Fully Automated Shape Analysis Based on Forest Automat

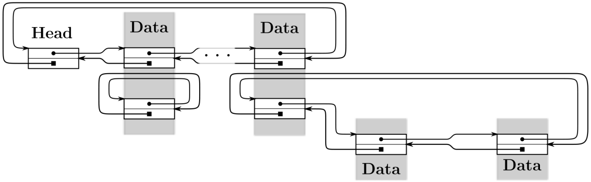
Parosh A. Abdulla Peter Habermehl Lukáš Holík
Bengt Jonsson **Ondřej Lengál** Cong Quy Trinh
Adam Rogalewicz Jiří Šimáček Tomáš Vojnar

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

August 22, 2016
IMS/NUS, Singapore

Shape Analysis

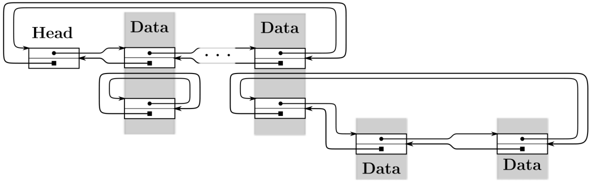
- Shape analysis:
  - reasoning about programs with dynamic linked data structures
  - notoriously difficult: infinite sets of complex graphs



  - 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
  - ⇒ still quite far from a general push-button solution

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

Forest Automata

- Combine
  - ☺ flexibility of ARTMC

The Forest Automata-based Approach

Forest Automata

- Combine
  - ☺ flexibility of ARTMC

  with
  - ☺ scalability of SL

The Forest Automata-based Approach

Forest Automata

- Combine
  - ☺ flexibility of ARTMC

  with
  - ☺ scalability of SL

  by
  - splitting heaps into tree components

The Forest Automata-based Approach

Forest Automata

- Combine
  - ☺ flexibility of ARTMC

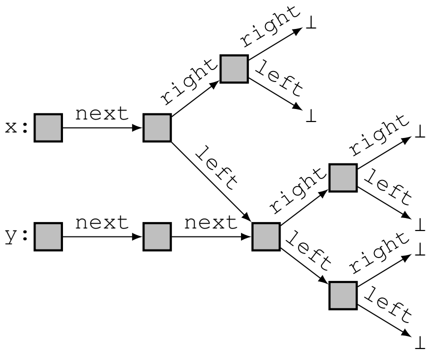
  with
  - ☺ scalability of SL

  by
  - splitting heaps into tree components

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

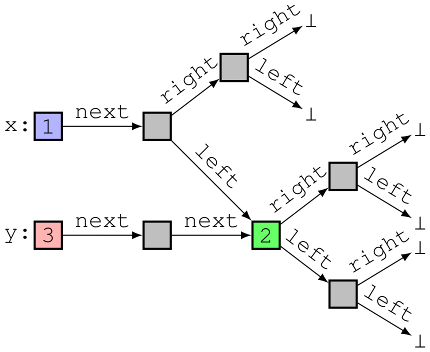
Heap Representation

- Forest decomposition of a heap



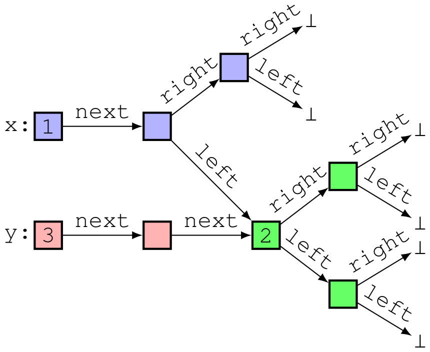
Heap Representation

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



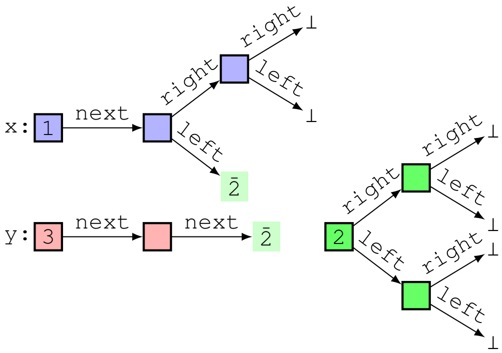
Heap Representation

- Forest decomposition of a heap
  - Identify cut-points ← nodes referenced:
    - by variables, or
    - multiple times
  - Split the heap into tree components



Heap Representation

- Forest decomposition of a heap
  - Identify cut-points ← nodes referenced:
    - by variables, or
    - multiple times
  - Split the heap into tree components
  - References are explicit



Heap Representation

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

Heap Representation

- a heap $h \mapsto$ a forest $(\text{⇞}_1, \text{⇞}_2, \ldots, \text{⇞}_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\text{⇞}_1, \text{⇞}_2, \ldots, \text{⇞}_n), (\text{⇞}'_1, \text{⇞}'_2, \ldots, \text{⇞}'_n), \ldots\}$
  - the same number of cut-points and the general structure of the heaps required

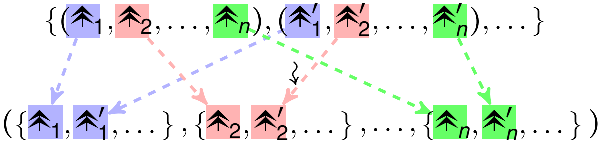
Heap Representation

- a heap $h \mapsto$ a forest $(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\twoheaduparrow_1, \twoheaduparrow_2, \ldots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \ldots, \twoheaduparrow'_n), \ldots\}$
  - the same number of cut-points and the general structure of the heaps required
- Cartesian representation of forests in $\mathcal{H}$:

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

Heap Representation

- a heap $h \mapsto$ a forest $(\twoheaduparrow_1, \twoheaduparrow_2, \dots, \twoheaduparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\twoheaduparrow_1, \twoheaduparrow_2, \dots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \dots, \twoheaduparrow'_n), \dots\}$
  - the same number of cut-points and the general structure of the heaps required
- Cartesian representation of forests in $\mathcal{H}$:

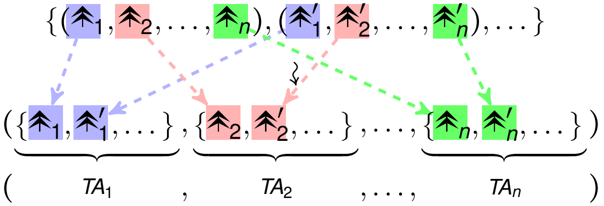


$\{(\twoheaduparrow_1, \twoheaduparrow_2, \dots, \twoheaduparrow_n), (\twoheaduparrow'_1, \twoheaduparrow'_2, \dots, \twoheaduparrow'_n), \dots\}$
$(\{\twoheaduparrow_1, \twoheaduparrow'_1, \dots\}, \{\twoheaduparrow_2, \twoheaduparrow'_2, \dots\}, \dots, \{\twoheaduparrow_n, \twoheaduparrow'_n, \dots\})$

- We assume working with rectangular sets, i.e., for a set C,
  (🟦↟, −), (−, 🟧↟) $\in C \Rightarrow$ (🟦↟, 🟧↟) $\in C$.

