Fully Automated Shape Analysis Based on Forest Automata[†]

P.A. Abdulla P. Habermehl L. Holík M. Hruška B. Jonsson
O. Lengál C.Q. Trinh A. Rogalewicz J. Šimáček **T. Vojnar**

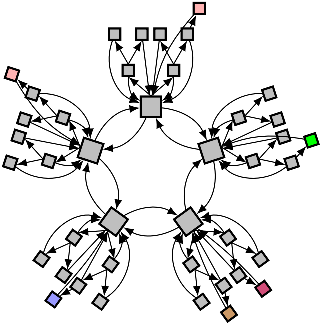
Brno University of Technology, Czech Republic
LIAFA, Université Paris Diderot, France
Uppsala University, Sweden
Academia Sinica, Taiwan

Vienna UT 2015

[†] Publications: *CAV'11, FMSD'12, CAV'13, ATVA'13, AI'15, SV-COMP'15.*

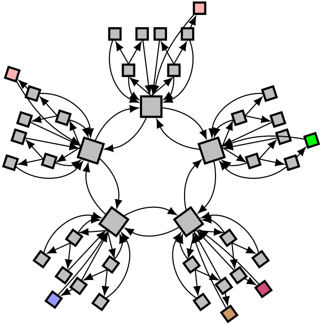
Shape Analysis

- Shape analysis:
  - characterizes shapes of dynamic linked data structures,
  - notoriously difficult: infinite sets of complex graphs.



Shape Analysis

- Shape analysis:
  - characterizes shapes of dynamic linked data structures,
  - notoriously difficult: infinite sets of complex graphs.



- Applications:
  - memory safety: invalid dereferences, double free, memory leakage,
  - checking pointer-related assertions in the code,
  - shape invariants (checked automatically/manually), ...

Motivation

- Many approaches to shape analysis have been proposed:
  - logics (TVLA, PALE, separation logic, ...), automata, grammars, graphs, ...

Motivation

- Many approaches to shape analysis have been proposed:
  - logics (TVLA, PALE, separation logic, ...), automata, grammars, graphs, ...

- Limitations of the current approaches:
  - often specialized (lists) or of a limited generality,
  - require human help (loop invariants, inductive predicates),
  - insufficient scalability.

Motivation

- Many approaches to shape analysis have been proposed:
    - logics (TVLA, PALE, separation logic, ...), automata, grammars, graphs, ...

- Limitations of the current approaches:
    - often specialized (lists) or of a limited generality,
    - require human help (loop invariants, inductive predicates),
    - insufficient scalability.

- Separation Logic:
    - ☺ local reasoning: well scalable,
    - ☹ often fixed abstraction.

Motivation

- Many approaches to shape analysis have been proposed:
  - logics (TVLA, PALE, separation logic, ...), automata, grammars, graphs, ...

- Limitations of the current approaches:
  - often specialized (lists) or of a limited generality,
  - require human help (loop invariants, inductive predicates),
  - insufficient scalability.

- Separation Logic:
  - ☺ local reasoning: well scalable,
  - ☹ often fixed abstraction.

- Abstract Regular Tree Model Checking (ARTMC):
  - ☺ uses tree automata (TA): flexible and refinable abstraction,
  - ☹ monolithic encoding of the heap: limited scalability.

The Forest Automata-based Approach

- Our approach based on forest automata combines
  - ☺ flexibility of ARTMC

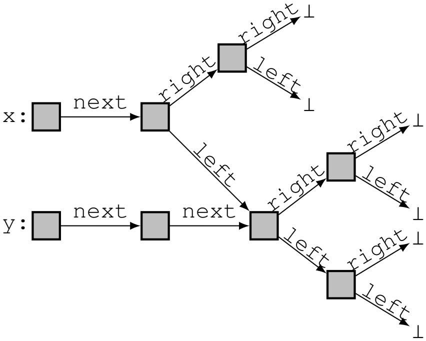
  with
  - ☺ scalability of SL

  by
  - ‣ splitting heaps into tree components

  and
  - ‣ using tuples of tree automata to represent tuples of sets of tree components of heaps.

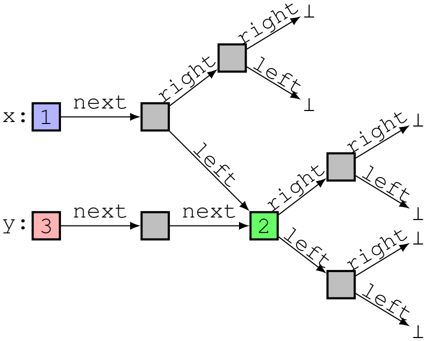
Canonical Heap Representation

- Forest decomposition of a heap:



Canonical Heap Representation

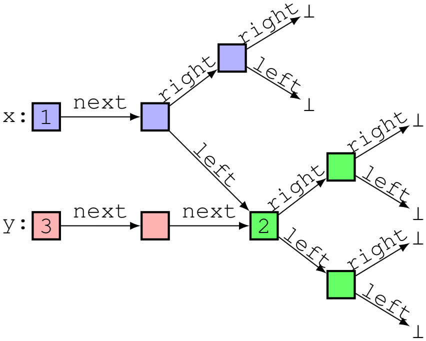
- Forest decomposition of a heap:
  - Identify cut-points. ← nodes referenced:
    - by variables or
    - multiple times



Canonical Heap Representation

- Forest decomposition of a heap:
  - Identify cut-points.
  - Identify tree components.

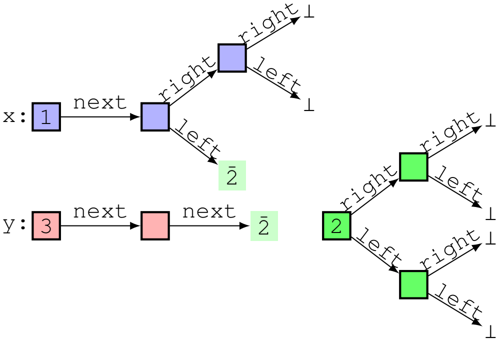
nodes referenced:
- by variables or
- multiple times



Canonical Heap Representation

- Forest decomposition of a heap:
  - Identify cut-points.
  - Identify tree components.
  - Split the tree components using explicit references to cut-points.

nodes referenced:
- by variables or
- multiple times



Canonical Heap Representation

- A heap $h \mapsto$ a forest $(↟_1, ↟_2, \ldots, ↟_n)$.

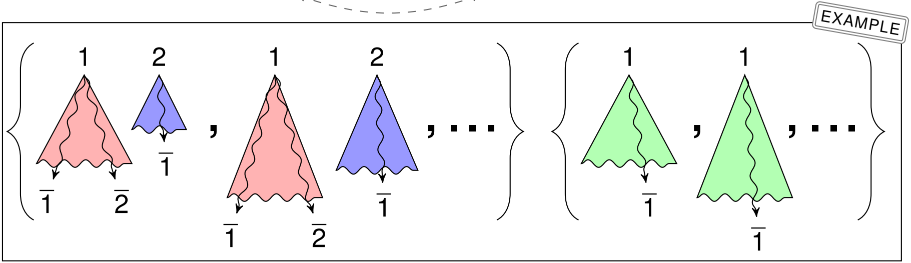
Canonical Heap Representation

- A heap $h \mapsto$ a forest $(⇞_1, ⇞_2, \ldots, ⇞_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(⇞_1, ⇞_2, \ldots, ⇞_n), (⇞'_1, ⇞'_2, \ldots, ⇞'_m), \ldots\}$.

