Fully Automated Shape Analysis Based on Forest Automata[†]

Lukáš Holík **Ondřej Lengál** Adam Rogalewicz
Jiří Šimáček Tomáš Vojnar

Brno University of Technology, Czech Republic

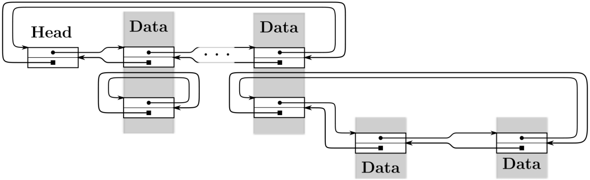
@LIAFA, Université Paris Diderot - Paris 7

July 1, 2013

[†]To appear in *Proc. of CAV'13.*

Shape Analysis

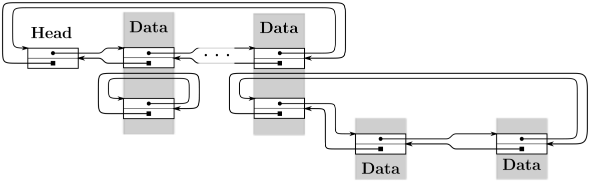
- Precise shape analysis:
  - a notoriously difficult problem



  - specialized solutions (lists)
  - help from the outside (loop invariants, inductive predicates)

Shape Analysis

- Precise shape analysis:
  - a notoriously difficult problem



  - specialized solutions (lists)
  - help from the outside (loop invariants, inductive predicates)

- Classes of errors:
  - error line reachability
  - invalid pointer dereference
  - occurrence of garbage

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, not very scalable

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
  - ☺ local reasoning of SL

The Forest Automata-based Approach

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

  with

  - ☺ local reasoning of SL

  by

  - splitting the heap into tree components

The Forest Automata-based Approach

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

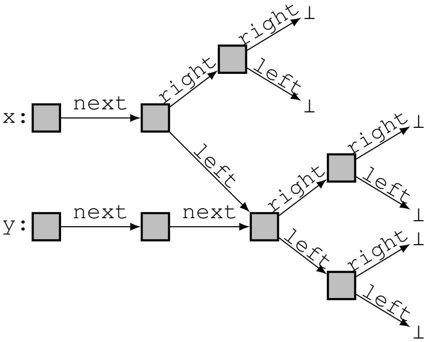
  with
  - ☺ local reasoning of SL

  by
  - splitting the heap into tree components

  and
  - TA-based representation of sets of heaps

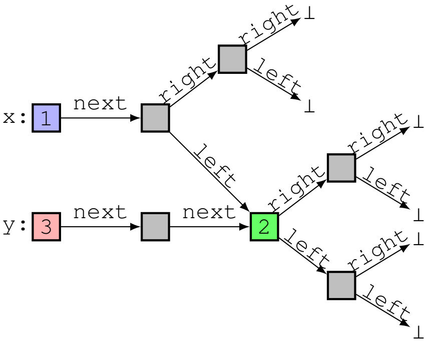
Heap Representation

- Forest decomposition of a heap



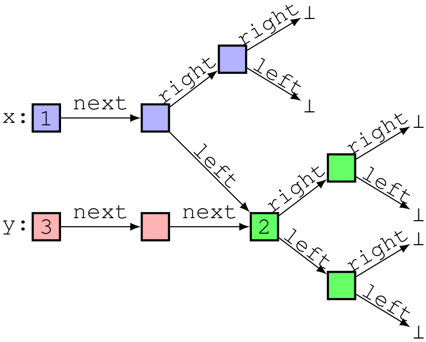
Heap Representation

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



Heap Representation

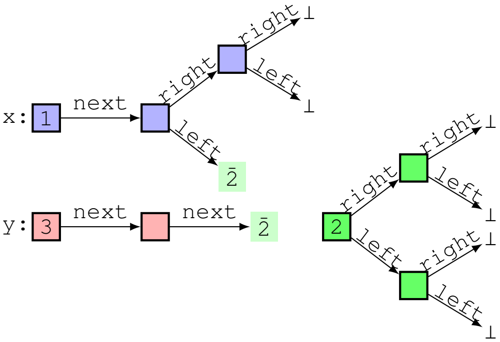
- Forest decomposition of a heap
  - Identify cut-points ← nodes referenced:
    - by variables, or
    - multiple times
  - Split the heap into tree components



Heap Representation

- Forest decomposition of a heap
  - Identify cut-points
  - 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 $(\text{🌲}_1, \text{🌲}_2, \ldots, \text{🌲}_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\text{🌲}_1, \text{🌲}_2, \ldots, \text{🌲}_n), (\text{🌲}'_1, \text{🌲}'_2, \ldots, \text{🌲}'_m), \ldots\}$

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

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 $(T_1, T_2, \ldots, T_n)$
- 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
    3. in the same order

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

Heap Representation

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

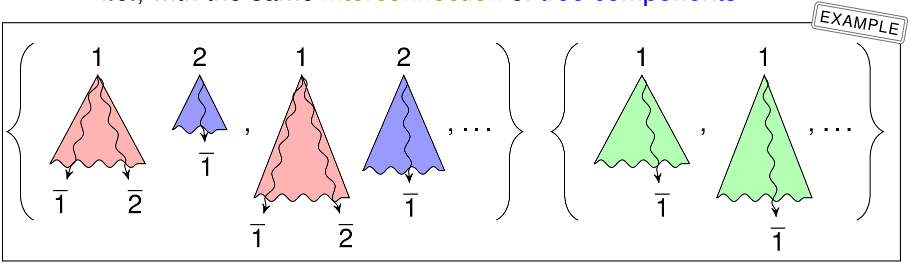
    $(🌲_1, 🌲_2, \ldots, 🌲_n), (🌲'_1, 🌲'_2, \ldots, 🌲'_n)$
  - i.e., with the same interconnection of tree components

Heap Representation

- a heap $h \mapsto$ a forest $(\bigwedge_1, \bigwedge_2, \ldots, \bigwedge_n)$
- a set of heaps $\mathcal{H} \mapsto \{(\bigwedge_1, \bigwedge_2, \ldots, \bigwedge_n), (\bigwedge'_1, \bigwedge'_2, \ldots, \bigwedge'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

$(\bigwedge_1, \bigwedge_2, \ldots, \bigwedge_n), (\bigwedge'_1, \bigwedge'_2, \ldots, \bigwedge'_n)$