Heap Representation

- a heap $h \mapsto$ a forest $(⇞_1, ⇞_2, \ldots, ⇞_n)$
- a set of heaps $\mathcal{H} \mapsto \{(⇞_1, ⇞_2, \ldots, ⇞_n), (⇞'_1, ⇞'_2, \ldots, ⇞'_n), \ldots\}$
  - the same number of cut-points and the general structure of the heaps required
- Cartesian representation of forests in $\mathcal{H}$:



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

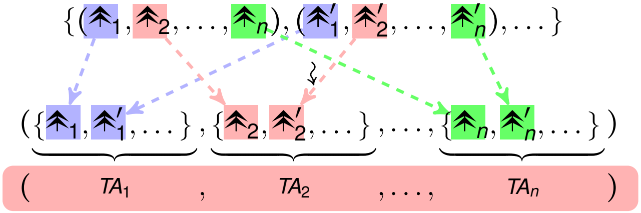
$$\wr$$

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

- We assume working with rectangular sets, i.e., for a set C, $(⇞, -), (-, ⇞) \in C \Rightarrow (⇞, ⇞) \in C$.

Heap Representation

- a heap $h \mapsto$ a forest $(\Uparrow_1, \Uparrow_2, \dots, \Uparrow_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\Uparrow_1, \Uparrow_2, \dots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \dots, \Uparrow'_n), \dots\}$
  - the same number of cut-points and the general structure of the heaps required
- Cartesian representation of forests in $\mathcal{H}$:

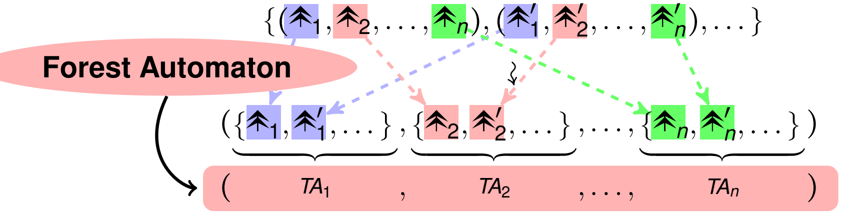


$\{(\Uparrow_1, \Uparrow_2, \dots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \dots, \Uparrow'_n), \dots\}$
$(\{\Uparrow_1, \Uparrow'_1, \dots\}, \{\Uparrow_2, \Uparrow'_2, \dots\}, \dots, \{\Uparrow_n, \Uparrow'_n, \dots\})$
$(TA_1, TA_2, \dots, TA_n)$

- We assume working with rectangular sets, i.e., for a set C, $(\Uparrow, -), (-, \Uparrow) \in C \Rightarrow (\Uparrow, \Uparrow) \in C$.

Heap Representation

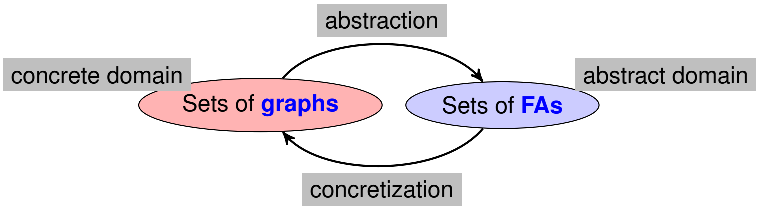
- a heap $h \mapsto$ a forest $(\text{↟}_1, \text{↟}_2, \ldots, \text{↟}_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\text{↟}_1, \text{↟}_2, \ldots, \text{↟}_n), (\text{↟}'_1, \text{↟}'_2, \ldots, \text{↟}'_n), \ldots\}$
  - the same number of cut-points and the general structure of the heaps required
- Cartesian representation of forests in $\mathcal{H}$:



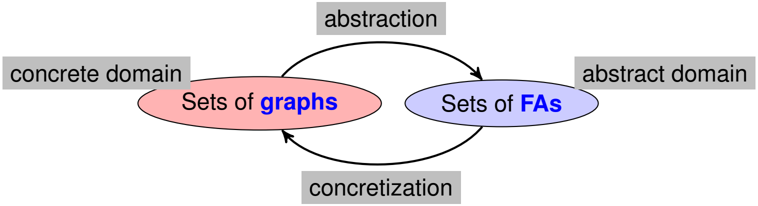
$\{(\text{↟}_1, \text{↟}_2, \ldots, \text{↟}_n), (\text{↟}'_1, \text{↟}'_2, \ldots, \text{↟}'_n), \ldots\}$
Forest Automaton
$(\{\text{↟}_1, \text{↟}'_1, \ldots\}, \{\text{↟}_2, \text{↟}'_2, \ldots\}, \ldots, \{\text{↟}_n, \text{↟}'_n, \ldots\})$
$(TA_1, TA_2, \ldots, TA_n)$

- We assume working with rectangular sets, i.e., for a set C,
  $(\text{↟}, -), (-, \text{↟}) \in C \Rightarrow (\text{↟}, \text{↟}) \in C$.

