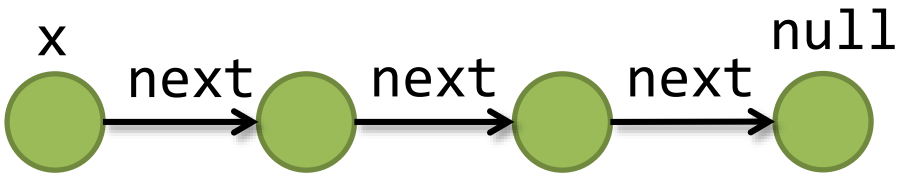
SMT-based Verification of Heap-manipulating Programs

Ruzica Piskac

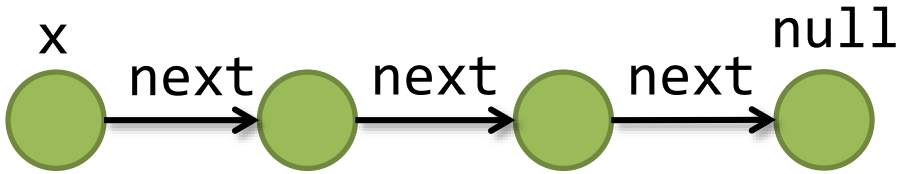
VTSA 2016

A Motivating Example

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



From Hand-Waving to Proofs

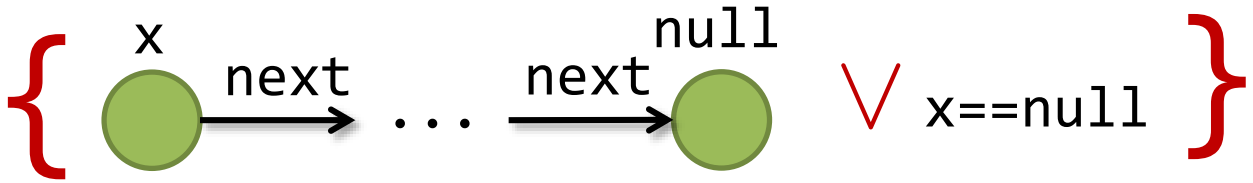
{ 

}

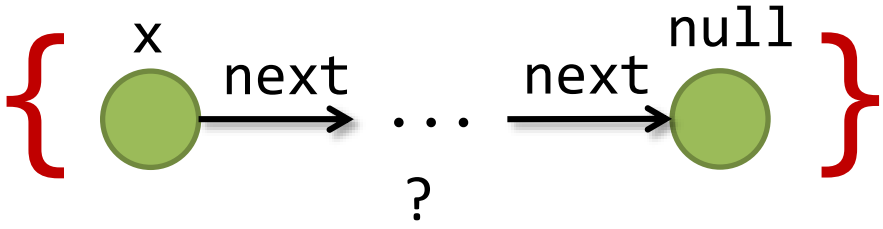
```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```

{ }

From Hand-Waving to Proofs

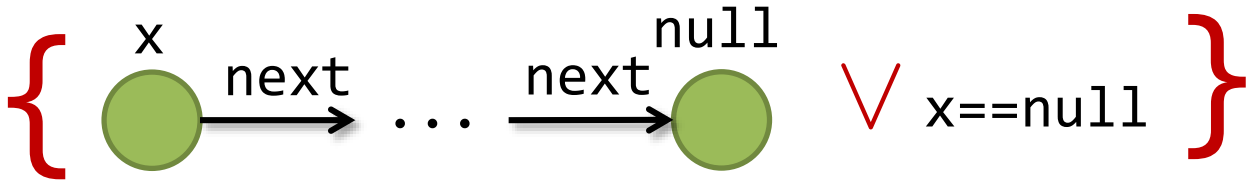



```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```

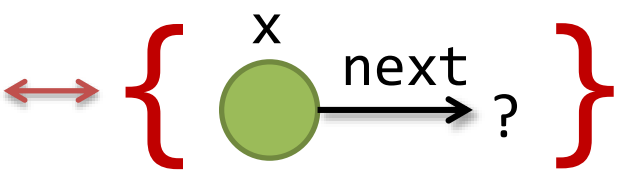


{ }

From Hand-Waving to Proofs

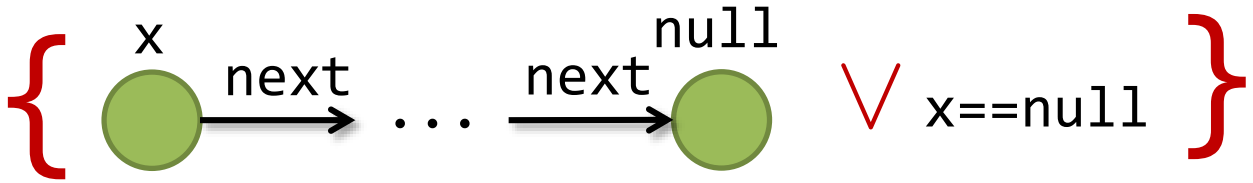



```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```



{ }

From Hand-Waving to Proofs

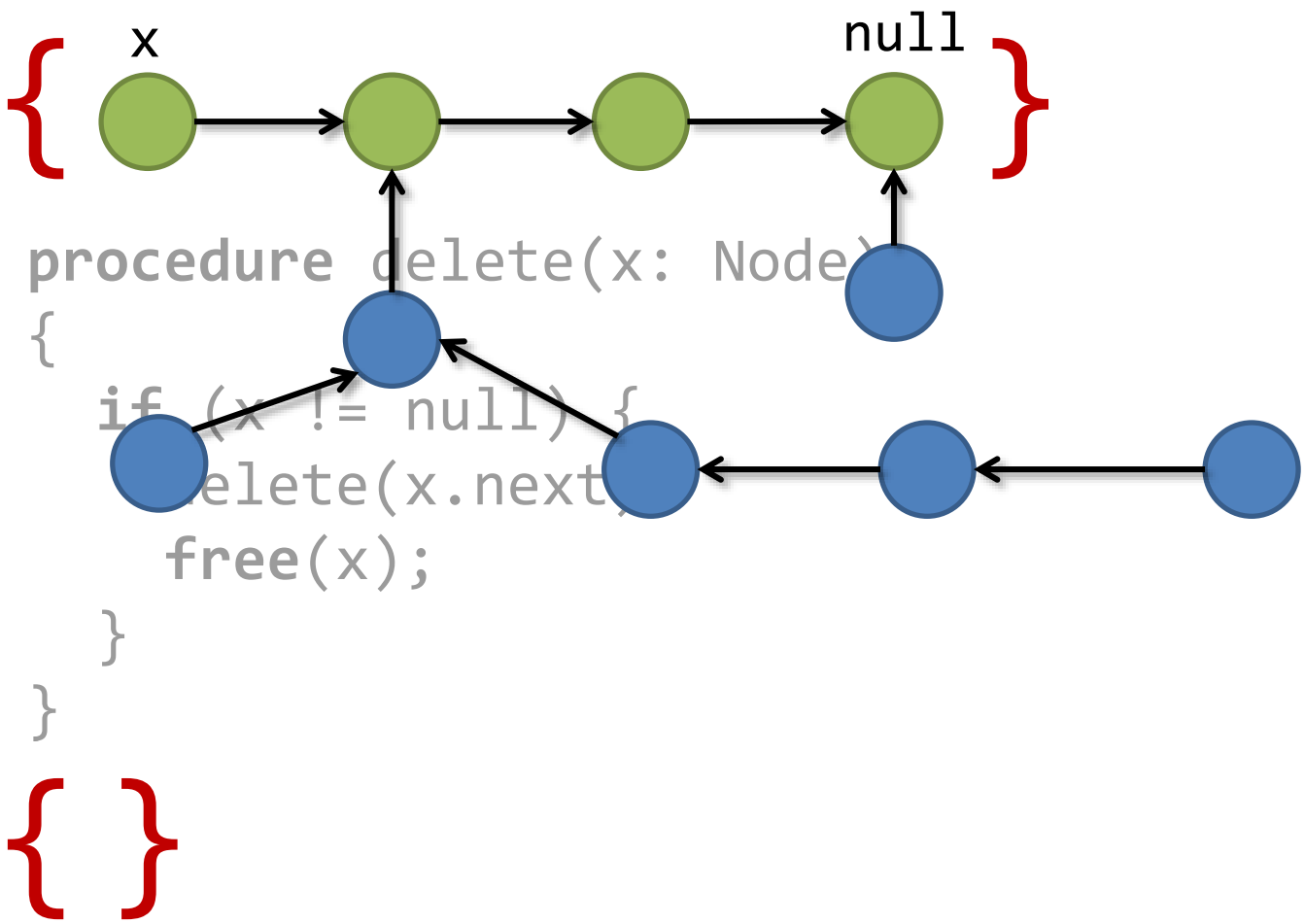



