Fully Automated Shape Analysis Based on Forest Automata†

Parosh A. Abdulla **Lukáš Holík** Bengt Jonsson
Ondřej Lengál Cong Quy Trinh Adam Rogalewicz
Jiří Šimáček **Tomáš Vojnar**

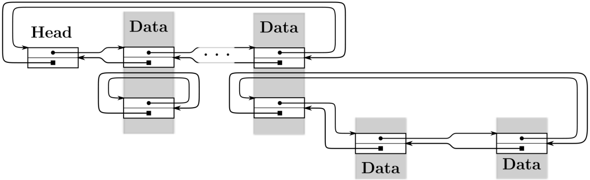
Brno University of Technology, Czech Republic
Uppsala University, Sweden

5th WAVAS

† Published in *Proc. of CAV'11, CAV'13, ATVA'13*

Shape Analysis

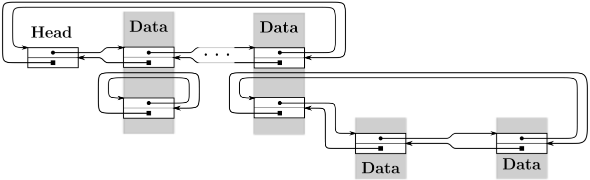
- Shape analysis:
  - reasoning about programs with dynamic linked data structures
  - notoriously difficult: infinite sets of complex graphs



  - memory safety: invalid dereferences, double free, memory leakage
  - error line reachability (assertions), shape invariance (testers), ...

Shape Analysis

- Shape analysis:
  - reasoning about programs with dynamic linked data structures
  - notoriously difficult: infinite sets of complex graphs



  - memory safety: invalid dereferences, double free, memory leakage
  - error line reachability (assertions), shape invariance (testers), ...

- Existing solutions:
  - often specialized (lists)
  - require human help (loop invariants, inductive predicates)
  - low scalability

Inspiration

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

Inspiration

- Separation Logic
  - ☺ local reasoning: well scalable
  - ☹ 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

- Introduced at CAV'11.

The Forest Automata-based Approach

- Introduced at CAV'11.
- Combines
  - ☺ flexibility of ARTMC

The Forest Automata-based Approach

- Introduced at CAV'11.
- Combines
  - ☺ flexibility of ARTMC

  with
  - ☺ scalability of SL

The Forest Automata-based Approach

- Introduced at CAV'11.
- Combines
  - ☺ flexibility of ARTMC

  with
  - ☺ scalability of SL

  by
  - splitting heaps into tree components

The Forest Automata-based Approach

- Introduced at CAV'11.
- Combines
  - ☺ flexibility of ARTMC

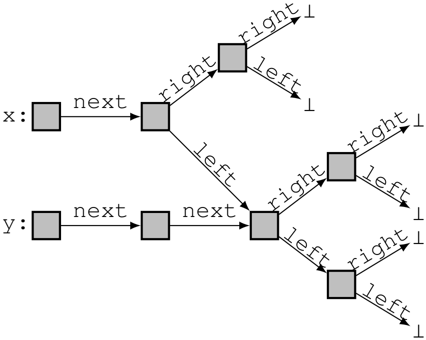
  with
  - ☺ scalability of SL

  by
  - splitting heaps into tree components

  and
  - using tree automata to represent sets of tree components of heaps

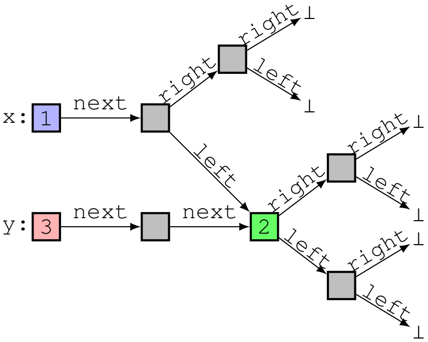
Heap Representation

- Forest decomposition of a heap



Heap Representation

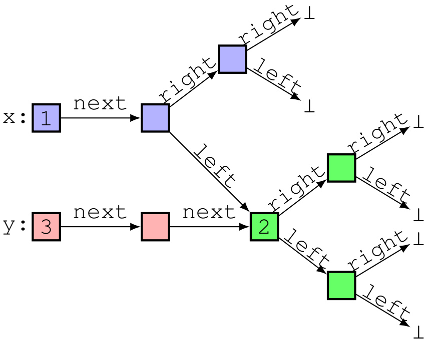
- Forest decomposition of a heap
  - Identify cut-points ← nodes referenced:
    - by variables, or
    - multiple times



Heap Representation

- Forest decomposition of a heap
  - Identify cut-points
  - Split the heap into tree components

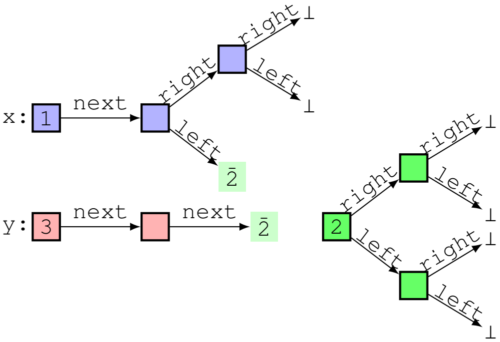
nodes referenced:
- by variables, or
- multiple times



Heap Representation

- Forest decomposition of a heap
  - Identify cut-points
  - Split the heap into tree components
  - References are explicit

nodes referenced:
- by variables, or
- multiple times



Heap Representation

- a heap $h \mapsto$ a forest $(\text{🌲}_1, \text{🌲}_2, \ldots, \text{🌲}_n)$

Heap Representation

- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_m), \ldots\}$

Heap Representation

- a heap $h \mapsto$ a forest $(T_1, T_2, \ldots, T_n)$
- a set of heaps $\mathcal{H} \mapsto \{(T_1, T_2, \ldots, T_n), (T'_1, T'_2, \ldots, T'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees $(T_1, T_2, \ldots, T_{\boxed{n}}), (T'_1, T'_2, \ldots, T'_{\boxed{n}})$

Heap Representation

- a heap $h \mapsto$ a forest $(T_1, T_2, \ldots, T_n)$
- a set of heaps $\mathcal{H} \mapsto \{(T_1, T_2, \ldots, T_n), (T'_1, T'_2, \ldots, T'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references

$$(T_1, T_2, \ldots, T_n), (T'_1, T'_2, \ldots, T'_n)$$

Heap Representation

- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

$$(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_{\boxed{n}}), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_{\boxed{n}})$$

Heap Representation

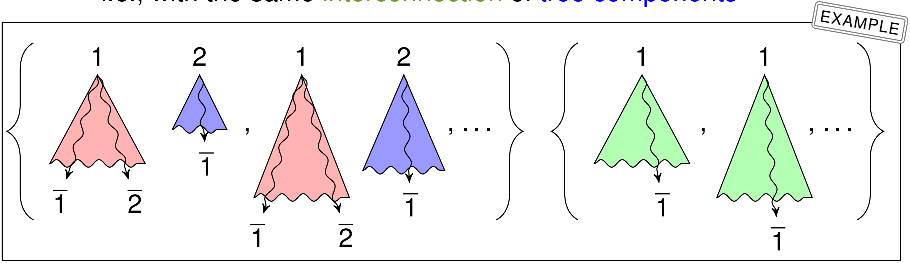
- a heap $h \mapsto$ a forest $(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \ldots, \twoheaduparrow'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \ldots, \twoheaduparrow'_n)$
  - i.e., with the same interconnection of tree components

Heap Representation

- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_{\boxed{n}}), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_{\boxed{n}})$
  - i.e., with the same interconnection of tree components