  - i.e., with the same interconnection of tree components



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
    3. in the same order

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

  - i.e., with the same interconnection of tree components

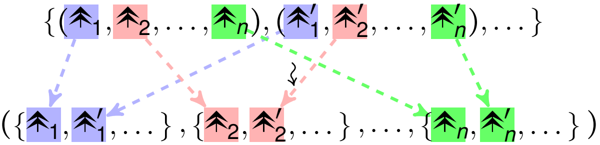
- Cartesian representation of classes of $\mathcal{H}$:

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

Heap Representation

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

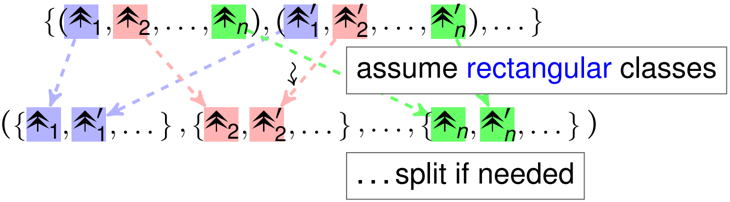
    $(⇞_1, ⇞_2, \ldots, ⇞_n), (⇞'_1, ⇞'_2, \ldots, ⇞'_n)$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:



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

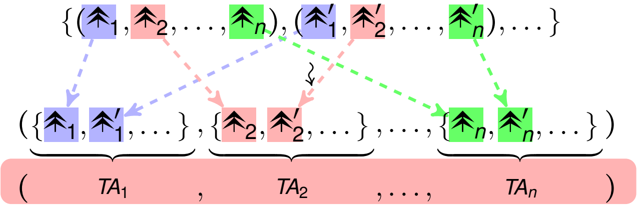
    $(\Uparrow_1, \Uparrow_2, \ldots, \Uparrow_n), (\Uparrow'_1, \Uparrow'_2, \ldots, \Uparrow'_n)$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:



Heap Representation

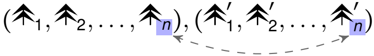
- a heap $h \mapsto$ a forest $(T_1, T_2, \ldots, T_n)$
- a set of heaps $\mathcal{H} \mapsto \{(T_1, T_2, \ldots, T_n), (T'_1, T'_2, \ldots, T'_m), \ldots\}$
  - split $\mathcal{H}$ into classes of forests with:
    1. the same number of trees
    2. having the same references
    3. in the same order

    $(T_1, T_2, \ldots, T_n), (T'_1, T'_2, \ldots, T'_n)$
  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:

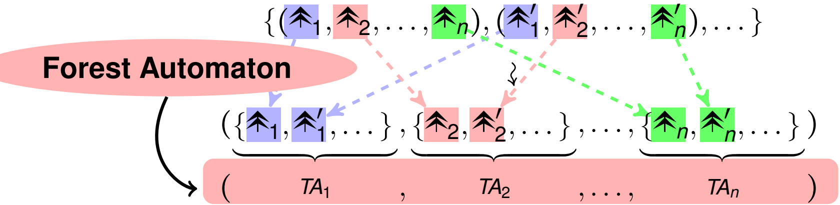


Heap Representation

- 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

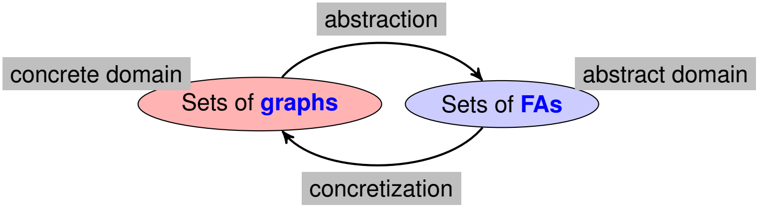


  - i.e., with the same interconnection of tree components
- Cartesian representation of classes of $\mathcal{H}$:



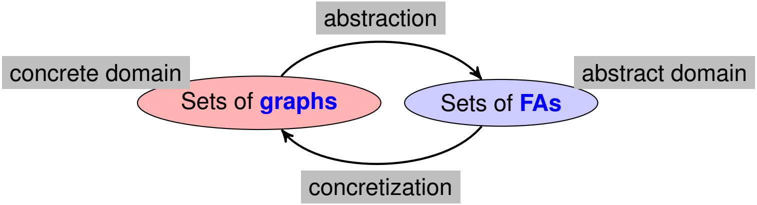
Symbolic Execution

- Abstract Interpretation



Symbolic Execution

- Abstract Interpretation

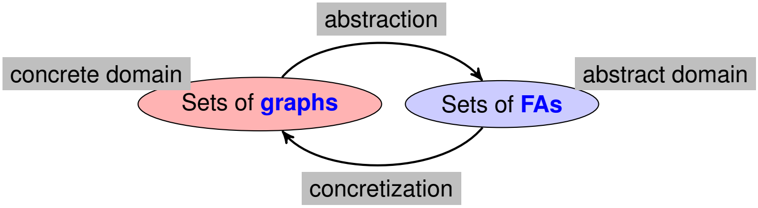


Statements

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

Symbolic Execution

- Abstract Interpretation



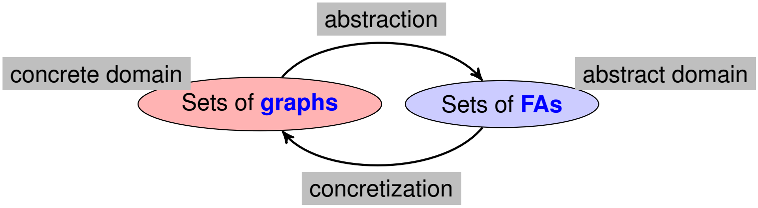
Statements

Abstract Transformers

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

Symbolic Execution

- Abstract Interpretation

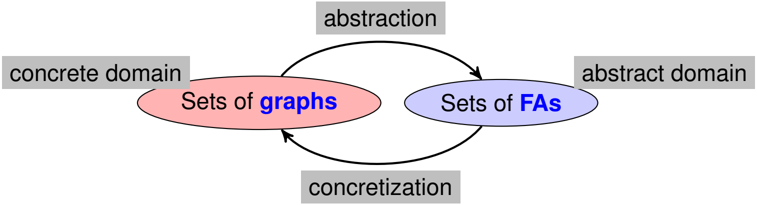


Statements | **Abstract Transformers**

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

Symbolic Execution

- Abstract Interpretation

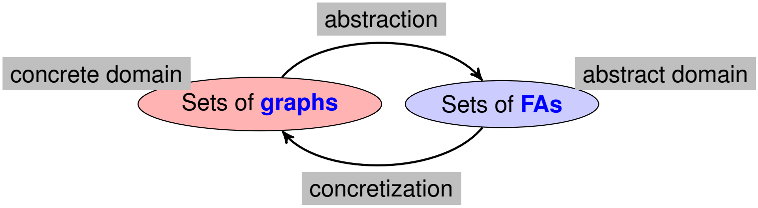


Statements | **Abstract Transformers**

- `x := new T()` ← append a TA
- `delete(x)` ← remove a TA
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Symbolic Execution

