Symbolic Shape Analysis

Andreas Podelski, Thomas Wies

University of Freiburg, Germany
{podelski,wies}@informatik.uni-freiburg.de

```
class SortedList {
  private static Node first;
  /*: public static specvar content :: objset;
      vardefs "content == {v. v ≠ null ∧ next* first v}";
      invariant "tree [next]";
      invariant "∀ v. v ∈ content ∧ v.next ≠ null
                              ⟶ v..Node.data ≤ v.next.data"; */
  public static void insert(Node n)
    /*: requires "n ≠ null ∧ n ∉ content"
        modifies content
        ensures "content = old content ∪ {n}" */
  {
    Node prev = null;
    Node curr = first;
    while ((curr != null) && (curr.data < n.data)) {
      prev = curr;
      curr = curr.next;
    }
    n.next = curr;
    if (prev != null) prev.next = n;
    else first = n;
  }
}
```

Shape Analysis à la SRW

- States are graphs
- Define partitioning of nodes through node predicates
- Abstract states are graphs of abstract nodes
- Abstract nodes are equivalence classes of concrete nodes

Predicate Abstraction

- Take transition graph (nodes are states)
- Define partitioning of nodes through state predicates
- Abstract transition graph is graph of abstract nodes
- Abstract nodes are equivalence classes of concrete nodes

$$\text{shape analysis} = 2^{\text{predicate abstraction}}$$

Why go symbolic?

Apply not only idea, but also techniques of predicate abstraction.

Shape Analysis is tough!

it is not about constructing a finite abstraction of the transition graph (whose nodes are finite abstractions of graphs)

it is about constructing an abstraction of the post operator, i.e. of a transformer of infinite sets of graphs

Generic Benefits of Predicate Abstraction

- use formulae to represent infinite sets of states
  - no need to define meaning of abstract values
  - abstract domain $\subseteq$ concrete domain
  - abstraction $=$ entailment $\models$
  - logical operators more rich than lattice operators

Generic Benefits of Predicate Abstraction

- use formulae to represent infinite sets of states
  - no need to define meaning of abstract values
  - abstract domain $\subseteq$ concrete domain
  - abstraction $=$ entailment $\models$
  - logical operators more rich than lattice operators
- use reasoning procedures
  - automation
  - separation of concerns (black-boxing)
  - soundness by construction, loss of precision identifiable
  - get leverage from theorem proving and formal methods
  - abstraction $=$ provable entailments $\vdash$

Generic Benefits of Predicate Abstraction

- use formulae to represent infinite sets of states
  - no need to define meaning of abstract values
  - abstract domain $\subseteq$ concrete domain
  - abstraction $=$ entailment $\models$
  - logical operators more rich than lattice operators
- use reasoning procedures
  - automation
  - separation of concerns (black-boxing)
  - soundness by construction, loss of precision identifiable
  - get leverage from theorem proving and formal methods
  - abstraction $=$ provable entailments $\vdash$
- abstraction refinement
  - more automation
  - symbolic execution of counter-examples
  - abstract domain $\subset$ refined abstract domain

How can we make shape analysis symbolic?

- Which class of formulae to represent infinite sets of graphs?
- Can we construct an abstract post by defining it locally, *i.e.* on node predicates P?

$$\text{formula} \models \text{weakest precondition}(P)$$

- What is a predicate transformer for node predicates?
- Can we again use Cartesian abstraction?
- Should we again use Cartesian abstraction?

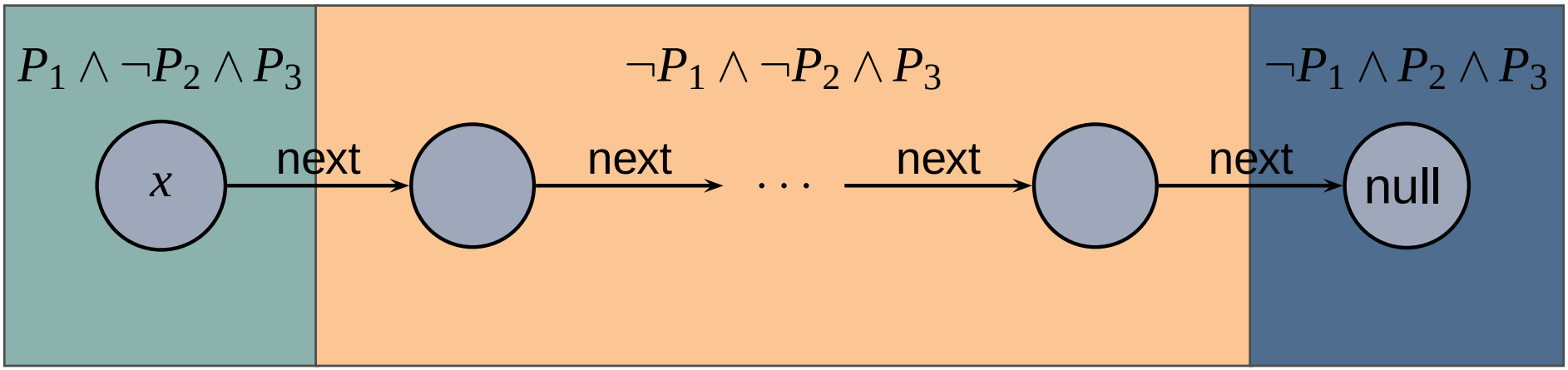
Outline

1. Boolean heaps (abstract domain)
2. Cartesian post (abstract transformer)
3. Abstraction refinement
4. Bohne - implementation of symbolic shape analysis

Boolean Heaps

Partition heap according to finitely many predicates on heap objects.

$$P_1 = \{\, v \mid v = x \,\} \quad P_2 = \{\, v \mid v = \mathsf{null} \,\} \quad P_3 = \{\, v \mid \mathsf{next}^*(x, v) \,\}$$



Describe partitioning as a universally quantified formula

$$\forall v \,.\, P_1 \wedge \neg P_2 \wedge P_3 \;\vee\; \neg P_1 \wedge \neg P_2 \wedge P_3 \;\vee\; \neg P_1 \wedge P_2 \wedge P_3$$