Heap Representation

- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_{\boxed{n}}), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_{\boxed{n}})$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:

$$\{(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_n), \ldots\}$$

Heap Representation

- a heap $h \mapsto$ a forest $(t_1, t_2, \dots, t_n)$
- a set of heaps $\mathcal{H} \mapsto \{(t_1, t_2, \dots, t_n), (t'_1, t'_2, \dots, t'_m), \dots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(t_1, t_2, \dots, t_n), (t'_1, t'_2, \dots, t'_n)$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:

$$\{(t_1, t_2, \dots, t_n), (t'_1, t'_2, \dots, t'_n), \dots\}$$

$$\rightsquigarrow$$

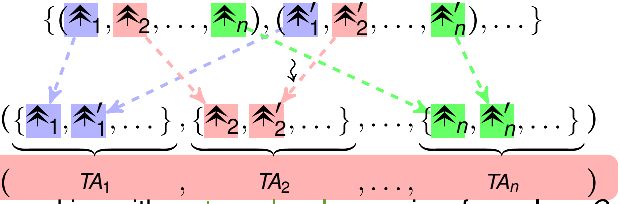
$$(\{t_1, t'_1, \dots\}, \{t_2, t'_2, \dots\}, \dots, \{t_n, t'_n, \dots\})$$

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

Heap Representation

- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \dots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \dots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \dots, \Uparrow'_m), \dots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(\Uparrow_1, \Uparrow_2, \dots, \Uparrow_{\boxed{n}}), (\Uparrow'_1, \Uparrow'_2, \dots, \Uparrow'_{\boxed{n}})$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:

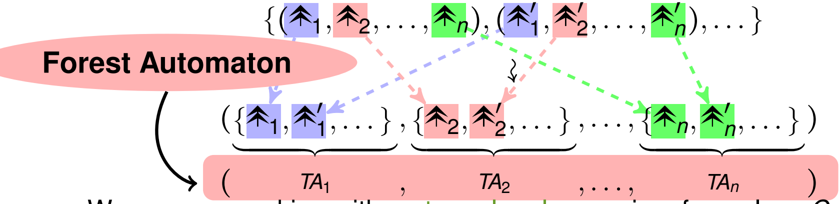


- We assume working with rectangular classes, i.e., for a class C, $(\Uparrow, \Uparrow), (\Uparrow, \Uparrow) \in C \Rightarrow (\Uparrow, \Uparrow), (\Uparrow, \Uparrow) \in C$, otherwise C is

Heap Representation

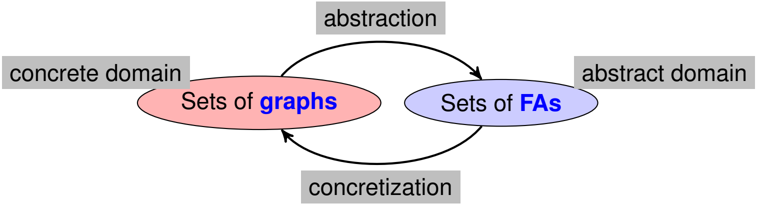
- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_{\boxed{n}}), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_{\boxed{n}})$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:

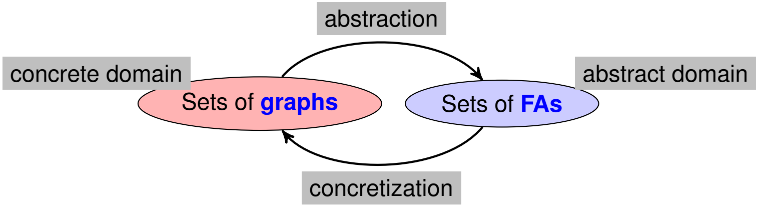


- We assume working with rectangular classes, i.e., for a class C, $(\Uparrow, \Uparrow), (\Uparrow, \Uparrow) \in C \Rightarrow (\Uparrow, \Uparrow), (\Uparrow, \Uparrow) \in C$, otherwise C is

Abstract Interpretation



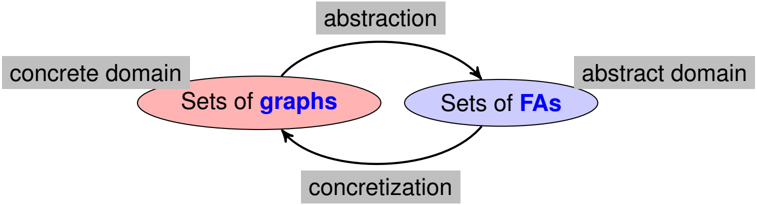
Abstract Interpretation



Statements

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

Abstract Interpretation

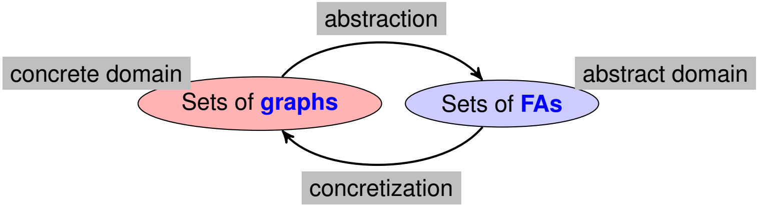


Statements

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

Abstract Transformers

Abstract Interpretation



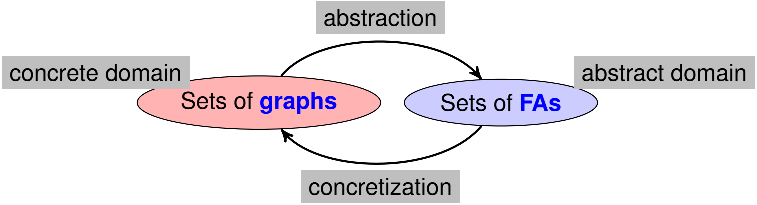
Statements

- `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

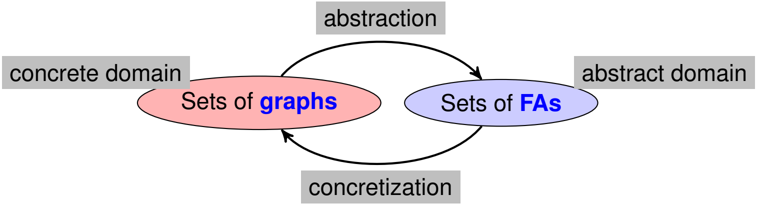
- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