Canonical Heap Representation

- A heap $h \mapsto$ a forest $(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \ldots, \twoheaduparrow'_m), \ldots\}$.
  - Sort tuples of trees w.r.t. a DFS.
  - Split $\mathcal{H}$ into classes of forests with the same number of trees,

    $(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \ldots, \twoheaduparrow'_n)$



Canonical Heap Representation

- A heap $h \mapsto$ a forest $(\text{⇞}_1, \text{⇞}_2, \ldots, \text{⇞}_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(\text{⇞}_1, \text{⇞}_2, \ldots, \text{⇞}_n), (\text{⇞}'_1, \text{⇞}'_2, \ldots, \text{⇞}'_m), \ldots\}$.
  - Sort tuples of trees w.r.t. a DFS.
  - Split $\mathcal{H}$ into classes of forests with the same number of trees,

    $(\text{⇞}_1, \text{⇞}_2, \ldots, \text{⇞}_n), (\text{⇞}'_1, \text{⇞}'_2, \ldots, \text{⇞}'_n)$
- Cartesian representation of classes of $\mathcal{H}$:

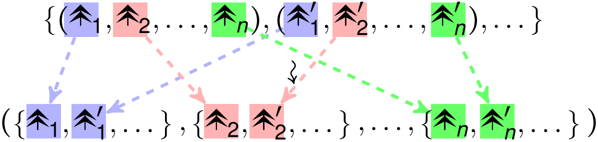
$$\{(\text{⇞}_1, \text{⇞}_2, \ldots, \text{⇞}_n), (\text{⇞}'_1, \text{⇞}'_2, \ldots, \text{⇞}'_n), \ldots\}$$

$$\rightsquigarrow$$

Canonical Heap Representation

- A heap $h \mapsto$ a forest $(↟_1, ↟_2, \ldots, ↟_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_m), \ldots\}$.
  - Sort tuples of trees w.r.t. a DFS.
  - Split $\mathcal{H}$ into classes of forests with the same number of trees,

    $(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n)$
- Cartesian representation of classes of $\mathcal{H}$:



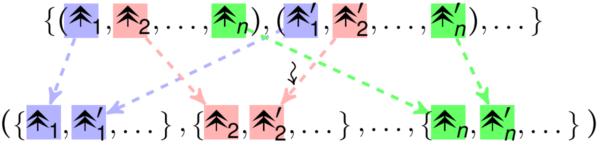
$\{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n), \ldots\}$

$\rightsquigarrow$

$(\{↟_1, ↟'_1, \ldots\}, \{↟_2, ↟'_2, \ldots\}, \ldots, \{↟_n, ↟'_n, \ldots\})$

Canonical Heap Representation

- A heap $h \mapsto$ a forest $(↟_1, ↟_2, \ldots, ↟_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_m), \ldots\}$.
  - Sort tuples of trees w.r.t. a DFS.
  - Split $\mathcal{H}$ into classes of forests with the same number of trees, $(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n)$
- Cartesian representation of classes of $\mathcal{H}$:



$\{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n), \ldots\}$

$\rightsquigarrow$

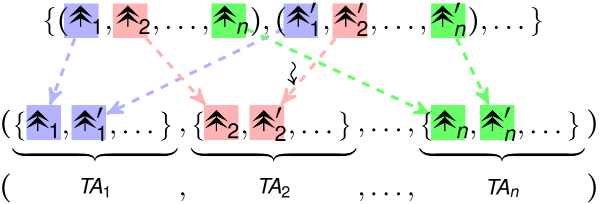
$(\{↟_1, ↟'_1, \ldots\}, \{↟_2, ↟'_2, \ldots\}, \ldots, \{↟_n, ↟'_n, \ldots\})$

- We assume working with rectangular classes, i.e., for a class C, $(↟, \_\_), (\_\_, ↟) \in C \Rightarrow (↟, ↟) \in C$, otherwise C is split.

Canonical Heap Representation

- A heap $h \mapsto$ a forest $(↟_1, ↟_2, \ldots, ↟_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_m), \ldots\}$.
  - Sort tuples of trees w.r.t. a DFS.
  - Split $\mathcal{H}$ into classes of forests with the same number of trees,

    $(↟_1, ↟_2, \ldots, ↟_{\boxed{n}}), (↟'_1, ↟'_2, \ldots, ↟'_{\boxed{n}})$
- Cartesian representation of classes of $\mathcal{H}$:



$$\{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n), \ldots\}$$

$$\leadsto$$

$$(\underbrace{\{↟_1, ↟'_1, \ldots\}}_{TA_1}, \underbrace{\{↟_2, ↟'_2, \ldots\}}_{TA_2}, \ldots, \underbrace{\{↟_n, ↟'_n, \ldots\}}_{TA_n})$$

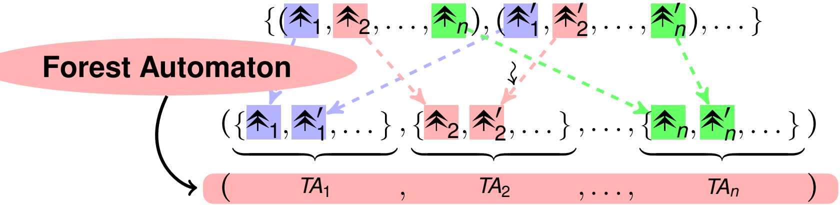
$$(\quad TA_1 \quad, \quad TA_2 \quad, \ldots, \quad TA_n \quad)$$

- We assume working with rectangular classes, i.e., for a class C, $(↟, \_\_), (\_\_, ↟) \in C \Rightarrow (↟, ↟) \in C$, otherwise C is split.

Canonical Heap Representation

- A heap $h \mapsto$ a forest $(↟_1, ↟_2, \ldots, ↟_n)$.
- A set of heaps $\mathcal{H} \mapsto \{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_m), \ldots\}$.
  - Sort tuples of trees w.r.t. a DFS.
  - Split $\mathcal{H}$ into classes of forests with the same number of trees,

    $(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n)$
- Cartesian representation of classes of $\mathcal{H}$:



$\{(↟_1, ↟_2, \ldots, ↟_n), (↟'_1, ↟'_2, \ldots, ↟'_n), \ldots\}$

$\rightsquigarrow$

Forest Automaton

$(\underbrace{\{↟_1, ↟'_1, \ldots\}}, \underbrace{\{↟_2, ↟'_2, \ldots\}}, \ldots, \underbrace{\{↟_n, ↟'_n, \ldots\}})$

$(\quad TA_1 \quad , \quad TA_2 \quad , \ldots , \quad TA_n \quad)$

  - We assume working with rectangular classes, i.e., for a class C, $(↟, \_\_), (\_\_, ↟) \in C \Rightarrow (↟, ↟) \in C$, otherwise C is split.

Maintaining Rectangularity and Canonicity

- Maintaining rectangularity:
  - A problem can appear when a TA is split since a new cut-point is introduced (e.g., after an `x := y.next`) statement.
  - Resolve by having a separate FA for each pair of states p and q linked by a root transition $p \xrightarrow{f} (..., q, ...)$ that is to be split.
    - Any tree accepted from q combines with any context accepted from p.

Maintaining Rectangularity and Canonicity

- Maintaining rectangularity:
  - A problem can appear when a TA is split since a new cut-point is introduced (e.g., after an `x := y.next`) statement.
  - Resolve by having a separate FA for each pair of states p and q linked by a root transition $p \xrightarrow{f} (..., q, ...)$ that is to be split.
    - Any tree accepted from q combines with any context accepted from p.

- Canonicity respecting FA:
  - Take any tuple of trees from the component TAs, compose into a heap, decompose in a canonical way, get the same tuple.
  - Mininum number of TAs, in the right order.

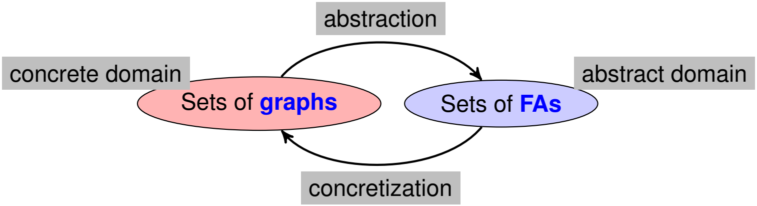
Maintaining Rectangularity and Canonicity

- Maintaining rectangularity:
  - A problem can appear when a TA is split since a new cut-point is introduced (e.g., after an `x := y.next`) statement.
  - Resolve by having a separate FA for each pair of states p and q linked by a root transition $p \xrightarrow{f} (..., q, ...)$ that is to be split.
    - Any tree accepted from q combines with any context accepted from p.

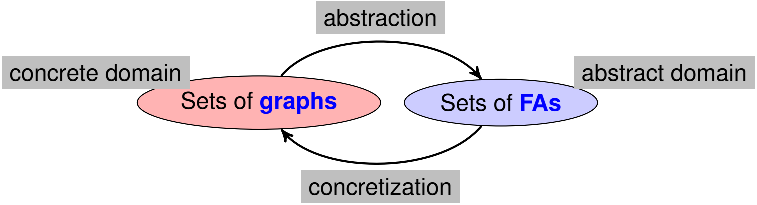
- Canonicity respecting FA:
  - Take any tuple of trees from the component TAs, compose into a heap, decompose in a canonical way, get the same tuple.
  - Mininum number of TAs, in the right order.

- Maintaining canonicity:
  - In a single bottom-up pass propagate information about the order in which root references can appear in the leaves.
    - Reorder accordingly, split if several orders appear in a single TA.
  - In a single bottom-up pass compute which root references appear once and which multiple times in a single tree.
    - Use to judge which roots are necessary, glue TAs if need be.

