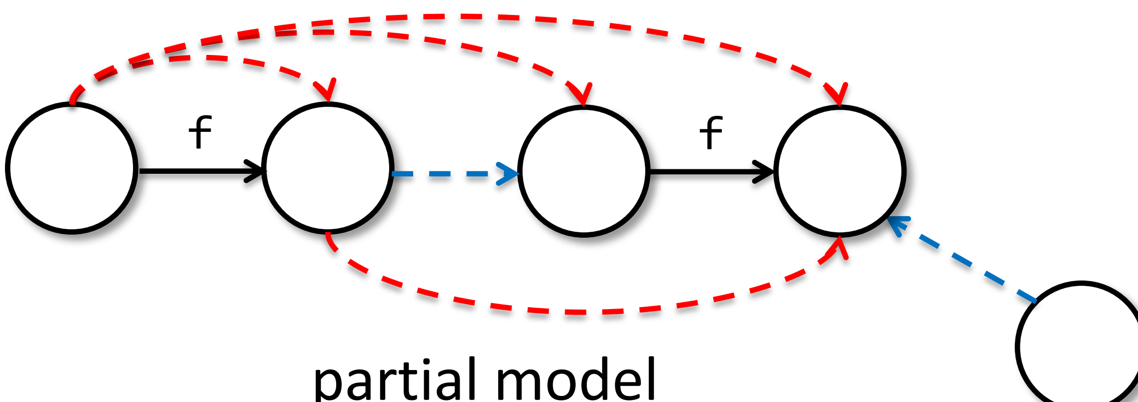


Model Completion for Local Theory Extensions



total model

Yields NP decision procedure for GRASS



partial model

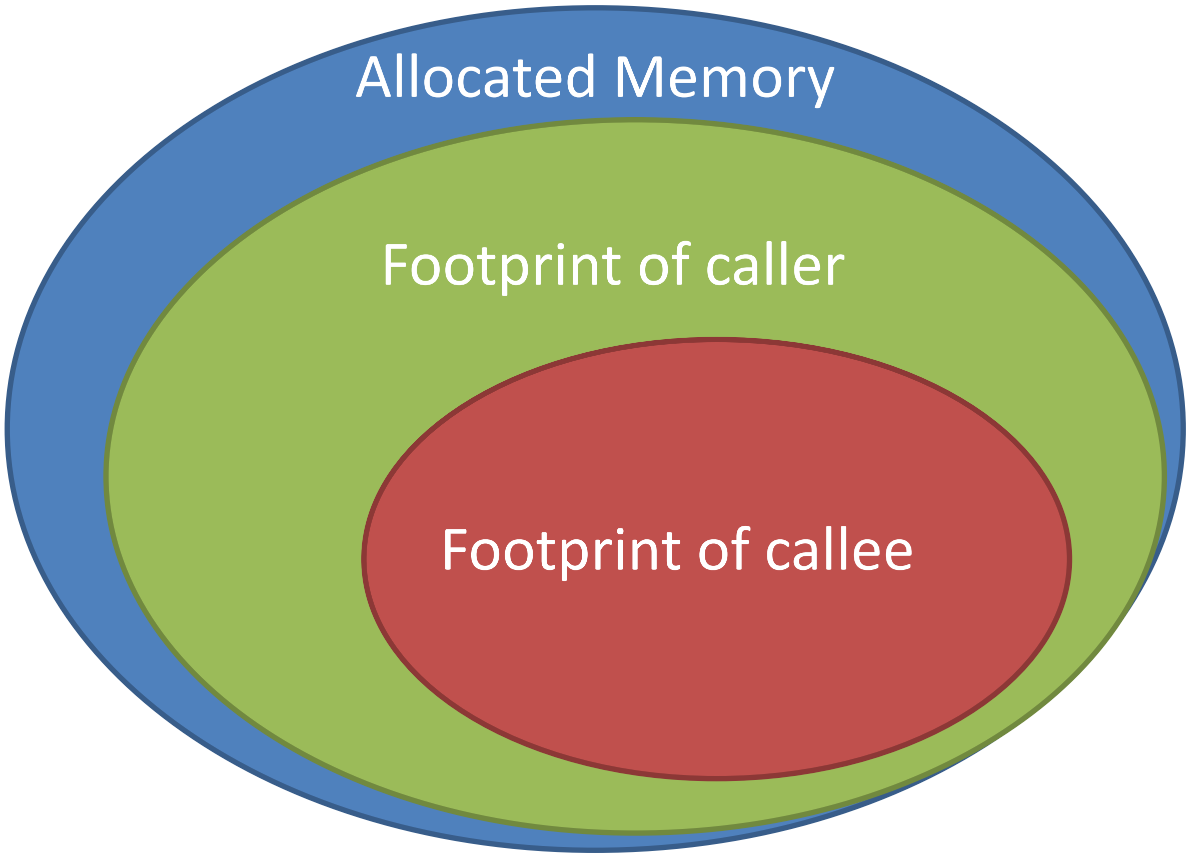