append a TA

remove a TA

Abstract Interpretation



Statements

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

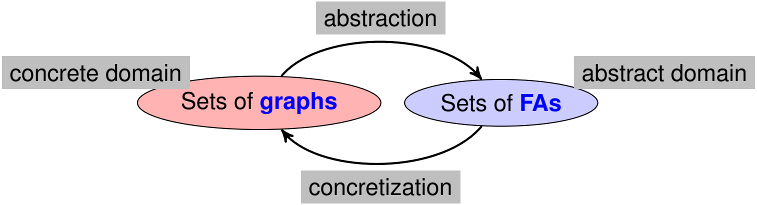
Abstract Transformers

append a TA

remove a TA

modify transitions

Abstract Interpretation



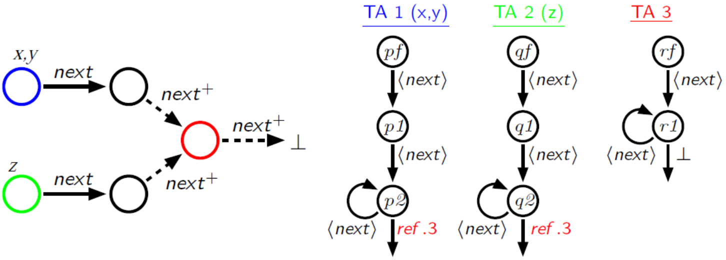
Statements

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

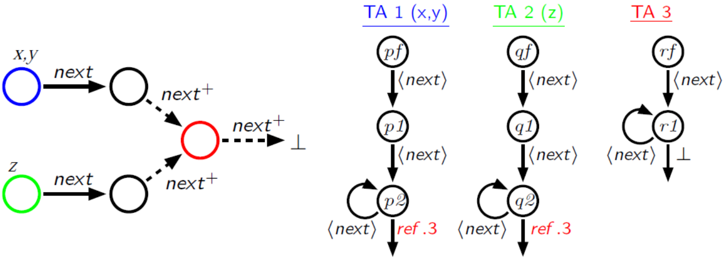
Abstract Transformers

- append a TA
- remove a TA
- modify transitions
- check symbols on transitions

Abstract Transformers for Pointer Updates

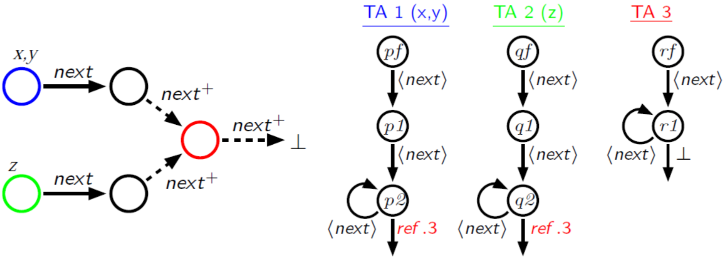


Abstract Transformers for Pointer Updates

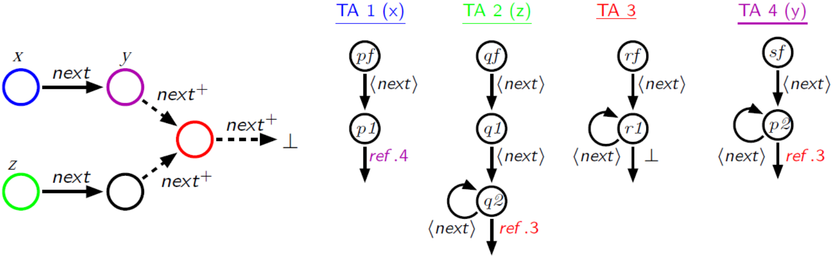


- y:=x.next

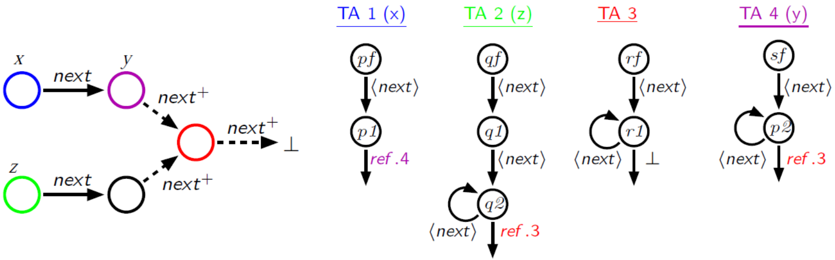
Abstract Transformers for Pointer Updates



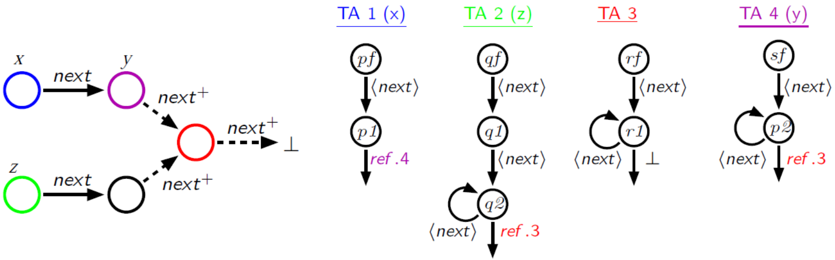
- y:=x.next



Abstract Transformers for Pointer Updates

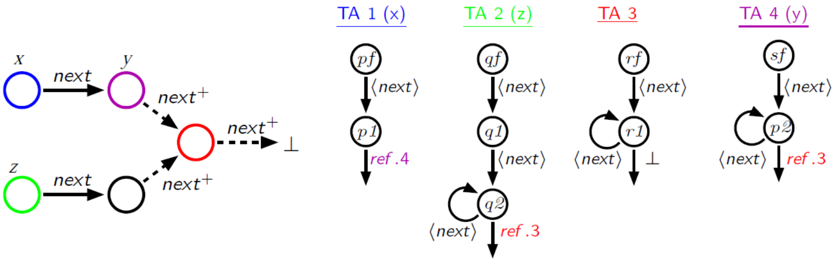


Abstract Transformers for Pointer Updates

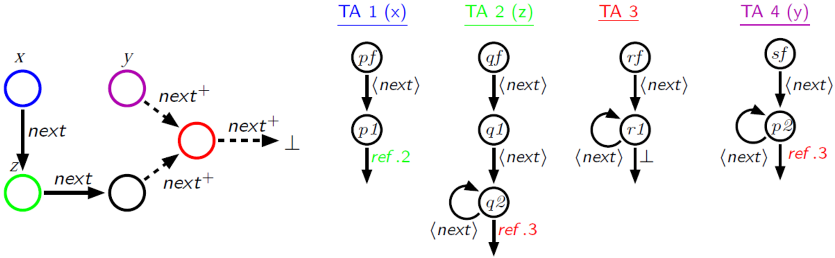


- x.next:=z;

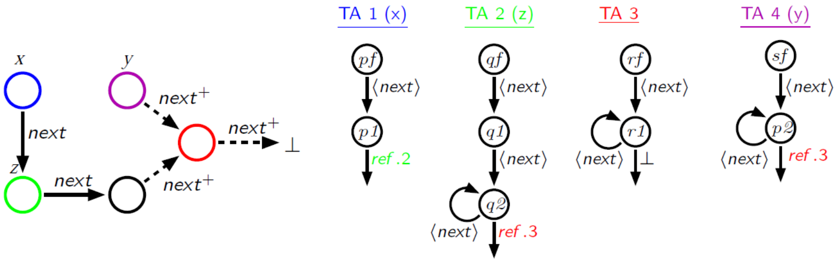
Abstract Transformers for Pointer Updates



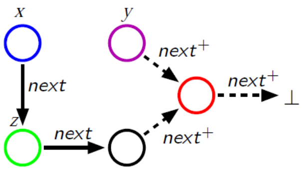
- x.next:=z;

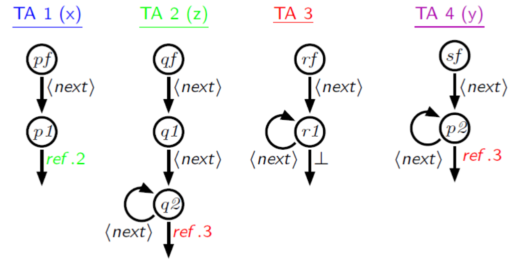


Abstract Transformers for Pointer Updates



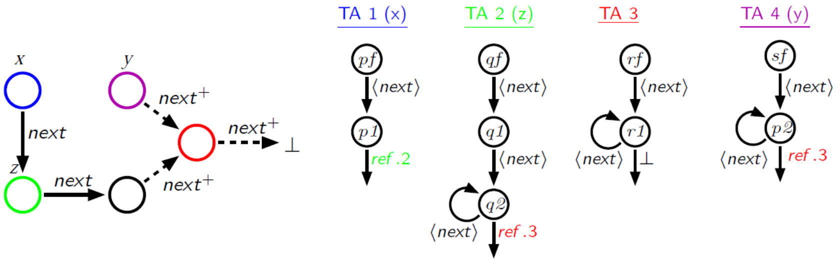
Abstract Transformers for Pointer Updates



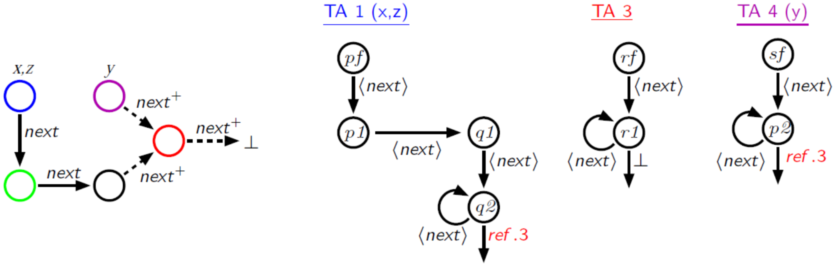


- z:=x;

Abstract Transformers for Pointer Updates



- z:=x;