Abstract Interpretation



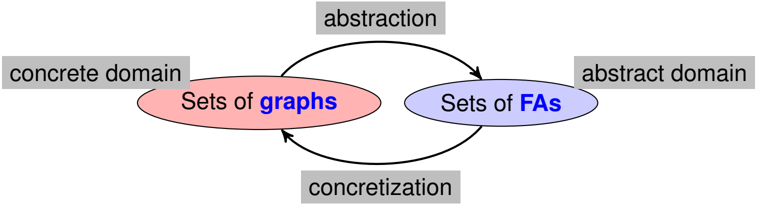
Abstract Interpretation



Statements

- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Interpretation

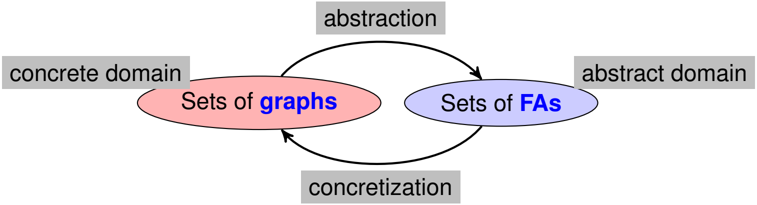


Statements

- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

Abstract Interpretation



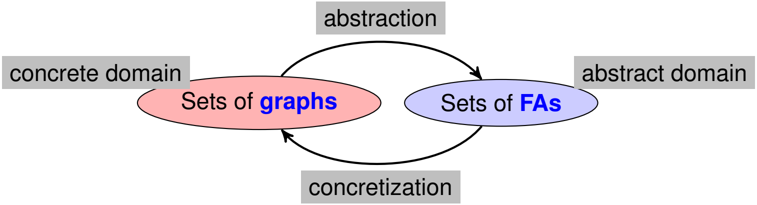
Statements

- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

append a TA

Abstract Interpretation



Statements

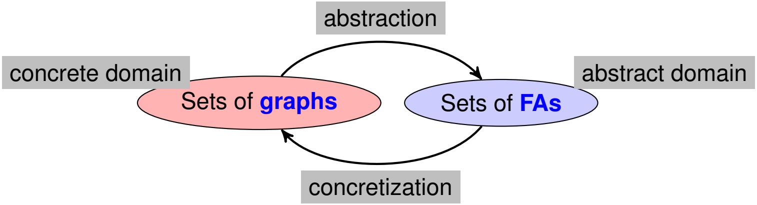
- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

append a TA

remove a TA

Abstract Interpretation



abstraction

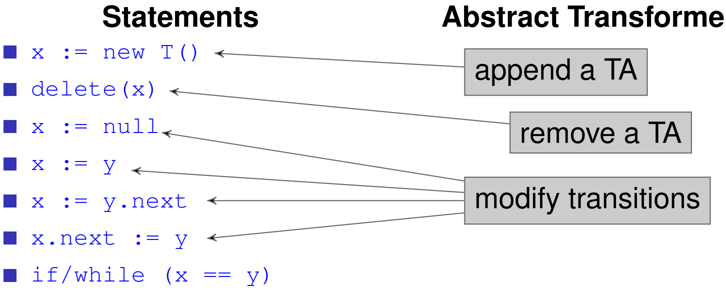
concrete domain

Sets of **graphs**

Sets of **FAs**

abstract domain

concretization



Statements

Abstract Transformers

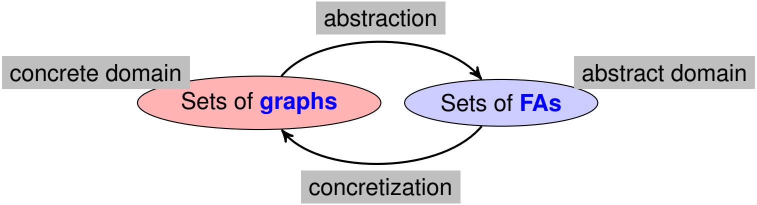
- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

append a TA

remove a TA

modify transitions

Abstract Interpretation



abstraction

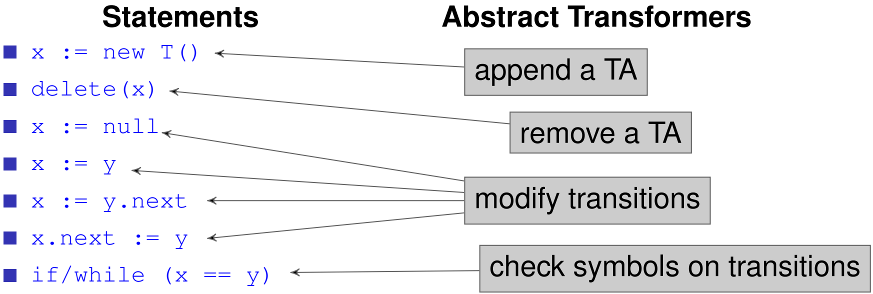
concrete domain

Sets of **graphs**

Sets of **FAs**

abstract domain

concretization



Statements

- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

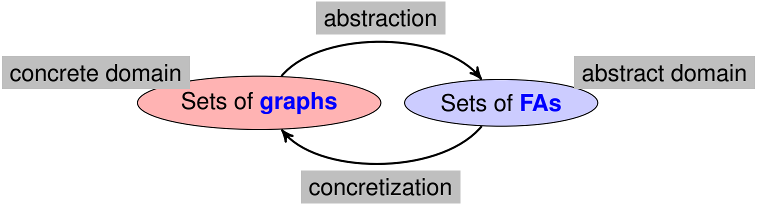
append a TA

remove a TA

modify transitions

check symbols on transitions

Abstract Interpretation



Statements

- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

append a TA

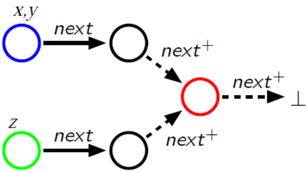
remov

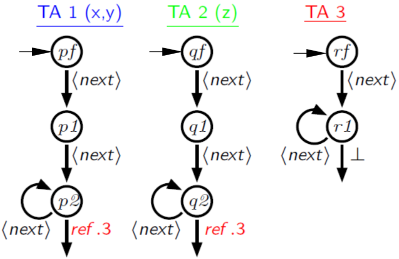
tions

heck symbols on transitions

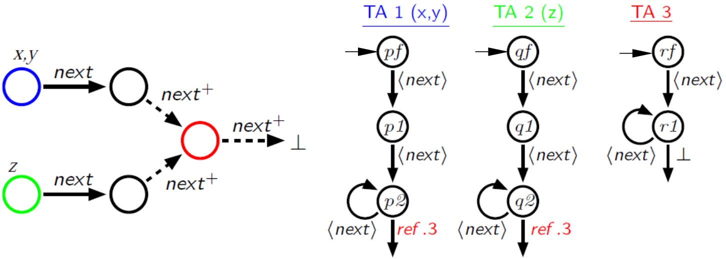
Followed by canonization: ordering, splitting, merging

Abstract Transformers for Pointer Updates



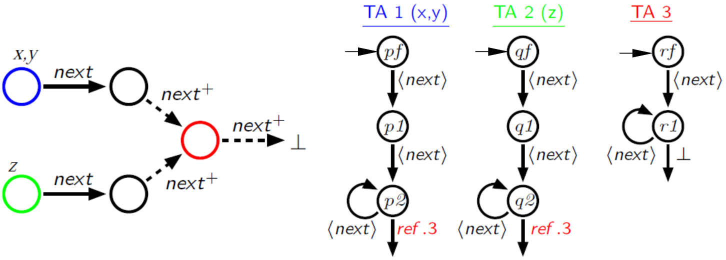


Abstract Transformers for Pointer Updates

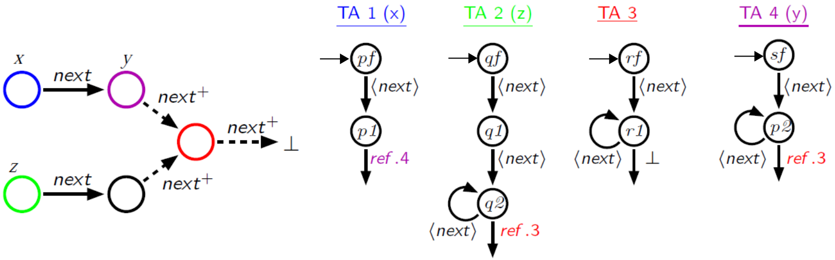


- y:=x.next

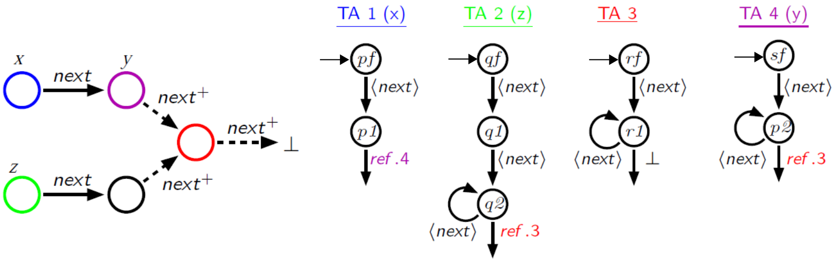
Abstract Transformers for Pointer Updates