```
procedure delete(x: Node)
{
  if (x != null) {
    delete(x.next);
    free(x);
  }
}
```

{ ? } ⟷ { }

{ }

From Hand-Waving to Proofs

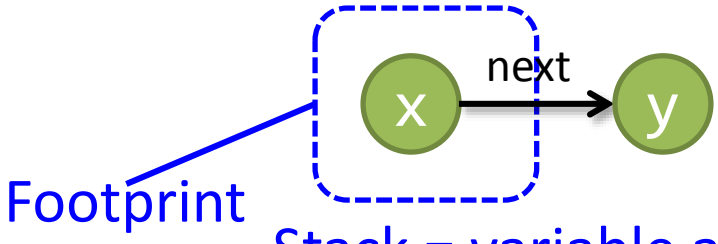


Separation Logic

[Reynolds, O'Hearn, Yang, 2001]

- assertions express heap portions (heaplets)
  - emp : "the heaplet is empty"
  - acc(x) : "the heaplet has *exactly* one single node x"
  - F ∗ G : "the heaplet can be divided so F is true of one partition and G of the other"
  - x = y : "the heaplet is *empty* and x and y are equal"
  - ... (as in classical logic)

acc(x) ∗ x.next = y

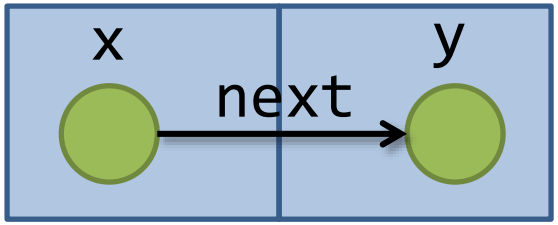


Stack = variable assignments

A Simple Permission-based Logic

- Separating conjunction

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



A Simple Permission-based Logic

- Separating conjunction

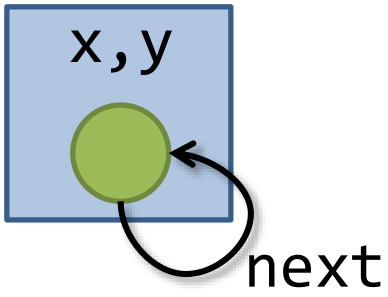
```
acc(x) * acc(x) * x.next == y
```

unsatisfiable

A Simple Permission-based Logic

- Classical conjunction

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



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

Syntactic Short-hands

- Empty heap:

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

- Points-to predicates:

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

Formal Semantics

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

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

Entailment

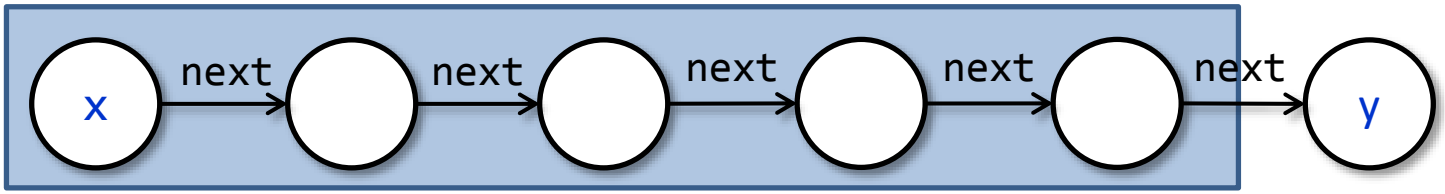
- $F \models G \quad \Leftrightarrow \quad$ for all M, D : $M, D \models F$ implies $M, D \models G$

- Various decidable fragments
  - Linked lists
    [Berdine, Calcagno, O'Hearn 2005], [Cook et al. 2011]
  - Recursive predicates of bounded tree width
    [Iosif, Rogalewicz, Simacek 2013]
  - …
  - also see survey [Demri, Deters 2015]

Recursive Predicates

- acyclic list segment

$$\texttt{lseg(x, y)} \equiv$$
$$\texttt{x} = \texttt{y} \vee$$
$$\texttt{x} \neq \texttt{y} * \textbf{acc}(\texttt{x}) * \texttt{lseg(x.next, y)}$$



Expressing Properties in SL

```
/* A node in a singly-linked list. */
struct Node {
  var next: Node;
}

/* Predicate denoting an acyclic singly-linked list
 * starting from 'x'. */
predicate list(x: Node) {
  acc({ z: Node :: Btwn(next, x, z, null) && z != null }) &*&
  Reach(next, x, null)
}

/* Concatenate two lists 'a' and 'b'.
 * The result is a single list 'res'. */
procedure concat(a: Node, b: Node)
  returns (res: Node)
  requires list(a) &*& list(b)
  ensures list(res)
{
  if (a == null) {
    return b;
  } else {
    var curr := a;

    while (curr.next != null)
      invariant acc(curr) **- list(a)
    {
      curr := curr.next;
    }
    curr.next := b;
    return a;
  }
}
```

Decidable Fragment of Linked Lists

[Berdine, Calcagno, O'Hearn 2005]

Some of the proof rules:

$$\frac{A \vdash B}{A * F \vdash B * F}$$

$$\frac{x \neq y * \mathbf{acc}(x) * \mathbf{acc}(y) * A \vdash B}{\mathbf{acc}(x) * \mathbf{acc}(y) * A \vdash B}$$

$$\frac{\begin{array}{c} x = y * A \vdash B \\ x \neq y * z \neq y * x.\mathrm{next} \mapsto z * z.\mathrm{next} \mapsto y * A \vdash B \end{array}}{\mathbf{lseg}(x, y) * A \vdash B} \quad z \text{ fresh}$$

$$\frac{A \vdash B}{A \vdash \mathbf{lseg}(x, x) * B}$$

$$\frac{x \neq z * A \vdash \mathbf{lseg}(y, z) * B}{x \neq z * x.\mathrm{next} \mapsto y * A \vdash \mathbf{lseg}(x, z) * B}$$

Permission-based Hoare Logic

Hoare Rules

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

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

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

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

Validity of Hoare Triples

- Hoare triple {P} c {Q} is valid iff for every state M and set of heap locations D of M:
  - if M,D ⊨ P and all locations in D are allocated, then
  - if c terminates in a state M' and D' consists of the locations in M' that are freshly allocated by c and the remaining locations in D not deallocated by c, then M',D' ⊨ Q, and
  - executing c in M will only access locations that are in D or freshly allocated by c

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} 

Frame Rules

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

Frame inference

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

Specification of `delete`

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

Verifying delete

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

Frame inference: ? = acc(x) * x≠null

acc(x) * x≠null ⊢ ?

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

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

Verifying delete

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

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)

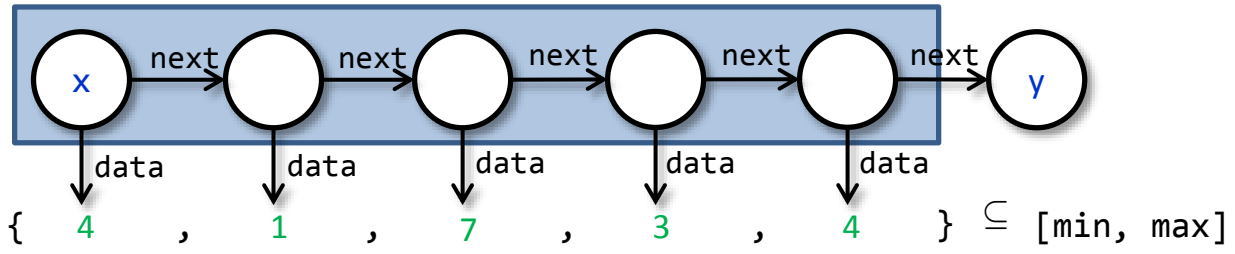
- Strong theoretical guarantees:
  sound, **complete**, **tractable complexity (NP)**
- Mixed specs: escape hatch when SL is not suitable.

Reasoning about Heap and Data

Inductive Predicates with Data

- bounded list segment

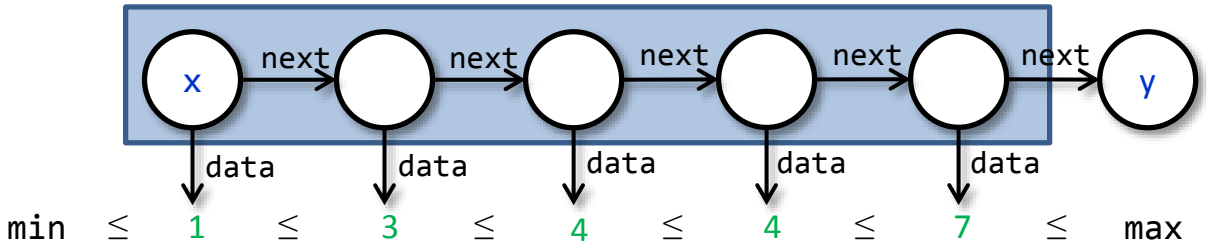
```
bnd_lseg(x, y, min, max) =
      x = y ∨
      x ≠ y * acc(x) * min ≤ x.data ≤ max *
      bnd_lseg(x.next, y, min, max)
```



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;
}
```

Limitations of Decidable SL Fragments

- No robust support for reasoning about data
  - can combine with SMT solvers [Bouajjani, Dragoi, Enea, Sighireanu, ATVA'12], [Pérez, Rybalchenko, APLAS'13]
  - but equality propagation and frame inference are tricky

Reduce Separation Logic to First-Order Logic

Requirements: approach should

- integrate well with SMT solvers
  - support theory combination
- provide theoretical guarantees
  - soundness
  - completeness
  - decidability (in NP)
- automate the frame rule

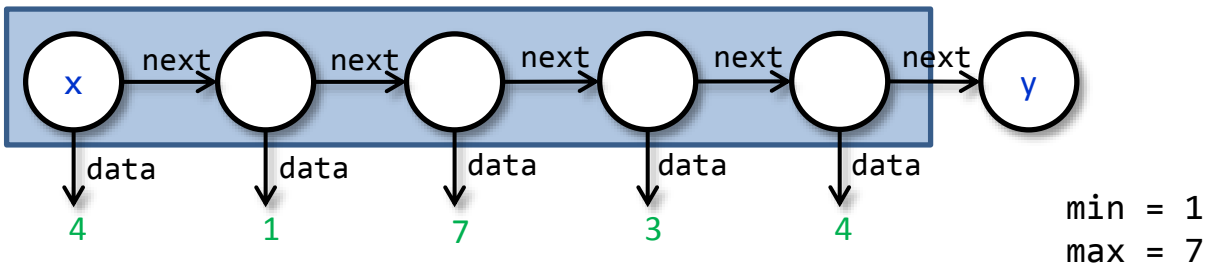
Why Does Completeness Matter?