Widening

- Abstraction on forest automata $(TA_1, \ldots, TA_n)$

Widening

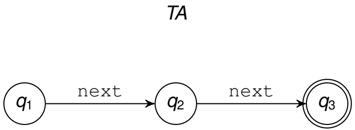
- Abstraction on forest automata $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\quad\rightsquigarrow\quad (TA_1^\alpha, \ldots, TA_n^\alpha)$

Widening

- Abstraction on forest automata $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\leadsto$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k

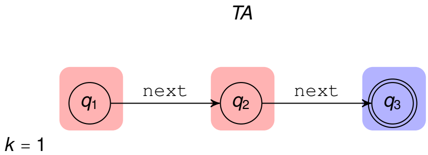
Widening

- Abstraction on forest automata $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k



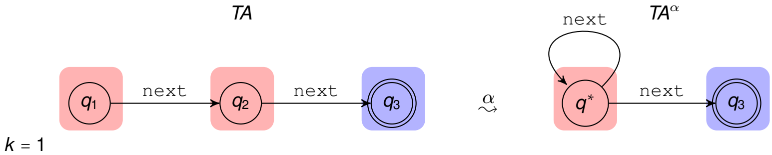
Widening

- Abstraction on forest automata $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k



Widening

- Abstraction on forest automata $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\leadsto$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k



Nondeterministic Tree Automata

- For efficiency reasons, we never determinize TAs.
- All operations done on NTAs, including:
  - inclusion checking: based on antichains and simulations,
    - discarding macro-states during an implicit subset construction,
    - inclusion on (normalized) FA can be checked component-wise —used for detecting the fixpoint
  - size reduction: based on simulation equivalences.
    - collapsing simulation-equivalent states.

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, ...

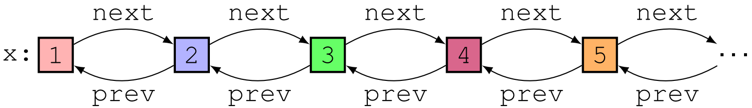
Summary

$$(\pitchfork_1, \pitchfork_2, \ldots, \pitchfork_n) \approx (\pitchfork'_1, \pitchfork'_2, \ldots, \pitchfork'_n)$$

$\ldots$

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
  - 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

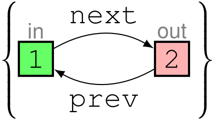
- Example: a box $\boxed{\text{DLS}}$

Hierarchical Forest Automata

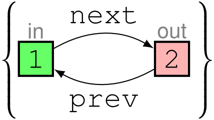
- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - intuition: replace repeated subgraphs by a single symbol, hiding some cut-points

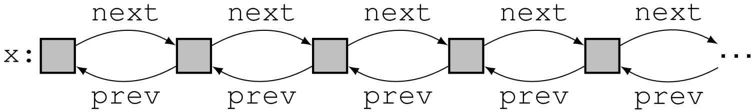
doubly linked segment

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



- Hierarchical Forest Automata
  - 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

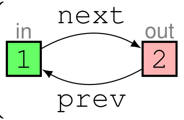
- Example: a box DLS : $\mathcal{L}($ DLS $) =$ 

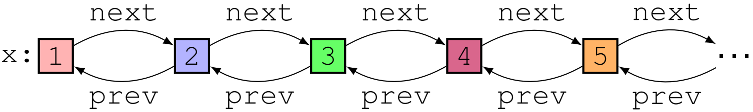


Hierarchical Forest Automata

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

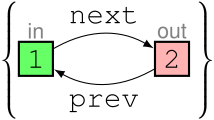


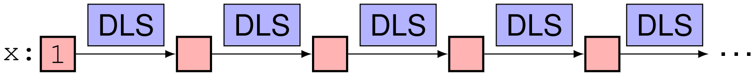
Hierarchical Forest Automata

- 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 $)$ =

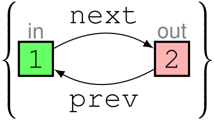


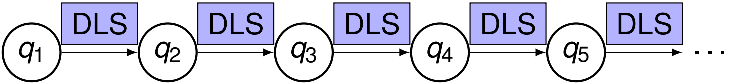


Hierarchical Forest Automata

- 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

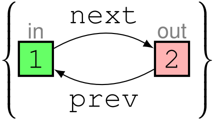
- Example: a box $\boxed{\text{DLS}}$: $\mathcal{L}(\boxed{\text{DLS}}) =$ 

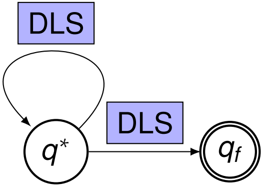


- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - intuition: replace repeated subgraphs by a single symbol, hiding some cut-points

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

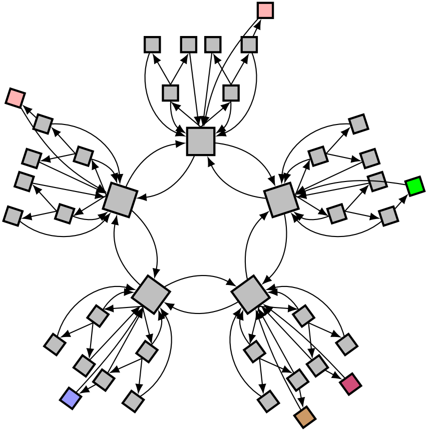
- compromise between

Learning of Boxes

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

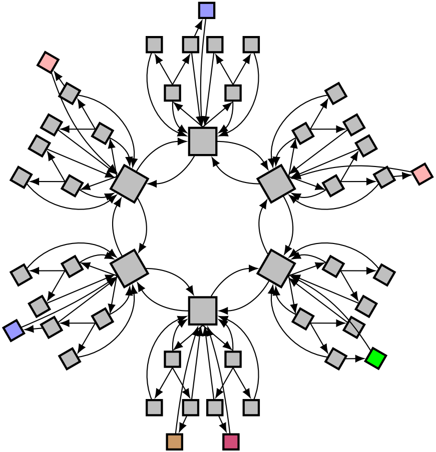
Learning of Boxes

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



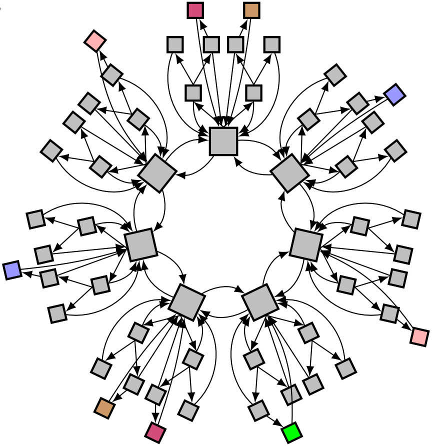
Learning of Boxes

- compromise between
  - reusability: use on different heaps of the same kind
    $\leadsto$ use small boxes



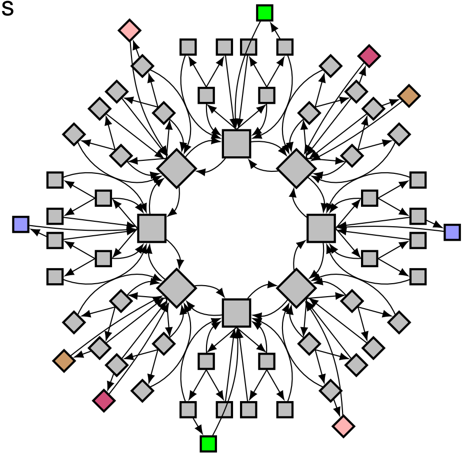
Learning of Boxes

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



Learning of Boxes

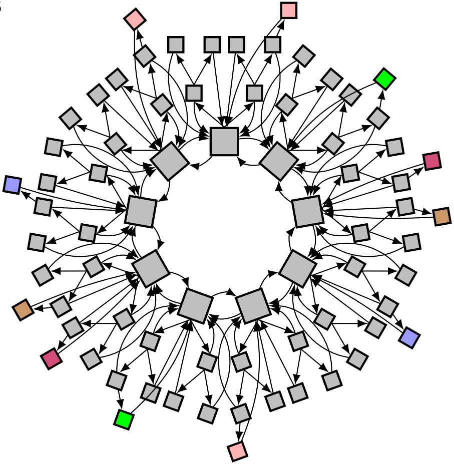
- compromise between
  - reusability: use on different heaps of the same kind
    $\rightsquigarrow$ use small boxes