Abstract Interpretation



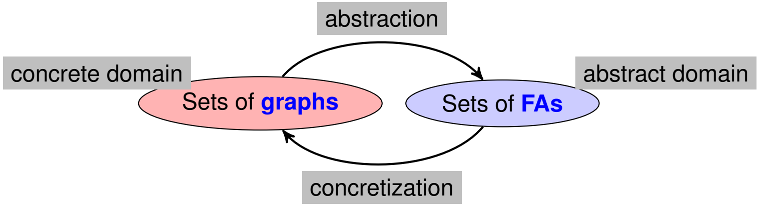
Abstract Interpretation



Statements

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

Abstract Interpretation

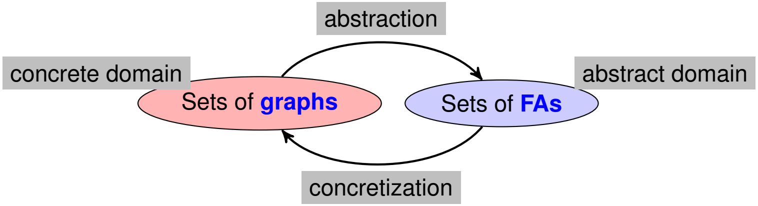


Statements

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

Abstract Transformers

Abstract Interpretation



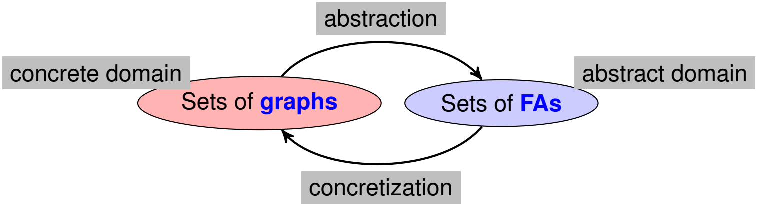
Statements

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

Abstract Transformers

$$(TA_1, \dots, TA_n) \leadsto (TA_1, \dots, TA_n, TA_{n+1})$$

Abstract Interpretation



Statements

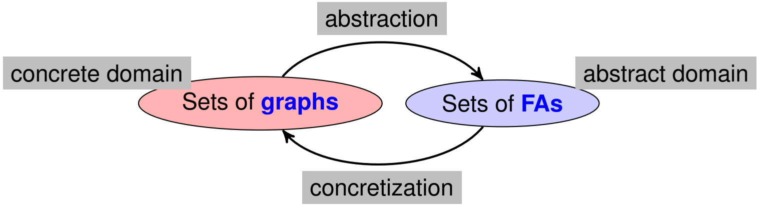
- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

$$(TA_1, \ldots, TA_n) \rightsquigarrow (TA_1, \ldots, TA_n, TA_{n+1})$$

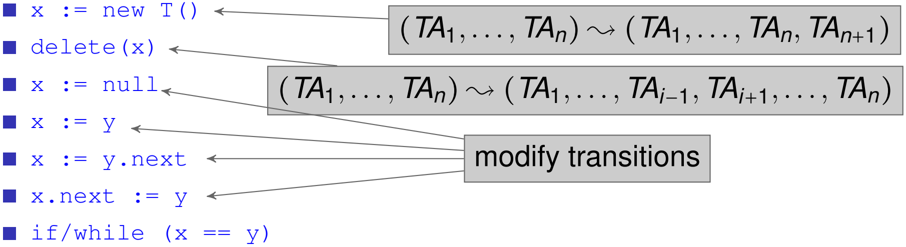
$$(TA_1, \ldots, TA_n) \rightsquigarrow (TA_1, \ldots, TA_{i-1}, TA_{i+1}, \ldots, TA_n)$$

Abstract Interpretation



Statements

Abstract Transformers



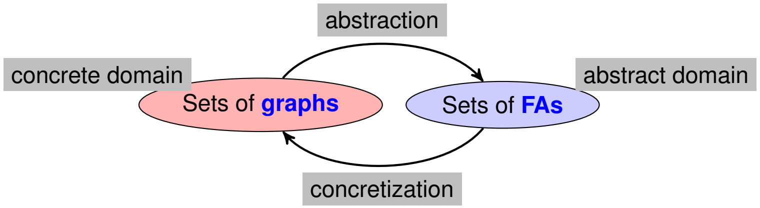
- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

$(TA_1, \ldots, TA_n) \rightsquigarrow (TA_1, \ldots, TA_n, TA_{n+1})$

$(TA_1, \ldots, TA_n) \rightsquigarrow (TA_1, \ldots, TA_{i-1}, TA_{i+1}, \ldots, TA_n)$

modify transitions

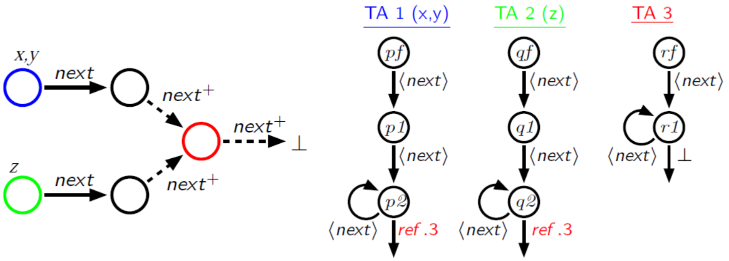
Abstract Interpretation



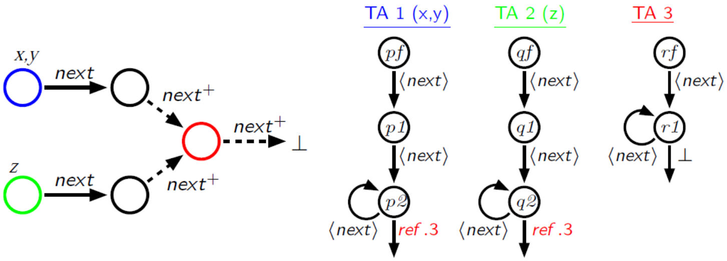
Statements **Abstract Transformers**

- `x := new T()` ← $(TA_1, \ldots, TA_n) \rightsquigarrow (TA_1, \ldots, TA_n, TA_{n+1})$
- `delete(x)` ← $(TA_1, \ldots, TA_n) \rightsquigarrow (TA_1, \ldots, TA_{i-1}, TA_{i+1}, \ldots, TA_n)$
- `x := null` ← modify transitions
- `x := y` ← modify transitions
- `x := y.next` ← modify transitions
- `x.next := y` ← modify transitions
- `if/while (x == y)` ← check symbols on transitions