- Abstract Interpretation



Statements

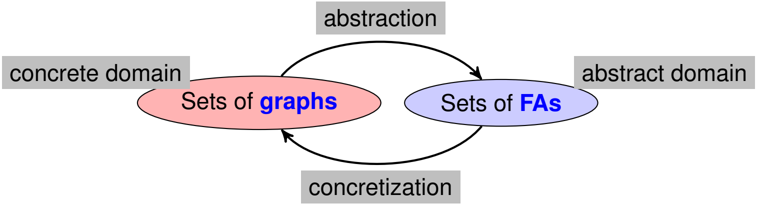
- `x := new T()`
- `delete(x)`
- `x := null`
- `x := y`
- `x := y.next`
- `x.next := y`
- `if/while (x == y)`

Abstract Transformers

- append a TA → `x := new T()`
- remove a TA → `delete(x)`
- modify transitions → `x := null`, `x := y`, `x := y.next`, `x.next := y`

Symbolic Execution

- Abstract Interpretation



Statements / **Abstract Transformers**

- `x := new T()` ← append a TA
- `delete(x)` ← remove a TA
- `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

Acceleration

- abstraction on forest automaton $(TA_1, \ldots, TA_n)$

Acceleration

- abstraction on forest automaton $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$

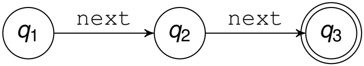
Acceleration

- abstraction on forest automaton $(TA_1, \ldots, TA_n)$
  - 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

Acceleration

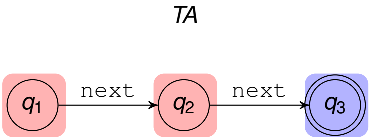
- abstraction on forest automaton $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k

TA



Acceleration

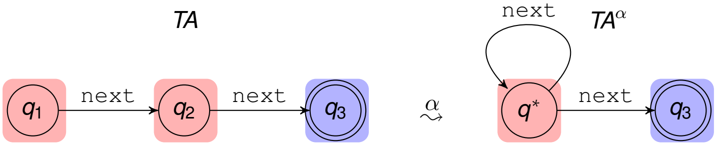
- **abstraction** on forest automaton $(TA_1, \ldots, TA_n)$
  - collapse states of component TAs $\rightsquigarrow$ $(TA_1^\alpha, \ldots, TA_n^\alpha)$
  - finite-height abstraction (from ARTMC)
    - collapse states with languages whose prefixes match up to height k



$k = 1$

Acceleration

- abstraction on forest automaton $(TA_1, \ldots, TA_n)$
  - 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



$k = 1$

Summary

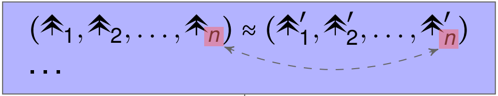
The so-far-presented:

Summary

The so-far-presented:

- ☺ works well for singly linked lists (SLLs), trees

Summary



The so-far-presented:

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

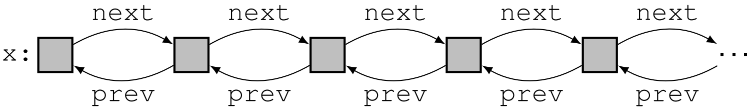
Summary

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

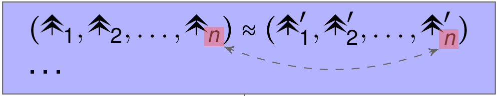
$\ldots$

The so-far-presented:

- ☺ works well for singly linked lists (SLLs), trees
- ☹ fails for more complex data structures
  - unbounded number of cut-points $\rightsquigarrow \infty$ classes of $\mathcal{H}$

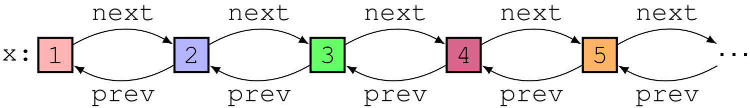


Summary



The so-far-presented:

- ☺ works well for singly linked lists (SLLs), trees
- ☹ fails for more complex data structures
  - unbounded number of cut-points $\rightsquigarrow \infty$ classes of $\mathcal{H}$



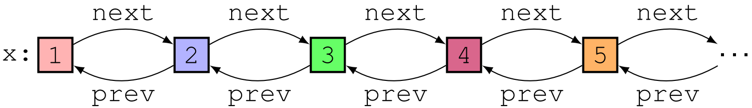
Summary

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

$\ldots$

The so-far-presented:

- ☺ works well for singly linked lists (SLLs), trees
- ☹ fails for more complex data structures
  - unbounded number of cut-points $\leadsto \infty$ 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 with a single symbol

Hierarchical Forest Automata

- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - Intuition: replace repeated subgraphs with a single symbol

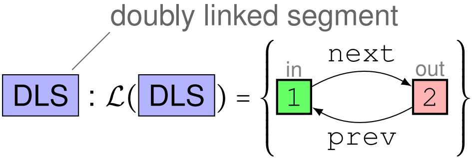
doubly linked segment

- Example: a box 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 with a single symbol

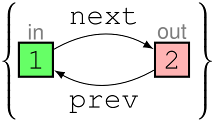
- Example: a box DLS : $\mathcal{L}($ DLS $) =$

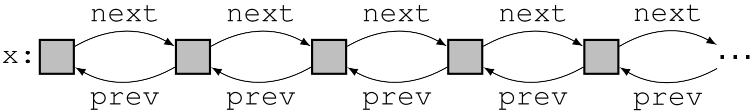


Hierarchical Forest Automata

- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - Intuition: replace repeated subgraphs with a single symbol

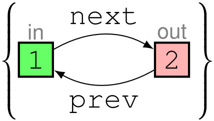
doubly linked segment

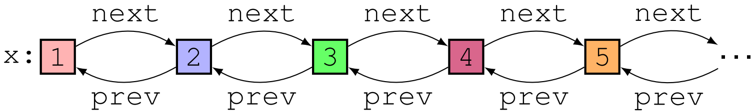
- Example: a box DLS : $\mathcal{L}($DLS$)$ = 