Learning of Boxes

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

    $\rightsquigarrow$ use small boxes



Learning of Boxes

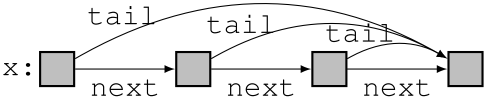
- compromise between
  - reusability: use on different heaps of the same kind
    $\rightsquigarrow$ use small boxes

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

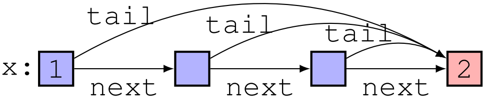
Learning of Boxes

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



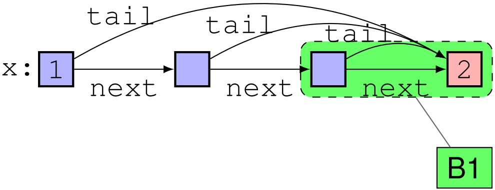
Learning of Boxes

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



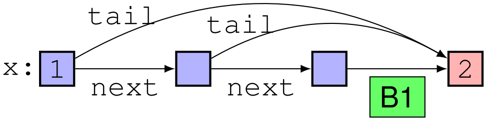
Learning of Boxes

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



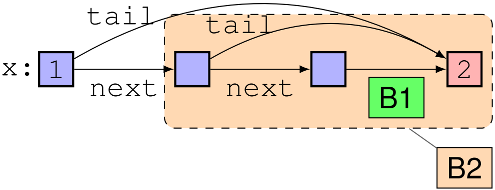
Learning of Boxes

- 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



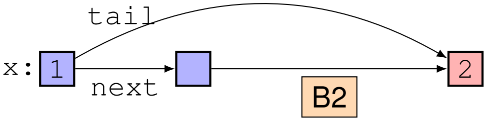
Learning of Boxes

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



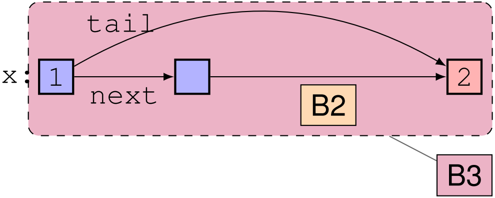
Learning of Boxes

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



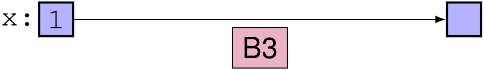
Learning of Boxes

- 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



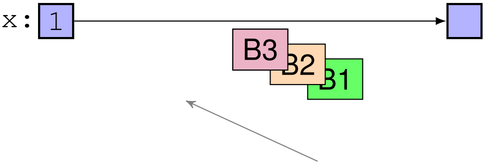
Learning of Boxes

- 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

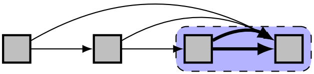
- compromise between
  - reusability: use on different heaps of the same kind
    $\leadsto$ use small boxes
  - ability to hide cut-points
    $\leadsto$ do not use too small boxes

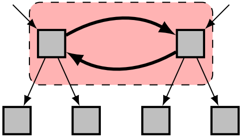


list of any length $\leadsto$ ∞ hierarchy

Learning of Boxes: Knots

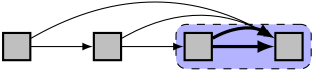
1. Smallest subgraphs meaningful to be folded:

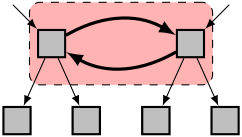




Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:

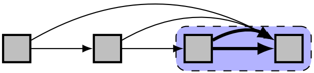


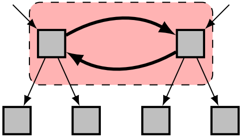


2. Handle interface

Learning of Boxes: Knots

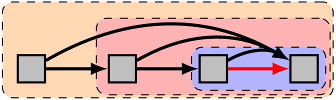
1. Smallest subgraphs meaningful to be folded:





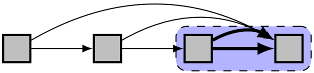
2. Handle interface
   - compose intersecting knots

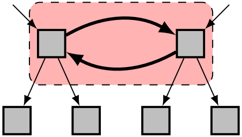
prevent ∞ nesting



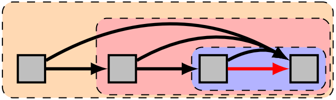
Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:



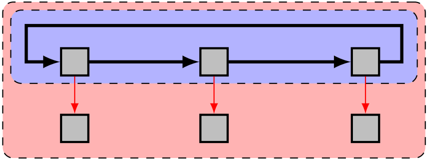


2. Handle interface
   - compose intersecting knots



prevent ∞ nesting

   - enclose paths from inner nodes to leaves

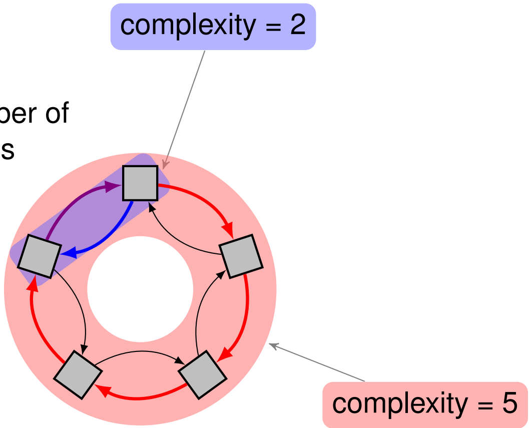


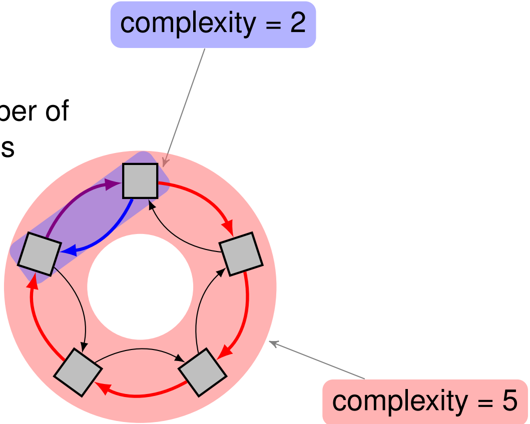
prevent ∞ interface nodes

Learning of Boxes: Knots

3. Complexity: max number of cutpoints in basic knots

3. Complexity: max number of cutpoints in basic 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.

$\Rightarrow$ hide unboundedly many cut-points

Widening Revisited

- learning and folding of boxes in the abstraction loop

The Goal

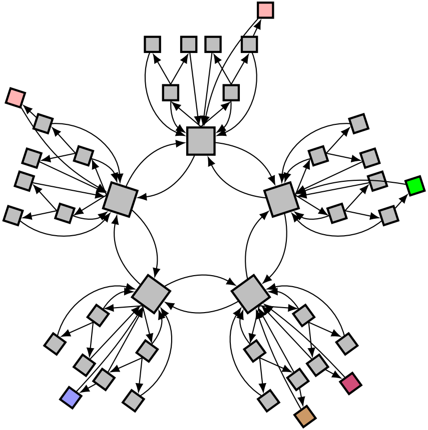
Fold boxes that will, after abstraction, appear on cycles of automata.