Abstract Transformers for Pointer Updates

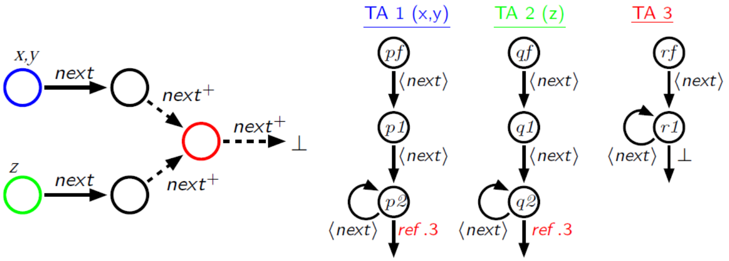


Abstract Transformers for Pointer Updates

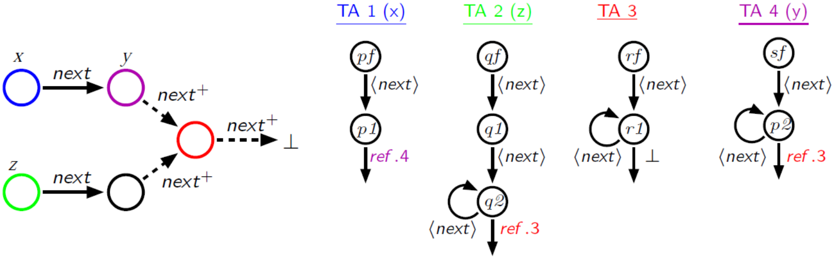


- y:=x.next

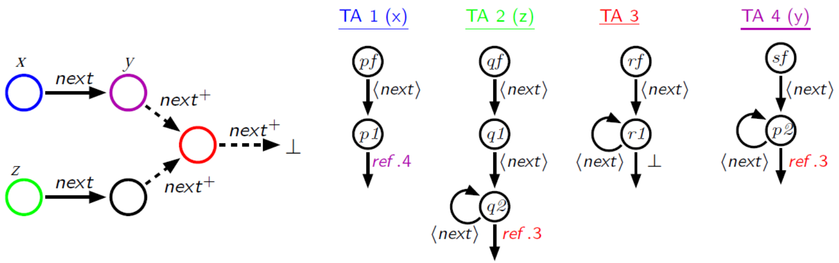
Abstract Transformers for Pointer Updates



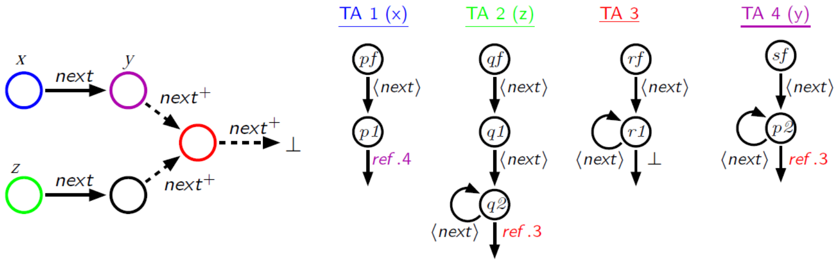
- y:=x.next



Abstract Transformers for Pointer Updates

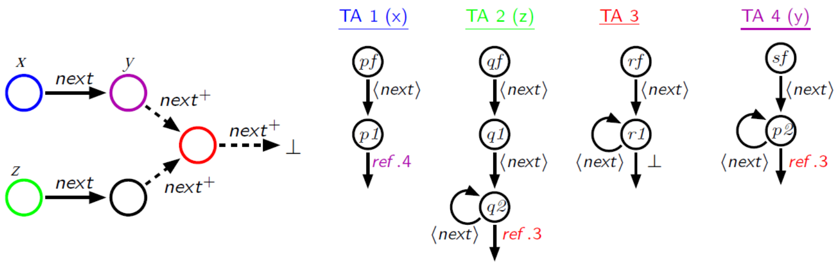


Abstract Transformers for Pointer Updates

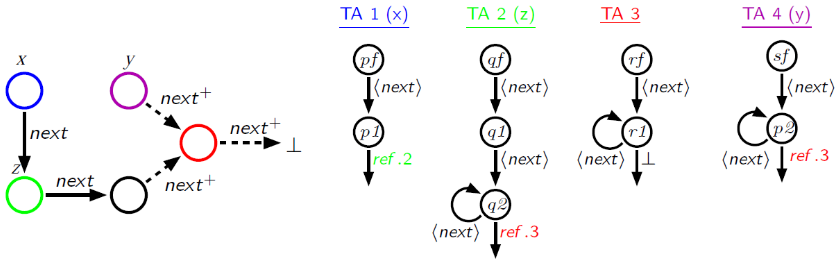


- x.next:=z;

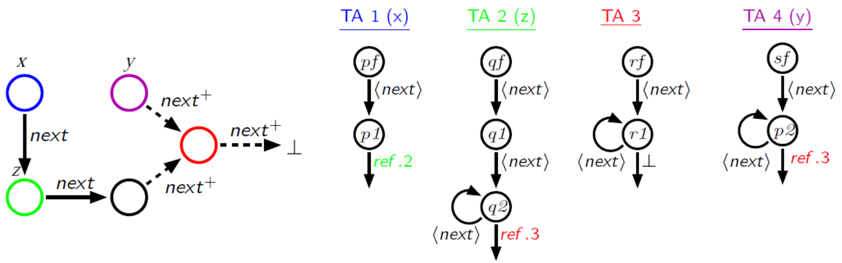
Abstract Transformers for Pointer Updates



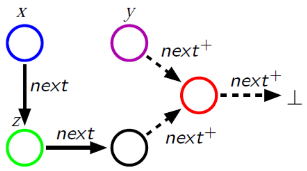
- x.next:=z;

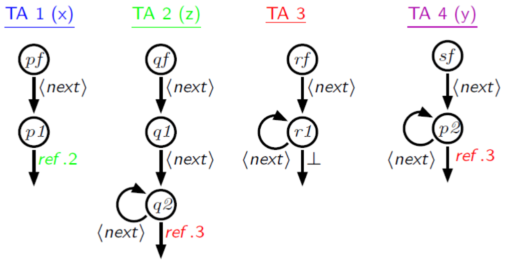


Abstract Transformers for Pointer Updates



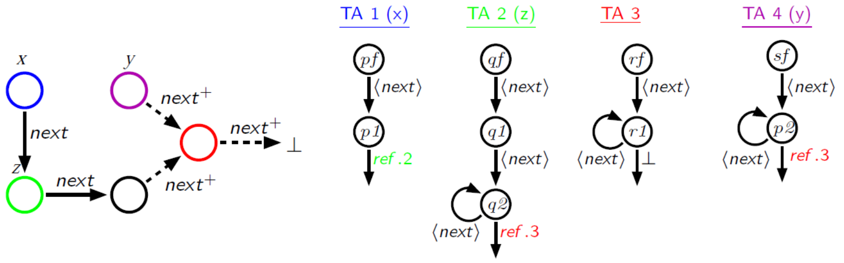
Abstract Transformers for Pointer Updates



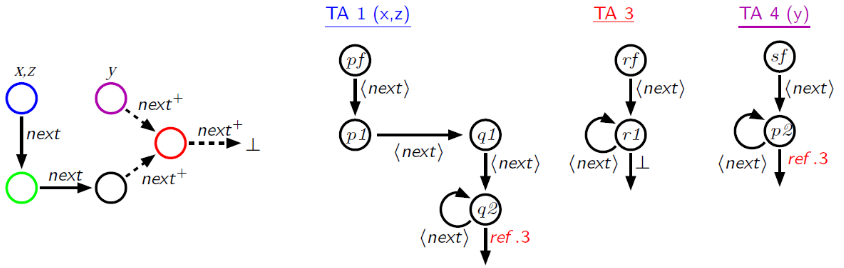


- z:=x;

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 $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$

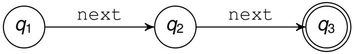
Widening

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

Widening

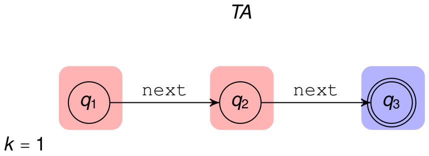
- Abstraction on forest automata $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k

TA



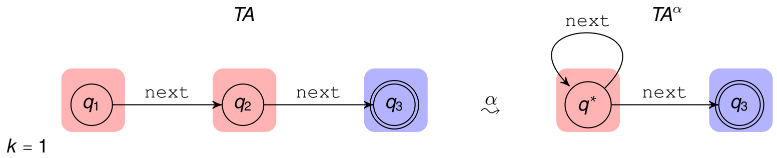
Widening

- 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



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,
  - size reduction: based on simulation equivalences.
    - collapsing simulation-equivalent states.