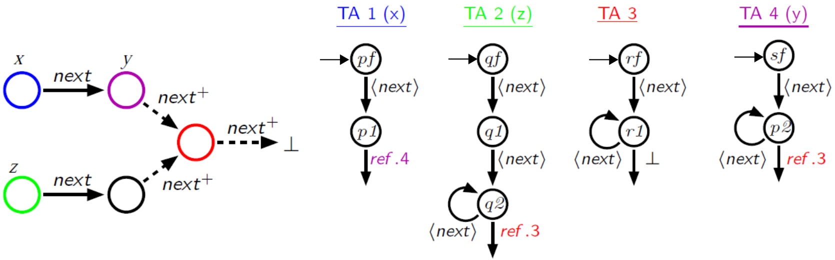
- y:=x.next



Abstract Transformers for Pointer Updates

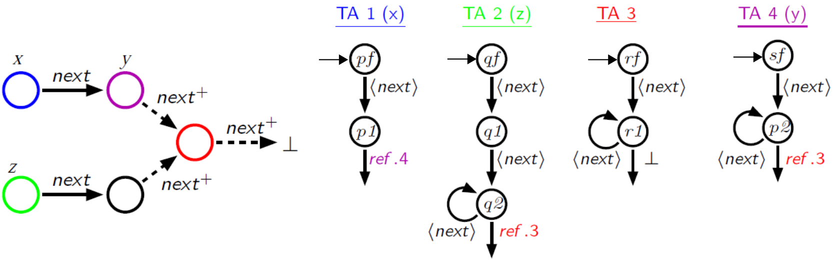


Abstract Transformers for Pointer Updates

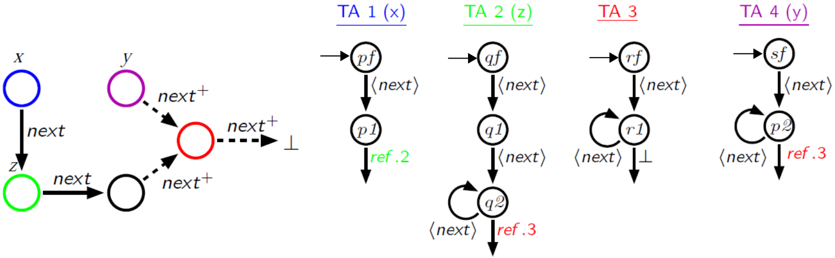


- x.next:=z;

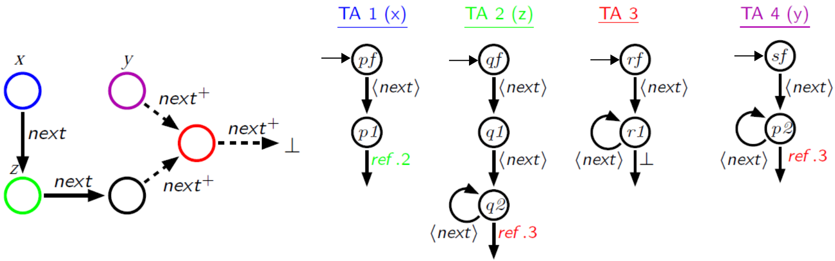
Abstract Transformers for Pointer Updates



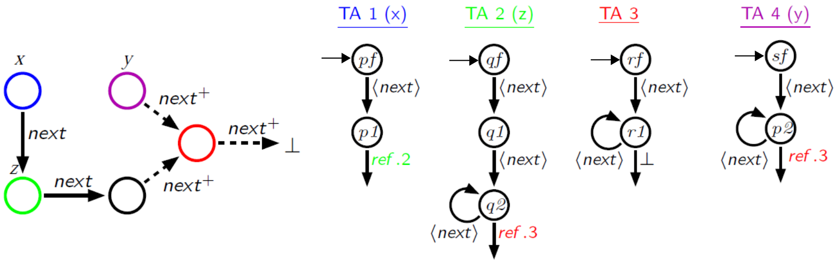
- x.next:=z;



Abstract Transformers for Pointer Updates

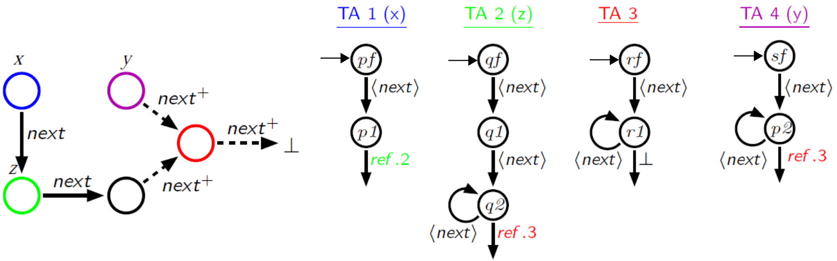


Abstract Transformers for Pointer Updates

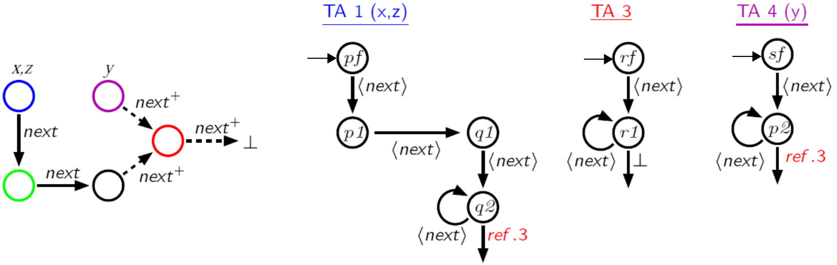


- z:=x;

Abstract Transformers for Pointer Updates

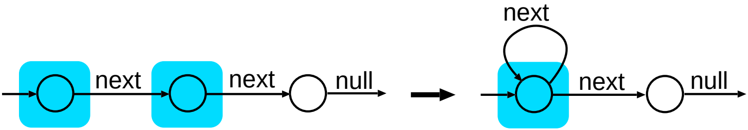


- z:=x;



Widening

- Abstraction on an FA $(TA_1, \ldots, TA_n)$:
  - Collapses states of component TAs leading to an FA $(TA_1^\alpha, \ldots, TA_n^\alpha)$.
  - Finite-height abstraction (from ARTMC),
    - collapses states with languages whose prefixes match up to height k:



  - Abstraction based on predicate languages refineable in a CEGAR loop is under preparation (first working prototype exists).

Nondeterministic Tree Automata

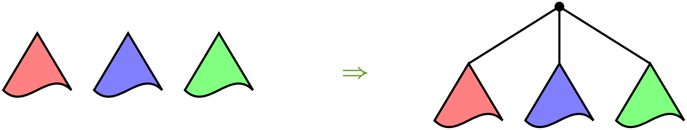
- For efficiency reasons, we never determinize TAs.
- All operations done on NTAs, including:
  - inclusion checking:
    - used for detecting the fixpoint,
    - inclusion on (normalized) FA can be checked component-wise,
    - precise even for sets of FAs,
    - based on antichains and simulations.
  - size reduction: based on simulation equivalences.
    - collapsing simulation-equivalent states.

Inclusion Checking

- Need to check inclusion between a new FA and a set of FAs computed so far the given line of the program being analysed.
  - Cannot be done componentwise!
  - One would loose information about which trees can and which cannot appear together.

Inclusion Checking

- Need to check inclusion between a new FA and a set of FAs computed so far the given line of the program being analysed.
  - Cannot be done componentwise!
  - One would loose information about which trees can and which cannot appear together.
- Inclusion of sets of canonical FAs can be easily reduced to inclusion of ordinary TAs.
  - One can convert a tuple of TAs into a single TA by adding a designated node on top of each tuple of trees.
  - Subsequently, a set of such TAs can be united into a single TA since there is no more a risk of loosing connection between the trees.



Summary

The so-far-presented:

Summary

The so-far-presented:

- ☺ works well for singly linked lists (SLLs), trees, SLLs with head/tail pointers, trees with root pointers, ...

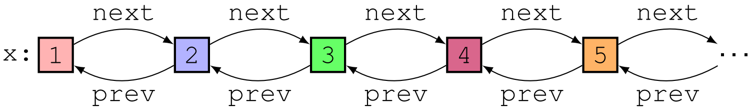
Summary

$(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n) \approx (\twoheaduparrow'_1, \twoheaduparrow'_2, \ldots, \twoheaduparrow'_n)$

...

The so-far-presented:

- ☺ works well for singly linked lists (SLLs), trees, SLLs with head/tail pointers, trees with root pointers, ...
- ☹ fails for more complex data structures:
  - unbounded number of cut-points $\rightsquigarrow$ ∞ classes of $\mathcal{H}$:



- doubly linked lists (DLLs), circular lists, nested lists,
- trees with parent pointers,
- skip lists.

Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.

Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

- Example: a box $\boxed{\text{DLS}}$

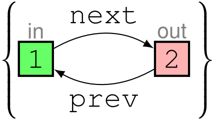
doubly linked segment

Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

doubly linked segment

- Example: a box $\boxed{\text{DLS}}$: $\mathcal{L}(\boxed{\text{DLS}})$ = $\left\{ \right.$



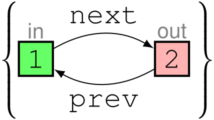
$\left. \right\}$

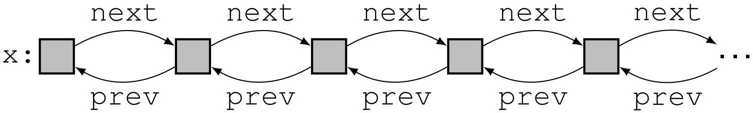
Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

doubly linked segment

- Example: a box $\boxed{\text{DLS}}$: $\mathcal{L}(\boxed{\text{DLS}}) =$

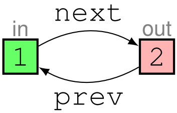


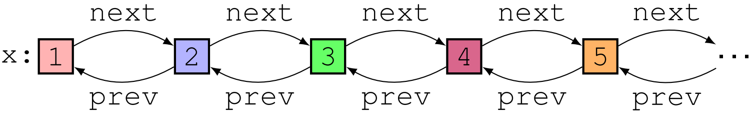


Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

doubly linked segment

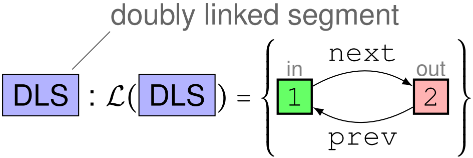
- Example: a box $\boxed{\text{DLS}}$: $\mathcal{L}(\boxed{\text{DLS}}) = \{$  $\}$

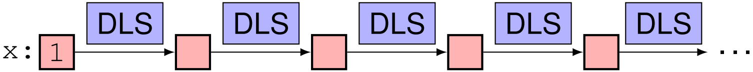


Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

- Example: a box DLS : $\mathcal{L}(\text{DLS}) =$



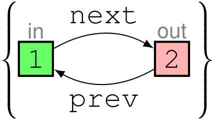


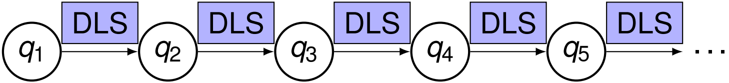
Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

doubly linked segment

- Example: a box $\boxed{\text{DLS}}$: $\mathcal{L}(\boxed{\text{DLS}}) =$



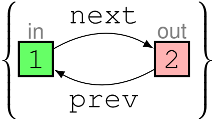


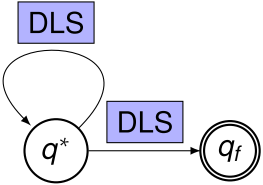
Hierarchical Forest Automata

- Hierarchical Forest Automata:
  - FAs are symbols (**boxes**) of FAs of a higher level.
  - A hierarchy of FAs.
  - Intuition: replace repeated subgraphs by a single symbol, hiding some cut-points.

doubly linked segment

- Example: a box DLS : $\mathcal{L}($DLS$) =$





Learning of Boxes

The Challenge

How to find the “right” boxes?

Learning of Boxes

The Challenge

How to find the “right” boxes?

- CAV’11 — database of boxes
- CAV’13 — automatic discovery

Learning of Boxes

- Compromise between

Learning of Boxes

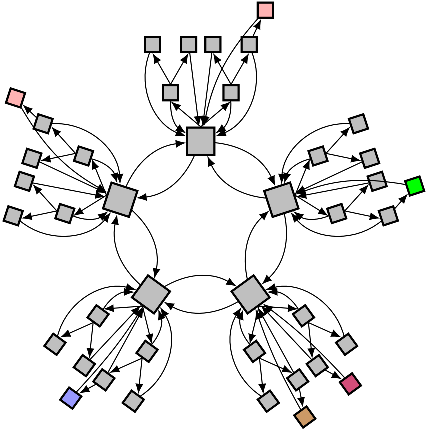
- Compromise between
  - reusability: use on different heaps of the same kind,

    $\leadsto$ use small boxes,

Learning of Boxes

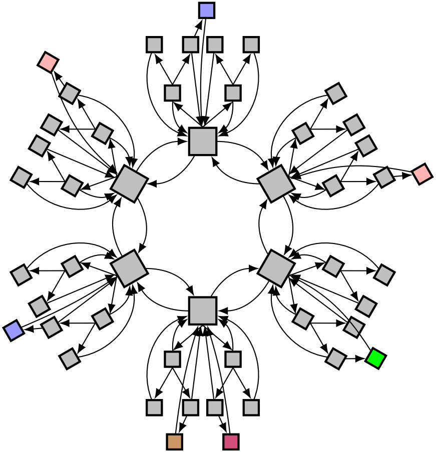
- Compromise between
  - reusability: use on different heaps of the same kind,

    $\rightsquigarrow$ use small boxes,



Learning of Boxes

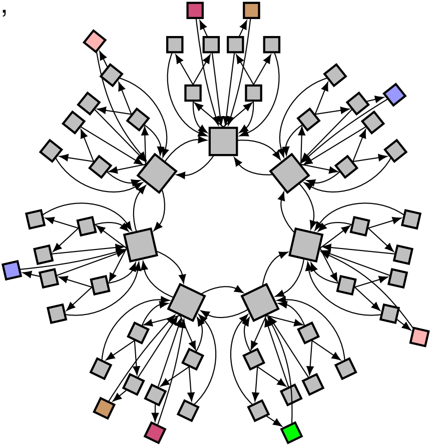
- Compromise between
  - reusability: use on different heaps of the same kind,
    $\rightsquigarrow$ use small boxes,



Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,

    $\rightsquigarrow$ use small boxes,

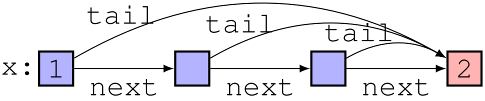


Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    $\rightsquigarrow$ use small boxes,
  - ability to hide cut-points,
    $\rightsquigarrow$ do not use too small boxes.

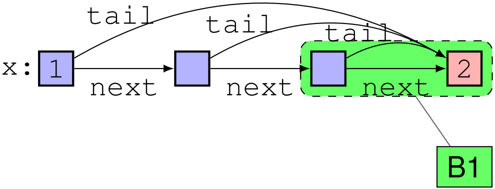
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    $\rightsquigarrow$ use small boxes,
  - ability to hide cut-points,
    $\rightsquigarrow$ do not use too small boxes.



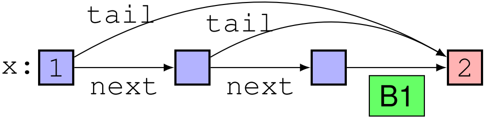
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    $\rightsquigarrow$ use small boxes,
  - ability to hide cut-points,
    $\rightsquigarrow$ do not use too small boxes.



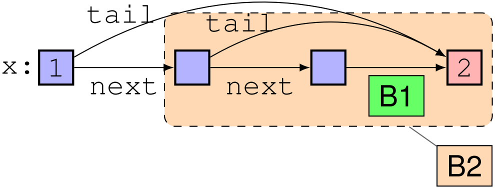
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    $\rightsquigarrow$ use small boxes,
  - ability to hide cut-points,
    $\rightsquigarrow$ do not use too small boxes.



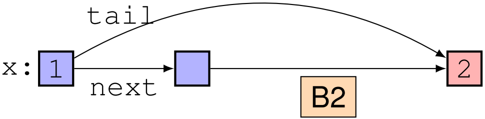
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    ⇝ use small boxes,
  - ability to hide cut-points,
    ⇝ do not use too small boxes.



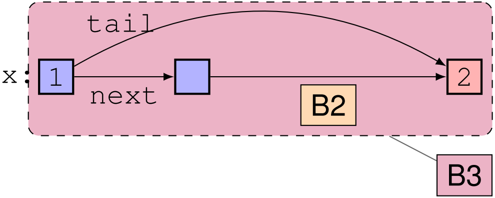
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    $\leadsto$ use small boxes,
  - ability to hide cut-points,
    $\leadsto$ do not use too small boxes.



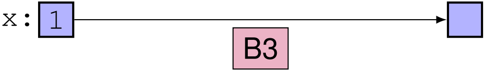
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    ⇝ use small boxes,
  - ability to hide cut-points,
    ⇝ do not use too small boxes.