⇒ hide unboundedly many cut-points

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

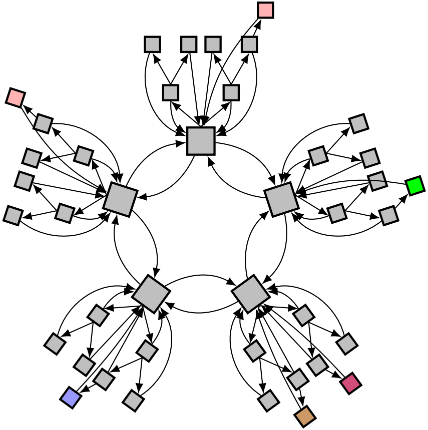
not on a cycle

Learning of Boxes: Example



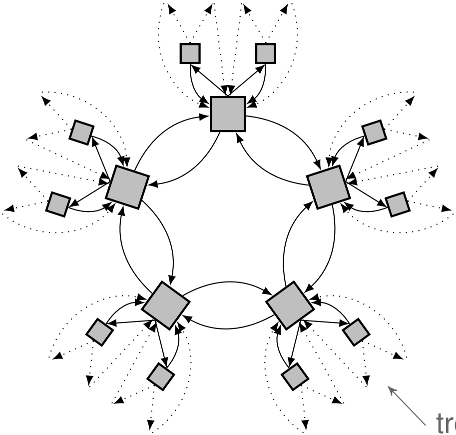
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example



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

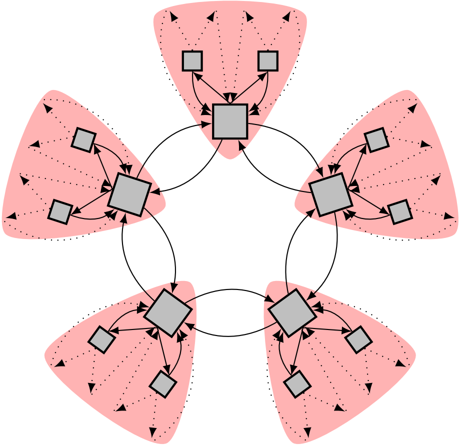
Learning of Boxes: Example



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

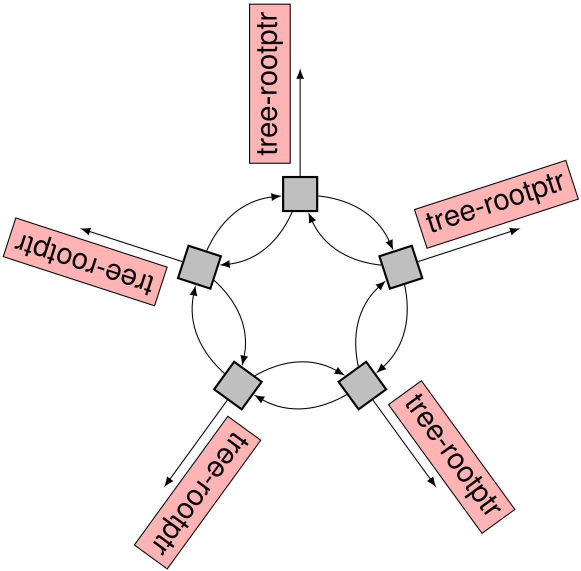
tree with root ptrs of any height

Learning of Boxes: Example



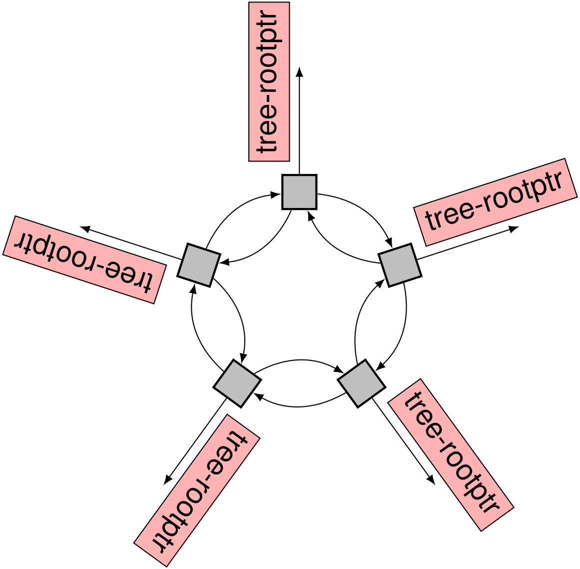
1 *Unfold solo boxes*
2 **repeat**
3 *Abstract*
4 *Fold*
5 **until** *fixpoint*

Learning of Boxes: Example



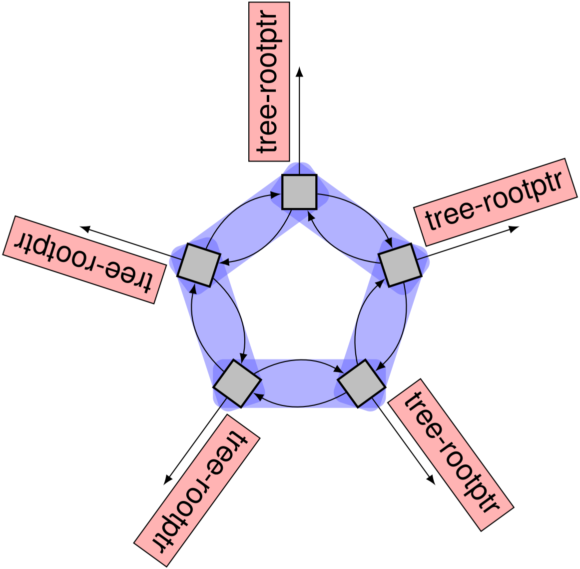
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example



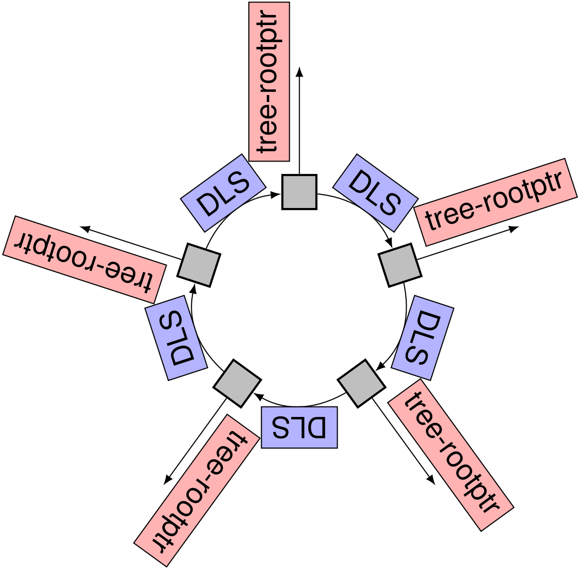
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example



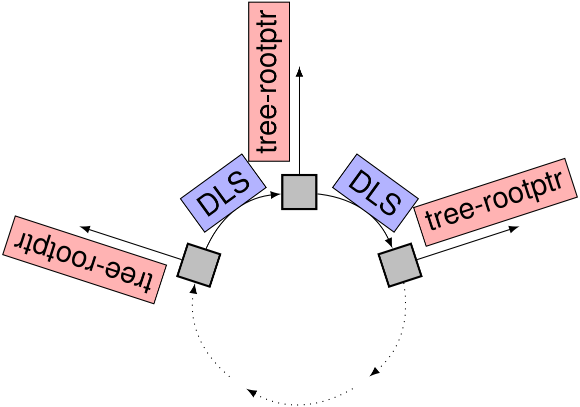
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example



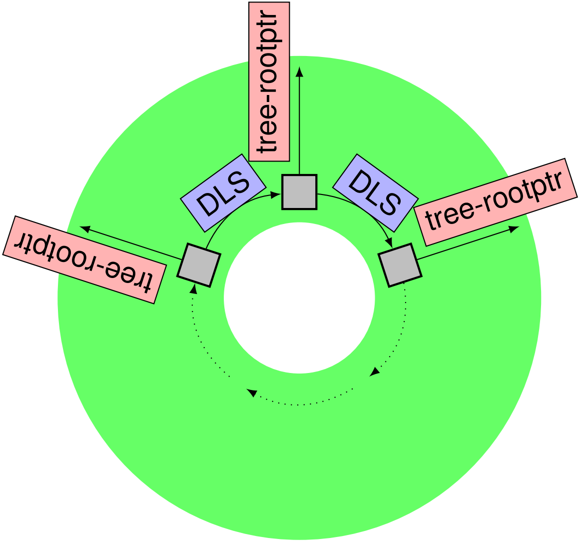
1 *Unfold solo boxes*
2 **repeat**
3 *Abstract*
4 *Fold*
5 **until** *fixpoint*