➜ Boolean heaps

Abstract domain = {sets of Boolean heaps}

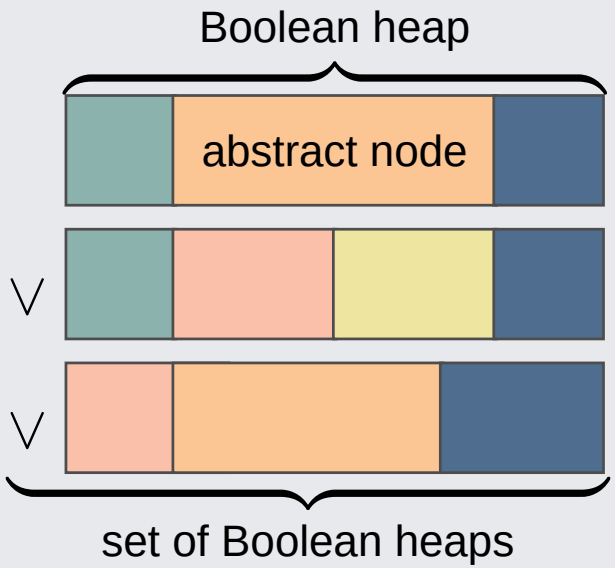
Abstract element

$$\bigvee_i \forall v \,.\, \bigvee_j \bigwedge_k P_{i,j,k}(v)$$

$\bigwedge_k P_{i,j,k}(v)$: abstract node

$\bigvee_j \bigwedge_k P_{i,j,k}(v)$: Boolean heap

$\bigvee_i \forall v \,.\, \bigvee_j \bigwedge_k P_{i,j,k}(v)$: set of Boolean heaps



Symbolic shape analysis

$$\underbrace{\underbrace{\bigvee_i \forall v \,.\, \bigvee_j \underbrace{\bigwedge_k P_{i,j,k}(v)}_{\text{abstract node}}}_{\text{Boolean heap}}}_{\text{set of Boolean heaps}}$$

➜ sets of sets of bit-vectors

Predicate abstraction

$$\underbrace{\underbrace{\bigvee_i \bigwedge_j P_{i,j}}_{\text{abstract state}}}_{\text{sets of abstract states}}$$

➜ sets of bit-vectors

➜ Boolean heaps provide extra precision needed for shape analysis.

Abstract Post on Boolean Heaps

How to compute abstract post on Boolean heaps?

$$\text{post}^{\#}(H) = ?$$

Abstract Post on Boolean Heaps

How to compute abstract post on Boolean heaps?

$$\mathsf{post}^{\#}(H) = \alpha \circ \mathsf{post} \circ \gamma(H)$$

Abstract Post on Boolean Heaps

How to compute abstract post on Boolean heaps?

$$\text{post}^{\#}(H) = \alpha \circ \text{post} \circ \gamma(H)$$

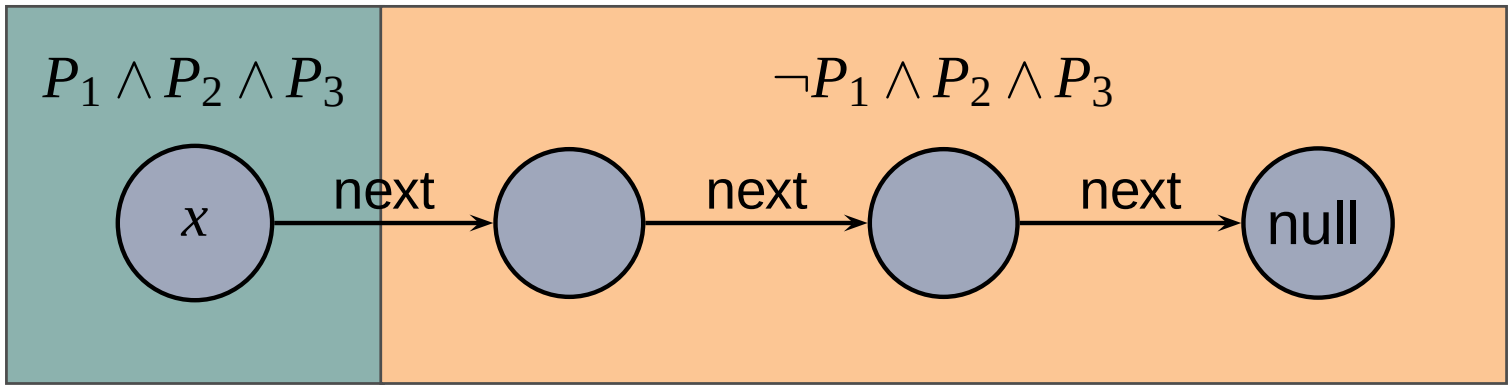
$$\text{post}^{\#}_{\text{Bohne}} = \text{clean} \circ \text{CartesianPost} \circ \text{split}$$

- split $\approx$ focus
- clean $\approx$ coerce

Next slides: CartesianPost.

Abstract Post on Boolean Heaps

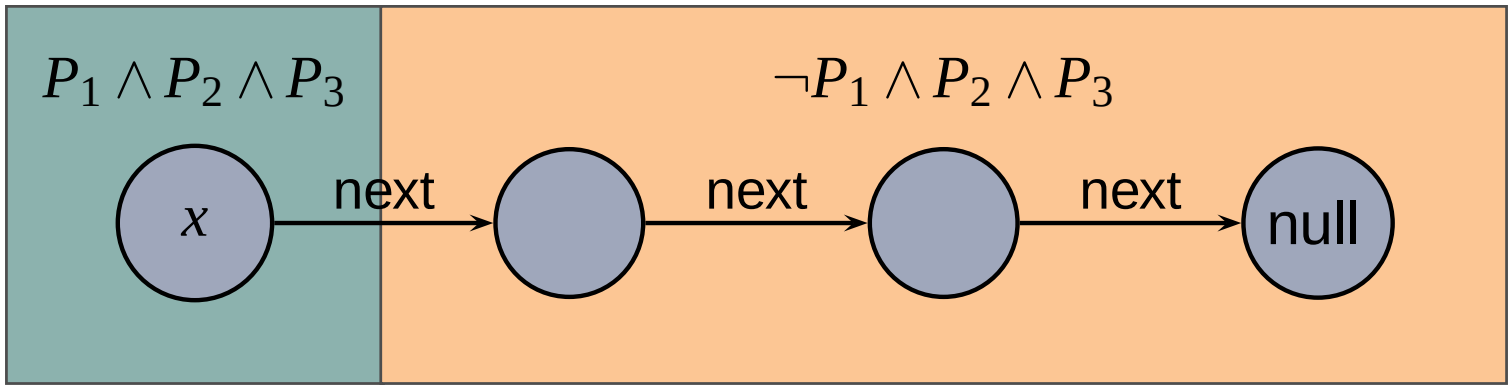
$$P_1 = \{\, v \mid v = x \,\} \quad P_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P_3 = \{\, v \mid \mathsf{next}^*(x, v) \,\}$$



$$\forall v \,.\, P_1 \wedge P_2 \wedge P_3 \;\vee\; \neg P_1 \wedge P_2 \wedge P_3$$