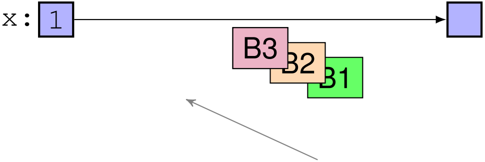
Learning of Boxes

- Compromise between
  - reusability: use on different heaps of the same kind,
    ⇝ use small boxes,
  - ability to hide cut-points,
    ⇝ do not use too small boxes.



Learning of Boxes

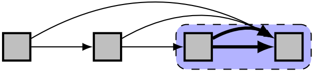
- Compromise between
  - reusability: use on different heaps of the same kind,
    $\leadsto$ use small boxes,
  - ability to hide cut-points,
    $\leadsto$ do not use too small boxes.

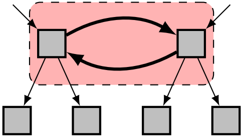


list of any length $\leadsto$ ∞ hierarchy

Learning of Boxes: Knots

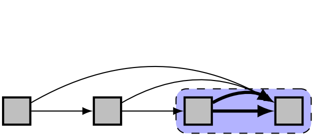
1. Smallest subgraphs meaningful to be folded:

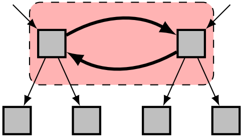




Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:

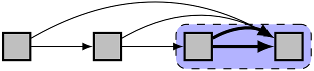


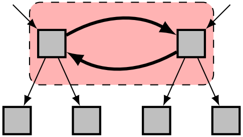


2. Build larger knots inductively:

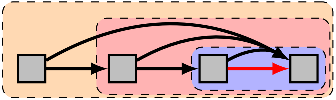
Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:





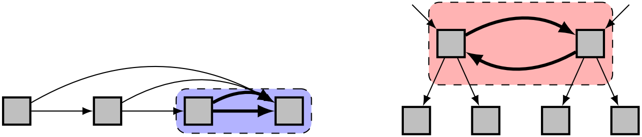
2. Build larger knots inductively:
   - Compose knots sharing edges:



prevent ∞ nesting

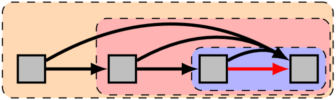
Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:



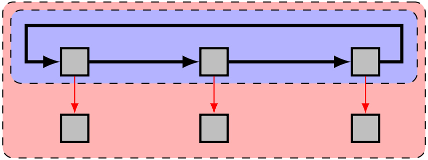
2. Build larger knots inductively:

   - Compose knots sharing edges:



prevent ∞ nesting

   - Enclose paths from inner nodes to leaves:



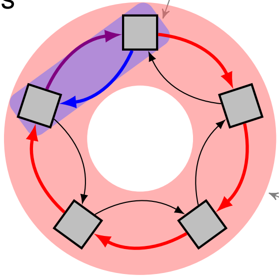
prevent ∞ interface nodes

Learning of Boxes: Knots

3. Complexity: max number of cutpoints in basic knots

Learning of Boxes: Knots

3. Complexity: max number of cutpoints in basic knots

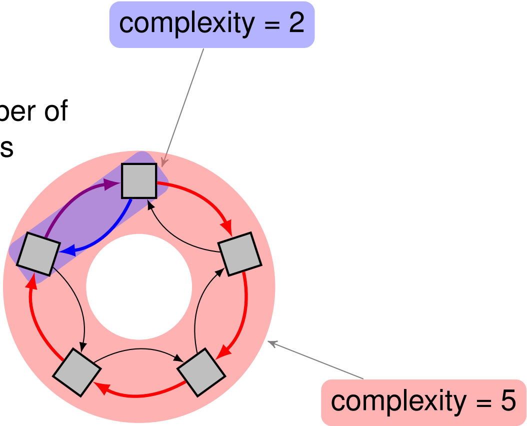


complexity = 2

complexity = 5

Learning of Boxes: Knots

3 Complexity: max number of cutpoints in basic knots



- Find basic knots with $1, 2, \ldots$ cut-points.

Widening Revisited

- Learning and folding of boxes in the abstraction loop:

Widening Revisited

- Learning and folding of boxes in the abstraction loop:

The Goal

Fold boxes that will, after abstraction, appear on cycles of automata.

⇒ hide unboundedly many cut-points

Widening Revisited

- Learning and folding of boxes in the abstraction loop:

The Goal

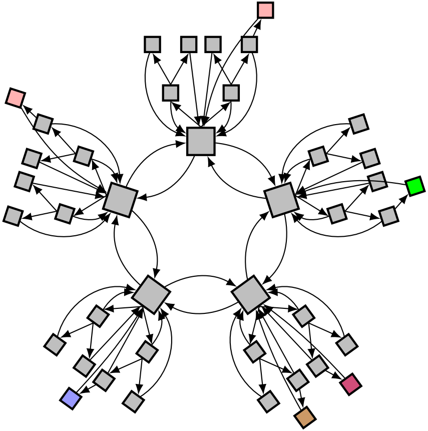
Fold boxes that will, after abstraction, appear on cycles of automata.

⇒ hide unboundedly many cut-points

```
Algorithm: Abstraction Loop
Unfold solo boxes
repeat
    Abstract
    Fold
until fixpoint
```

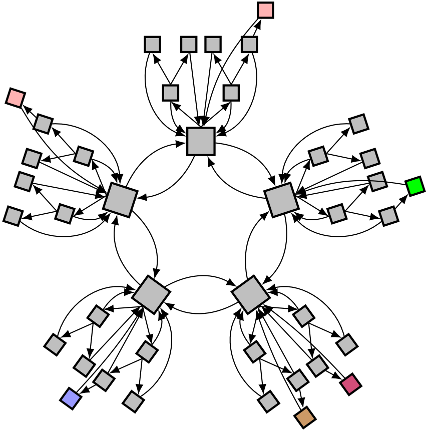
not on a cycle

Learning of Boxes: Example



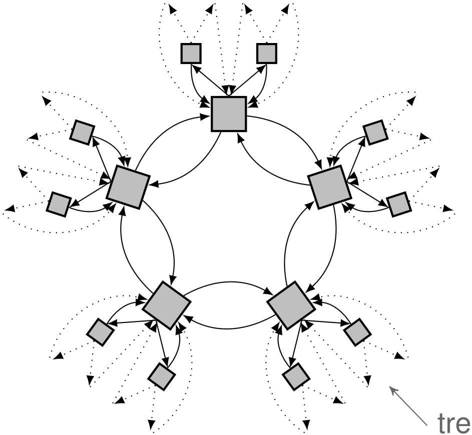
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

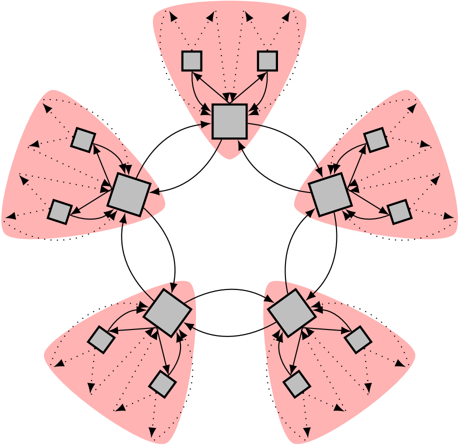
Learning of Boxes: Example



```
1  Unfold solo boxes
2  repeat
3      Abstract
4      Fold
5  until fixpoint
```

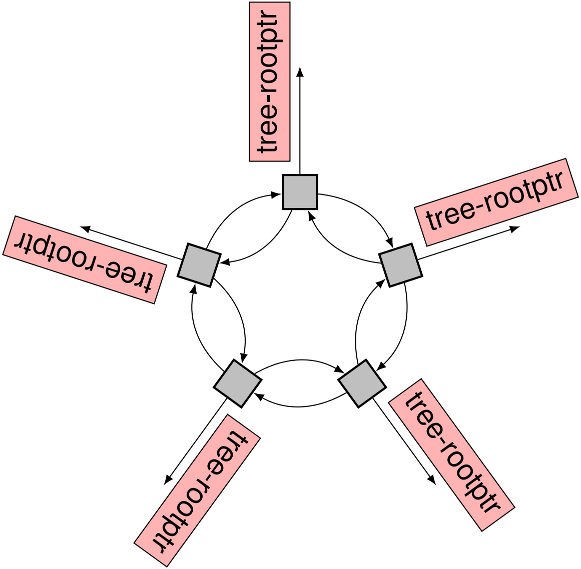
tree with root ptrs of any height

Learning of Boxes: Example



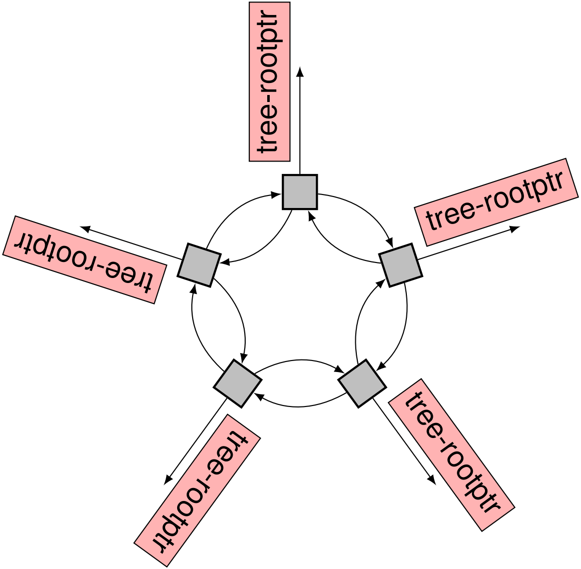
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