Hierarchical Forest Automata

- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - Intuition: replace repeated subgraphs with a single symbol

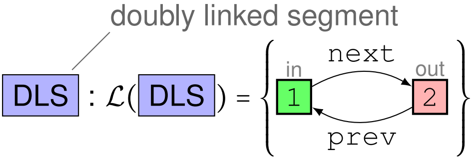
- Example: a box DLS : $\mathcal{L}($DLS$)$ = 

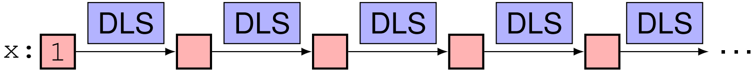


Hierarchical Forest Automata

- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - Intuition: replace repeated subgraphs with a single symbol

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

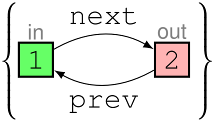


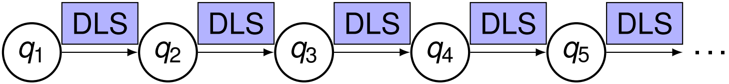


Hierarchical Forest Automata

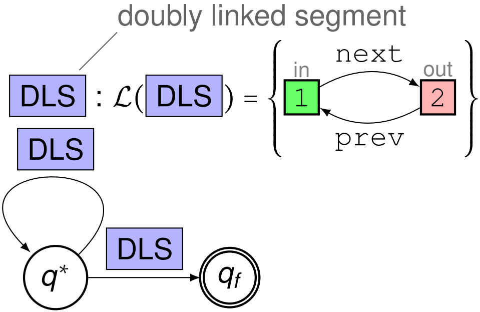
- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - Intuition: replace repeated subgraphs with a single symbol

doubly linked segment

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



- Hierarchical Forest Automata
  - FAs are symbols (**boxes**) of FAs of a higher level
  - a hierarchy of FAs
  - Intuition: replace repeated subgraphs with a single symbol
- Example: a box