Summary

The so-far-presented:

Summary

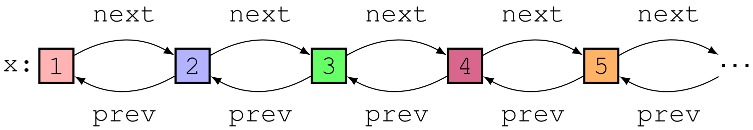
The so-far-presented:

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

Summary

The so-far-presented:

- ☺ works well for singly linked lists (SLLs), circular lists, trees, SLLs with head/tail pointers, trees with root pointers, ...
- ☹ fails for more complex data structures
  - unbounded number of cut-points $\rightsquigarrow$
    heaps with different numbers of cut-points need to be treated separately



- doubly linked lists (DLLs),
- 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

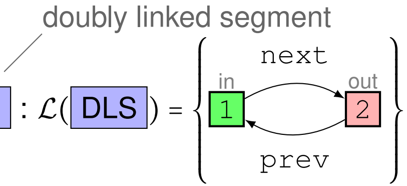
doubly linked segment

- 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

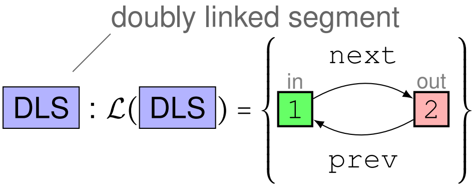
- Example: a box $\boxed{\text{DLS}}$: $\mathcal{L}(\boxed{\text{DLS}}) =$

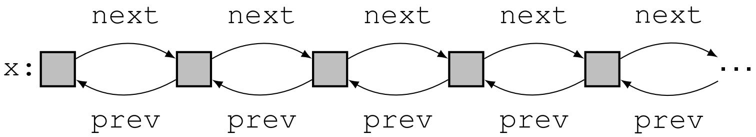


Hierarchical Forest Automata

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

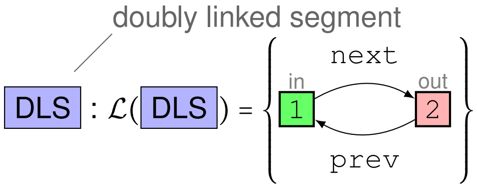


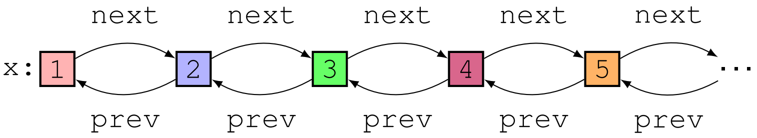


Hierarchical Forest Automata

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

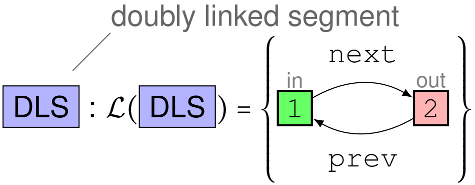


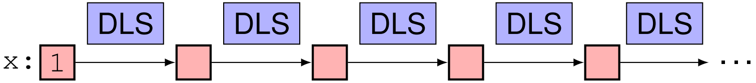


Hierarchical Forest Automata

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

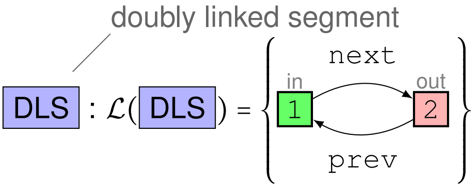


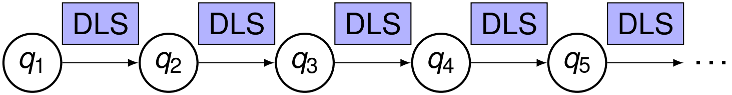


Hierarchical Forest Automata

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

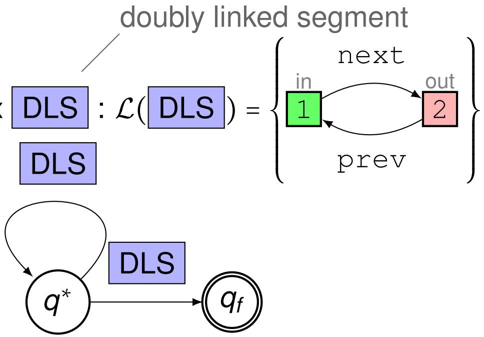




Hierarchical Forest Automata

- 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 : $\mathcal{L}($DLS$) = \{$ … $\}$



Learning of Boxes

The Challenge

How to find the "right" boxes?

Learning of Boxes

The Challenge

How to find the “right” boxes?