The Frame Predicate

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

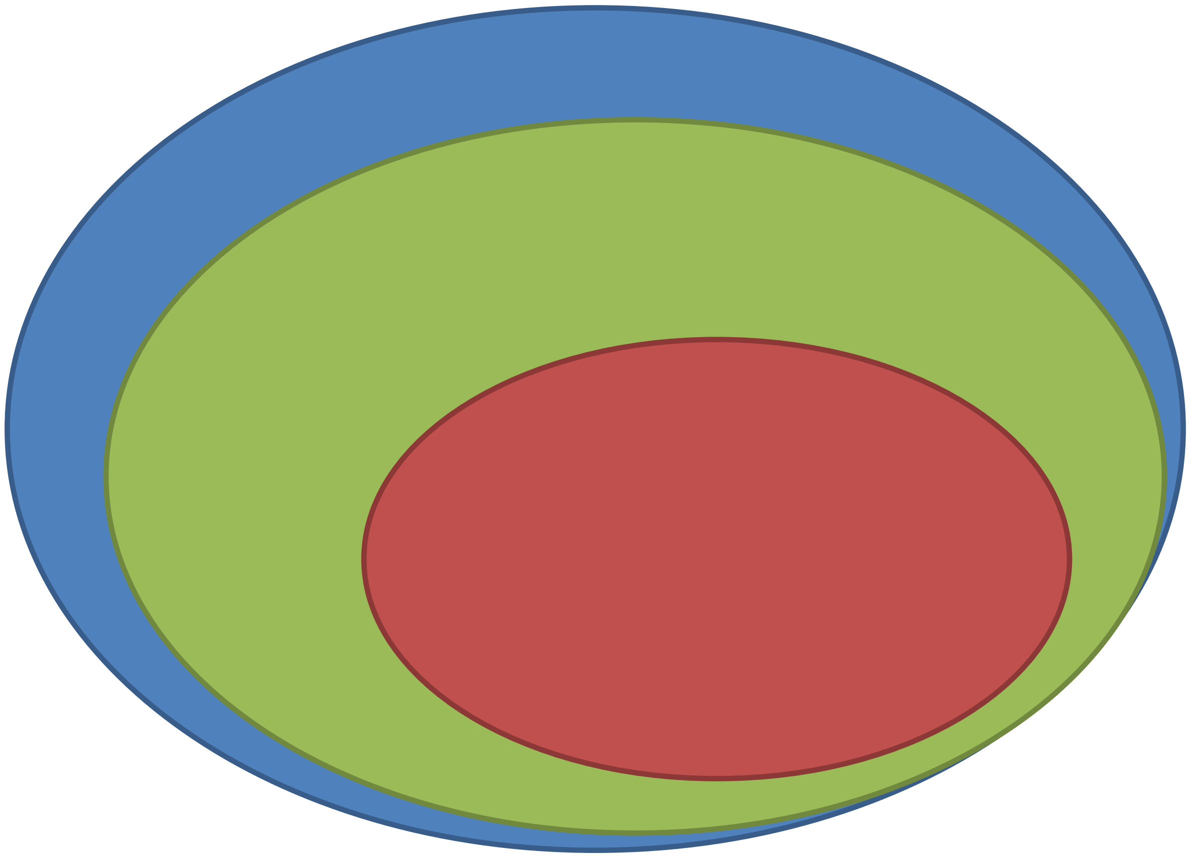
- Does not work with finite instantiation.

- Need to preserve reachability information in frame.