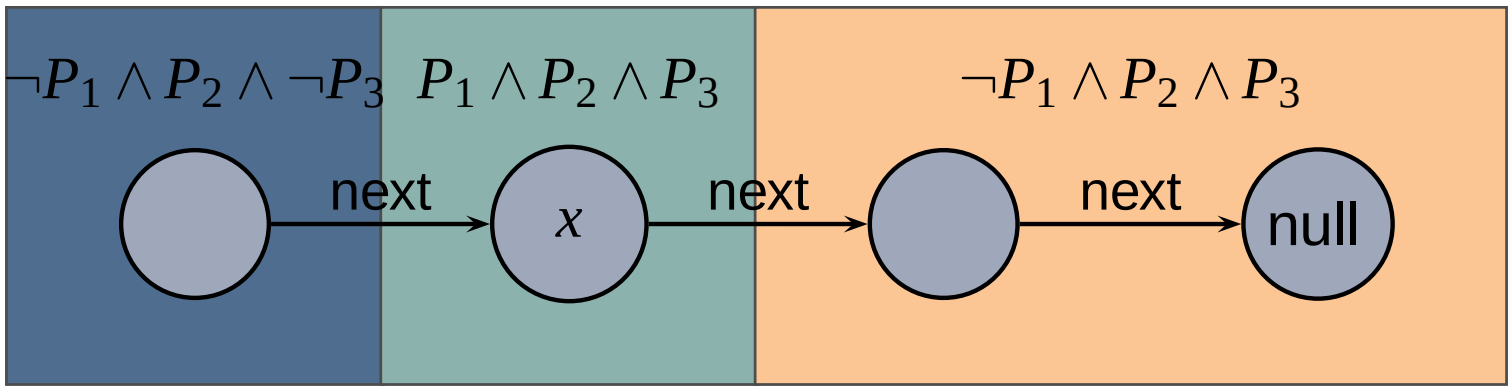
Abstract Post on Boolean Heaps

$P_1 = \{ v \mid v = x \}$ $P_2 = \{ v \mid \mathsf{next}^*(v, \mathsf{null}) \}$ $P_3 = \{ v \mid \mathsf{next}^*(x, v) \}$



$\alpha \circ \mathsf{post}_c \circ \gamma(\forall v \,.\, P_1 \wedge P_2 \wedge P_3 \;\vee\; \neg P_1 \wedge P_2 \wedge P_3)$

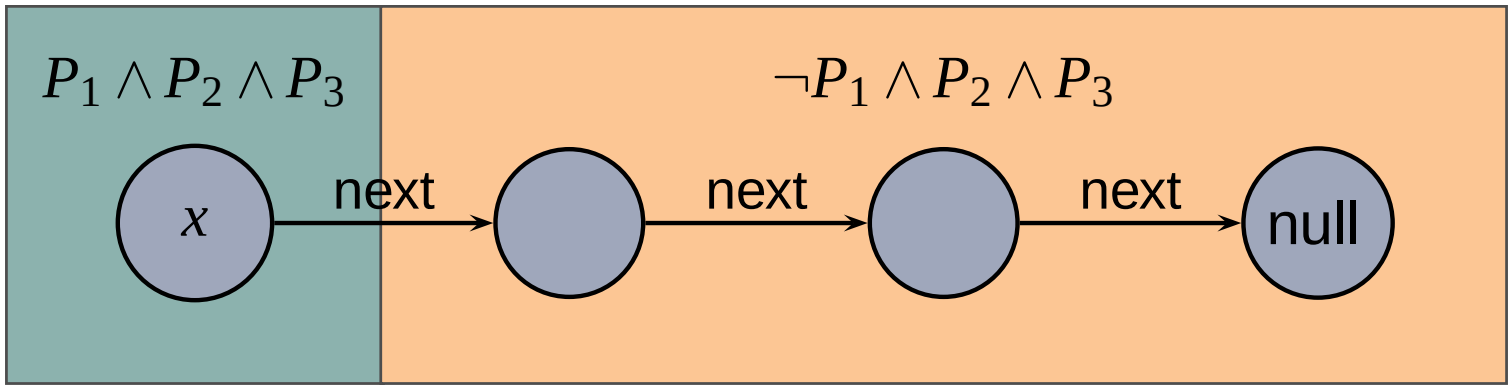
for command $c = (\texttt{x:=x.next})$



$\forall v \,.\, \neg P_1 \wedge P_2 \wedge \neg P_3 \;\vee\; P_1 \wedge P_2 \wedge P_3 \;\vee\; \neg P_1 \wedge P_2 \wedge P_3$

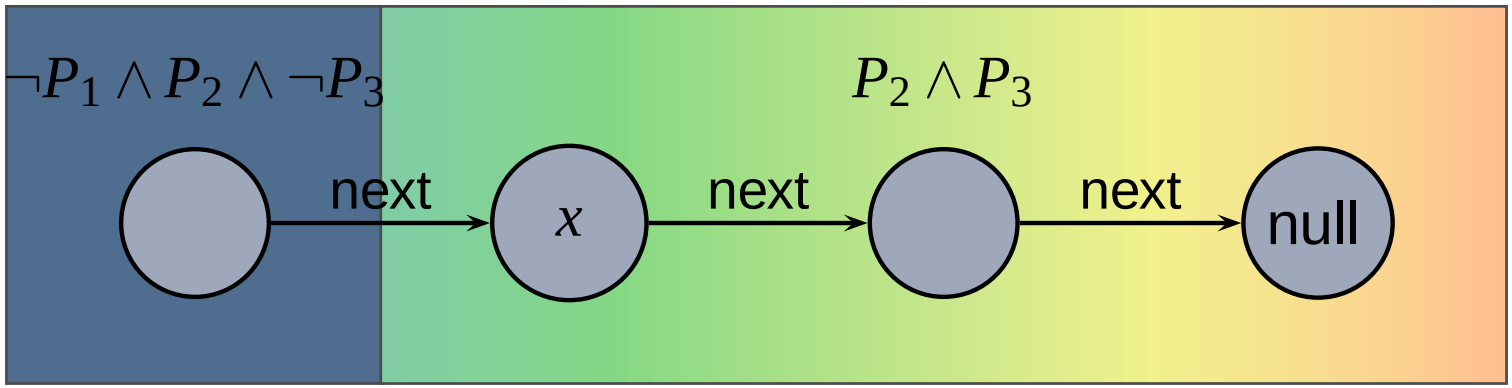
Abstract Post on Boolean Heaps

$$P_1 = \{\, v \mid v = x \,\} \quad P_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P_3 = \{\, v \mid \mathsf{next}^*(x, v) \,\}$$



$\mathsf{CartesianPost}_c(\forall v \,.\, P_1 \wedge P_2 \wedge P_3 \;\vee\; \neg P_1 \wedge P_2 \wedge P_3)$

for command $c = (\texttt{x:=x.next})$



$$\forall v \,.\, \neg P_1 \wedge P_2 \wedge \neg P_3 \;\vee\; P_2 \wedge P_3$$

Cartesian Post

$$\mathsf{CartesianPost}(\forall v\,.\,\bigvee_i C_i)$$

$$= \forall v\,.\,\bigvee_i \bigwedge \{\, P \mid C_i \models \mathsf{wlp}(P) \,\}$$