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;
}
```

Specification of Split

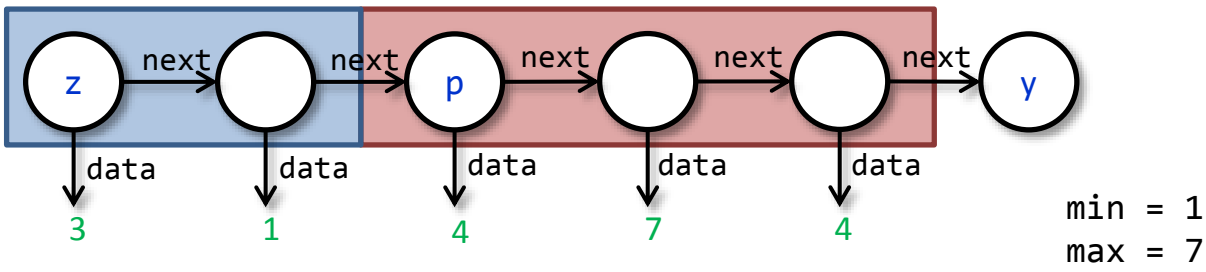
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y;
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max);
  ensures p ≠ y * min ≤ p.data ≤ max;
```



Specification of Split

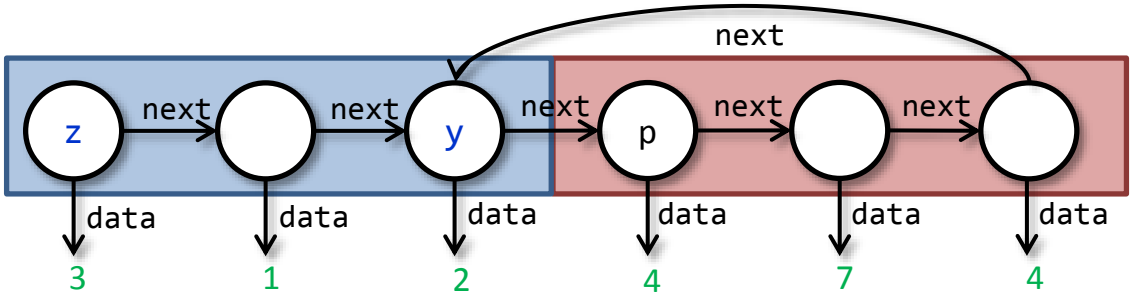
```
procedure split(x: Node, y: Node,
                ghost min: int, ghost max: int)
returns (z: Node, p: Node)
  requires bnd_lseg(x, y, min, max) * x ≠ y;
  ensures bnd_lseg(z, p, min, p.data) *
          bnd_lseg(p, y, p.data, max);
  ensures p ≠ y * min ≤ p.data ≤ max;
```

free memory



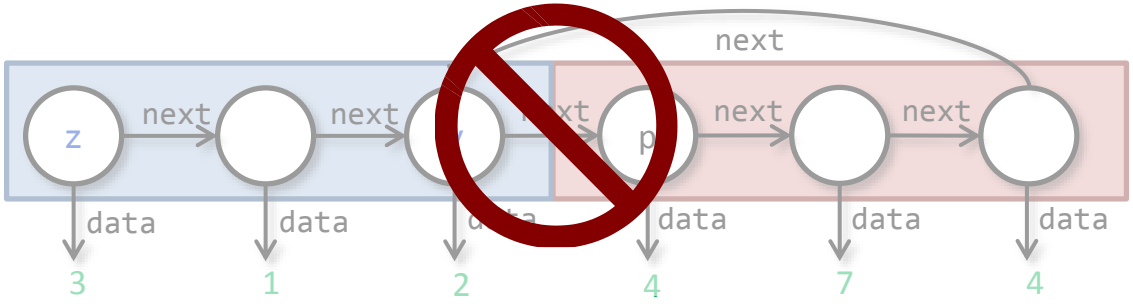
Counterexample for Quicksort Spec.

```
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;
```



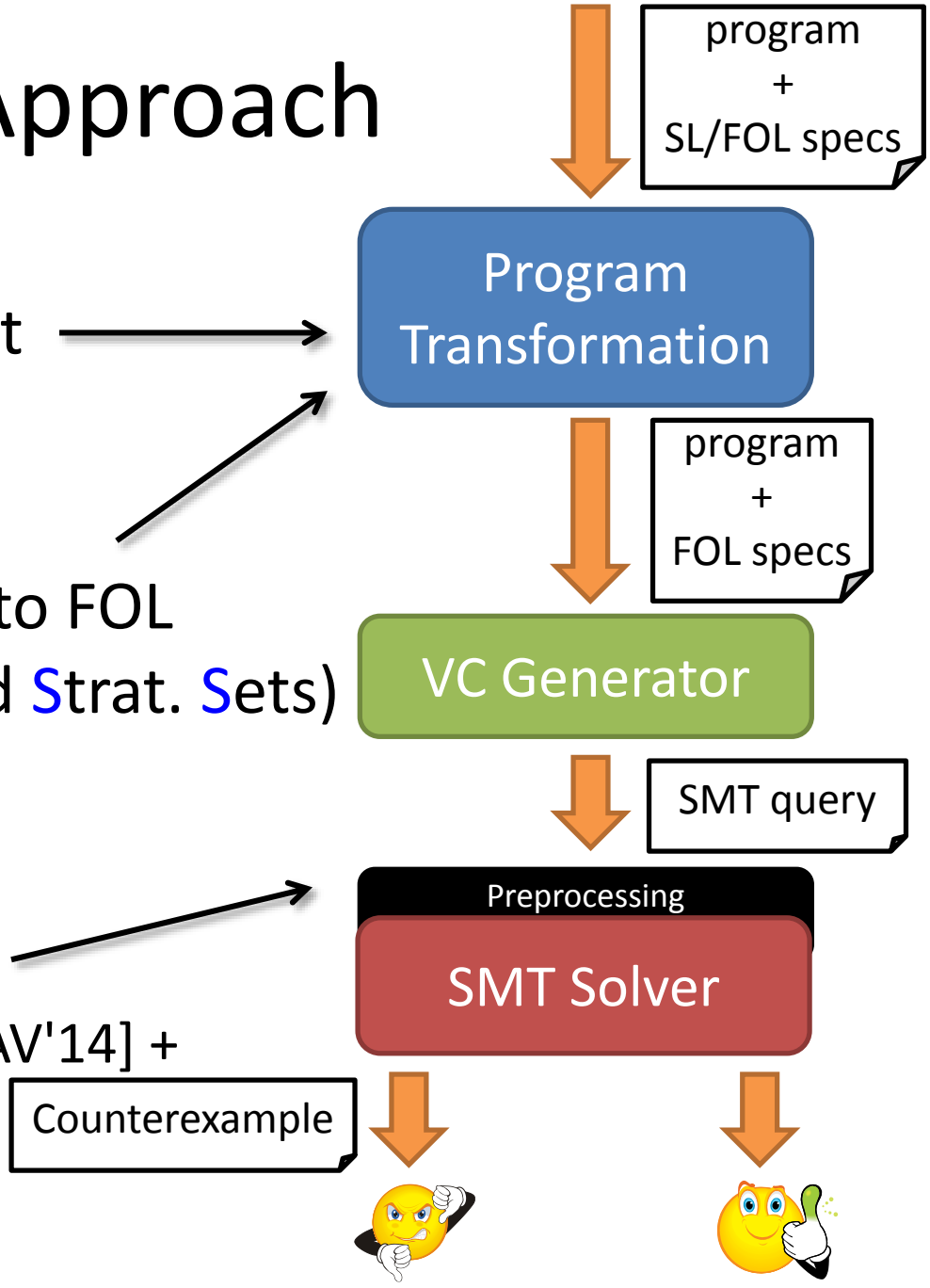
Split with Mixed Specification

```
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,z,p,y);
  ensures p ≠ y * min ≤ p.data ≤ max;
```

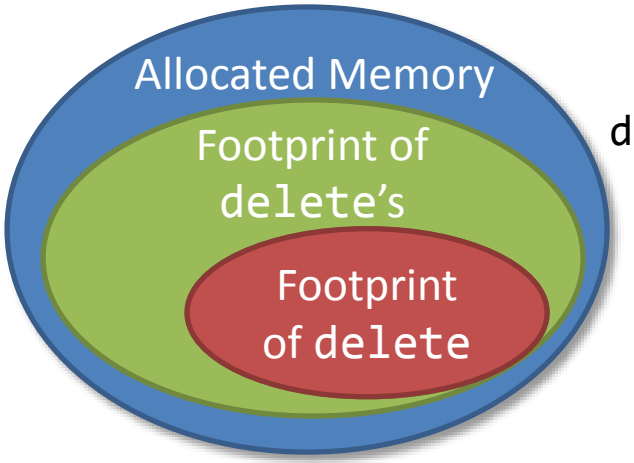


GRASShopper Approach

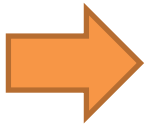
1. Make frame rule explicit
   [TACAS'14]
2. Translate SL assertions to FOL
   (Graph Reachability and Strat. Sets)
   [CAV'13]
3. Decide generated VCs
   [CAV'13] + [TACAS'14] + [CAV'14] +



Encoding the Frame Rule



`delete(x);`

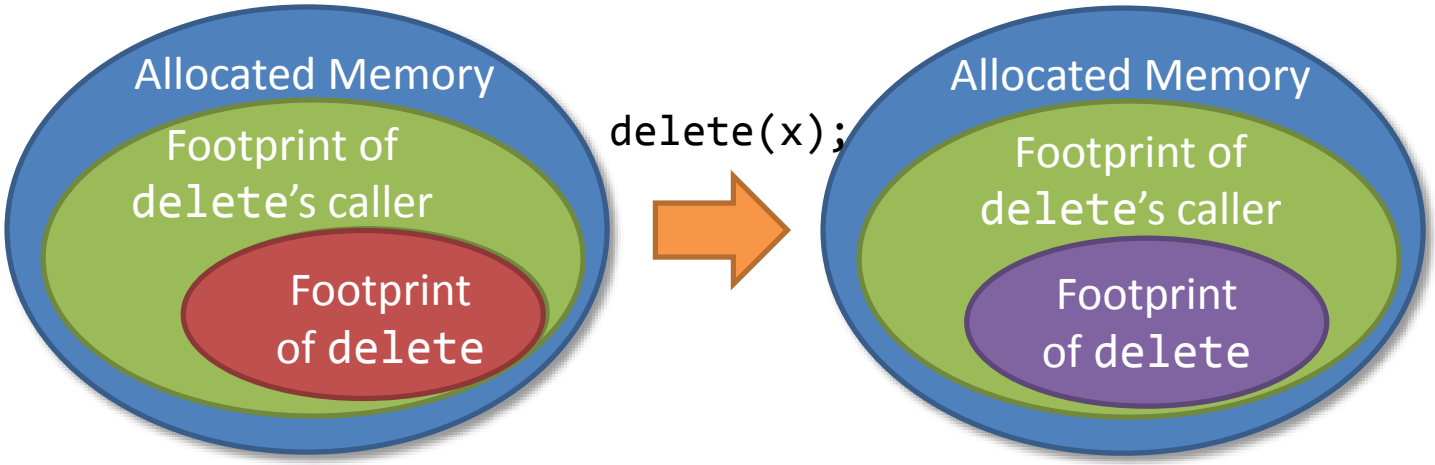


?