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

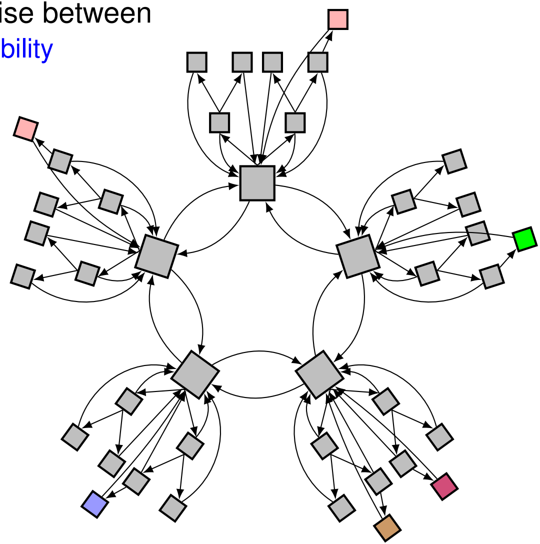
- compromise between

Learning of Boxes

- compromise between
  - reusability

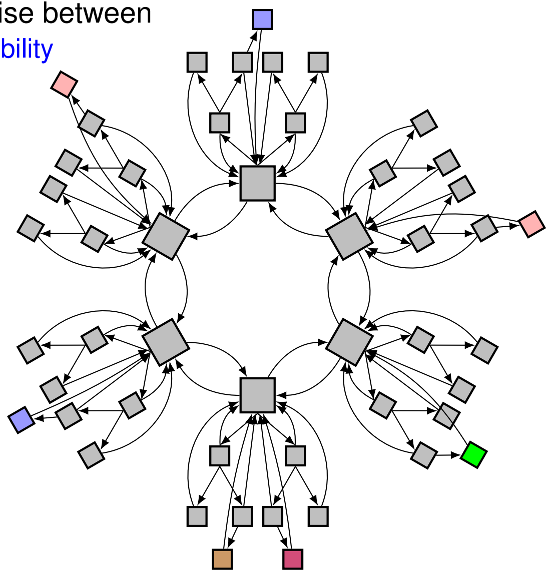
Learning of Boxes

- compromise between
  - reusability



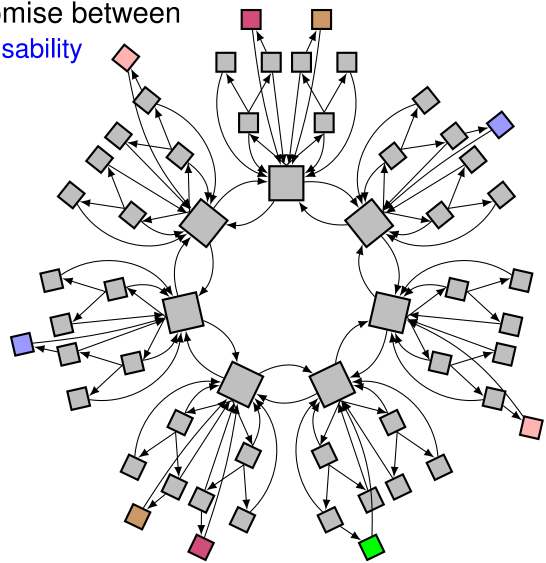
Learning of Boxes

- compromise between
  - reusability



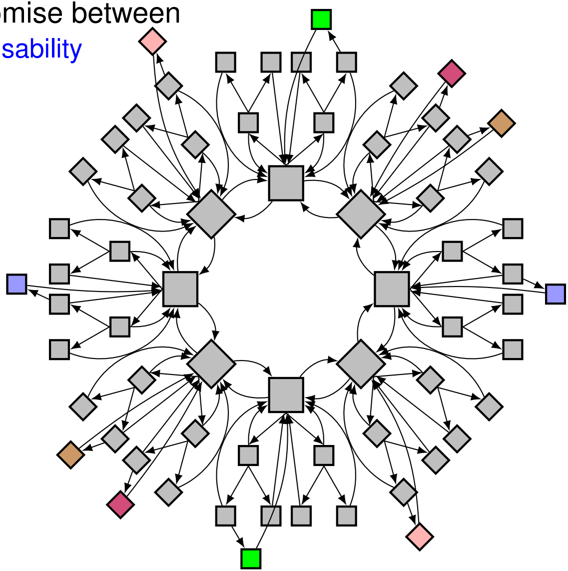
Learning of Boxes

- compromise between
  - reusability



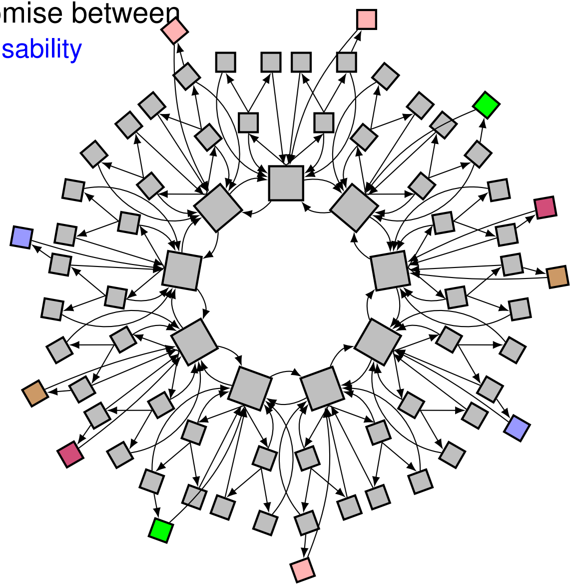
Learning of Boxes

- compromise between
  - reusability



Learning of Boxes

- compromise between
  - reusability



Learning of Boxes

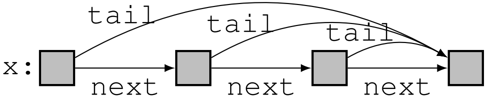
- compromise between
  - reusability

Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points

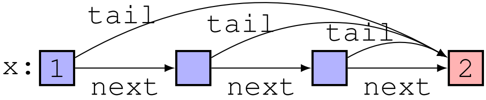
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



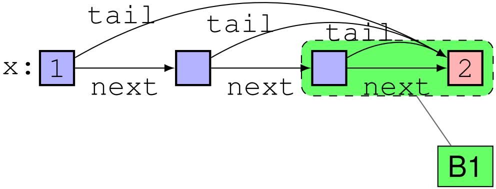
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



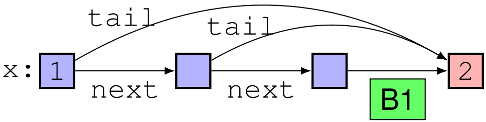
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



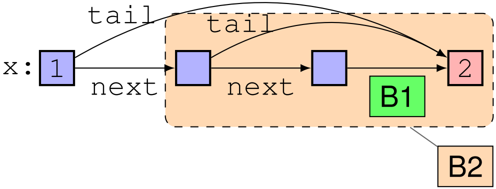
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



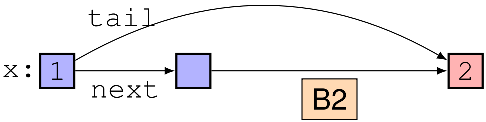
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



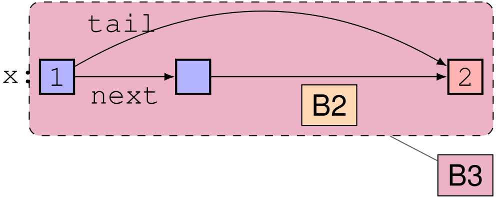
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



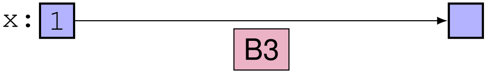
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



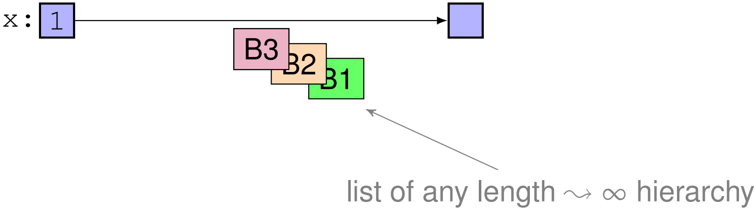
Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



Learning of Boxes

- compromise between
  - reusability
  - ability to hide cut-points



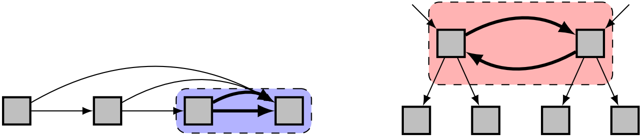
Learning of Boxes: Knots

Knots

Learning of Boxes: Knots

Knots

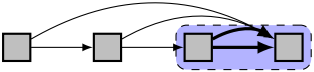
1. smallest subgraphs meaningful to be folded:

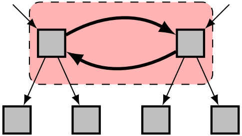


Learning of Boxes: Knots

Knots

1. smallest subgraphs meaningful to be folded:



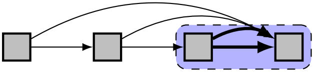


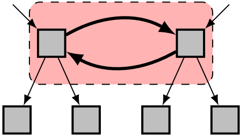
2. handle interface

Learning of Boxes: Knots

Knots

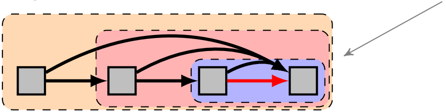
1. smallest subgraphs meaningful to be folded:





2. handle interface
   - compose intersecting knots

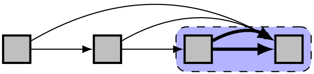
prevent ∞ nesting

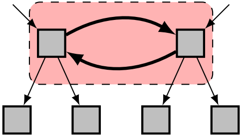


Learning of Boxes: Knots

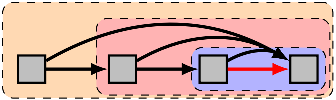
Knots

1. smallest subgraphs meaningful to be folded:



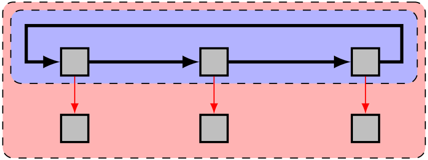


2. handle interface
   - compose intersecting knots



prevent ∞ nesting

   - enclose paths from inner nodes to leaves



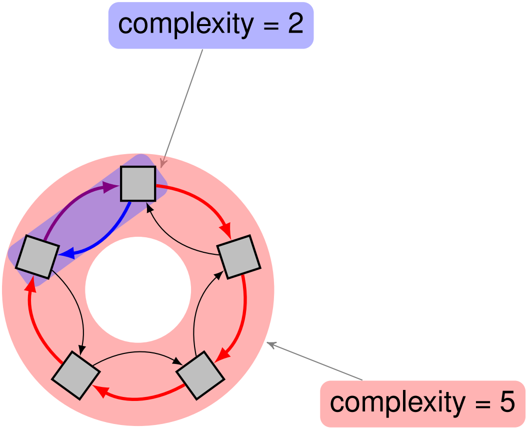
prevent ∞ interface nodes

Learning of Boxes: Knots

3 complexity

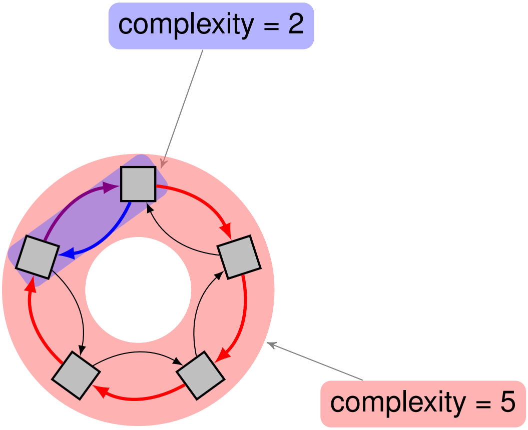
Learning of Boxes: Knots

3 complexity



Learning of Boxes: Knots

3 complexity



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

Acceleration Revisited

- learning and folding of boxes in the abstraction loop

Acceleration 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

Acceleration Revisited

- learning and folding of boxes in the abstraction loop

The Goal

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

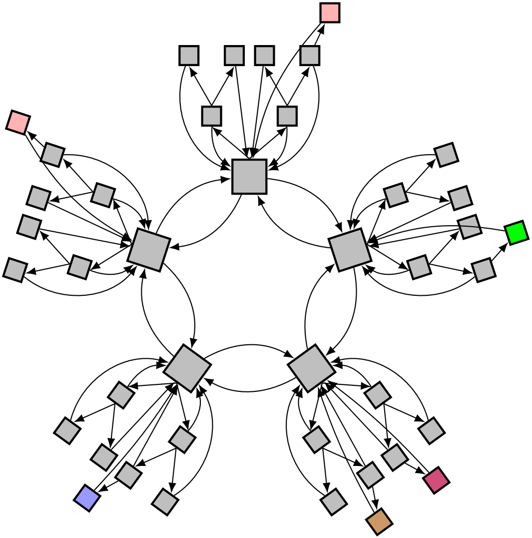
$\Rightarrow$ hide unboundedly many cut-points

Algorithm: Abstraction Loop

Unfold solo boxes
repeat
    Abstract
    Fold
until *fixpoint*

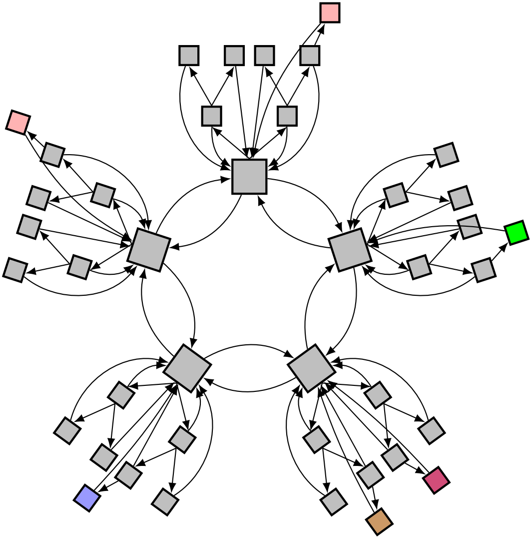
not on a cycle

Learning of Boxes: Example



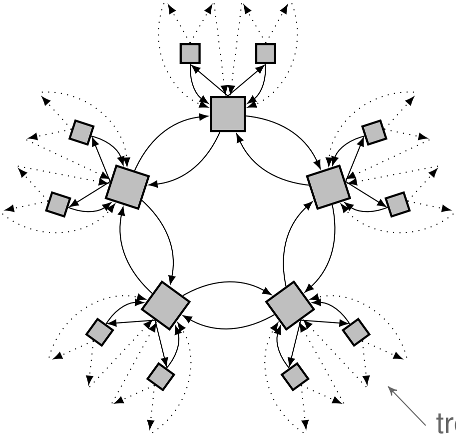
1 *Unfold solo boxes*
2 **repeat**
3 *Abstract*
4 *Fold*
5 **until** *fixpoint*

Learning of Boxes: Example



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

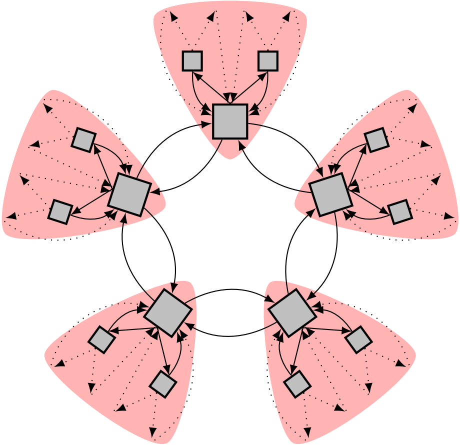
Learning of Boxes: Example



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

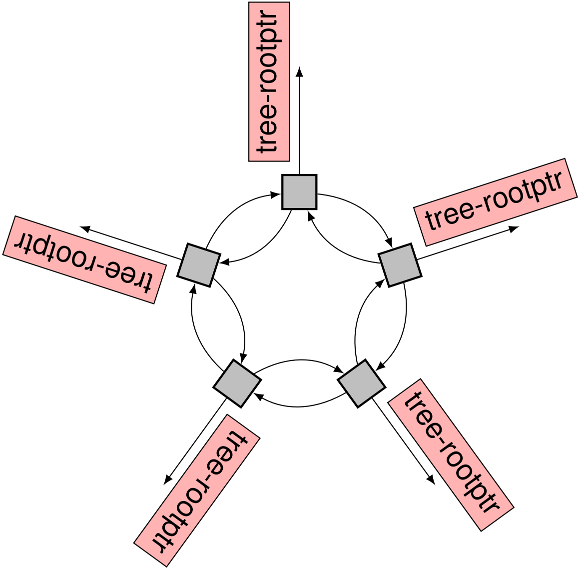
tree with root ptrs of any height

Learning of Boxes: Example



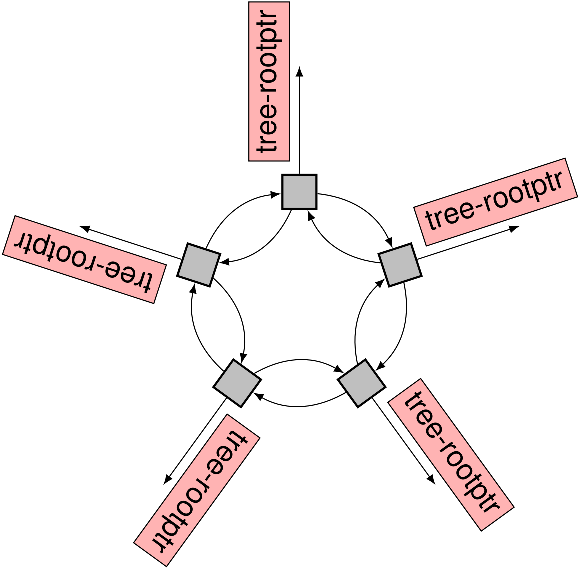
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example