- database of boxes
- automatic discovery

Learning of Boxes

- compromise between

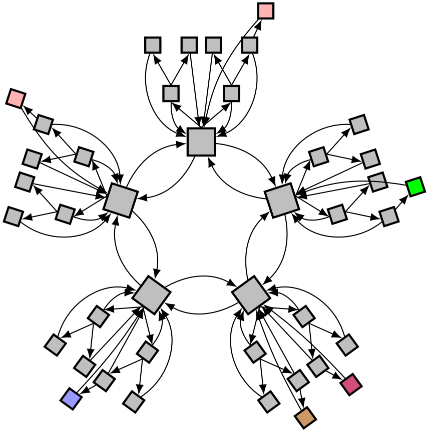
Learning of Boxes

- 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

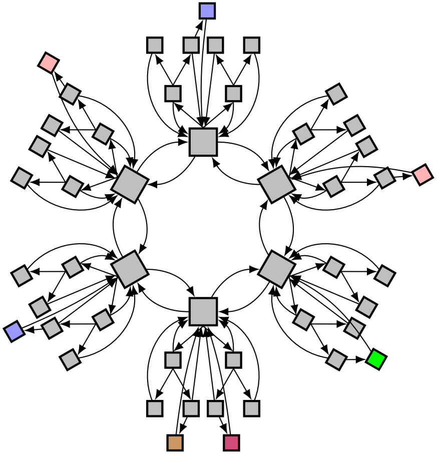
    $\rightsquigarrow$ use small boxes



Learning of Boxes

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

    $\leadsto$ use small boxes



Learning of Boxes

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

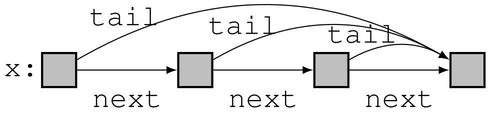
    $\leadsto$ use small boxes

Learning of Boxes

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

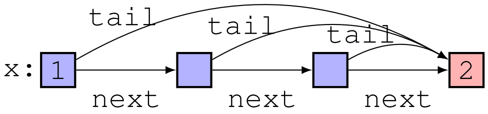
Learning of Boxes

- 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



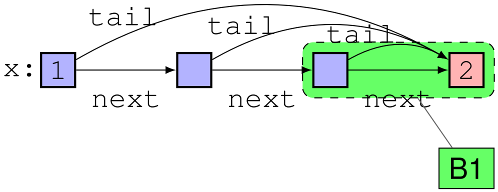
Learning of Boxes

- 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



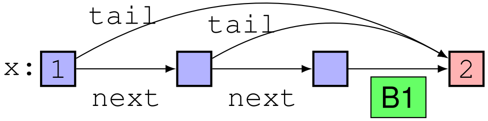
Learning of Boxes

- 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



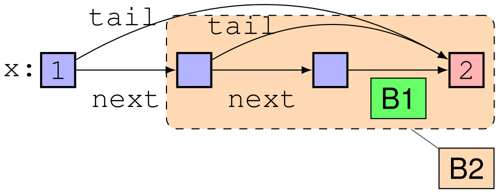
Learning of Boxes

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



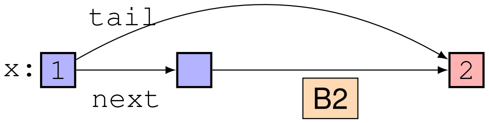
Learning of Boxes

- 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



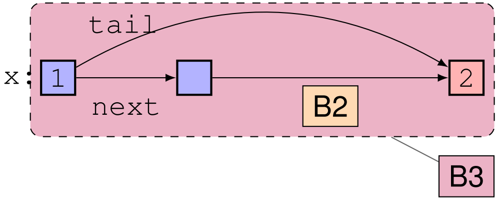
Learning of Boxes

- 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



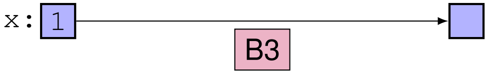
Learning of Boxes

- 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



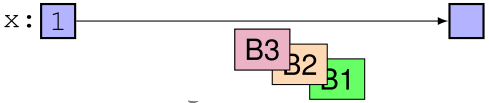
Learning of Boxes

- 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

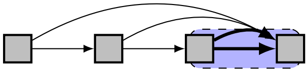
- compromise between
  - reusability: use on different heaps of the same kind
    $\leadsto$ use small boxes
  - ability to hide cut-points
    $\leadsto$ do not use too small boxes

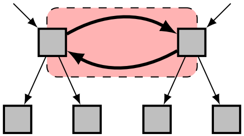


list of any length $\leadsto$ ∞ hierarchy

Learning of Boxes: Knots

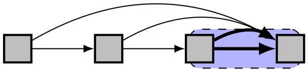
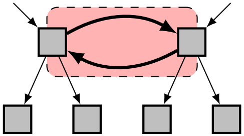
1. Smallest subgraphs meaningful to be folded:





Learning of Boxes: Knots

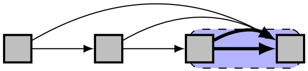
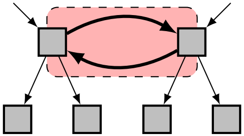
1. Smallest subgraphs meaningful to be folded:

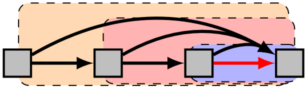
2. Handle interface

Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:

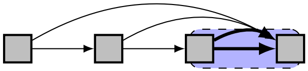
2. Handle interface
    - compose intersecting knots

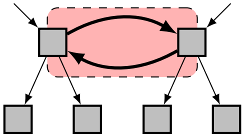


prevent ∞ nesting

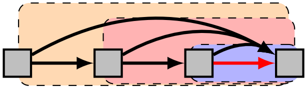
Learning of Boxes: Knots

1. Smallest subgraphs meaningful to be folded:

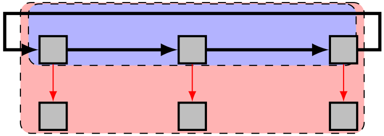




2. Handle interface
   - compose intersecting knots



   - enclose paths from inner nodes to leaves



prevent ∞ nesting

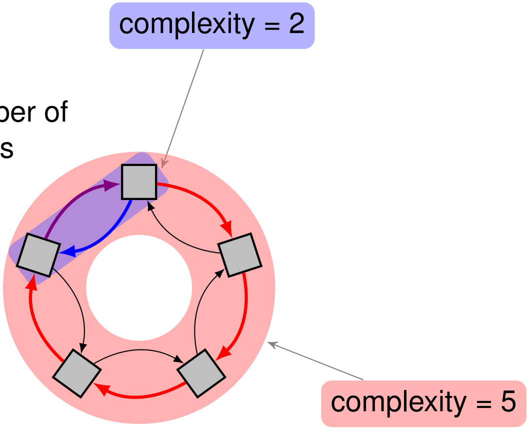
prevent ∞ interface nodes

Learning of Boxes: Knots

3. Complexity: max number of cutpoints in basic knots

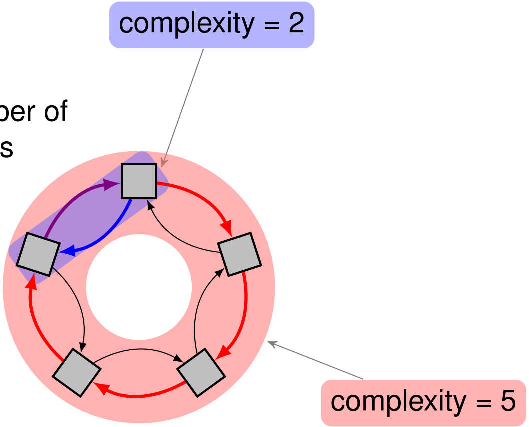
Learning of Boxes: Knots

3. Complexity: max number of cutpoints in basic knots



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.