before call to `delete`

after call to `delete`

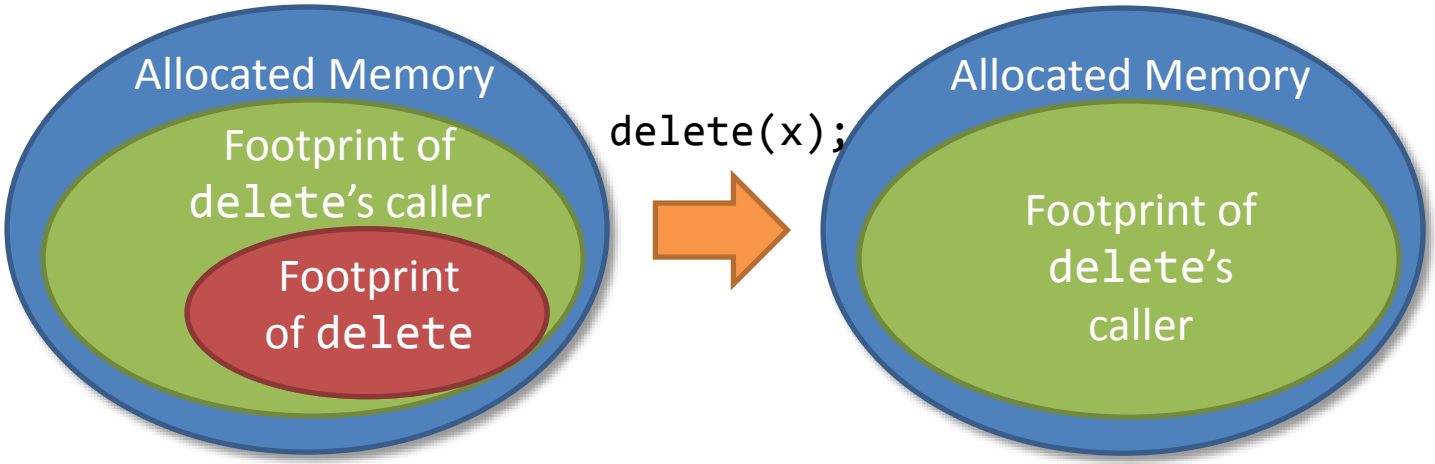
Encoding the Frame Rule



before call to `delete`

after call to `delete`

Encoding the Frame Rule

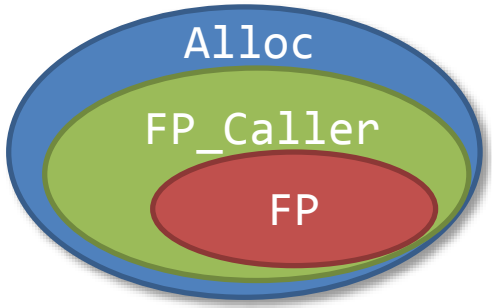


before call to `delete`

after call to `delete`

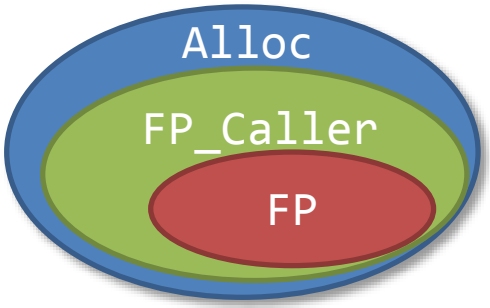
Encoding the Frame Rule

```
procedure delete(x: Node

                                                    )

{

  if (x != null) {

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

}
```



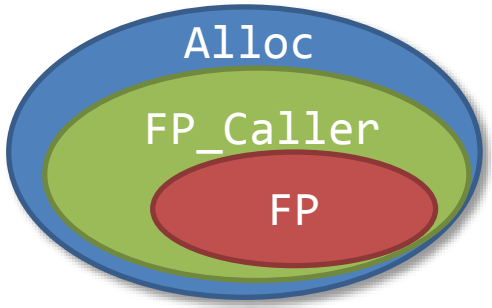
Encoding the Frame Rule

```
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);
  }

}
```



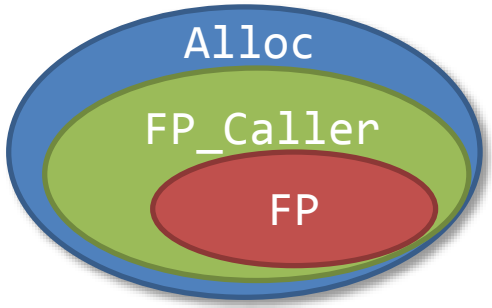
Encoding the Frame Rule

```
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

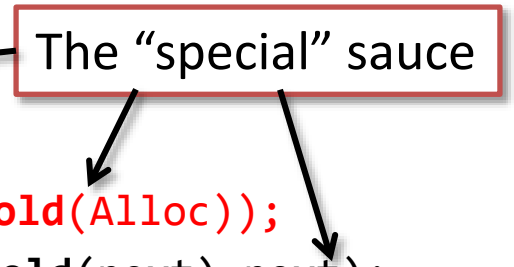
```
procedure delete(x: Node,
                 ghost FP_Caller: set<Node>,
                 implicit ghost FP: set<Node>)
returns (ghost FP_Caller': set<Node>)
  requires FP ⊆ FP_Caller;
  requ
  free
  free
  ensu
  free
next);
  free ensures FP_Caller' = (FP_Caller\FP) ∪
                 (Alloc∩FP)∪(Alloc\old(Alloc));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



Step 2: Translating SL Assertions

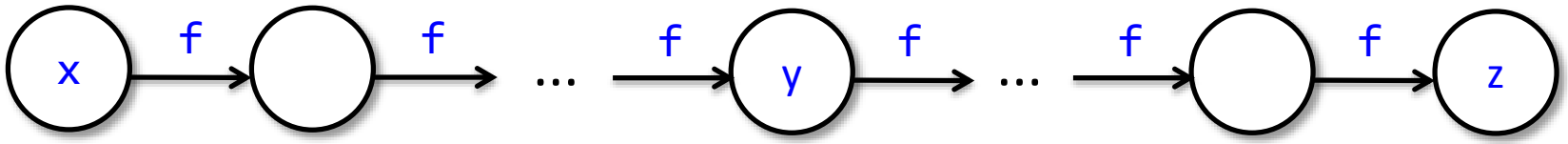
Target of Translation: GRASS (Graph Reachability and Stratified Sets)

- 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])

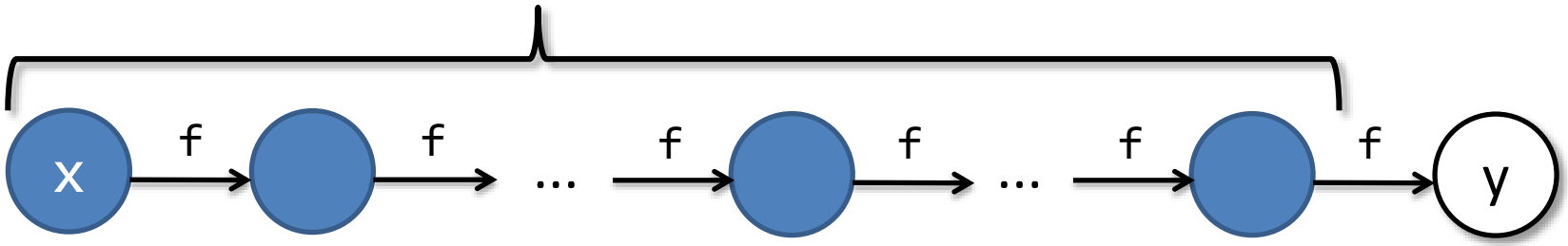
• sel(f, x)	field access	x.f
• upd(f,x,y)	field update	f[x := y]
• Btwn(f,x,y,z)	reachability	$x \xrightarrow{f} y \xrightarrow{f} z$



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

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

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

Permission sets are uniquely determined by formula structure

Quantifiers can be eliminated

First-Order Axioms for Btwn

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

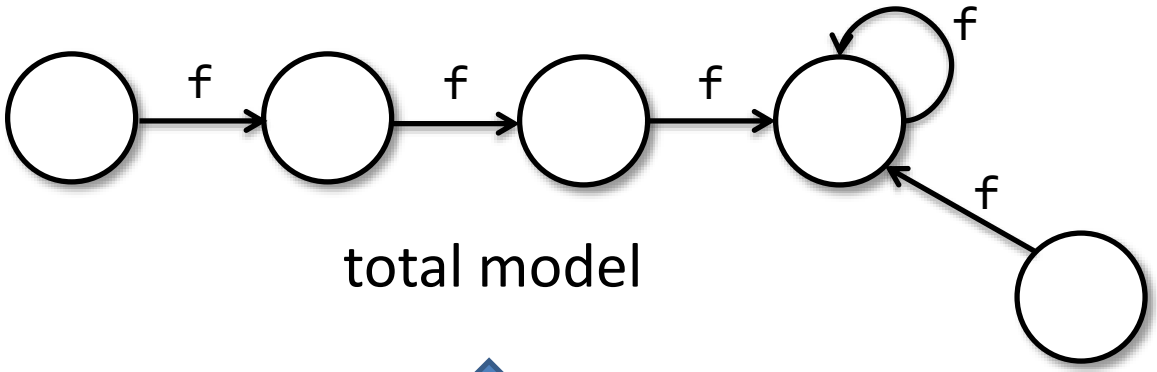
First-Order Axioms for Btwn

- $\forall$ 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)

Almost in EPR!