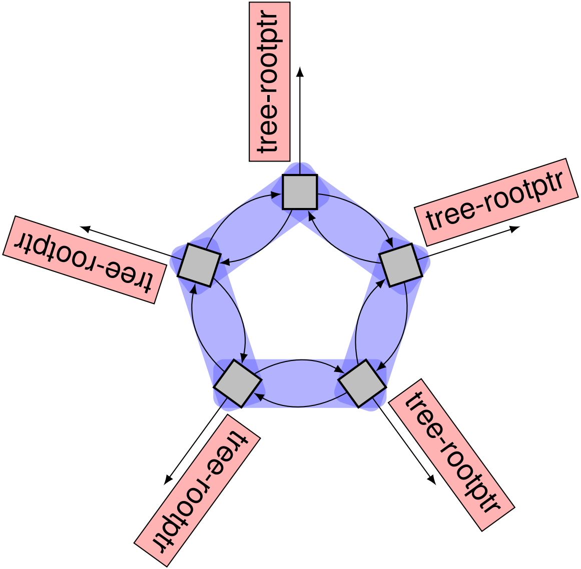
```
1 Unfold solo boxes
2 repeat
3       Abstract
4       Fold
5 until fixpoint
```

Learning of Boxes: Example



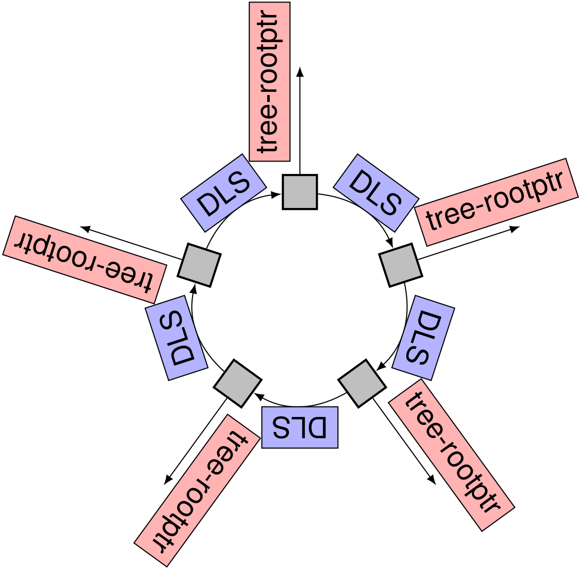
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



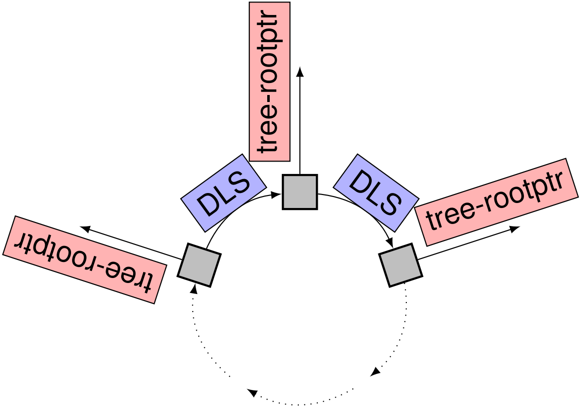
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



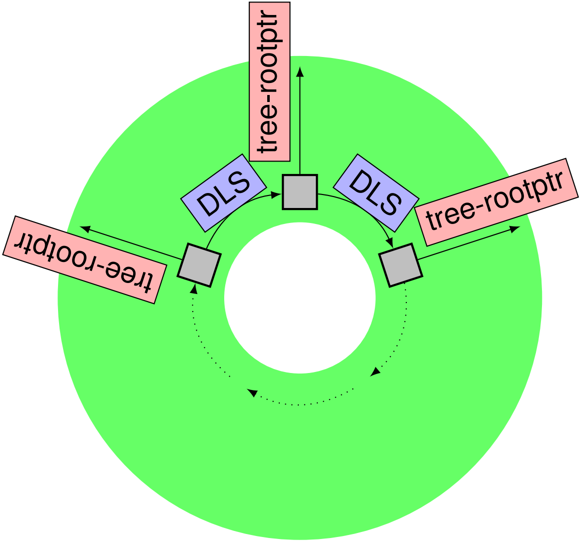
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



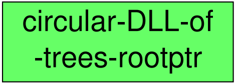
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

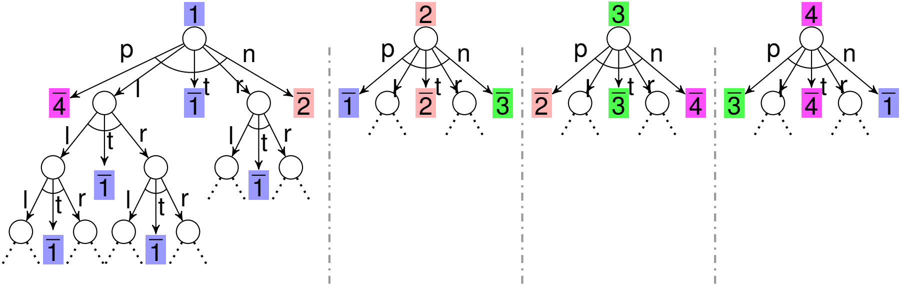
Learning of Boxes: Example



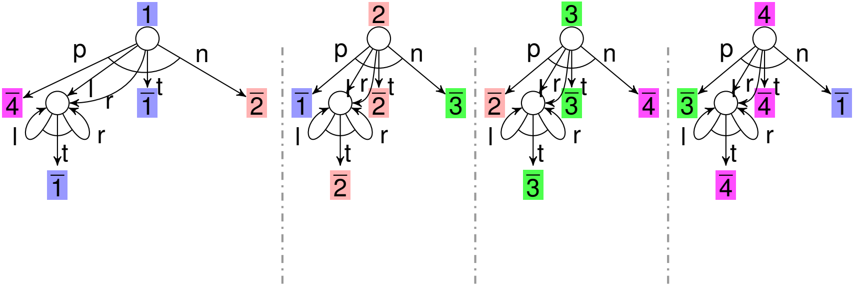
circular-DLL-of -trees-rootptr

```
Unfold solo boxes
repeat
    Abstract
    Fold
until fixpoint
```

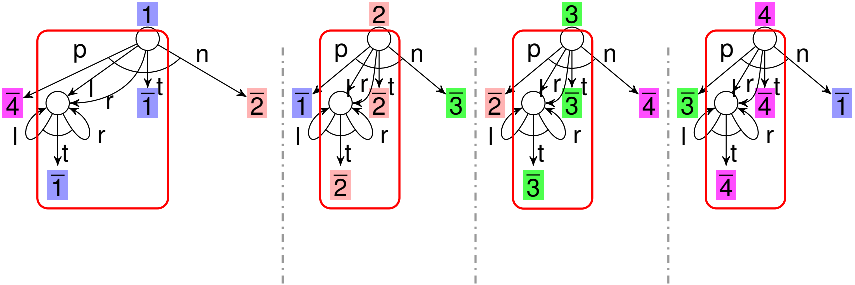
Learning, Folding, and Abstraction on FA