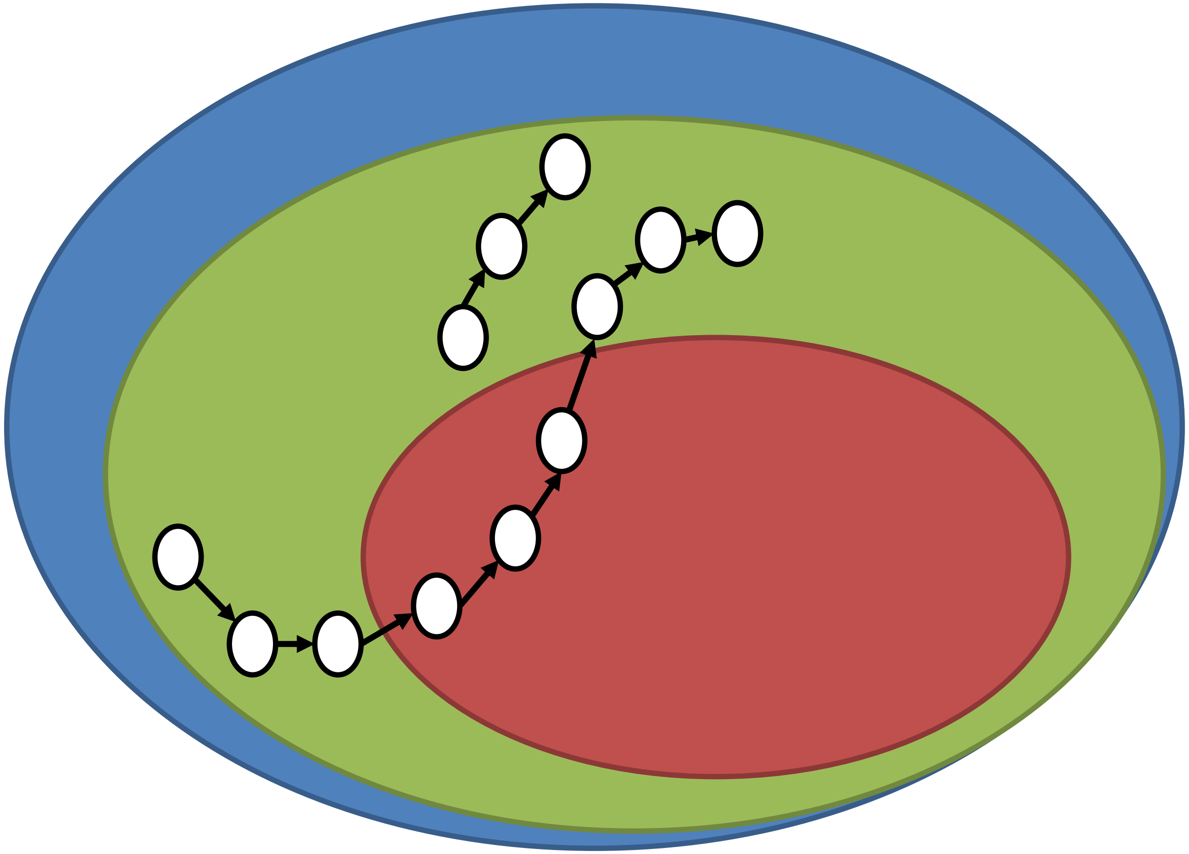
Axiomatizing the Frame Predicate



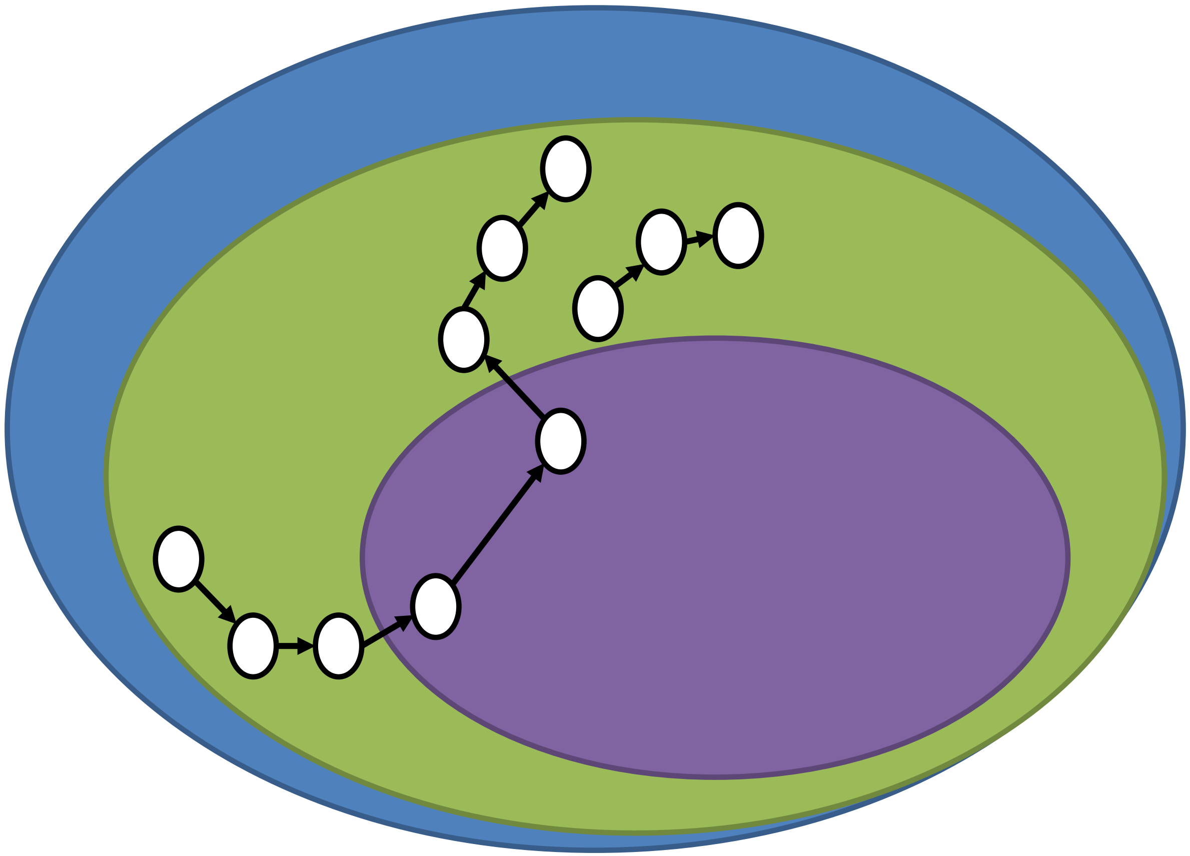
Axiomatizing the Frame Predicate



Axiomatizing the Frame Predicate

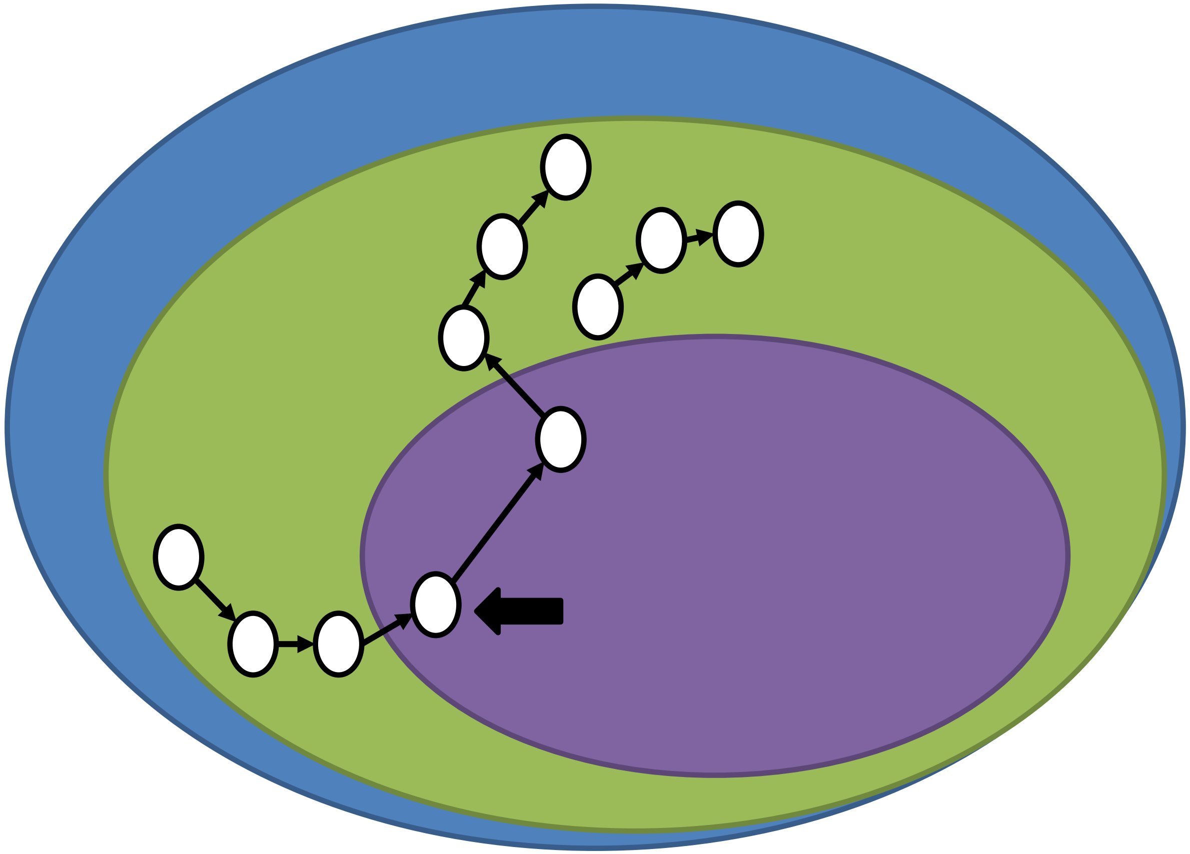


Axiomatizing the Frame Predicate



Local changes have global effect on reachability

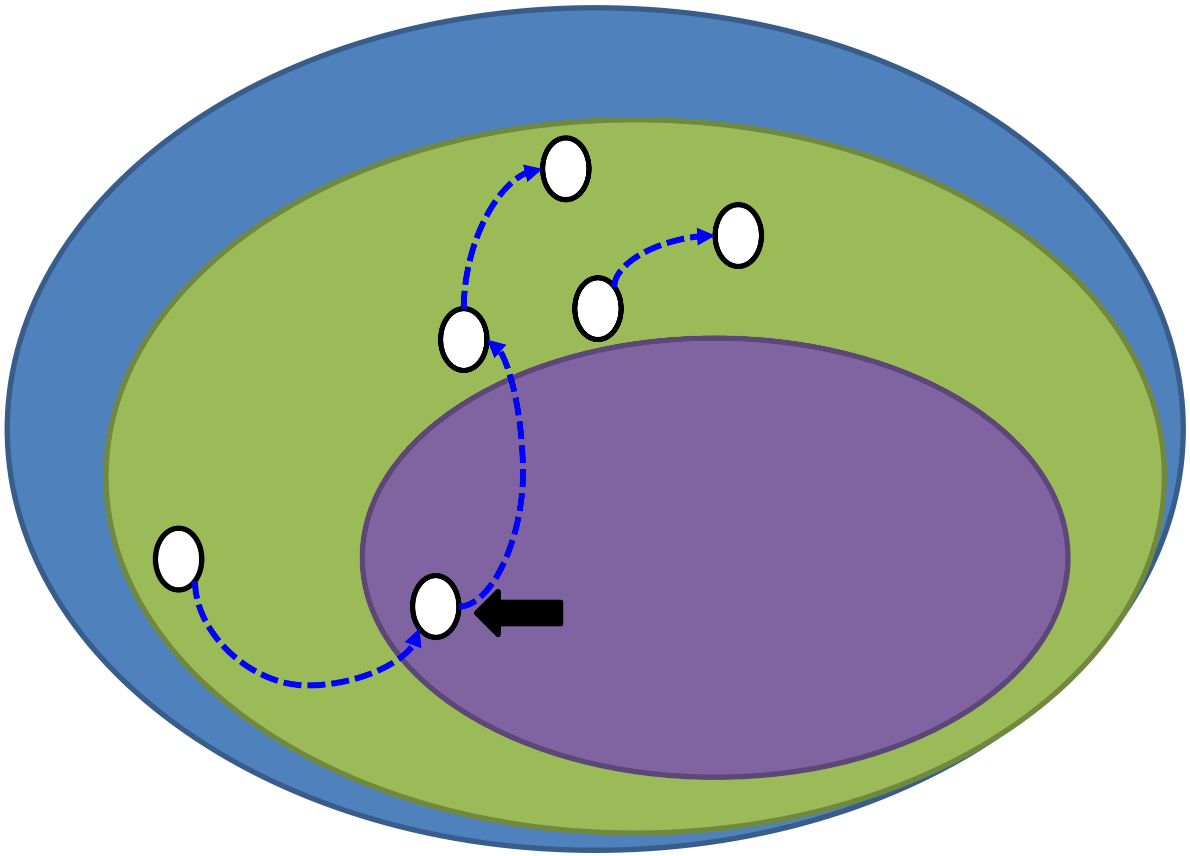
Axiomatizing the Frame Predicate



Local changes have global effect on reachability