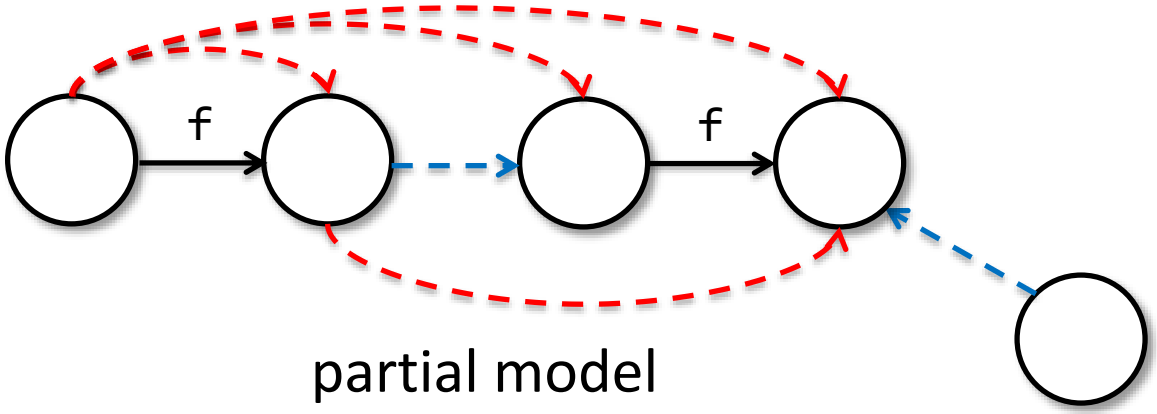
Need to consider more general decidable fragments:
Local Theory Extensions
[Sofronie-Stokkermans, CADE'05], [Bansal et al., CAV'15]

Model Completion for Local Theory Extensions



total model

Yields NP decision procedure for GRASS



partial model

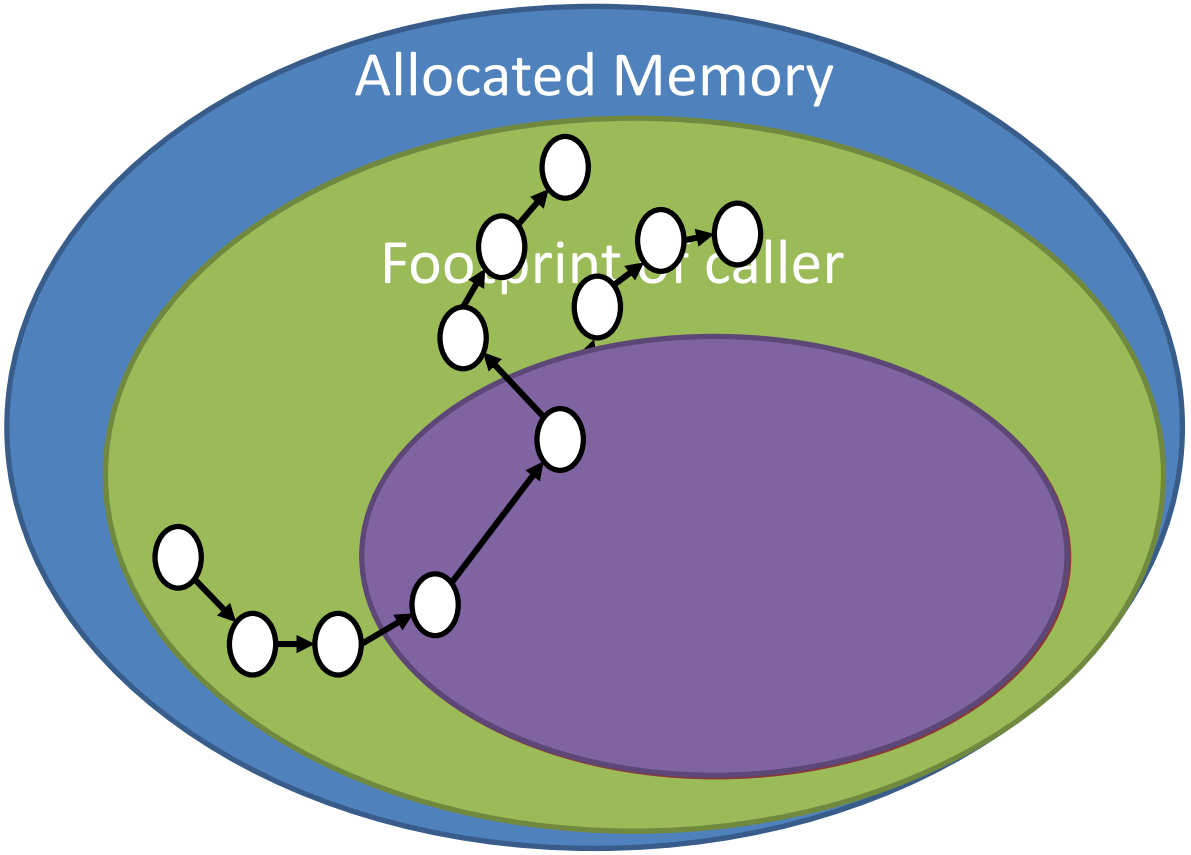
The Frame Predicate

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

- Does not work with finite instantiation.

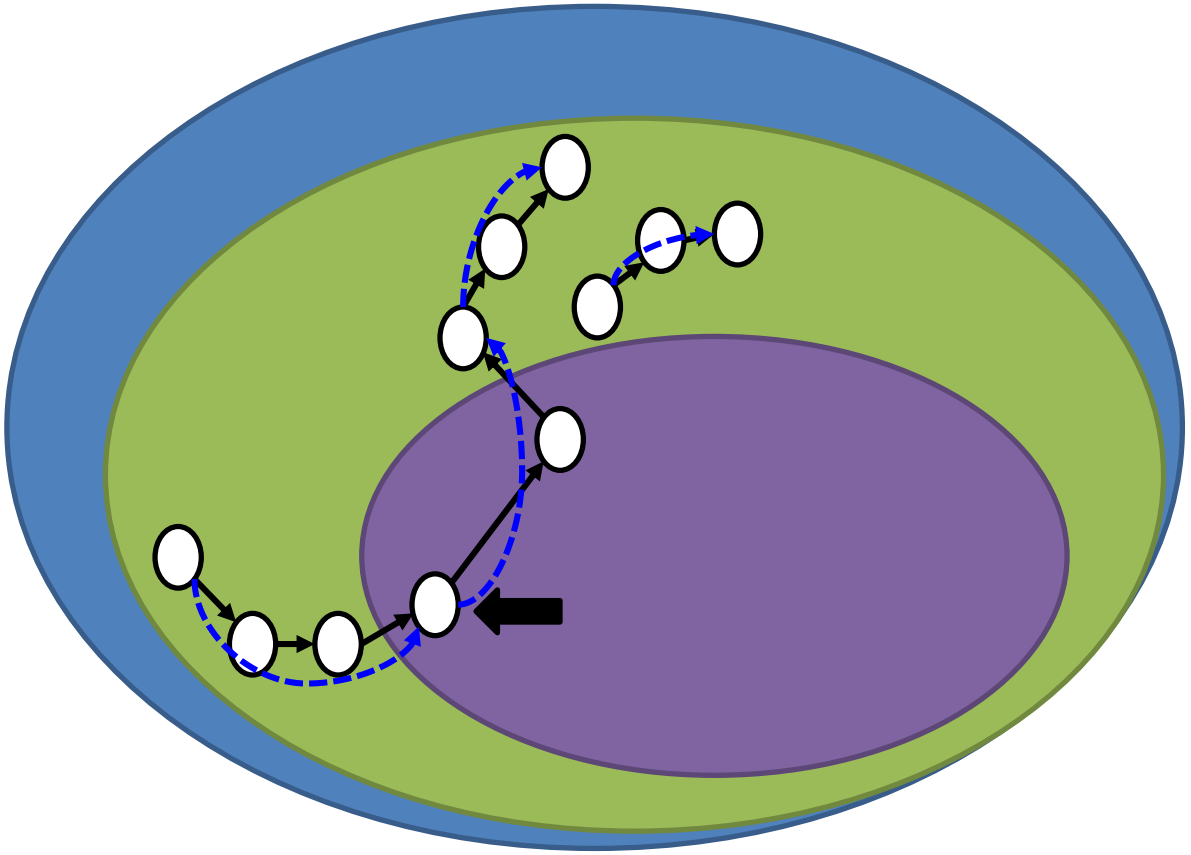
- Need to preserve reachability information in frame.