Learning of Boxes: Example



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

Learning of Boxes: Example



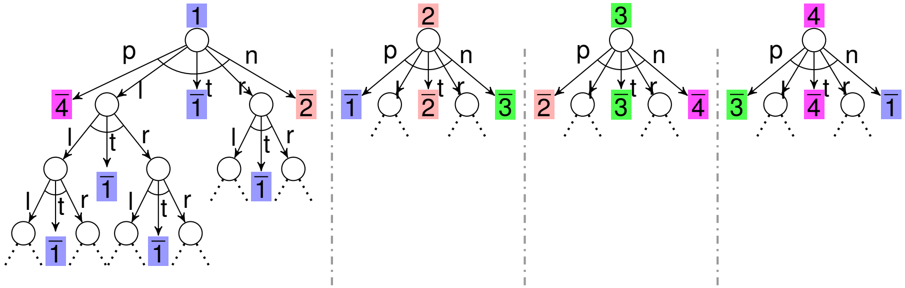
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example

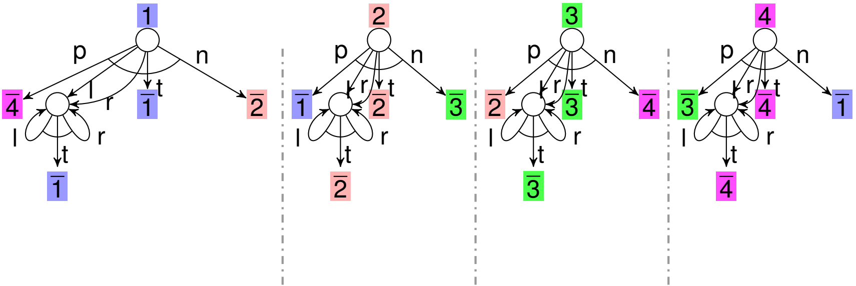
circular-DLL-of
-trees-rootptr

Unfold solo boxes
repeat
    Abstract
    Fold
until *fixpoint*

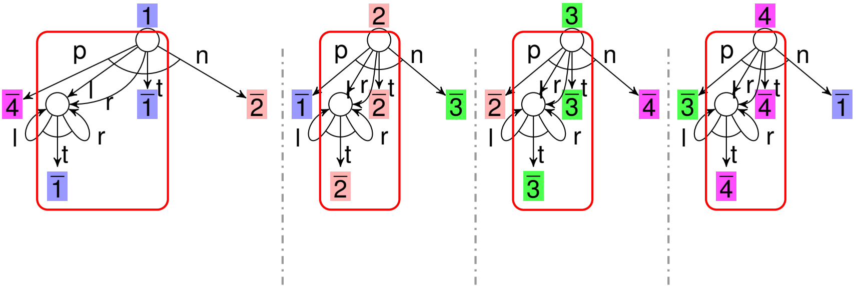
Learning, Folding, and Abstraction on FA

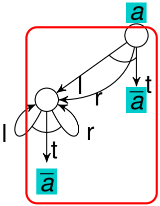


Learning, Folding, and Abstraction on FA

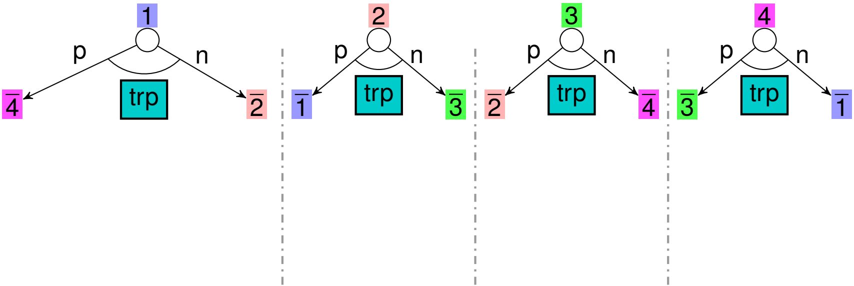


Learning, Folding, and Abstraction on FA

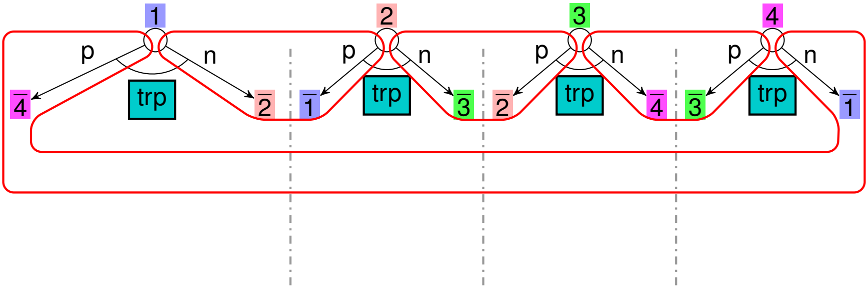


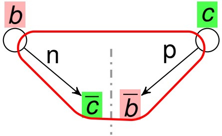


Learning, Folding, and Abstraction on FA

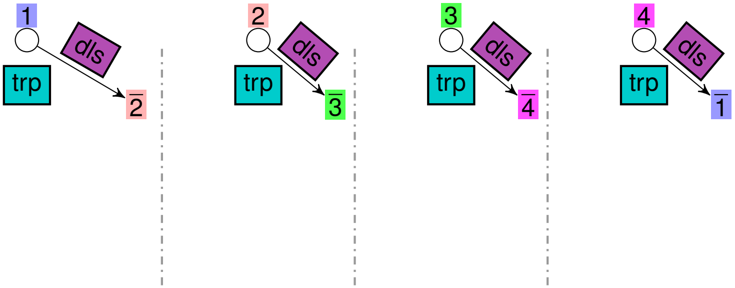


Learning, Folding, and Abstraction on FA

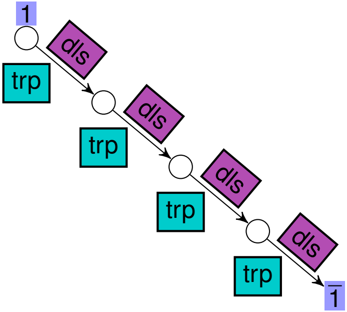




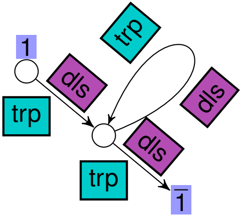
Learning, Folding, and Abstraction on FA



Learning, Folding, and Abstraction on FA



Learning, Folding, and Abstraction on FA



Experimental Results

- implemented in the **Forester** tool

Experimental Results

- implemented in the **Forester** tool
- 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
- 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	Example	FA	Predator
SLL (delete)	0.04	0.04	DLL (reverse)	0.06	0.03
SLL (bubblesort)	0.04	0.03	DLL (insert)	0.07	0.05
SLL (mergesort)	0.15	0.10	DLL (insertsort_1)	0.40	0.11
SLL (insertsort)	0.05	0.04	DLL (insertsort_2)	0.12	0.05
SLL (reverse)	0.03	0.03	DLL of CDLLs	1.25	0.22
SLL+head	0.05	0.03	DLL+subdata	0.09	T
SLL of 0/1 SLLs	0.03	0.11	CDLL	0.03	0.03
$\text{SLL}_{\text{Linux}}$	0.03	0.03	tree	0.14	Err
SLL of CSLLs	0.73	0.12	tree+parents	0.21	T
SLL of $\text{2CDLLs}_{\text{Linux}}$	0.17	0.25	tree+stack	0.08	Err
skip list_2	0.42	T	tree (DSW) Deutsch-Schorr-Waite	0.40	Err
skip list_3	9.14	T	tree of CSLLs	0.42	Err