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

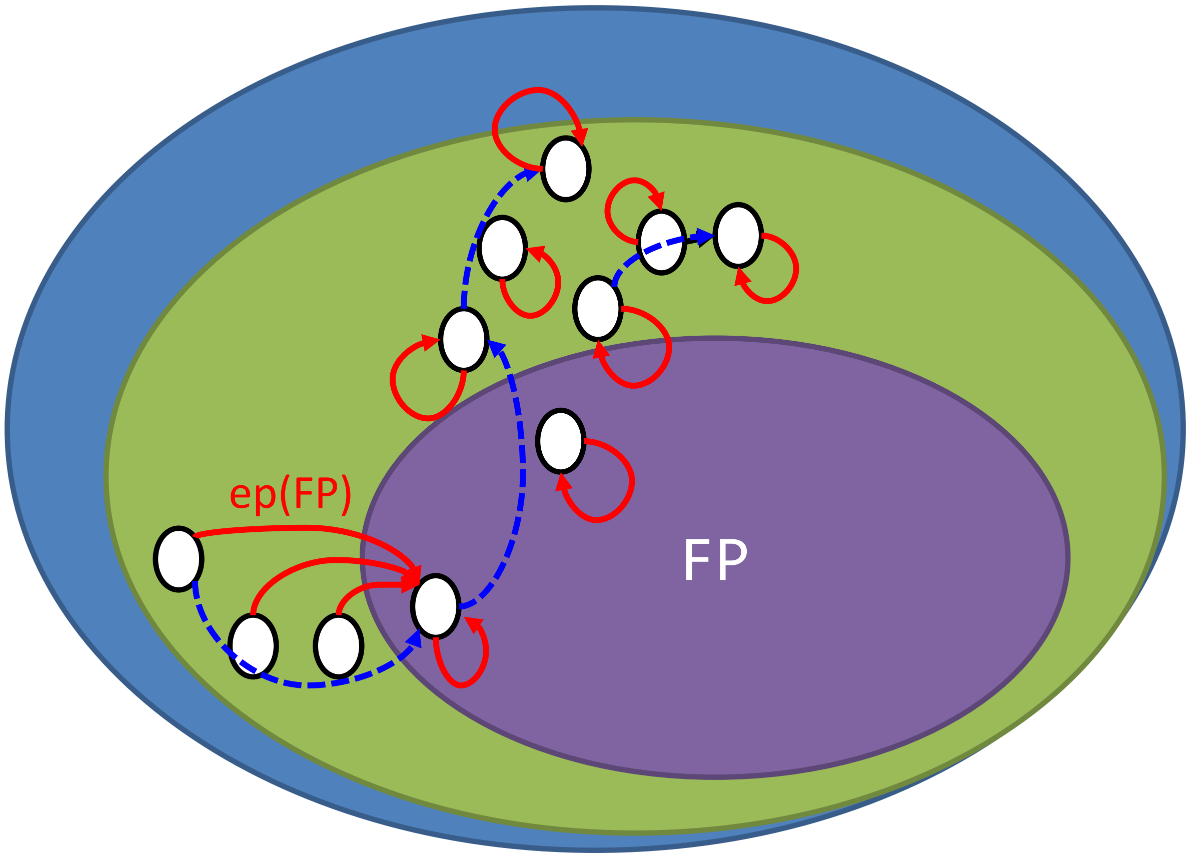
Axiomatizing the Frame Predicate



Local changes have global effect on reachability

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

Axiomatizing the Frame Predicate



Local changes have global effect on reachability

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

GRASShopper

http://github.com/wies/grasshopper

emacs: filter.spl

File Edit Options Buffers Tools C YASnippet Help

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

predicate lseg(x: Node, y: Node) {
  x == y || x != y &*& acc(x) &*& lseg(x.next, y)
}

procedure filter(x: Node) returns (y: Node)
  requires lseg(x, null);
  ensures lseg(y, null);
{
  var curr: Node, prv: Node;
  y := x;
  prv := null;
  curr := y;
  while (curr != null)
    invariant (lseg(y, prv) &*& prv.next |-> curr ||
               prv == null &*& y == curr) &*& lseg(curr, null);
  {
    var nondet: bool;
    if (nondet) {
      if (prv != null) {
        prv.next := curr.next;
      } else {
        y := curr.next;
      }
      free curr;
    } else {
      prv := curr;
    }
!   curr := curr.next;
  }
```

-:--- filter.s[...] Abbrev)

Possible heap access through null or dangling reference

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]