Axiomatizing the Frame Predicate



Local changes have global effect on reachability

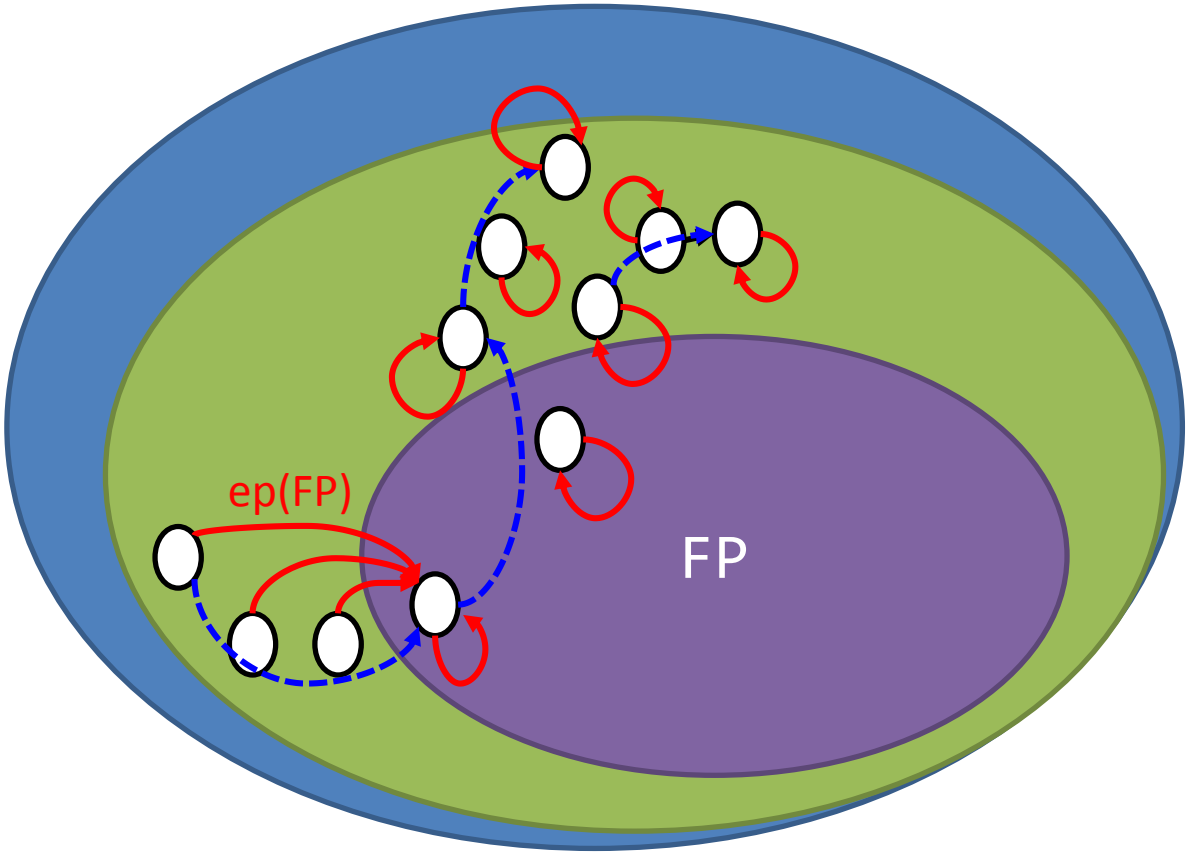
Axiomatizing the Frame Predicate



Local changes have global effect on reachability

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

Axiomatizing the Frame Predicate



Local changes have global effect on reachability

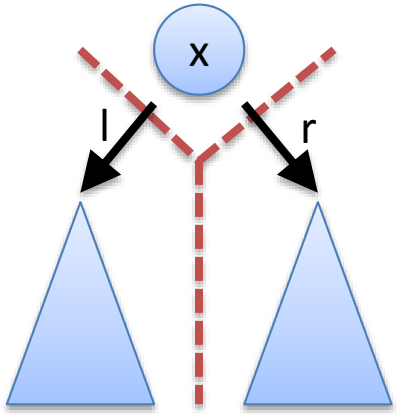
Track **entry points (ep)** into footprint **FP**

Trees in SL

$$Tree(x) \;\coloneqq\; x = null$$
$$\vee \;\; acc(x) * Tree(x.l) * Tree(x.r)$$

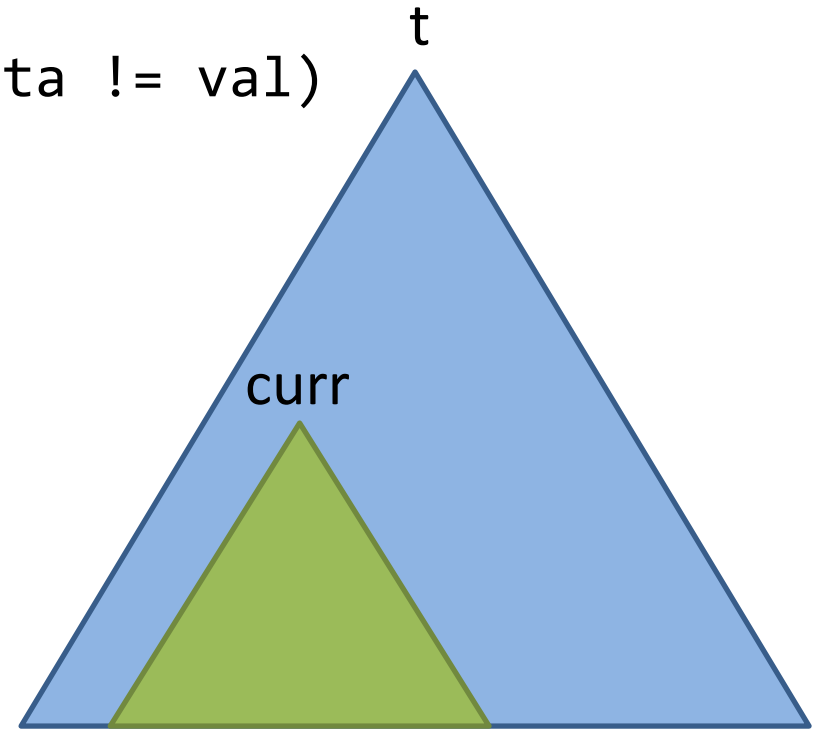
Allocated (access)

Separating conjunction



Tail-Recursive Tree Traversal

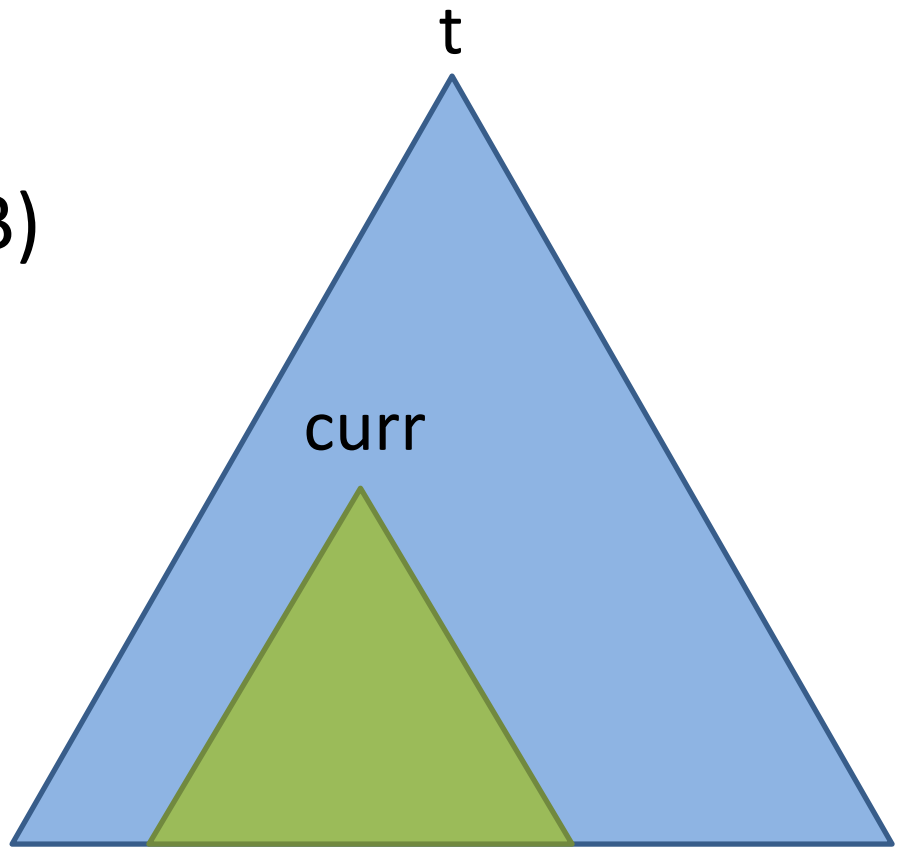
```
procedure contains(t: Tree, val: Int)
  returns (res: Bool)
  requires tree(t)
  ensures tree(t)
{
  var curr := t;
  while (curr != null && curr.data != val)
    invariant ?
  {
    if (curr.data > val)
      curr := curr.left;
    else if (curr.data < val)
      curr := curr.right;
  }
  return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



Poor Man's Magic Wand

$A \mathrel{-\!\!**} B \equiv$

$A * (A \mathrel{-\!\!*} B)$



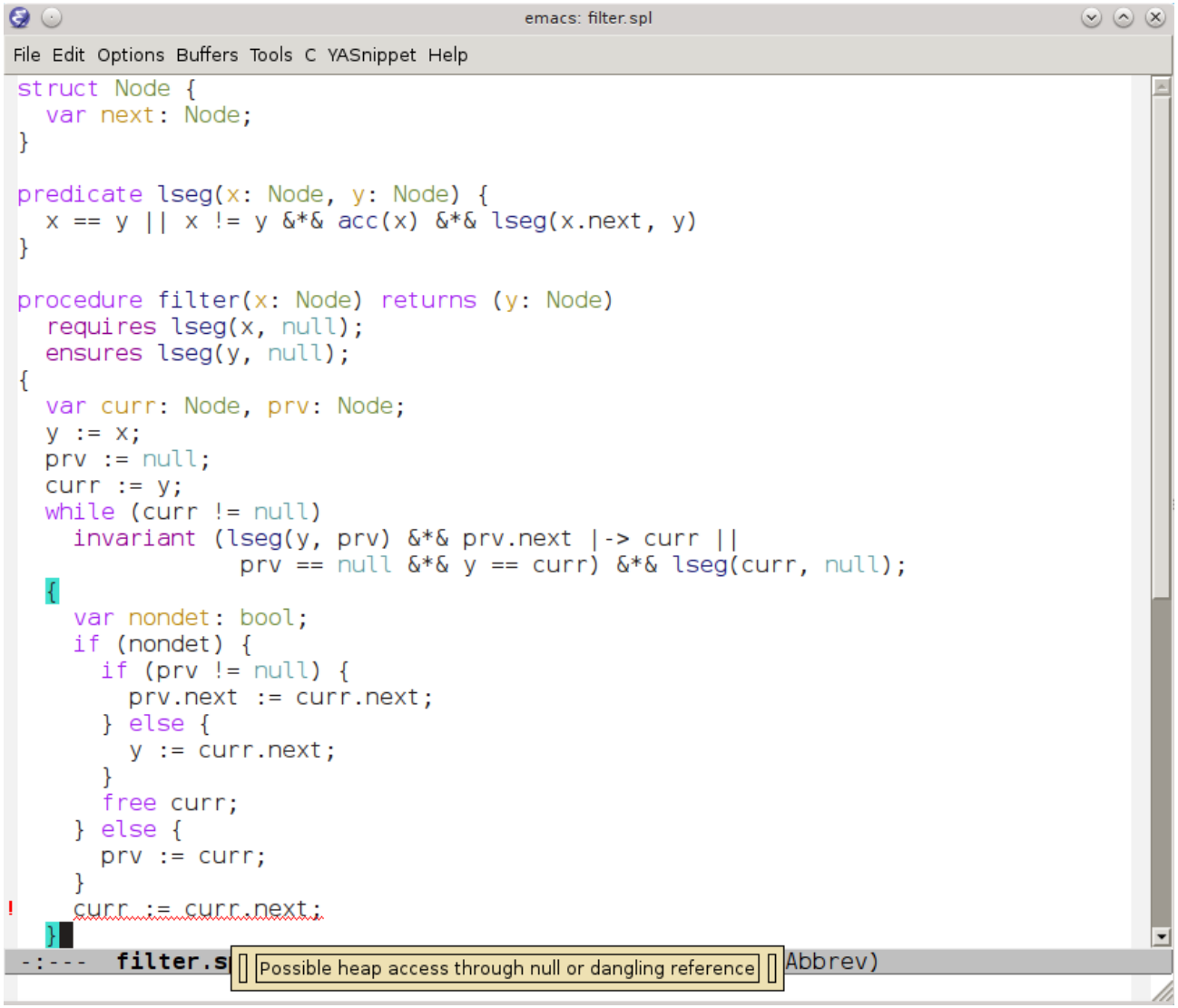