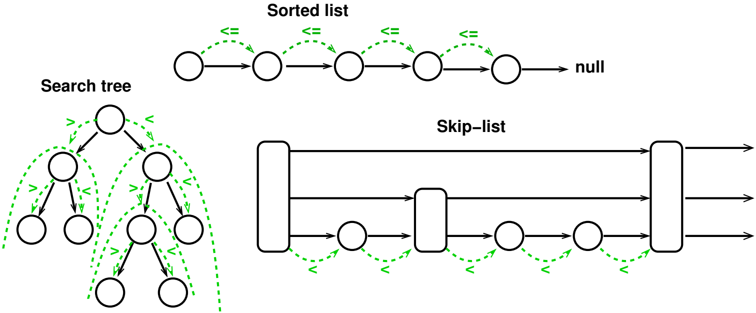
timeout

false positive

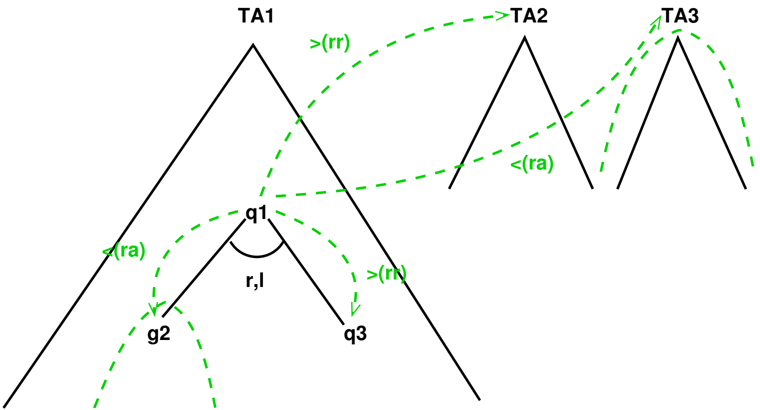
Tracking Relations Over Data Values

- Verify data-related properties such as sortedness.
- Verify memory safety even if it depends on relations over data.

Motivation



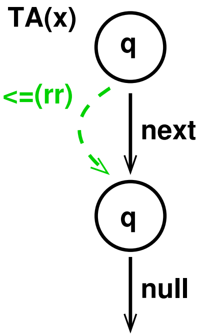
Forest Automata with Data Constraints



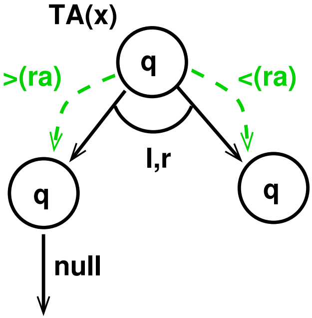
$$q1 \xrightarrow{r,l} (q2, q3) : \{0 <_{ra} 1, 0 <_{rr} 2, 0 <_{ra} TA2, 0 >_{rr} TA3\}$$

Examples of Encoded Structures

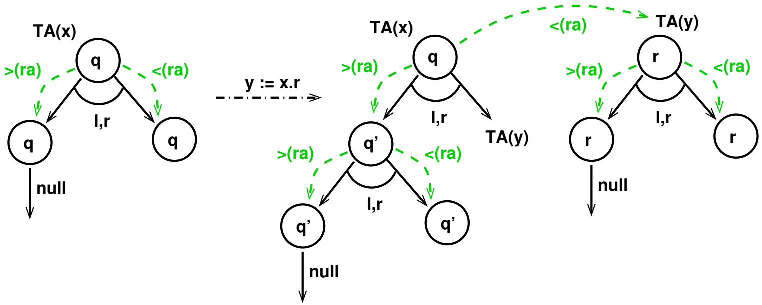
Sorted list



Search tree



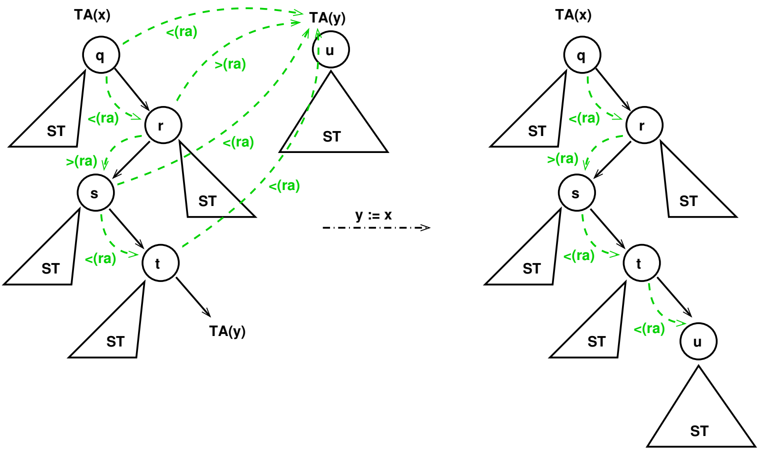
Update



TA(x)
q
>(ra)
<(ra)
l,r
q
q
null
y := x.r
TA(x)
q
>(ra)
<(ra)
l,r
TA(y)
q'
>(ra)
<(ra)
l,r
q'
q'
null
<(ra)
TA(y)
r
>(ra)
<(ra)
l,r
r
r
null

More Complex Update

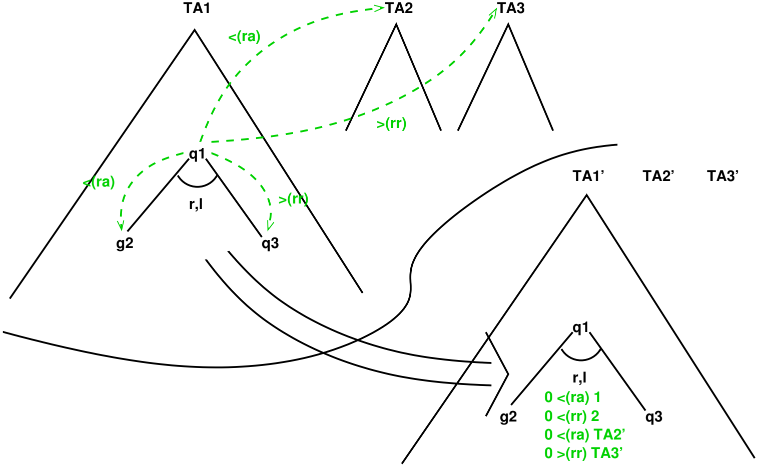
An intermediate state of a traversal of a search tree



Needed Machinery

- FA machinery must be extended with handling data constraints.
- Particularly, we need to be able to do:
  - Language Inclusion Check
  - Simulation Reduction
  - Abstraction
- This is done with a help of
  - Saturation which infers valid data constraints from existing ones.
  - Translation to ordinary FA and subsetquent use of ordinary FA algorithms.

Translation to Plain FA



Experimental Resutls

Example	time	Example	time
SLL insert	0.06	DLL insert	0.14
SLL delete	0.08	DLL delete	0.38
SLL reverse	0.07	DLL reverse	0.16
SLL bubblesort	0.13	DLL bubblesort	0.39
SLL insertsort	0.10	DLL insertsort	0.43

Example	time	Example	time
BST insert	6.87	SL_2 insert	9.65
BST delete	114.00	SL_2 delete	10.14
BST left rotate	7.35	SL_3 insert	56.99
BST right rotate	6.25	SL_3 delete	57.35

Conclusion

Shape analysis with forest automata:

- fully automated, very flexible

Conclusion

Shape analysis with forest automata:

- fully automated, very flexible
- the **Forester** tool
  - `http://www.fit.vutbr.cz/research/groups/verifit/tools/forester`

Conclusion

Shape analysis with forest automata:

- fully automated, very flexible
- the **Forester** tool
  - `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
  - tracking ordering relations
- not covered here:
  - support for pointer arithmetic

Future Work

- CEGAR loop
  - **red**-**black** trees, ...
- concurrent data structures
  - lockless skip lists, ...
- recursive boxes
  - B+ trees, ...