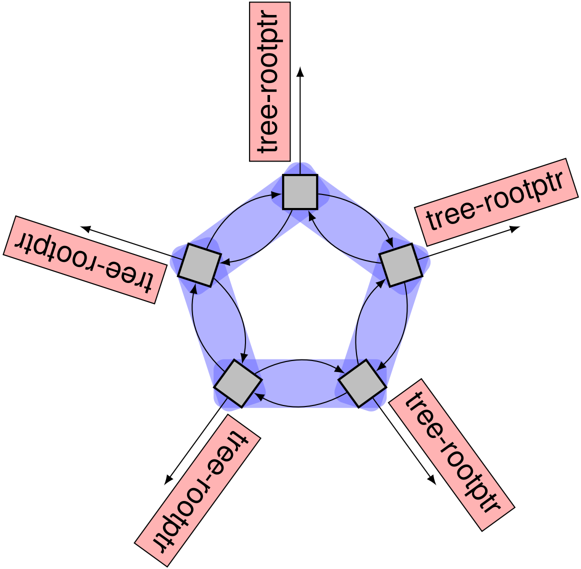
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

Learning of Boxes: Example



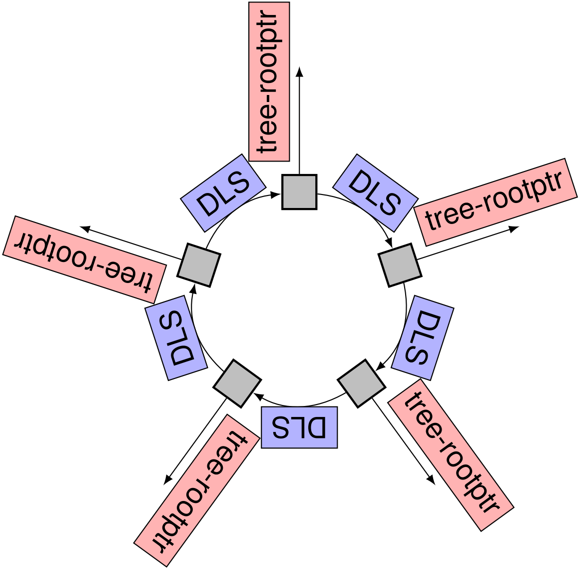
1 *Unfold solo boxes*
2 **repeat**
3   *Abstract*
4   *Fold*
5 **until** *fixpoint*

Learning of Boxes: Example



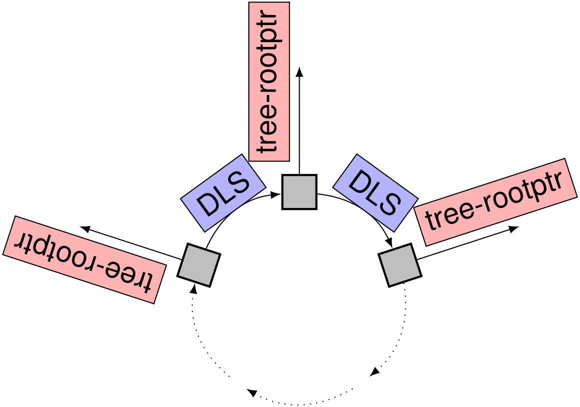
1 *Unfold solo boxes*
2 **repeat**
3 *Abstract*
4 *Fold*
5 **until** *fixpoint*

Learning of Boxes: Example



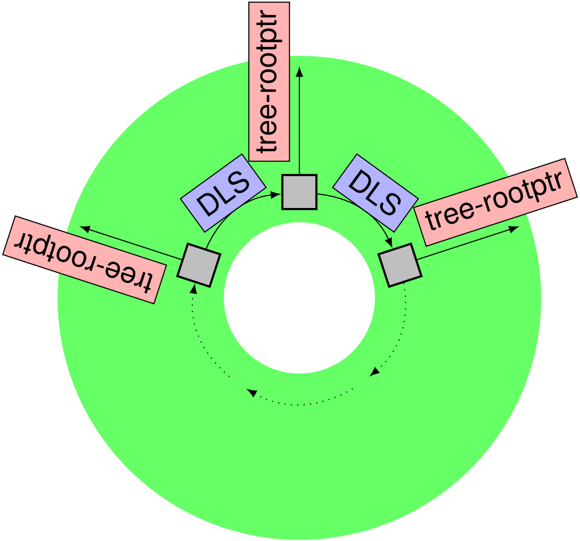
1. *Unfold solo boxes*
2. **repeat**
3. *Abstract*
4. *Fold*
5. **until** *fixpoint*