```
tree(curr) * (tree(curr) –* tree(t))
```

Poor Man's Magic Wand

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

GRASShopper

http://github.com/wies/grasshopper



GRASShopper

http://github.com/wies/grasshopper

- Key features
  - C-like language with mixed SL/FOL specifications
  - Compiles to C
  - Supported back-end solvers: Z3 and CVC4
- Benchmarks (several thousand LoC)
  - List data structures
    - Singly/doubly linked, bounded/sorted, with content, …
    - sorting algorithms, set containers, ...
  - Tree data structures (still in NP!)
    - Binary search trees, skew heaps, union/find, ...
  - Arrays

Conclusions

- Translation of SL to FOL

- Trade-off of presented approach
  + adds great expressivity
  + succinct specifications
  - no intuitive proofs
  - heavier machinery

- Broader Message
  - today's SMT solvers can get you a long way
    - far beyond what is natively supported by the solvers
    - without sacrificing decidability or completeness
  - if you use clever encodings
  - need richer SMT interface to deal with quantifiers

FROM VERIFICATION CONDITIONS TO RESEARCH QUESTIONS

Combination of Non-disjoint Logics

```
//: L = data[root.next*]
//: invariant: size = card L
public void add (Object x)
//: ensures L = old L + {x}
{
  Node e = new Node();
  e.data = x;
  e.next = root;
  root = e;
  size = size + 1;
}
```

Verification condition:

$$\neg next_0^*(root_0,n) \wedge x \notin \{data_0(v) \mid next_0^*(root_0,v)\} \wedge next = next_0\,[n:=root_0] \wedge data = data_0[n:=x] \rightarrow |\{data(v) \,.\, next^*(n,v)\}| = |\{data_0(v) \,.\, next_0^*(root_0,v)\}| + 1$$

Combination of Non-disjoint Logics

- Defined new combination technique for theories sharing sets by reduction to a common shared theory (BAPA)

- Identified an expressive decidable set-sharing combination of theories by extending their decision procedures to BAPA-reductions
- **1)** WS2S, **2)** $C^{2,}$ **3)** BSR, **4)** BAPA, **5)** qf-multisets

- Resulting theory is useful for automated verification of complex properties of data structure implementations

Verification of Data Structures

```
def removeDuplicates(l: List[String]): List[String] = {
var c = l
var duplExist = false
var s: List[String] = List()
//:  (duplExist ↔ S ⊎ C ≠  L ) ∧ S ⊎ C ⊆  L
while (! c.isEmpty) {
  val elem = c.first
  if (! s.contains(elem))  s = elem :: s
     else  duplExist = true
  c = c.tail
}
s
//:  duplExist → |S| < |L|
}
```

abstracting lists as multisets

- S *denotes* s
- C *denotes* c
- L *denotes* l

Verification of Data Structures