Cartesian Post

Compute effect of heap updates locally

- for each abstract object C_i
- and independently for each heap predicate P

Cartesian Post

$$\text{CartesianPost}(\forall v\,.\,\bigvee_i C_i)$$

$$= \forall v\,.\,\bigvee_i \bigwedge \{\, P \mid C_i \models \text{wlp}(P) \,\}$$

In practice: precompute abstract weakest preconditions

$$\text{wlp}^{\#}(P) = \bigvee \{\, \phi \in \text{BoolExp}(\text{Pred}) \mid \phi \models \text{wlp}(P) \,\}$$

Cartesian Post

Compute effect of heap updates locally

- for each abstract object C_i
- and independently for each heap predicate P

Cartesian Post

$$\text{CartesianPost}(\forall v\,.\,\bigvee_i C_i)$$

$$= \forall v\,.\,\bigvee_i \bigwedge \{\, P \mid C_i \models \text{wlp}(P) \,\}$$

In practice: precompute abstract weakest preconditions

$$\text{wlp}^{\#}(P) = \bigvee \{\, \phi \in \text{BoolExp}(\text{Pred}) \mid \phi \models \text{wlp}(P) \,\}$$

Cartesian Post

Same advantages as for predicate abstraction:

- ➜ abstraction reduced to checking verification conditions
- ➜ requires $\mathcal{O}(n^k)$ decision procedure calls (in practice)
- ➜ abstract transformer computed once for the whole analysis
- ➜ best abstract post can be computed from Cartesian post.

What is wlp(P)?

where P is not an assertion on states, but defined by a formula in a variable v ranging over nodes, such as $next^*(x, v)$

Node Predicates

Denotation of a formula with a free variable v:

$$[\![next(v) = z]\!] = \lambda s \in \text{State} \,.\, \{ o \in \text{Obj} \mid next_s\ o = z_s \}$$

$$\text{or } [\![next(v) = z]\!] = \lambda o \in \text{Obj} \,.\, \{ s \in \text{State} \mid next_s\ o = z_s \}$$

Node Predicates

Denotation of a formula with a free variable v:

$$[\![next(v) = z]\!] = \lambda s \in \text{State} \,.\, \{\, o \in \text{Obj} \mid next_s\ o = z_s \,\}$$
$$\text{or } [\![next(v) = z]\!] = \lambda o \in \text{Obj} \,.\, \{\, s \in \text{State} \mid next_s\ o = z_s \,\}$$

Node predicates

$$\text{NodePred} \stackrel{def}{=} \text{Obj} \to 2^{\text{State}}$$
$$[\![\phi(v)]\!] \stackrel{def}{=} \lambda o \,.\, \{\, s \in \text{State} \mid s, [v \mapsto o] \models \phi(v) \,\}$$

Node Predicate Transformers

Remember: $\text{NodePred} = \text{Obj} \to 2^{\text{State}}$.

Lift predicate transformers post and wlp to node predicates.

$$\begin{aligned} \text{lift} &\in (2^{\text{State}} \to 2^{\text{State}}) \to \text{NodePred} \to \text{NodePred} \\ \text{lift}\ \tau\ p &= \lambda o \,.\, \tau\ (p\ o) \end{aligned}$$

Node Predicate Transformers

Remember: $\mathsf{NodePred} = \mathsf{Obj} \to 2^{\mathsf{State}}$.

Lift predicate transformers post and wlp to node predicates.

$$\begin{aligned} \mathsf{lift} &\in (2^{\mathsf{State}} \to 2^{\mathsf{State}}) \to \mathsf{NodePred} \to \mathsf{NodePred} \\ \mathsf{lift}\ \tau\ p &= \lambda o \,.\, \tau\ (p\ o) \end{aligned}$$

Definition

Node predicate transformers :

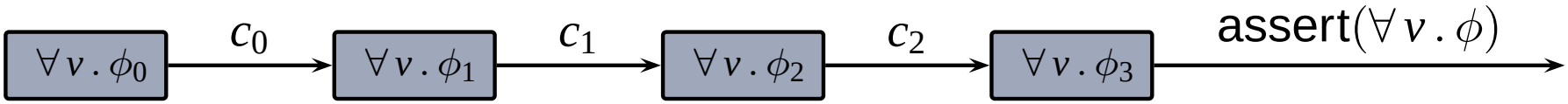
$$\begin{aligned} \mathsf{hpost}, \mathsf{hwlp} &\in \mathsf{Com} \to \mathsf{NodePred} \to \mathsf{NodePred} \\ \mathsf{hpost}\ c &\stackrel{def}{=} \mathsf{lift}\ (\mathsf{post}\ c) \\ \mathsf{hwlp}\ c &\stackrel{def}{=} \mathsf{lift}\ (\mathsf{wlp}\ c) \end{aligned}$$

Outline

1. Boolean heaps (abstract domain)
2. Cartesian post (abstract transformer)
3. Abstraction refinement
4. Bohne - implementation of symbolic shape analysis

Counter-Example based Abstraction Refinement

Abstract error trace

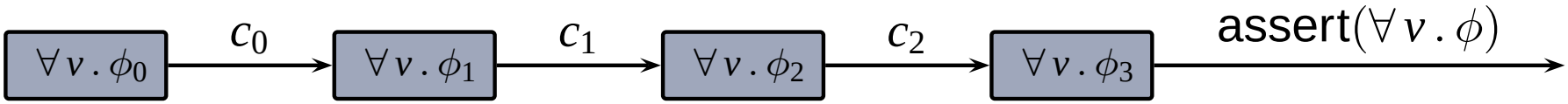


$\forall v \,.\, \phi_3 \not\models \forall v \,.\, \phi$

assertion fails

Counter-Example based Abstraction Refinement

Abstract error trace

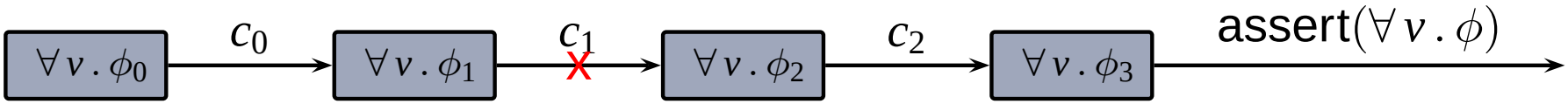


$\forall v \,.\, \phi_2 \not\models \mathsf{wlp}_{c_2}(\forall v \,.\, \phi)$

backwards analyze error trace

Counter-Example based Abstraction Refinement

Abstract error trace



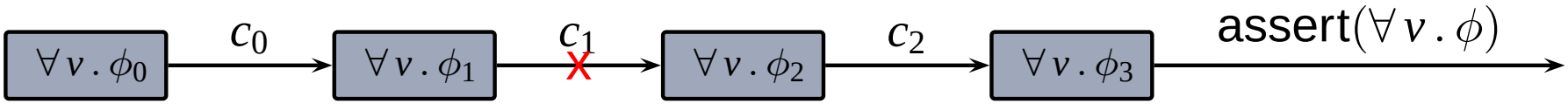
$\forall v \,.\, \phi_1 \models \mathsf{wlp}_{c_1;c_2}(\forall v \,.\, \phi)$

error trace is spurious

Refinement: add atoms of weakest preconditions along the path as new predicates.