$\Rightarrow$ hide unboundedly many cut-points

Widening Revisited

- learning and folding of boxes in the abstraction loop

The Goal

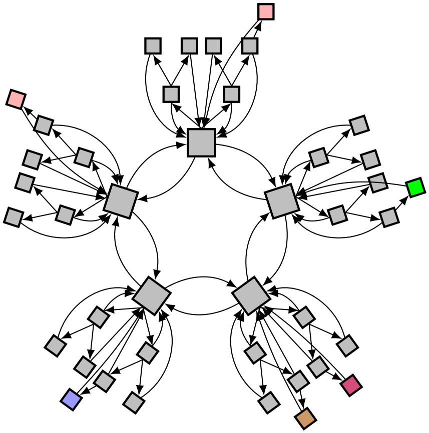
Fold boxes that will, after abstraction, appear on cycles of automata.

⇒ hide unboundedly many cut-points

```
1 Algorithm: Abstraction Loop
2 Unfold solo boxes
3 repeat
4     Abstract
5     Fold
6 until fixpoint
```

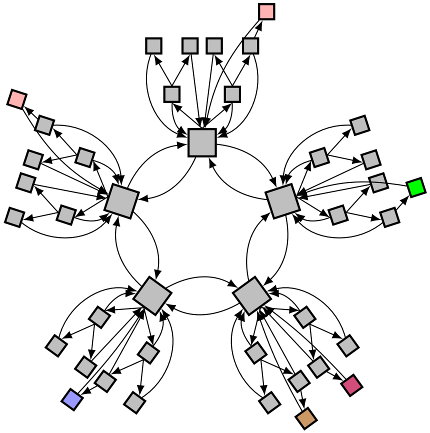
not on a cycle

Learning of Boxes: Example



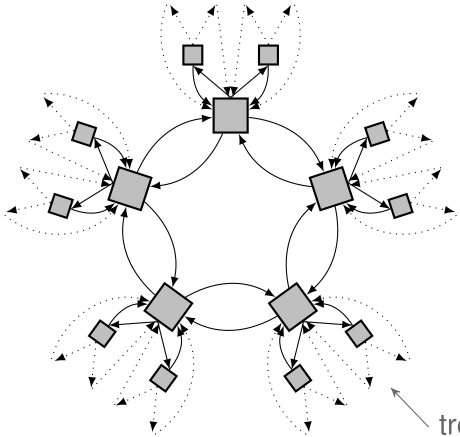
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



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

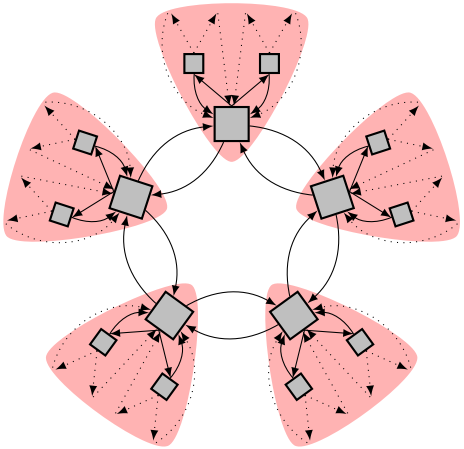
Learning of Boxes: Example



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

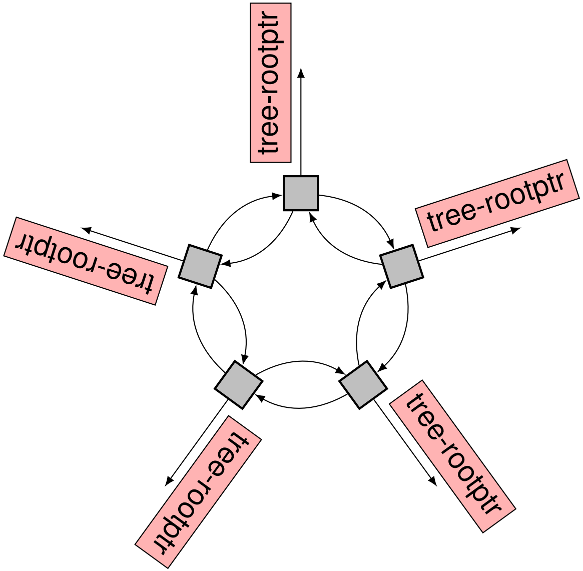
tree with root ptrs of any height

Learning of Boxes: Example



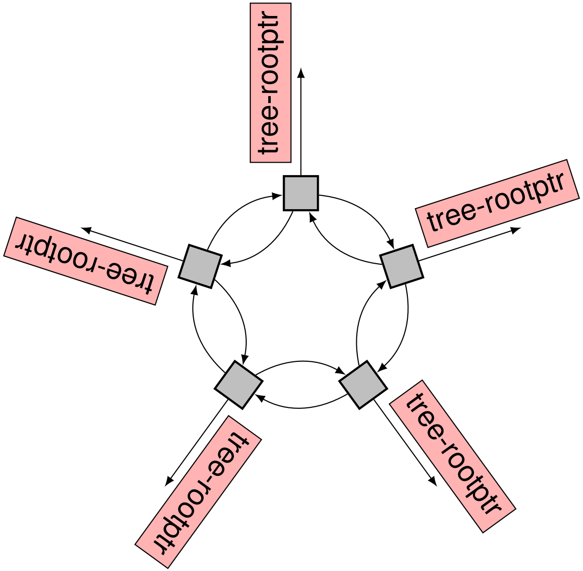
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



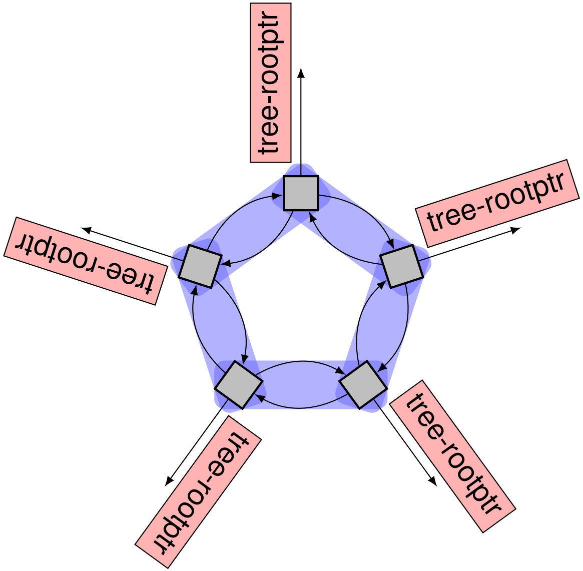
```
1  Unfold solo boxes
2  repeat
3      Abstract
4      Fold
5  until fixpoint
```