```
def removeDuplicates(l: List[String]): List[String] = {
var c = l
var duplExist = false
var s: List[String] = List()
//:  (duplExist ↔ S ⊎ C ≠  L ) ∧ S ⊎ C ⊆  L
while (! c.isEmpty) {
  val elem = c.first
  if (! s.contains(elem))  s = elem :: s
     else  duplExist = true
  c = c.tail
}
s
//:  duplExist → |S| < |L|
}
```

abstracting lists as multisets

- S *denotes* s
- C *denotes* c
- L *denotes* l

$$S \uplus C \neq L \wedge S \uplus C \subseteq L \rightarrow |S| < |L|$$

Reasoning about Multisets

Multiset (bag) is a collection of elements where an element can occur several times

Formally, multisets is a function $m : E \rightarrow \{0, 1, 2, \ldots\}$
(E - finite universe of unknown size)

Operations and Relations on Multisets:

- Plus: $(m_1 \uplus m_2)(e) = m_1(e) + m_2(e)$
- Subset: $m_1 \subseteq m_2 \leftrightarrow \forall e.m_1(e) \leq m_2(e)$
- $\forall e.F(m_1(e), \ldots, m_k(e))$, F - QF linear integer arithmetic formula
- Cardinality: $|m| = \Sigma_{e \in E}\, m(e)$

Decision Procedure - Overview

$$|X \uplus Y| \neq |X| + |Y|$$

$$m_0 = X \uplus Y,\ k_0 = |m_0|,\ k_1 = |Y|,\ k_2 = |X| = \Sigma_{e \in E}\, X(e)$$

$$k_0 \neq k_1 + k_2 \wedge (k_0, k_1, k_2) = \Sigma_{e \in E}(m_0(e), Y(e), X(e)) \wedge \forall e.m_0(e) = Y(e) + X(e)$$

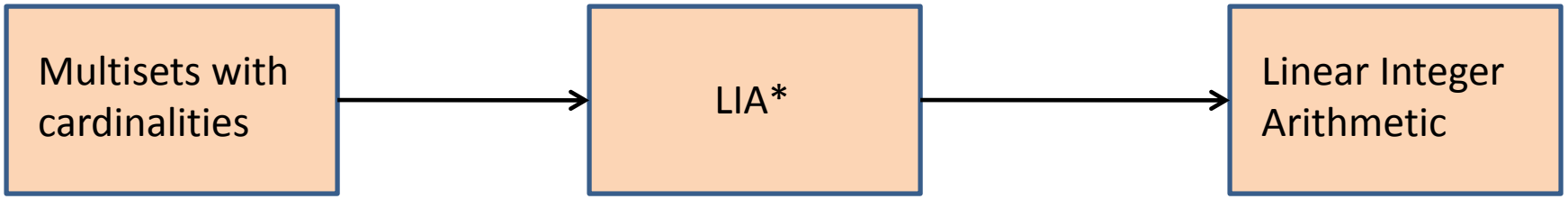
Sum Normal Form

$$P \wedge (u_1, \ldots, u_n) = \Sigma_{e \in E}(t_1, \ldots, t_n) \wedge \forall e.F$$

Every multiset formula can be reduced to its sum normal form.

How can we automatically reason about such sums?

Introducing: LIA*



For a given set of integer vectors S,

$$S^* = \{x_1 + \ldots + x_n \mid x_i \in S \wedge n \geq 0\}$$

LIA∗ Formulas

$$P \wedge (u_1, \ldots, u_n) \in \{(t_1, \ldots, t_n) \mid F; x_1, \ldots, x_p \in N\}^*$$

where P, F are quantifier-free linear integer arithmetic formulas

From Multisets to LIA*

A formula in the sum normal form:

$$P \wedge (u_1, \dots, u_n) = \Sigma_{e \in E}(t_1, \dots, t_n) \wedge \forall e.F$$

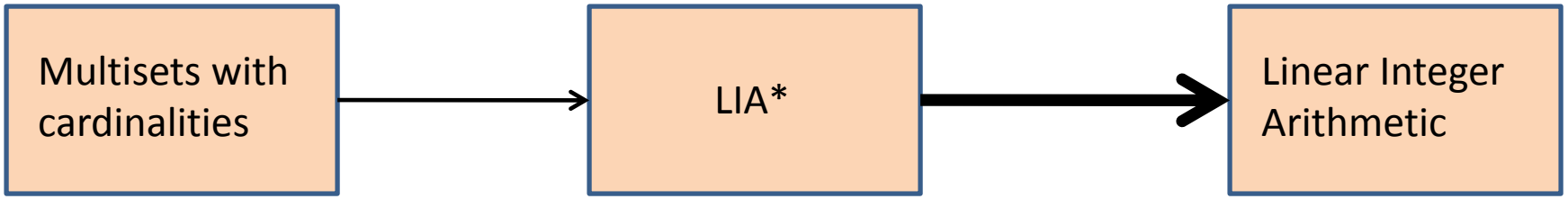
is equisatisfiable with the formula

$$P \wedge (u_1, \dots, u_n) \in \{(t'_1, \dots, t'_n) \mid F; x_1, \dots, x_p \in N\}^*$$

Example (continued):

$$(k_0, k_1, k_2) = \Sigma_{e \in E}(m_0(e), Y(e), X(e)) \wedge \forall e.m_0(e) = Y(e) + X(e)$$
$$(k_0, k_1, k_2) \in \{(m_0, y, x) \mid m_0 = y + x;\ m_0, y, x \in N\}^*$$

Semilinear Sets



Let C_1 and C_2 be sets of vectors of non-negative integers.

$$C_1 + C_2 = \{x_1 + x_2 \mid x_1 \in C_1 \wedge x_2 \in C_2\}$$

Linear set = set of form $\{x\} + C^*$ for $x \in \mathbb{N}^n$ and $C \subseteq \mathbb{N}^n$ finite

Semilinear set = finite union of linear sets

$\{2\} + \{10\}^* = \{2, 12, 22, \ldots\} = \{x \mid x = 2 + 10n, n \in \mathbb{N}\}$

$\{(3, 3)\} + \{(0, 1), (1, 1)\}^* = \{(x, y) \mid 3 \leq x \wedge x \leq y\}$

From LIA* to LIA

- [Ginsburg,Spanier1968] showed:
  - a solution of a LIA formula is a semilinear set
- Observation:
  - if S is semilinear, then S* can be described in LIA
  - thus, S* is also semilinear
- Consequently:

$$(u_1, \ldots, u_n) \in \{(t_1, \ldots, t_n) \mid F\}^*$$

is equivalent to a LIA formula

Complete Reduction Process

- Starting formula: $|X \uplus Y| \neq |X| + |Y|$
- LIA* translation:

$$k_0 \neq k_1 + k_2 \wedge (k_0, k_1, k_2) \in \underbrace{\{(m_0, y, x) \mid m_0 = y + x;\ m_0, y, x \in N\}^*}_{S^*}$$

- semilinear set describing S:

$S = (0, 0, 0) + \{(1, 0, 1), (1, 1, 0)\}^*$

$(k_0, k_1, k_2) \in S^*$ becomes: $(k_0, k_1, k_2) = \lambda(1, 0, 1) + \mu(1, 1, 0)$

- Final result:

$$k_0 \neq k_1 + k_2 \wedge k_0 = \lambda + \mu \wedge k_1 = \lambda \wedge k_2 = \mu$$

Complete Reduction Process

- Starting formula: $|X \uplus Y| \neq |X| + |Y|$
- Final result:
  - $k_0 \neq k_1 + k_2 \wedge k_0 = \lambda + \mu \wedge k_1 = \lambda \wedge k_2 = \mu$
- After elimination of of λ and μ:
  - $k_0 \neq k_1 + k_2 \wedge k_0 = k_1 + k_2$
- The resulting formula is UNSAT, which means that the original formula is also UNSAT

Summary of Decision Procedure

- Summary:
  - reduce multiset formula to a form $P \wedge u \in \{z \mid F\}^*$
  - find semilinear set $\bigcup_{i=1} (a_i + \{b_{i1}, \ldots, b_{ini}\}^*)$ *for F*
  - use a_i , b_{ij} *to construct formula* F_1 *describing* $u \in \{z \mid F\}^*$ *:*

$$\exists \mu_i, \lambda_{ij} \,.\, (u_1, \ldots, u_n) = \Sigma_{i=1}(\mu_i a_i + \Sigma_{j=1} \lambda_{ij} b_{ij}) \wedge$$

$$\wedge_{i=1}(\mu_i = 0 \Rightarrow \Sigma_{j=1} \lambda_{ij} = 0)$$

  - check satisfiability of formula $P \wedge F_1$

Complexity too High?

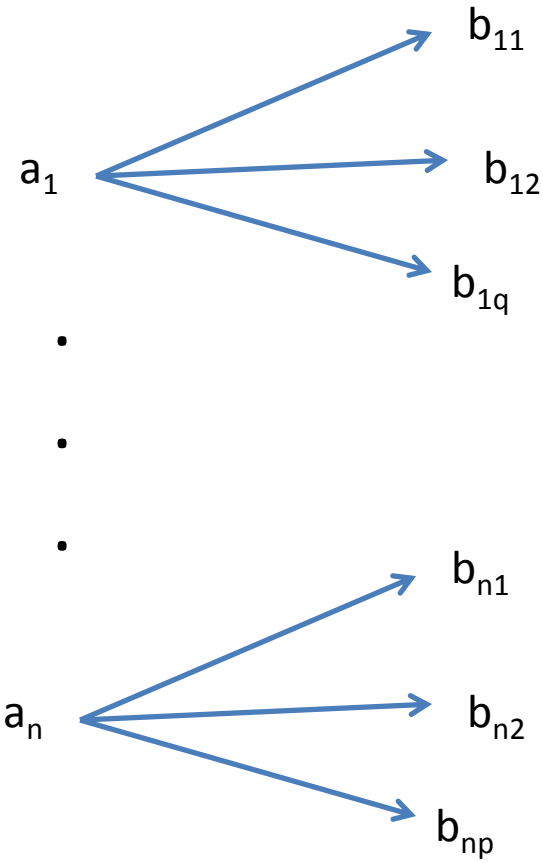
Problems:

• there can be exponentially many generators of semilinear sets → NEXPTIME decision procedure

• computing semilinear sets is hard

Solution:

• if u is generated by a_i , b_{ij} , then is generated by polynomial subset of them



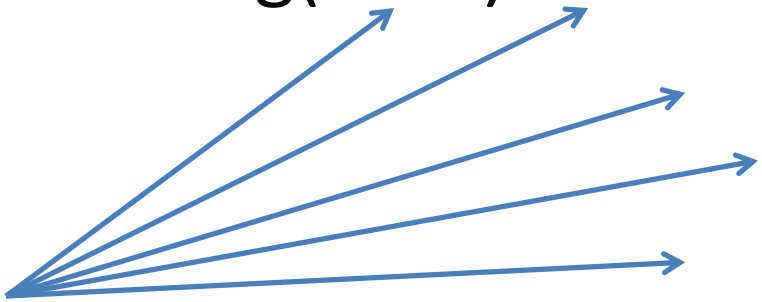
Caratheodory Theorem For Integer Cones

Theorem (Eisenbrand, Shmonin)

Let $X \subseteq Z^d$ be a finite set of integer vectors and let $b \in X^*$.

Then there exists $Y \subseteq X$ such that $b \in Y*$ and

$|Y| \leq 2d \log(4dM)$ where $M = \max_{x \in X} ||x||_\infty$.



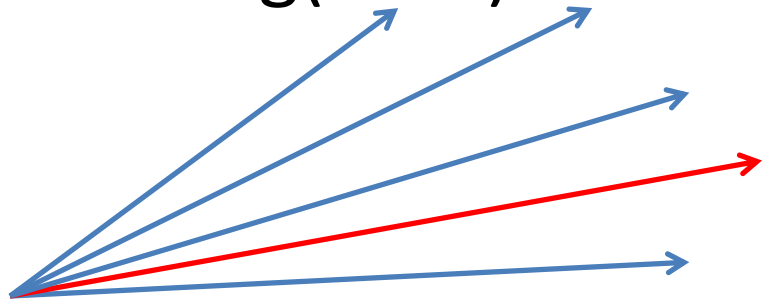
Caratheodory Theorem For Integer Cones

Theorem (Eisenbrand, Shmonin)

Let $X \subseteq Z^d$ be a finite set of integer vectors and let $b \in X^*$.

Then there exists $Y \subseteq X$ such that $b \in Y*$ and

$|Y| \leq 2d \log(4dM)$ where $M = \max_{x \in X} ||x||_\infty$.



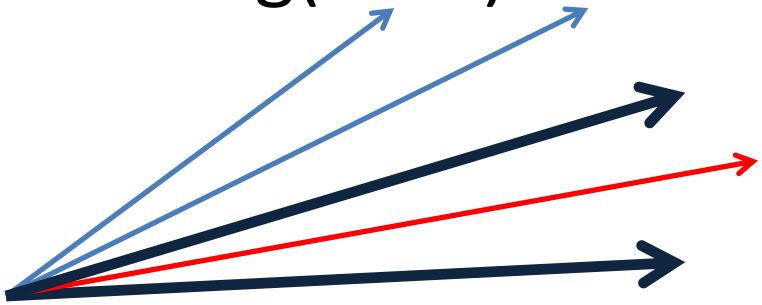
Caratheodory Theorem For Integer Cones

Theorem (Eisenbrand, Shmonin)

Let $X \subseteq Z^d$ be a finite set of integer vectors and let $b \in X^*$.

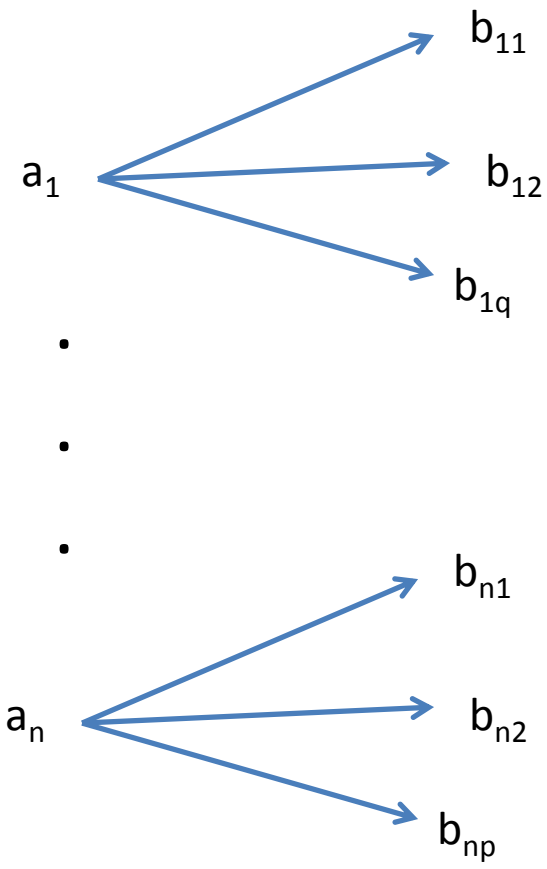
Then there exists $Y \subseteq X$ such that $b \in Y*$ and

$|Y| \leq 2d \log(4dM)$ where $M = \max_{x \in X} ||x||_\infty$.



Small Number of Generators

1. u = a + b
2. apply Eisenbrand-Shmonin theorem as black box on b_{ij} vectors
3. there are only polynomially vectors b_{ij} needed to represent b
4. join them with associated a_i vectors
5. apply Eisenbrand-Shmonin theorem on remaining a_i vectors



How to Guess a_i and b_{ij} Vectors?

- Observation: instead of guessing a_i , b_{ij} , *guess solutions of formula F*
- Result:

$$P \wedge u \in \{x \mid F(x)\}^*$$

is equisatisfiable with

$$P \quad \wedge \quad u = \Sigma_{i\in\{1,\ldots,Q\}} \lambda_i v_i \quad \wedge \quad \Lambda_{i\in\{1,\ldots,Q\}} F(v_i)$$

where $Q \in \mathcal{O}(n^2 \log n)$ *for formula of a size n*

Number of Needed Solutions

- In formula

$$P \quad \wedge \quad u = \Sigma_{i \in \{1,\ldots,Q\}} \lambda_i v_i \quad \wedge \quad \Lambda_{i \in \{1,\ldots,Q\}} F(v_i)$$

Q is a number (not a variable!) It depends on

- dimension of a problem
- $||_\infty$ of generating vectors of semilinear sets
- their size can be computed without actually computing vectors
- Pottier determined the upped bounds on generators of semilinear sets [Pottier 1991]

Last Hurdle

- Formula

$$P \wedge u = \Sigma_{i \in \{1,\ldots,Q\}} \lambda_i v_i \wedge \bigwedge_{i \in \{1,\ldots,Q\}} F(v_i)$$

is polynomially large formula:

  - but it multiplies variables λ_i , v_i - not linear?
  - solution vectors are bounded [Papadimitriou 1981]
  - multiplication can be expanded using the ite construct

- Result: **NP completeness**