Counter-Example based Abstraction Refinement

Abstract error trace



$$\forall v \,.\, \phi_1 \models \mathsf{wlp}_{c_1;c_2}(\forall v \,.\, \phi)$$

error trace is spurious

Refinement: add atoms of weakest preconditions along the path as new predicates.

Theorem (Progress)

If analysis is based on best abstract post then refinement step eliminates spurious error trace.

Progress property holds for best abstract post,
but does not hold for Cartesian post.

- Folklore says: best abstract post does not pay off.
- Theory says: best abstract post does pay off in the presence of abstraction refinement.
- Practice says: yes, it does indeed.

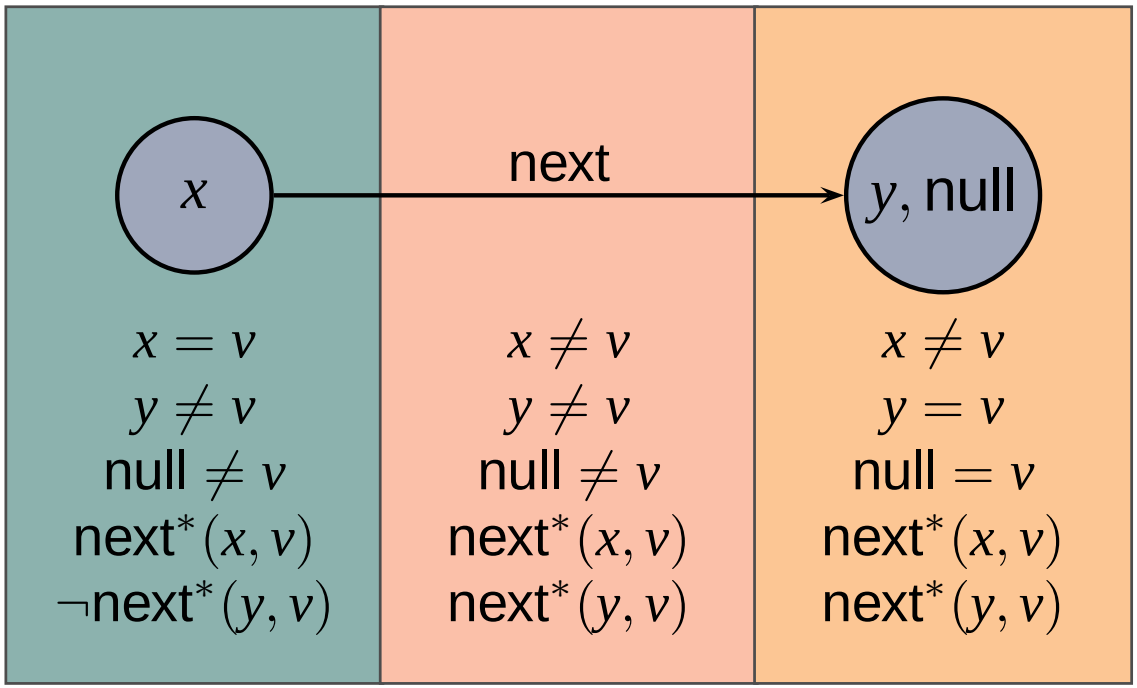
But: we have to efficiently implement best abstract post.

Implementing the Best Abstract Post

Cleaning operator: clean

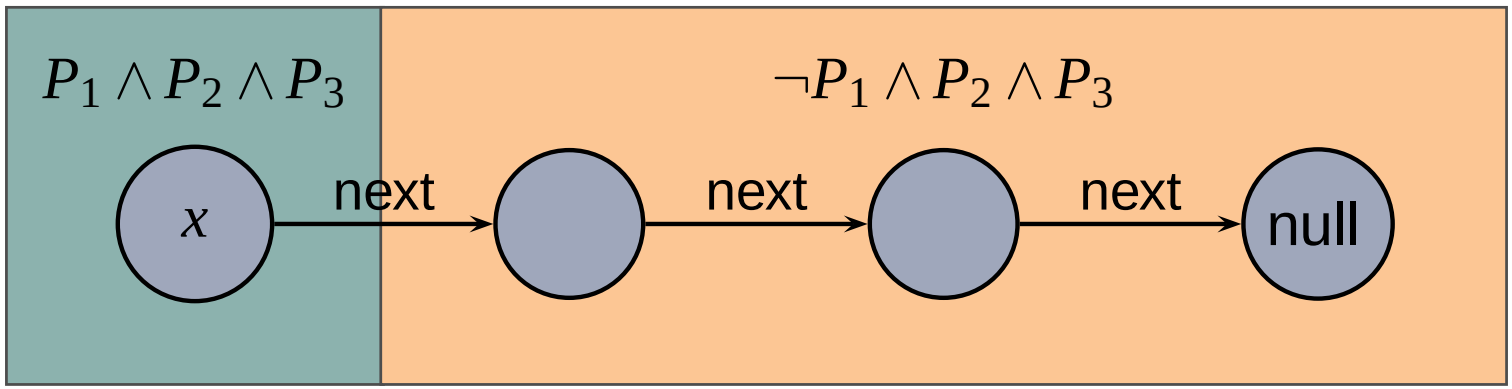
Semantics: clean $= \alpha \circ \gamma$

Effect: strengthens Boolean heaps by removing unsatisfiable abstract objects.