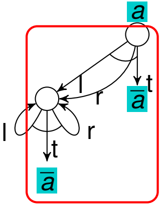


Learning, Folding, and Abstraction on FA

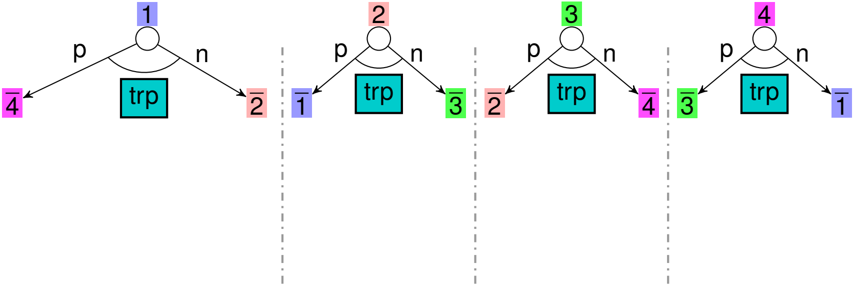


Learning, Folding, and Abstraction on FA

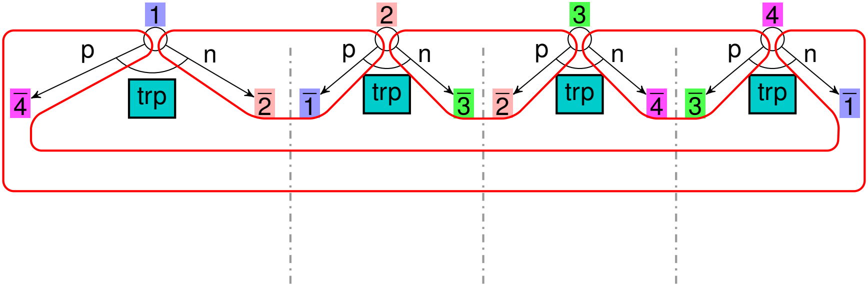


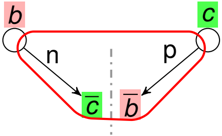


Learning, Folding, and Abstraction on FA

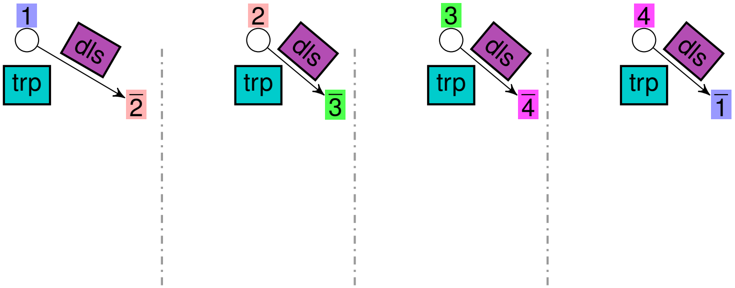


Learning, Folding, and Abstraction on FA

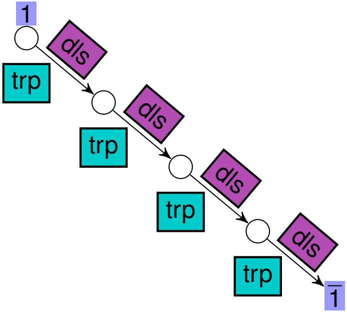




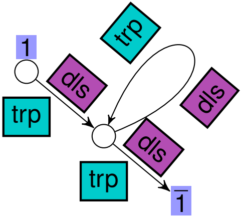
Learning, Folding, and Abstraction on FA



Learning, Folding, and Abstraction on FA



Learning, Folding, and Abstraction on FA



Experimental Results

- Implemented in the Forester tool as a gcc plugin.

Experimental Results

- Implemented in the Forester tool as a gcc plugin.
- Comparison with Predator (a state-of-the-art tool for lists),
  - winner of HeapManipulation and MemorySafety of SV-COMP'13:

Experimental Results

- Implemented in the Forester tool as a gcc plugin.
- Comparison with Predator (a state-of-the-art tool for lists),
  - winner of HeapManipulation and MemorySafety of SV-COMP'13:

Table : Results of the experiments [s]

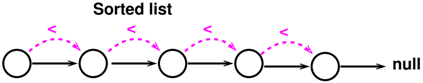
Example	**FA**	Predator
SLL (delete)	0.04	0.04
SLL (bubblesort)	0.04	0.03
SLL (mergesort)	0.15	0.10
SLL (insertsort)	0.05	0.04
SLL (reverse)	0.03	0.03
SLL+head	0.05	0.03
SLL of 0/1 SLLs	0.03	0.11
SLL$_{\text{Linux}}$	0.03	0.03
SLL of CSLLs	0.73	0.12
SLL of 2CDLLs$_{\text{Linux}}$	0.17	0.25
skip list$_2$	0.42	T
skip list$_3$	9.14	T
DLL (reverse)	0.06	0.03
DLL (insert)	0.07	0.05
DLL (insertsort$_1$)	0.40	0.11
DLL (insertsort$_2$)	0.12	0.05
DLL of CDLLs	1.25	0.22
DLL+subdata	0.09	T
CDLL	0.03	0.03
tree	0.14	Err
tree+parents	0.21	T
tree+stack	0.08	Err
tree (DSW) Deutsch-Schorr-Waite	0.40	Err
tree of CSLLs	0.42	Err

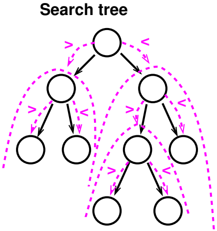
timeout

false positive

Tracking Relations over Data Values

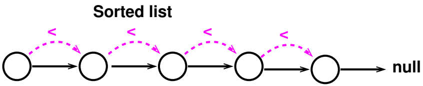
- Verify data-related properties such as sortedness.

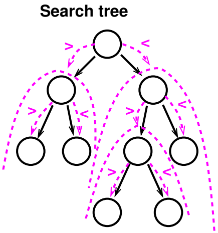




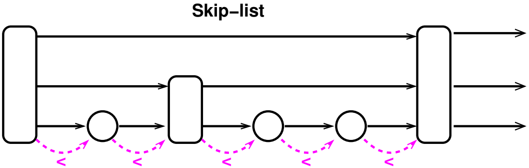
Tracking Relations over Data Values

- Verify data-related properties such as sortedness.





- Verify data-dependent memory safety/shape invariance.

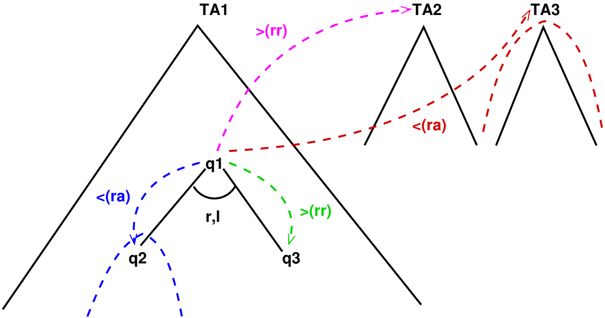


Forest Automata with Data Constraints

- TA rules extended with constraints
  - local: between states of a single rule,
  - global: between the LHS state and a root state of any TA

  comparing
  - two nodes: root-root (rr),
  - a node and all nodes of a tree: root-all (ra).

$$q1 \xrightarrow{r,l} (q2, q3) : \{0 <_{ra} 1, 0 <_{rr} 2, 0 <_{ra} \mathit{TA2}, 0 >_{rr} \mathit{TA3}\}$$



Operations on FA with Data Constraints

- Saturation:
  - Adds data constraints implied by the existing ones.
  - Improves precision of other operations.

Operations on FA with Data Constraints

- Saturation:
  - Adds data constraints implied by the existing ones.
  - Improves precision of other operations.

- Abstract transformers:
  - local constraints change to global when splitting TA,
  - global constraints change to local when merging TA, or they are dropped when relating distant states.

Operations on FA with Data Constraints

- Saturation:
  - Adds data constraints implied by the existing ones.
  - Improves precision of other operations.
- Abstract transformers:
  - local constraints change to global when splitting TA,
  - global constraints change to local when merging TA, or they are dropped when relating distant states.
- Inclusion checking, simulation reduction, abstraction:
  - Translation to ordinary FA
    - by embedding constraints into alphabet symbols.
  - Use of ordinary FA algorithms.

Experimental Results

Support for ordering relations implemented in an extension of Forester.

Example	time
SLL insert	0.06
SLL delete	0.08
SLL reverse	0.07
SLL bubblesort	0.13
SLL insertsort	0.10

Example	time
DLL insert	0.14
DLL delete	0.38
DLL reverse	0.16
DLL bubblesort	0.39
DLL insertsort	0.43

Example	time
BST insert	6.87
BST delete	114.00
BST left rotate	7.35
BST right rotate	6.25

Example	time
SL$_2$ insert	9.65
SL$_2$ delete	10.14
SL$_3$ insert	56.99
SL$_3$ delete	57.35

Conclusion

Shape analysis with forest automata:

- Fully automated, quite flexible.
- The Forester tool – a gcc plugin:

`http://www.fit.vutbr.cz/research/groups/verifit/tools/forester`

- Successfully verified:
  - (singly/doubly linked (circular)) lists (of (. . .) lists),
  - trees (with additional pointers),
  - skip lists,
  - ordered data structures.
- Not covered here:
  - support for pointer arithmetic: lists with embedded heads, ...

Future Work

- Cleaning and optimizing Forester.
- Adding a full support of the gcc intermediate code.
- Adding a CEGAR loop:
  - **red**-**black** trees, . . .
- Allowing Forester to run on incomplete code.
- Recursive boxes:
  - B+ trees, . . .
- Concurrent data structures:
  - lockless skip lists, . . .