Learning of Boxes: Example



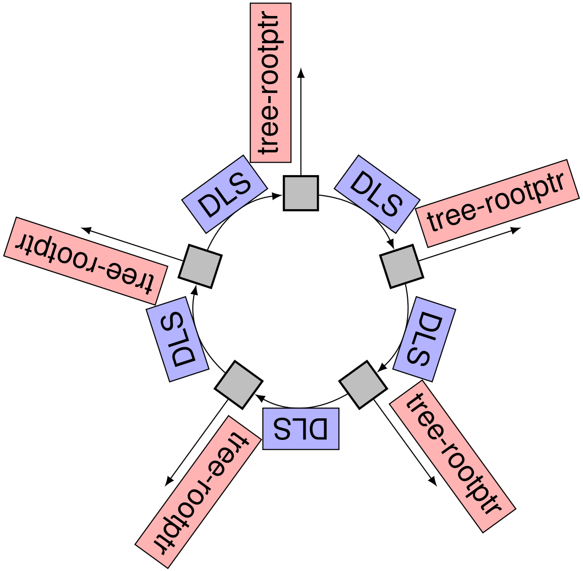
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



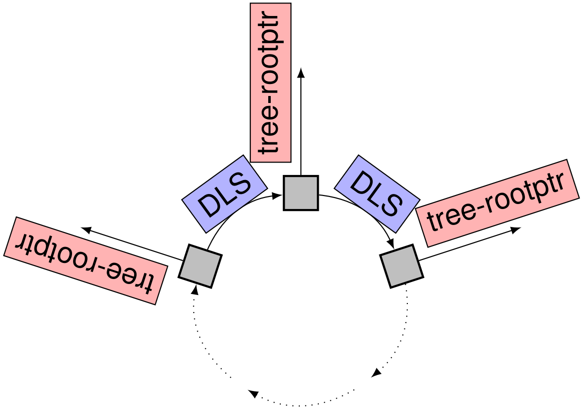
```
1 Unfold solo boxes
2 repeat
3       Abstract
4       Fold
5 until fixpoint
```

Learning of Boxes: Example



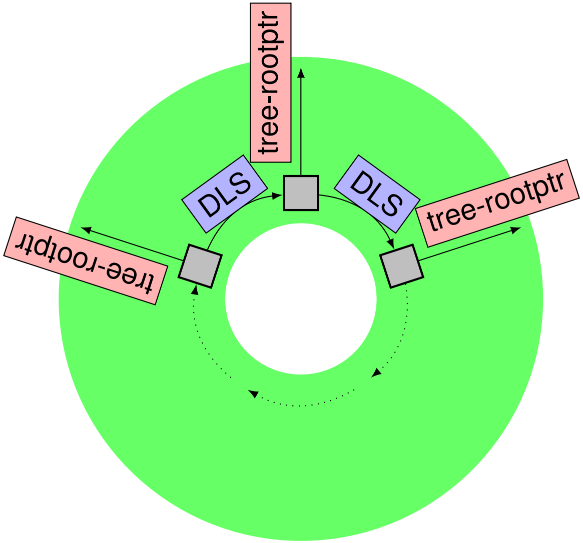
```
1 Unfold solo boxes
2 repeat
3       Abstract
4       Fold
5 until fixpoint
```

Learning of Boxes: Example



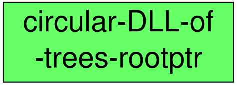
```
1 Unfold solo boxes
2 repeat
3     Abstract
4     Fold
5 until fixpoint
```

Learning of Boxes: Example



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

Learning of Boxes: Example



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

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)
  - many medals from HeapManip. and MemorySafety of SV-COMP

Experimental Results

- implemented in the **Forester** tool
- comparison with Predator (a state-of-the-art tool for lists)
  - many medals from HeapManip. and MemorySafety of SV-COMP

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
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 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

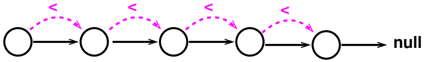
Extension to data

Extension to data

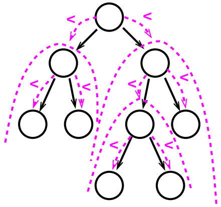
Tracking Relations over Data Values

- Verify data-related properties such as sortedness.

Sorted list

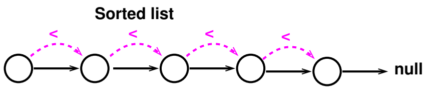


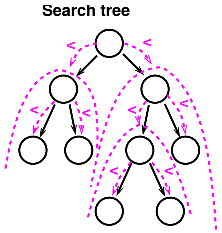
Search tree



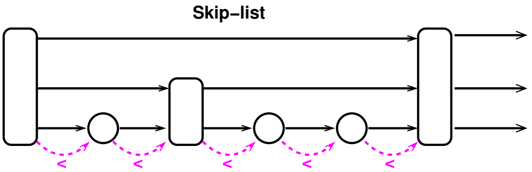
Tracking Relations over Data Values

- Verify data-related properties such as sortedness.





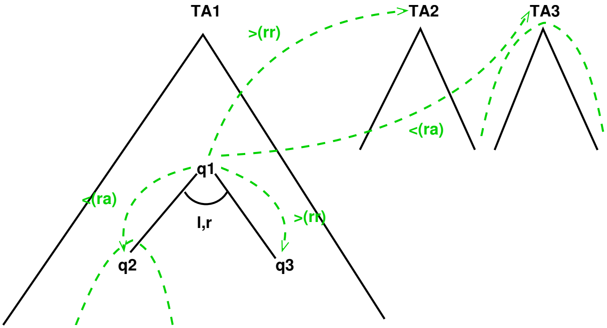
- Verify data-dependent memory safety/shape invariance.



Forest Automata with Data Constraints

- TA rules extended with constraints
  - local: between states of a single rule,
  - global: between a state and a whole TA
- comparing:
  - two nodes: root-root (rr),
  - a node and all nodes of a tree: root-all (ra).

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



Experimental Results

Support for ordering relations implemented in an extension of Forester.

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

Example	time [s]
BST insert	6.87
BST delete	114.00
BST left rotate	7.35
BST right rotate	6.25
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, 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
    - needed for lists used e.g. in the Linux kernel

Future Work

- CEGAR loop
  - **red-black** trees, . . .
  - already some preliminary results for lists
- concurrent data structures
  - lockless skip lists, . . .
- recursive boxes
  - B+ trees, . . .
- support for incomplete code