Implementing the Best Abstract Post

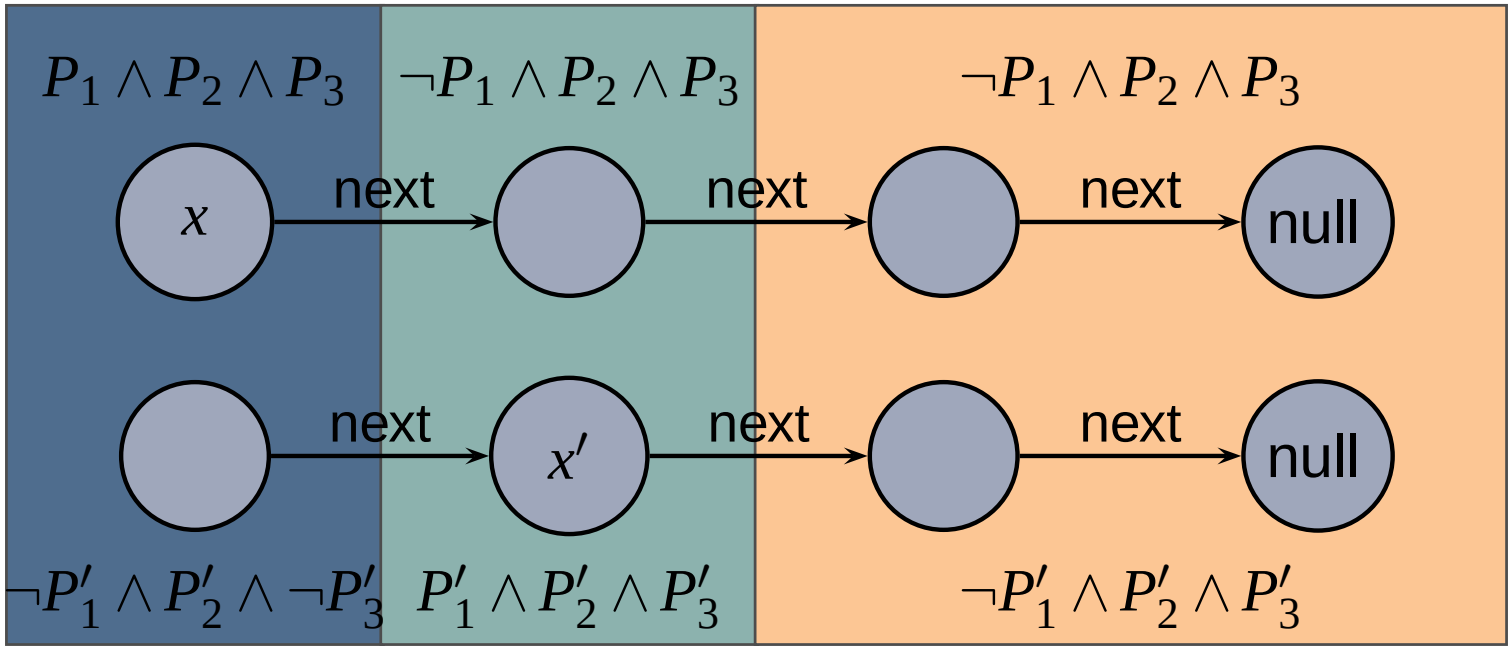
$$P_1 = \{\, v \mid v = x \,\} \quad P_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P_3 = \{\, v \mid \mathsf{next}^*(x, v) \,\}$$



command $c = (\texttt{x:=x.next})$

Implementing the Best Abstract Post

$$P_1 = \{\, v \mid v = x \,\} \quad P_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P_3 = \{\, v \mid \mathsf{next}^*(x, v) \,\}$$

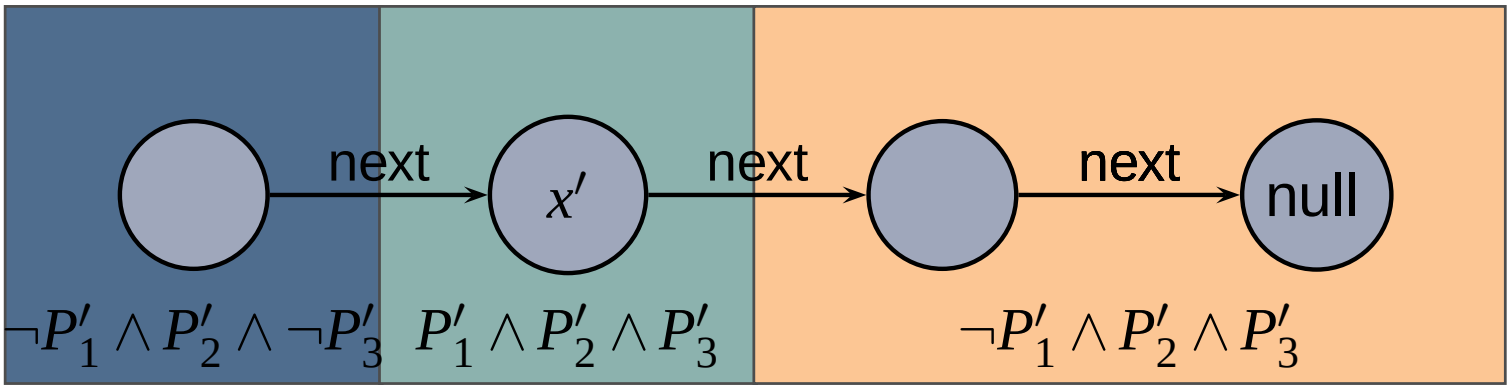


command $c = (\texttt{x:=x.next})$

$$P'_1 = \{\, v \mid v = \mathsf{next}(\mathsf{x}) \,\} \quad P'_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P'_3 = \{\, v \mid \mathsf{next}^+(x, v) \,\}$$

Implementing the Best Abstract Post

$$P_1 = \{\, v \mid v = x \,\} \quad P_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P_3 = \{\, v \mid \mathsf{next}^*(x, v) \,\}$$



command $c = (\texttt{x:=x.next})$

$$P'_1 = \{\, v \mid v = \mathsf{next}(\mathsf{x}) \,\} \quad P'_2 = \{\, v \mid \mathsf{next}^*(v, \mathsf{null}) \,\} \quad P'_3 = \{\, v \mid \mathsf{next}^+(x, v) \,\}$$

But it is exponential in number of predicates. . .

Implementing the Best Abstract Post

First compute Cartesian post and then clean.