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
```

Experimental Results

- implemented in **Forester** tool

Experimental Results

- implemented in **Forester** tool
- comparison with Predator (state-of-the-art tool for lists)
    - winner of HeapManipulation and MemorySafety of SV-COMP'13

Experimental Results

- implemented in **Forester** tool
- comparison with Predator (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 (T) — false positive (Err)

Conclusion

Shape analysis with forest automata:

Conclusion

Shape analysis with forest automata:

- fully automated

Conclusion

Shape analysis with forest automata:

- fully automated
- very flexible framework

Conclusion

Shape analysis with forest automata:

- fully automated
- very flexible framework
- **Forester** tool

Conclusion

Shape analysis with forest automata:

- fully automated
- very flexible framework
- **Forester** tool
- successfully verified:
  - (singly/doubly linked (circular)) lists (of (. . .) lists)
  - trees
  - skip lists

Conclusion

Shape analysis with forest automata:

- fully automated
- very flexible framework
- **Forester** tool
- successfully verified:
  - (singly/doubly linked (circular)) lists (of (. . .) lists)
  - trees
  - skip lists

- not covered here:
  - support for pointer arithmetic
  - tracking ordering relations
    - P. Abdulla, L. Holík, B. Jonsson, O. Lengál, C.Q. Tring, and T. Vojnar. **Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata.** To appear in *Proc. of ATVA'13*.

Future work

- CEGAR loop
  - **red**-**black** trees, ...

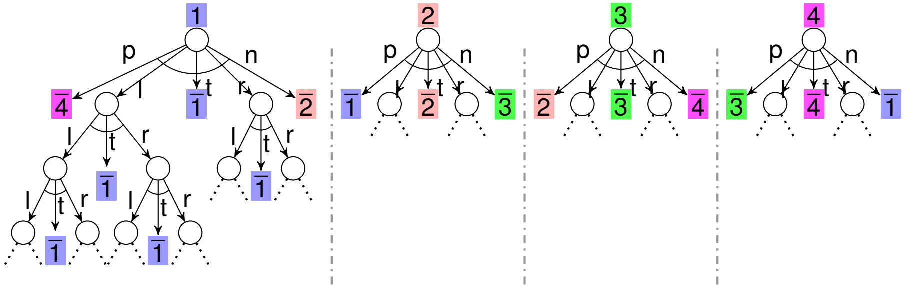
Future work

- CEGAR loop
  - **red**-**black** trees, . . .
- concurrent data structures
  - lockless skip lists, . . .

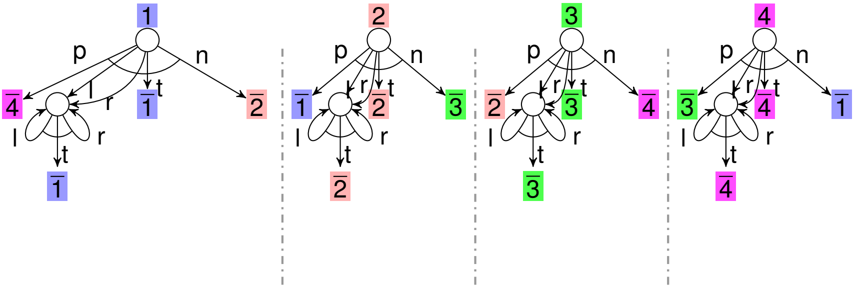
Future work

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

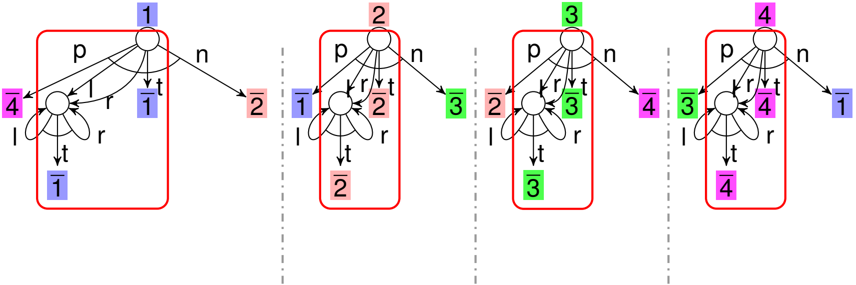
Appendix

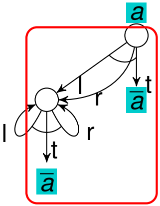


Appendix

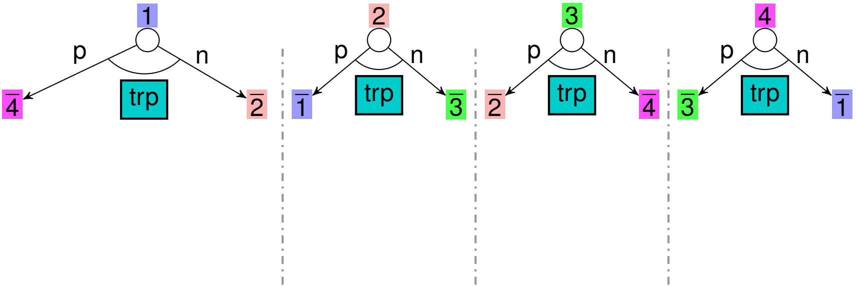


Appendix

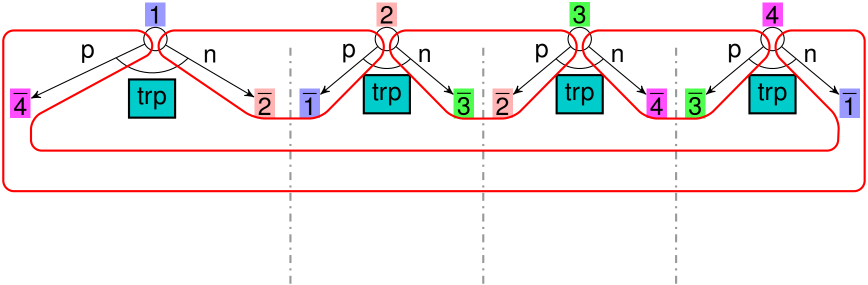

1
p
n
l
r
t
l
r
t
4
1
2
1
2
p
n
r
r
t
l
r
t
1
2
3
2
3
p
n
r
r
t
l
r
t
2
3
4
3
4
p
n
r
r
t
l
r
t
3
4
1
4

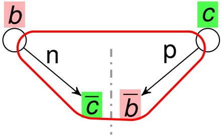

a
l
r
t
ā
l
r
t
ā

Appendix

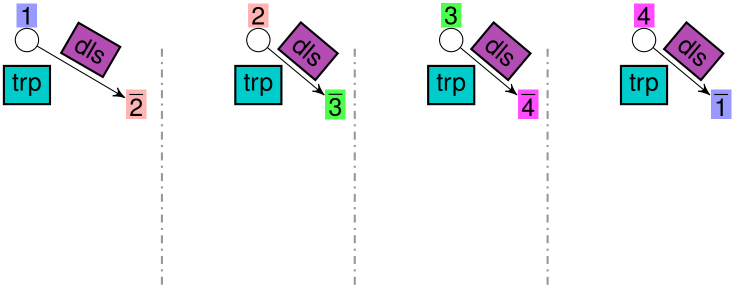


Appendix



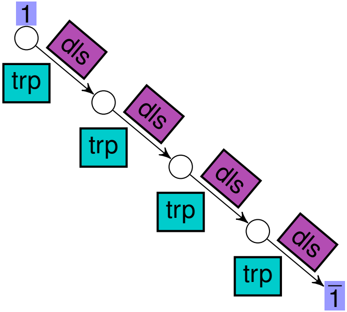


Appendix



1
dls
trp
$\overline{2}$
2
dls
trp
$\overline{3}$
3
dls
trp
$\overline{4}$
4
dls
trp
$\overline{1}$

Appendix



Appendix