Implementation of $\text{post}_c^\#$

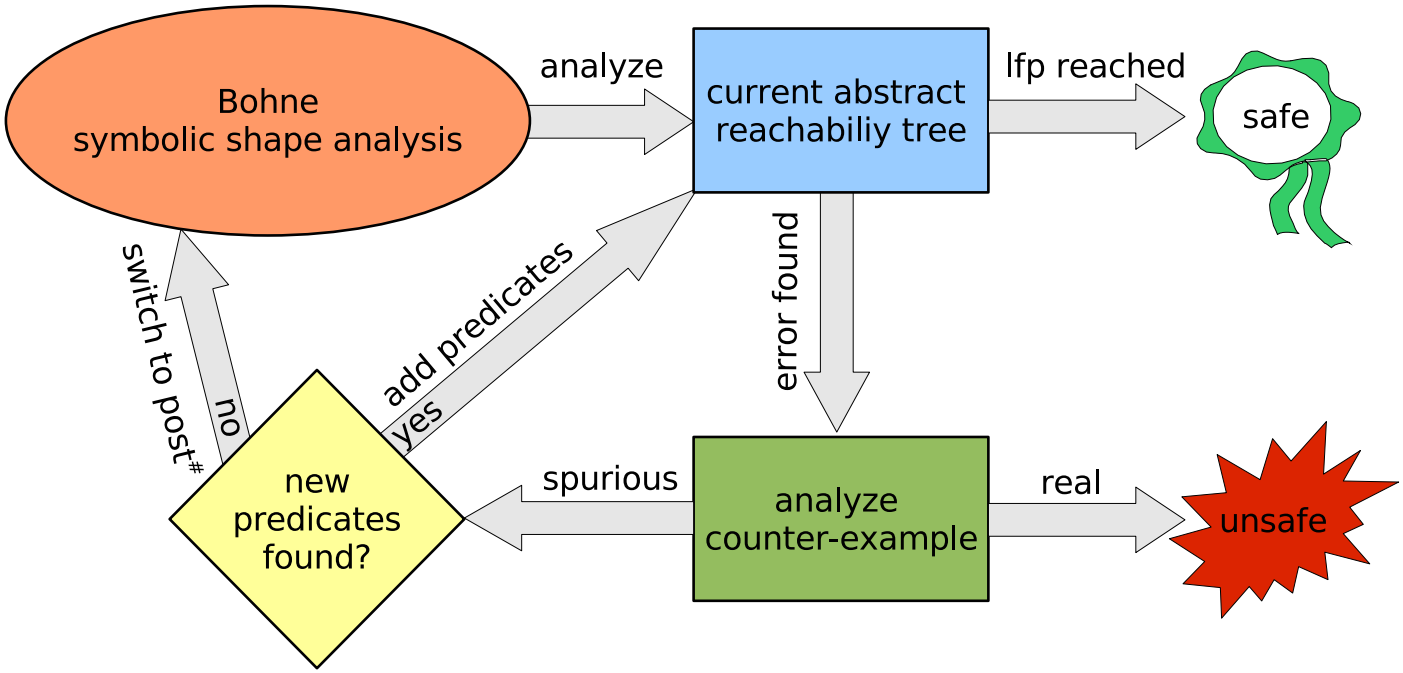
Given Boolean heap $H = \forall v \,.\, \phi$ over predicates Pred

1. for each predicate $P \in \text{Pred}$ consider a new predicate P' with $P' = \text{wlp}_c(P)$
2. compute $H_0 = \forall v \,.\, \phi \wedge \bigwedge_{P \in \text{Pred}} (\text{wlp}_c^\#(P) \;\rightarrow\; P')$
3. compute $H_1 = \text{clean}[\text{Pred} \cup \text{Pred}'](H_0)$
4. compute $H_2 = (\exists \text{Pred}.H_1)[\text{Pred}/\text{Pred}']$

then we have $H_2 = \text{post}_c^\#(H)$.

Best abstract post efficient when exploiting pre-computed Cartesian post.

Bohne's Abstraction Refinement Loop



Two refinements within lazy abstraction

1. add new predicates
2. switch from Cartesian post to best abstract post

Outline

1. Boolean heaps (abstract domain)
2. Cartesian post (abstract transformer)
3. Abstraction refinement
4. Bohne - implementation of symbolic shape analysis

```
class SortedList {
  private static Node first;
  /*: public static specvar content :: objset;
      vardefs "content == {v. v ≠ null ∧ next* first v}";
      invariant "tree [next]";
      invariant "∀ v. v ∈ content ∧ v.next ≠ null
                              ⟶ v..Node.data ≤ v.next.data"; */
  public static void insert(Node n)
    /*: requires "n ≠ null ∧ n ∉ content"
        modifies content
        ensures "content = old content ∪ {n}" */
  {
    Node prev = null;
    Node curr = first;
    while ((curr != null) && (curr.data < n.data)) {
      prev = curr;
      curr = curr.next;
    }
    n.next = curr;
    if (prev != null) prev.next = n;
    else first = n;
  }
}
```

Bohne, Symbolic Shape Analysis Implementation

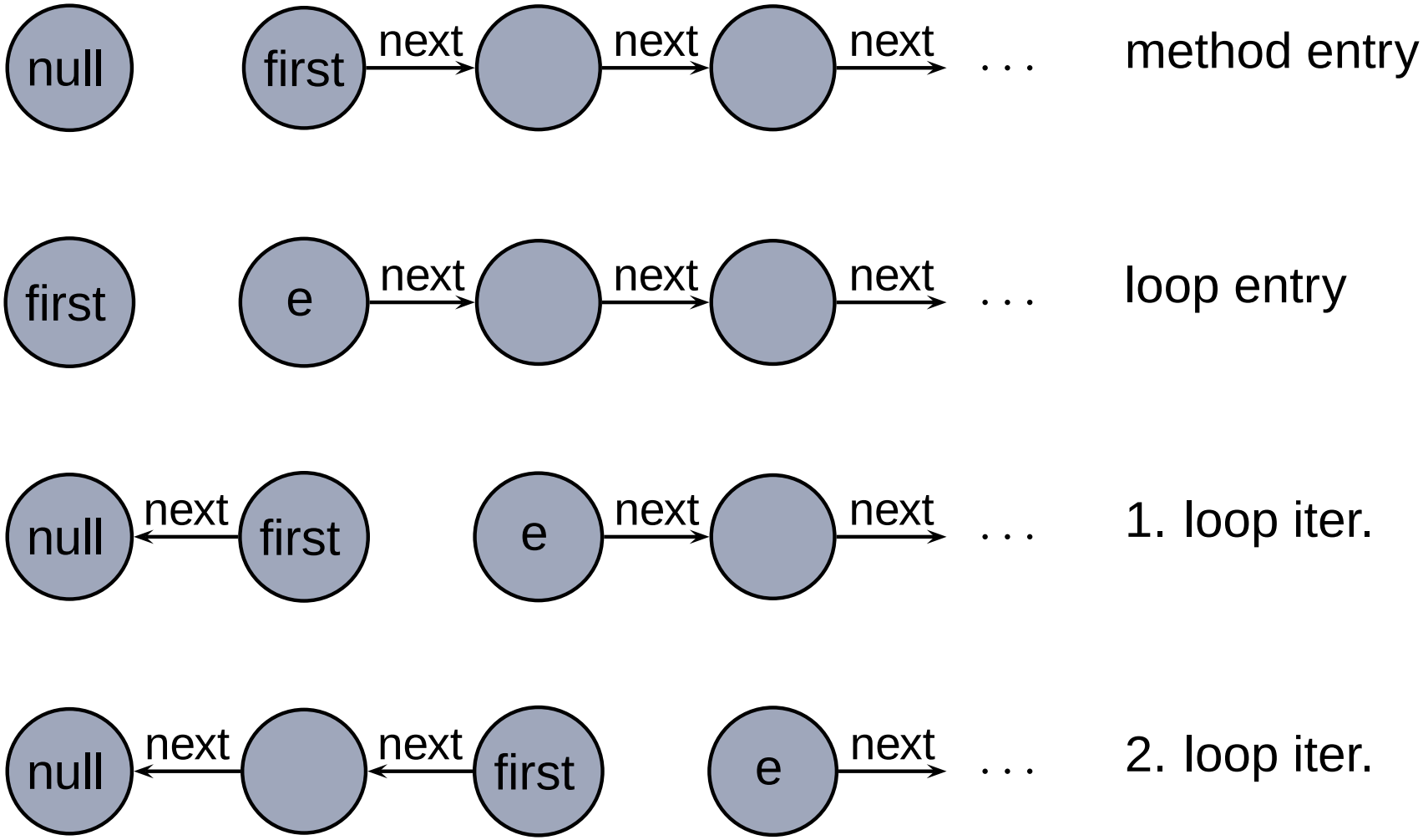
Properties verified in previous example:

- correctly inserts the element into the list
  (relates pre- and post states of procedure)
- list remains sorted
- data structure remains acyclic list
- no null pointer dereferences

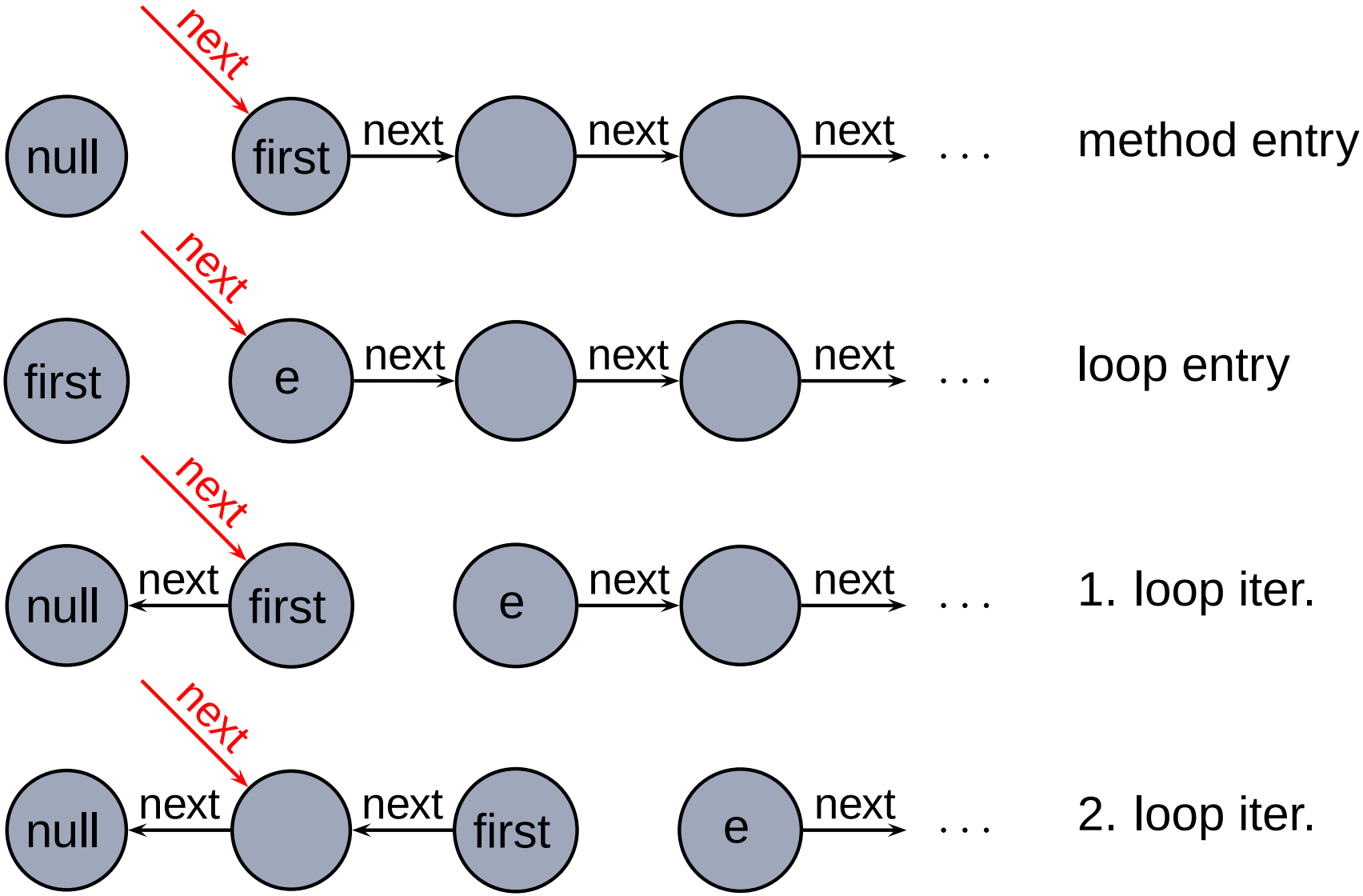
Bohne

- accepts annotated Java programs as input
- annotations are user-specied formulae:
  - data structure invariants
  - procedure contracts (pre- and post conditions)
- automatically computes quantified loop invariants
- proves desired properties and absence of errors

List Reversal



List Reversal



Some Experimental Results

benchmark	used DP	# predicates	# validity checker calls total (cache hits)	running time total (DP)
DLL.addLast	MONA	7	118 (19%)	2s (69%)
List.reverse	MONA	10	465 (33%)	7s (64%)
SortedList.add	MONA, CVC lite	17	623 (56%)	14s (59%)
Skiplist.add	MONA	20	787 (44%)	26s (57%)
Tree.add	MONA	13	358 (31%)	31s (92%)
ParentTree.add	MONA	13	362 (32%)	33s (91%)

No manually supplied predicates in any of the examples.

Checked properties include

- procedure contracts: elements are inserted into/removed from the data structure
- data structure consistency: sortedness, treeness
- absence of errors: null pointer dereferences

Conclusion

Bohne - symbolic shape analyzer

- verifies complex user-specified properties of Java programs
  - procedure contracts
  - data structure invariants
- infers loop invariants automatically
  - disjunctions of universally quantified Boolean combinations of predicates on heap objects
  - predicates are inferred automatically

Future Work

- specialized reasoning procedures for lists/trees
- exploit combinations of reasoning procedures
- abstraction refinement with interpolation